An Evaluation of Cassandra for Hadoop

Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, Madhusudhan Govindaraju
Grid and Cloud Computing Research Laboratory
SUNY Binghamton, New York, USA
Email:{ededе1,bsendir1,pkuzlu1,jhartog1,mgovinda}@binghamton.edu

***Abstract*—In the last decade, the increased use and growth of social media, unconventional web technologies, and mobile applications, have all encouraged development of a new breed of database models. NoSQL data stores target the unstructured data, which by nature is dynamic and a key focus area for "Big Data" research. New generation data can prove costly and unpractical to administer with SQL databases due to lack of structure, high scalability, and elasticity needs. NoSQL data stores such as MongoDB and Cassandra provide a desirable platform for fast and efficient data queries. This leads to increased importance in areas such as cloud applications, e-commerce, social media, bio-informatics, and materials science. In an effort to combine the querying capabilities of conventional database systems and the processing power of the MapReduce model, this paper presents a thorough evaluation of the Cassandra NoSQL database when used in conjunction with the Hadoop MapReduce engine. We characterize the performance for a wide range of representative use cases, and then compare, contrast, and evaluate so that application developers can make informed decisions based upon data size, cluster size, replication factor, and partitioning strategy to meet their performance needs.** [1]

I. Introduction

With the advent of the "Big Data" era, the size and structure of data have become highly dynamic. As application developers deal with a deluge of data from various sources, they face challenges caused by the data's lack of structure and schema. As such data grows and is constantly modified via social media, news feeds, and scientific sensor input, the requirements from the storage models have also changed. As the unstructured nature of the data limits the applicability of the traditional SQL model, NoSQL has emerged as an alternative paradigm for this new non-relational data schema. NoSQL frameworks, such as DynamoDB [1], MongoDB [6], BigTable [10] and Cassandra [23], address this "Big Data" challenge by providing horizontal scalability. This, unlike the vertical scalability scheme of traditional databases, results in lower maintenance costs.

While the NoSQL model provides an easy and intuitive way to store unstructured data, the performance under operations common in cloud applications is not well understood. The MapReduce model has evolved as the paradigm of choice for "Big Data" processing. However, studies with performance insights on the applicability of the MapReduce model to NoSQL offshoots, such as MongoDB and Cassandra, have been lacking. As "modern" data is increasingly produced from various sources, it becomes increasingly unstructured while continually growing in size with user interaction; it is important to evaluate the NoSQL model when used with the MapReduce processing paradigm. In this paper, we analyze various considerations when using Cassandra as the data store and Apache Hadoop for processing. Cassandra is an open source non-relational, column oriented distributed database. It is used for storing large amounts of unstructured data. Apache Hadoop is a well-known platform for data intensive processing.

We first identify and analyze various aspects of Cassandra, and by extension NoSQL object stores, such as locality, scalability, data distribution, load balancing, and I/O performance. We provide insights on strengths and pitfalls of using Cassandra as the underlying storage model with Apache Hadoop for typical application loads in a cloud environment. We also present and analyze the performance data of running the same experiments using Hadoop native, which uses Hadoop Distributed Hadoop Distributed File System (HDFS) for storage.

The contributions of this paper are as follows:

- Identify and describe the key NoSQL features required for efficient performance with Hadoop.
- Discuss how the various features of Cassandra, such as replication and data partitioning, affect Apache Hadoop's performance.
- Analyze the performance implications of running Hadoop with Cassandra as the underlying data store. Verify performance gains and losses by processing application data typical in cloud environments, as well as classical I/O and memory intensive workloads.

II. Background

A. MapReduce and Hadoop

The MapReduce model proposes splitting a data set to enable its processing in parallel over a cluster of commodity machines, which are called `workers`. Input distribution, scheduling, parallelization and machine failures are all handled by the framework itself and monitored by a node called the `master`. The idea is to split parallel execution into two phases: `map` and `reduce`. `Map` processes a key and produces a set of intermediate key/value pairs. `Reduce` uses the intermediate results to construct the final output.

Apache Hadoop [2] is the most popular open source implementation of the model. Hadoop consists of two core components: The Hadoop MapReduce Framework and the Hadoop Distributed File System (HDFS) [27]. Hadoop MapReduce consists of a *JobTracker* that runs on the `master` node and

[1]Supported in part by NSF grant CNS-0958501.

TaskTackers running on each of the `workers`. The *JobTracker* is responsible for determining job specifications (i.e., number of `mappers`, etc), submitting the user job to the cluster and monitoring `workers` and the job status. The *TaskTrackers* execute the user specified `map` or `reduce` tasks. Hadoop relies on HDFS for data distribution and input management. It automatically breaks data into chunks and spreads them over the cluster. Nodes hosting the input splits and replicas are called *DataNodes*. Each *TaskTacker* processes the input chunk hosted by the local *DataNode*. This is done to leverage data locality. The input splits are replicated among the *DataNodes* based on a user-set `replication-factor`. This design prevents data loss and helps with fault tolerance in case of node failures.

