Making Sense of Performance in Data Analytics Frameworks

Kay Ousterhout*, Ryan Rasti*†◇, Sylvia Ratnasamy*, Scott Shenker*†, Byung-Gon Chun‡
*UC Berkeley, †ICSI, ◇VMware, ‡Seoul National University

Abstract

There has been much research devoted to improving the performance of data analytics frameworks, but comparatively little effort has been spent systematically identifying the performance bottlenecks of these systems. In this paper, we develop blocked time analysis, a methodology for quantifying performance bottlenecks in distributed computation frameworks, and use it to analyze the Spark framework's performance on two SQL benchmarks and a production workload. Contrary to our expectations, we find that (i) CPU (and not I/O) is often the bottleneck, (ii) improving network performance can improve job completion time by a median of at most 2%, and (iii) the causes of most stragglers can be identified.

1 Introduction

Large-scale data analytics frameworks such as Hadoop [13] and Spark [51] are now in widespread use. As a result, both academia and industry have dedicated significant effort towards improving the performance of these frameworks.

Much of this performance work has been motivated by three widely-accepted mantras about the performance of data analytics:

1. **The network is a bottleneck.** This has motivated work on a range of network optimizations, including load balancing across multiple paths, leveraging application semantics to prioritize traffic, aggregating data to reduce traffic, isolation, and more [6, 14, 17–21, 27, 28, 41, 42, 48, 53].
2. **The disk is a bottleneck.** This has led to work on using the disk more efficiently [43] and caching data in memory [9, 30, 47, 51].
3. **Straggler tasks significantly prolong job completion times and have largely unknown underlying causes.** This has driven work on mitigation using task speculation [8, 10, 11, 52] or running shorter tasks to improve load balancing [39]. Researchers have been able to identify and target a small number of underlying causes such as data skew [11, 26, 29] and popularity skew [7].

Most of this work focuses on a particular aspect of the system in isolation, leaving us without a comprehensive understanding of which factors are most important to the end-to-end performance of data analytics workloads.

This paper makes two contributions towards a more comprehensive understanding of performance. First, we develop a methodology for analyzing end-to-end performance of data analytics frameworks; and second, we use our methodology to study performance of two SQL benchmarks and one production workload. Our results run counter to all three of the aforementioned mantras.

The first contribution of this paper is *blocked time analysis*, a methodology for quantifying performance bottlenecks. Identifying bottlenecks is challenging for data analytics frameworks because of pervasive parallelism: jobs are composed of many parallel tasks, and each task uses pipelining to parallelize the use of network, disk, and CPU. One task may be bottlenecked on different resources at different points in execution, and at any given time, tasks for the same job may be bottlenecked on different resources. Blocked time analysis uses extensive white-box logging to measure how long each task spends blocked on a given resource. Taken alone, these per-task measurements allow us to understand straggler causes by correlating slow tasks with long blocked times. Taken together, the per-task measurements for a particular job allow us to simulate how long the job would have taken to complete if the disk or network were infinitely fast, which provides an upper bound on the benefit of optimizing network or disk performance.

The second contribution of this paper is using blocked time analysis to understand Spark's performance on two industry benchmarks and one production workload. In studying the applicability of the three aforementioned claims to these workloads, we find:

1. **Network optimizations can only reduce job completion time by a median of at most 2%.** The network is not a bottleneck because much less data is sent over the network than is transferred to and from disk. As a result, network I/O is mostly irrelevant to overall performance, even on 1Gbps networks.
2. **Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%.** CPU utilization is typically much higher than disk utilization; as a result, engineers should be careful about trading off I/O time for CPU time by, for example, using more sophisticated serialization and compression techniques.
3. **Optimizing stragglers can only reduce job completion time by a median of at most 10%, and in**

Workload name	Total queries	Cluster size	Data size
Big Data Benchmark [46], Scale Factor 5 (BDBench)	50 (10 unique queries, each run 5 times)	40 cores (5 machines)	60GB
TPC-DS [45], Scale Factor 100	140 (7 users, 20 unique queries)	160 cores (20 machines)	17GB
TPC-DS [45], Scale Factor 5000	260 (13 users, 20 unique queries)	160 cores (20 machines)	850GB
Production	30 (30 unique queries)	72 cores (9 machines)	tens of GB

Table 1: Summary of workloads run. We study one larger workload in §6.

