From in-disk to in-memory big data with Hadoop: Performance experiments with nucleotide sequence data

A. Radenski[1], **L. Ehwerhemuepha**[1], **and K. Anderson**[1]
[1]Schmid College of Science and Technology, Chapman University, Orange, California, U.S.A.

Abstract. *Apache's Hadoop, the de facto standard big data business analytics platform, has been increasingly used for big data projects in the sciences in general and in bioinformatics in particular. While the numerous strengths of Hadoop have been widely recognized, its deficiencies have been in the focus of constructive criticism. In particular, the inadequate run time efficiency of the original Hadoop MapReduce in-disk engine has driven an on-going transition to the more efficient Spark in-memory engine. It has been acknowledged that Spark has a pronounced efficiency edge over MapReduce; at the same time, strict performance comparisons and analysis are scarce. To help fill the relative void, we experimented with codon count algorithms on nucleotide sequence data. To do so, we measured the performance of a Spark codon count algorithm on the Amazon cloud platform and compared it to our earlier MapReduce algorithms: a basic codon count algorithm and an optimized "local in-memory aggregation" (or simply "local aggregation") algorithm. As expected, our experiments confirmed that in-memory codon count with Spark is much faster (about 15 times) than basic in-disk codon count with MapReduce. Surprisingly, however, in-memory codon-count with Spark remains about two times slower than optimized "local aggregation" codon count with MapReduce. This shows that properly optimized big data analysis with MapReduce can be faster than analysis with Spark, while working reliably with larger data sets that do not fit in memory. Our results can be beneficial to researchers and practitioners who need to choose a suitable big data execution model for their current needs.*

Keywords: Hadoop, MapReduce, Spark, codons, performance

1 Introduction

Big data is an informal term used to refer to data sets that cannot be stored and processed with widespread, off the shelf hardware and software systems. Big data has at least one of the following attributes: large volume, high velocity (very fast data) and significant variety (largely heterogeneous data) [17, 7, 14]. While there are no specific boundary delineating big data, the size of big data sets usually range from terabytes to exabytes. The size of data exchanged through telecommunication network in 1986, 1993, 2000 and 2007 were 281 petabytes, 471 petabytes, 2.2 exabytes and 65 exabytes respectively [37]. However, in 2012, 2.5 exabytes of data were created daily (this is more than all data generated in 2000) and doubling every 40 months [7].

The interest in big data lies in the opportunity to reveal hidden, non-obvious knowledge that can be derived through data mining, statistical analysis, machine learning and other suitable methods in business, science and engineering, healthcare, and virtually any realm of human activity.

Although various big data tools and platforms have been developed over the years, we chose to focus on the Apache Hadoop Ecosystem because it is currently the most widely known and used platform for storage and analysis of big data.

The origins of Hadoop can be traced back to the early 2000s with the development of a MapReduce engine and distributed file system for the Nutch web crawler. Nutch grew into Hadoop which in 2008 was elevated to a top-level Apache project [34, 1].

Historically, Hadoop was built for batch processing of large textual data across multiple commodity servers. It grew out of the need to process, in fault-tolerant manner, voluminous text data across commodity hardware in such a way that computation is moved to the data and I/O latency from moving large chunks of data is consequently averted. Since its development, Hadoop has been considered a low cost, scalable, flexible, and fault-tolerant alternative for batch processing large data sets. By its original design, Hadoop was intended to address mainly the "large volume" aspect of big data.

We refer to MapReduce as *in-disk* engine because it involves the file system in all communications, including those on the same node. While this provides fault tolerance and scalability, it is detrimental to performance. The performance deficiency of MapReduce has stimulated the development of Spark, a faster alternative to MapReduce [38, 28]. We refer to the Spark as *in-memory* engine because it uses, in contrast to MapReduce, as much as possible the entire available memory for data storage and communication.

