A Processing Pipeline for Cassandra Datasets Based on Hadoop Streaming

E. Dede, B. Sendir, P. Kuzlu, J. Weachock, M. Govindaraju
Grid and Cloud Computing Research Laboratory
SUNY Binghamton, New York, USA
Email:{edede1,bsendir1,pkuzlu1,jweacho1,mgovinda}@binghamton.edu

L. Ramakrishnan
Lawrence Berkeley National Lab
Berkeley, CA 94720
Email:lramakrishnan@lbl.gov

Abstract—[1]

The progressive transition in the nature of both scientific and industrial datasets has been the driving force behind the development and research interests in the NoSQL data model. Loosely structured data poses a challenge to traditional data store systems, and when working with the NoSQL model, these systems are often considered impractical and expensive. As the quantity of unstructured data grows, so does the demand for a processing pipeline that is capable of seamlessly combining the NoSQL storage model and a "Big Data" processing platform such as MapReduce. Although, MapReduce is the paradigm of choice for data-intensive computing, Java-based frameworks such as Hadoop require users to write MapReduce code in Java. Hadoop Streaming, on the other hand, allows users to define non-Java executables as map and reduce operations. Similarly, for legacy C/C++ applications and other non-Java executables, there is a need to allow NoSQL data stores access to the features of Hadoop Streaming. In this paper, we present approaches in solving the challenge of integrating NoSQL data stores with MapReduce for non-Java application scenarios, along with advantages and disadvantages of each approach. We compare Hadoop Streaming alongside our own streaming framework, MARISSA, to show performance implications of coupling NoSQL data stores like Cassandra with MapReduce frameworks that normally rely on file-system based data stores.

I. Introduction

The size of data collected on social media interactions, scientific experiments, and e-commerce is growing to very large sizes, and the nature of data has been evolving. The structure of data differs vastly as it is being collected from various sources. A similar phenomenon has arisen in the scientific community, where data coming from a single experiment may involve various sensors monitoring disparate aspects of a given test. Data relevant to an experiment may be formatted in different ways since it originates from different sources. While similar challenges existed before the advent of the NoSQL data model, earlier approaches involved storing differently structured data in separate databases, and subsequently analyzing each dataset in isolation, potentially missing a "bigger picture" or critical link between datasets. Currently, NoSQL offers a solution to this problem of data isolation by allowing datasets, sharing the same context but not the same structure or format, to be collected together. This allows the data not only to be stored in the same tables but to subsequently be analyzed collectively.

When non-uniform data grows to large sizes however, a distributed approach to analyze NoSQL data needs to be considered. MapReduce has emerged as the model of choice for processing "Big Data". While MapReduce frameworks offer both storage and processing capabilities for data in any form, structured or not, the inability on MapReduce's part to allow such data to be queried presents a significant challenge. Growing datasets not only need to be queried to enable real time information collection and sharing, but also need to undergo complex data analysis operations.

NoSQL data stores offer not only the potential of storing large, loosely structured data that can later be analyzed and mined as a whole, but also the ability for queries to be applied on such data. This is especially beneficial when real time answers are needed on only slices of the stored data. Despite the presence of this valuable querying potential from NoSQL stores, there is a need for a software pipeline allowing "Big Data" processing models such as MapReduce to tap NoSQL data stores as sources of input. Furthermore, there is a need for enabling legacy programs written in C, C++, and other non-Java executables to be used within such "Big Data" processing paradigms when data is collected under NoSQL stores.

In this paper, we present a processing pipeline allowing not only native Hadoop MapReduce programs written in Java to make use of the NoSQL storage systems, but also any non-Java executable used for data processing. We use Apache Cassandra in our analysis, a well-known NoSQL solution, and combine it with both Apache Hadoop and a streaming MapReduce framework we have developed, named *MARISSA* [10]. We show on a case-by-case basis when it is beneficial to process the data directly from NoSQL stores and the performance impact of first downloading it to a MapReduce framework.

The contributions of this paper are as follows:

- We introduce a MapReduce streaming pipeline for running non-Java executables with a MapReduce framework over datasets downloaded from Cassandra and compare the performance at different stages of this pipeline for two different MapReduce streaming frameworks, *MARISSA* and *Hadoop Streaming*.
- We compare and analyze the performance implications of processing data directly from Cassandra servers versus using the streaming pipeline for processing data columns

[1]This manuscript has been co-authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. Additional support was provided by NSF grant CNS-0958501.

of interest under various application scenarios.

- We provide recommendations on when it is beneficial to off-load data to the file system for MapReduce processing versus each worker reading the input records from the local running database servers.

II. Background

Cassandra [16] is an open source, non-relational, column-oriented, distributed database developed by Facebook. It is designed to store large datasets over a set of commodity machines clustered in a peer-to-peer structure to promote horizontal scalability. Interesting aspects of the Cassandra framework include independence from any additional file systems like HDFS, scalability, replication support for fault tolerance, balanced data partitioning, and MapReduce support with a Hadoop plug-in.

MapReduce: Taking inspiration from functional programming, MapReduce starts with the idea of splitting an input dataset over a set of commodity machines, called `workers`, and processing these data splits in parallel with user-defined `map` and `reduce` functions. The model abstracts the details of input distribution, parallelization, scheduling and fault tolerance.

