Computers **2014**, *3*, 117-129; doi:10.3390/computers3040117

OPEN ACCESS

computers

ISSN 2073-431X

www.mdpi.com/journal/computers

Review

Beyond Batch Processing: Towards Real-Time and Streaming Big Data

Saeed Shahrivari

Department of Computer Engineering, Tarbiat Modares Univeristy, Tehran 14115-194, Iran;
E-Mail: s.shahrivari@modares.ac.ir; Tel./Fax: +98-21-8282-3521

External Editor: Aaron Quigley

Received: 24 June 2014; in revised form: 26 September 2014 / Accepted: 9 October 2014 / Published: 17 October 2014

Abstract: Today, big data are generated from many sources, and there is a huge demand for storing, managing, processing, and querying on big data. The MapReduce model and its counterpart open source implementation Hadoop, has proven itself as the de facto solution to big data processing, and is inherently designed for batch and high throughput processing jobs. Although Hadoop is very suitable for batch jobs, there is an increasing demand for non-batch requirements like: interactive jobs, real-time queries, and big data streams. Since Hadoop is not suitable for these non-batch workloads, new solutions are proposed to these new challenges. In this article, we discussed two categories of these solutions: real-time processing, and stream processing of big data. For each category, we discussed paradigms, strengths and differences to Hadoop. We also introduced some practical systems and frameworks for each category. Finally, some simple experiments were performed to approve effectiveness of new solutions compared to available Hadoop-based solutions.

Keywords: big data; MapReduce; real-time processing; stream processing

---

1. Introduction

The "*Big Data*" paradigm has experienced expanding popularity recently. The "*Big Data"* term is generally used for datasets which are so huge that they cannot be processed and managed using classical solutions like *Relational Data Base Systems (RDBMS)*. Besides volume, large velocity and variety are other challenges of big data [1]. Numerous sources generate big data. Internet, Web, Online

Social Networks, Digital Imaging, and new sciences like Bioinformatics, Particle Physics, and Cosmology are some example of sources for big data to be mentioned [2]. The emerging of cloud computing have also made big data processing and mining easier [3].

Until now, the most notable solution that is proposed for managing and processing big data is the MapReduce framework which has been initially introduced and used by Google [4]. MapReduce offers three major features in a single package. These features are: a simple and easy programming model, automatic and linear scalability, and built-in fault tolerance. Google announced its MapReduce framework as three major components: a MapReduce execution engine, a distributed file system called *Google File System (GFS)* [5], and a distributed NoSQL database called *BigTable* [6].

After Google's announcement of its MapReduce Framework, the Apache foundation started some counterpart open source implementation of the MapReduce framework. *Hadoop MapReduce* and *Hadoop YARN* as execution engines, the *Hadoop Distributed File System (HDFS)*, and *HBase* as a replacement for BigTable, were the three major projects [7]. Apache has also gathered some extra projects like: *Cassandara* a distributed data management system resembling to *Amazon Dynamo*, *Zookeeper* a high-performance coordination service for distributed applications, *Pig* and *Hive* for data warehousing, and *Mahout* for scalable machine learning.

From its inception, the Mapreduce framework has made complex large-scale data processing easy and efficient. Despite this, MapReduce is designed for batch processing of large volumes of data, and it is not suitable for recent demands like real-time and online processing. MapReduce is inherently designed for high throughput batch processing of big data that take several hours and even days, while recent demands are more centered on jobs and queries that should finish in seconds or at most, minutes [8,9].

In this article, we give a brief survey with focus on two new aspects: real-time processing and stream processing solutions for big data. An example for real-time processing is fast and interactive queries on big data warehouses, in which user wants the result of his queries in seconds rather than in hours. Principally, the goal of real-time processing is to provide solutions that can process big data very fast and interactively. Stream processing deals with problems that their input data must be processed without being totally stored. There are numerous use cases for stream processing like: online machine learning, and continuous computation. These new trends need systems that are more elaborate and agile than the currently available MapReduce solutions like the Hadoop framework. Hence, new systems and frameworks have been proposed for these new demands and we discuss these new solutions.

We have divided the article into three main sections. First, we discuss the strength, features, and shortcomings of the standard MapReduce framework and its de facto open source implementation Hadoop. In addition to standard MapReduce, we introduce significant extensions of MapReduce. Then, we discuss real-time processing solutions. Afterwards, we discuss stream processing systems. At the end of the article, we give some experimental results comparing the discussed paradigms. Finally, we present a conclusion.