The big data community has accepted that Spark has a pronounced efficiency edge over Hadoop. At the same time, strict performance comparisons and analysis are scarce. To provide additional systematic insight, we experimented with codon count algorithms on nucleotide sequence data. In particular, we measured the execution times of an in-memory (with Spark) algorithm on the Amazon cloud platform and compared them to the execution times of a basic and an optimized in-disk (with MapReduce) algorithms.

In the rest of this paper, we review the Hadoop ecosystem, including its in-disk and in-memory aspects, then describe our experiments and compare the performance of Spark to the performance of MapReduce.

2 The Hadoop ecosystem

Since 2008, Hadoop and associated Apache projects have grown steadily to what is now referred to as the Hadoop Ecosystem. The Hadoop ecosystem comprises core projects and a number of related projects that can run on top or alongside of the core.

2.1 Core projects

The Hadoop core consists of the Hadoop Distributed File System (HDFS), the Hadoop MapReduce in-disk engine (MR), the Hadoop Common, and the YARN resource manager.

The *HDFS* is capable of storing very large data sets reliably and in a fault-tolerant way on potentially unreliable clusters of commodity servers. The HDFS design presumes that hardware failure is a norm; it also presumes that HDFS-based applications perform batch processing of large data sets, are write-once-read-many applications, are portable across heterogeneous commodity hardware, and are such that moving computation is cheaper than moving data [2]. The HDFS is not well suited for low-latency data access, data sets made of many small files, scenarios with multiple writers and arbitrary file modifications [19].

With *MR*, users specify serial map and reduce methods (one of each kind) that transform key-value records into new key-value records. The Hadoop MR implementation feeds input data to the mapper tasks and distributes intermediate key-value pairs to reducer tasks for final processing and output. In that, all intermediate records with the same key are distributed to the same reducer [24]. All communication in Hadoop MR goes via the file system.

The Hadoop *Common* contains libraries and utilities used by other Hadoop projects, while *YARN* is a resource negotiator that controls resource allocation within a cluster including application scheduling, also used by other projects. YARN was introduced to Apache Hadoop to decouple programming models from resource management and to delegate scheduling functions such as task fault tolerance to per-application components [33].

2.2 Related projects

The Hadoop ecosystem includes a variety of related projects that can interact with the Hadoop core, including the HDFS and the in-disk Hadoop MR engine.

Hive provides the ability to query and analyze large amount of historic (static) data in data warehouses by means of a SQL-like language, HiveQL. Hive translates higher-level user queries to lower-level Hadoop MR jobs, therefore freeing users from the relatively complex Hadoop MR Java API. Hive, however, is not a complete DBMS because record-level update, insert and delete operations cannot be performed directly by the underlying HDFS and MR. Hive is not well suited for rapidly changing data sets and is comparatively slower than traditional databases, partly because of the delay introduced by calling on and initiating MR jobs.

Pig is a high level programming language. The implementation translates Pig programs into Hadoop MR jobs.

HBase is a distributed, versioned, non-relational database modeled after, but not identical with Google's Bigtable [6, 10]. It can capture incremental data, such as user interactions in social networks and data produced by large cluster health monitors for example.

Presto is a distributed SQL engine for interactive big data analysis spanning from giga- to petabytes [23]. Presto accepts connections to different data sources (such as Hive, HBase, Cassandra, Scribe, relational databases, and proprietary data stores such as Amazon S3). Presto is optimized for ad-hoc analysis at interactive speed, supporting "standard ANSI SQL, including complex queries, aggregations, joins, and window functions" [4].

Mahout is a big data library of classification and recommendation algorithms. Originally implemented in Hadoop MR, Mahout is highly scalable and is able to support distributed processing of large data sets across commodity clusters.

Further examples include *Cassandra* (developed by Facebook in 2008 as an offshoot of BigTable), *Voldemort* (distributed key-value store created by LinkedIn in 2009), *Tajo* (SQL query engine for Hadoop created in South Korea in 2010), *Kafka* (data ingest framework originally developed by LinkedIn and open sourced in early 2011), *Storm* (stream computing framework released by Twitter in 2011), and *Impala* (SQL query engine created by Cloudera in 2012).

