Driving Big Data With Big Compute

Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Jeremy Kepner, Andrew McCabe, Peter Michaleas, Julie Mullen, David O'Gwynn, Andrew Prout, Albert Reuther, Antonio Rosa, Charles Yee
MIT Lincoln Laboratory, Lexington, MA, U.S.A.

Abstract— **Big Data (as embodied by Hadoop clusters) and Big Compute (as embodied by MPI clusters) provide unique capabilities for storing and processing large volumes of data. Hadoop clusters make distributed computing readily accessible to the Java community and MPI clusters provide high parallel efficiency for compute intensive workloads. Bringing the big data and big compute communities together is an active area of research. The LLGrid team has developed and deployed a number of technologies that aim to provide the best of both worlds. LLGrid MapReduce allows the map/reduce parallel programming model to be used quickly and efficiently in any language on any compute cluster. D4M (Dynamic Distributed Dimensional Data Model) provided a high level distributed arrays interface to the Apache Accumulo database. The accessibility of these technologies is assessed by measuring the effort to use these tools and is typically a few lines of code. The performance is assessed by measuring the insert rate into the Accumulo database. Using these tools a database insert rate of 4M inserts/second has been achieved on an 8 node cluster.**

Keywords-component; LLGridMapReduce; parallel ingestion; concurrent query; scheduler; hdfs; parallel matlab, d4m

I. INTRODUCTION

In recent years, the proliferation of sensor devices and the growth of the Internet, has created a deluge of data. When dealing with big data there are many hurdles: data capture, storage, search, sharing, analytics and visualization. For database analysis efficient and high performance data ingest and query are very important. Hadoop clusters [1] are a data oriented distributed computing environment. As such, it is a good foundation for building distributed databases in Java and there are number of databases that have been built using Hadoop (e.g., HBase [2] and Accumulo [3]). Likewise, MPI clusters [4] are a language agnostic parallel computing environment that are a good foundation for building efficient data analysis applications. Bringing these two worlds together is an active area of research [5].

Uniting Hadoop clusters and MPI clusters requires addressing several technical differences. First, Hadoop clusters are Java centric, while MPI clusters are multi-lingual. Second, Hadoop clusters provide the map/reduce parallel programming model, while the MPI clusters supports all parallel programming models (map/reduce, message passing, distributed arrays). Third, Hadoop clusters provide a Java API to data, while MPI clusters use operating system filesystem calls. Fourth, Hadoop clusters manage their own jobs, while in MPI clusters jobs are managed by a scheduler.

Based on our experiences with MIT Lincoln Laboratory Grid (LLGrid) [6], we (the LLGrid team) have identified four specific use cases where it would make sense to bring these worlds together: (1) applications written in any language that would like to use the map/reduce programming model and/or interact to a Hadoop database, (2) applications written in MATLAB/GNU Octave that need to interact with a Hadoop database, (3) applications written in any language that need to access data stored in the Hadoop file system, and (4) Java applications written in Hadoop MapReduce that need to run on an MPI cluster.

For each use case, the LLGrid team has developed or is testing a new technology. For case (1), we have developed LLGrid MapReduce that allows any language to run the map/reduce parallel programming model on an MPI cluster. For case (2), we have developed D4M (Dynamic Distributed Dimensional Data Model) technology [7] to provide a mathematically rich interface to tuple stores and relational databases. For case (3), we are testing Fuse [8] operating system bindings to the Hadoop file system. Finally, for case (4), we are testing Apache Hadoop integration with Grid Engine [9] that allows Hadoop map/reduce jobs to have their resources from a central scheduler.

The remainder of this paper presents the details of LLGrid MapReduce and D4M and demonstrates how these tools can be used to support the use cases we identified important for LLGrid users. In addition, we discussed about the performance results obtained with each.

II. LLGRID MAPREDUCE

The map/reduce parallel programming model is the simplest of all parallel programming models, which is much easier to learn than message passing or distributed arrays. The map/reduce parallel programming model consists of two user written programs: Mapper and Reducer. The input to Mapper is a file and the output is another file. The input to Reducer is the set of Mapper output files. The output of Reducer is a single file. Launching consists of starting many Mapper programs each with a different file. When the Mapper programs have completed the Reduce program is run on the Maper outputs.