B. YSCB Benchmarking Suite

The Yahoo Cloud Serving Benchmark (YCSB) [12] has been developed by Yahoo! engineers to analyze different data stores under several workloads. YCSB is an open source project that features benchmarks for many NoSQL technologies like Hbase [3], MongoDB [6], PNUTS [11], and Cassandra [23]. The YCSB Core Package features five basic workloads and each can be extended to produce new ones. In **Section** III-A, we present performance results for running the YSCB benchmarks Workload C on Cassandra under various scenarios. When executed in `load` mode, this workload inserts a user-specified number of randomly generated records into the Cassandra database. Each record contains a randomly generated key, 10 fields and each field is of 100 bytes. In order to compare Cassandra performance under different configurations and loads, we use the YCSB default "insertorder" which inserts the records in hashed order of keys. In the `run` mode, the user-specified number of records are read. Each record is read as a whole without specifying any *columns*.

C. Cassandra

Cassandra [23], developed by Facebook, is an open source, non-relational, column oriented, distributed database, developed for storing large amounts of unstructured data over commodity servers. Cassandra is a peer-to-peer model, which makes it tolerant against single points of failure and provides horizontal scalability. A Cassandra cluster can be expanded on demand by simply starting new servers. These servers only know the address of the node to contact to get the start-up information.

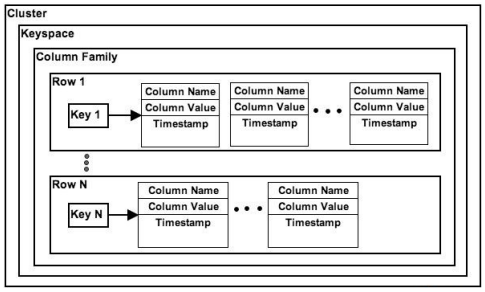


Fig. 1. Hierarchy of the Cassandra data model. A Cassandra *cluster* can host many *keyspaces*, which can contain various *column families*. These contain *rows*, which are further composed of *columns*. The number of *columns* and even the *columns* themselves can vary between *rows*.

1) Data Model: **Figure** 1 shows the column oriented data model of Cassandra. A *column* is the smallest component of data and it is a tuple of *name*, *value* and *time stamp*. Time stamps are used for conflict resolution as multiple versions of the same record may be present. *Columns* associated with a certain key can be depicted as a *row*; *rows* do not have a pre-determined structure as each of them may contain several *columns*. A *column family* is a collection of *rows*, like a table in a relational database. *Column families* are stored using separate files, which are sorted in row key order. The placement of *rows* on the nodes of a Cassandra cluster depends on the row key and the partitioning strategy. *Keyspaces* are containers for *column families* just as databases have tables in RDBMSs.

2) Replication and Consistency: Cassandra has automatic replication to duplicate records throughout the cluster by a user set `replication-factor`. This is to ensure that failing nodes do not result in data loss. Cassandra offers configurable consistency, which provides the flexibility to consciously make trade-offs between latency and consistency. For each read and write request, users choose one of the pre-defined consistency levels: *ZERO, ONE, QUORUM, ALL* or *ANY* [23]. In the experiments of **Section** III-A we use the consistency level of *ONE*. This means that a write request is only considered done when at least one server returns success in writing the entry to its commit log. For the reads level *ONE* means that consulting only one replica node is sufficient to return the client request.

3) Data Partitioning: Cassandra offers two main partitioning strategies: *RandomPartitioner* and *ByteOrderedPartitioner* [19]. The former is the default strategy and it is recommended for use in most cases. A hashing algorithm is used to create an *md5* hash value of row key. Each Cassandra node has a token value that specifies the range of keys for which they are responsible. A row is stationed in the cluster based on the hash and range tokens. Distributing the records evenly throughout the cluster balances the load by spreading out client requests. The *ByteOrderedPartitioner* simply orders *rows* lexically by keys, so it may not distribute data evenly.

4) Read and Write: A client can contact any Cassandra node for any operation. The node being connected to serves as a *coordinator*. The *coordinator* forwards the client request to the replica node(s) owning the data being claimed.

For each write request, first a commit log entry is created. Then, the mutated columns are written to an in-memory structure called *Memtable*. A *Memtable*, upon reaching its size limit, is committed to disk as a new *SSTable*. This operation is executed as a background process. An *SSTable* is an immutable structure, i.e. any updates to a *column* committed to a *SSTable* are not reflected in the existing *SSTable* but are eventually committed to a new one. A write request is sent to all replica nodes, but the consistency level determines the number of nodes whose responses are needed before the transaction can be considered complete.

For a read request, the *coordinator* contacts the replica nodes specified by the consistency level. This is a digest call to get an *md5* hash of all column names, values and timestamps.

If replicas are inconsistent the out-of-date replicas are auto-repaired in the background. In case of inconsistent replicas, a full data request is sent out and after comparing timestamps the most recent version is forwarded to the client.

Cassandra is optimized for large volumes of writes as each write request is treated like an in-memory operation. The I/O is executed as a background process. For reads, first the versions of the record are collected from all *Memtables* and *SSTables*. Later consistency checks and read repair calls are performed. Keeping the consistency level low makes read operations faster as fewer replicas are checked before returning the call. However, read repair calls to each replica still happen in the background. Thus, a record with more replicas means more read repair calls. In **Section** III-A, we quantify the performance of writes and the penalty for the I/O prone reads.

5) Hadoop Support: Cassandra and Hadoop integration is important as it facilitates data management and real-time analysis along with complex data intensive processing. In a co-located Hadoop-Cassandra cluster, Cassandra servers are overlapped with Hadoop *TaskTrackers* and *DataNodes*. This is to ensure data locality; each *TaskTracker* processes the data that is stored in the local Cassandra node. The *DataNodes* are required because Hadoop needs HDFS for copying the dependency jars, static and intermediary data.

