Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis

E. Dede, M. Govindaraju
SUNY Binghamton
Binghamton, NY 13902
{edede,mgovinda}@cs.binghamton.edu

D. Gunter, R. Canon, L. Ramakrishnan
Lawrence Berekely National Lab
Berkeley, CA 94720
{dkgunter,scanon,lramakrishnan}@lbl.gov

ABSTRACT

Scientific facilities such as the Advanced Light Source (ALS) and Joint Genome Institute and projects such as the Materials Project have an increasing need to capture, store, and analyze dynamic semi-structured data and metadata. A similar growth of semi-structured data within large Internet service providers has led to the creation of NoSQL data stores for scalable indexing and MapReduce for scalable parallel analysis. MapReduce and NoSQL stores have been applied to scientific data. Hadoop, the most popular open source implementation of MapReduce, has been evaluated, utilized and modified for addressing the needs of different scientific analysis problems. ALS and the Materials Project are using MongoDB, a document oriented NoSQL store. However, there is a limited understanding of the performance trade-offs of using these two technologies together. In this paper we evaluate the performance, scalability and fault-tolerance of using MongoDB with Hadoop, towards the goal of identifying the right software environment for scientific data analysis.

1. INTRODUCTION

Scientific domains such as bio-informatics, material science and light source communities are seeing exponential growth in data volumes, data variety, and the rate at which data needs to be analyzed. Exploratory data analysis, where scientists use data mining and statistical techniques to look for patterns, is difficult at this scale with currently available tools.

These scientific communities need new tools to handle the large amounts of semi-structured data. They require scalable methods for both simple queries as well as complex analysis. For example, the Materials Project [17] provides a community accessible datastore of calculated (and soon, experimental) materials properties of interest to both theorists and experimentalists. The Materials Project's underlying MongoDB datastore is used to capture the state of its high-throughput calculation pipeline outputs and views of the calculated material properties. The data being stored is complex, with hundreds of attributes per material, and continually evolving as new types of calculations and collaborators are added onto the project over time. MongoDB provides an appropriate data model and query language for this application. However, the project also needs to perform complex statistical data mining to discover patterns in materials and validate and verify the correctness of the data (*e.g.*, to see that a reference chemical reaction agrees with the computed values). These kinds of analytics are difficult with MongoDB, but naturally implemented as MapReduce programs (*e.g.* Mahout). Similarly, the Advanced Light Source's Tomography beamline uses MongoDB to store the metadata from their experiments and requires support for queries as well as analysis capabilities.

The MapReduce [11] model and NoSQL datastores have evolved due to large amounts of Internet and log data. The MapReduce programming model emerged for processing large data sets on large commodity clusters. The model allows users to write *map* and *reduce* functions to operate on the data and the system provides scalability and fault-tolerance. Traditionally, MapReduce jobs have relied on specialized file systems such as Google File System [15]. Apache Hadoop [3] is an open-source MapReduce implementation and in the last few years has gained significant traction. NoSQL [21] databases, does not use SQL or the relational model. NoSQL datastores achieve scalability through a distributed architecture and by trading consistency for availability.

Recently, a number of NoSQL stores have provided extensions that allow users to use Hadoop/MapReduce to operate on the data stored in these NoSQL data stores [19]. An integrated environment that allowed one to capture semi-structured data using a NoSQL store such as MongoDB, yet use the MapReduce model for analysis, would address the data requirements from these scientific communities. However, there is a limited understanding of the performance, scalability and fault-tolerance trade-offs in this integrated environment.

We take here a broad definition of semi-structured data as presented in [2], which includes all data that is neither "raw" nor very strictly typed as in conventional database systems. This includes data with irregular, implicit, or partial structure, and also integrated data that may be well-structured individually but which jointly defines a large and evolving schema. We believe that a significant portion of scientific data falls under this definition.

In this paper, we present the performance and reliability of an integrated analysis environment that consists of a

scalable document-based NoSQL datastore (MongoDB) and MapReduce framework (Hadoop). Specifically, we evaluate the performance, scalability and fault-tolerance of MongoDB with Hadoop.

Our results compare and contrast the trade-offs in performance and scalability between MongoDB and HDFS backends for Hadoop. We also study the effects of failures in both cases. Our results show that:

- HDFS performs much better than MongoDB for both reads and writes, with writes to MongoDB being the most expensive. Our experiments quantify this performance difference and we provide insights into the design choices that result in this gap.
- The *mongo-hadoop* connector provides the best approach to analyzing data that already resides or needs to be stored in MongoDB. The performance in this mode can be improved by directing the output to HDFS if that is feasible for the application.
- Hadoop's design strives to achieve data locality by placing the data on the compute nodes. Thus, node failures can block access to data. The *mongo-hadoop* framework, by separating the data servers and compute nodes, can tolerate high failure rates of the compute nodes; this is desirable in some scientific environments.