75% of queries, we can identify the cause of more than 60% of stragglers. Blocked-time analysis illustrates that the two leading causes of Spark stragglers are Java's garbage collection and time to transfer data to and from disk. We found that targeting the underlying cause of stragglers could reduce non-straggler runtimes as well, and describe one example where understanding stragglers in early experiments allowed us to identify a bad configuration that, once fixed, reduced job completion time by a factor of two.

These results question the prevailing wisdom about the performance of data analytics frameworks. By necessity, our study does not look at a vast range of workloads nor a wide range of cluster sizes, because the ability to add finer-grained instrumentation to Spark was critical to our analysis. As a result, we cannot claim that our results are broadly representative. However, the fact that the prevailing wisdom about performance is so incorrect on the workloads we do consider suggests that there is much more work to be done before our community can claim to understand the performance of data analytics frameworks.

To facilitate performing blocked time analysis on a broader set of workloads, we have added almost all[1] of our instrumentation to Spark and made our analysis tools publicly available. We have also published the detailed benchmark traces that we collected so that other researchers may reproduce our analysis or perform their own [37].

The remainder of this paper begins by describing blocked time analysis and the associated instrumentation (§2). Next, we explore the importance of disk I/O (§3), the importance of network I/O (§4), and the importance and causes of stragglers (§5); in each of these sections, we discuss the relevant related work and contrast it with our results. We explore the impact of cluster and data size on our results in §6. We end by arguing that future system designs should consider performance measurement as a first-class concern (§7).

2 Methodology

This section describes the workloads we ran, the blocked time analysis we used to understand performance, and our experimental setup.

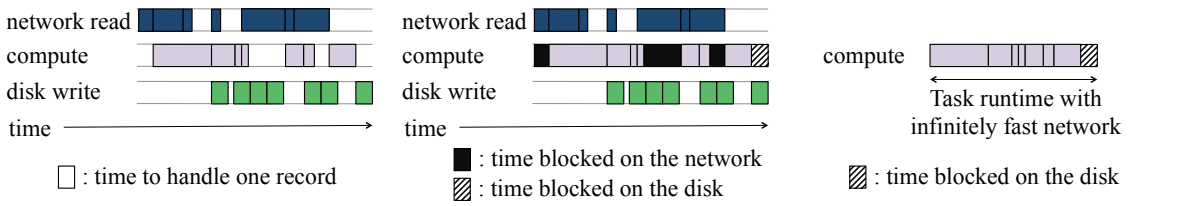
[1]Some of our logging needed to be added outside of Spark, as we elaborate on in §2.3.1, because it could not be implemented in Spark with sufficiently low overhead.

2.1 Workloads

Our analysis centers around fine-grained instrumentation of two benchmarks and one production workload running on Spark, summarized in Table 1.

The big data benchmark (BDBench) [46] was developed to evaluate the differences between analytics frameworks and was derived from a benchmark developed by Pavlo et al. [40]. The input dataset consists of HTML documents from the Common Crawl document corpus [2] combined with SQL summary tables generated using Intel's Hadoop benchmark tool [50]. The benchmark consists of four queries including two exploratory SQL queries, one join query, and one page-rank-like query. The first three queries have three variants that each use the same input data size but have different result sizes to reflect a spectrum between business-intelligence-like queries (with result sizes that could fit in memory on a business intelligence tool) and ETL-like queries with large result sets that require many machines to store. We run the queries in series and run five iterations of each query. We use the same configuration that was used in published results [46]: we use a scale factor of five (which was designed to be run on a cluster with five worker machines), and we run two versions of the benchmark. The first version operates on data stored in-memory using SparkSQL's columnar cache (cached data is not replicated) and the second version operates on data stored on-disk using Hadoop Distributed File System (HDFS), which triply replicates data for fault-tolerance.