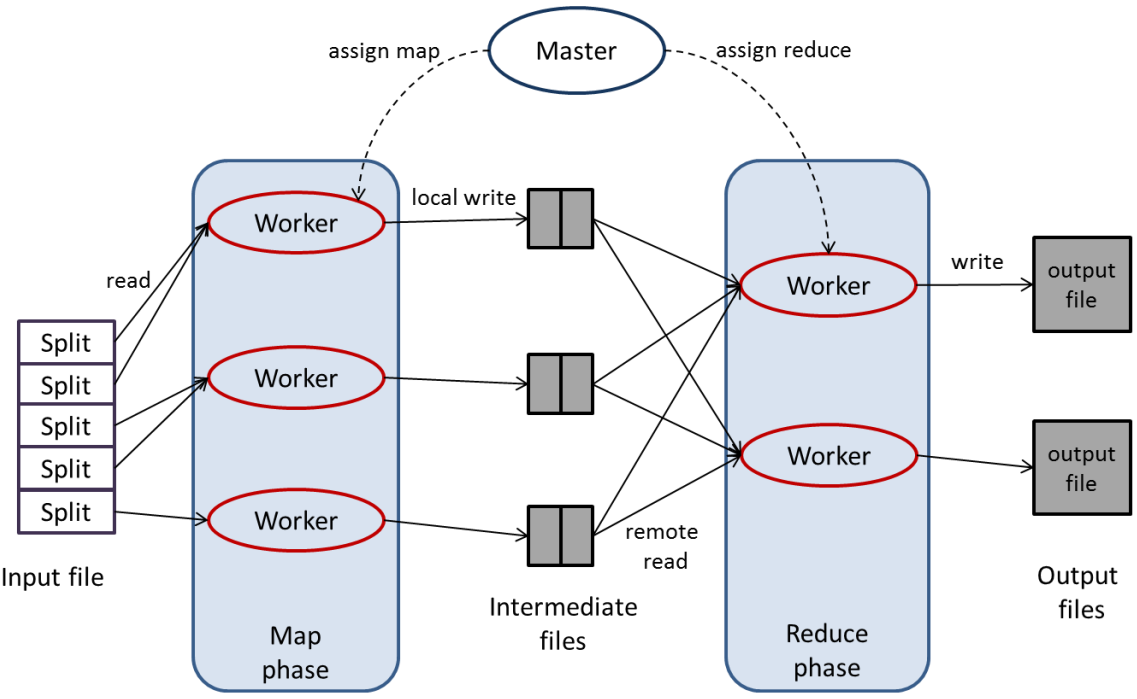
2. The MapReduce Framework

Essentially, MapReduce is a programming model that enables large-scale and distributed processing of big data on a set of commodity machines. MapReduce defines the computation as two functions:

map and *reduce*. The input is a set of key/value pairs, and the output is a list of key/value pairs. The *map* function takes an input pair and results in a set of intermediate key/value pairs (which can be empty). The *reduce* function takes an intermediate key and a list of intermediate values associated with that key as its input, and results set of final key/value pairs as the output. Execution of a MapReduce program involves two phases. In the first phase, each input pair is given to *map* function and a set of input pairs is produced. Afterwards, in the second phase, all of the intermediate values that have the same key are aggregated into a list, and each intermediate key and its associated intermediate value list are given to a reduce function. More explanation and examples are available in [4].

The execution of a MapReduce program obeys the same two-phase procedure. Usually, distributed MapReduce is implemented using master/slave architecture [10]. The master machine is responsible of assignment of tasks and controlling the slave machines. A schematic for execution of a MapReduce program is given in Figure 1. The input is stored over a shared storage like distributed file system, and is split into chunks. First, a copy of *map* and *reduce* functions' code is sent to all workers. Then, the master assigns *map* and *reduce* tasks to workers. Each worker assigned 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. After the *map* phase is finished, 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.

Figure 1. Execution of a MapReduce program.



2.1. Apache Hadoop

There are several MapReduce-like implementations for distributed systems like *Apache Hadoop*, *Disco* from Nokia, *HPCC* from LexisNexis, *Dryad* from Microsoft [11], and *Sector/Sphere*. However, Hadoop is the most well-known and popular open source implementation of MapReduce. Hadoop uses master/slave architecture and obeys the same overall procedure like Figure 1, for executing programs.

By default, Hadoop stores input and output files on its distributed file system, HDFS. However, Hadoop provides pluggable input and output sources. For example, it can also use NoSQL databases like HBase and Cassandra and even relational databases instead of HDFS.

