Evaluating Cassandra Data-sets with Hadoop Approaches

Ruchira A. Kulkarni
Student (BE), Computer Science & Engineering Department,
Shri Sant Gadge Baba College of Engineering & Technology, Bhusawal, India

Abstract: The progressive transition in both scientific and industrial datasets has been the driving force behind the development and study interests in the NoSQL model. Loosely structured data poses a challenge to traditional data store systems, and when working with NoSQL model, these systems are often considered impractical and costly. As the quantity and quality of less structured data grows, so does the demand for a processing pipeline that is capable of seamlessly bind the NoSQL storage model and mapReduce which is “Big Data” processing platform. Although MapReduce is the exemplar of choice for data intensive computing, Java based frameworks like Hadoop requires users to write MapReduce code in Java while Hadoop Streaming module let users to define non Java executables as map and reduce operations. When challenged with legacy C/C++ applications and non Java executables, there arises a further need to permit NoSQL data stores access to the functions of Hadoop Streaming. We present approaches in solving the difficulty of integrating NoSQL data stores with MapReduce using non Java application scenarios, along with benefits and drawbacks of each approach. We compare Hadoop Streaming with our own streaming framework, MARISSA, to see performance implications of coupling NoSQL data stores like Cassandra with MapReduce structure that normally trust on file-system based data stores. this experiments also include Hadoop-C*, which is a configuration where a Hadoop cluster is Located with a Cassandra cluster in order to process data by using Hadoop with non java executables.

Keywords- *Hadoop, Cassandra, NoSQL, Pipelines, Map Reduce*

I. INTRODUCTION

With the increased amount of data collection taking place as a result of social media interaction, scientific experiments, and even e-commerce applications, the nature of data as we know it has been evolving. As a result of this data generation from various different sources “new generation” data, presents challenges as it is not all relational and lacks predefined structures. As an example, blog sections for commercial entities collect various inputs from customers about their products from Twitter, Facebook, and social media outlets. However, the structure of this data differs vastly because it is collected from varied sources. A similar phenomenon has arisen in the scientific arena, such as at NERSC where data coming from a single experiment may involve different sensors monitoring disparate aspects of a given test. In this circumstance, data relevant to that experiment as a whole will be produced, but it may be formatted in different ways since it produced from different sources.

While similar challenges existed before the advent of the NoSQL model, earlier approaches involved storing differently structured data in separate databases, and subsequently analyzing each dataset in isolation, potentially lost a “bigger picture” or important link between datasets. Currently, NoSQL provides a solution to this problem of data isolation by permitting datasets, sharing the same context but not same structure or format, to be gathered together. This allows the data not only to be stored in the same tables but to subsequently be analyzed collectively.

When unstructured data grows to huge sizes however, a distributed approach to analyze unstructured data needs to be considered. MapReduce [1] has emerged as the model of choice for processing “Big Data” problems. MapReduce frameworks such as Hadoop provide storage and processing capabilities for data in any form, structured or not. However, they do not directly provide support for querying the data. Growing datasets not only required to be queried to enable real time information collection, but also need to undergo complex batch data analysis operations to extract the best possible knowledge.

NoSQL data stores offer not the potential of storing large, loosely structured data that can later be analyzed and combined as a whole, but they also offer the ability for queries to be applied on such data. This is especially beneficial when real time answers are needed on only pieces of the stored data. Despite the presence of this valuable batch processing potential from NoSQL stores, there is a need for a software pipeline allowing “Big Data” processing models like MapReduce to spout NoSQL data stores as wellspring of input. There is also a need for a software pipeline allowing MapReduce legacy programs written in C, C++, and non-Java executables to use “Big Data” technologies.

A. Cassandra

Cassandra [2] is a non-relational and column-based, distributed database. It was originally developed by Facebook. It is now an open source Apache project. Cassandra is designed to store large datasets over a set of commodity systems by using a peer-to-peer cluster structure to promote horizontal scalability. In the column-based data model of Cassandra, a column is the smallest component of data. Columns associated with a certain key constitute a row. Each row can contain any number of columns. A set of rows forms a column family, which is similar to tables in relational databases. Records in the column families are stored in sorted order by row keys, in separate files. The keyspace congregates one or more column families in the application, similar to a schema in a relational database.

B. 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 processes these data splits in parallel with user-defined map and reduce functions. MapReduce abstracts away from the application programmers the details of input distribution, parallelization, and scheduling and fault tolerance.

Hadoop & Hadoop Streaming: Apache Hadoop is the leading open source MapReduce implementation .Hadoop relies on two fundamental parts: the Hadoop Distributed File System (HDFS) [3] and the Hadoop MapReduce Framework for data management and job execution respectively. A Hadoop JobTracker, running on the master node is responsible for resolving job details (i.e., number of mappers/reducers), monitoring the job progress and worker status. Once a dataset is put into the HDFS, it is split into data chunks and distributed throughout the cluster. Each worker hosting a data split runs a process called DataNode and a TaskTracker that is responsible for processing the data splits owned by the local DataNode.