III. Performance Results

In this section we evaluate the performance of Cassandra and Hadoop under different scenarios, including the YCSB Benchmark's Workload C.

The following experiments were performed on the Grid and Cloud Computing Research Lab Cluster at Binghamton University.

- 8 Nodes in a cluster, each of which has two 2.6Ghz Intel Xeon CPUs, 8 GB of RAM, 8 cores, and run a 64-bit version of Linux 2.6.15.

A. Cassandra Preliminary Studies

We focus on data reads and writes since the interaction of the MapReduce model with the data is limited to these two. The model reads the input data and writes the output data as a new set of records. The input data is not modified.

1) Data Locality: We are interested in data locality experiments as it is a core facet of the MapReduce model. Typically, data locality is accomplished by placing the input splits on the local disks of computing nodes. However, when using a distributed database a record might be sitting on any node in the cluster. In order to force data locality, the processing tasks of a compute node should be chosen based on the range of records stored in the local database instance.

We examine the performance for reading and writing records locally versus remotely on a single Cassandra server. **Figure** 2 shows processing times for increasing read and write sizes with a single Cassandra server and single YCSB client. As explained in **Section** II-C, Cassandra is optimized for writes and the superiority over reads is observed both in local and remote cases. While writing 0.25 million records

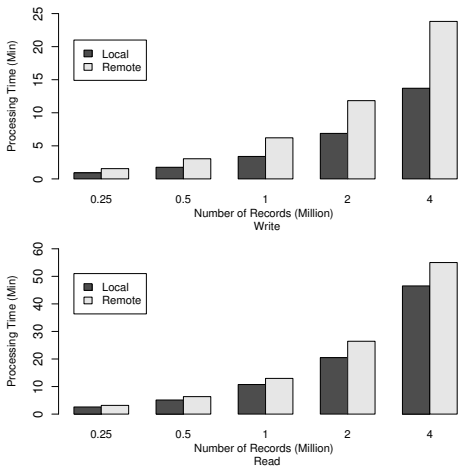


Fig. 2. Reading/writing data with a YCSB client from/to a local versus remote Cassandra instance. Writing 0.25 million records locally is 1.6 times faster while writing 4 million is 1.8 times faster. Reading data locally is up to 1.3 times faster with the increasing data size.

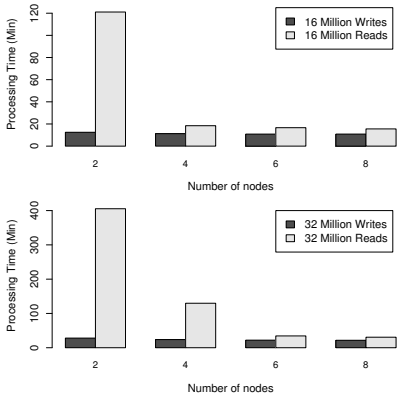


Fig. 3. Scaling up the cluster size from 2 to 8 improves the performance 1.3 times for writing 32 million records. Reading is improved to be 13 times faster at the same scale.

locally is 1.6 times faster, as the data size reaches 4 million it becomes 1.8 times faster. The reads on the other hand are up to 1.3 times faster when done locally. The performance gap grows with increasing data sizes. In the case of MapReduce implementations, multiple mappers run simultaneously and lack of data locality results in constant exchange of data between nodes. In such cases, the small performance loss observed here with a single node will be compounded for each worker. This in turn will cause the overall performance to suffer.

2) Scaling up the Cluster: In **Figure** 3, we show the effect of scaling up the Cassandra cluster size for reads and writes as we keep the data size constant. We use 16 and 32 million records and show results in separate sections of the same graph. The number of servers is varied from 2 to 8 for both data sizes and a single YCSB client with 32 threads is used to create the workload. **Figure** 3 shows that the reads are more sensitive to the cluster size than writes. While reading 32 million records is 13 times faster in an 8 node cluster than a 2 node one, writes get 1.3 times faster for the same records. This performance for writes is due to the in-memory optimizations described in **Section** II-C.

We use two different data volumes and show how changing

cluster sizes affects each of them. For 16 million records, 2 servers are expectedly slower than 4, 6 and 8. The same behavior is also observed for 32 million records. Although, in this case there is a more obvious variation from 4 to 6 servers. Moreover, an 8 node cluster provides 1.2 times better performance than a cluster of 6 nodes. Cassandra has a peer-to-peer design which makes it easy to add new servers to the cluster. This feature is especially beneficial for expeditiously growing data sets.