Hadoop has numerous strengths. Some of its strengths come from the MapReduce model. For example, easy programming model, near-linear speedup and scalability, and fault tolerance are three major features. Besides these, Hadoop itself provides some extra features like: different schedulers, more sophisticated and complex job definitions using YARN, high available master machines, pluggable I/O facilities, and *etc.* Hadoop provides the basic platform for big data processing. For more usability, other solutions can be mounted over Hadoop [7]. Major examples are HBase for storing structured data on very large tables, Pig and Hive for data warehousing, and Mahout for machine learning.

Although Hadoop, *i.e.*, standard MapReduce, has numerous strengths, but it has several shortcomings too. MapReduce is not able to execute recursive or iterative jobs inherently [12]. Total batch behavior is another problem. All of the input must be ready before the job starts and this prevents MapReduce from online and stream processing use cases. The overhead of framework for starting a job, like copying codes and scheduling, is another problem that prevents it from executing interactive jobs and near real-time queries. MapReduce cannot run continuous computations and queries, too.

2.2. MapReduce Extensions

Several extensions of the standard MapReduce framework have been proposed which try to improve its usability and performance. Some works have focused on adding iteration and recursion to MapReduce. The major examples are *Twister* [13] and *HaLoop* [14]. Both of these solutions support iterative jobs efficiently, and provide data caching and fault tolerance between iterations. Some other frameworks provide easier program expression on top of MapReduce. *Tez* provides an easy API to write applications for YARN and supports both interactive and batch jobs [15]. *FlumeJava* is a library created by Google that allows building data pipelines on top of MapReduce [16]. Although writing single jobs in MapReduce is easy, maintaining a series of job designed to handle a complex procedure is not easy. FlumeJava makes this 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. *Cascading* is another abstraction layer that allows the creation of complex workflows on Hadoop. MapReduce is implemented for other platforms, too. *Phoenix* and Metis are two MapReduce frameworks that are designed for execution on shared memory parallel systems [17]. Mars is another MapReduce framework that executes on GPU [18].

2.3. Other Models

MapReduce is the most well-known model for distributed big data processing, but there are also other models. *Bulk Synchronous Parallel (BSP)* model is older than MapReduce and recently has gained some popularity [19]. In the BSP model, the program is defined as a series of *supersteps*. Each superstep consists of small steps, which are actually local computation. After each superstep, there is a synchronization barrier. At the synchronization point, system waits until all processor units finish their steps and communications are performed. Google has found the BSP model more suitable for processing graph data and running graph algorithms, and used BSP in *Pregel* framework which is

inherently designed for graph processing [20]. *Apache Hama* is an open source framework which implements the BSP model. *Giraph* is another open source graph processing framework that is based on the BSP model and executes on Hadoop. The most high performance graph processing framework is *GraphLab* which is developed at Carnegie Melon University and uses the BSP model and executes on MPI [21].

As we discussed above, MapReduce and its extensions are mostly designed for batch processing of fully staged big data and they are not appropriate for processing interactive workloads and streaming big data. These shortcomings have triggered creation of new solutions. Next, we will discuss two types of these solutions: (i) solutions that try to add real-time processing, and interactivity capabilities to MapReduce; and (ii) solutions that try to provide stream processing of big data.

3. Real-Time Big Data Processing

Solutions in this sector can be classified into two major categories: (i) Solutions that try to reduce the overhead of MapReduce and make it faster to enable execution of jobs in less than seconds; (ii) Solutions that focus on providing a means for real-time queries over structured and unstructured big data using new optimized approaches. Here, we discuss both categories respectively.

3.1. In-Memory Computing

Slowness of Hadoop is rooted in two major reasons. First, Hadoop was initially designed for batch processing. Hence, starting execution of jobs is not optimized for fast execution. Scheduling, task assignment, code transfer to slaves, and job startup procedures are not designed and programmed to finish in less than seconds. The second reason is the HDFS file system. HDFS by itself 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 disk drives which with current technology can deliver transfer rates between 100 and 200 megabytes per second.

The first problem can be solved by redesigning job startup and task execution modules. However, the file system problem is inherently caused by hardware. Even if each machine is equipped with several hard disk modules, the I/O rate would be several hundreds of megabytes per seconds. This means that if we store 1 terabytes of data on 20 machines, even a simple search over the data will take minutes rather than seconds. An elegant solution to this problem is *In-Memory Computing*. In a nutshell, in-memory computing is based on using a distributed main memory system to store and process big data in real-time.