2. RELATED WORK

The applicability of Hadoop/HDFS and NoSQL data stores for scientific applications and its individual performance has been studied before. However, there is no prior work that characterizes the performance of NoSQL when used with Hadoop or another MapReduce implementation.

Bonnet et al. [7] describe how the MapReduce algorithm of MongoDB could help aggregate large volumes of data, but do not provide quantitative results. Verma et al. [24] evaluate both MongoDB and Hadoop MapReduce performance for an evolutionary genetic algorithm, but the MongoDB and Hadoop results use different configurations and the authors do not attempt to compare them. Neither work directly compares, as we do here, different combinations of using MongodB and Hadoop in an analysis environment.

There are other NoSQL databases that provide Hadoop support. Cassandra [16] is a peer to peer key-value store that has the ability to replace Hadoop's HDFS storage layer with Cassandra (CassandraFS). HBase [4] is an open source distributed column oriented database that provides Bigtable [8] inspired features on top of HDFS. HBase includes Java classes that allow it to be used transparently as a source and/or sink for MapReduce jobs. It has been previously shown that the right architectural changes to Hadoop and HBase can provide an improved solution to store and study big unstructured data [5]. These studies are not representative of performance one can get from NoSQL data stores that do not rely on HDFS. Our choice of MongoDB is motivated by the need for a document-oriented store for the scientific communities we work with.

Cooper et al. [10] compare different NoSQL and relational databases (Cassandra, HBase, Yahoo!'s PNUTS [9], and MySQL) through an extensible benchmark framework. This benchmark suite is an open source effort and can be extended to test different systems and workloads. Dory et al. [12] study the elastic scalability of MongoDB, HBase and Cassandra on a cloud infrastructure. These studies do not consider the performance achievable when using a MapReduce framework with these data stores.

Floratou et al. [14] compare NoSQL databases MongoDB and Hive to the relational database SQL Server PDW using YCSB [10] and TPC- H DSS [23] benchmarks. They compare these technologies for data analysis and interactive data serving. They show that while relational databases may perform better, NoSQL systems have usability advantages such as flexible data models, auto-sharding and load balancing (all of which are present in MongoDB).

The increasing use of NoSQL data stores in science and gaps in current literature led us to conclude that there is a need for more detailed evaluations of the use of NoSQL data stores as a back-end storage for Hadoop, such as this study of MongoDB.

3. TECHNOLOGY OVERVIEW

Figure 1 shows the configuration we used for our evaluation. In the center of the figure, the Hadoop MapReduce Framework connects the reads and writes from the Mappers and Reducers (boxes labeled "M" and "R") to either HDFS or, via the *mongo-hadoop* connector, to MongoDB. The remainder of this section gives more details on MongoDB and *mongo-hadoop*.

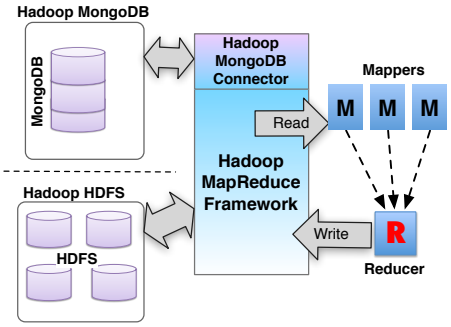

Figure 1. The high-level architecture of *mongo-hadoop*. Multiple mappers read the the input splits from either MongoDB (via the Hadoop/MongoDB connector) or HDFS. The intermediate output is collected in one reducer, which writes the results back to MongoDB or HDFS.

3.1 MongoDB

MongoDB [18][20] is an open source NoSQL "document store" database, commercially supported by 10gen [1]. Although MongoDB is non-relational, it implements many features of relational databases, such as sorting, secondary indexing and range queries.

MongoDB does not organize data in tables with columns and rows. Instead, data is stored in "documents", each of which is an associative array of scalar values, lists, or nested associative arrays. MongoDB documents are serialized naturally as Javascript Object Notation (JSON) objects, and are in fact stored internally using a binary encoding of JSON called BSON [6].

To scale its performance on a cluster of servers, MongoDB uses a technique called *sharding*, which is the process of splitting the data evenly across the cluster to parallelize access.

This is implemented by breaking the MongoDB server into a set of front-end routing servers (`mongos`), that route operations to a set of back-end data servers (`mongod`).

MongoDB queries examine one record at a time, which means that queries across multiple records must be implemented on the client or use MongoDB's built-in MapReduce (MR). Though MongoDB's MR can be executed in parallel at each shard, there are two major drawbacks: (1) the language for MR scripts is JavaScript, which is slow and has poor analytics libraries, and (2) the SpiderMonkey [22] Javascript implementation used by MongoDB, is not thread-safe, so only one MapReduce program can run at a time.

3.2 MongoDB-Hadoop Connector

The MongoDB-Hadoop Connector [19] is an open-source plugin for Hadoop that allows MongoDB to be used, instead of HDFS, as a source and sink of data.