Our second benchmark is a variant of the Transaction Processing Performance Council's decision-support benchmark (TPC-DS) [45]. The TPC-DS benchmark was designed to model multiple users running a variety of decision-support queries including reporting, interactive OLAP, and data mining queries. All of the users run in parallel; each user runs the queries in series in a random order. The benchmark models data from a retail product supplier about product purchases. We use a subset of 20 queries that was selected in an existing industry benchmark that compares four analytics frameworks [25]. Similar to with the big data benchmark, we run two variants. The first variant stores data on-disk using Parquet [1], a compressed columnar storage format that is the recommended storage format for high performance with Spark SQL, and uses a scale factor of 5000. The



(a) Pipelined execution of a typical Spark task (b) Blocked times that we measured (c) Task runtime without blocking on network

Figure 1: Each Spark task pipelines use of network, CPU, and disk, as shown in (a). To understand the importance of disk and network, we measure times when a task's compute thread blocks on the network or disk, as shown in (b). To determine the best-case task runtime resulting from network (or disk) optimizations, we subtract time blocked on network (or disk), as shown in (c).

second, in-memory variant uses a smaller scale factor of 100; this small scale factor is necessary because SparkSQL's cache is not well optimized for the type of data used in the TPC-DS benchmark, so while the data only takes up 17GB in the compressed on-disk format, it occupies 200GB in memory. We run both variants of the benchmark on a cluster of 20 machines.

The final Spark workload described by the results in this paper is a production workload from Databricks that uses their cloud product [3] to submit ad-hoc Spark queries. Input data for the queries includes a large fact table with over 50 columns. The workload includes a small number of ETL queries that read input data from an external file system into the memory of the cluster; subsequent queries operate on in-memory data and are business-intelligence-style queries that aggregate and summarize data. Data shown in future graphs breaks the workload into the in-memory and on-disk components. For confidentiality reasons, further details of the workload cannot be disclosed.

2.2 Framework architecture

All three workloads are SQL workloads that use SparkSQL [4] to compile SQL queries into Spark jobs. Spark jobs are broken down into stages composed of many parallel tasks. The tasks in a stage each perform the same computation using different subsets of the stage's input data. Early stages read input data from a distributed file system (e.g., HDFS) or Spark's cache, whereas later stages typically read input data using a network shuffle, where each task reads a subset of the output data from all of the previous stage's tasks. In the remainder of this paper, we use "map task" to refer to tasks that read blocks of input data stored in a distributed file system, and "reduce task" to refer to tasks that read data shuffled from the previous stage of tasks. Each Spark job is made up of a directed acyclic graph of one or more stages. As a result, a single Spark job may contain multiple stages of reduce tasks; for example, to compute the result of a SQL query that includes multiple joins (in contrast, with MapReduce, all jobs include exactly one map and optionally one reduce).

2.3 Blocked time analysis

The goal of this paper is to understand performance of workloads running on Spark; this is a challenging goal for two reasons. First, understanding the performance of a single task is challenging because tasks use pipelining, as shown in Figure 1(a). As a result, tasks often use multiple resources simultaneously, and different resources may be the bottleneck at different times in task execution. Second, understanding performance is challenging because jobs are composed of many tasks that run in parallel, and each task in a job may have a unique performance profile.

To make sense of performance, we focus on *blocked time analysis*, which allows us to quantify how much more quickly a job would complete if tasks never blocked on the disk or the network (§3.3 explains why we cannot use blocked time analysis to understand CPU use). The resulting job completion time represents a best-case scenario of the possible job completion time after implementing a disk or network optimization. Blocked time analysis lacks the sophistication and generality of general purpose distributed systems performance analysis tools (e.g., [5, 15]), and unlike black-box approaches, requires adding instrumentation within the application. We use blocked-time analysis because unlike existing approaches, it provides a single, easy to understand number to characterize the importance of disk and network use.