LLGrid MapReduce enables map/reduce for any language using a simple one line command. Although Hadoop provides a Java API for executing map/reduce programs and, through Hadoop Streaming, allows to run map/reduce jobs with any executables and scripts on files in the Hadoop file system, LLGrid MapReduce can use data from central storage filesystem or a FUSE-mounted Hadoop file system. LLGrid

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

MapReduce identifies the input files to be processed by scanning a given input directory or reading a list from a given input file as shown in the step 1 in Fig. 1. Then, by accessing the scheduler at the step 2, it creates an array of many tasks, called an array job, which is noted as "Mapper Task 1", "Mapper Task 2", and so on. Modern schedulers such as the open source Grid Engine [10] provide an array job with an option to control how many tasks can be processed concurrently. Once the array job is created and dispatched for execution, each input file will be processed by one of the tasks with the specified application at the command line, noted as "Mapper" in Fig. 1. The application can be any type of executable, such as a shell script, a Java program or any executable programs that are written in any languages.

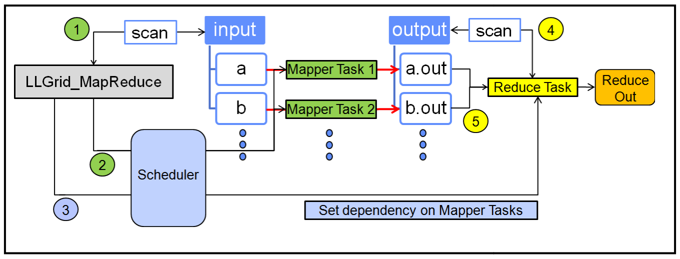

Figure 1. An example diagram showing how LLGrid MapReduce works.

Once all the input data are processed, there is an option to collect the results, if there are any, by creating a dependent task at the step 3, which is noted as "Reduce Task" in Fig. 1. The reduce task will wait until all the mapper tasks are completed by setting a job dependency between the mapper and reduce tasks. The reduce application is responsible to scan the output from the mapper tasks at the step 4 and to merge them into the final results at the step 5.

The LLGrid MapReduce command API is shown in Fig. 2. The map application has two input arguments, one for input filename and the other for the output filename. The reduce application takes one argument as input, the directory path where the results of the map tasks reside. The reduce application scans and reads the output generated by the map tasks.

```
LLGrid_MapReduce --np nTasks\
                 --mapper myMapper \
                 --reducer myReducer \
                 --input input_dir \
                 --output output_dir \
             [--redout output_filename]
```

Figure 2. LLGrid MapReduce API

One of the advantages using LLGrid MapReduce is that the number of concurrent map tasks are controlled by the user with the --np option. This feature gives the user precise control of how many resources they are using and greatly reduces conflicts between users on the same system. In addition, it make much simpler for the user to optimize their programs to determine what the optimal number of resources are to consume. Allowing users to control and optimize their resource usage makes it possible for one cluster to support many more map/reduce users. An important example of this phenomena is ingesting into a database where there is almost always a maximum ingest rate beyond which adding more ingestors will not help and can even degrade performance.

Another advantage of LLGrid MapReduce is that there is no internal map/reduce API. So the Mapper and Reducer programs can be written and debugged on the users' workstation without any additional software.

III. Dynamic Distributed Dimensional Data Model (D4M)

The MATLAB/GNU Octave language, sometimes referred as M language, is the most popular language at the Lincoln Laboratory. We have developed several parallel computation technologies (e.g., pMatlab [11,12], MatlabMPI [4], and gridMatlab [6]) that allow efficient distribution of data across a parallel computer. In addition, the LLGrid team has developed and deployed the D4M to allow these users to work naturally with databases. D4M allows linear algebra to be readily applied to databases. Using D4M, it is possible to create composable analytics with significantly less effort than using traditional approaches. Furthermore, with existing LLGrid technologies, D4M parallel MATLAB implementation can provide significant performance enhancement in database insertion and query.

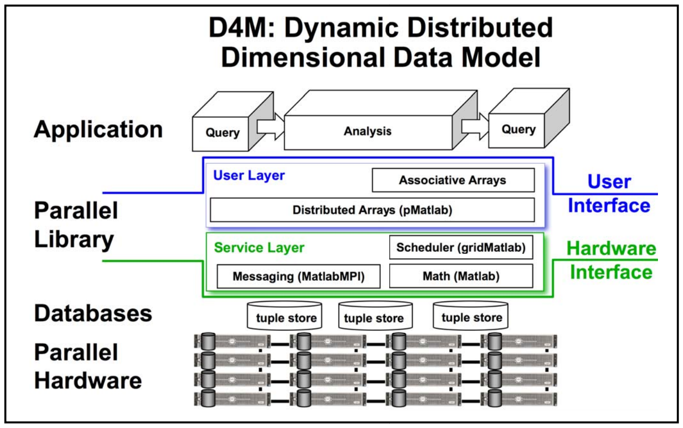

Figure 3. D4M Matlab prototype architecture. At the top is the user application consisting of a series of query and analysis steps. In the middle is the parallel library that hides the parallel mapping of the operations. On the bottom are the databases (typically tuple stores) running on parallel computing hardware.