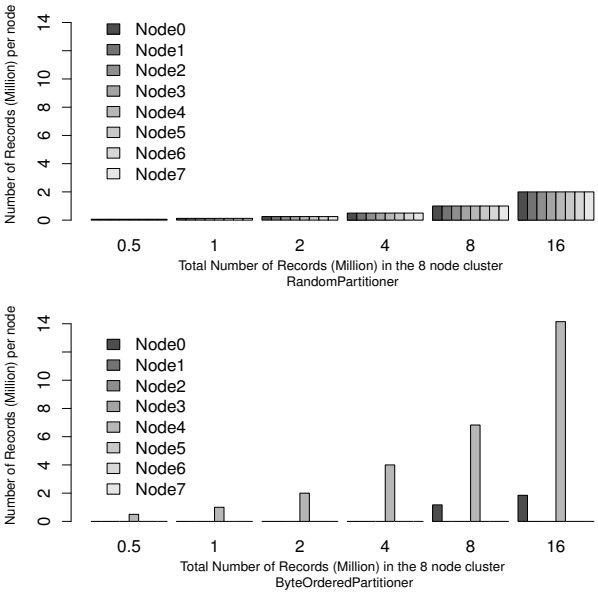


Fig. 4. We use 6 different record sizes (0.5, 1, 2, 4, 8, 16 million) and show how the two partitioning strategies distribute the data over an 8-node Cassandra cluster. While *RandomPartitioner* in the top graph evenly distributes the data, while in the bottom graph the *ByteOrderedPartitioner* places the data in one or two nodes.

3) Data Partitioning: The distribution of data in the cluster plays a crucial role in a MapReduce implementation, as the paradigm favors local processing. The idea is to make each worker read the input from the local Cassandra instance. The even distribution of data not only contributes to load balancing but also minimizes data movement. **Section** II-C explains that Cassandra offers two distinct partitioning strategies, each regulating the placement of data differently. **Figure** 4 compares the distribution of data over an 8-node cluster under these contrasting strategies. Data is added to the cluster using the YCSB Workload C in `load` mode. This graph demonstrates that with the *ByteOrderedPartitioner*, the data is concentrated in a small set of nodes while the *RandomPartitioner* spreads it almost evenly. Cassandra recommends *RandomPartitioner* at almost all times as it provides a load balanced approach.

MapReduce implementations read input datasets in parallel, process them in parallel, and then write the resulting output set. Hence, the framework interacts with the underlying storage only to read and write. **Figure** 5, using YCSB benchmarks, compares read and write times of a dataset partitioned differently on an 8-node Cassandra cluster. We read and write 0.5 to 16 million records and compare times for each. **Figure** 5 shows that 0.5 million writes are 1.2 times faster under *RandomPartitioner* and this ratio becomes 1.7 as the data size

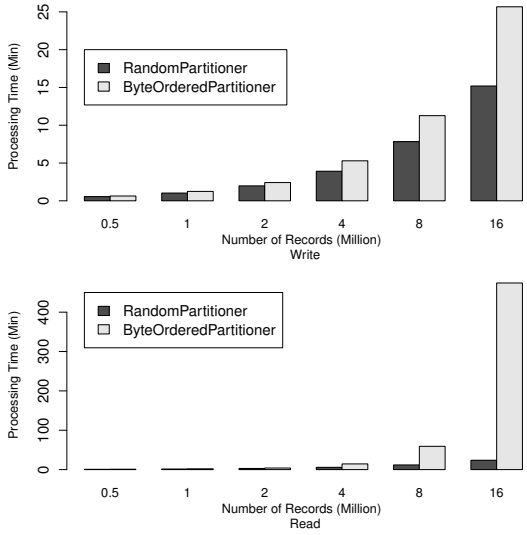


Fig. 5. The effect of the partitioning strategy on read/write performance. With *ByteOrderedPartitioner*, reading 8 million records is 5 times slower and as the data is doubled to 16 million, the performance drops significantly to be almost twenty times inferior.

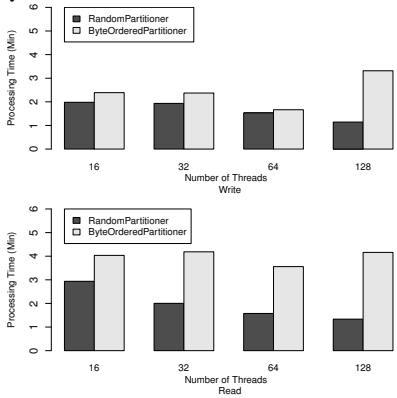


Fig. 6. Showing how each partitioning strategy responds to increasing concurrent client requests. RandomPartitioner is significantly better as the load is distributed evenly throughout the cluster.

is increased to 16 million. Reading 0.5 million is 1.4 times faster with *RandomPartitioner*, while it is 5 times faster for 8 million and almost 20 times faster for 16 million reads.

In **Figure** 6, we study how the partitioning strategy affects the performance in case of increasingly parallel operations. This is significant as we are interested in studying Cassandra when used with the highly parallel MapReduce programming model. We use YCSB Workload C with a range of 16 to 128 YCSB client threads, to read/write 2 million records from/to an 8-node Cassandra cluster. **Figure** 6 shows that the *RandomPartitioner* responds to 128 parallel write requests almost 3 times faster than *ByteOrderedPartitioner*. With the former, performance improves as more threads are added to complete 2 million writes while the latter becomes almost 2 times slower from 64 to 128 threads. Moreover, 16 client read requests under *RandomPartitioner* is 1.3 times faster and the turnaround time keeps improving as more client threads join. The *ByteOrderedPartitioner* causes the data to be accumulated in a small set of nodes. Therefore, increasing the number of parallel read requests cannot be split by all nodes. This leads to slower performance as data-carrying nodes are

overwhelmed due to lack of load balancing. Consequently, under *RandomPartitioner* 128 threads are able to finish the read intensive job 3.2 times faster than they would under *ByteOrderedPartitioner*. This shows that random partitioning ensures load balancing such that operations can be completed by the participating servers collectively.