In recent years, *Spark* has gained popularity as an in-memory implementation of the map-reduce parallel model and is now emerging as a faster substitute for the original Hadoop MR in-disk engine. Because of Spark’s growing importance, we discuss it separately in a later section.

2.3 Deploying and running Hadoop

Hadoop can be deployed on traditional on-site clusters as well on public, private, and hybrid clouds. It can run on virtual machines where it is known to perform marginally slower than the physical machines [13].

Several companies provide Hadoop in public cloud services at a cost to users. Three notable examples are Amazon (Amazon Web Services), Microsoft (Microsoft Azure), and Google (Google App Engine). We have chosen to work with the Amazon Web Services (AWS) which is the largest cloud computing platform in the world and which provides AWS usage grants to universities for research and teaching.

Amazon Elastic MapReduce (Amazon EMR) is a web service that makes it easy to quickly and cost-effectively process vast amounts of data. Amazon EMR uses Hadoop to process data across clusters of desired (by users) sizes. Launching an Amazon EMR is a high-level task with node provisioning, cluster setup, Hadoop configuration and cluster tuning all abstracted from the user; this abstraction allows first time users ease of use. EMR is reliable and provides fault tolerance through automatic monitoring of the cluster to handle failed or failing nodes. It is "elastic" because any number of compute nodes can be provisioned and because it is easy to scale up or down and the user has complete control (such as root access) over the cluster. For security, EMR automatically configures firewall settings that control network access in a logically isolated user-defined network. A beneficial recent addition to AWS is Hue, a web-based graphical user interface for interactive access to AWS clusters (including EMR and HDFS) and the S3 cloud storage.

Cloudera provides an easy to install and configure pre-packaged distribution of Apache Hadoop, enhanced with custom Cloudera components. Other popular distributions include Apache Hadoop and MapR.

2.4 Hadoop deficiencies

There are, of course, drawbacks inherent in the original Hadoop architecture. Originally, Hadoop was specifically designed to run in a fault-tolerant manner long non-iterative batch jobs over large sets of static text data. Jobs over large numbers of small datasets can be inefficient. Interactive jobs and jobs that require data updates can be quite inefficient, too, if at all possible. The Hadoop MR engine employs the file system in all communications which can be detrimental to efficiency. These and other Hadoop difficulties are being addressed with new additions to the Hadoop ecosystem, most notably the Spark in-memory engine.

3 In-memory big data with Spark

Spark is a big data framework developed to take advantage of in-memory computation. Apart from increased speed, Spark provides support for cyclical data flow data model applications, thus eliminating another weakness of MapReduce [38].

Spark is a fast, general purpose engine for large-scale data processing that employs Resilient Distributed Datasets (RDD) and a distributed memory abstraction for in-memory computation on large clusters [28, 39]. Spark can be deployed as either a standalone application or on top of Hadoop. While Spark aims to fully utilize available memory, it also has the capacity to perform in-disk processing with larger data sets that do not fit entirely in available memory. Spark provides fault-tolerance through techniques that permit the restoration of RDDs upon node failure. Differences between MR and Spark are outlined in [9].

A Spark application involves the following principal components:

- Driver program written in Python, Java, or Scala.
- SparkContext object which is created within the driver program and coordinates the Spark processes running on the cluster. SparkContext connects to a supported cluster manager (such as YARN, Mesos, and Spark's own standalone cluster manager) and distributes tasks across worker nodes.
- Executor processes (referred to as executers) that carry out application-specific computations on the worker nodes under SparkContext.

The parallel computing primitives available in Spark include reduce, collect and foreach operations. Shared variables called broadcasters and accumulators are available to help with map, filter and reduce operations.

Spark is flexible framework as it can be run on Hadoop, Mesos, in the cloud or standalone, and can process a variety of data sources such as the HDFS, Cassandra, HBase and Amazon. Several organizations such as UC Berkeley AMPLab, Amazon, IBM Almaden and NASA JPL use Spark for building applications for large scale analytics and interactive exploration of large data [28].