D4M uses layered implementation that allows each layer to address a specific challenge as shown in Fig. 3. The top layer consists of composable associative arrays that provide a one-to-one correspondence between database queries and linear algebra. Associative arrays can be both the input and output of a wide range of database operations and allow complex operations to be composed with a small number of statements. Associative arrays are implemented in Java and M languages, approximately 12,000 lines of source code, which provide an easy way to create the interface to the middle and bottom layers. The middle layer consists of several parallel computation technologies (e.g., pMatlab [11,12], MatlabMPI

[4], and gridMatlab [6]) that allow associative arrays to be distributed efficiently across a parallel computer. Furthermore, the pieces of the associative array can be bound to specific parts of one more databases to optimize the performance of data insertion and query across a parallel database system. The bottom layer consists of databases (e.g., Accumulo [3] or SQLServer) running on parallel computation hardware. D4M can fully exploit the power of databases that use an internal sparse tuple representation (e.g., a row/col/val triple store) to store all data regardless of type. Constructing complex composable query operations can be expressed using simple array indexing of the associative array keys and values, which themselves return associative arrays. For example

`A('alice ',:)`	alice row
`A('alice bob ',:)`	alice and bob rows
`A('al* ',:)`	rows beginning with al
`A('alice : bob ',:)`	rows alice to bob
`A(1:2,:)`	first two rows
`A == 47.0`	all entries equal to 47.0

Finally, associative arrays can represent complex relationships in either a sparse matrix or a graph form. Associative arrays are thus a natural data structure for environments such as Matlab to perform both matrix and graph algorithms.

IV. Data Ingestion Performance

Ingest into a database is a good test of showing how easy to use the LLGrid MapReduce and D4M are for such a task. This is also an important example of the kind of application that can take advantage of combining a Hadoop cluster and a MPI cluster.

There are several approaches to ingest data into the databases. In the LLGrid environment, the LLGrid MapReduce command can deploy the data ingestion tasks to the LLGrid with minimum user efforts by using the underlying scheduler. It creates an array job for the given data set. The array job is made of many tasks and each task processes a given input data. The task can be used to ingest the results into the databases. LLGrid MapReduce allows to execute any programs that are using the appropriate database binding API commands. Through the scheduler's feature with the array job, it can control how many ingestion tasks can be processed concurrently. This allows users to specify the maximum number of concurrent ingestion tasks in order to optimize the database performance as well as controlling the number of compute resources in the pool.

As a demonstration for LLGrid MapReduce, a Python script is launched by LLGrid MapReduce to parse a set of web proxy log files, stored in the tabular CSV formation and to ingest the results into the Accumulo database. In this demonstration, the LLGrid MapReduce command creates an array job of 24 tasks that read and parse the input files of 3.8 GBytes as shown in Fig. 4. The processing takes about 5 minutes. The demonstration has been performed with two different configurations: an Accumulo setup with 8 nodes (7 tablet servers) and an Accumulo setup with 4 nodes (3 tablet servers). Each tablet server handles requests such as ingestion and query for one or more tablets, which in turn form an Accumulo database table. [1] Each node has 24 processing cores. Both cases were able to achieve about 1.5 million and 800,000 ingestion per second, respectively, in average.

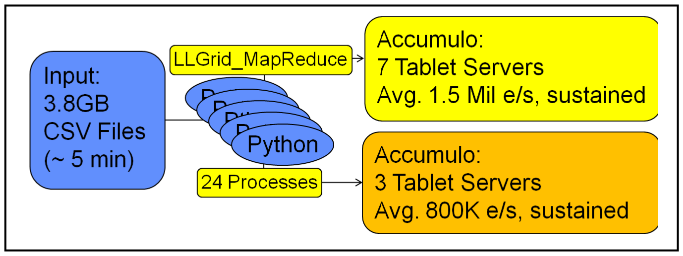

Figure 4. LLGrid MapReduce for Accumulo data ingestion using a Python script.