B. Cassandra and Hadoop

In the following tests we evaluate the combination of Hadoop MapReduce and Cassandra by running Hadoop with 3 different configurations:

- *Hadoop-native*, HDFS for input and output placement.
- *Hadoop-Cassandra-FS*, reads the input from Cassandra and writes the output to the file system shared by the workers.
- *Hadoop-Cassandra-Cassandra*, reads input from Cassandra and writes output back to Cassandra.

Figure 7 shows the performance for 3 different Hadoop setups under 3 different workloads. These workloads are classified in 3 different cases based on the input size to output size ratio. They have been used by Fadika et al. in [18] and are adopted here to examine Hadoop's performance when used with Cassandra.

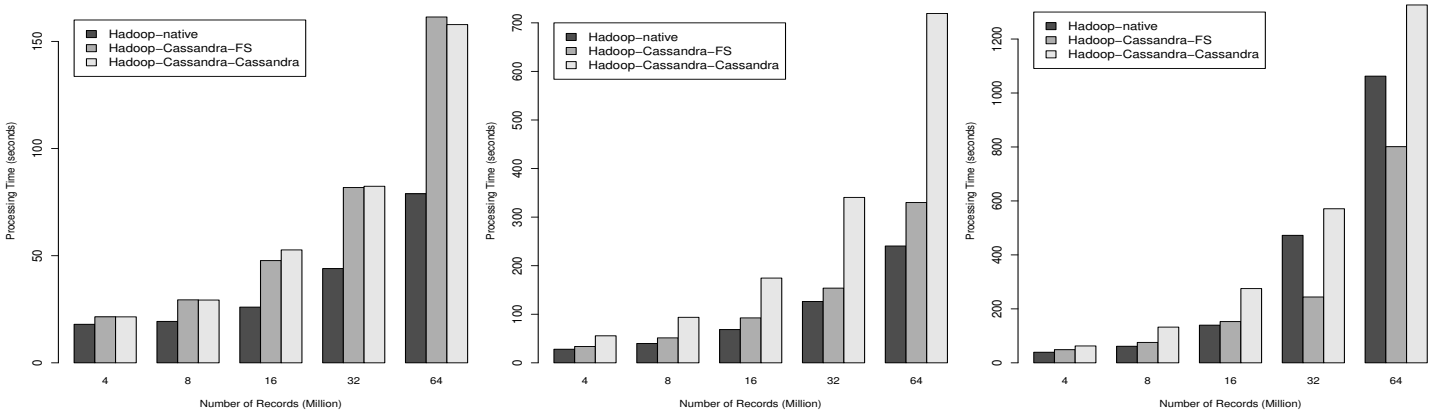
Figure 7a shows the case where the input data set is significantly larger than the output. Examples of such data can be found in applications searching for data patterns, like "Filtering" satellite image data by removing undesired areas to create high value images. The image data can be collected in a Cassandra cluster to provide search and query capabilities. In **Figure** 7a, we show that while *Hadoop-Cassandra-Cassandra* is 1.1 times slower than *Hadoop-native* at processing 4 million input records, it is 1.8 times slower for 64 million. Increasing the input 16 times leads to a slowdown by a factor of 0.7. As we show in the previous graphs, this performance loss can be compensated by adding more nodes to the cluster. *Hadoop-Cassandra-Cassandra* and *Hadoop-Cassandra-FS* display very similar times as the output is so small, the write location does not have a considerable effect on performance. **Figure** 8a shows the times spent on read and write phases for 3 of the data points in **Figure** 7a. As stated previously, the output dataset is considerably smaller than the input (over ten thousand times smaller). Each of the setups completes output writing in under 1.2 seconds. Note that reads are a big percentage of the job time. While Hadoop reads 4 million records from HDFS 1.3 times faster than it does from Cassandra, it is 2.1 times faster for 64 million. This shows that increasing the read size dramatically does not effect *Hadoop-Cassandra* performance to a great extent, which is desirable for processing expeditiously growing datasets.

In **Figure** 7b, we use a workload where the output size is very close to, if not the same as, the input size. An example of such workload is shuffling a gene sequence dataset to provide various gene combinations. **Figure** 7b shows that *Hadoop-native* is 1.2 to 1.4 times faster than *Hadoop-Cassandra-FS* and 1.9 to 2.9 times from *Hadoop-Cassandra-Cassandra*. Moreover, unlike **Figure** 8a, the location of writes leads to considerable variance in execution times between *Hadoop-Cassandra-FS* and *Hadoop-Cassandra-Cassandra* as the former is up to 2 times faster with the increasing input size. **Figure** 8b helps us to understand how much time each spends in reads and writes explicitly. The input sizes used are the same as in **Figure** 7a and **Figure** 8a therefore the same read numbers and trends are observed. Writes, however, deviate significantly. Hadoop write performance to HDFS and to the file system are very similar for all the data points shown; writing 4 million records to HDFS is almost 7 times faster than Cassandra. However, for 64 million records, this ratio drops to 4.2. This can be explained by HDFS' poor performance under heavy writes [18] while Cassandra is optimized for write operations [23].

