Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics

Juwei Shi[‡], Yunjie Qiu[†], Umar Farooq Minhas[§], Limei Jiao[†], Chen Wang[♯], Berthold Reinwald[§], and Fatma Özcan[§] *
[†]IBM Research - China [§]IBM Almaden Research Center
[‡]DEKE, MOE and School of Information, Renmin University of China [♯]Tsinghua University

ABSTRACT

MapReduce and Spark are two very popular open source cluster computing frameworks for large scale data analytics. These frameworks hide the complexity of task parallelism and fault-tolerance, by exposing a simple programming API to users. In this paper, we evaluate the major architectural components in MapReduce and Spark frameworks including: shuffle, execution model, and caching, by using a set of important analytic workloads. To conduct a detailed analysis, we developed two profiling tools: (1) We correlate the task execution plan with the resource utilization for both MapReduce and Spark, and visually present this correlation; (2) We provide a break-down of the task execution time for in-depth analysis. Through detailed experiments, we quantify the performance differences between MapReduce and Spark. Furthermore, we attribute these performance differences to different components which are architected differently in the two frameworks. We further expose the source of these performance differences by using a set of micro-benchmark experiments. Overall, our experiments show that Spark is about 2.5x, 5x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank, respectively. The main causes of these speedups are the efficiency of the hash-based aggregation component for combine, as well as reduced CPU and disk overheads due to RDD caching in Spark. An exception to this is the Sort workload, for which MapReduce is 2x faster than Spark. We show that MapReduce's execution model is more efficient for shuffling data than Spark, thus making Sort run faster on MapReduce.

1. INTRODUCTION

In the past decade, open source analytic software running on commodity hardware made it easier to run jobs which previously used to be complex and tedious to run. Examples include: text analytics, log analytics, and SQL like query processing, running at a very large scale. The two most popular open source frameworks for such large scale data processing on commodity hardware are: MapReduce [7] and Spark [19]. These systems provide simple APIs, and hide the complexity of parallel task execution and fault-tolerance from the user.

1.1 Cluster Computing Architectures

MapReduce is one of the earliest and best known commodity cluster frameworks. MapReduce follows the functional programming model [8], and performs explicit synchronization across computational stages. MapReduce exposes a simple programming API in terms of `map()` and `reduce()` functions. Apache Hadoop [1] is a widely used open source implementation of MapReduce.

The simplicity of MapReduce is attractive for users, but the framework has several limitations. Applications such as machine learning and graph analytics iteratively process the data, which means multiple rounds of computation are performed on the same data. In MapReduce, every job reads its input data, processes it, and then writes it back to HDFS. For the next job to consume the output of a previously run job, it has to repeat the read, process, and write cycle. For iterative algorithms, which want to read once, and iterate over the data many times, the MapReduce model poses a significant overhead. To overcome the above limitations of MapReduce, Spark [19] uses Resilient Distributed Datasets (RDDs) [19] which implement in-memory data structures used to cache intermediate data across a set of nodes. Since RDDs can be kept in memory, algorithms can iterate over RDD data many times very efficiently.

Although MapReduce is designed for batch jobs, it is widely used for iterative jobs. On the other hand, Spark has been designed mainly for iterative jobs, but it is also used for batch jobs. This is because the new big data architecture brings multiple frameworks together working on the same data, which is already stored in HDFS [17]. We choose to compare these two frameworks due to their wide spread adoption in big data analytics. All the major Hadoop vendors such as IBM, Cloudera, Hortonworks, and MapR bundle both MapReduce and Spark with their Hadoop distributions.

1.2 Key Architectural Components

In this paper, we conduct a detailed analysis to understand how Spark and MapReduce process batch and iterative jobs, and what architectural components play a key role for each type of job. In particular, we (1) explain the behavior of a set of important analytic workloads which are typically run on MapReduce and Spark, (2) quantify the performance differences between the two frameworks, (3) attribute these performance differences to the differences in their architectural components.

We identify the following three architectural components and evaluate them through detailed experiments. Studying these components covers the majority of architectural differences between MapReduce and Spark.

*This work has been done while Juwei Shi and Chen Wang were Research Staff Members at IBM Research-China.

Proceedings of the VLDB Endowment, Vol. 8, No. 13

Shuffle: The shuffle component is responsible for exchanging intermediate data between two computational stages [1]. For example, in the case of MapReduce, data is shuffled between the map stage and the reduce stage for bulk synchronization. The shuffle component often affects the scalability of a framework. Very frequently, a sort operation is executed during the shuffle stage. An external sorting algorithm, such as merge sort, is often required to handle very large data that does not fit in main memory. Furthermore, aggregation and combine are often performed during a shuffle.

Execution Model: The execution model component determines how user defined functions are translated into a physical execution plan. The execution model often affects the resource utilization for parallel task execution. In particular, we are interested in (1) parallelism among tasks, (2) overlap of computational stages, and (3) data pipelining among computational stages.

Caching: The caching component allows reuse of intermediate data across multiple stages. Effective caching speeds up iterative algorithms at the cost of additional space in memory or on disk. In this study, we evaluate the effectiveness of caching available at different levels including OS buffer cache, HDFS caching [3], Tachyon [11], and RDD caching.

For our experiments, we use five workloads including Word Count, Sort, k-means, linear regression, and PageRank. We choose these workloads because collectively they cover the important characteristics of analytic workloads which are typically run on MapReduce and Spark, and they stress the key architectural components we are interested in, and hence are important to study. Word Count is used to evaluate the aggregation component because the size of intermediate data can be significantly reduced by the map side combiner. Sort is used to evaluate the external sort, data transfer, and the overlap between map and reduce stages because the size of intermediate data is large for sort. K-Means and PageRank are used to evaluate the effectiveness of caching since they are both iterative algorithms. We believe that the conclusions which we draw from these workloads running on MapReduce and Spark can be generalized to other workloads with similar characteristics, and thus are valuable.

1.3 Profiling Tools

To help us quantify the differences in the above architectural components between MapReduce and Spark, as well as the behavior of a set of important analytic workloads on both frameworks, we developed the following tools for this study.

Execution Plan Visualization: To understand a physical execution plan, and the corresponding resource utilization behavior, we correlate the task level execution plan with the resource utilization, and visually present this correlation, for both MapReduce and Spark.

Fine-grained Time Break-down: To understand where time goes for the key components, we add timers to the Spark source code to provide the fine-grained execution time break-down. For MapReduce, we get this time break-down by extracting this information from the task level logs available in the MapReduce framework.

1.4 Contributions

The key contributions of this paper are as follows. (1) We conduct experiments to thoroughly understand how MapReduce and Spark solve a set of important analytic workloads including both batch and iterative jobs. (2) We dissect MapReduce and Spark frameworks and collect statistics from detailed timers to quantify differences in their shuffle component. (3) Through a detailed analysis of the execution plan with the corresponding resource utilization, we attribute performance differences to differences in major architectural components for the two frameworks. (4) We conduct micro-benchmark experiments to further explain non-trivial observations regarding RDD caching.

The rest of the paper is organized as follows. We provide the workload description in Section 2. In Section 3, we present our experimental results along with a detailed analysis. Finally, we present a discussion and summary of our findings in Section 4.

[1]For MapReduce, there are two stages: map and reduce. For Spark, there may be many stages, which are built at shuffle dependencies.

2. WORKLOAD DESCRIPTION

In this section, we identify a set of important analytic workloads including Word Count (WC), Sort, k-means, linear regression (LR), and PageRank.

Table 1: Characteristics of Selected Workloads

		Word Count	Sort	K-Means (LR)	Page-Rank
Type	One Pass	√	√		
	Iterative			√	√
Shuffle Sel.	High		√		
	Medium				√
	Low	√		√	
Job/Iter. Sel.	High		√		
	Medium				√
	Low	√		√	

As shown in Table 1, the selected workloads collectively cover the characteristics of typical batch and iterative analytic applications run on MapReduce and Spark. We evaluate both one-pass and iterative jobs. For each type of job, we cover different shuffle selectivity (i.e., the ratio of the map output size to the job input size, which represents the amount of disk and network I/O for a shuffle), job selectivity (i.e., the ratio of the reduce output size to the job input size, which represents the amount of HDFS writes), and iteration selectivity (i.e., the ratio of the output size to the input size for each iteration, which represents the amount of intermediate data exchanged across iterations). For each workload, given the I/O behavior represented by these selectivities, we evaluate its system behavior (e.g., CPU-bound, disk-bound, network-bound) to further identify the architectural differences between MapReduce and Spark.

Furthermore, we use these workloads to quantitatively evaluate different aspects of key architectural components including (1) shuffle, (2) execution model, and (3) caching. As shown in Table 2, for the shuffle component, we evaluate the aggregation framework, external sort, and transfers of intermediate data. For the execution model component, we evaluate how user defined functions are translated into to a physical execution plan, with a focus on task parallelism, stage overlap, and data pipelining. For the caching component, we evaluate the effectiveness of caching available at different levels for caching both input and intermediate data. As explained in Section 1.2, the selected workloads collectively cover all the characteristics required to evaluate these three components.