A similar but bigger set of web proxy log files are used for the ingestion scalability study as shown in Fig 5. In this case, two different sets of data are used. The size of the smaller set was about 17 GBytes, which holds about 200 files, with sizes are ranging from 8 to 200 Mbytes. This set is used for the ingestion experiment with up to 64 processes. With 128 and 256 processes, we used another set of web proxy log files made up of about 1000 files, approximately 90GBytes. The database used for the study is Accumulo (version 1.3.5) with 8 nodes (7 tablet servers). In this experiment, because of the nature of the row keys (base64-encoded hashes), we pre-split the table by the base64 characters to ensure 100% tablet coverage and even distribution across tablet servers. This, of course, requires secondary indices of relevant columns (i.e. the transpose of original table), where pre-spitting is done via decomposition of the value domain.

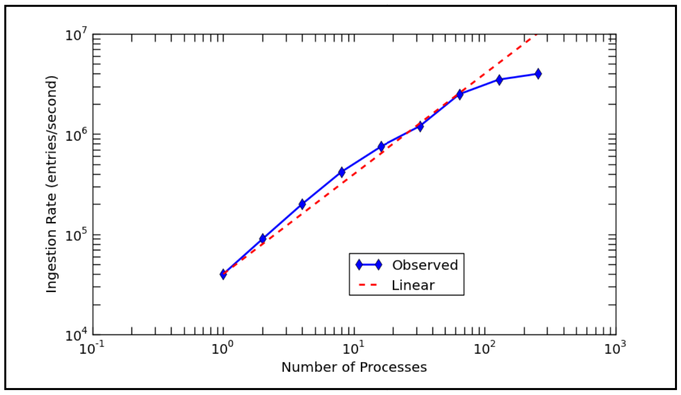

Figure 5. Total ingestion rate versus number of processes when running 7 tablet servers on an Accumulo cluster.

The result, shown in Fig. 5, shows that the ingestion rate scales superlinearly in the beginning and then, linearly up to 64 processes. With 128 and 256 cores, we were able to achieve average tablet server loads between 500K and 600K ingestion per second. However, beyond 64 processes, the total ingestion rate starts diminishing as the number of processes are growing. This indicates that peak ingestion rate has been achieved. In fact, with 7 tablet servers, using 128 processes produces significantly higher ingest performance per unit of load than using 256 processes. With 256 processes, the total ingestion rate was increased up to 4 million inserts per second. This is 100x performance improvement over the single ingestion process. The observed performance is considered to be a significant achievement when comparing the ingestion rate reported by a YCSB++ benchmark [13] although their cluster configuration and ingested data are quite different.

Another demonstration for data ingestion into the Accumulo has been performed using a D4M parallel MATLAB implementation on LLGrid. In this case, we have used the Graph500 benchmark [14] to generate the input data. A pMatlab application constructs a D4M associative array in parallel and ingests the associative array using the D4M put method, which binds to the Accumulo database API. By parallelizing the code with the parallel MATLAB, we can control the number of concurrent processes for data ingestion. Similar to LLGrid MapReduce example, the parallel MATLAB processes are executed on the LLGrid cluster (HPC compute nodes) and each process communicates with the Accumulo database directly.

Fig. 6 shows the average data insertion performance with multiple nodes when one or six tablet servers running in the Accumulo cluster (each node has 2 cores). The Accumulo database cluster is made of one Hadoop name node, which is also the master server for the Accumulo, and six Hadoop data nodes, which are also running the Accumulo tablet servers.

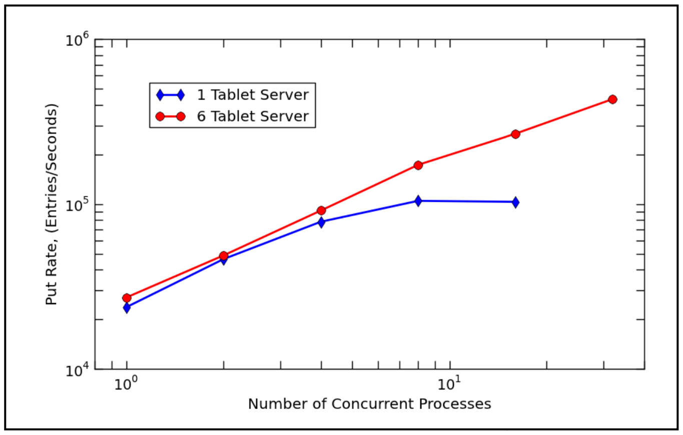

Figure 6. Total data insertion rate versus number of nodes when one or six tablet servers are running in the Accumulo cluster.