2.3.1 Instrumentation

To understand the performance of a particular task, we focus on blocked time: time the task spends blocked on the network or the disk (shown in Figure 1(b)). We focus on blocked time from the perspective of the compute thread because it provides a single vantage point from which to measure. The task's computation runs as a single thread, whereas network requests are issued by multiple threads that operate in the background, and disk I/O is pipelined by the OS, outside of the Spark process. We focus on blocked time, rather than measuring all time when the task is using the network or the disk, because network or disk performance improvements cannot speed up parts of the task that execute in parallel with network or disk use.

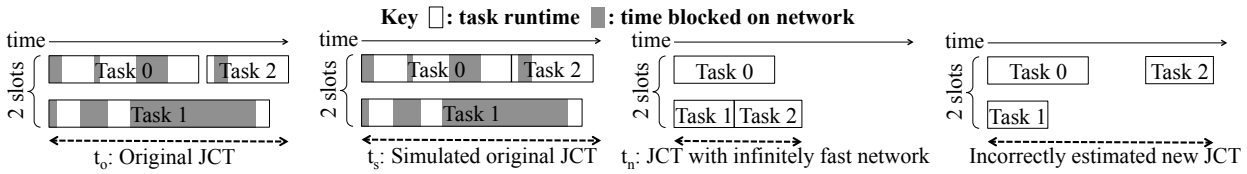


Figure 2: To compute a job's completion time (JCT) without time blocked on network, we subtract the time blocked on the network from each task, and then simulate how the tasks would have been scheduled, given the number of slots used by the original runtime and the new task runtimes. We perform the same computation for disk.

Obtaining the measurements shown in Figure 1(b) required significant improvements to the instrumentation in Spark and in HDFS.While some of the instrumentation required was already available in Spark, our detailed performance analysis revealed that existing logging was often incorrect or incomplete [33–36, 38, 44]. Where necessary, we fixed existing logging and pushed the changes upstream to Spark.

We found that cross validation was crucial to validating that our measurements were correct. In addition to instrumentation for blocked time, we also added instrumentation about the CPU, network, and disk utilization on the machine while the task was running (per-task utilization cannot be measured in Spark, because all tasks run in a single process). Utilization measurements allowed us to cross validate blocked times; for example, by ensuring that when tasks spent little time blocked on I/O, CPU utilization was correspondingly high.

As part of adding instrumentation, we measured Spark's performance before and after the instrumentation was added to ensure the instrumentation did not add to job completion time. To ensure logging did not affect performance, we sometimes had to add logging outside of Spark. For example, to measure time spent reading input data, we needed to add logging in the HDFS client when the client reads large "packets" of data from disk, to ensure that timing calls were amortized over a relatively time-consuming read from disk.[2] Adding the logging in Spark, where records are read one at a time from the HDFS client interface, would have degraded performance.

2.3.2 Simulation

Spark instrumentation allowed us to determine how long each task was blocked on network (or disk) use; subtracting these blocked times tells us the shortest possible task runtime that would result from optimizing network (or disk) performance, as shown in Figure 1(c).[3] Next, we used a simulation to determine how the shorter task completion times would affect job completion time. The simulation replays the execution of the scheduling of the job's tasks, based on the number of slots used by the original job and the new task runtimes. Figure 2 shows a simple example. The example on the far right illustrates why we need to replay execution rather than simply use the original task completion times minus blocked time: that approach underestimates improvements because it does not account for multiple waves of tasks (task 2 should start when the previous task finishes, not at its original start time) and does not account for the fact that tasks might have been scheduled on different machines given different runtimes of earlier tasks (task 2 should start on the slot freed by task 1 completing). We replay the job based only on the number of slots used by the original job, and do not take into account locality constraints that might have affected the scheduling of the original job. This simplifying assumption does not significantly impact the accuracy of our simulation: at the median, the time predicted by our simulation is within 4% of the runtime of the original job. The ninety-fifth percentile error is at most 7% for the benchmark workloads and 27% for the production workload.[4] In order to minimize the effect of this error on our results, we always compare the simulated time without network (or disk) to the simulated original time. For example, in the example shown in Figure 2, we would report the improvement as t_n/t_s, rather than as t_n/t_o. This focuses our results on the impact of a particular resource rather than on error introduced by our simulation.