3. EXPERIMENTS

3.1 Experimental Setup

3.1.1 Hardware Configuration

Our Spark and MapReduce clusters are deployed on the same hardware, with a total of four servers. Each node has 32 CPU cores at 2.9 GHz, 9 disk drives at 7.2k RPM with 1 TB each, and 190

(a) MapReduce (b) Spark

Figure 1: The Execution Details of Word Count (40 GB)

Table 2: Key Architectural Components of Interest

		Word Count	Sort	K-Means (LR)	Page-Rank
Shuffle	Aggregation	√		√	√
	External sort		√		
	Data transfer		√		√
Execution	Task parallelism	√	√	√	√
	Stage overlap		√		
	Data pipelining				√
Caching	Input			√	√
	Intermediate data				√

GB of physical memory. In aggregate, our four node cluster has 128 CPU cores, 760 GB RAM, and 36 TB locally attached storage. The hard disks deliver an aggregate bandwidth of about 125 GB/sec for reads and 45 GB/sec for writes across all nodes, as measured using `dd`. Nodes are connected using a 1 Gbps Ethernet switch. Each node runs 64-bit Red Hat Enterprise Linux 6.4 (kernel version 2.6.32).

As a comparison, hardware specifications of our physical cluster are roughly equivalent to a cluster with about 100 virtual machines (VMs) (e.g., m3.medium on AWS[2]). Our hardware setup is suitable for evaluating various scalability bottlenecks in Spark and MapReduce frameworks. For example, we have enough physical cores in the system to run many concurrent tasks and thus expose any synchronization overheads. However, experiments which may need a large number of servers (e.g., evaluating the scalability of master nodes) are out of the scope of this paper.

3.1.2 Software Configuration

Both MapReduce and Spark are deployed on Java 1.7.0.

Hadoop: We use Hadoop version 2.4.0 to run MapReduce on YARN [17]. We use 8 disks to store intermediate data for MapReduce and for storing HDFS data as well. We configure HDFS with 128 MB block size and a replication factor of 3. We configure Hadoop to run 32 containers per node (i.e., one per CPU core). To better control the degree of parallelism for jobs, we enable CGroups in YARN and also enable CPU-Scheduling [3]. The default JVM heap size is set to 3 GB per task. We tune the following parameters to optimize performance: (1) We use Snappy compression for map output; (2) For all workloads except Sort, we disable the overlap between map and reduce stages. This overlap hides the network overhead by overlapping computation and network transfer. But it comes at a cost of reduction in map parallelism, and the network overhead is not a bottleneck for any workload except Sort; (3) For Sort, we set the number of reduce tasks to 60 to overlap the shuffle stage (network-bound) with the map stage (without network overhead), and for other workloads, we set the number of reduce tasks to 120; (4) We set the number of parallel copiers per task to 2 to reduce the context switching overhead [16]; (5) We set the map output buffer to 550 MB to avoid additional spills for sorting the map output; (6) For Sort, we set the reduce input buffer to 75% of the JVM heap size to reduce the disk overhead caused by spills.

Spark: We use Spark version 1.3.0 running in the standalone mode on HDFS 2.4.0. We also use 8 disks for storing Spark intermediate data. For each node, we run 8 Spark workers where each worker is configured with 4 threads (i.e., one thread per CPU core). We also tested other configurations. We found that when we run 1 worker with 32 threads, the CPU utilization is significantly reduced. For example, under this configuration, the CPU utilization decreases to 33% and 25%, for Word Count and the first iteration of k-means, respectively. This may be caused by the synchronization overhead of memory allocation for multi-threaded tasks. However, CPU can be 100% utilized for all the CPU-bound workloads when using 8 workers with 4 threads each. Thus, we use this setting for our experiments. We set the JVM heap size to 16 GB for both the Spark driver and the executors. Furthermore, we tune the following parameters for Spark: (1) We use Snappy to compress the intermediate results during a shuffle; (2) For Sort with 500 GB input, we set the number of tasks for shuffle reads to 2000. For other workloads, this parameter is set to 120.

3.1.3 Profiling tools

In this section, we present the visualization and profiling tools which we have developed to perform an in-depth analysis of the selected workloads running on Spark and MapReduce.

Execution Plan Visualization: To understand parallel execution behavior, we visualize task level execution plans for both MapReduce and Spark. First, we extract the execution time of tasks from job history of MapReduce and Spark. In order to create a compact view to show parallelism among tasks, we group tasks to horizontal

[2]http://aws.amazon.com/ec2/instance-types/
[3]https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

Table 3: Overall Results: Word Count

Platform	Spark	MR	Spark	MR	Spark	MR
Input size (GB)	1	1	40	40	200	200
Number of map tasks	9	9	360	360	1800	1800
Number of reduce tasks	8	8	120	120	120	120
Job time (Sec)	30	64	70	180	232	630
Median time of map tasks (Sec)	6	34	9	40	9	40
Median time of reduce tasks (Sec)	4	4	8	15	33	50
Map Output on disk (GB)	0.03	0.015	1.15	0.7	5.8	3.5

lines. We deploy Ganglia [13] over the cluster, and persist Round-Robin Database [5] to MySQL database periodically. Finally, we correlate the system resource utilization (CPU, memory, disk, and network) with the execution plan of tasks using a time line view.

Figure 1 is an example of such an execution plan visualization. At the very top, we present a visualization of the physical execution plan where each task is represented as an horizontal line. The length of this line is proportional to the actual execution time of a task. Tasks belonging to different stages are represented in different colors. The physical execution plan can then be visually correlated with the CPU, memory, network, and disk resources, which are also presented in Figure 1. In the resource utilization graphs, x-axis presents the elapsed time in seconds, and is correlated with the horizontal axis of the physical execution plan. Moreover, we break down reduce tasks for MapReduce to three sub-stages: copy, sort, and reduce. For each batch of simultaneously running task (i.e., wave), we can read directly down to correlate resources with the processing that occurs during that wave.

This tool can show: (1) parallelism of tasks (i.e., waves), e.g., the first wave is marked in Figure 1 (a), (2) the overlap of stages, e.g., the overlap between map and reduce stages is shown in Figure 2 (b), (3) skewness of tasks, e.g., the skewness of map tasks is shown in Figure 3 (b), and (4) the resource usage behavior for each stage, e.g., we can see that the map stage is CPU-bound in Figure 3 (a). Note that the goal of our visualization tool is to help a user analyze details of the physical execution plan, and the corresponding resource utilization, in order to gain deeper insights into a workload's behavior.

Fine-grained Time Break-down: To understand where time goes for the shuffle component, we provide the fine-grained execution time break-down for selected tasks. For Spark, we use `System.nanoTime()` to add timers to each sub-stage, and aggregate the time after a task finishes. In particular, we add timers to the following components: (1) `compute()` method for RDD transformation, (2) `ShuffleWriter` for combine, and (3) `BlockObjectWriter` for serialization, compression, and shuffle writes. For MapReduce, we use the task logs to provide the execution time break-down. We find that such detailed break-downs are sufficient to quantify differences in the shuffle components of MapReduce and Spark.

3.2 Word Count

We use Word Count (WC) to evaluate the aggregation component for both MapReduce and Spark. For these experiments, we use the example WC program included with both MapReduce and Spark, and we use Hadoop's random text writer to generate input.

3.2.1 Overall Result

Table 3 presents the overall results for WC for various input sizes, for both Spark and MapReduce. Spark is about 2.1x, 2.6x, and 2.7x faster than MapReduce for 1 GB, 40 GB, and 200 GB input, respectively.

Interestingly, Spark is about 3x faster than MapReduce in the map stage. For both frameworks, the application logic and the amount of intermediate data is similar. We believe that this difference is due to the differences in the aggregation component. We evaluate this observation further in Section 3.2.3 below.

Table 4: Time Break-down of Map Tasks for Word Count

Platform	Load (sec)	Read (sec)	Map (sec)	Combine (sec)	Serialization (sec)	Compression &write (sec)
Spark	0.1	2.6	1.8	2.3	2.6	0.1
MapReduce	6.2	12.6		14.3	5.0	

For the reduce stage, the execution time is very similar in Spark and MapReduce because the reduce stage is network-bound and the amount of data to shuffle is similar in both cases.

3.2.2 Execution Details

Figure 1 shows the detailed execution plan for WC with 40 GB input. Our deployment of both MapReduce and Spark can execute 128 tasks in parallel, and each task processes 128 MB of data. Therefore, it takes three waves of map tasks (shown in blue in Figure 1) to process the 40 GB input. As we show the execution time when the first task starts on the cluster, there may be some initialization lag on worker nodes (e.g., 0 to 4 seconds in Figure 1 (b)).

Map Stage: We observe that in Spark, the map stage is disk-bound while in MapReduce it is CPU-bound. As each task processes the same amount of data (i.e., 128 MB), this indicates that Spark takes less CPU time than MapReduce in the map stage. This is consistent with the 3x speed-up shown in Table 3.