The connector allows the user to specify a query, and breaks the results of that query into *input splits* for Hadoop. For sharded MongoDB servers, the splits are done on 64MB shard chunks. The splits are shipped to the mappers as queries to retrieve the actual data, so that each mapper can read its splits in parallel. Results are written back to MongoDB by the Hadoop reducer. Note that HDFS is not involved in any one of these operations. This provides an alternative approach to either running MapReduce in MongoDB directly, or performing a three-stage operation: loading the data from MongoDB to HDFS, running Hadoop MapReduce, and importing the output back into MongoDB. Both of these approaches have drawbacks for complex operations on large data sets. The problems with the Mongo-MapReduce approach were noted in 3.1. The three-stage approach is inconvenient and requires a large database and HDFS I/O. The MongoDB-Hadoop Connector, which allows the user to leave the input data in database, is thus an attractive option to explore. The connector can optionally leave the output in HDFS, which allows for different combinations of read and write resources.

3.3 HDFS vs MongoDB Design: A Comparison

Table 1 summarizes the data access and reliability characteristics of HDFS and MongoDB.

As our results show, these characteristics have important implications for the relative performance of MapReduce workloads. HDFS is optimized for sequential reads and writes of data in relatively large chunks. MongoDB is optimized for random and parallel access, i.e. queries to the data. Our results also show that MongoDB exhibits poor performance for parallel writes due to the global write lock.

Both MongoDB and HDFS provide data reliability through replication. In the case of MongoDB, the client can choose how many replicas finish a write (to the journal) before the operation returns success; this has scalability advantages, but is also an opportunity for catastrophic data loss if the client is mis-configured. HDFS replication replicates the data the specified number of times either based on a system wide parameter or as controlled by the user.

4. EXPERIMENT SETUP

For our experiments we configured and deployed Hadoop, HDFS, MongoDB, and the Mongo-Hadoop connector on machines at NERSC and Binghamton University.

Hadoop/HDFS Setup. We deployed HDFS using standard configuration parameters. We allowed the Hadoop framework to choose the number of maps (the user's suggestion may be ignored anyways), and chose the default one (1) reducer.

Mongo-Hadoop Connector Setup. We ran MongoDB in two modes: as a single server and with sharding. For the single server tests, each mapper connects to the single MongoDB server. For sharded servers, load balancing among `mongos` routers is achieved by having each mapper randomly pick one `mongos` to communicate with. Our monitoring showed that our randomization process resulted in a fairly even load balancing.

Machines. We conducted our experiments on the NERSC Hopper machine and the cluster at Grid and Cloud Computing Research Lab Cluster (GCRL) at Binghamton University. This provided performance results for both an HPC system and a typical smaller research cluster.

Hopper is a Cray XE6 with 153,216 compute cores, 217 Terabytes of memory, and 2 Petabytes of disk. Each node has 24 cores, 2 twelve-core AMD 'MagnyCours' 2.1-GHz processors and 32 GB of RAM. On Hopper, we ran the MongoDB server on a "MOM node" that is used for managing and launching parallel applications.

On the GCRL cluster we ran the Hadoop namenode on a dual core 2.4Ghz Intel Core 6600 with 2 GB of RAM, running Linux 2.6.24. The Hadoop cluster nodes are Quad core, 1U nodes with 2.6Ghz Intel Xeon CPUs, 8 GB of RAM and run a 64-bit version of Linux 2.6.15. The MongoDB servers ran on two nodes: one which is has two 2.6Ghz Intel Xeon CPUs, 16 GB of RAM and 48 cores and another which has two 2.4Ghz Intel Xeon CPUs, 16 GB of RAM 24 cores.

5. WORKLOADS

Our evaluation uses representative benchmarks that operate on public datasets such as the U.S. Census data. In the U.S. Census dataset, each record is a row of 111 comma separated values. The total census data is about 300GB and we use subsets of the data as applicable for the experiments.

Our evaluation performs three operations, which represent different input and output data size ratios: filter, merge, and reorder. The *filter* operation extracts the counties with a bigger female population than a male population; the output data is relatively small. For *merge*, we label the census data for human search and viewing; since each line is annotated, the output is larger than the input. For the *reorder* operation, we globally change the field separator to a different character; here, the output and input are the same size. In this evaluation, we do not include our results for reorder and merge due to space constraints.

