Data Locality Practices of MapReduce and Spark: Efficiency and Effectiveness

Literature Review and Related Sections

Alex Snow

November 17, 2017

Contents

1 Background

As data records made the transition from physical to digital, computer scientists sought to process large amounts of data to perform simple and complicated analysis on it. Apache Hadoop is an open-source software framework created for handling this "big data", officially released in 2011. Hadoop's software library includes Hadoop Distributed File System (HDFS), its data storage module, Hadoop YARN, its job scheduling and resource management module, and Hadoop MapReduce, its data processing module. In particular, MapReduce adapts sequential algorithms to run in parallel over large computer clusters, which can vastly improve runtime. MapReduce requires a partitioned dataset as input so that each piece of the dataset can be assigned to individual worker nodes, which execute map or reduce operations and report to the master node [W17].

MapReduce provides fault-tolerance, or the ability to recover lost data in the event of machine failure and scalability, but struggles with iterative computation, excessive recomputation, and high communication costs between nodes. Many of these problems stem from the protocol MapReduce follows for storing input data in relation to the computation unit, otherwise known as data locality practices. In light of these flaws, computer scientists strove to create alternatives to MapReduce that would address its weaknesses while still maintaining its strengths. The popular alternative is Spark, an open-source cluster computing framework created by UC Berkeley's AMPLab. Spark's developers in an early paper claim that their framework runs iterative jobs up to 10 times faster than MapReduce, yet still retains a similar level of fault-tolerance and scalability. They place most of the credit with their special read-only dataset type, called Resilient Distributed Datasets or RDDs for short. RDDs are created and divided between multiple machines and have a unique built-in lost data recovery system which provides fault-tolerance. RDDs allow Spark to also enjoy greater data locality, where input data is stored very close to the computation unit to reduce computation, because they can be saved in a computer's main physical memory or RAM. Spark was also built around Apache Mesos, a cluster management project that also serves as the base for MapReduce. As such, Spark can be used as a module of Hadoop, and is often packaged with MapReduce in the Hadoop software suite [ZCF10].

1.1 Motivation

Seeing as how many of MapReduce's and Spark's strengths and weaknesses originate from how they handle data locality, understanding that process would prove fruitful for any programmer. From compilers to small scripts, the fundamentals of data locality are universal, but how to apply the concepts is dependent upon the programming language and architecture [KM92].

In the age of "big data", its also understandable to want to process data as efficiently as possible. Spark and MapReduce enable data analysis to be done in parallel with essential fault-tolerance packaged behind a simple API. This accessibility and convenience is a main reason why the two cluster computing frameworks have become so popular [SQM+15]. From the perspective of an individual not knowledgeable on the subject, distributing both cluster computing frameworks together in Hadoop seems redundant, given AMPLab's claims. However, there do exist cases where MapReduce outperforms Spark. These performance differences between Spark and MapReduce stem mainly from differing approaches to data locality.

2 Literature Review

The literature review will address which data locality design and other design decisions give rise to Spark and MapReduces respective strengths and weaknesses. It will also present brief performance comparisons between the two cluster computing frameworks to show how their characteristics impact performance when executing various algorithms.