Reduce Stage: The network resource utilization in Figure 1 shows that the reduce stage is network-bound for both Spark and MapReduce. However, the reduce stage is not a bottleneck for WC because (1) most of the computation is done during the map side combine, and (2) the shuffle selectivity is low ($< 2\%$), which means that reduce tasks have less data to process.

3.2.3 Breakdown for the Map Stage

In order to explain the 3x performance difference during the map stage, we present the execution time break-down for map tasks in both Spark and MapReduce in Table 4. The reported execution times are an average over 360 map tasks with 40 GB input to WC. First, MapReduce is much slower than Spark in task initialization. Second, Spark is about 2.9x faster than MapReduce in input read and map operations. Last, Spark is about 6.2x faster than MapReduce in the combine stage. This is because the hash-based combine is more efficient than the sort-based combine for WC. Spark has lower complexity in its in-memory collection and combine components, and thus is faster than MapReduce.

3.2.4 Summary of Insights

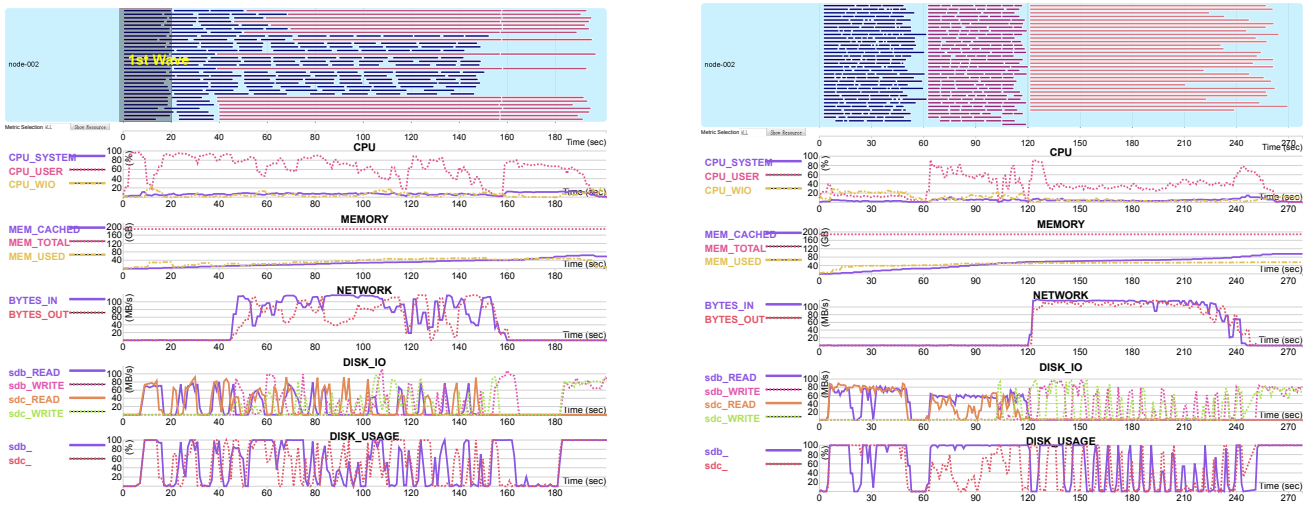
For WC and similar workloads such as descriptive statistics, the shuffle selectivity can be significantly reduced by using a map side combiner. For this type of workloads, hash-based aggregation in Spark is more efficient than sort-based aggregation in MapReduce due to the complexity differences in its in-memory collection and combine components.

3.3 Sort

For experiments with Sort, we use TeraSort [15] for MapReduce, and implement Sort using `sortByKey()` for Spark. We use `gensort` [4] to generate the input for both.

We use experiments with Sort to analyze the architecture of the shuffle component in MapReduce and Spark. The shuffle component is used by Sort to get a total order on the input and is a bottleneck for this workload.

[4] http://www.ordinal.com/gensort.html



(a) MapReduce (b) Spark

Figure 2: The Execution Details of Sort (100 GB Input)

Table 5: Overall Results: Sort

Platform	Spark	MR	Spark	MR	Spark	MR
Input size (GB)	1	1	100	100	500	500
Number of map tasks	9	9	745	745	4000	4000
Number of reduce tasks	8	8	248	60	2000	60
Job time	32s	35s	4.8m	3.3m	44m	24m
Sampling stage time	3s	1s	1.1m	1s	5.2m	1s
Map stage time	7s	11s	1.0m	2.5m	12m	13.9m
Reduce stage time	11s	24s	2.5m	45s	26m	9.2m
Map output on disk (GB)	0.63	0.44	62.9	41.3	317.0	227.2

3.3.1 *Overall Result*

Table 5 presents the overall results for Sort for 1 GB, 100 GB, and 500 GB input, for both Spark and MapReduce. For 1 GB input, Spark is faster than MapReduce because Spark has lower control overhead (e.g., task load time) than MapReduce. MapReduce is 1.5x and 1.8x faster than Spark for 100 GB and 500 GB inputs, respectively. Note that the results presented in [6] show that Spark outperformed MapReduce in the Daytona Gray Sort benchmark. This difference is mainly because our cluster is connected using 1 Gbps Ethernet, as compared to a 10 Gbps Ethernet in [6], i.e., in our cluster configuration network can become a bottleneck for Sort in Spark, as explained in Section 3.3.2 below.

From Table 5 we can see that for the sampling stage, MapReduce is much faster than Spark because of the following reason: MapReduce reads a small portion of the input file (100,000 records from 10 selected splits), while Spark scans the whole file. For the map stage, Spark is 2.5x and 1.2x faster than MapReduce for 100 GB and 500 GB input, respectively. For the reduce stage, MapReduce is 3.3x and 2.8x faster than Spark for 100 GB and 500 GB input, respectively. To better explain this difference, we present a detailed analysis of the execution plan in Section 3.3.2 and a break-down of the execution time in Section 3.3.3 below.

3.3.2 *Execution Details*

Figure 2 shows the detailed execution plan of 100 GB Sort for MapReduce and Spark along with the resource utilization graphs.

Sampling Stage: The sampling stage of MapReduce is performed by a lightweight central program in less than 1 second, so it is not shown in the execution plan. Figure 2 (b) shows that during the initial part of execution (i.e., sampling stage) in Spark, the disk utilization is quite high while the CPU utilization is low. As we mentioned earlier, Spark scans the whole input file during sampling and is therefore disk-bound.

Table 6: Time Break-down of Map Tasks for Sort

Platform	Load (sec)	Read (sec)	Map (sec)	Combine (sec)	Serialization (sec)	Compression &write (sec)
Spark-Hash	0.1	1.1	0.8	-	3.0	5.5
Spark-Sort	0.1	1.2	0.6	6.4	2.3	2.4
MapReduce	6.6	10.5		4.1	2.0	

Map Stage: As shown in Figure 2, both Spark and MapReduce are CPU-bound in the map stage. Note that the second stage is the map stage for Spark. Even though Spark and MapReduce use different shuffle frameworks, their map stages are bounded by map output compression. Furthermore, for Spark, we observe that disk I/O is significantly reduced in the map stage compared to the sampling stage, although its map stage also scans the whole input file. The reduced disk I/O is a result of reading input file blocks cached in the OS buffer during the sampling stage.

Reduce Stage: The reduce stage in both Spark and MapReduce uses external sort to get a total ordering on the shuffled map output. MapReduce is 2.8x faster than Spark for this stage. As the execution plan for MapReduce in Figure 2 (a) shows, the main cause of this speed-up is that the shuffle stage is overlapped with the map stage, which hides the network overhead. The current implementation of Spark does not support the overlap between shuffle write and read stages. This is a notable architectural difference between MapReduce and Spark. Spark may want to support this overlap in the future to improve performance. Last, note that the number of replicas in this experiment is set to 1 according to the sort benchmark [15], thus there is no network overhead for HDFS writes in reduce tasks.

When the input size increases from 100 GB to 500 GB, during the map stage in Spark, there is significant CPU overhead for swapping pages in OS buffer cache. However, for MapReduce, we observe much less system CPU overhead during the map stage. This is the main reason that the map stage speed-up between Spark and MapReduce is reduced from 2.5x to 1.2x.

3.3.3 *Breakdown for the Map Stage*

Table 6 shows a break-down of the map task execution time for both MapReduce and Spark, with 100 GB input. A total of 745 map tasks are executed, and we present the average execution time.

We find that there are two stages where MapReduce is slower than Spark. First, the load time in MapReduce is much slower than that in Spark. Second, the total times of (1) reading the input (Read), and (2) for applying the map function on the input (Map), is higher than Spark. The reasons why Spark performs better include: (1) Spark reads part of the input from the OS buffer cache since its sampling stage scans the whole input file. On the other hand, MapReduce only partially reads the input file during sampling thus OS buffer cache is not very effective during the map stage. (2) MapReduce collects the map output in a map side buffer before flushing it to disk, but Spark's hash-based shuffle writer, writes each map output record directly to disk, which reduces latency.