These benchmarks are representative of the kind of operations that scientists need to perform on their data. For example, the *filter* operation can be generalized to any operation that groups some value calculated for each record by another set of fields in those records. An example of the

Table 1. HDFS and MongoDB

Characteristic	HDFS	MongoDB
Storage	Distributed file system	Distributed schema-less database, in-memory at each node
Reading	Sequential, block access	Random and sequential access, BTree indexes
Writing	Cached locally then sent to DataNode	Journaled writes to index and data blocks, per-server global write lock
Reliability	Replication	Replication

same type of query in the Materials Project data would be, *"find all materials containing oxygen and group them by the ceiling of their formation energy in eV"*. This type of operation is related to a join between two sets of records, for example checking that the reduced (simplified) formula in a material is the same across all the calculation tasks that were performed with this material, or more complex comparisons, such as verifying that phase diagrams are calculated correctly. Because these operations are needed in many contexts, our benchmarking will provide valuable information to scientific communities using these technologies.

6. EVALUATION

In this section, we analyze performance results to quantify the performance differences of *mongo-hadoop* in different usage scenarios. We repeated all experiments three times. The results use the mean value; the variance was uniformly low.

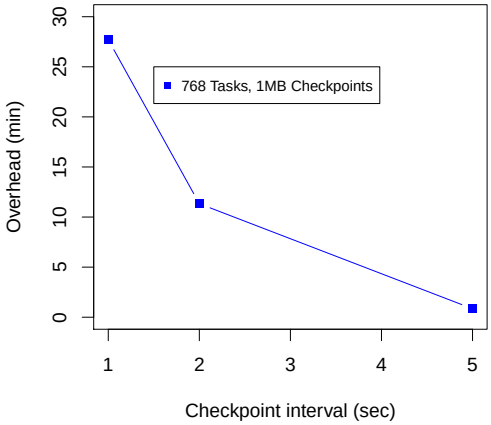

Figure 2. Task manager running on 192 cores on NERSC Hopper. 768 tasks being processed by each worker with increasing checkpoint intervals. From checkpointing every second to every five seconds the overhead of the `task manager` drops almost 5 times.

6.1 Evaluation of MongoDB

The first experiment is designed to study how MongoDB would perform at large scale, *e.g.* on NERSC's HPC cluster, Hopper. Our experiment used MongoDB to keep a centralized state for a distributed system with thousands of worker nodes running in parallel. A standard approach to providing fault tolerance and reliability in such a system is *checkpointing*, in which each worker periodically reports their status to the central (MongoDB) server. This test stresses the ability of MongoDB to handle large numbers of parallel connections.

Figure 2 shows the application time to complete 768 tasks with different checkpoint intervals. The tasks take 10 minutes, and all run in parallel across the 192 cores (i.e. four tasks per core). With no checkpoints, the total run-time is

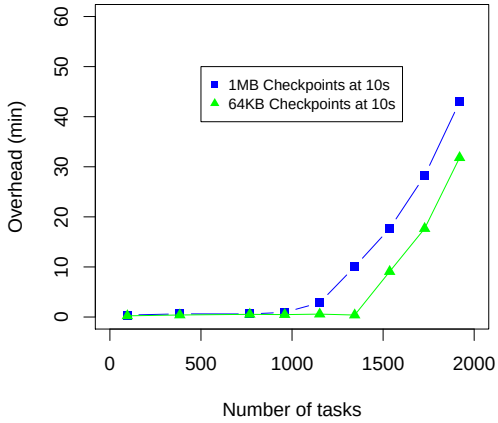

Figure 3. Task manager running on 24 to 480 cores on NERSC's Hopper cluster. Number of tasks have also been increased along with the core increases. More tasks require more cores which creates more connections to the MongoDB server. The checkpoint interval here is 10 seconds while each task takes nearly 10 minutes.

$\approx$ 10 minutes, so we subtract this time and report the difference as "overhead". When the checkpoint interval is five seconds the overhead is below 1 minute ($< 10\%$). Reducing the checkpoint interval from five seconds to one second increases the overhead to over 25 minutes ($> 250\%$). This degradation in performance is due to the increasing number of connections per second, from 154 to 768, as well as the increase in total write volume to 768MB/s.

Figure 3 shows how the overhead increases with the number of tasks and nodes, for two checkpoint sizes. In this test, the individual task time is 10 minutes, all the tasks are run in parallel. The checkpoint interval is held fixed at 10 seconds. In either case, the checkpoints do not add overhead until after 1000 parallel tasks. For 1MB checkpoint size runs, the performance is five times slower from 1152 tasks to 1920 tasks. On the other hand, for 64KB checkpoint size the performance drops almost four times from 1344 to 1920 tasks.

Although data volume and number of connections both contribute to the overhead, the results make it clear that large numbers of connections are a more significant bottleneck than the total data volume: at 1920 tasks, the 1MB checkpoints add only $\approx 30\%$ more overhead than the 64KB checkpoints, despite having over $15\times$ the data volume. This is due to MongoDB's large per-connection overhead, since a separate thread with its own stack is allocated for every new connection.

6.2 HDFS vs MongoDB Performance

Next, we compare the read and write performance of MongoDB and HDFS. For this, we developed a Java program and a Python script that each read 37M records, and wrote

19M records, to and from HDFS and MongoDB. These tests are independent of our later evaluation of *mongo-hadoop*.

