Future of Big Data beyond Batch Processing

Mansi Shah[1] Vatika Tayal[2]
[1,2]Department of Computer Science and Engineering
[1,2]N.S.I.T, Jetalpur, Gujarat

Abstract— In recent years, big data are generated from a variety of sources, and there is an enormous demand for storing, managing, processing, and querying on big data. The MapReduce framework and its open source implementation Hadoop, has proven itself as the de facto solution for processing large amounts of data in parallel, and is intrinsically designed for batch processing and high throughput jobs. Although Hadoop has proven as de facto solution for batch jobs, there is growing demand for non-batch applications like: real-time queries, interactive jobs, and big data streams. Since Hadoop is not suitable for these non-batch jobs, new solutions are proposed to meet these new challenges. In this paper, we discuss the strength, features, and shortcomings of the standard MapReduce framework and its open source implementation Hadoop. In addition; we have discussed the significant extensions of MapReduce. Further, we have considered two categories of these solutions: real-time processing, and stream processing of big data. For each category, we have included paradigms and strengths.

Key words: Big data, MapReduce, Hadoop, Real-time processing, Stream processing

I. INTRODUCTION

Recently, the "Big Data" paradigm is expanding its popularity. The term "Big Data" is generally used for datasets which are so enormous that they cannot be managed and processed using solutions like Relational Data Base Systems (RDBMS). Apart from huge volume, large velocity and variety are other challenges of big data [1]. Big data is generated from numerous sources like Internet, Web, Sensors, Online Social Networks, Bioinformatics, Telematics, Earth Sciences, e-Commerce, Cosmology and so on[2].

So far, the most well-known solution that is proposed for processing and managing big data is the MapReduce framework which has been initially developed and used by Google [3]. Its significant features are: a simple and clean programming model, automatic and linear scalability, and fault tolerance. After Google, Apache started some counterpart open source implementation of the MapReduce framework with Hadoop MapReduce and Hadoop YARN as the execution engines, the Hadoop Distributed File System (HDFS), and HBase [4]. From its initiation, the MapReduce framework has made complex large-scale data processing simple and efficient. However, MapReduce framework is designed for high throughput batch processing jobs that several hours, and it is not suitable for current demands like real-time and stream processing jobs that should complete in seconds or at max, minutes. [5,6].

This paper focuses on two new aspects: real-time and stream processing solutions for big data. Primarily, the objective of real-time processing is to provide solutions that can process big data very rapidly and interactively. Stream processing deals with the issues that the input data must be processed without being totally stored. There are various use cases for stream processing like network traffic monitoring, spam detection, fault detection, online machine learning, continuous computation and so on. These new trends need systems that are more sophisticated and quick than the currently available MapReduce solutions like the Hadoop framework. Therefore, new systems and frameworks have been proposed for these new demands and we discuss these new solutions.

II. MAPREDUCE FRAMEWORK

A. Overview:

MapReduce is a programming model that enables massive and distributed processing of big data on a set of cluster machines. MapReduce model defines the computation as two phases: map and reduce. The input is a set of key/value pairs, and the output is a list of key/value pairs. The map phase takes key/value pairs as input, performs computation on this input, and produces intermediate key/value pairs. The reduce phase takes an intermediate key and a list of intermediate values associated with that key as its input, and produces a set of final key/value pairs as the output.

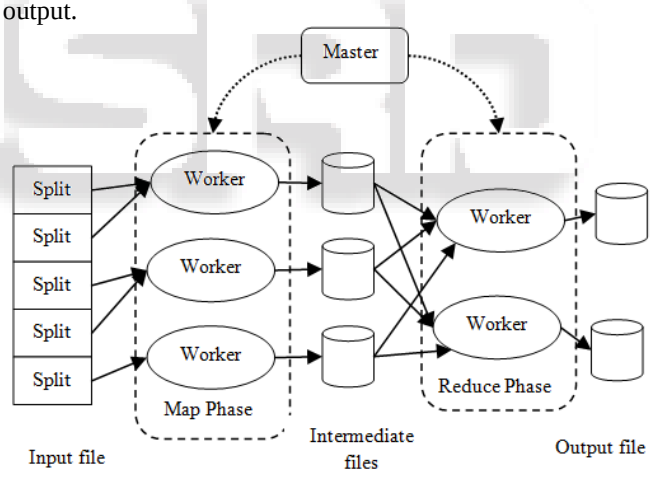

Fig. 1: MapReduce Model

MapReduce model follows the master/slave architecture [7]. The master machine is responsible for task assignment and controlling the slave machines. A schematic diagram of a MapReduce model is shown in Fig.1. The input data is stored over a shared storage such as a distributed file system, and is split into chunks. First, a copy of map and reduce functions' code is issued to all workers. Then, the master assigns map and reduce jobs to workers. Each worker with a map task reads the corresponding input split and passes all of its pairs to map function and writes the results of the map function into intermediate files. The data from the map phase is shuffled, i.e. exchanged and merge-sorted, to the set of machines performing the reduce phase. On completion of the map phase, the reducer workers read intermediate files and pass the intermediate pairs to reduce function and finally the pairs resulted by reduce tasks are written to final output files.