Spark comes with higher level extensions for big data analytics, such as the SQL-like query extension, the MLlib machine learning extension, and the GraphX graph processing extension [28]. Most notably, the Spark Streaming extension was developed to process discretized data streams enabling real-time data analysis [38].

Given that Spark can handle static or slowly changing data as well as fast stream data, while at the same time maintaining a speed-up advantage over MR [39, 29, and 38], it is not surprising that Spark is considered as a viable in-memory alternative to the in-disk MR implementation. Projects, originally implemented with the in-disk MR engine have been ported (Hive) or are now being ported (Mahout) onto the Spark in-memory engine.

4 Hadoop bioinformatics applications

Since MapReduce was implemented as a module of the open-source Apache Hadoop platform, it has found application not only in business analytics, but also in various scientific and engineering domains, such as sets and graphs; artificial intelligence, machine learning and data mining; bioinformatics; image and video; evolutionary computing; a-life modeling, statistics; and numerical mathematics, including PDE solvers [26]. In this section we review some bioinformatics applications of the Hadoop ecosystem, including MR and Spark.

SparkSeq is a general-purpose genomic tool built with Apache Spark for next-generation sequencing (NGS) [35]. It provides convenient methods for common tasks in bioinformatics and genomic studies such as sample, exon and position encoding using Ensembl gene annotation forma [36]. The RSparkSeq package is available to connect SparkSeq with R. SparkSeq accepts BAM and BED [3, 18, and 31] files for a variety of analysis tasks such as nucleotide and genomic coverage, number of short reads, and others.

Adam is a tool for large scale genomics analysis that consists of data formats, APIs and algorithms implemented on top of Spark [20]. For a particular configuration, a “50 fold decrease in time required to compute base substitution was achieved in comparison to using BCF tools on a local file system” [32].

Error correction of erroneous bases is an important first (preprocessing) step, to precede genome assembly and variant discovery in high-throughput NGS. Apache Spark has been used to provide a parallel algorithm for screening NGS sequence data for errors, ensuring faster processing that Hadoop MapReduce [8].

SeqHBase was built on Hadoop and HBase for applications in whole-genome sequencing (WGS) and whole-exome sequencing (WES) [11]. Earlier on, the SeqWare Query Engine was developed in 2010 to provide an easy way to make the U87MG genome available to both skill programmers and non-programmers alike. The SeqWare Query Engine uses Apache HBase as the backend database because of its robust querying abilities and auto-sharding of data across commodity cluster via Hadoop [5].

BioPig is a sequence analysis toolkit built on the Hadoop MR engine and the Pig programming language. It contains modules such as pigKmer for computing the frequencies of each kmer as a histogram; pigDuster for searching known sequences for near exact match; and other modules such as pigDereplicator [22].

SeqPig is a similar scalable tool for sequencing large data sets in Hadoop. It uses Apache Pig for automated parallelization across computing nodes [27].

Cloud BioLinux is a publicly accessible VM that can be deployed on the Amazon EC cloud and on Eucalyptus. Cloud BioLinux provides, by default, a range of pre-configured “command line and graphical software applications, including a full-featured desktop interface and over 135 bioinformatics packages for sequence alignment, clustering, assembly and phylogeny” [16]. Cloud BioLinux has been used to analyze protein disorder for whole organisms and for obtaining all possible single sequence variants in protein coding regions of the human genome [15].

FX is an RNA sequence analysis tool that can be installed on local Hadoop clusters and Amazon EC2 cloud. FX provides an enhanced mapping of short reads by using references made of known genes and their isoforms [12].

Hadoop MR has been used for distributed a-life simulation on the cloud [25].

Additional bioinformatics applications of the Hadoop ecosystem can be found in [30, 24].