2.4 Cluster setup

For the benchmark workloads, we ran experiments using a cluster of Amazon EC2 m2.4xlarge instances, which each have 68.4GB of memory, two disks, and eight cores. Our experiments use Apache Spark version 1.2.1 and Hadoop version 2.0.2. Spark runs queries in long-running processes, meaning that production users of Spark will run queries in a JVM that has been running for a long period of time. To emulate that environment, before running each benchmark, we ran a single full trial of all of the benchmark queries to warm up the JVM. For the big data benchmark, where only one query runs at a time, we cleared the

[2]This logging is available in a modified version of Hadoop at `https://github.com/kayousterhout/hadoop-common/tree/2.0.2-instrumented`

[3]The task runtime resulting from subtracting all time blocked on the network may be lower than the runtime that would result even if the network were infinitely fast, because eliminating network blocked time might result in more time blocked on disk. As we emphasize throughout the paper, our results represent a bound on the largest possible improvement from optimizing network or disk performance.

[4] The production workload has higher error because we don't model pauses between stages that occur when SparkSQL is updating the query plan. If we modeled these pauses, the impact of disk and network I/O would be lower.

OS buffer cache on all machines before launching each query, to ensure that input data is read from disk. While our analysis focuses on one cluster size and data size for each benchmark, we illustrate that scaling to larger clusters does not significantly impact our results in §6.

The production workload ran on a 9-machine cluster with 250GB of memory; further details about the cluster configuration are proprietary. The cluster size and hardware is representative of Databricks' users.

2.5 Production traces

Where possible, we sanity-checked our results with coarse-grained analysis of traces from Facebook, Google, and Microsoft. The Facebook trace includes 54K jobs run during a contiguous period in 2010 on a cluster of thousands of machines (we use a 1-day sample from this trace). The Google data includes all MapReduce jobs run at Google during three different one month periods in 2004, 2006, and 2007, and the Microsoft data includes data from an analytics cluster with thousands of servers on a total of eighteen different days in 2009 and 2010. While our analysis would ideally have used only data from production analytics workloads, all data made available to us includes insufficient instrumentation to compute blocked time. For example, the default logs written by Hadoop (available for the Facebook cluster) include only the total time for each map task, but do not break map task time into how much time was spent reading input data and how much time was spent writing output data. This has forced researchers to rely on estimation techniques that can be inaccurate, as we show in §4.4. Therefore, our analysis begins with a detailed instrumentation of Spark, but in most cases, we demonstrate that our high-level takeaways are compatible with production data.

3 How important is disk I/O?

Previous work has suggested that reading input data from disk can be a bottleneck in analytics frameworks; for example, Spark describes speedups of 40× for generating a data analytics report as a result of storing input and output data in memory using Spark, compared to storing data on-disk and using Hadoop for computation [51]. PACMan reported reducing average job completion times by 53% as a result of caching data in-memory [9]. The assumption that many data analytics workloads are I/O bound has driven numerous research proposals (e.g., Themis [43], Tachyon [30]) and the implementation of in-memory caching in industry [47]. Based on this work, our expectation was that time blocked on disk would represent the majority of job completion time.

3.1 How much time is spent blocked on disk I/O?

Using blocked time analysis, we compute the improvement in job completion time if tasks did not spend any time blocked on disk I/O.[5] This involved measuring time blocked on disk I/O at four different points in task execution:

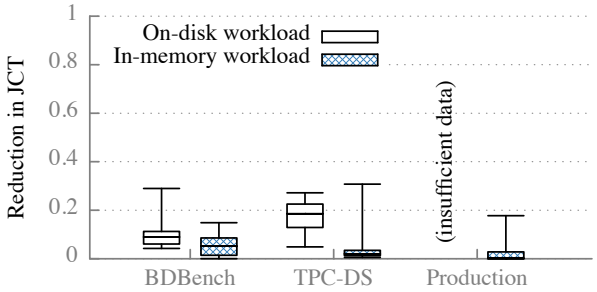
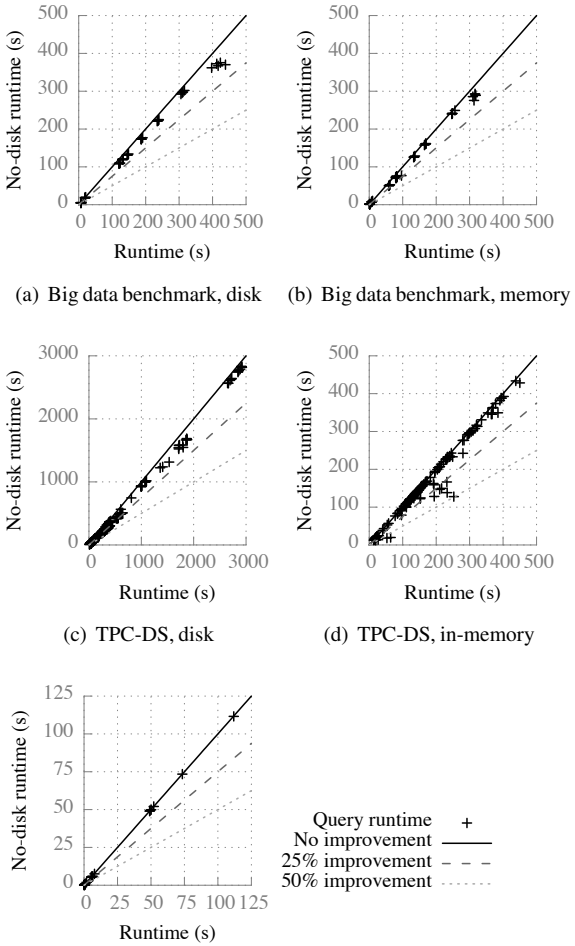


Figure 3: Improvement in job completion time (JCT) as a result of eliminating all time spent blocked on disk I/O. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles.

1. Reading input data stored on-disk (this only applies for the on-disk workloads; in-memory workloads read input from memory).
2. Writing shuffle data to disk. Spark writes all shuffle data to disk, even when input data is read from memory.
3. Reading shuffle data from a remote disk. This time includes both disk time (to read data from disk) and network time (to send the data over the network). Network and disk use is tightly coupled and thus challenging to measure separately; we measure the total time as an *upper bound* on the improvement from optimizing disk performance.
4. Writing output data to local disk and two remote disks (this only applies for the on-disk workloads). Again, the time to write data to remote disks includes network time as well; we measure both the network and disk time, making our results an upper bound on the improvement from optimizing disk.

Using blocked time analysis, we find that the median improvement from eliminating all time blocked on disk is at most 19% across all workloads, as shown in Figure 3. The y-axis in Figure 3 describes the relative reduction in job completion time; a reduction of 0.1 means that the job could complete 10% faster as a result of eliminating time blocked on the disk. The figure illustrates the distribution over jobs, including all trials of each job in each workload. Boxes depict 25th, 50th, and 75th percentiles; whiskers

[5] This measurement includes only time blocked on disk requests, and does not include CPU time spent deserializing byte buffers into Java objects. This time is sometimes considered disk I/O because it is a necessary side effect of storing data outside of the JVM; we consider only disk hardware performance here, and discuss serialization time separately in §3.5.



(a) Big data benchmark, disk (b) Big data benchmark, memory

(c) TPC-DS, disk (d) TPC-DS, in-memory

(e) Production, in-memory

Figure 4: Comparison of original runtimes of Spark jobs to runtimes when all time blocked on the disk has been eliminated.

depict 5th and 95th percentiles. The variance stems from the fact that different jobs are affected differently by disk. The on-disk queries in the production workload are not shown in Figure 3 because, as described in §2.3.1, instrumenting time to read input data required adding instrumentation to HDFS, which was not possible to do for the production cluster. We show the same results in Figure 4, which instead plots the absolute runtime originally and the absolute runtime once time blocked on disk has been eliminated. The scatter plots illustrate that long jobs are not disproportionately affected by time reading data from disk.