3.3.4 Comparison of Shuffle Components

Since Sort is dominated by the shuffle stage, we evaluate different shuffle frameworks including hash-based shuffle (Spark-Hash), sort-based shuffle (Spark-Sort), and MapReduce.

First, we find that the execution time of the map stage increases as we increase the number of reduce tasks, for both Spark-Hash and Spark-Sort. This is because of the increased overhead for handling opened files and the commit operation of disk writes. As opposed to Spark, the number of reduce tasks has little effect on the execution time of the map stage for MapReduce.

The number of reduce tasks has no affect on the execution time of Spark's reduce stage. However, for MapReduce, the execution time of the reduce stage increases as more reduce tasks are used because less map output can be copied in parallel with the map stage as the number of reduce tasks increases.

Second, we evaluate the impact of buffer sizes for both Spark and MapReduce. For both MapReduce and Spark, when the buffer size increases, the reduced disk spills cannot lead to the reduction in the execution time since disk I/O is not a bottleneck. However, the increased buffer size may lead to slow-down in Spark due to the increased overhead for GC and page swapping in OS buffer cache.

3.3.5 Summary of Insights

For Sort and similar workloads such as Nutch Indexing and TFIDF [10], the shuffle selectivity is high. For this type of workloads, we summarize our insights from Sort experiments as follows: (1) In MapReduce, the reduce stage is faster than Spark because MapReduce can overlap the shuffle stage with the map stage, which effectively hides the network overhead. (2) In Spark, the execution time of the map stage increases as the number of reduce tasks increase. This overhead is caused by and is proportional to the number of files opened simultaneously. (3) For both MapReduce and Spark, the reduction of disk spills during the shuffle stage may not lead to the speed-up since disk I/O is not a bottleneck. However, for Spark, the increased buffer may lead to the slow-down because of increased overhead for GC and OS page swapping.

3.4 Iterative Algorithms: K-Means and Linear Regression

K-Means is a popular clustering algorithm which partitions N observations into K clusters in which each observation belongs to the cluster with the nearest mean. We use the generator from HiBench [10] to generate training data for k-means. Each training record (point) has 20 dimensions. We use Mahout [2] k-means for MapReduce, and the k-means program from the example package for Spark. We revised the Mahout code to use the same initial centroids and convergence condition for both MapReduce and Spark.

K-Means is representative of iterative machine learning algorithms. For each iteration, it reads the training data to calculate updated parameters (i.e., centroids). This pattern for parameter optimization covers a large set of iterative machine learning algorithms such as linear regression, logistic regression, and support vector machine. Therefore, the observations we draw from the results presented in this section for k-means are generally applicable to the above mentioned algorithms.

Table 7: Overall Results: K-Means

Platform	Spark	MR	Spark	MR	Spark	MR
Input size (million records)	1	1	200	200	1000	1000
Iteration time 1st	13s	20s	1.6m	2.3m	8.4m	9.4m
Iteration time Subseq.	3s	20s	26s	2.3m	2.1m	10.6m
Median map task time 1st	11s	19s	15s	46s	15s	46s
Median reduce task time 1st	1s	1s	1s	1s	8s	1s
Median map task time Subseq.	2s	19s	4s	46s	4s	50s
Median reduce task time Subseq.	1s	1s	1s	1s	3s	1s
Cached input data (GB)	0.2	-	41.0	-	204.9	-

As shown in Table 1, both the shuffle selectivity and the iteration selectivity of k-means is low. Suppose there are N input records to train K centroids, both map output (for each task) and job output only have K records . Since K is often much smaller than N, both the shuffle and iteration selectivities are very small. As shown in Table 2, the training data can be cached in-memory for subsequent iterations. This is common in machine learning algorithms. Therefore, we use k-means to evaluate the caching component.

3.4.1 Overall Result

Table 7 presents the overall results for k-means for various input sizes, for both Spark and MapReduce. For the first iteration, Spark is about 1.5x faster than MapReduce. For subsequent iterations, because of RDD caching, Spark is more than 5x faster than MapReduce for all input sizes.

We compare the performance of various RDD caching options (i.e., storage levels) available in Spark, and we find that the effectiveness of all the persistence mechanisms is almost the same for this workload. We explain the reason in Section 3.4.3.

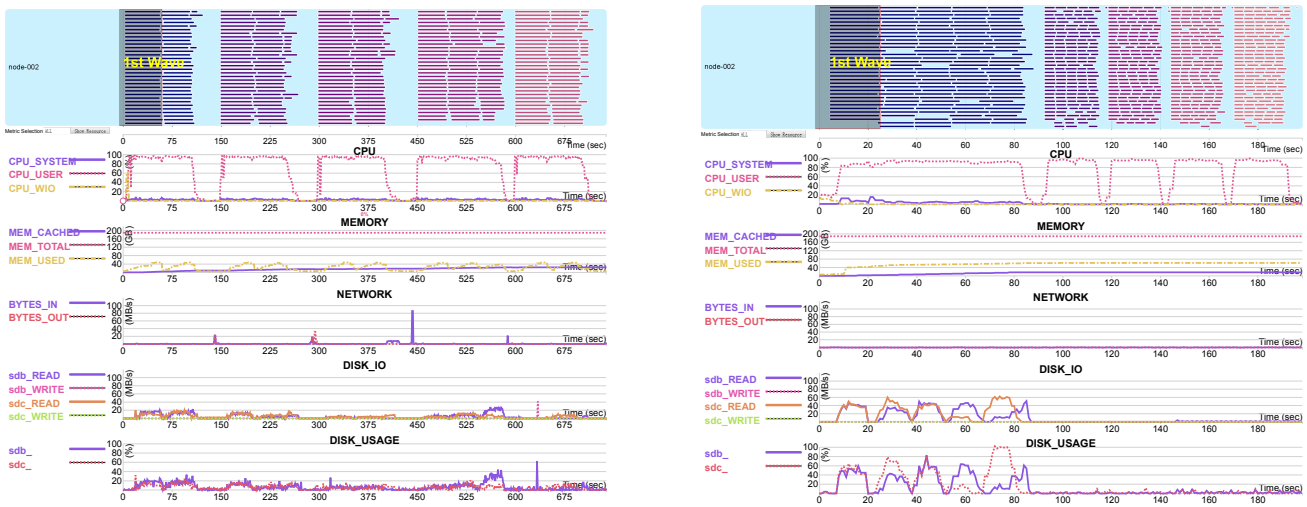
Another interesting observation is that when there is no in-heap caching (i.e., for Spark without RDD caching and MapReduce), the disk I/O decreases from one iteration to the next iteration because of the increased hit ratio of OS buffer cache for input from HDFS. However, this does not result in a reduction in the execution time. We explain these observations in Section 3.4.4.

3.4.2 Execution Details

Figure 3 shows the detailed execution plan for k-means with an input of 200 million records for MapReduce and Spark. We present the first five iterations. Subsequent iterations exhibit similar characteristics.

Map Stage: For both MapReduce and Spark, the map stage takes up more than 99% of the total execution time in each iteration and hence is the bottleneck. During the map stage, both MapReduce and Spark scan the training data to update the cluster centroids. For MapReduce (Mahout k-means), it uses a map side hash table to combine the updated value of the centroid. For Spark, it uses a map side combiner (implemented in `reduceByKey()`) to update centroids. The shuffle selectivity is very small. For example, with 200 million records, the selectivity is 0.00001 and 0.004, for MapReduce and Spark, respectively.

The map stage is CPU-bound for all iterations of both MapReduce and Spark due to the low shuffle selectivity. Moreover, for MapReduce, we observe that the disk read overhead decreases from one iteration to the next iteration. Since there is no in-heap caching in MapReduce framework, it depends on OS buffer to cache the training data. However, OS buffer cache does not result in execution time improvements, as we show later in Section 3.4.4. As



(a) MapReduce (b) Spark

Figure 3: The Execution Details of K-Means (200 Million Records)

Table 8: The Impact of Storage Levels for K-Means

Storage Levels	Caches Size	First Iterations	Subsequent Iterations
NONE	-	1.3m	1.2m
DISK_ONLY	37.6 GB	1.5m	29s
DISK_ONLY_2	37.6 GB	1.9m	27s
MEMORY_ONLY	41.0 GB	1.5m	26s
MEMORY_ONLY_2	41.0 GB	2.0m	25s
MEMORY_ONLY_SER	37.6 GB	1.5m	29s
MEMORY_ONLY_SER_2	37.6 GB	1.9m	28s
MEMORY_AND_DISK	41.0 GB	1.5m	26s
MEMORY_AND_DISK_2	41.0 GB	2.0m	25s
MEMORY_AND_DISK_SER	37.6 GB	1.5m	29s
MEMORY_AND_DISK_SER_2	37.6 GB	1.9m	27s
OFF_HEAP (Tachyon)	37.6 GB	1.5m	30s

Table 9: The Impact of Memory Constraints

Storage Levels	Fraction	Persistence ratio	First Iterations	Subsequent Iterations
MEMORY_ONLY	0	0.0%	1.4m	1.2m
MEMORY_ONLY	0.1	15.6%	1.5m	1.2m
MEMORY_ONLY	0.2	40.7%	1.5m	1.0m
MEMORY_ONLY	0.3	67.2%	1.6m	47s
MEMORY_ONLY	0.4	98.9%	1.5m	33s
MEMORY_ONLY	0.5	100%	1.6m	30s
MEMORY_AND_DISK	0	100%	1.5m	28s
MEMORY_AND_DISK	0.1	100%	1.5m	30s
MEMORY_AND_DISK	0.2	100%	1.6m	30s
MEMORY_AND_DISK	0.3	100%	1.6m	28s
MEMORY_AND_DISK	0.4	100%	1.5m	28s
MEMORY_AND_DISK	0.5	100%	1.5m	28s
DISK_ONLY	-	100%	1.5m	29s

opposed to MapReduce, there is no disk read overhead in subsequent iterations for Spark, because the input is stored in memory as RDDs. Note that the 3.5x speed-up for subsequent iterations in Spark cannot be fully attributed to the reduced disk I/O overhead. We explain this in Section 3.4.4.