B. Apache Hadoop:

There are several implementations of MapReduce for distributed systems like Apache Hadoop, HPCC from LexisNexis, Dryad from Microsoft [8], and Sector/Sphere. Undoubtedly, Hadoop is the most popular open source implementation of MapReduce. Hadoop primarily consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. By default, Hadoop uses its distributed file system HDFS; to store input and output files. In addition, Hadoop provides pluggable input and output sources. For example, it can use NoSQL databases like HBase and Cassandra and also relational databases instead of HDFS.

Hadoop has several features that come from the MapReduce model like easy programming model, speed and high scalability, and fault tolerance are three major features. Apart from these, Hadoop itself provides few extra features like distinct schedulers, more complex and sophisticated job definitions using YARN, highly available master machines, pluggable I/O facilities, and etc. Hadoop offers the basic platform for big data processing.

C. Limitations and Weaknesses:

Despite its evident strengths, MapReduce often fails to exhibit acceptable performance for various processing tasks. Quite often this is due to the weaknesses related to the nature of MapReduce or the applications and use-cases it was originally designed for. In other cases, it is due to the limitations of the processing model adopted in MapReduce [9].

1) High Communication Cost:

Once the map tasks complete processing, selected data are sent to reduce tasks for further processing i.e. shuffling. Depending on the query under execution and on the kind of processing that takes place during the map phase, the size of the output of the map phase can be quite significant and its transmission may delay the overall execution time of the job.

2) Redundant and Wasteful Processing:

Often multiple MapReduce jobs are started at over lapping time intervals and require to be processed over the same dataset. In such cases, it is possible that two or more jobs need to perform the same processing over the same dataset. In MapReduce, such jobs are processed independently from each other, thus resulting in redundant processing which increases the execution time.

3) Lack of Interactive or Real-Time Processing:

MapReduce is designed as a very fault-tolerant system for batch processing of long-running jobs on very huge data sets. However, it is unsuitable for interactive or real-time processing, which requires fast processing times. The main reason is that in order to guarantee fault-tolerance, MapReduce introduces significant overheads like frequent writing of output to disk, extensive communication between tasks for failure detection, limited exploitation of main memory, delays for job initiation and scheduling , that negatively impact its performance.

4) Recomputation:

To ensure fault-tolerance during the processing of the job, MapReduce clusters produce output results that are stored on disk. This is basically a check-pointing mechanism that allows long-running jobs to complete processing in the event of failures, without the need to restart processing from scratch. However, MapReduce does not provide mechanism for future reuse of output results. Thus, there exists that requires the result of a no opportunity for reusing the results produced by previous queries, which means that a future query previously processed query will have to resolve in recomputing everything.

5) Configuration and Automation Issues:

When deploying a Hadoop MapReduce cluster there are numerous configuration parameters to be set which impact the performance. Options include the number of parallel tasks, the file block size and the replication factor. Proper configuration of these parameters requires knowledge of both the available workload characteristics and hardware, whereas misconfiguration might lead to underutilization of resources and inefficient execution.

D. MapReduce Extensions:

Several extensions of the MapReduce framework have been proposed which try to improve its performance and usability. These extensions have focused on adding recursion and iteration to MapReduce. The major examples are HaLoop [11] and Twister [10]. Both of these solutions efficiently support iterative jobs, and provide fault tolerance and data caching between iterations. Few other frameworks provide easier program expression on top of MapReduce. Tez supports both interactive and batch jobs and provides an easy API to write applications for YARN [12]. FlumeJava is a library created by Google which allows creating data pipelines on top of MapReduce [13]. Writing single jobs in MapReduce is easy but maintaining a series of job designed to handle a complex procedure is not easy. FlumeJava makes the above procedure easier and it translates the defined pipeline to an efficient series of MapReduce jobs. Apache Crunch is an open source implementation of FlumeJava for Hadoop. The two MapReduce frameworks that are designed for execution on shared memory parallel systems are Phoenix and Metis [14]. The MapReduce framework that executes on GPU is Mars [15].