5 Performance experiments on AWS, the Amazon Cloud Platform

5.1 Algorithms on nucleotide sequence data

Codons are triples over the four DNA nucleotides traditionally represented by the letters A, C, G and T. Given a dataset of DNA nucleotides, codon usage calculation is expected to produce the frequencies of codons in the set [24]. The calculated frequencies can then be studied with various statistical methods for a number of purposes, such as back-translation of protein sequences to their probable DNA sequences, identification of protein-coding regions of DNA, and identification of regions that probably do not encode a protein [21].

In earlier research, we have developed MR streaming algorithms that gather statistics on codon usage count (both single codons and codon pairs). For each of these tasks, we developed a basic non-optimized MR algorithm and a MR algorithm optimized with local in-mapper aggregation or simply local aggregation (LA) [19, 24]. Local aggregation is a technique that helps reduce the intermediate data volume between mapper and reducer tasks in MR. For local aggregation, the mapper uses an in-memory data structure to aggregate multiple intermediate counts and emit them at once, rather than emitting multiple trivial counts. Thus, the mapper is explicitly designed to perform part of the reducer’s work.In this paper, we limit ourselves to single codon usage count. Pseudo-code for our basic MR algorithm is shown in Fig. 1, while Fig. 2 offers the pseudo-code for the optimized MR LA algorithm.

Then we evaluated the performance advantage of local aggregation by running our basic MR and MR LA algorithms on Amazon’s EMR cloud over sequence data from a tuberculosis database and by measuring the algorithms’ execution times [24].

```
1: class Mapper
2:    method Map ()
3:       for line ∈ stdin do
4:          codn-list = Parse (line)
5:          for each codon ∈ codon-list do
6:             Emit(codon, count=1)

1: class Reducer
2:    method Reduce ()
3:       previous ← None
4:       for line ∈ stdin do
5:          codon, count = Parse (line)
6:          if codon ≠ previous then
7:          if previous ≠ None then
8:                Emit(codon, count-total)
9:             previous = codon; count-total = 0
10:         Increment (count-total, count)
11:      if codon ≠ None then Emit(codon, count-total)
```

Figure 1: Pseudo-code for the basic codon count algorithm in MR streaming. The mapper emits an intermediate key-value pair for each codon occurrence; the reducer sums up all counts for each individual codon using the fact that keys (i.e. codons in this case) are supplied in sorted order.

In this paper, we revisit the codon usage count problem and run a basic Spark implementation (Fig. 3) on Amazon EMR clusters that are identically configured as in our previous experiments. We have also repeated representative set of our previous experiments to confirm that current cluster performance remains the same. This approach permitted us to compare Spark and MR performance by accumulating performance data for Spark and reusing previously accumulated data for MR.

```
1: class Mapper
2:    method Map ()
3:       codon-hash = ∅
4:       for line ∈ stdin do
5:          codon-list = Parse(line)
6:          for each codon ∈ codon-list do
7:             Increment (codon-hash[codon], 1)
8:       for codon ∈codon-hash do
9:          Emit(codon, count = codon-hash[codon])
```

Figure 2: Pseudo-code for the mapper part of the LA codon count algorithm in MR streaming. The mapper uses an in-memory data structure to accumulate partial counts, before finally emitting all of them. The reducer is the same as in Figure 1.

```
1: method getCodons (nucleotide_sequence):
2: codon_list = []
3: for each codon ϵ nucleotide do
4:    codon_list.append (codon)
5: return codon_list

6: file <- SparkContext.textFile ("path_to_data_on_S3")
7: codons <- file.flatMap (for line ϵ file: getCodons(line))
8: codon_count_1s <- codons.map (lambda codon: (codon, 1))

9: results <- codon_count_1s.reduceByKey (lambda x, y: x+y)
10: results.saveAsTextFile ("path_to_output_storage_on_S3")
```

Figure 3: Pseudo-code for the Spark codon count algorithm.