The goal is to compare the HDFS and MongoDB read/write performance from a single node. In this experiment, we have two HDFS data nodes and a MongoDB setup with two sharding servers. In the read tests, the single node reads 37.2 million input records. Each record consists of 111 fields and it is the same type of input records used in the Census benchmarks. For this record size, we can read from HDFS at a rate of 9.3 million records/minute and from MongoDB at a rate of 2.7 million records/minute.

For write experiments, the records had only one field, also as in our Census benchmarks. This means that the records being written were roughly 100 times smaller than the records that were read. For these records, the HDFS writes the 19 million records in 15 seconds (74.5 million records/minute), whereas MongoDB takes 6 minutes (3.2 million records/minute).

In summary, we see a 3:1 ratio between HDFS and MongoDB in read performance for large records and a significantly larger 24:1 difference in write performance for very small records.

6.3 MongoDB MapReduce

MongoDB has its own built-in MapReduce implementation (described in 3.1). The native MapReduce scales with the number of shard servers, *e.g.*, for eight MongoDB servers eight map tasks will be launched.

Figure 4 compares MongoDB's native MapReduce (MR) with *mongo-hadoop* MapReduce. To obtain these results, we ran MongoDB on a single server and used a 2-node Hadoop cluster. For this configuration, the *mongo-hadoop* plug-in provides roughly five times better performance. The graph shows that the performance gain from using *mongo-hadoop* increases linearly with input size in the give range of input records.

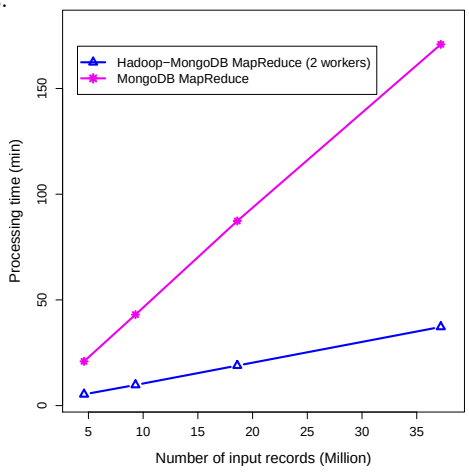

Figure 4. Native MongoDB MapReduce (MR) versus *mongo-hadoop* MR. Experiment used 1 MongoDB server and, for *mongo-hadoop*, 2 Hadoop workers.

6.4 Evaluating MongoDB Configurations

The split size is an important performance parameter because it affects the number of mappers. Each mapper reads a split from the MongoDB server, does processing and sends its intermediate output to the reducer. For large data sets,

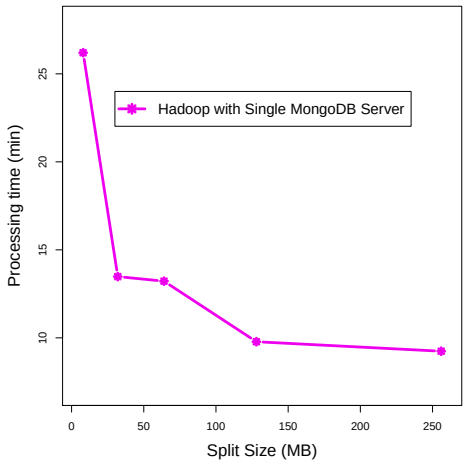

Figure 5. 9.3 Million input records on a *mongo-hadoop* setup. The split size is varied from 8MB to 256MB.

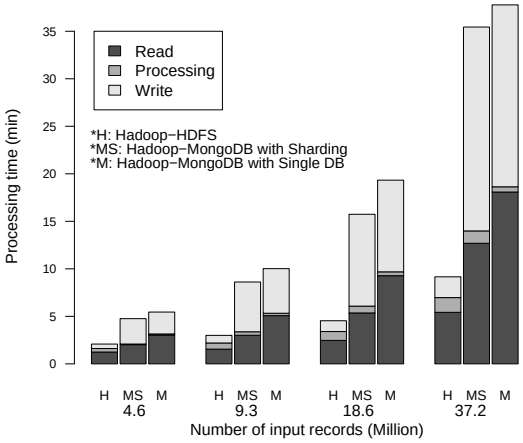

Figure 6. `Hadoop-HDFS` versus *mongo-hadoop* with varying input sizes on a two node Hadoop cluster. For *mongo-hadoop*, most of the time is spent in writing while for `Hadoop-HDFS` it is in reading since this is not a write intensive test and output sizes do not overwhelm HDFS.

a split size of 8MB results in hundreds of thousands of mappers. Previous work shows [13] that too many mappers can add a significant task management overhead. For example, while 100 mappers can provide great performance for a 100GB input, the same 100 mappers will degrade the performance for an input set of a few megabytes.