Reduce Stage: Both MapReduce and Spark use a map-side combiner. Therefore, the reduce stage is not a bottleneck for both frameworks. The network overhead is low due to the low shuffle selectivity. Furthermore, there is no disk overhead for Spark during the reduce stage since it aggregates the updated centroids in Spark's driver program memory. Even though MapReduce writes to HDFS for updating the centroids, the reduce stage is not a bottleneck due to the low iteration selectivity.

Furthermore, we observe the same resource usage behavior for the 1 billion data set when the input data does not fit into memory. The only difference is the increased system CPU time for page swapping at the end of the last wave during the map stage. But this does not change the overall workload behavior.

3.4.3 Impact of RDD caching

RDD caching is one of the most notable features of Spark, and is missing in MapReduce. K-means is a typical iterative machine learning algorithm that can benefit from RDD caching. In the first iteration, it parses each line of a text file to a Point object, and persists the Point objects as RDDs in the storage layer. In subsequent iterations, it repeatedly calculates the updated centroids based on cached RDDs. To understand the RDD caching component, we evaluate the impact of various storage levels on the effectiveness of RDD caching. We also evaluate the impact of memory constraints on the effectiveness of RDD caching.

Impact of Storage Levels: The version of Spark we evaluate provides 11 storage levels including MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, and Tachyon [11]. For memory only related levels, we can choose to serialize the object before storing it. For all storage levels except Tachyon, we can also choose to replicate each partition on two cluster nodes (e.g., DISK_ONLY_2 means to persist RDDs on disks of two cluster nodes).

Table 8 presents the impact of storage levels on the execution time of first iterations, subsequent iterations, and the size of RDD caches. Surprisingly, the execution time of subsequent iterations is almost the same regardless of whether RDDs are cached in memory, on disk, or in Tachyon. We explain this in Section 3.4.4. In addition, replication of partitions leads to about 30% increase in the execution time of first iterations, because of the additional network overhead.

Finally, we observe that the raw text file is 69.4 GB on HDFS. The size of RDDs for the Point object is 41.0 GB in memory, which reduces to 37.6 GB after serialization. Serialization is a trade-off between CPU time for serialization/de-serialization and the space in-memory/on-disk for RDD caching. Table 8 shows that RDD caching without serialization is about 1.1x faster than that with serialization. This is because k-means is already CPU-bound, and serialization results in additional CPU overhead.

Impact of Memory Constraints: For MEMORY_ONLY, the number of cached RDD partitions depends on the fraction of memory

allocated to the `MemoryStore`, which is configurable. For other storage levels, when the size of `MemoryStore` exceeds the configured limit, RDDs which have to be persisted will be stored on disk. Table 9 presents the impact of memory constraints on the persistence ratio (i.e., the number of cached partitions / the number of all partitions), execution time of first iterations, and execution time of subsequent iterations. The persistence ratio of MEMORY_ONLY decreases as less memory is allocated for `MemoryStore`. Moreover, the persistence ratio rather than the storage level is the main factor that affects the execution time of subsequent iterations. This is consistent with the results shown in Table 8. From these results, we conclude that there is no significant performance difference when using different storage levels for k-means.

3.4.4 What is the bottleneck for k-means?

From the experiments, we make several non-trivial observations for k-means: (a) The storage levels do not impact the execution time of subsequent iterations. (b) For DISK_ONLY, there are almost no disk reads in subsequent iterations. (c) When there is no RDD caching, disk reads decrease from one iteration to the next iteration, but this does not lead to execution time improvements.

To explain these observations and further understand RDD caching in Spark, we conduct the following micro-benchmark experiments. We use DISK_ONLY as the baseline. (1) We set the fraction of Java heap for `MemoryStore` to 0 MB. Compared to the baseline, there is no difference in execution time and memory consumption. This means that DISK_ONLY does not store any RDDs in Java heap. (2) We reduce the number of disks for storing intermediate data (i.e., RDD caching) from 8 to 1. The execution time is still the same as the baseline, but we observe that the disk utilization increases by about 8x on the retained disk compared to the baseline. This means that disks are far from 100% utilized when we have 8 disks to store intermediate data. (3) We reduce memory of executors from 32 GB to 200 MB. The execution time is 17% slower compared to the baseline. This is because of the increased GC overhead. Note that there is still no disk read in subsequent iterations. (4) We drop OS buffer cache after the first iteration. We observe that the execution time of the subsequent iteration is increased from 29 seconds to 53 seconds. Moreover, we find heavy disk reads after OS buffer cache is dropped. The retained disk is 100% utilized, and 80% of CPU time becomes iowait time. This means that RDDs are cached in pages of the OS buffer after the first iteration when we use DISK_ONLY. (5) To further evaluate the performance of DISK_ONLY RDD caching without OS buffer cache, we restore 8 disks to store intermediate results. We also drop OS buffer cache after the first iteration. We observe that the execution time of the subsequent iteration is reduced from 53 seconds to 29 seconds. There are still disk reads after OS buffer cache is dropped, but user CPU time is restored to 100%. This means that when all disks are restored for RDD caching, disk reads are no longer the bottleneck even without OS buffer cache. With 8 disks, the aggregate disk bandwidth is more than enough to sustain the IO rate for k-means, with or without OS buffer cache.

However, these results lead to the following additional observations: (d) When there is no RDD caching, OS buffer cache does not improve the execution time. (e) In the case without RDD caching, disk reads decrease faster than with DISK_ONLY RDD caching. To explain (c), (d), and (e), we design another set of micro-benchmark experiments to detect the bottleneck for iterations which read input from HDFS. We first design a micro-benchmark to minimize the CPU overhead to evaluate the behavior of HadoopRDD (An RDD that provides core functionality for reading data stored in Hadoop). The HadoopRDD is scanned and persisted in the first iteration, then the cached RDDs are scanned with `count()` in subsequent iterations. For NONE (i.e., without RDD caching), we observe that the execution time decreases in subsequent iterations as more blocks from HDFS are cached in the OS buffer. As opposed to k-means, the OS buffer cache reduces the execution time of subsequent iterations, since disk reads is the bottleneck for this micro-benchmark. Further, because of data locality, the hit ratio of OS buffer cache is about 100% in the second iteration for DISK_ONLY, but that ratio is about only 30% for NONE. When HDFS caching [3] is enabled, the execution time of subsequent iterations decreases as more replicas are stored in the HDFS cache. From the above experiments, we established that the OS buffer cache improves the execution time of subsequent iterations if disk I/O is a bottleneck.

Next, we change a single line in k-means code to persist HadoopRDD before parsing lines to Point objects. We observe that the execution time of subsequent iterations is increased from 27 and 29 seconds to 1.4 and 1.3 minutes, for MEMORY_ONLY and DISK_ONLY, respectively. Moreover, the disk utilization of subsequent iterations is the same as k-means and HadoopRDD scan. This indicates that the CPU overhead of parsing each line to the Point object is the bottleneck for the first iteration that reads input from HDFS. Therefore, for k-means without RDD caching, the reduction of disk I/O due to OS buffer cache does not result in execution time improvements for subsequent iterations, since the CPU overhead of transforming the text to Point object is a bottleneck.

3.4.5 Linear Regression

We also evaluated linear regression with maximum $1000000 * 50000$ training records (372 GB), for both MapReduce and Spark. We observed the same behavior as k-means. Thus, we do not repeat those details here. The only difference is that linear regression requires larger executor memory as compared to k-means. Therefore, the total size of the OS buffer cache and the JVM heap exceeds the memory limit on each node. This leads to about 30% system CPU overhead for OS buffer cache swapping. This CPU overhead can be eliminated by using DISK_ONLY RDD caching which reduces memory consumption.