Main memory delivers higher bandwidth, more than 10 gigabytes per second compared to hard disk's 200 megabytes per second. Access latency is also much better, nanoseconds *versus* milliseconds for hard disks. Price of RAM is also affordable. Currently, 1 TB of RAM can be bought with less than 20,000$. These performance superiority combined with the dropping price of RAM makes in-memory computing a promising alternative to disk-based big data processing. There are few in-memory computing solutions available like: *Apache Spark* [22], *GridGain*, and *XAP*. Amongst them, Spark is both open source and free, but others are commercial.

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 [23]. 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.

Another important point to be mentioned is the difference between in-memory computing and in-memory databases and data grids. Although in-memory databases like Oracle *Times Ten* and VMware *GemFire* and in-memory data grids like *Hazelcast*, Oracle *Coherence*, and Jboss *Infinispan* are fast and have important use cases for today, but they differ from in-memory computing. In-memory computing is rather a paradigm than a product. As its name implies, in-memory computing deals with computing, too; in contrast to in-memory data solutions that just deal with data. Hence, it should also take care of problems like efficient scheduling, and moving code to data rather than wrongly moving data to code. Despite this, in-memory data grids can be used as a building block of in-memory computing solutions.

3.2. Real-Time Queries over Big Data

First, we should mention that the "real-time" term in big data is closer to interactivity rather than milliseconds response. In big data processing realm, real-time queries should respond in order of seconds and minutes rather than batch jobs which finish in hours and days. The first work in the area of solutions that try to enable real-time ad-hoc queries over big data is *Dremel* by Google [24]. Dremel uses two major techniques to achieve real-time queries over big data: (i) Dremel uses a novel columnar storage format for nested structures (ii) Dremel uses scalable aggregation algorithms for computing query results in parallel. These two techniques enable Dremel to process complex queries in real-time. *Cloudera Impala* is an open source counterpart that tries to provide an open source implementation of Dremel techniques. For this purpose, Impala has developed an efficient columnar binary storage for Hadoop called *Parquet* and uses techniques of parallel DBMSs to process ad hoc queries in real-time. Impala claims considerable 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 analytics and queries.

There are even more solutions in this sector. *Apache Drill* is also another Dremel-like solution. However, Drill is not designed to just be a Hadoop-only solution and it provides real-time queries against other storage systems like Cassandra. *Shark* is another solution that is built on top of Spark [25]. Shark is designed to be compatible with Apache Hive and it can execute all queries that are possible for Hive. Using in-memory computing capability and the fast execution engine of Spark, Shark claims up to 100x faster response times compared to Hive [25]. We should also mention the *Stinger* project by *Hortonworks* which is an effort to make 100x performance improvement and add SQL semantics to future versions of Apache Hive. The final mentionable solution is *Amazon Redshift* which is a

propriety solution from Amazon that aims to provide a very fast solution to petabytes-scale warehousing.

4. Streaming Big Data

Data streams are now very common. Log streams, click streams, message streams, and event streams are some good examples. However, the standard MapReduce model and its implementations like Hadoop, is completely focused on batch processing. That is to say, before any computation is started, all of the input data must be completely available on the input store, e.g., HDFS. The framework processes the input data and the output results are available only when all of the computation is done. On the other hand, a MapReduce job execution is not continuous. In contrast to these batch properties, today's applications need more stream-like demands in which the input data is not available completely at the beginning and arrives constantly. Also, sometimes an application should run continuously, e.g., a query that detects some special anomalies from incoming events.

Although MapReduce does not support stream processing, but it can partially handle streams using a technique known as *micro-batching* [26]. The idea is to treat the stream as a sequence of small batch chunks of data. On small intervals, the incoming stream is packed to a chunk of data and is delivered to the batch system to be processed. Some MapReduce implementations, especially real-time ones like Spark and GridGain support this technique. In Spark the streaming support is called discretized stream or DStream which is represented as a sequence of RDDs. A schematic of stream processing in Spark is given in Figure 2. The in-memory computing feature of Spark enables it to compute data batches quicker than Hadoop. However, this technique is not adequate for the demands of a true stream system. Furthermore, the MapReduce model is not suitable for stream processing.

Figure 2. Schematic of stream processing in Spark.