Apache Hadoop [1], the leading open source MapReduce implementation, relies on two fundamental components: the Hadoop Distributed File System (HDFS) [19] and the Hadoop MapReduce Framework for data management and job execution respectively. Hadoop is implemented in Java and requires the `map` and `reduce` operations also to be implemented in Java. This creates a challenge for legacy applications where it may be impractical to rewrite the applications in Java. *Hadoop Streaming* is designed to address this need by allowing users to create MapReduce jobs where any executable (written in any language or script) can be specified to be *map* or *reduce* operations. Although *Hadoop Streaming* has a restricted model [10], [14], it is commonly used to run numerous scientific applications from various disciplines.

MARISSA leaves the input management to the underlying shared file system to solely focus on processing. In previous work [10], we explain the details of *MARISSA* and provide a comparison to *Hadoop Streaming* under various application requirements. Unlike *Hadoop Streaming*, *MARISSA* does not require processes like `TaskTrackers` and `DataNodes` for execution of MapReduce jobs. Once the input is split by the `master` with the `Splitter` module and placed into the shared file system, each `worker` has access to the input chunks awaiting execution. Unlike Hadoop, *MARISSA* does not force explicit data locality – rather it leaves such optimizations to the shared file system. Each `worker` points the target executables to the input splits they are responsible for, monitors the status of the local job, and informs the `master` when the local tasks are all completed. Compatibility with POSIX file-systems, the ability to run applications not using standard input and output, and the ease of iteration support are some of the features implemented within *MARISSA* that are often considered lacking in *Hadoop Streaming*.

III. MapReduce Streaming over Cassandra

A. MapReduce Streaming Pipeline For Cassandra Datasets

In this paper, we introduce a MapReduce pipeline that can be adopted by MapReduce frameworks like *Hadoop Streaming* and *MARISSA*, each offering MapReduce capabilities with non-Java executables. This pipeline, shown in **Figure** 1, has three main stages: *Data Preparation*, *Data Transformation (MR1)* and *Data Processing (MR2)*. Each of these stages is explained in the following subsections and performance implications are discussed in **Section** IV.

1) Data Preparation: Data Preparation, **Figure** 1**a**, is the step of downloading the data from Cassandra servers to the corresponding file systems – HDFS for *Hadoop Streaming* and the shared file system for *MARISSA*. For both of these frameworks this step is initiated in parallel. Cassandra allows exporting the records of a dataset in JSON formatted files [3]. Using this feature, each node downloads the data from the local Cassandra server to the file system. In our experimental setup, each node that is running a Cassandra server is also a worker node for the MapReduce framework in use.

We implemented a set of tools to launch this process of exporting data from a Cassandra cluster. Every worker, connects to its local database server and starts the export operations. Each worker collects the exported records in unique files on the shared file system. In *MARISSA*, we introduced these tools into the `Splitter` module. For *Hadoop Streaming* however, we implemented additional ones to initiate the data preparation process on all the workers. Next, this data is placed into the HDFS executing the `put` command from the Hadoop master. In Hadoop's case, the `put` includes creating input splits and placing them into HDFS. In MARISSA, however, the worker nodes flush the data to the shared file system and later these data files are split by the master one-by-one for each core. We compare the performance of the *Data Preparation* stage for *Hadoop Streaming* and *MARISSA* in **Section** IV-A.

2) Data Transformation (MR1): Cassandra allows users to export datasets as JSON formatted files. As our assumption is that the MapReduce applications to be run are legacy applications which are either impossible or impractical to be modified and the input data needs to be converted into a format that is expected by these target executables. For this reason, our software pipeline includes a MapReduce stage, **Figure** 1**b**, where JSON data can be transformed into other formats. In this phase each input record is processed to be converted to another format and stored in intermediary output files. This step does not involve any data or processing dependencies between nodes and therefore is a great fit for the MapReduce model.

We implemented Python scripts for this stage which can be utilized both by *MARISSA* and *Hadoop Streaming* without any modifications. As this is the first of a series of MapReduce operations whose output will be used as the input by the ensuing MapReduce streaming jobs, we call this stage *MR1*. Our system not only allows users to convert the dataset into the desired format but also makes it possible to specify the