Figure 5 shows the effect of the split size on performance using *mongo-hadoop*. The number of input records is ≈ 9.3 million, or 4GB of input data. With the default split size of 8MB, Hadoop schedules over 500 mappers; by increasing the split size, we are able to reduce this number to around 40 and achieve a considerable performance improvement. The curve levels off between 128MB and 256MB, so we decided to use 128MB as the split size for the rest of our tests both for native `Hadoop-HDFS` and *mongo-hadoop*.

Figure 6 shows a *filter* test run on a 2-node Hadoop cluster. This figure shows `Hadoop-HDFS` and *mongo-hadoop* with a single MongoDB server and a sharding setup with two MongoDB servers. For 4.6 million input records, HDFS performs two times better than MongoDB, and for increasing input sizes the gap becomes larger: at 37.2 million records, HDFS is five times faster than MongoDB. The graph shows

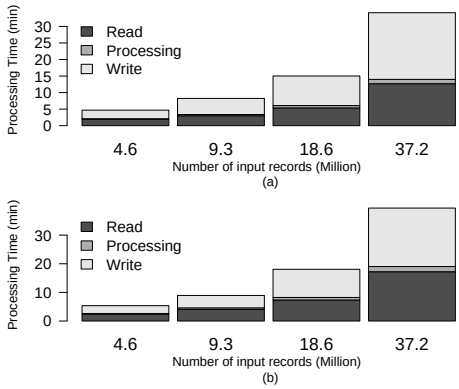

Figure 7. MongoDB Sharding setup is placed on the worker nodes versus remote nodes on a 2 node Hadoop cluster. Placing the servers on the workers causes competition for CPU time and memory and the workers get slower.

a slight performance advantage for *mongo-hadoop* with a sharding setup vs. a single server.

The figure shows that replacing HDFS with MongoDB causes worse performance for both reads and writes. Compared to `Hadoop-HDFS`, *mongo-hadoop* with a single data server is more than 3 times slower in reading while it performs nine times worse for writes. This is for processing 37.2 million input records. In a sharded setup, *mongo-hadoop* reading times improve considerably, as there are multiple servers to respond to parallel worker requests. However, the writing times are still over 9 times slower.

For the results shown in Figure 7, we placed the MongoDB sharding servers onto the Hadoop worker nodes and forced each worker to use the local sharding server as the input/output source. The aim of this test was to imitate HDFS input data locality. The performance slightly worsened compared to running the servers on different machines. We believe that this was due to memory contention. MongoDB uses `mmap` to aggressively cache data from disk into memory, and thus shares badly with other memory-hungry programs such as the Java virtual machine. As we increase input data sizes, we observe increased memory and CPU usage on the nodes for the MongoDB operations and this in turn lead to slower times.

6.5 Scalability Tests

Figure 8 shows the performance over increasing cluster sizes from 16 to 64 cores with HDFS, single MongoDB server and two MongoDB sharded servers. The read times improve in the sharded setup as it is able to balance the increasing number of concurrent connections. As explained in the previous graphs, the write time is bound by the reduce phase for this MapReduce job. For a case where many reducers are working in parallel, a sharding setup will also help the write times by providing better response for multiple concurrent connections. However, the write performance of MongoDB still remains to be a bottleneck along with the overhead of routing data to be written between sharding servers.

Figure 9 shows the comparison of performance of various combinations of using `Hadoop` and `MongoDB` for reads and writes. The *mongo-hadoop* setup can be used in various combinations like reading the input from MongoDB, then storing the output to HDFS, and vice versa. `Hadoop-HDFS`

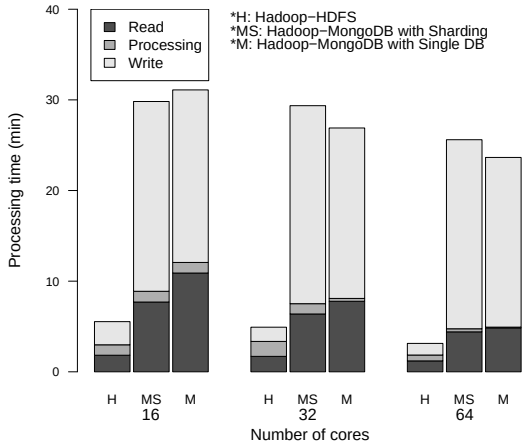

Figure 8. `Hadoop-HDFS` versus *mongo-hadoop* with varying cluster sizes from 16 cores to 64 for 37 million input records. Sharding MongoDB performs better against increasing number of parallel mappers.