2.1 MapReduce

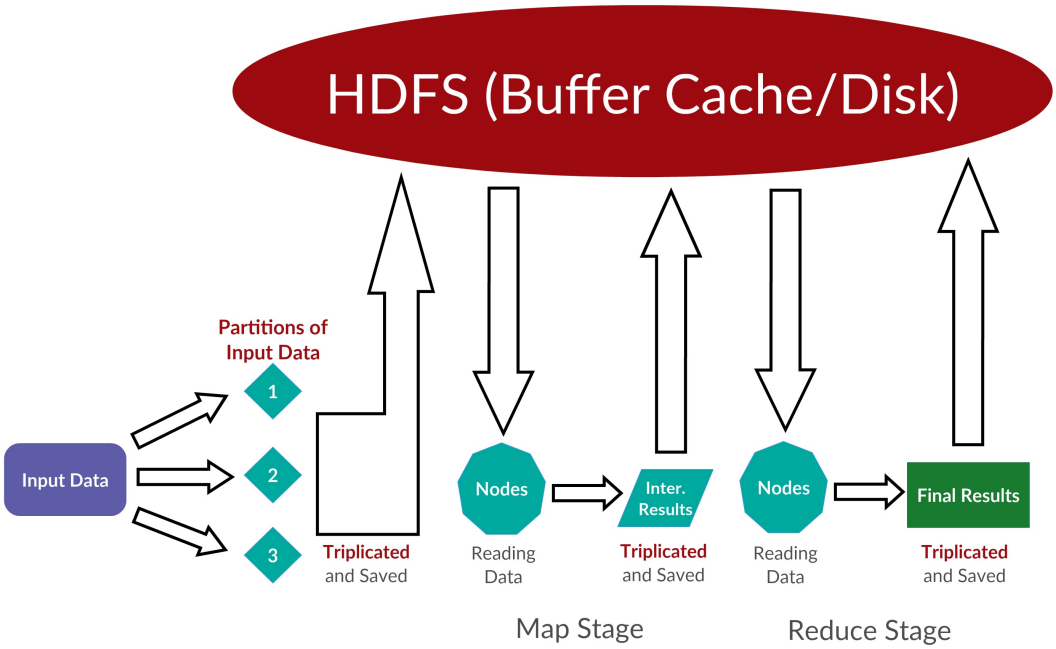

Figure 1: MapReduce Execution

Before executing a task, MapReduce first partitions the input data, copies them twice, and sends the data to HDFS to be saved. HDFS stores data it receives in the buffer cache or the hard disk, depending upon whether the data will fit in the buffer cache or not. The partitions are then distributed amongst the worker nodes, which apply the specified map function, then yield intermediate results. Next, the master node sorts the intermediate results by key and per user's request, combine the data to reduce communication costs between nodes and HDFS. The nodes then hand all restructured data to HDFS to save in preparation for the reduce stage. After being assigned a reduce task, the worker nodes read the intermediate data, pulling its partition from either the buffer cache or disk via shuffling. Following which, the nodes try to further minimize computation costs by externally merge-sorting the intermediate data by the intermediate keys before applying the reduce function. Later, the worker nodes run and complete their reduce tasks, which prompts the master node to triplicate the final results and store them using HDFS [LLC+11]. Worker nodes continually notify the master node if they are executing a task. If an assigned node

fails to communicate with the master node after a certain amount of time, it signifies machine failure. Following which, the master node hands the failed task to another available worker node and a copy of the task's input data is read from HDFS [DG08].

MapReduce's method of triplicating data at every section of execution guarantees quality fault-tolerance [LLC+11]. The constant checkpointing and simple scheduler also ensure improved scalability over other systems [LLC+11]. On the other hand, MapReduce does struggle with a few drawbacks because of its overall design and implementation. For example, MapReduce was designed with acyclic data flow in mind, as each task it runs must read the entirety of the input data before processing it. As such, the processing module struggles with iterative computation, which is centered around accessing and reusing previously read data [ZCF10]. MapReduce, as seen in figure 1, saves all intermediate and final results using HDFS, which stores the data in either the buffer cache or disk depending on the size of data for fault-tolerance purposes. However, since the framework cannot reuse output results, many copies of the same intermediate results are computed and saved by various worker nodes. Consequently, MapReduce suffers from excessive recomputation [DN14]. MapReduce, along with other cluster computing frameworks, sometimes assign tasks that a single node cannot handle within a certain time frame alone to multiple nodes. The downside to this decision is that running a task on multiple nodes requires communication between task fragments, replication of data and aggregation of output results [CGB+06]. In turn, the cost of communication between nodes significantly increases. This protocol along with others is why MapReduce receives its other significant criticism, its communication costs.

Parallel computing frameworks need a methodology of choosing when to store and process data with the framework's set of nodes. This methodology, known as job scheduling, plays a main role in communication costs in the system and efficiency of a parallel computing framework. MapReduce uses the Hadoop FIFO (first-in, first-out) scheduling algorithm by default, denoted as $dl - sched$, which focuses solely on good data locality, but forgoes system load and fairness. The algorithm takes advantage of the ability of the framework's nodes to handle both computation and storage, and strives to achieve "node-level" data locality, where worker nodes are assigned tasks that process input data the node already contains. If this cannot be done, the algorithm then tries to assign worker nodes jobs whose input data is on the same rack as that worker node to achieve "rack-level" data locality. In the event that rack-level data locality isn't feasible either, tasks are randomly assigned to worker nodes [GFZ12].

Common intuition dictates that the overall data locality of MapReduce using $dl - sched$ should improve proportionally with the number of nodes, but the relationship between data locality and number of nodes is not linear. Users can set the maximum number of tasks a single worker node can be assigned at any one time by setting a certain number of "task slots". As such, overall data locality improves proportional to the number of task slots each node has. However, data locality worsens proportional to the number of tasks yet to be done. Therefore, the total number of task slots should ideally be greater than the number of tasks to be done at all times [GFZ12].