In this case, only one MATLAB process is running per node and each node is inserting about 2 million entries at a time and repeating it 8 times, which is total of 16 million entries per each MATLAB process. Since we fixed the ingestion data per each MATLAB process, as the number of MATLAB processes grows, so the size of the ingested data grows linearly. For a single tablet server case, as increasing the number of clients, the ingestion rate increases linearly initially, up to 4 clients and then, flattened out beyond 8 clients. When using 8 MATLAB clients, the ingestion rate peaked at about 105K entries/second and then decreased with 16 clients. However, if 6 tablet servers (one tablet server per each Hadoop HDFS data node) are running, the ingestion rate continues to scale well up to 32 clients.

Fig. 7 and 8 show the ingestion performance history with a single and six tablet servers in the Accumulo cluster, respectively. In both cases, 16 MATLAB clients are ingesting data into the Accumulo. Fig. 7 shows that the ingestion rate is topped out around 100K entries/second. It appears that the single tablet server was busy with 16 MATLAB clients all the time and reached its maximum ingestion rate with the current configuration.

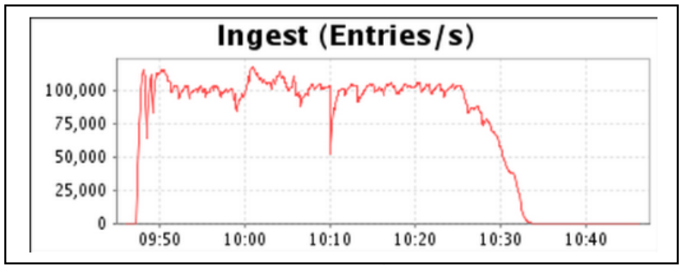

Figure 7. Time history of the data ingest rate when a single tablet server is running in the Accumulo cluster.

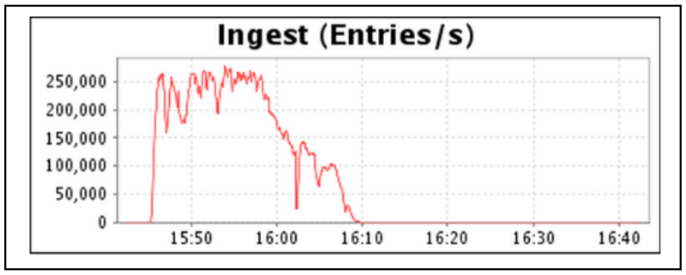

Figure 8. Time history of the data ingest rate when six tablet servers are running in the Accumulo cluster.

However, as shown in Fig. 8, when running six tablet servers (one tablet server per each Hadoop HDFS data node) there are still more rooms to accommodate incoming data ingestion requests and its peak ingestion rate becomes more than 250K entries/second. This is approximately 2.5 times greater than the case with a single tablet server. Although the ingestion rate does not scale linearly with the number of tablet servers, for ingestion performance, it is desirable to run one tablet server per each Hadoop data node.

V. Database Query Performance

Using appropriate query commands, the desired information can be extracted from databases. Accumulo is designed to search large amounts of data. We have studied how well it scales under various conditions. In this study, we used the D4M parallel MATLAB implementation of the Graph500 Benchmark to demonstrate a multi-node query. With D4M, the query operation is accomplished via an array indexing, which simplifies coding significantly.

In this study, we selected an arbitrary vertex in the graph and queried any column or row entries associated with it. The times for a couple of queries in the column and row direction, respectively, were measured and compared in Fig. 9 and 10. As expected, the column query times are 3 to 4 orders of magnitude larger than those of the row query. As increasing the number of the concurrent query clients, the column query time increases significantly where as the row query time is remained almost same although there is some abnormal deviation when running 8 MATLAB clients with a single tablet server as shown in Fig. 10. Fig. 9 also shows that the query takes a lot longer time to perform with one tablet server as compared to 6 tablet servers.