However, MapReduce and above discussed extensions are widely designed for batch processing of fully staged big data and they are inappropriate for processing interactive workloads and streaming big data. These shortcomings have triggered a need to create new solutions. Further, we will discuss two types of these solutions: solutions that try to add real-time processing, and interactivity features to MapReduce; and solutions that try to facilitate stream processing of big data.

III. REAL-TIME BIG DATA PROCESSING

Solutions in this class can be classified into two main categories: (i) Solutions which try to reduce the overhead of MapReduce and make it quicker to enable execution of jobs in less than seconds; (ii) Solutions which focus on providing a way for real-time queries over structured and unstructured big data using new improved approaches. Further, we discuss both categories respectively.

A. In-Memory Processing:

Slowness of Hadoop is due to two main reasons. First, Hadoop was inherently designed for batch processing jobs.

Scheduling, code transfer to slaves, task assignment, and job startup procedures are not designed and programmed to finish in less than seconds. The second bottleneck to its performance is the HDFS file system. HDFS is designed for high throughput data I/O rather than high performance I/O. Data blocks in HDFS are very large and stored on hard drives which with current technology can deliver transfer rates between 100 and 200 megabytes per second.

The first issue can be resolved by redesigning job startup and task execution modules. But, the file system problem is inherently caused by hardware. Even if every machine is equipped with several hard disk modules, the I/O rate would be several hundreds of megabytes per seconds which will take minutes rather than seconds. An optimized solution to this issue is In-Memory processing which uses a distributed main memory system to store and process big data in real-time. In-memory computing does not mean the entire data should be kept in memory. Even if a distributed pool of memory is available and the framework uses that memory for caching of frequently used data, the whole job execution performance can be improved significantly. Main memory generates higher bandwidth compared to hard disk so access latency is also much better.

Apache Spark, GridGain, and XAP are few in-memory computing solutions available. Amongst them, Spark is both open source and free, but others are commercial. Caching paradigm is supported by both Spark and GridGain. Spark uses an abstraction called Resilient Distributed Dataset (RDD) that is a distributed, fault-tolerant collection of data items [15]. Spark can be easily integrated with Hadoop and RDDs can be generated from data sources like HBase and HDFS. GridGain also has its own in-memory file system called GridGain File System (GGFS) that acts as either a standalone file system or in combination with HDFS, acting as a caching layer. In-memory caching is useful in handling enormous streaming data that can easily curb disk-based storages.

We must mention that in-memory computing does not mean the whole data should be kept in memory. Even if a distributed pool of memory is available and the framework uses that memory for caching of frequently used data, the whole job execution performance can be improved significantly. Efficient caching is especially effective when an iterative job is being executed. Both Spark and GridGain support this caching paradigm. Spark uses a primary abstraction called Resilient Distributed Dataset (RDD) that is a distributed collection of items [16]. Spark can be easily integrated with Hadoop and RDDs can be generated from data sources like HDFS and HBase. GridGain also has its own in-memory file system called GridGain File System (GGFS) that is able to work as either a standalone file system or in combination with HDFS, acting as a caching layer. In-memory caching can also help handling huge streaming data that can easily stifle disk-based storages.

B. Real-Time Interactive Queries over Big Data:

Firstly, the term "real-time" in big data is closer to interactivity rather than milliseconds response. In big data processing domain, real-time queries must respond in order of seconds and minutes rather than batch jobs which complete in hours and days. The first solution in this area that try to enable real-time ad-hoc queries over big data is Dremel by Google [17]. Dremel uses a columnar storage format for nested structures and uses scalable aggregation algorithms for processing query results in parallel. Cloudera Impala is an open source equivalent that tries to provide an open source implementation of Dremel. For this purpose, Impala has developed a more efficient columnar binary storage for Hadoop called Parquet and uses techniques of parallel DBMSs to compute ad hoc queries in real-time. Impala claims significant performance gains for queries with joins, over Apache Hive. Although Impala shows promising improvements over Hive, it is still a stable solution for long running queries and analytics.

There are even more solutions in this category. Apache Drill is another Dremel-like solution and it provides real-time queries against other storage systems like Cassandra. Shark is another solution that is built on top of Spark that is designed to be compatible with Apache Hive and can execute all queries that are possible for Hive. The in memory computing capability and the fast execution engine of Spark makes Shark execute 100x times faster compared to Hive [18]. The final notable solution is Amazon Redshift which is a solution from Amazon that mainly aims to provide a very fast solution to petabytes-scale warehousing.