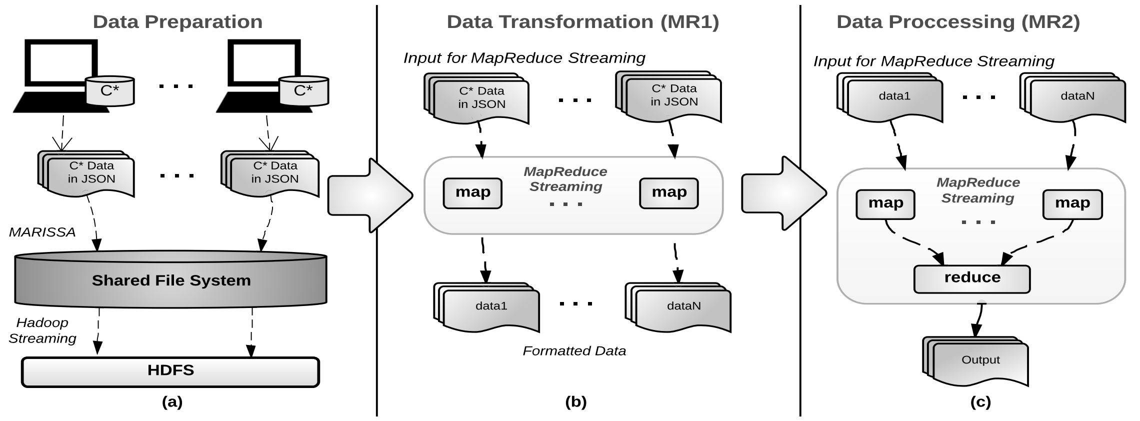


Fig. 1. MapReduce streaming pipeline. In **(a)**, *Data Preparation*, each worker node exports the dataset from the local Cassandra servers to the shared file system and to the HDFS. In **(b)**, the dataset is converted into the user specified format using MapReduce. In **(c)**, user set non-Java executables are used within MapReduce to process the reformatted data produced in **(b)**.

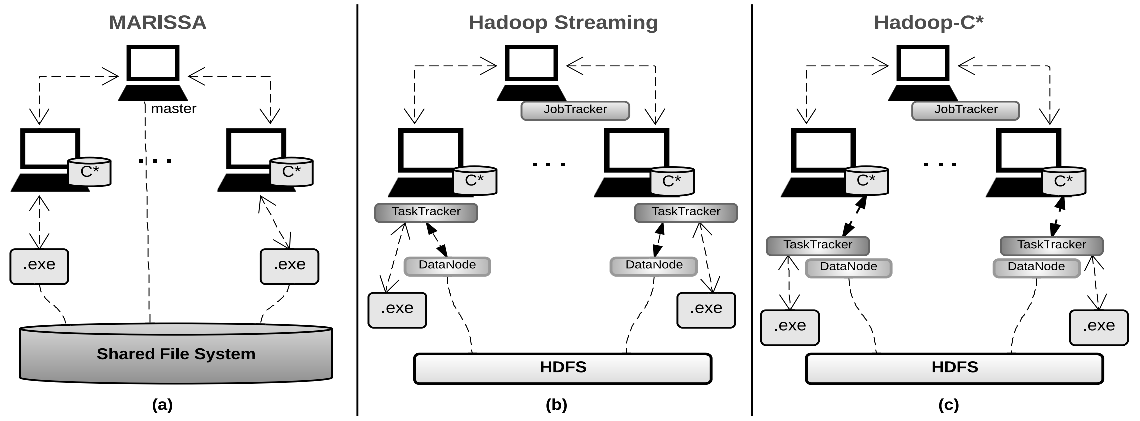


Fig. 2. **(a)** shows using *MARISSA* for the MapReduce streaming pipeline pictured in **Figure** 1. The data is first downloaded from the database servers to the shared file system, pre-processed for the target application and at the final stage processed with the user set non-Java executables. **Figure (b)**, is the layout of using *Hadoop Streaming* in such a setting where the dataset is also pushed into the HDFS. **Figure (c)** shows the structure of *Hadoop-C**, which we use to process Cassandra data directly from the local database servers using Hadoop and non-Java executables.

columns of interest. This is especially useful when only a vertical subset of the input data is sufficient for the final processing. This stage helps to reduce data size, and in turn positively affects the performance of the next MapReduce stage by leading to fewer I/O and data parsing operations. In the following sections of this paper we will refer to this stage either as *MR1* or as *Data Transformation*. **Section** IV-B provides a comparison between the performance of *Data Transformation* using *MARISSA* and *Hadoop Streaming*.

3) Data Processing (MR2): This is the final step of the MapReduce Streaming pipeline shown in **Figure** 1. In **Figure** 1**c** we run the non-Java executables, over the output of *MR1*, since the data, now, is in a format that can be processed by these target applications. We can use either *MARISSA* or *Hadoop Streaming* in this stage. Since this is the second MapReduce stage in our pipeline, we name it *MR2*. That is, any MapReduce streaming job run after *MR1* is regarded as *MR2*. In **Section** IV-C, we first compare the performance of *MARISSA* and *Hadoop Streaming* considering only this stage under various application scenarios. Later, in order to show the full operation span, we include the time taken for *Data Preparation* and *Data Transformation* under each MapReduce framework and repeat our comparisons.

B. MapReduce Streaming Pipeline with MARISSA

As explained in **Section** III-A1, the `Splitter` module of *MARISSA* has been modified such that each worker connects to the local database server to take a snapshot of the input dataset in JSON format and place it into the shared file system. After the *Data Preparation* stage shown in **Figure** 1**a**, the input is split and ready for *Data Transformation*. **Figure** 2**a** shows the architecture of *MARISSA*. It allows each non-Java executable to interact with the corresponding input splits directly without needing to mediate this interaction. In the stage of *Data Transformation*, each *MARISSA* mapper runs an executable to convert the JSON data files to the user-specified input format. These converted files are placed back into the shared file system. *MARISSA* runs the user given executables to create the next MapReduce stage, which we call *Data Processing*. This is accomplished using the previous output as the input. There is no re-distribution or re-creation of splits required since *MARISSA* is designed to allow iteration of MapReduce operations where the output of one operation is fed as input to the next.