5.2 Performance experiments with nucleotide sequence data on the Amazon cloud platform

We implemented a Spark codon count program (Fig. 3) in Python 2.7.9. To assess performance, we ran our Spark implementation of codon count algorithm as a bootstrap program on the Elastic MR cloud with Hadoop 2.4.0. In all experiments, current with Spark and previous with MR, we used four m1.large AWS instances – a master instance and three core instances. This permitted us to compare performance results without rerunning earlier time-consuming MR experiments. Likewise, we used exactly the same gene sequence data in both experiments, replicating the data to scale from 1GB to 100GB of sequence data.

We uploaded the collection of datasets onto an AWS's S3 cloud. After provisioning a cluster, we SSH-ed into the cluster's master node and submitted our Spark application by using "spark-submit", setting "--master" to "yarn-client" and "--num-executors" to "6". We chose six executors after performing several experiments on 25GB of data to determine the optimal number of executors, performance wise. As in the experimental setup for our earlier MR jobs, we ran the Spark application twice and calculated the average execution time. All performance results are presented in Table 1 and visualized in Fig. 4 and Fig. 5.

As previously known [24], our optimized LA algorithm runs 16 to 34 times faster than our basic MR algorithm (Table 1). As expected, our Spark algorithm runs (about 15 times) faster than our basic MR algorithm. Somewhat surprisingly, our optimized LA MR algorithm performs two times faster than our Spark implementation, at least for the particular datasets used in these experiments (Table 1). We attribute the performance advantage of MR over Spark to the custom in-memory local aggregation optimization.

Table 1: Performance data (measured in minutes on AWS's EMR) for the (i) basic MapReduce (MR), (ii) MapReduce with local aggregation (MR LA), and (iii) Spark algorithms for codon count, plus speed gains of MR LA against Spark and MR, and of Spark against MR.

Elapsed time (in minutes)												
Data size (GB	Basic MR algorithm			MR LA algorithm			Spark algorithm			MR / MR LA	MR / Spark	Spark / MR LA
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg			
1	31	32	31.5	2	2	2.0	2.3	2.3	2.3	16	14	1.1
4	106	120	113.0	4.5	5	4.75	7.9	8.0	8.0	24	14	1.6
9	263	270	266.5	8	9	8.5	17.5	17.5	17.5	31	15	2.0
16	465	482	473.5	15	16	15.5	33.8	34.3	34.1	31	14	2.2
25	720	780	750.0	21	23	22.0	46.3	47.5	46.9	34	16	2.1
36	-	-	-	34	34.5	34.25	66.3	73.4	69.9	-	-	2.0
49	-	-	-	40	44	42.0	91.8	102.3	97.1	-	-	2.3
64	-	-	-	54	59	56.5	119.8	119.8	119.8	-	-	2.1
81	-	-	-	71	74	72.5	147.8	148.0	147.9	-	-	2.0
100	-	-	-	85	86	85.5	182.6	227.6	205.1	-	-	2.4

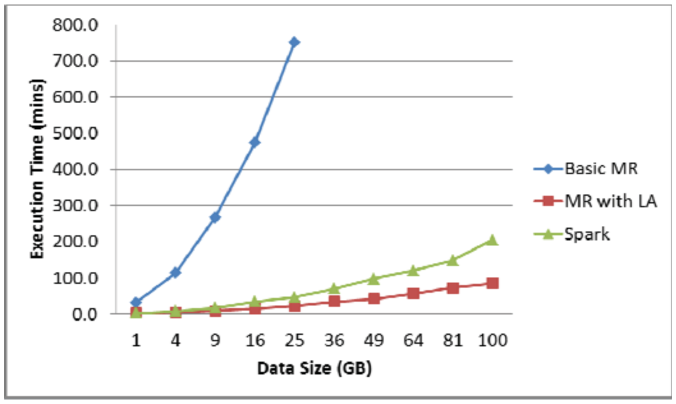

Figure 4: Execution times of the basic MR, the LA MR, and Spark algorithms for codon count.