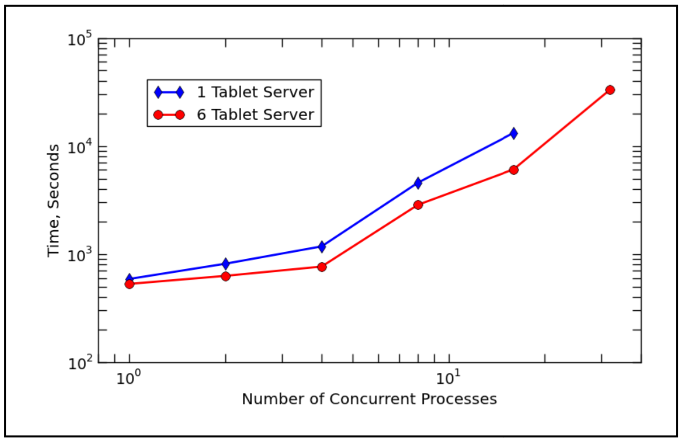

Figure 9. The column query times with respect to various number of concurrent query clients to the Accumulo database when running a single and six tablet servers, respectively.

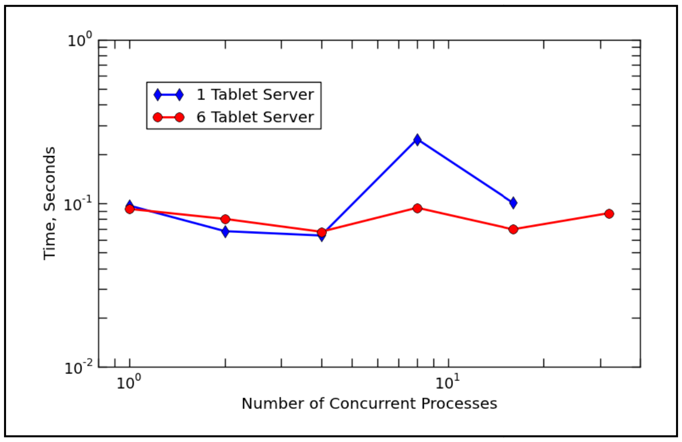

Figure 10. The row query times with respect to various number of concurrent query clients to the Accumulo database when running a single and six tablet servers, respectively.

Fig. 11 and 12 show the time history of the scan rate at the beginning and at the end of the query operations, which were requested by 16 concurrent MATLAB clients while only a single tablet server was running in the Accumulo cluster. The scan rate indicates how fast the scanner are able to retrieve the value associated with a given key. It also provides an insight of how many of the Accumulo tablets and tablet servers are being used and how busy they are for the given query. As shown in Fig. 11, over the period of the query time, its scan rate fluctuated significantly.

In Fig. 11, during the first 10 minutes, the scan rate was quite small as compared to the rest of the history. This could be caused by the fact that the test code does not have a barrier to synchronize the process between the ingestion and the query steps. Another interesting thing is that the scan rate was changed as a multiple of approximately 250K ingestions per second. Since in this simulation, only one tablet server is running in the Accumulo cluster, it indicates that the number of active tablets are varied over the time, which caused the scan rate changes as a step fashion.

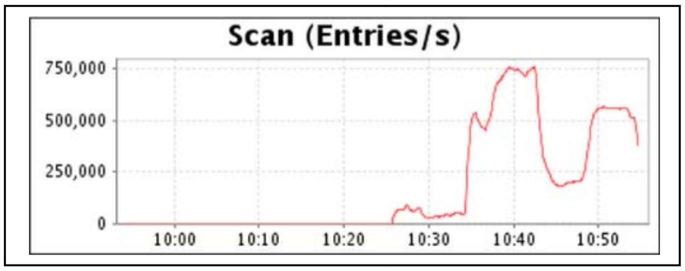

Figure 11. The scan rate history at the beginning of the query operation by 16 concurrent MATLAB clients while only a single tablet server is running.

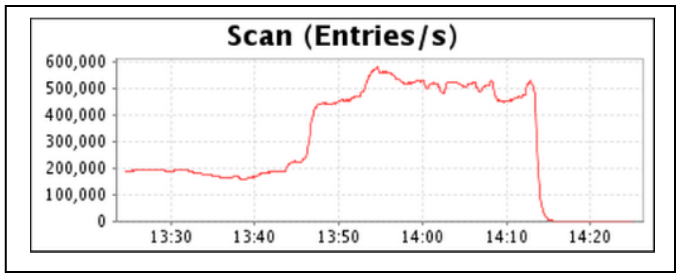

Figure 12. The scan rate history at the end of the query operation by 16 concurrent MATLAB clients while only a single tablet server is running.