Currently, there are a few stream processing frameworks that are inherently designed for big data streams. Two notable ones are *Storm* from Twitter, and S4 from Yahoo [27]. Both frameworks run on the Java Virtual Machine (JVM) and both process keyed streams. However, the programming model of the frameworks is different. In S4, a program is defined in terms of a graph of *Processing Elements (PE)* and S4 instantiates a PE per each key. On the other hand, in Storm, a program is defined by two abstractions: *Spouts* and *Bolts*. A spout is a source of stream. Spouts can read data from an input queue or even generate data themselves. A bolt process one or more input streams and produces a number of output streams. Most of the process logic is expressed in bolts. Each Storm program is a graph of spouts and bolts which is called a *Topology* [28].

Considering the programming models we can say that in S4 the program is expressed for keys while in Storm the program is expressed for the whole stream. Hence, programming for S4 has a simpler logic while for Storm; programming is more complex but more versatile too. A major strength of Storm over S4 is fault tolerance. In S4, at any stage of the process, if the input buffer of a PE gets full,

the incoming messages will be simply dropped. S4 also uses a check pointing strategy for fault tolerance. If a node crashes, its PEs will be restarted on another node from their latest state. Hence, any process after the latest state is lost [27]. However, Storm guarantees process of each tuple if the tuple successfully enters Storm. Storm does not store states for bolts and spouts but if a tuple does not traverse the Storm topology in a predefined period, the spout that had generated that tuple will replay it.

Actually, S4 and Storm take two different strategies. S4 proposes a simpler programming model and it restricts the programmer in declaring the process, but instead it provides simplicity and more automated distributed execution, for example automatic load balancing. In contrast, Storm gives the programmer more power and freedom for declaring the process, but instead the programmer should take care of things like load balancing, tuning buffer sizes, and parallelism level for reaching optimum performance. Each of Storm and S4 has its own strengths and weaknesses, but currently Storm is more popular and has a larger community of users compared to S4. Due to the demands and applications, streaming big data will certainly grow much more in the future.

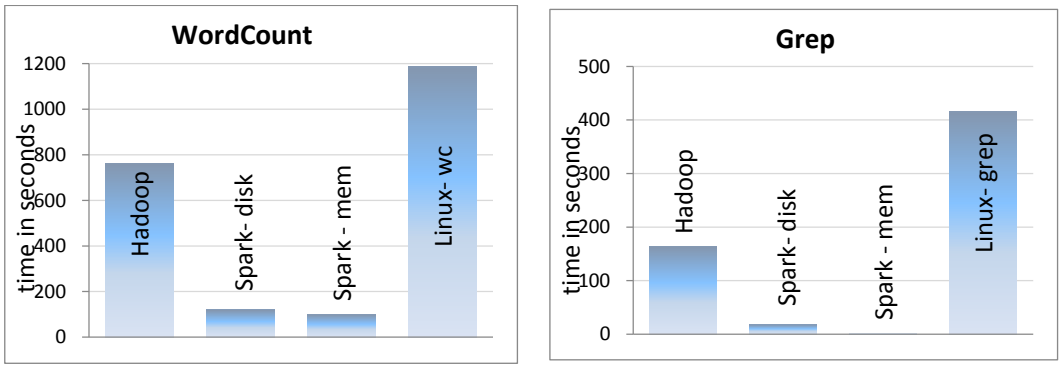
5. Experimental Results

We undertook some simple experiments for better illumination of the discussed concepts. The experiments are aimed to show the improvements of some of the mentioned systems compared to Hadoop. We do not intend to compare the performance of technologies in details. We simply want to show that recent systems in the area of real-time and streaming big data are more suitable than Hadoop. For the case of real-time in-memory computing, we selected Spark. We executed simple programs like *WordCount* and *Grep*, and compared the performance results to Hadoop. For the case of real-time queries over big data, a comprehensive benchmark is done by the Berkeley AMP Lab [29]. Hence, for this category, we just reported a summary of that benchmark. For the case of streaming big data, we used S4, Storm, and Hadoop, and solved a web page stream analysis problem using the mentioned solutions.

WordCount counts occurrences of each word in a given text and Grep extracts matching strings of a given pattern in a given text. WordCount is CPU intensive, but Grep is I/O intensive if a simple pattern is being searched. We searched for simple word; hence, Grep is totally I/O intensive here. For this experiment, we used a cluster of five machines, each having two 4-core 2.4 GHz Intel Xeon E5620 CPU and 20 GBs of RAM. We installed Hadoop 1.2.0, and Spark 0.8 on the cluster. For running WordCount and Grep, we used an input text file of size 40 GBs containing texts of about 4 million Persian web pages. For better comparison, we also executed the standard *wc* (Version 8.13) and *grep* (Version 2.10) programs of Linux on a single machine and reported their times, too. Actually, the wc command of Linux just counts number of all words, not occurrences of each word. Hence, it performs a much simpler job. The results are reported in Figure 3.