C. MapReduce Streaming Pipeline with Hadoop Streaming

In the *Data Preparation* stage, each Hadoop worker connects to the local Cassandra server and exports the input dataset in JSON formatted files. Next, these files are placed into the HDFS using the `put` command. This distributes the input files among the `DataNodes` of the HDFS and later they become the input for the *Data Transformation* stage. HDFS is a non-POSIX compliant file system that requires the use of HDFS API to interact with the files. Since *Hadoop Streaming* uses non-Java application for `map` and/or `reduce`, these executables do not use this API and therefore do not have immediate access to the input splits. So, Hadoop *TaskTrackers* read the input from HDFS *DataNodes*, feed to the executables for processing and collect the results to write back to HDFS. In the *Data Transformation* step shown in **Figure** 1**b**, *Hadoop Streaming* uses our input conversion code to transform the input to the desired format and later *Data Processing* is performed on the output of this stage. Note that at the *Data Processing* stage the input is already in HDFS as it is the output of the previous MapReduce job.

*D. Hadoop-C**

*Hadoop-C** is the setup where a Hadoop cluster is co-located with a Cassandra cluster to provide an input source and an output placement alternative to the MapReduce operations. This setup, illustrated in **Figure** 2**c**, allows users to leave the input dataset on local Cassandra servers. We use Hadoop *TaskTrackers* to read the input records directly from the local servers to ensure data locality. That is, there is no need for taking a snapshot of the dataset and placing it into the file system for MapReduce processing. Therefore, no *Data Preparation* or *Data Transformation* steps are required.

Before proceeding to the map operations, Hadoop `mappers` start the user specified non-Java executable to be used for data processing. In each `map` operation, `mappers` read a record from the database, convert it to the expected input format and stream it to the running application using `stdin`. Later, the output is collected back from this application, using `stdout`, which is then turned into a database record and written back to the Cassandra data store.

Figure 2**c** shows that *DataNodes* are running on each worker node, but they are not used for input management. `DataNodes` are required since HDFS is used for dependency jars and other static and intermediary data. In the following sections we refer to this setup as *Hadoop-C**. Furthermore, we will use the notation *Hadoop-C*-FS* for the cases when Hadoop *TaskTrackers* read the input records directly from the local Cassandra servers, but the output is collected in the shared file system.

IV. Performance Results

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.

In the following tests we use the 3 different setups that we explain in **Section** III to perform MapReduce operations over a dataset that resides in a Cassandra distributed database cluster. Our experiments are conducted with two common data operations: `Filter` and `Reorder` [14]. We also used a memory intensive workload as "Big Data" operations might a large memory footprint.

Filter. A `Filter` case occurs when the output of data processing results in a subset of the entire data set such as when a certain pattern is being searched. `Filter` applications are common in Geographic Information Systems. In the case of spatial data processing, a large amount of aerial imaging data is indexed to provide better satellite image resolution [8].

Reorder. A `Reorder` case occurs when the input is reordered in some way resulting in an output dataset that is close to identical in size to the input, such as sorting of long and fixed gene sequences [6].

A. Data Preparation

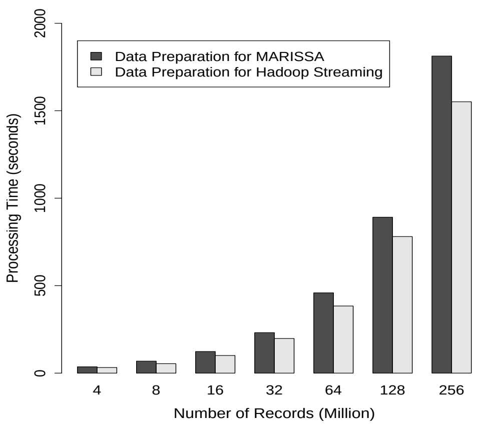


Fig. 3. The overhead of moving data from Cassandra into the file system for MapReduce processing. The cost increases with growing data sizes.

Figure 3 shows the performance for taking a snapshot of the input dataset from Cassandra into the shared file system and HDFS for processing with *MARISSA* and *Hadoop Streaming* respectively. The cost of moving data from Cassandra servers expectedly increases with growing data sizes. Moving 256 million input records takes nearly 50 times more than moving 4 million. **Figure** 3 also shows the disparity for the cost of *Data Preparation* for *Hadoop Streaming* and *MARISSA*. At four million records *Data Preparation* for *Hadoop Streaming* is 1.1 times faster than *MARISSA* and it is over 1.2 times faster at 64 and 256 million records. This performance variation of *Data Preparation* for each system can be explained with the inefficiencies, laid out in **Section** III-A1, of the *MARISSA* `Splitter` module which is responsible for creating the data splits for individual cores of the worker nodes.