The workload in **Figure** 7c shows the output data set to be larger than the input. An example of this is adding new properties and values to each of the input records to generate an output set, which is later to be used as input for data analysis. Fadika et al. in [18] calls this sort of applications as 'Merge" and explain it with linking real world seismic events with regional ground characteristics. **Figure** 7c displays some interesting results; after 16 million input records *Hadoop-Cassandra-FS* becomes faster than *Hadoop-native*. As for *Hadoop-Cassandra-Cassandra*, *Hadoop-native* is almost 2 times faster for 4 million records while the ratio drops to 1.2 as input data grows to 64 million.

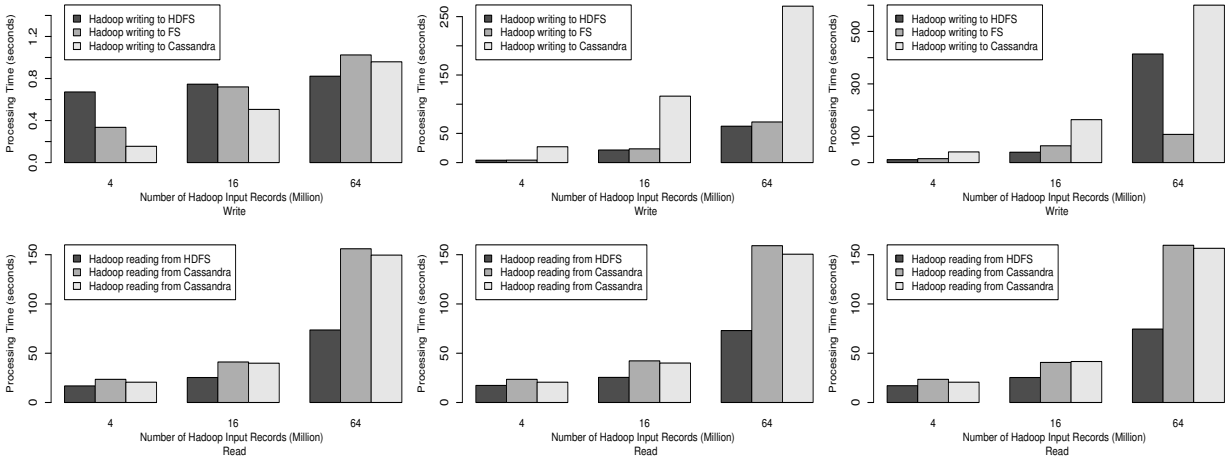
In order to understand the details we will, once again, refer to **Figure** 8. **Figure** 8c shows the read and write times explicitly corresponding to for 4, 16 and 64 million records in **Figure** 7c. Since the input size is similar to the previous figures, the reading trends observed in **Figure** 8a apply. Writes drive the performance difference. At 4 million input records, Hadoop writes to the HDFS 1.3 and 3.6 times faster than it does to the file system and Cassandra respectively. However, with the increasing data size, HDFS writes are only 1.5 times faster than Cassandra writes while 3.8 times slower than writing to the file system. Fadika et al. in [18] show that Hadoop does not perform well under write intensive operations as HDFS shows poor performance under such demands. Therefore, Hadoop demonstrates better performance as the writes are directed to the file system rather than HDFS. In **Section** II-C we explain that Cassandra is a write optimized database system, and show in **Figures** 2 and 5 that writing to Cassandra is consistently and considerably faster than reading. These optimizations help *Hadoop-Cassandra-Cassandra* to close up the performance gap with *Hadoop-native* under write intensive arrangements.

In the MapReduce model, computation is moved to the data, which means each worker node processes the data split they own. In Hadoop, data is split and distributed evenly among the *DataNodes* and each *TaskTracker* processes the data from the local *DataNode*. Using Cassandra with Hadoop leaves the distribution of data to the partitioning strategy set for the input and output *column families* within Cassandra. As explained previously, *RandomPartitioner* evenly splits the data among the nodes. This means that with *RandomPartitioner*



(a) Read $>>$ Write (b) Read = Write (c) Read $<<$ Write

Fig. 7. Hadoop with and without Cassandra under 3 different workloads. In **(a)** number of input records is over a thousand times greater than number of output records. In **(b)**, input and output records are almost equal both in size and count; and in **(c)**, number of output records is equal to the number of input records but more than ten times greater in size.



(a) Read $>>$ Write (b) Read = Write (c) Read $<<$ Write

Fig. 8. Extracting the times spent on read and write phases for each of the 4, 16 and 64 million runs shown in **Figure** 7. Input size being the same, the read graphs for all workloads are very similar. The changing write size is leading the difference in performance ratios. In **(a)**, the output size is very small and each of the writes complete in under 1.2 seconds. In **(b)** is a more write intensive workload and with growing data sizes, Hadoop writes to HDFS 4.2 times faster than it does to the Cassandra and 1.1 times faster than to the file system. In **(c)**, the output is approximately ten times greater than the input. Hadoop completes these write heavy operations to the file system up to almost 4 times faster than it does to HDFS. Under the same load, writing to Cassandra is only 1.4 times slower than HDFS.