Hadoop is implemented in Java and needs the map and reduce operations to also be implemented in Java and use the Hadoop API. This creates a challenge for legacy applications where it may be not practical to rewrite the applications in Java or where the source code is no longer available. 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 the map or reduce operations. Hadoop Streaming has a restricted model [4]; it is commonly used to run numerous scientific applications from various disciplines. It allows domain scientists to use legacy applications for complex scientific processing or use simple scripting languages that eliminate the sharp learning curve needed to

Write scalable MapReduce programs for Hadoop in Java. Protein sequence comparison, tropical storm detection, atmospheric river detection and numerical linear algebra are a few examples of domain scientists using Hadoop Streaming on NERSC [5] systems [6]..

Marissa: In earlier work, we highlighted both the performance penalty and application challenges of Hadoop Streaming and introduced MARISSA to address these shortcomings [4], [5]. MARISSA leaves the input management to the underlying shared file system to solely focus on processing. Unlike Hadoop Streaming, MARISSA does not require processes like TaskTrackers and DataNodes for execution of MapReduce operations. Once the input data is split by the master node using the Splitter module and placed into the shared file system, each worker node has access to the input chunks awaiting execution. Unlike Hadoop, MARISSA does not force manifest data locality – rather it leaves such optimizations to the shared file system. Each worker node 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.

II. MAPREDUCE STREAMING OVER CASSANDRA DATA

A. MapReduce Streaming Pipeline for Cassandra Datasets

In this paper, we introduce a MapReduce pipeline that can be used by MapReduce frameworks like Hadoop Streaming and MARISSA that offer MapReduce ability with not Java executables. This pipeline, shown in Figure 1, has three main stages: Data Preparation, Data Transformation (MR1) and Data Processing (MR2).

1) Data Preparation:

Data Preparation, Figure 1a, is the step where input datasets are downloaded from Cassandra servers to the corresponding file system – HDFS for Hadoop Streaming and the shared file system for MARISSA. In both of these frameworks, this step is starting in parallel. Cassandra allows exporting the records of a target dataset in JSON formatted files and using this built-in feature each node downloads the data from the local Cassandra servers to the file system. In experimental setup, each node that is running a Cassandra server is a worker node for the MapReduce framework in use. In experimental data have 3 columns. This choice aims to mimic storing Twitter user interaction logs in Cassandra. For 64 million records, we have 20GB data distributed through Cassandra servers with the replication factor set to 1. They implemented a set of tools to launch the process of exporting data from a Cassandra cluster. For each write request, Cassandra creates a commit log entry and writes mutated columns to an in-memory structure called as Memtable. This inmemory structure is written into an immutable data file named SSTable at a certain size limit or predefined period of time. In implementation, each worker connects in parallel to its local database

Server and flushes Memtables into SSTables After flushing data, a worker begins the exporting operations. Every worker collects the exported records in unique files stored on the shared file system. In MARISSA, they were able to introduce these tools into the Sash module. For Hadoop Streaming, however, they implemented additional tools to initiate the data preparation process in parallel on all worker nodes. Next, this. Data was placed into the HDFS using the put command from the Hadoop master node. In Hadoop's case, the put operation includes splitting the input into chunks and placing those chunks throughout the HDFS cluster. 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.

2) Data Transformation (MR1):

In the Data Preparation stage the input dataset from Cassandra servers is downloaded and placed into the shared file system or HDFS in JSON format. Moving the input dataset out of the database and into the file system also requires the exported data to be transformed into a format that can be processed by the target non-Java applications. Cassandra allows users to export its dataset as JSON formatted files. As Per assumption is that the target executables are legacy applications which are either impossible or impractical to be modified, the input data needs to be converted into a format that is liked by these target applications. For this reason, our software pipeline includes a MapReduce stage, Figure 1b, where JSON data can be converted into other formats. This phase simply processes each input record and converts it to another format, writing the results into the intermediate output files. This step does not include any data or processing dependencies between nodes and hence is a great fit for the MapReduce model. In fact, we only initiate the map stage of MapReduce since no reduce operations are needed. If necessary for the conversion of JSON files to the proper format, a reduce step may be added conveniently. They implemented this stage in Python scripts that can be run using either MARISSA or Hadoop Streaming without any modifications. As this is the first of a series of iterative MapReduce functions whose output will be used as the input by the following MapReduce streaming functions, we simply call this stage MR1. The system not only permits users to convert the dataset into the desired format but also makes it possible to specify the columns of interest. This is extremely useful when a vertical subset of the dataset is sufficient for the actual data processing. This stage helps to reduce data size, in turn affecting the performance of the next MapReduce stage in a positive manner. This performance gain is a result of fewer I/O and data parsing operations.

3) Data Processing (MR2):

This is the final step of the MapReduce Streaming pipeline shown in Figure 1. In Figure 1c they run the non-Java executables, which were the initial target applications, over the output data of MR1, as the data is now in a format that can be processed. They use MARISSA and Hadoop Streaming to run executables as map and reduce operations. Since this is the second MapReduce stage in our pipeline we name it MR2. Any MapReduce streaming job being run after MR1 is considered an MR2 step.