Fig. 13 and 14 show a similar scan rate history graphs when 6 tablet servers (one per each data node) were running. These scan rate history graphs show that they are highly fluctuating with time. However, when comparing the peak scan rate, the scan rate with 6 tablet servers is about twice faster as compared to the scan rate with a single tablet server.

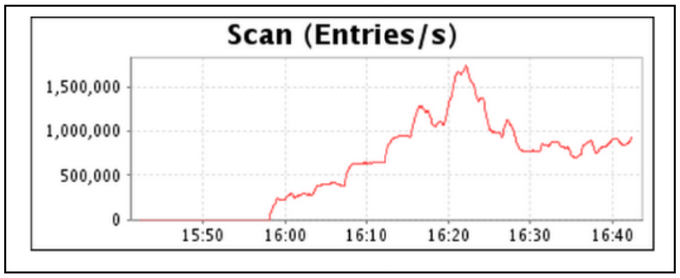

Figure 13. The scan rate history at the beginning of the query operation by 16 concurrent MATLAB clients while six tablet servers (one per data node) are running.

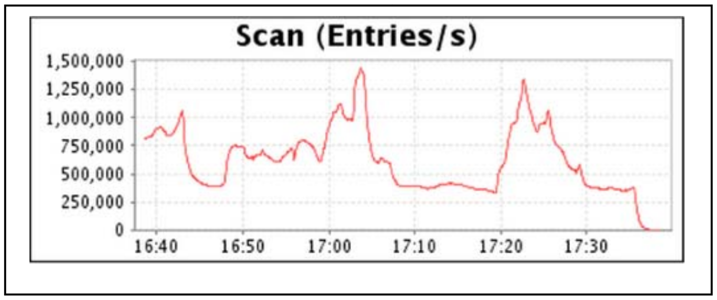

Figure 14. The scan rate history at the end of the query operation by 16 concurrent MATLAB clients while six tablet servers (one per data node) are running.

The performance of the scan rate fluctuates significantly with time when using 6 tablet servers because it depends on how many active Hadoop data nodes are participated at a given time in addition to the number of active tablets. With 6 tablet servers, since the scan operation is spread out among 6 tablet servers, the rate change becomes more volatile than what was observed with a single tablet server. As expected, overall query time is much shorter with 6 tablet servers: approximately two hours (6 tablet servers) and approximately four hours (single tablet server).

VI. SUMMARY

We have demonstrated that an MPI cluster environment can be used efficiently with a Hadoop cluster envrionment. LLGrid MapReduce and D4M along with pMATLAB technologies make it easy to write the big data applications. Both cases show that the data insertion and query scales well with the increasing the number of clients and nodes while running fully configured Accumulo clusters.

REFERENCES

[1] Apache Hadoop MapReduce http://hadoop.apache.org/mapreduce/

[2] Apache HBase http://hbase.apache.org

[3] Apache Accumulo http://accumulo.apache.org

[4] J. Kepner and S. Ahalt, "MatlabMPI," Journal of Parallel and Distributed Computing, vol. 64, issue 8, August, 2004.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", NSDI 2011, March 2011.

[6] N. Bliss, R. Bond, H. Kim, A. Reuther, and J. Kepner, "Interactive grid computing at Lincoln Laboratory," Lincoln Laboratory Journal, vol. 16, no. 1, 2006.

[7] J. Kepner et al., "Dynamic distributed dimensional data model (D4M) database and computation system," 37th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, March 2012.

[8] FUSE http://fuse.sourceforge.net/

[9] Beta Testing the Sun Grid Engine Hadoop Integration https://blogs.oracle.com/templedf/entry/beta_testing_the_sun_grid

[10] Open Grid Scheduler http://gridscheduler.sourceforge.net

[11] J. Kepner, Parallel Matlab for Multicore and Multinode Computers, SIAM Press, Philadelphia, 2009.

[12] N. Bliss and J. Kepner, "pMatlab parallel Matlab library," International Journal of High Performance Computing Applications: Special Issue on High Level Programming Languages and Models, J. Kepner and H. Zima (editors), Winter 2006 (November).

[13] S. Patil, et all., "YCSB++: Benchmarking and performance debugging advanced features in scalable table stores," ACM Symposium on Cloud Computing 2011, Cascais, Portugal, October 2011.

[14] Graph500 benchmark http://www.graph500.org