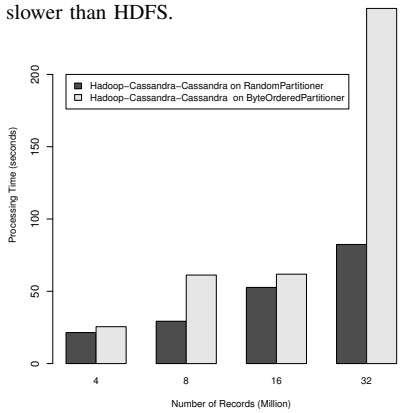


Fig. 9. Running *Hadoop-Cassandra-Cassandra* with *RandomPartitioner* is 1.2 times faster for 4 million input records but the gap increases with growing data and *ByteOrderedPartitioner* is 3 times slower for 32 million records.

each Hadoop *worker* has local access to almost the same amount of data. On the other hand, the *ByteOrderedPartitioner* causes the data splits to be collected on a small set of nodes (see **Figure** 4). Hence, the data locality is restricted to a small set of nodes and the remaining workers have to pick up splits remotely. This causes extraneous data movement in the cluster and in turn affects the turnaround time negatively. **Figure** 9 compares processing from 4 to 32 million input records with *Hadoop-Cassandra-Cassandra* using *RandomPartitioner* versus *ByteOrderedPartitioner*. The figure shows that using *RandomPartitioner* is 1.2 times faster for 4 million records and 3 times for 32 million. The performance also changes dramatically with growth of the data size, as more data movement is required to complete the MapReduce job. While **Figures** 5 and 6 display the poor read/write performance of Cassandra under *ByteOrderedPartitioner*, **Figure** 2 reveals the cost of losing data locality on a single node. These results prove that for efficient MapReduce performance, it is crucial for the underlying data storage to distribute data evenly and allow for data locality.

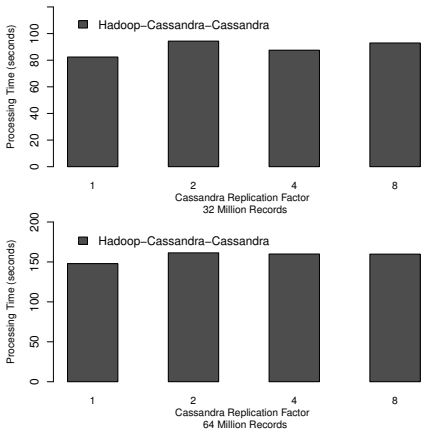


Fig. 10. Increasing Cassandra replication factor does not affect Hadoop performance negatively.

Distributed storage systems like HDFS store data in multiple nodes to avoid data loss in case of node failures. The data is replicated by a user set `replication-factor` and each replica is placed on a different node. When using Cassandra with Hadoop, HDFS does not have any control over data replication. Therefore, if the data is not replicated through Cassandra itself, node failures would result in data loss and incomplete jobs. In **Figure** 10, we show *Hadoop-Cassandra-Cassandra* performance under different replication factors for 32 and 64 million records. We observe that increasing replication factor up to 8 times only leads up to 1.1 times slower performance. This is because Hadoop reads the input from Cassandra using range scans and the overhead of read repair calls to each replica is not performed for range reads. Therefore Cassandra's replication factor can be increased without any concern for affecting the MapReduce application turnaround time. This is significant as MapReduce relies on commodity machines and node failures are always possible.

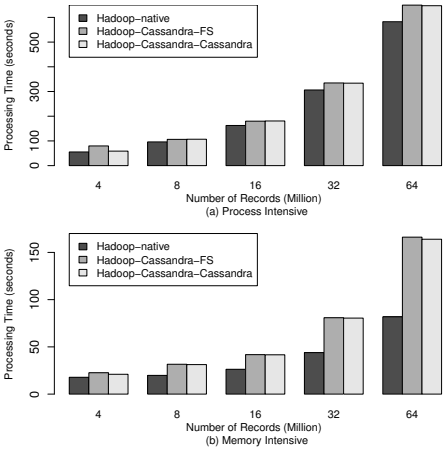


Fig. 11. Running Hadoop with different setups for processing intensive and memory intensive operations.

Figure 11 shows performance of CPU and memory intensive jobs with Hadoop and Cassandra. Apart from being data intensive, the application memory and CPU demands are also shown to affect performance in various MapReduce implementations [13]. In most MapReduce applications it is observed that the output is significantly smaller than the input; applications in 11(a) and (b) share this characteristic. The output being small makes the write time negligible and consequently *Hadoop-Cassandra-Cassandra* and *Hadoop-Cassandra-FS* display similar times. In **Figure** 11(a) *Hadoop-Cassandra-Cassandra* under CPU intensive workload shows closer times to *Hadoop-native*. For each data point shown here *Hadoop-native* is only up to 1.1 times faster. This implies that the processing of map/reduce operations takes up most of the total job time and input/output locations changes only a little.

In **Figure** 11(b) *Hadoop-native* performs 1.2 times faster than *Hadoop-Cassandra-Cassandra* for 4 million input records and this ratio reaches up to 2 as data increases to 64 million. In a *Hadoop-Cassandra-Cassandra* setup, each worker node runs a Cassandra server in addition to a *TaskTracker* and *DataNode*. Cassandra servers have their own memory load on the worker machines because there are in-memory structures like *Memtables* and other cached data. From 4 to 64 million input records we do not observe a considerable difference in the performance ratios. For 4 million input records *Hadoop-native* is 1.2 times faster, and for 64 million it is 2 times faster than *Hadoop-Cassandra-Cassandra*. Cassandra's memory footprint is more affected by the number of *column families* than the data size in the family. In all tests shown here, the Cassandra servers host only one column family which allows more memory for caching. Although the caching improves the read times, running an application with large memory demands adds up to the memory usage of the worker node and contributes to the overall slower performance.