B. Data Transformation (MR1)

Figure 4 shows the performance of *Data Transformation* which is the stage for converting the snapshot of the target dataset to the required format using both *MARISSA* and

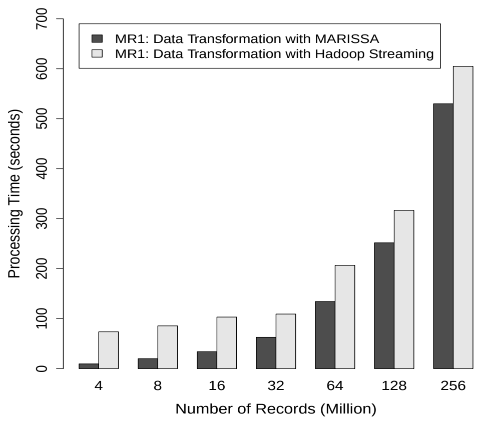


Fig. 4. Cassandra data pre-processing for MapReduce streaming applications. *Hadoop Streaming* performance get closer to *MARISSA* with increasing data sizes however at 256 million records *MARISSA* is still 14% faster.

Hadoop Streaming. This figure shows that *MARISSA* is almost eight times faster than *Hadoop Streaming* at four million input records but with growing data sizes this performance difference lessens. At 32 million records *MARISSA* performs 74 percent faster and with the larger data sizes it is faster by around 14 percent. This performance divergence between the two frameworks can be explained by the fact that Hadoop start-up cost is more visible in small data sizes. [10], [12], [13]. The *MARISSA* framework is designed as a lightweight MapReduce platform where the start-up overhead is minimal.

C. Data Processing (MR2)

In the following tests we run the target non-Java applications on the output of *MR1*. We show the performance of running various applications scenarios in stage *MR2* with *MARISSA* and *Hadoop Streaming*. In addition, we compare these two MapReduce streaming pipelines with *Hadoop-C** where there is no *Data Preparation* and/or *Data Transformation* steps necessary and the target applications in *MR2* can be run directly. We first show the performance of *MR2* exclusively with each of the setups (*MARISSA, Hadoop Streaming* and *Hadoop-C**) and later, in order to show the overall cost, with the times for *Data Preparation, MR1* and *MR2* combined.

1) MR2: Read Intensive Applications: **Figure** 5 shows the performance for running a "Filter" class of applications [14]. In this class of jobs, the input dataset is immense both in number of records and the size compared to the output set. These, observed commonly in MapReduce use cases, are not necessarily considered to be processing heavy workloads and considerable portion of the job time is spent on reading the input. As the output data size is small, the write times have a minimal effect on job performance [11]. In **Figure** 5 we show the relative performance of running such examples of non-Java executables using *MARISSA*, *Hadoop Streaming* and *Hadoop-C**. **Figure** 5a shows the job times of only the *MR2* stage for *MARISSA* and *Hadoop Streaming*. While for these two frameworks, *MR2* is run only after the steps of *Data Preparation* and *MR1*, *Hadoop-C** does not require any of these steps since the input records are directly read from the local Cassandra servers.

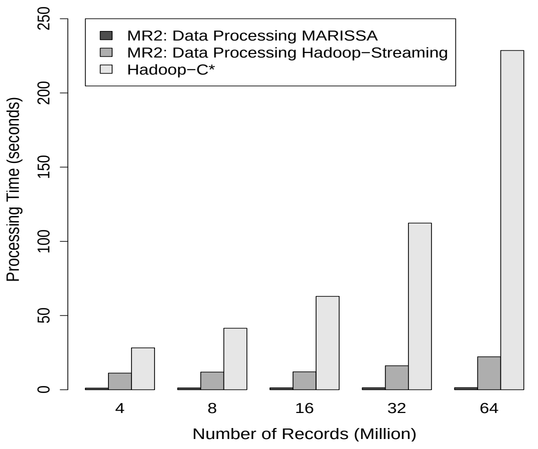
As we explain in **Sections** III-A2 and IV-B, after the *Data Transformation* step the input from the database is not only converted to the required format but also can be remarkably reduced in size. This reduction in data size causes the *MR2* input to be much less than *MR1* and in turn helps the performance in this stage, especially for *MARISSA* whose performance advantage is more visible in smaller data sizes.

In **Figure** 5a, for 64 million input records, *MARISSA* is over 100 times faster than *Hadoop-C** while *Hadoop Streaming* is over 10 times faster. Since the input size is greatly reduced, *MARISSA* takes advantage of its low-overhead MapReduce model and is up to 15 times faster than *Hadoop Streaming*. This difference in performance between the two streaming frameworks is less pronounced with growing data since the performance of Hadoop improves for such sizes [10], [12].