IV. STREAMING BIG DATA

With the acceleration in Web Technology, E-commerce application, Machine-to-Manchine communication (M2M), data streams are very common. Log streams, event streams, click streams, message streams, and are few good examples. However, the standard MapReduce programming model and its de facto standard implementations like Hadoop is completely focused on batch processing. That is, before any computation is started, all of the input data to be processed must be completely available on the input store, e.g., HDFS. The framework processes the input data and when all of the computation is done the output results are available. Further, a MapReduce job execution is not continuous. But for the recent streaming applications the input data is not available completely at the beginning and arrives constantly. In addition, sometimes an application must run continuously, e.g., a query that detects some failures or anomalies in network.

Although MapReduce does not provide support stream processing, but it can partially handle streams using a technique called micro-batching [19]. The idea is to consider the stream as a sequence of small batch of data. On small time intervals, the incoming stream is packed to a chunk of data and is delivered to the batch system for processing. Spark and GridGain support this technique. In Spark the streaming support is called DStream or discretized stream which is represented as a sequence of Resilient Distributed Datasets (RDDs). The in-memory processing feature of Spark enables it to compute data batches much faster than Hadoop.

V. CONCLUSION

MapReduce has brought new excitement in the parallel data processing outlook. This is due to its remarkable features that include simplicity, scalability, fault-tolerance, and flexibility. Still, a number of its shortcomings show that MapReduce is imperfect for non-batch jobs like interactive

jobs, real-time queries, and stream data. Considering the high demands for non-batch jobs, in-memory computing proves to be a notable solution that can handle both real-time and stream requirements. We expect to see more frameworks targeting requirements for real-time, interactive and stream processing and analysis in the near future.

REFERENCES

[1] A. Jacobs, "The pathologies of big data," Communications of the ACM, vol. 52, no. 8, pp. 36–44, 2009.
[2] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.
[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Transactions on Computer Systems, vol. 26, no. 2, p. 4, 2008.
[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Proceedings of the 6th Symposium on Opearting Systems Design & Implementation, 2004.
[5] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in Proceedings of the 14th International Conference on Extending Database Technology, 2011, pp. 1–8.
[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010.
[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012.
[8] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive Analysis of Web-scale Datasets," Proceedings of VLDB Endowment, vol. 3, no. 1–2, pp. 330–339, 2010.
[9] Christos Doulkeridis; Kjetil Nørvåg, A survey of large-scale analytical query processing in MapReduce, The VLDB Journal, vol. 23, pp. 355-380, 2014.
[10] Ekanayake, J.; Li, H.; Zhang, B.; Gunarathne, T.; Bae, S.-H.; Qiu, J.; Fox, G. Twister: A runtime for iterative MapReduce. In Proceedings of the 19th ACM International Symposium on High. Performance Distributed Computing, Chicago, IL, USA, 20–25 June 2010; pp. 810–818.
[11] Bu, Y.; Howe, B.; Balazinska, M.; Ernst, M.D. HaLoop: Efficient iterative data processing on large clusters. Proc. VLDB Endow. 2010, 3, 285–296.
[12] Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; p. 5.
[13] Chambers, C.; Raniwala, A.; Perry, F.; Adams, S.; Henry, R.R.; Bradshaw, R.; Weizenbaum, N. FlumeJava: Easy, efficient data-parallel pipelines. ACM Sigplan Not. 2010, 45, 363–375.
[14] Yoo, R.M.; Romano, A.; Kozyrakis, C. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In Proceedings of IEEE International Symposium on Workload Characterization, Austin, TX, USA, 4–6 October 2009; pp. 198–207.
[15] Fang, W.; He, B.; Luo, Q.; Govindaraju, N.K. Mars: Accelerating MapReduce with Graphics Processors. IEEE Trans. Parallel Distrib. Syst. 2010, 22, 608–620.
[16] Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012.
[17] Melnik, S.; Gubarev, A.; Long, J.J.; Romer, G.; Shivakumar, S.; Tolton, M.; Vassilakis, T. Dremel: Interactive Analysis of Web-scale Datasets. Proc. VLDB Endow. 2010, 3, 330–339.
[18] Engle, C.; Lupher, A.; Xin, R.; Zaharia, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Shark: Fast data analysis using coarse-grained distributed memory. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012; pp. 689–692.
[19] Hadian, A.; Shahrivari, S. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. J. Supercomput. 2014, 69, 845–863.