2.2 Spark

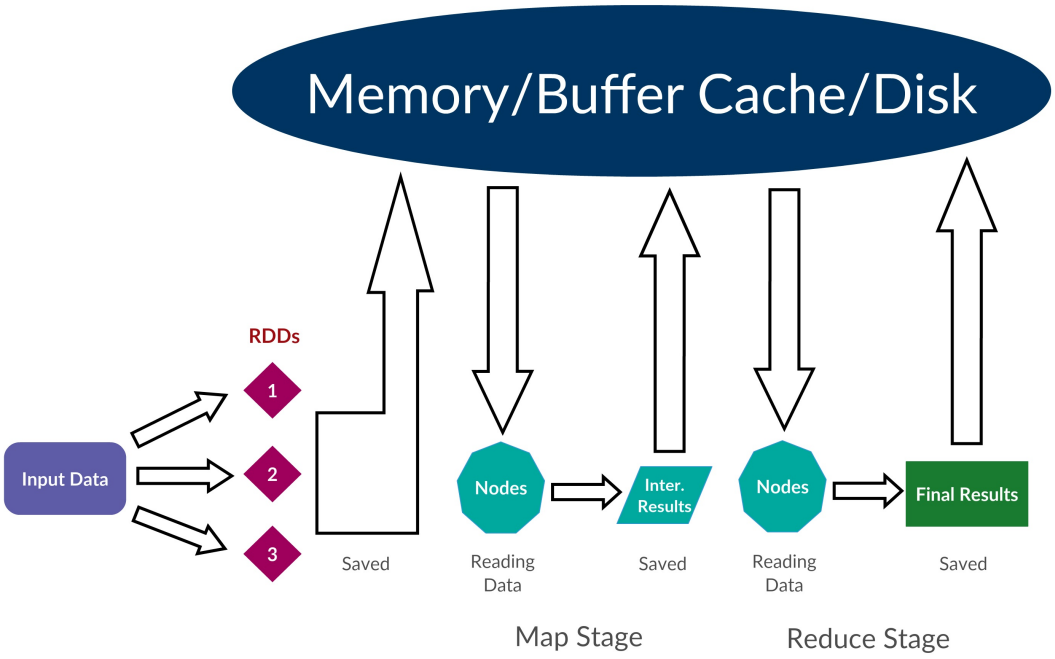

Figure 2: Spark Execution

The steps Spark takes to execute a task is quite similar to MapReduce's process. The key changes lie in how the two cluster computing frameworks manipulate the input data and where they store the data. Spark partitions the input data and

converts it into RDDs. Next, Spark saves the RDDs in memory, then in the buffer cache, then on disk if necessary and runs the same map stage and reduce stage as MapReduce, without the triplication of results.

Spark's central and most touted accomplishment is impressive performance executing iterative jobs, much faster than its predecessors. The key to their success, RDDs, are recomputed after each use if a cluster has enough memory to accommodate [ZCF10]. When there isn't enough space to store all RDDs into the buffer cache or memory, Spark saves the remaining data on disk, similarly to MapReduce [ZCD+Aug12]. This way, Spark attains good data locality on most partitions of its input data as opposed to great data locality for a few partitions and poor data locality for others [ZCF10].

The developers of Spark advertised the framework as having competent fault-tolerance, which is handled also through RDDs. Spark represents RDDs with five key pieces of information to differentiate one from another: its set of partitions, its set of parent RDD dependencies in the form of pointers to the parent and its transformation info, the transformation function used to compute itself, its partitioning strategy, and its data locality directives. Dependencies are classified as either narrow or wide dependencies. Narrow dependencies refer to partitions of the parent RDD used by only one partition of the child RDD. Wide dependencies on the other hand define partitions of the parent RDD used and shared by multiple RDD partition children. Narrow dependencies enable single cluster nodes to compute new RDD partitions, whereas wide dependencies require more nodes and overhead since pointers to be shuffled around between the nodes [ZCD+Apr12]. In the event that a computer crashes during a certain iteration of a job, Spark will use the pointers in the transformation function to recreate the lost RDD instead of recalculating the data from scratch [ZCD+Aug12]. As such, these transformations are lazy operations, operations executed only when necessary [ZCD+Apr12].