IV. Related Work

There are various studies on benchmarking and enhancing the MapReduce paradigm [13]–[18], [20]–[22] and NoSQL technologies separately. Rabl et al. with Application Management Systems(APM) [26] analyze new distributed storage systems such as Cassandra and HBase. They describe tools used for monitoring response time of services, failure rates and resource utilization in enterprise systems. Cattell et al. [9] examine various traditional RDBMS and new generation NoSQL databases. Some of the features they study include data models, consistency and storage mechanisms, availability and query support. Padhy et al. [25] explore and compare data model and architectural decisions made for some popular NoSQL technologies like Hbase, Cassandra, MongoDB, and BigTable [10].

MapReduce and NoSQL technologies have been largely used together in the industry for web technologies like social media, online gaming and e-commerce. DataStax [4] has introduced DataStax Enterprise [5] which is a "Big Data" platform built on top of Cassandra and includes support for Apache Hadoop and side products like Hive [32] and Pig [24]. Sumbaly et al. [29] presents read only extensions to Project Voldemort [7] of LinkedIn making it more suitable for batch computing with Hadoop. Silbertein et al. [28] provides techniques to improve bulk insertions in distributed databases like PNUTS [11] which in turn helps the performance with batched workloads.

Taylor et al. [31] provide an overview on the use of Hadoop and HBase in Bio-informatics. They offer strong computation capabilities supported by MapReduce and data management that can efficiently collect various data resources in a distributed manner through HBase. Sun et al. [30] use Hadoop and HBase for fast growing RDF data set processing and storage. RDF triples are indexed and stored in Hbase and MapReduce is used for pattern processing. Ball et al. [8] with Data Aggregation System (DAS) present a single interface to collect and query relational and non-relational data from different sources. DAS uses MongoDB to provide a caching layer for data services along with storing the server logs, analytics data and key mapping.

V. Conclusion

In this paper we show that:

- Cassandra's *RandomPartitioner* distributes data evenly, improving Hadoop's performance by a factor of 3.
- Increasing the `replication-factor` on Cassandra does not affect Hadoop turn around time; leveraging range scans reduces read repair calls on replicas, immunizing Hadoop from replication related performance degradation.
- CPU intensive loads perform better using *Hadoop-native*, but the difference using Cassandra is minimal.
- Memory intensive loads perform better, by a factor of two, using *Hadoop-native*.
- Write-heavy loads favor *Hadoop-Cassandra-FS* by a large margin, but *Hadoop-Cassandra-Cassandra* is comparable to *Hadoop-native*.
- Applications that generate a small set of output data perform best using *Hadoop-native*.

References

[1] Amazon dynamodb. http://aws.amazon.com/dynamodb/.

[2] Apache Hadoop. http://hadoop.apache.org.

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

[4] Datastax. http://www.datastax.com/.

[5] Datastax enterprise. http://www.datastax.com/products/enterprise.

[6] Mongodb. http://www.mongodb.org.

[7] Project voldemort. http://www.project-voldemort.com/voldemort/.

[8] G. Ball, V. Kuznetsov, D. Evans, and S. Metson. Data aggregation system-a system for information retrieval on demand over relational and non-relational distributed data sources. In *Journal of Physics: Conference Series*, volume 331, page 042029. IOP Publishing, 2011.

[9] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[10] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[11] 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. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[12] 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.

[13] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Benchmarking MapReduce Implementations for Application Usage Scenarios. *Grid 2011: 12th IEEE/ACM International Conference on Grid Computing*, 0:1–8, 2011.

[14] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. MARIANE: MApReduce Implementation Adapted for HPC Environments. *Grid 2011: 12th IEEE/ACM International Conference on Grid Computing*, 0:1–8, 2011.

[15] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju. MARLA: MapReduce for Heterogeneous Clusters. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 49–56, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Z. Fadika and M. Govindaraju. LEMO-MR: Low Overhead and Elastic MapReduce Implementation Optimized for Memory and CPU-Intensive Applications. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:1–8, 2010.

[17] Z. Fadika and M. Govindaraju. DELMA: Dynamically ELastic MapReduce Framework for CPU-Intensive Applications. In *CCGRID*, pages 454–463, 2011.

[18] Z. Fadika, M. Govindaraju, R. Canon, and L. Ramakrishnan. Evaluating Hadoop for Data-Intensive Scientific Operations. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 67–74. IEEE, 2012.

[19] E. Hewitt. *Cassandra: the definitive guide*. O'Reilly Media, Incorporated, 2010.

[20] S. Huang, J. Huang, J. Dai, T. Xie, and B. H. 0002. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*, pages 41–51. IEEE, 2010.

[21] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom. Mrbench: A benchmark for mapreduce framework. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18, 2008.

[22] M. Kontagora and H. Gonzalez-Velez. Benchmarking a mapreduce environment on a full virtualisation platform. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '10, pages 433–438, 2010.

[23] 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.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[25] R. P. Padhy, M. R. Patra, and S. C. Satapathy. Rdbms to nosql: Reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[26] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, Aug. 2012.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, May 2010.

[28] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 765–778. ACM, 2008.

[29] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[30] J. Sun and Q. Jin. Scalable rdf store based on hbase and mapreduce. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 1, pages V1–633 –V1–636, aug. 2010.

[31] R. Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.

[32] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.