As the diagrams show, Spark outperforms Hadoop in both experiments when input is on disk and when input is totally cached in RAM. In the WordCount problem, there is a little difference between in-memory Spark and on-disk Spark. On the other hand, for the Grep problem, there is a significant difference between in-memory Spark and on disk cases. Especially, when the input file is totally cached in memory, Spark executes Grep in a second while Hadoop takes about 160 s.

Figure 3. Hadoop performance compared to Spark.

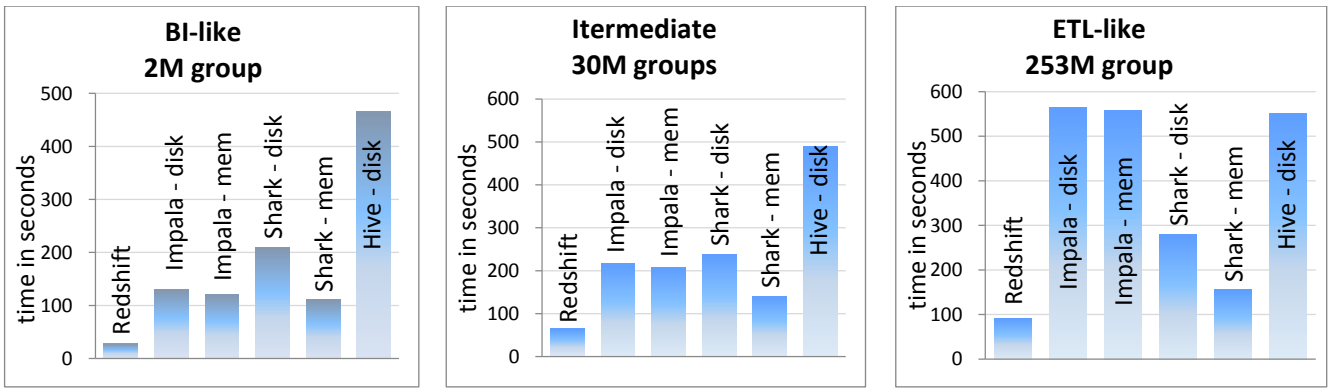


The Berkeley AMP Lab has compared several frameworks and benchmarked their response time on different types of queries like: scans, aggregations, and joins on different data sizes. The benchmarked solutions are: Amazon Redshift, Hive, Shark, and Impala. The input data set is a set of HTML documents and two SQL tables. For better benchmarking, queries were executed with varying results sets: (i) BI-like results which can be easily fit in a BI tool; (ii) Intermediate results which may not fit in memory of a single node; and (iii) ETL-like results which are so large that they require several nodes to store.

Two different clusters were used for this experiment: a cluster of five machines with total 342 GBs of RAM, 40 CPU cores, and 10 hard disks for running Impala, Hive, and Shark, and a cluster of 10 machines with total 150 GBs of RAM, 20 CPU cores, and 30 hard disks for executing Redshift. Both clusters were launched on the Amazon EC2 cloud computing infrastructure. The Berkeley AMP Lab benchmark is very comprehensive [29]. In this article, we just report the aggregation query results. The executed aggregation query is like "*SELECT * FROM foo GROUP BY bar*" SQL statement. The results are given in Figure 4.