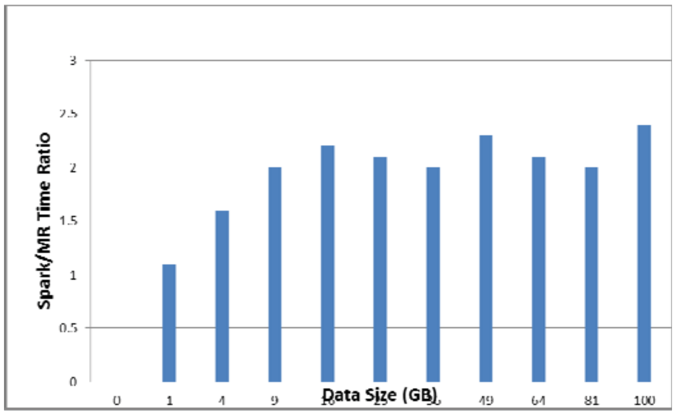

Figure 5: Performance advantage of the LA MR algorithm for codon count over the Spark algorithm.

6 Conclusions

In this paper, we analyzed the state of the art of the Hadoop big data echo-system and focused on the current trend from in-disk to in-memory big data processing. We reviewed and experimented with Spark, the emerging in-memory alternative to the classic Hadoop MapReduce framework.

To compare the performance of Spark and Hadoop MR, we executed, in Amazon's EMR cloud, Spark and Hadoop MR algorithms over large nucleotide sequence data. Specifically, we showed that optimized (with local aggregation) MR algorithm for simple codon analysis can be twice as fast as corresponding Spark algorithm; to the best of our knowledge, we the first to demonstrate the potential performance edge of MR due to custom LA. As expected, the performance of non-optimized MR lags behind Spark. We therefore suggest that available optimization techniques be considered for existing Hadoop MR applications before making a decision to re-implement them in Spark for performance gains.

We believe that this research can be beneficial to scholars and practitioners who need to choose a suitable big data execution model for their current needs.

7 References

1. Apache Hadoop, http://Hadoop.apache.org/ (Retrieved May 14, 2015).
2. Borthakur, D. (2007). The Hadoop distributed file system: Architecture and design. The Apache Software Foundation, (2007).
3. Broad Institute, https://www.broadinstitute.org/ (Retrieved May 14, 2015).
4. Chan L., Presto: Interacting with petabytes of data at Facebook, https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920 (Retrieved May 14, 2015).
5. D. O'Connor, B. Merriman, S. Nelson. SeqWare Query Engine: storing and searching sequence data in the cloud. Bmc Bioinformatics, 11(Suppl 12), S2, (2010).
6. George, L. (2011). HBase: the definitive guide. O'Reilly Media, Inc. (2011).
7. Gerhardt, B., Griffin, K., Klemann, R. (2012). Unlocking Value in the Fragmented World of Big Data Analytics. How Information Infomediaries Will Create a New Data Ecosystem, Cisco Internet Business Solutions Group (IBSG), 2013-2017 (2012).
8. Gong, Y. Parallel Algorithms for Screening NGS Data Using Spark (2014).
9. Gopalani, S., Arora R. Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means. International Journal of Computer Applications (0975 – 8887), Volume 113 – No. 1 (2015).
10. HBase, A. A Distributed Database for Large Datasets. The Apache Software Foundation, http://hbase.apache.org (Retrieved May 14, 2015).
11. He, M., Person, T. Hebbring, S., Heinzen, E., Ye, Z., Schrodi, S., Wang, K.. SeqHBase: a big data toolset for family based sequencing data analysis. Journal of medical genetics, jmedgenet-2014 (2014).
12. Hong, D., Rhie, A., Park, S., Lee, J., Ju, Y., Kim, S., Seo, J. S. FX: an RNA-Seq analysis tool on the cloud. Bioinformatics, 28(5), 721-723 (2012).
13. Ibrahim, S., Jin, H., Lu, L., Qi, L., Wu, S., & Shi, X. Evaluating mapreduce on virtual machines: The hadoop case." In Cloud Computing, 519-528, Springer Berlin Heidelberg (2009).
14. Intel IT Center. Planning Guide: Getting Started with Hadoop. Steps IT Managers Can Take to Move Forward with Big Data Analytics (2012).
15. Kaján, L., Yachdav, G., Vicedo, E., Steinegger, M., Mirdita, M., Angermüller, C., Rost, B. (2013). Cloud prediction of protein structure and function with PredictProtein for Debian. BioMed research international (2013).