The developers claimed the fault-tolerance found in Spark rivals MapReduce, however this isn't always the case. The types of dependencies present in a lineage of RDDs directly influences the performance of Spark's fault-tolerance system. Recovering lost RDDs becomes increasingly more arduous if the RDD lineage involved contained many wide dependencies. In such a case, an RDD's transformation info must be passed to all affected nodes for recomputation in the event of machine failure and data loss [ZCD+Apr12]. This extra overhead hinders the system's performance, making it slower than MapReduce's fault-tolerance system in some situations. For example, when reduce tasks are unexpectedly ended, Spark suffers a much greater

slowdown in comparison to MapReduce. MapReduce only needs to re-execute the tasks, whereas Spark must re-execute the portion of map tasks which lost the block information pertaining to those lost reduce tasks. In addition, when tasks are unexpectedly killed during later iterations of a job, Spark's toted iterative algorithm performance suffers a sizeable hit from the RDD lineage system's extra overhead. All input objects must be completely parsed and initialized as RDDs again to find the lost RDD's parent transformation info to recreate the lost data, effectively re-executing the entire algorithm [SQM+15].

The job scheduling algorithm used in Spark assigns and executes jobs in a FIFO order similar to $dl - sched$. However, Spark's algorithm categorizes tasks into either the map or reduce phase before assigning them to nodes. The algorithm gives resource priority to all map phase tasks until every single one finishes, after which, the reduce phase tasks gain priority. Newer versions of Spark also contain a more fair scheduling algorithm, which distributes resources evenly amongst all tasks running. There is an option to implement a user-defined scheduling algorithm as well [JS17].

2.3 Performance Comparison

Juwei Shi et al. in [SQM+15] run performance tests on both MapReduce and Spark, comparing their executions of various algorithms. These performance comparisons focus on the architectural differences between Spark and MapReduce to illuminate each framework's strengths and weaknesses. In particular, the frameworks' shuffle component, execution model component, and caching component is examined closely. The shuffle component combines, sorts, and aggregates data, and majorly affects the scalability of any framework. The execution model component compiles execution plans from user-defined algorithms, and has a considerable amount of control over the resource management of a framework. The caching component saves intermediate data locally to be reused across the entire execution of an algorithm, and is a contributing factor to the overhead required to transfer data [SQM+15].

2.3.1 Word Count

A test is run comparing executions of Word Count, a simple algorithm that tallies the number of times each unique word appears in a string. For small and large datasets, Spark performs about 3x faster than MapReduce.

Spark outperforms MapReduce during the map stage due to its more efficient aggregation component. The component initializes tasks, reads input data, finishes map

operations, and combines data faster than MapReduce's aggregation component. Spark utilizes a hash-based combine method while MapReduce opts for a sort-based combine method, and in this test, the hash-based method handily defeats the sort-based method. On the other hand, Spark and MapReduce perform the reduce stage similarly and as such, have similar execution times.

2.3.2 Sorting

Juwei Shi et al. run another test comparing executions of sorting algorithms on a randomly generated dataset from *gensort* [GDG11]. MapReduce uses a TeraSort algorithm and Spark uses its built-in function, sortByKey(). Spark runs faster than MapReduce for small datasets of around 1 GB, but MapReduce surpasses Spark for larger datasets finishing around 1.8x faster than its competitor.

Spark outperforms MapReduce during the map stage thanks to its ability to reread the input via the OS buffer cache. As mentioned previously, MapReduce cannot reread data so it falls behind in this regard. Spark also utilizes a hash-based shuffle writer to save the map stage's results directly to disk as opposed to MapReduce, which collects the results in a side buffer before saving it to the disk. MapReduce however answers back in spades during the reduce stage, where it finishes the stage faster than Spark because MapReduce performs part of its map stage during the shuffle stage, thereby mitigating network overhead. Spark's shuffle and map stage unfortunately do not overlap. In addition, increasing the buffer size for Spark also increases the overhead for garbage collection and page swapping in the OS buffer cache, whereas an increase to the buffer size doesn't adversely affect MapReduce's performance.

2.3.3 K-Means Clustering

Moving into iterative algorithms, Juwei Shi et al. try running similar executions of K-Means algorithms and analyze the results. K-Means Clustering is a clustering algorithm that classifies the data points of a dataset by iteratively defining centers of clusters, and categorizes a point as part of a cluster that it's closest to. When testing the frameworks on a K-Means algorithm, Spark handily beats MapReduce in terms of speed and efficiency. In particular, Spark runs 1.5x faster on the first iteration, and 5x faster on subsequent iterations.