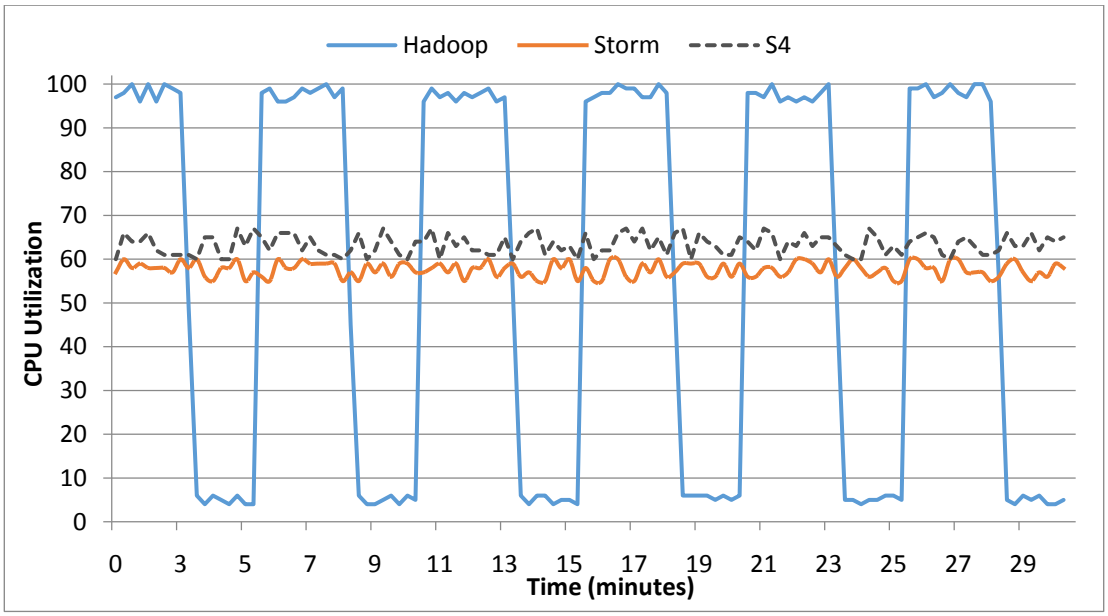
As the results show, new-generation solutions show promising better performance compared to classic MapReduce-based solution, Hive. The only exception is the ETL-like query, in which Hive performs the same as Impala and this is because the result is so large that Impala cannot handle it properly. Despite this, other solutions like Shark and Redshift perform better than Hive for all result sizes.

Figure 4. Berkeley AMP Lab's benchmark results for real-time queries.



For comparing Hadoop, Storm, and S4, we considered a web page stream problem in which a distributed crawler fetches pages from the web and makes a stream of crawled web pages. The crawled pages should be analyzed in real-time. Analyzing a web page consists of parsing, link extraction, NLP tasks, *etc.*, which are complex tasks. The crawler fetches about 400 pages per second and the average time for analyzing a web page is about 50 ms. Hence, the workload cannot be handled on a single machine and needs a distributed system with at least 20 processor cores. For this experiment, we used a cluster of five machines, each having two 4-core 2.4 GHz Intel Xeon E5620 CPU and 20 GBs of RAM. We used Hadoop 1.2.0, S4 0.6.0, and Storm 0.9.1 in this experiment.

Figure 5. Comparison of CPU utilization in Hadoop, Storm, and S4.



While S4 and Storm can consume the input stream directly, Hadoop cannot handle data streams. Therefore, we used a micro batching approach for Hadoop. We used an intermediate store in HDFS and put web pages in the store. Every 5 min, we start a Hadoop job and process all pages in the intermediate store. A plot of aggregated CPU utilization of the cluster for a period of 30 min is given in Figure 5. As Figure 5 shows, Storm and S4 have a steady CPU utilization while Hadoop have idle, and CPU burst periods. For each job execution, Hadoop wastes about 15 s for job setup. The most notable point which shows the weakness of Hadoop is the average time needed to process each page. For Hadoop, the average time is about 3.8 min for each page while for Storm the average time is 57 ms, and for S4 is 64 ms. This means that while S4 and Storm finish analysis of each page in less than 65 ms, Hadoop may take at least 2.5 min to analyze a webpage after it is fetched.

We can say that S4 is a little slower than Storm but expressing the program was simpler for S4. During this experiment, we shut down one node suddenly to test the reliability of frameworks. Hadoop and Storm did not miss any pages, but S4 lost some pages when a node failed. This shows that Storm is more reliable than S4. We should mention that, to date, S4 has not been updated for more than a year. On the other hand, Storm has a very active community and seems to be more popular.

6. Conclusions

Big data has become a trend and some solutions have been provided for management and processing big data. The most popular solutions are MapReduce-based solutions and among them the Apache Hadoop framework is the most well known. However, Hadoop is inherently designed for batch and high throughput job execution and it is suitable for jobs that process large volumes of data over a long time. In contrast, there are also new demands like interactive jobs, real-time queries, and stream data that cannot be handled efficiently by batch-based frameworks like Hadoop. These non-batch demands have caused the creation of new solutions. In this article we discussed two categories: real-time processing, and streaming big data.

In the real-time processing sector, there are two major solutions: in-memory computing, and real-time queries over big data. In-memory computing uses a distributed memory storage that can be used either as a standalone input source or as a caching layer for disk-based storages. In particular, when the input totally fits in distributed memory or when the job has multiple iterations over input, in-memory computing can significantly reduce execution time. Solutions to real-time querying over big data mostly use custom storage formats and well-known techniques from parallel DBMSs to join and aggregation, and hence can respond to queries in less than a few seconds. In the stream-processing sector, there are two popular frameworks: Storm, and S4. Each one has its own programming model, strengths and weaknesses. We discussed both frameworks and their superiority to MapReduce-based systems for stream processing.