3.4.6 Summary of Insights

A large set of iterative machine learning algorithms such as k-means, linear regression, and logistic regression read the training data to calculate a small set of parameters iteratively. For this type of workloads, we summarize our insights from k-means as follows: (1) For iterative algorithms, if an iteration is CPU-bound, caching the raw file (to reduce disk I/O) may not help reduce the execution time since the disk I/O is hidden by the CPU overhead. But on the contrary, if an iteration is disk-bound, caching the raw file can significantly reduce the execution time. (2) RDD caching can reduce not only disk I/O, but also the CPU overhead since it can cache any intermediate data during the analytic pipeline. For example, the main contribution of RDD caching for k-means is to cache the Point object to save the transformation cost from a text line, which is the bottleneck for each iteration. (3) If OS buffer is sufficient, the hit ratio of both OS buffer cache and HDFS caching for the training data set increases from one iteration to the next iteration, because of the replica locality from previous iterations.

3.5 PageRank

PageRank is a graph algorithm which ranks elements by counting the number and quality of links. To evaluate PageRank algorithm on MapReduce and Spark, we use Facebook [5] and Twitter [6]

[5] http://current.cs.ucsb.edu/facebook/index.html
[6] http://an.kaist.ac.kr/traces/WWW2010.html

Table 10: Overall Results: PageRank

Platform	Spark-Naive	Spark-GraphX	MR	Spark-Naive	Spark-GraphX	MR
Input (million edges)	17.6	17.6	17.6	1470	1470	1470
Pre-processing	24s	28s	93s	7.3m	2.6m	8.0m
1st Iter.	4s	4s	43s	3.1m	37s	9.3m
Subsequent Iter.	1s	2s	43s	2.0m	29s	9.3m
Shuffle data	73.1MB	69.4MB	141MB	8.4GB	5.5GB	21.5GB

data sets. The interaction graph for Facebook data set has 3.1 million vertices and 17.6 million directed edges (219.4 MB). For Twitter data set, the interaction graph has 41.7 million vertices and 1.47 billion directed edges (24.4 GB). We use X-RIME PageRank [18] for MapReduce. We use both PageRank programs from the example package (denoted as Spark-Naive) and PageRank in GraphX (denoted as Spark-GraphX), since the two algorithms represent different implementations of graph algorithms using Spark.

For each iteration in MapReduce, in the map stage, each vertex loads its graph data structure (i.e., adjacency list) from HDFS, and sends its rank to its neighbors through a shuffle. During the reduce stage, each vertex receives the ranks of its neighbors to update its own rank, and stores both the adjacency list and ranks on HDFS for the next iteration. For each iteration in Spark-Naive, each vertex receives ranks of its neighbors through shuffle reads, and joins the ranks with its vertex id to update ranks, and sends the updated ranks to its neighbors through shuffle writes. There is only one stage per iteration in Spark-Naive. This is because Spark uses RDDs to represent data structures for graphs and ranks, and there is no need to materialize these data structures across multiple iterations.

As shown in Table 1, both shuffle selectivity (to exchange ranks) and iteration selectivity (to exchange graph structures) of PageRank is much higher as compared to k-means. We can use RDDs to keep graph data structures in memory in Spark across iterations. Furthermore, the graph data structure in PageRank is more complicated than Point object in k-means. These characteristics make PageRank an excellent candidate to further evaluate the caching component and data pipelining for MapReduce and Spark.

3.5.1 Overall Result

Table 10 presents the overall results for PageRank for various social network data sets, for Spark-Naive, Spark-GraphX and MapReduce. Note that the graph data structure is stored in memory after the pre-processing for Spark-Naive and Spark-GraphX. For all stages including pre-processing, the first iteration and subsequent iterations, Spark-GraphX is faster than Spark-Naive, and Spark-Naive is faster than MapReduce.

Spark-GraphX is about 4x faster than Spark-Naive. This is because the optimized partitioning approach of GraphX can better handle data skew among tasks. The degree distributions of real world social networks follow power-law [14], which means that there is significant skew among tasks in Spark-Naive. Moreover, the Pregel computing framework in GraphX reduces the network overhead by exchanging ranks via co-partitioning vertices [12]. Finally, when we serialize the graph data structure, the optimized graph data structure in GraphX reduces the computational overhead as compared to Spark-Naive.

3.5.2 Execution Details

Figure 4 shows the detailed execution plan along with resource utilization graphs for PageRank, with the Twitter data set for both MapReduce and Spark. We present five iterations since the rest of the iterations show similar characteristics.

As shown in Figure 4 (a), the map and reduce stages take almost the same time. We observe two significant communication and disk I/O overheads during each iteration: 1) exchanging ranks among vertices during the shuffle stage, and 2) materializing the adjacency list and ranks on HDFS for the next iteration in the reduce write stage. As shown in Figure 4 (b), the network and disk I/O overhead for each iteration is caused by the shuffle for exchanging ranks among vertices. Compared to MapReduce, the overhead for persisting adjacency list on HDFS is eliminated in Spark due to RDD caching.

However, the difference of execution time between MapReduce and Spark-Naive cannot be fully attributed to network and disk I/O overheads. As shown in Figure 4, both Spark and MapReduce are CPU-bound for each iteration. By using HPROF [4], we observe that more than 30% of CPU time is spent on serialization and deserialization for the adjacency list object in MapReduce. Therefore, materialization for the adjacency list across iterations leads to significant disk, network, and serialization overheads in MapReduce.

Table 11: The Impact of Storage Levels for PageRank (Spark)

Storage Levels	Algorithm	Caches (GB)	First Iteration (min)	Subsequent Iteration (min)
NONE	Naive	-	4.1m	3.1m
MEMORY_ONLY	Naive	74.9GB	3.1m	2.0m
DISK_ONLY	Naive	14.2GB	3.0m	2.1m
MEMORY_ONLY_SER	Naive	14.2GB	3.0m	2.1m
OFF_HEAP (Tachyon)	Naive	14.2GB	3.0m	2.1m
NONE	GraphX	-	32s	-
MEMORY_ONLY	GraphX	62.9GB	37s	29s
DISK_ONLY	GraphX	43.1GB	68s	50s
MEMORY_ONLY_SER	GraphX	43.1GB	61s	43s

Table 12: Execution Time with Varying Number of Disks

	WC (40 GB)		Sort (100 GB)		k-means (200 m)	
Disk #	Spark	MR	Spark	MR	Spark	MR
1	1.0m	2.4m	4.8m	3.5m	3.6m	11.5m
2	1.0m	2.4m	4.8m	3.4m	3.7m	11.5m
4	1.0m	2.4m	4.8m	3.5m	3.7m	11.7m
8	1.0m	2.4m	4.8m	3.4m	3.6m	11.7m

3.5.3 Impact of RDD caching

Table 11 presents the impact of various storage levels on the effectiveness of RDD caching for PageRank. For Spark-Naive, the execution time of both the first iteration and the subsequent iterations is not sensitive to storage levels. Furthermore, serialization can reduce the size of RDD by a factor of five.

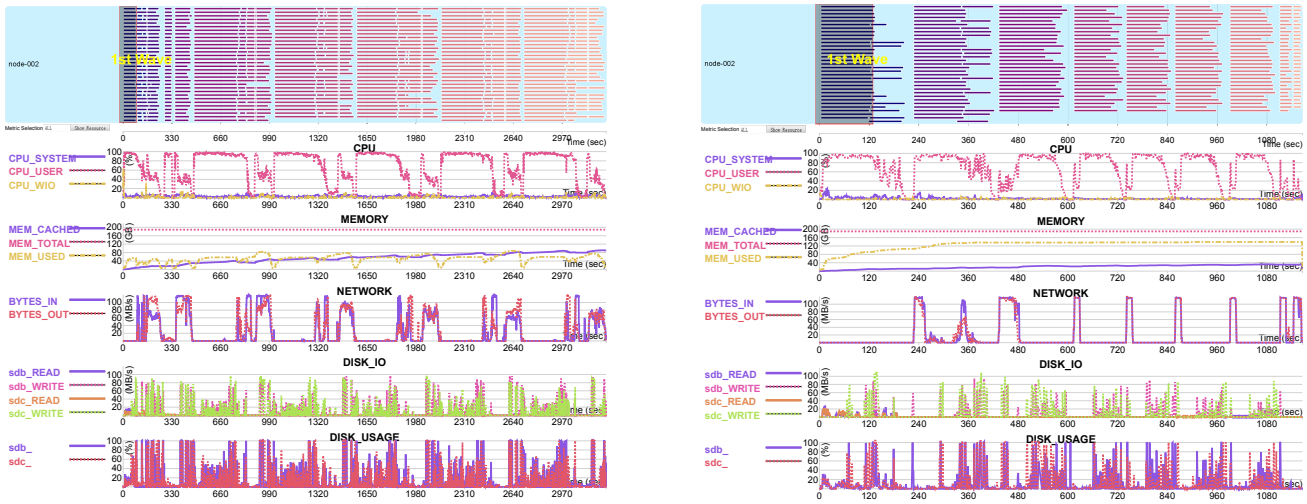
For Spark-GraphX, RDD caching options with serialization result in about 50% performance degradation in subsequent iterations compared to MEMORY_ONLY due to the CPU overhead for deserialization. In addition, when we disable RDD caching, the execution time, and the amount of shuffled data increase from one iteration to the next iteration.