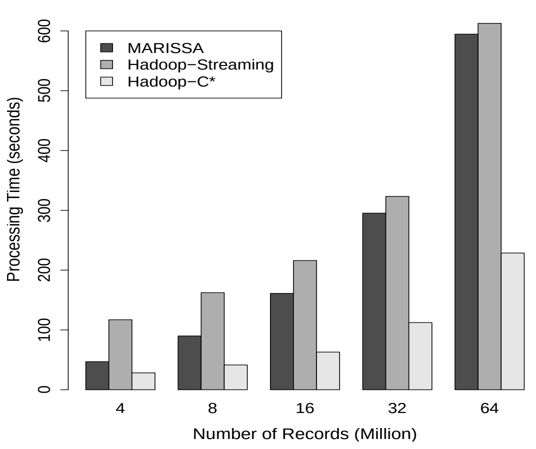
Figure 5b shows the total job time for the same application in this case including *Data Preparation* and *MR1* stages. Moving the input dataset out of the database does not just come with the cost of data movement. The exported data also needs to be explicitly split for each worker node/core in the cluster, which is the case in *MARISSA*, and to be transformed from the JSON files to a format that is expected by the applications in *MR2*. These additional steps contribute to the total job time significantly. That is, leaving the data in Cassandra servers and using *Hadoop-C** provides better performance even with the considerable overhead observed when reading the input from the Cassandra servers [11]. At four million records, *Hadoop-C** is 1.7 times faster than *MARISSA* while it is 2.6 times faster for 64 million records. The speedups are four and 2.7 compared to *Hadoop Streaming* for the same data sizes. Considering the initial stages, which display a gradually increasing cost, at four million records *MARISSA* is 2.5 times faster than *Hadoop Streaming*. However, as the data grows to 64 millions it is only 1.05 times. This can be explained with the performance inefficiencies in the *Data Preparation* stage of *MARISSA* revealed in **Figure** 3.

These two graphs show that the cost of moving data out of the database system for additional processing can easily exceed the overhead observed with database read/write operations. This leads us to the conclusion that if the target application, which is the application to be run in *MR2*, or its variations is not expected to be repeated many times over this dataset, it is better to leave the input data in the cluster and use our *Hadoop-C** approach in which the input is read by the Hadoop `mappers` directly from local Cassandra servers and passed to the designated applications. On the other hand, if the interest is on a particular snapshot of the database, which is downloaded to the file system for processing over and over again by this class of applications, we recommend downloading the data to the file system to eliminate the overhead of database reads on every single MapReduce job.

2) MR2: Write Intensive Applications: **Figure** 6 shows the performance of running "Reorder" and "Merge" type operations that are classified as write intensive workloads [11] and [14]. In this group of applications, the size of the output is either nearly equal or greater than the input dataset. The

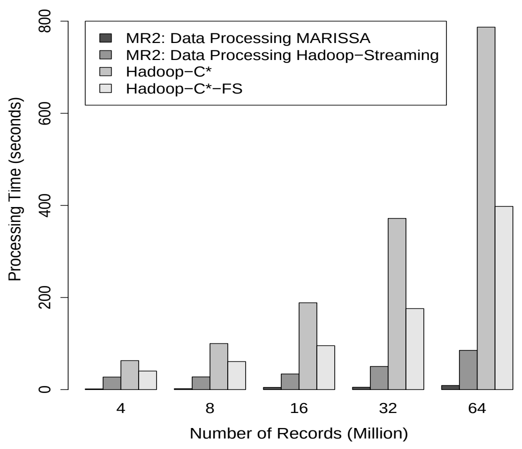


(a) Read>>Write NOT including *Data Preparation and MR1*

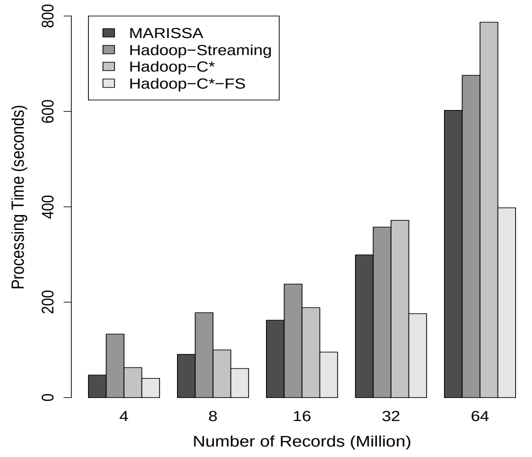


(b) Read>>Write including *Data Preparation and MR1*

Fig. 5. Read intensive workloads over Cassandra data using alternative streaming models for non-Java applications. In **(a)** *Data Preparation* and *Data Transformation(MR1)* is NOT included in the total job time while in **(b)**, the time for the two initial steps is included. *MR1* greatly reduces the input data size and this leads to a improved performance in the stage *MR2*. In both of the graphs there is no *Data Preparation* or *MR1* steps necessary for *Hadoop-C**.



(a) Read<=Write NOT including *Data Preparation and MR1*



(b) Read<=Write including *Data Preparation and MR1*

Fig. 6. Write intensive workloads over Cassandra data using alternative streaming models for non-Java applications. In this set of applications the output dataset is either equal or greater to the input. In **(a)**, extracting data from Cassandra to the file system and the data pre-processing (*MR1*) is NOT included in the total job time while in **(b)**, the time for the former two stages is included. In these graphs we also show the performance for *Hadoop-C*-FS* since the large output sizes cause a lot of write operations and where these writes are placed remarkably affects the overall MapReduce performance.

increase in the amount of write operations completed by the framework notably affects the overall performance. In previous studies [11], [14], we show that in such cases the write location becomes a strong determinant of MapReduce performance. Therefore, in **Figure** 6 we include two different setups for the Hadoop native and Cassandra consolidation. As we explained in **Section** III-D while *Hadoop-C** uses Cassandra servers for the input source and output destination, *Hadoop-C*-FS* reads the input from the local database servers but writes the output to the file system shared by the workers.