16. Krampis, K., Booth, T., Chapman, B., Tiwari, B., Bicak, M., Field, D., Nelson, K. E. Cloud BioLinux: pre-configured and on-demand bioinformatics computing for the genomics community. BMC bioinformatics, 13(1), 42 (2012).
17. Laney, D. 3D data management: Controlling data volume, velocity and variety. META Group Research Note, 6 (2001).
18. Li H., Handsaker B., Wysoker A., Fennell T., Ruan J., Homer N., Marth G., Abecasis G., Durbin R. The Sequence alignment/map (SAM) format and SAMtools. Bioinformatics, 25, 2078-9 (2009).
19. Lin, J., Dyer, C. Data-intensive text processing with MapReduce. Synthesis Lectures on Human Language Technologies, 3(1), 1-177 (2010).
20. Massie, M., Nothaft, F., Hartl, C., Kozanitis, C., Schumacher, A., Joseph, A., Patterson, D. Adam: Genomics formats and processing patterns for cloud scale computing. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207 (2013).
21. McInerney, J. GCUA: general codon usage analysis, Bioinformatics 14(4), 372-373 (1998).
22. Nordberg, H., Bhatia, K., Wang, K., Wang, Z. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. Bioinformatics, 29(23), 3014-3019 (2013).
23. Presto, https://prestodb.io/ (Retrieved May 14, 2015).
24. Radenski, A., Ehwerhemuepha, L. Speeding-up codon analysis on the cloud with local MapReduce aggregation, Information Sciences, Elsevier, 263, 175-185 (2014).
25. Radenski, A. Using MapReduce streaming for distributed life simulation on the cloud, Advances in Artificial Life, ECAL 2013, MIT Press, 284-291 (2013).
26. Radenski, A. Big data, high-performance computing, and MapReduce. Proceedings of the 15th International Conference on Computer Systems and Technologies (CompSysTech '14), ACM, New York, NY, USA, 13-24 (2014).
27. Schumacher, A., Pireddu, L., Niemenmaa, M., Kallio, A., Korpelainen, E., Zanetti, G., Heljanko, K. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. Bioinformatics, 30(1), 119-120 (2014).
28. Spark, Spark: Lightning-fast cluster computing, https://spark.apache.org/ (Retrieved May 14, 2015).
29. Tabaa, Y., Medouri, A., & Tetouan, M. Towards a next generation of scientific computing in the cloud. International Journal of Computer Science (2012).
30. Taylor, R. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. BMC bioinformatics 11.Suppl 12 (2010): S1.
31. UCSC Genome Bioinformatics, http://genome.ucsc.edu/ (Retrieved May 14, 2015).
32. van Hagen, S., Schoots-van der Ploeg, J., Weistra, W., van Bochove, K. Evaluation of Spark and ADAM for large scale genomics data.
33. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Baldeschwieler, E. Apache Hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, p. 5, ACM (2013).
34. White, T. Hadoop: The definitive guide. O'Reilly Media, Inc. (2012).
35. Wiewiórka, M., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., Okoniewski, M. SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics, btu343 (2014).
36. Wiewiorka M. SparkSeq, https://bitbucket.org/mwiewiorka/sparkseq/wiki/Home (Retrieved May 14, 2015).
37. Wikipedia, Big data, http://en.wikipedia.org/wiki/Big_data (Retrieved May 14, 2015).
38. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I. Spark: cluster computing with working sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, p. 10 (2010).
39. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p. 2 (2012).