3.5.4 Summary of Insights

For PageRank and similar graph analytic algorithms such as Breadth First Search and Community Detection [18] that read the graph structure and iteratively exchange states through a shuffle, we summarize insights from our PageRank experiments as follows: (1) Compared to MapReduce, Spark can avoid materializing graph data structures on HDFS across iterations, which reduces overheads for serialization/de-serialization, disk I/O, and network I/O. (2) In Graph-X, the CPU overhead for serialization/de-serialization may be higher than the disk I/O overhead without serialization.

3.6 Impact of Cluster Configurations

In this section, we measure the performance impact of varying two key parameters: 1) the number of disks 2) the JVM heap size.



(a) MapReduce (b) Spark-Naive

Figure 4: The Execution Details of PageRank (Twitter Data)

3.6.1 *Execution Time with Varying Number of Disks*

In this set of experiments, we vary the number of disks used to store intermediate data (i.e., map output for MapReduce and Spark, RDD caching on disk for Spark) to measure its impact on performance. We use DISK_ONLY configuration for Spark k-means to ensure that RDDs are cached on disk. As shown in Table 12, the execution time for these workloads is not sensitive to the number of disks for intermediate data storage. Also, through the analysis of the detailed execution plan, even for the single disk case, we find that the disk is not fully utilized.

Next, we manually drop OS buffer cache during the execution of a job to ensure that the intermediate data is read from the disks. This has little impact on most of the workloads. The only exception is Spark k-means when using only 1 disk for RDD caching. By breaking down the execution time for this case, we find that clearing the OS cache results in approximately 80% performance degradation for the next iteration.

From the above results, we conclude that disk I/O for intermediate data is not a bottleneck for typical cluster configurations, for a majority of MapReduce and Spark workloads. On the other hand, for some extremely unbalanced cluster configurations (e.g., 24 CPU cores with 1 disk and no OS buffer cache), disk I/O becomes a bottleneck.

3.6.2 *Impact of Memory Limits*

In the next set of experiments, we vary the JVM heap size to evaluate the impact of various memory limits including the case in which data does not fit in main memory. For a fair comparison, we disable the overlap between map and reduce stages for MapReduce. We run these experiments with 40 GB WC, 100 GB Sort and 200 million records k-means. Table 13 presents the impact of the JVM heap size on the task execution time, the fraction of time spent in GC, and additional data spilling. All the metrics presented in this table are median values among the tasks. We find that: (1) The fraction of GC time decreases as the JVM heap size increases, except for the reduce stage for Spark-Sort with 4 GB heap size per task. This is caused by the additional overhead for OS page swapping (explained in Section 3.3.4); (2) When the JVM heap size is larger than 256 MB, the fraction of GC time becomes relatively stable; (3) The execution time is not sensitive to additional data spills to disks since disk I/O is not a bottleneck for WC and Sort, for both MapReduce and Spark. As shown in Table 14, the impact of the JVM heap size on k-means performance is very similar in comparison to WC and Sort. In summary, if tasks run with the JVM heap size above a certain level (e.g., larger than 256 MB), the JVM heap size will not significantly affect the execution time for most of the MapReduce and Spark workloads even with additional data spills to disks.

Table 15: The Execution Times with Task Failures

		Sort (map killed)		Sort (reduce killed)		k-means (1st&4th iter killed)		
	N_f	% killed	%slow-down	% killed	%slow-down	% killed	% slow-down (1st)	% slow-down (4th)
Spark	1	1.6%	2.1%	33.3%	108.3%	1.2%	7.1%	57.7%
	1	1.6%	6.3%	49.2%	129.2%	4.9%	14.3%	80.8%
	1	4%	6.3%	63.3%	129.2%	9.8%	14.3%	92.3%
	4	3.6%	4.2%	40%	81.3%	4.9%	7.1%	57.7%
	4	14.6%	6%	106%	62.5%	18.2%	21.4%	176.9%
	4	27.3%	12.5%	122%	70.8%	37.4%	42.9%	269.2%
MapReduce	1	0.5%	36.8%	3.3%	18.2%	1.3%	7.9%	7.1%
	1	1.9%	40.0%	13.3%	18.6%	5%	18.6%	6.4%
	1	3.1%	58.2%	25.8%	18.6%	10%	26.4%	27.9%
	4	1.9%	7.3%	13.3%	14.1%	5%	5.7%	2.9%
	4	7.3%	10.5%	26.75%	13.6%	20%	10.0%	6.4%
	4	10.5%	18.2%	53.3%	29.1%	39.7%	22.1%	24.3%

3.7 Fault-Tolerance

This section evaluates the effectiveness of the built-in fault-tolerance mechanisms in MapReduce and Spark. We present the results for Sort and k-means in Table 15 (the results for other workloads are similar). The killed tasks are evenly distributed on N_f nodes, ranging from a single node to all four nodes in the cluster. These experiments confirm that both frameworks provide fault-tolerance for tasks. As shown in Table 15, when tasks are killed during the reduce stage for Sort, the slow-down for Spark is much worse than that for MapReduce. Comparing to MapReduce, where only reduce tasks are re-executed, in Spark the loss of an executor will lead to the re-execution of the portion of map tasks which lost the block information. Thus, a potential improvement for Spark is to increase the availability of the block information in case of failure. In the experiments with k-means, tasks are killed during the first, and the fourth iteration. As shown in the rightmost column

Table 13: The Impact of JVM Heap Size on WC and Sort

		Spark							MR						
	JVM per task	job time	map median	%GC map	reduce median	%GC reduce	map spill	reduce spill	job time	map median	%GC map	reduce median	%GC reduce	map spill	reduce spill
WC (40 GB)	32 MB	1.2m	12s	25%	12s	33%	2.4 MB	5.7MB	failed	-	-	-	-	-	-
	64 MB	1.1m	11s	18%	10s	20%	1.9 MB	4.3MB	8.6m	2.5m	72%	15s	19%	1.67MB	0
	128 MB	1.1m	10s	10%	8s	13%	0	0	2.7m	41s	10%	13s	15%	0.32MB	0
	256 MB	1.1m	10s	10%	8s	13%	0	0	2.6m	39s	7%	12s	10%	0	0
	512 MB	1.1m	9s	8%	8s	5%	0	0	2.5m	39s	6%	12s	6%	0	0
	1 GB	1.1m	9s	6%	8s	3%	0	0	2.5m	39s	7%	12s	1%	0	0
	2 GB	1.1m	9s	2%	8s	1%	0	0	2.6m	40s	6%	12s	3%	0	0
	4 GB	1.0m	9s	1%	7s	0%	0	0	2.6m	40s	5%	12s	4%	0	0
Sort (100 GB)	32 MB	failed	-	-	-	-	-	-	failed	-	-	-	-	-	-
	64 MB	failed	-	-	-	-	-	-	19.5m	1.7m	59%	1m16s	30%	59MB	58MB
	128MB	7.1m	24s	33%	27s	44%	65 MB	65.7 MB	6.1m	20s	13%	31s	21%	59MB	58MB
	256 MB	6.1m	21s	24%	21s	29%	65 MB	64.9 MB	5.6m	22s	11%	21s	13%	59MB	58MB
	512 MB	5.9m	20s	20%	20s	20%	63.8MB	62.7MB	5.5m	22s	8%	19s	8%	59MB	58MB
	1 GB	5.6m	18s	17%	19s	16%	45.3MB	52 MB	5.4m	22s	10%	18s	4%	59MB	58MB
	2 GB	5.5m	17s	12%	20s	10%	43.2MB	51.2 MB	5.3m	22s	6%	17s	2%	59MB	58MB
	4 GB	7.6m	13s	8%	27s	4%	0	0	5.3m	22s	6%	17s	2%	59MB	58MB

Table 14: The Impact of JVM Heap Size on k-means

	Spark (MEMORY_ONLY)					Spark (DISK_ONLY)					MR			
JVM	1st-iter	%GC	sub-iter	%GC	%cached	1st-iter	%GC	sub-iter	%GC	%cached	1st-iter	%GC	sub-iter	%GC
32 MB	4.1m	35%	7.8m	71%	0%	1.9m	15%	41s	14%	100%	-	-	-	-
64 MB	4.2m	40%	7.7m	71%	0.3%	1.8m	12%	37s	12%	100%	5.5m	59%	5.5m	59%
128 MB	1.7m	12%	5.6m	65%	6%	1.6m	5%	32s	8%	100%	2.4m	7%	2.4m	7%
256 MB	1.6m	12%	1.2m	13%	27%	1.5m	2%	31s	5%	100%	2.3m	5%	2.3m	5%
512 MB	1.6m	12%	50s	10%	68%	1.5m	1%	29s	2%	100%	2.3m	4%	2.3m	4%
1 GB	1.6m	13%	30s	8%	100%	1.5m	0.6%	29s	1%	100%	2.3m	6%	2.4m	6%
2 GB	1.5m	6%	27s	5%	100%	1.5m	0.3%	29s	0.5%	100%	2.3m	4%	2.3m	4%
4 GB	1.4m	4%	27s	3%	100%	1.5m	0.2%	28s	0.3%	100%	2.3m	3%	2.3m	3%

in Table 15, for Spark k-means, when tasks are killed in subsequent iterations, the relative performance advantage of Spark over MapReduce drops. This is mainly caused by re-parsing of objects for lost RDDs. In addition, we observe that failures in one iteration do not affect the execution time of the following iterations, for both MapReduce and Spark.