During the map stage, Spark vastly outperforms MapReduce thanks to RDD caching. By rereading the input data from memory each iteration, Spark avoids a consider-

able amount of disk overhead MapReduce has to handle. For the reduce stage, even though MapReduce must execute additional disk overhead, it performs similarly to Spark due to the test's low shuffle selectivity.

2.3.4 PageRank

Finally a comparison is done between MapReduce and Spark executing PageRank on Facebook and Twitter datasets, which contain millions of vertices and directed edges. PageRank is a graph algorithm that produces a ranking hierarchy of input elements based on the number and quality of links. Since PageRank is also an iterative algorithm, performance results of MapReduce and Spark were similar to the tests using K-Means clustering: Spark finishes execution much faster than MapReduce due to RDD caching.

3 Summary and Analysis

As seen in various evaluative papers and performance tests, Hadoop MapReduce has an equal distribution of strengths and weaknesses. MapReduce provides great fault-tolerance through its triplication of all results saved via HDFS. MapReduce also enjoys solid scalability thanks to its checkpointing and simple, static evaluation job scheduling algorithm $dl - sched$. However, the framework cannot reuse input data, resulting in poor performance executing iterative algorithms. As a consequence of ensuring good fault-tolerance and failing to reuse data, MapReduce also struggles with superfluous recomputation and costly communication between nodes.

MapReduce proved to be the cluster computing framework standard, as its advantages and drawbacks shaped development of the cluster computing field for years. By nature, computer scientists sought to improve and optimize MapReduce after recognizing its weaknesses. For example, Twister, HaLoop, and Spark are cluster computing frameworks created as alternatives to MapReduce whose biggest strength is MapReduce's greatest weakness: iterative computation [W17]. Whereas Twister and HaLoop give up some of MapReduce's advantages for iterative computation performance, the developers of Spark, the AMPLab at UC Berkeley, claimed that their framework executes iterative algorithms around ten times faster than MapReduce while still maintaining the fault-tolerance and scalability MapReduce is known for. They manage to achieve these feats through RDDs, which can be stored in memory and utilize a unique pointer-lineage system to recover lost partitions. With a new fierce competitor on the horizon, a new duality was formed in the cluster computing

field: one between MapReduce and Spark.

I discovered that Spark doesn't always outperform MapReduce, as shown in the performance tests in [SQM+15]. In particular, MapReduce finished the sorting test almost twice as fast as Spark because of Spark's reliance on the OS buffer cache to reuse input data. Furthermore, Spark was developed in Scala, a programming language considerably less popular than Java, the language MapReduce was created with. Spark also allows users to configure and customize the module to a higher degree compared to MapReduce, with its pre-packaged job scheduling and data locality design options [JS17].

In regards to data locality, data is optimally saved in the CPU registers, followed by the CPU cache, main physical memory, buffer cache, disk, and then remote disk. Considering that data storage spaces shrink as one moves closer and closer to the CPU, the CPU registers can hold less data than the CPU cache, which hold less than RAM, etc. Both MapReduce and Spark try to use the buffer cache then the hard disk to save data. However, before resorting to those locations, Spark tries to store its RDDs in memory, which is closer to the CPU and thus less computationally expensive to use. As such, Spark generally spends less time accessing and saving data than MapReduce does if Spark manages to write a sizable portion of data in memory. This along with MapReduce's inability to reread data and data triplication is the key to Spark's supremacy over MapReduce running iterative algorithms.

Consequently, I believe the performance bottleneck of Hadoop MapReduce is its use of HDFS. MapReduce's fault-tolerance system becomes its most computationally taxing set of operations in iterative algorithms because HDFS writes to either buffer cache or disk. If HDFS stored data in the CPU cache or RAM however, the data triplication required for the fault-tolerance system wouldn't impact MapReduce's performance as much. In addition, I believe the performance bottleneck of Spark is ironically its fault-tolerance system. In the event that a computer in the cluster crashes holding RDD wide dependencies, Spark suffers a major slowdown regenerating the dependency and redistributing the parent information to the dependents. As such, under ideal conditions (few/no machine failures, large memory pool, etc.), Spark will generally outperform MapReduce for most algorithms.