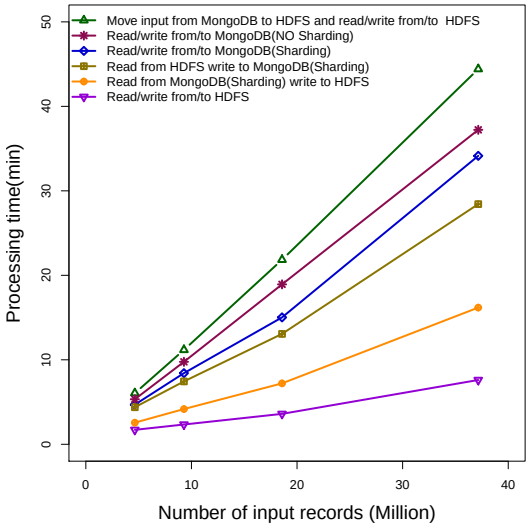

Figure 9. Hadoop and MongoDB cooperated in different usage scenarios on a 4 core Hadoop cluster with increasing data sizes. While `Hadoop-HDFS` performs best for a large dataset that is already in MongoDB, moving the input to HDFS creates huge overhead which makes *mongo-hadoop* approach a more desirable option.

provides the best performance, as expected. For an input dataset that is stored in MongoDB and needs to be analyzed, best performance can be achieved when data is read from MongoDB and the output is written to HDFS as it is more write efficient. The alternate approach for analyzing the data in MongoDB would entail downloading the data to HDFS before running the analysis which, as shown, has the slowest performance. Thus, for input data that is already stored in MongoDB, the plugin provides a good alternative, since the cost of downloading data to HDFS increases with the growing input sizes.

Figure 10 shows the performance over increasing cluster size (from 8 cores to 64) for 37.2 million input records. With an increasing number of worker nodes, the concurrency of the map phase also goes up, and therefore the map times get considerably faster. However, as the application does not require many reducers, the increasing number of nodes does

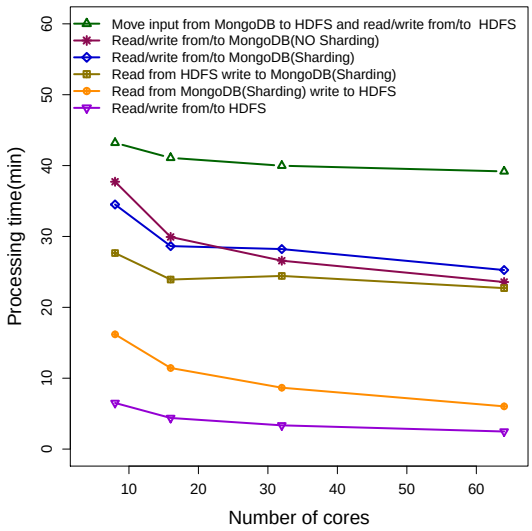

Figure 10. Hadoop and MongoDB cooperated in different usage scenarios for processing 37.2 million input records on a varying sized cluster from 8 cores to 64.

not have a big impact on the reduce phase. Moreover, we do not see much of a performance gain after 16 nodes since the overhead of launching a large number of parallel tasks for a relatively small data set in this case (nearly 17GB) reduces any effects of increased parallelism.

6.6 Fault Tolerance

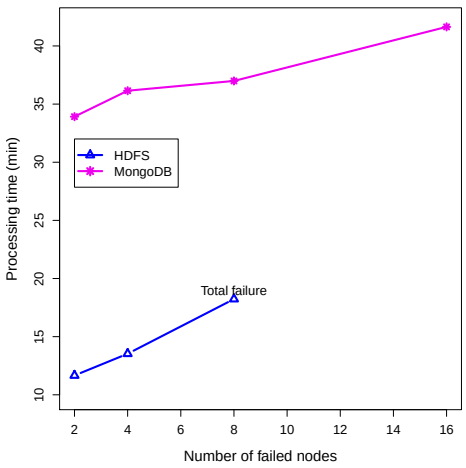

Figure 11. `Hadoop-HDFS` versus *mongo-hadoop* on a 32 node Hadoop cluster processing 37 million input records. After 8 faulted nodes `Hadoop-HDFS` loses too many data nodes and fails to complete the MapReduce job. *mongo-hadoop* gets the input splits from the MongoDB server therefore losing nodes does not lead to lose of input data.

Figure 11 shows the fault tolerance evaluation of *mongo-hadoop* versus `Hadoop-HDFS` on a 32 node Hadoop cluster processing 37 million input records. After eight faulted worker nodes `Hadoop-HDFS` loses too many data nodes and fails to complete the MapReduce job since the worker nodes also host chunks of the input data. On the other hand, *mongo-hadoop* gets the input splits from the MongoDB server therefore losing worker nodes does not lead to loss of input data. Thus, the job is able to finish even with half of the cluster lost. Note that losing the MongoDB server or one of the servers in the case of sharding will create the same effect and the MapReduce job will fail.