We believe that, solutions to batch and high throughput processing of big data, like Hadoop, have reached to an acceptable maturity level. However, they are not suitable enough for non-batch requirements. Considering high demands for interactive queries and big data streams, in-memory computing stands out as a notable solution that can handle both real-time and stream requirements. Among discussed frameworks, Spark is a good example for this case which supports in-memory computing using RDDs, real-time and interactive querying using Shark, and stream processing using fast micro-batching. However, the future will tell which approach will be popular in practice.

Acknowledgements

The author would like to thank Saeed Jalili for reviewing and commenting on this paper.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Jacobs, A. The pathologies of big data. *ACM Commun.* **2009**, *52*, 36–44.
2. Wu, X.; Zhu, X.; Wu, G.-Q.; Ding, W. Data mining with big data. *Knowl. IEEE Trans. Data Eng.* **2014**, *26*, 97–107.
3. Fernández, A.; del Río, S.; López, V.; Bawakid, A.; del Jesus, M.J.; Benítez, J.M.; Herrera, F. Big Data with Cloud Computing: An Insight on the Computing Environment, MapReduce and Programming Frameworks. *WIREs Data Min. Knowl. Discov.* **2014**, doi:10.1002/widm.1134.

4. Dean, J.; Ghemawat, S. MapReduce: A flexible data processing tool. *ACM Commun.* **2010**, *53*, 72–77.
5. Ghemawat, S.; Gobioff, H.; Leung, S.-T. The Google file system. *ACM SIGOPS Oper. Syst. Rev.* **2003**, *37*, 29–43.
6. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* **2008**, *26*, doi:10.1145/1365815.1365816.
7. White, T. *Hadoop: The Definitive Guide*; O'Reilly Media: Sebastopol, CA, USA, 2012.
8. Agneeswaran, V. *Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives*; Pearson FT Press: Upper Saddle River, NJ, USA, 2014.
9. Kambatla, K.; Kollias, G.; Kumar, V.; Grama, A. Trends in big data analytics. *J. Parallel Distrib. Comput.* **2014**, *74*, 2561–2573.
10. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Symposium on Opearting Systems Design & Implementation, San Francisco, CA, USA, 6–8 December 2004.
11. Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.* **2007**, *41*, 59–72.
12. Afrati, F.N.; Borkar, V.; Carey, M.; Polyzotis, N.; Ullman, J.D. Map-Reduce extensions and recursive queries. In Proceedings of the 14th International Conference on Extending Database Technology, Uppsala, Sweden, 22–24 March 2011; pp. 1–8.
13. 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.
14. Bu, Y.; Howe, B.; Balazinska, M.; Ernst, M.D. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.* **2010**, *3*, 285–296.
15. 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.
16. 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.
17. 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.
18. Fang, W.; He, B.; Luo, Q.; Govindaraju, N.K. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *22*, 608–620.
19. Cheatham, T.; Fahmy, A.; Stefanescu, D.C.; Valiant, L.G. Bulk synchronous parallel computing—A paradigm for transportable software. In Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, Wailea, HI, USA, 3–6 January 1995; pp. 268–275.
20. Malewicz, G.; Austern, M.H.; Bik, A.J.C.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 International Conference on Management of Data, Indianapolis, Indiana, USA, 6–11 June 2010; pp. 135–146.

21. Low, Y.; Bickson, D.; Gonzalez, J.; Guestrin, C.; Kyrola, A.; Hellerstein, J.M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* **2012**, *5*, 716–727.
22. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA, 22–25 June 2010.
23. 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.
24. 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.
25. 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.
26. Hadian, A.; Shahrivari, S. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. *J. Supercomput.* **2014**, *69*, 845–863.
27. Neumeyer, L.; Robbins, B.; Nair, A.; Kesari, A. S4: Distributed stream computing platform. In Proceedings of IEEE International Conference on Data Mining Workshops (ICDMW), New South Wales, Sydney, 13 December 2010; pp. 170–177.
28. Storm Homepage. Available online: http://storm-project.net/ (accessed on 1 December 2013).
29. Berkeley AMP Lab, "Big Data Benchmark." Available online: https://amplab.cs.berkeley.edu/benchmark/ (accessed on 1 December 2013).