4 Open Questions

One topic of interest to be researched further stems from $dl - sched$ and its impact on both MapReduce's performnace. Some evidence shown in [GFZ12] indicates $dl - sched$ may not be optimal, especially during situations where the total number of task slots equals the number of tasks to be done. The algorithm also schedules tasks with an isolated purview, seeing only the available worker node and the set of tasks that still need to be completed, not the overall landscape of the worker nodes' availability. As an alternative, Guo, Fox, and Zhou in [GFZ12] suggest a scheduling algorithm that schedules all tasks at once as opposed to $dl - sched's$ FIFO approach. It would run a linear sum minimization on a matrix of assignment costs, where a low cost would equate to good data locality and inversely a high cost would equate to poor data locality. I wonder how MapReduce would stack up to Spark through the discussed performance tests given a new, optimal scheduling algorithm.

Another avenue of research to pursue further would be to expand the performance tests between MapReduce and Spark, comparing runs of Multidimensional scaling algorithms and SNP Genotyping with varying dataset sizes. Multidimensional scaling refers to an iterative, dimensionality reduction process where input data in a certain dimensional space is transposed into a lower dimensional space such that the distance between elements of the data is represented as accurately as possible. Data scientists often scale down distance matrices to visualize the data. SNP Genotyping is another computationally intensive batch job where single nucleotide polymorphisms (SNPs) in a strand of DNA are compared to a reference genome and labeled accordingly [W17]. Geneticists commonly use this process to check for genetic variations between members of an arbitrary but particular species. Adding another iterative test and acyclic data flow test to the performance comparison narrative would reinforce and cement Spark and MapReduce's role in the cluster computing field.

One other place to take the topic would be to draw conclusions juxtaposing Hadoop MapReduce and Spark with MapReduce and Spark utilizing a different software suite or no suite at all. In recent years, Apache, the software company responsible for developing the Hadoop suite, has moved their resources away from developing Hadoop further and into developing more powerful and less disk-dependent tools to apply map and reduce functions to datasets. This initiative falls in line with the remote, cloud-oriented path technology is currently walking down. Apache's latest project, titled Mahout, is an open source software library containing various scalable data mining and machine learning algorithms implemented with the MapReduce

paradigm in mind. Many of the algorithms require no Hadoop dependencies whatsoever, so contrasting executions of Hadoop-dependent and not Hadoop-dependent implementations of the same algorithm could prove insightful towards the efficiency of Hadoop itself [SO12].

5 References

(CGB+06) Fan Chung, Ronald Graham, Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Maximizing data locality in distributed systems. Journal of Computer and System Sciences, vol. 72, no. 8, pp. 1309-1316, Dec. 2006.

(DG08) Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM - 50th anniversary issue: 1958 - 2008, vol. 51, pp. 107-113, Jan. 2008.

(DN14) Christis Doulkeridis, Kjetil Nrvg. A survey of large-scale analytical query processing in MapReduce. The VLDB Journal, vol. 23, pp. 355380, Jun. 2014.

(GFZ12) Zhenhua Guo, Geoffrey Fox, Mo Zhou. Investigation of Data Locality in MapReduce. CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 419-426, May 2012.

(KM92) Ken Kennedy and Kathryn S. McKinley. Optimizing for Parallelism and Data Locality. ICS '92 Proceedings of the 6th international conference on Supercomputing, pp. 323-334, Aug. 1992.

(LLC+11) Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel Data Processing with MapReduce: A Survey. ACM SIGMOD Record, vol. 40, no. 4, pp. 1120, Dec. 2011.

(SO12) Sebastian Schelter and Sean Owen. Collaborative Filtering with Apache Mahout. Recommender Systems Challenge 2012 in conjunction with the ACM Conference on Recommender Systems, 2012.

(SQM+15) Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Ozcan. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. Proceedings of the VLDB Endowment - Proceedings of the 41st International Conference on Very Large Data Bases, Sept. 2015.

(W17) Daniel Washburn. Performance Competitiveness of MapReduce: Applications in Scientific Research. Institutional Scholarship, 2017.

(ZCD+Aug12) Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Fast and Interactive Analytics over Hadoop Data with Spark. ;login: The Usenix Magazine, vol. 37, no. 4, Aug. 2012.

(ZCD+Apr12) Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 25 Apr., 2012.

(ZCF10) Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. HotCloud'10 Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 22 Jun. 2010.

(JS17) "Job Scheduling." Job Scheduling - Spark 1.3.0 Documentation, spark.apache.org/docs/1.2.0/job-scheduling.html.

(GDG11) "Gensort Data Generator." Sort Benchmark Data Generator and Output Validator, 2011, www.ordinal.com/gensort.html. I learned a general explanation of *gensort* from this source.