7. DISCUSSION

Our experiments provide important insights into the use of MongoDB for storage and Hadoop for analysis. Below we summarize our key insights:

- A single MongoDB shows a considerable deterioration in performance between 1000 and 1500 concurrent threads. Sharding helps to improve MongoDB's performance especially for reads.
- In cases where data is already stored in MongoDB and needs to be analyzed, the *mongo-hadoop* connector provides a convenient mechanism to use Hadoop for scalability. Better performance can be achieved if the output of the analysis could be written to HDFS instead of back to MongoDB especially with larger data volumes.
- One of Hadoop's strengths is data locality that is achievable by combining the data and compute servers. The downside of this configuration is that availability variations affect both systems simultaneously. Using MongoDB as a back-end separates data server availability from compute server availability, allowing the system to be more resilient (e.g. the loss of compute nodes).
- In MongoDB, reducing the checkpoint interval significantly increases the overhead. In our experiments, when the interval was changed from five seconds to one second, the overhead increased by a factor of 2.5. This degradation in performance is due to the increasing number of connections, increasing write requests per second, as well as the increase in total write volume.
- Although, data volume and number of connections both contribute to the overhead of MongoDB, our results show that the large numbers of connections cause a more significant bottleneck than the total data volume.
- The *mongo-hadoop* plug-in provides roughly five times better performance compared to using MongoDB's native MapReduce implementation. The performance gain from using *mongo-hadoop* increases linearly with input size.
- By increasing the split size, the performance of *mongo-hadoop* can be considerably improved. Split size, and the number of maps being launched, should be examined and configured accordingly in order to get the best performance for a given input data size and number of nodes in the Hadoop cluster.
- Replacing HDFS with MongoDB causes worse performance for both reads and writes. While processing 37.2 million input records, compared to `Hadoop-HDFS`, *mongo-hadoop* with a single data server is more than three times slower in reading, it performs nine times worse in the output write phase. In a sharded setup,

mongo-hadoop reading times improve considerably, as there are multiple servers to respond to parallel worker requests. The writing times, however, are still over nine times slower.

- In a 32 node cluster, with 8 node failures, `Hadoop-HDFS` fails to complete the MapReduce job since the worker nodes also host chunks of the input data. On the other hand, *mongo-hadoop* gets the input splits from the MongoDB server therefore losing worker nodes does not lead to loss of input data. Thus, the job is able to finish even with half of the cluster lost.

8. CONCLUSION

In this paper we evaluated the combination of the MapReduce capabilities of Hadoop with the schema-less database MongoDB, as implemented by the *mongo-hadoop*plugin [19]. This study provides insights into the relative strengths and weaknesses of using the MapReduce paradigm with different storage implementations, under different usage scenarios. MongoDB, like other datastores, is optimized for queries. In cases where the dataset needs to be both queried as well as analyzed, the connector provides a convenient mechanism to achieve scalability for the analysis tasks using Hadoop. In general, we found that this solution is appropriate if the workload uses MongoDB as a database that needs to be occasionally used as a source of data for analytics as in the case of the Materials Project. However, it is not appropriate when using MongoDB as an analytics platform that sometimes must act like a database. We show that using Hadoop for MapReduce jobs is several times faster than using the built-in MongoDB MapReduce capability. As a back-end for Hadoop, MongoDB is much slower than HDFS, primarily due to the design differences between MongoDB, a NoSQL database, and HDFS, a file management system.

References

[1] 10gen, The MongoDB Company. `http://www.10gen.com`.

[2] S. Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory*, ICDT '97, pages 1–18, London, UK, UK, 1997. Springer-Verlag.

[3] Apache Hadoop. `http://hadoop.apache.org`.

[4] Apache HBase. `http://hbase.apache.org`.

[5] K. Bakshi. Considerations for big data: Architecture and approach. In *Aerospace Conference, 2012 IEEE*, pages 1 –7, march 2012.

[6] Binary JSON. `http://bsonspec.org/`.

[7] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, you say: What can do for data aggregation and bi in large repositories. In *Proceedings of the 2011 22nd International Workshop on Database and Expert Systems Applications*, DEXA '11, pages 483–488, Washington, DC, USA, 2011. IEEE Computer Society.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] T. Dory, B. MejÃ■as, P. V. Roy, and N.-L. Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.

[13] Z. Fadika and M. Govindaraju. Lemo-mr: Low overhead and elastic mapreduce implementation optimized for memory and cpu-intensive applications. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society.

[14] A. Floratou, N. Teletia, D. Dewitt, J. Patel, and D. Z. Zhang. Can the elephants handle the nosql onslaught? *VLDB*, 2012.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[16] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[17] The Materials Project. `http://materialsproject.org`.

[18] MongoDB. `http://www.mongodb.org`.

[19] MongoDB + Hadoop Connector. `http://api.mongodb.org/hadoop/`.

[20] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[21] J. Pokorny. Nosql databases: a step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[22] Spider Monkey. `https://developer.mozilla.org/en/SpiderMonkey`.

[23] The TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[24] A. Verma, X. Llora, S. Venkataraman, D. Goldberg, and R. Campbell. Scaling ecga model building via data-intensive computing. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –8, july 2010.