In **Figure** 6a we display the performance disparity of running such set of applications under four different setups without including the time spent in the *Data Preparation* and *MR1* phases. This figure shows that the overhead of using a Cassandra cluster for write intensive workloads comes with a cost which increases for growing datasets. At four million input records this application is nearly 40 and three times faster with *MARISSA* and *Hadoop Streaming* respectively, than *Hadoop-C**. While the former two approaches are nearly 26 and four times faster than the alternative approach, *Hadoop-C*-FS*. At 64 million records *MARISSA* is almost 88 times faster than *Hadoop-C** while *Hadoop Streaming* is nearly ten times faster. Moreover, *MARISSA* and *Hadoop Streaming* perform almost 44 and five times better than *Hadoop-C*-FS* at the same data point. While at four million input records *Hadoop-C*-FS* is 1.6 times faster than *Hadoop-C**, at 64 million it manages to complete the same job over two times faster than *Hadoop-C** by directing the write operations to the file system instead of the underlying database cluster. Comparing *MARISSA* to *Hadoop Streaming* on this graph we see that it is nearly eight times faster at four million input records and at 64 million is faster by a factor of nine.

In **Figure** 6b we include the times for *Data Preparation* and *MR1* to the total job time and show the relative performances under four different scenarios. Comparing the total times we see that while at four million input records, *MARISSA* is

1.3 times faster than *Hadoop-C**, the latter is over 2 times faster than *Hadoop Streaming*. However increasing the data size to 64 million changes this picture dramatically. At this size, *MARISSA* keeps its advantage over *Hadoop-C** staying 1.3 times faster and *Hadoop Streaming* takes the lead over *Hadoop-C** and becomes almost 1.2 times faster. Although Cassandra is write optimized [16], it still creates a performance bottleneck when used with Hadoop for write intensive workloads. This picture alters even more as we compare the total times for *MARISSA* and *Hadoop Streaming* with *Hadoop-C*-FS* which is still more efficient than the former two with increasing data sizes. At four million input records *Hadoop-C*-FS* is nearly 1.2 times and at 64 million over 1.5 times faster than *MARISSA*. On the other hand, it is roughly 3.3 times faster than *Hadoop Streaming* at four million records and over 1.6 times for 64 million. Although the noticeable Hadoop startup overhead contributes to the speedup we see at 4 million input records, the increase of the output size with the growing input causes *Hadoop-C*-FS* to stay more efficient by using the file system for output placement. **Figure** 6b presents that while at four million input records *MARISSA* is nearly 2.9 times faster than *Hadoop Streaming*, at 64 million records, it performs only 1.1 times better. We explain this with the inefficiencies in the *Data Preparation* phase of *MARISSA* where the data growth escalates the split time.

3) MR2: Memory Intensive Applications: In **Figure** 7 we show the performance of a MapReduce job for running memory intensive non-Java executables over Cassandra data. This class of applications show high memory demands and in this case the application creates large data structures in memory for each record that is processed. **Figure** 7a shows the performance of such applications only for the *MR2* stage, the time for *Data Preparation* and *Data Transformation* is not included. This graph shows that *MARISSA* is over five times faster than *Hadoop Streaming* for four million input records and as the data size increases up to 256 million, it is around 1.5 times faster. This figure also presents that *MARISSA* is more than eight times faster than *Hadoop-C** with the increasing data sizes. *Hadoop Streaming*, on the other hand, is 1.5 times faster at four million records and over five times faster at 256 million records than *Hadoop-C**. As we explain in **Section** III-D each worker is not only running *TaskTrackers* and *DataNodes* but also Cassandra servers that feed data into the MapReduce framework for processing. In such cases, worker nodes are under the memory demands of not only the *MR2* applications but also Hadoop processes and Cassandra servers using in-memory data structures like *Memtables* and other cached data [11]. In **Figure** 7b, the performance data for *Data Preparation* and *Data Transformation* are included in the total times displayed. Here, again, note that when we introduce the time of data migration from the database servers to the file system and pre-processing of that data, *Hadoop-C** proves to be a better option as at 256 million it is over two times faster than both *MARISSA* and *Hadoop Streaming*.

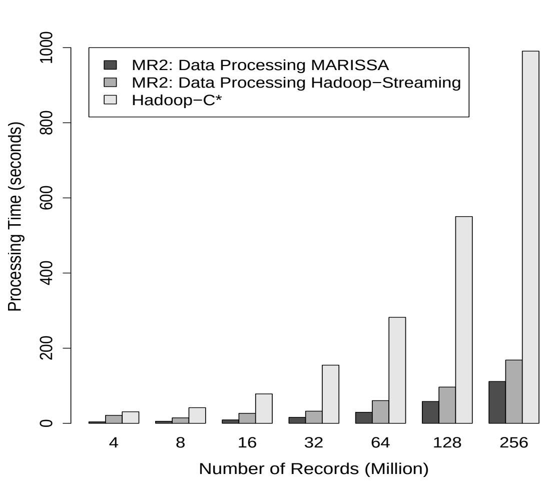
V. Summary of Experiments

Our findings can be summarized as following:

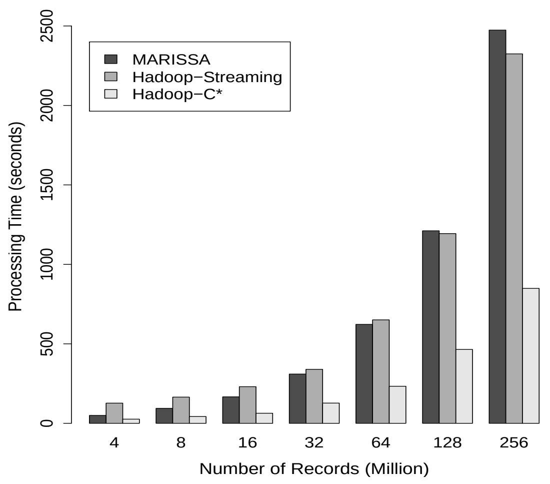
- *Data Preparation* for *Hadoop Streaming* is nearly 1.3 times faster than *MARISSA* at 256 million input records as the former creates the splits more efficiently. This performance gap is expected to grow as the number of records rises from hundreds of millions to billions.
- Based on the expected data format, the *Data Transformation* stage can lead to great reduction in data size. This reduction in data size helps the performance of stage *MR2* especially for *MARISSA* whose performance advantage is more visible in smaller data sizes.
- *Data Transformation* allows users to take a vertical subset of the input database in case processing is only needed to be performed on certain columns.
- *Data Transformation* stage with growing input is nearly 20 percent faster with *MARISSA* than *Hadoop Streaming*.
- *Data Processing* for read intensive applications with *MARISSA* is nearly 1.5 times faster than *Hadoop Streaming* up to 256 million records.
- When only the *Data Processing* stage is considered, *MARISSA* and *Hadoop Streaming* are over ten times faster than *Hadoop-C** under read intensive workloads. However, when the cost of initial steps required for the former two are added, *Hadoop-C** becomes nearly three times faster even with the considerable overhead introduced by database reads [11].
- Under write intensive workloads, *MARISSA* has a considerable advantage over *Hadoop Streaming* as HDFS struggles under heavy write load [14].

VI. Related Work

There are various examples of using NoSQL technologies with MapReduce. DataStax [4] offers a "Big Data" system built on top of Cassandra which also supports Apache Hadoop, Hive [24] and Pig [17]. Hive is an open source project [24] built on top of Hadoop to offer querying support over the datasets residing in distributed file systems like HDFS. By contributing read only extensions to the Project Voldemort [5]. Sumbaly et al. [21] aim to provide better batch computing performance when used with Hadoop. Silbertein et al. [20], on the other hand, enhance the bulk insertions of PNUTS [9] in order to improve performance with batched workloads. Sun et al. [22] show processing fast growing RDF datasets, indexed and stored in an HBase cluster, with Hadoop. Ball et al. [7] present Data Aggregation System (DAS) to collect and query relational and non-relational datasets through a single interface. DAS uses MongoDB for various caching and logging operations. Taylor et al. [23] combine Hadoop and HBase for Bioinformatics to provide a scalable data management and processing platform. They show examples of running Bioinformatics applications like BLAST [15] with *Hadoop Streaming* but do not provide a detailed study of cases when the target data is stored in the distributed database. OConnor et al. propose SeqWare Query Engine [18], to provide a querying platform for genome data. They use HBase [2] as



(a) Memory intensive NOT including *Data Preparation and MR1*



(b) Memory intensive including *Data Preparation and MR1*

Fig. 7. Running memory intensive workloads over Cassandra data using alternative streaming models for non-Java applications. In **(a)**, the time spend on downloading the data and pre-processing is not included while in **(b)** they both are to show the total job span.

a backend storage and use a web query interface to allow access to the datasets. They use the MapReduce model on such platforms and provide a scalable storage, querying and processing framework.

VII. Conclusion

In order to fully exploit ”Big Data” sets, we need a software pipeline that can effectively combine the use of data stores such as Cassandra with scalable distributed programming models such as MapReduce. In this paper we show two different approaches, one working with the distributed Cassandra cluster directly to perform MapReduce operations and the other exporting the dataset from the database servers to the file system for further processing.

References

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

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

[3] Cassandra wiki, operations. http://wiki.apache.org/cassandra/Operations.

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

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

[6] D. E. G. Abhishek Verma, Xavier Llor‘a. Scaling simple and compact genetic algorithms using mapreduce. October 2009.

[7] 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.

[8] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM 2009: Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, pages 302–319, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[10] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, D. Gunter, and R. Canon. Marissa: Mapreduce implementation for streaming science applications. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8, 2012.

[11] E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju. An Evaluation of Cassandra for Hadoop. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 494–501, Washington, DC, USA, 2013. IEEE Computer Society.

[12] 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.

[13] 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.

[14] 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.

[15] I. Korf, M. Yandell, and J. Bedell. *BLAST*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[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] 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.

[18] B. OConnor, B. Merriman, and S. Nelson. Seqware query engine: storing and searching sequence data in the cloud. *BMC bioinformatics*, 11(Suppl 12):S2, 2010.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.