4. SUMMARY & DISCUSSION

In this section, we summarize the key insights from the results presented in this paper, and discuss lessons learned which would be useful for both researchers and practitioners. Overall, our experiments show that Spark is approximately 2.5x, 5x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank, respectively. Although Spark's performance advantage over MapReduce is known, this paper presents the first in-depth analysis of these performance differences between the two frameworks. Particularly, we attribute Spark's performance advantage to a number of architectural differences from MapReduce: (1) For Word Count and similar workloads, where the map output selectivity can be significantly reduced using a map side combiner, hash-based aggregation in Spark is more efficient than sort-based aggregation in MapReduce. The execution time break-down result indicates that the hash-based framework contributes to about 39% of the overall improvement for Spark. (2) For iterative algorithms such as k-means and PageRank, caching the input as RDDs can reduce both CPU (i.e., parsing text to objects) and disk I/O overheads for subsequent iterations. It is noteworthy that the CPU overhead is often the bottleneck in scenarios where subsequent iterations do not use RDD caching. As a result, RDD caching is much more efficient than other low-level caching approaches such as OS buffer caches, and HDFS caching, which can only reduce disk I/O. Through micro-benchmark experiments, we show that reducing parsing (CPU) overhead contributes to more than 90% of the overall speed-up for subsequent iterations in k-means. (3) Since Spark enables data pipelining within a stage, it avoids materialization overhead for output data on HDFS (i.e., serialization, disk I/O, and network I/O) among iterations for graph analytics algorithms such as PageRank. An exception to Spark's performance advantage over MapReduce is the Sort workload, for which MapReduce is 2x faster than Spark. This is due to differences in task execution plans. MapReduce can overlap the shuffle stage with the map stage, which effectively hides network overhead which is often a bottleneck for the reduce stage.

We also observe several implementation differences between MapReduce and Spark. In early versions of Spark (before v1.1.0), the GC overhead for a shuffle is sensitive to the size of data processed by each Spark executor. A sort job can even exceed the GC overhead limit when more than 10 reduce tasks (with a total of 64 GB) are processed on an executor (with 32 GB JVM heap size). This may be due to memory leak related issues. Since Spark v1.3.0, this GC overhead issue has been fixed and an executor is now scalable to many reduce tasks. This improvement makes Spark's thread-based model outperform MapReduce's process-based model. Both MapReduce and Spark are now scalable to run many tasks during a shuffle, but Spark's thread model eliminates the context switching overhead among task JVM processes in MapReduce. In addition, through execution time breakdown, we find that MapReduce is more than 10x slower than Spark for task loading.

Despite architectural and implementation differences between MapReduce and Spark, we observe a few common characteristics for the selected workloads on both frameworks. (1) For one pass jobs like Word Count and Sort, the map stage is often CPU-bound (with compression for intermediate data) and the reduce stage is often network-bound. Disk I/O is often not a bottleneck for both map and reduce stages even when the shuffle selectivity is high (e.g., Sort). It means that the reduction of disk spills during a shuffle might not lead to performance speed-up. Even worse for Spark, increasing the JVM heap size to avoid disk spills may lead to slow-

down because of unexpected overhead for GC and OS page swapping. (2) For typical machine learning (e.g., k-means and linear regression) and graph analytics (e.g., PageRank) algorithms, parsing text to objects is often the bottleneck for each iteration. RDD caching addresses this issue effectively by reducing the CPU overhead for parsing. OS buffer cache and HDFS caching to eliminate disk I/O, are ineffective from this perspective.

To establish the generality of our key insights, we evaluate Word Count, Sort, and k-means under varying cluster configurations including varying the number of disks (for storing intermediate data) and the JVM heap size. The results indicate that our findings, in particular, the identified system behavior is not sensitive to these configurations in most cases: (1) for Spark, k-means using DISK_ONLY RDD caching remains CPU-bound when the number of disks for storing RDD caches is reduced from 8 to 1 on each node; (2) for both MapReduce and Spark, the map stage of Sort is CPU-bound when there is only one disk used to store the map output on each node; (3) for both MapReduce and Spark, there is no significant performance degradation for all the selected workloads when the JVM heap size per task (using 128 MB input split) is reduced from 4 GB to 128 MB. In addition to these findings, we also identified a few cases where the bottleneck changes with varying configuration: (1) GC overhead becomes a major bottleneck when the JVM heap size per task is reduced to be less than 64 MB, for both MapReduce and Spark; (2) disk I/O becomes the bottleneck when we perform disk-based RDD caching on nodes with extremely unbalanced I/O and CPU capacities.

We evaluated the effectiveness of fault-tolerance mechanisms built in both MapReduce and Spark, during different stages for each workloads. A potential improvement for Spark is on the availability of the block information in case of an executor failure to avoid recomputation of the portion of tasks that lose the block information in case of failure.

To the best of our knowledge, this is the first experimental study that drills-down sufficiently to explain the reasons for performance differences, attributes these differences to architectural and implementation differences, and summarizes the workload behaviors accordingly. To summarize, our detailed analysis of the behavior of the selected workloads on the two frameworks would be of particular interest to developers of core engines, system administrators/users, and researchers of MapReduce/Spark.

Developers: The core-engine developer of MapReduce/Spark can improve both the architecture and implementation through our observations. To improve the architecture, Spark might: (1) support the overlap between two stages with the shuffle dependency to hide the network overhead; (2) improve the availability of block information in case of an executor failure to avoid re-computation of some tasks from the previous stage. To catch up with the performance of Spark, the potential improvements to MapReduce are: (1) significantly reduce the task load time; (2) consider thread-based parallelism among tasks to reduce the context switching overhead; (3) provide hash-based framework as an alternative to the sort-based shuffle; (4) consider caching the serialized intermediate data in memory, or on disk for reuse across multiple jobs.

System Administrators/Users: The system administrators/users could have an in-depth understanding about the system behavior for typical workloads under various configurations, and gain insights for system tuning from OS, JVM to MapReduce/Spark parameters: (1) Once tuned properly, the majority of workloads are CPU-bound for both MapReduce and Spark, and hence are scalable to the number of CPU cores. (2) For MapReduce, the network overhead during a shuffle can be hidden by overlapping the map and reduce stages. For Spark, the intermediate data should always be compressed, because its shuffle cannot be overlapped. (3) For both MapReduce and Spark, more attention should be paid to GC and OS page swapping overhead rather than additional disk spills. (4) For iterative algorithms in Spark, counter-intuitively, DISK_ONLY configuration might be better than MEMORY_ONLY. Because both of them address the object parsing bottleneck, but the former avoids GC and page swapping issues by eliminating memory consumption which makes it scalable to very large data sets.

Researchers: The detailed analysis of workload behaviors characterize the nature of physical execution plans for both MapReduce and Spark. The researchers can further derive the trade-offs during the execution of a MapReduce or Spark jobs such as (1) trade-off between parallelism and context switching, (2) trade-off between in-memory and on-disk caching, and (3) trade-off between serialization and memory consumption. These trade-offs are the foundations for researchers to explore new cost models [9, 16] for MapReduce and Spark, which could be widely used for optimizations such as self-tuning jobs and capacity planning on the cloud.

5. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Mahout. https://mahout.apache.org/.

[3] HDFS caching. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[4] HPROF: A heap/cpu profiling tool. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

[5] RRDtool. http://oss.oetiker.ch/rrdtool/.

[6] Spark wins 2014 graysort competition. http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[8] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International London, 1980.

[9] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB*, 4(11):1111–1122, 2011.

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW*, pages 41–51, 2010.

[11] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC*, pages 1–15, 2014.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[13] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[14] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.

[15] O. OMalley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *Sort Benchmark*, 2009.

[16] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. MRTuner: A toolkit to enable holistic optimization for mapreduce jobs. *VLDB*, 7(13):1319–1330, 2014.

[17] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, pages 5:1–5:16, 2013.

[18] W. Xue, J. Shi, and B. Yang. X-RIME: Cloud-based large scale social network analysis. In *SCC*, pages 506–513, 2010.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.