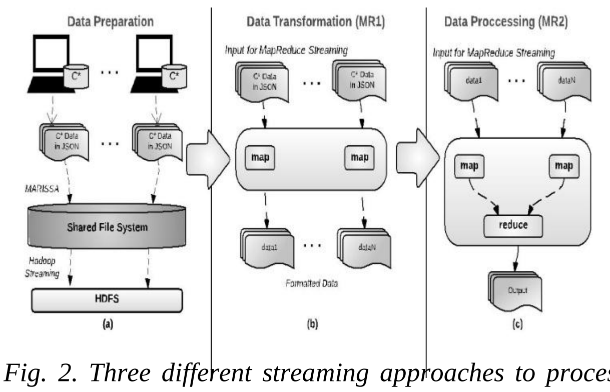

Fig. 2. Three different streaming approaches to process Cassandra datasets with MapReduce using non-Java executables. Figure (a) shows the architecture of using MARISSA for the MapReduce streaming pipeline in Figure 1. The data is first downloaded from the database servers to the shared file system, preprocessed for the target application and at the final stage processed with the user set non-Java executables. In (b), we show the layout of using Hadoop Streaming in such a setting where the dataset is also placed 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 with non-Java executables.*

B. MapReduce Streaming Pipeline with MARISSA

As explained in Section I-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 1a the input is split and ready for Data Transformation. Figure 2a 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.

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 are used as input for the Data Transformation stage. HDFS is a non-POSIX compliant file system that needs to use of HDFS API to interact with the files. Since Hadoop Streaming uses non-Java application for map and/or reduce, the assumption is that these executables do not use this API and therefore do not have quick access to the input splits. Hence, Hadoop TaskTrackers read the input from HDFS and feed into the executables for processing and collect the results to write back to the HDFS. In the Data Transformation step shown in Figure 1b, Hadoop Streaming requires our input conversion code to convert 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 2c, allows users to leave the input dataset on its own 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 starting any of the map operations each Hadoop mapper starts the user specified non-Java executable and later in each map it reads an input record from the database and converts it to the expected format – streaming 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. This design has the limitation that the user specified applications should start before-hand and should expect an input record from stdin and write the output to stdout. Although we explain the limitations of such a model in [4], in this case we use it to provide in-place processing of Cassandra data for cases when it is not practical to constantly download data to the file system in order to process the most up-to-date version.

Figure 2c shows that DataNodes are running on each worker node, however, 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.

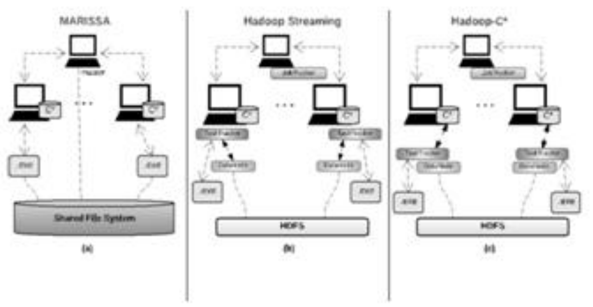

Fig. 2. Three different streaming approaches to process Cassandra datasets with MapReduce using non-Java executables. Figure (a) shows the architecture of using MARISSA for the MapReduce streaming pipeline in Figure 1. The data is first downloaded from the database servers to the shared file system, pre-processed for the target application and at the final stage processed with the user set non-Java executables. In (b), we show the layout of using Hadoop Streaming in such a setting where the dataset is also placed 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 with non-Java executables.*

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. We introduce a MapReduce streaming pipeline for the latter approach and use two different MapReduce streaming frameworks, Hadoop Streaming and MARISSA, to show the applicability of our system under different platforms. Furthermore, we present a detailed performance comparison of each approach under various application scenarios. Our results are summarized in Section V to help users make informed decisions for processing large Cassandra datasets with MapReduce using non-Java executables.

ACKNOWLEDGMENT

It is a great opportunity for me to express my profound gratitude towards our H.O.D, **Prof. Dinesh. D. Patil** who gave me the innovative idea of making this seminar & whose zeal & enthusiasm are source of inspiration for me. Also, I can't ignore the innumerable efforts undertaken by **Prof. Dinesh. D. Patil**, my Seminar Guide, Department of Computer Science & Engineering and I also would like to thank **Prof. R.R.Singh**, Seminar Incharge for his guidance and motivations. I would also like to thank all the staff members of Department of Computer Science & Engineering, Hindi Seva Mandal’s, Shri Sant Gadge Baba College of Engineering & Technology, Bhusawal, for their valuable assistance and support at all times.

REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, Jan. 2008.
[2] 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.
[3] 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.
[4] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, 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
[5] Fadika, Zacharia and Govindaraju, Madhusudhan and Canon, Shane and Ramakrishnan, Lavanya. Evaluting hadoop for data-intensive scientific operations. IEEE Cloud Computing, 2012.
[6] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. Magellan: experiences from a science cloud. In Proceedings of the 2nd international workshop on Scientific cloud computing, ScienceCloud ’11, pages 49–58, New York, NY, USA, 2011. ACM.