For in-memory workloads, the median improvement from eliminating all time blocked on disk is 2-5%; the fact that this improvement is non-zero is because even in-memory workloads store shuffle data on disk.

A median improvement in job completion time of 19% is a respectable improvement in runtime, but is lower than we expected for workloads that read input data and store

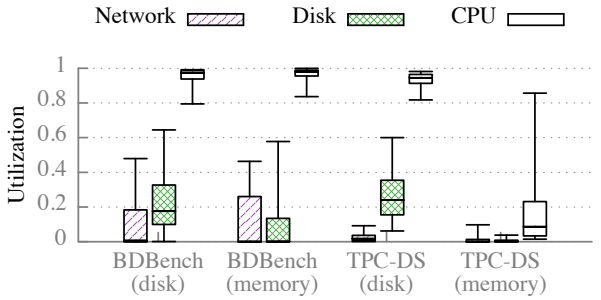


Figure 5: Average network, disk, and CPU utilization while tasks were running. CPU utilization is the total fraction of non-idle CPU milliseconds while the task was running, divided by the eight total cores on the machine. Network utilization is the bandwidth usage divided by the machine's 1Gbps available bandwidth. All utilizations are obtained by reading the counters in the /proc file system at the beginning and end of each task. The distribution is across all tasks in each workload, weighted by task duration.

output data on-disk. The following two subsections make more sense of this number, first by considering the effect of our hardware setup, and then by examining alternate metrics to put this number in the context of how tasks spend their time.

3.2 How does hardware configuration affect these results?

Hardware configuration impacts blocked time: tasks run on machines with fewer disks relative to the number of CPU cores would have spent more time blocked on disk, and vice versa. In its hardware recommendations for users purchasing Hadoop clusters, one vendor recommends machines with at least a 1:3 ratio of disks to CPU cores [31]. In 2010, Facebook's Hadoop cluster included machines with between a 3:4 and 3:1 ratio of disks to CPU cores [16]. Thus, our machines, with a 1:4 ratio of disks to CPU cores, have relatively under-provisioned I/O capacity. As a result, I/O may appear *more* important in our measurements than it would in the wild, so our results on time blocked on disk represent even more of an upper bound on the importance of I/O.

The second aspect of our setup that affects results is the number of concurrent tasks run per machine; we run one task per core, consistent with the Spark default.

3.3 How does disk utilization compare to CPU utilization?

Our result that eliminating time blocked on disk I/O can only improve job completion time by a median of at most 19% suggests that jobs may be CPU bound. Unfortunately, we cannot use blocked time analysis to understand the importance of compute time, because we cannot measure when task I/O is blocked waiting for computation. The operating system often performs disk

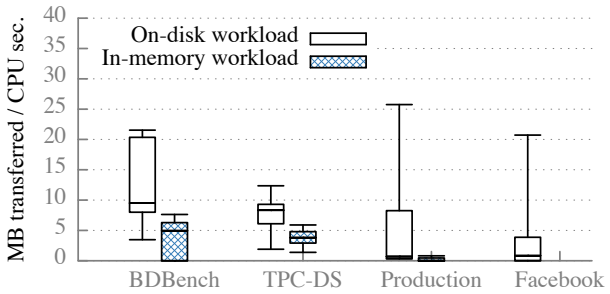


Figure 6: Average megabytes transferred to or from disk per non-idle CPU second for all jobs we ran. The median job transfers less than 10 megabytes to/from disk per CPU second; given that effective disk throughput was approximately 90 megabytes per second per disk while running our benchmarks, this demand can easily be met by the two disks on each machine.

I/O in the background, while the task is also using the CPU. Measuring when a task is using only a CPU versus when background I/O is occurring is thus difficult, and is further complicated by the fact that all Spark tasks on a single machine run in the same process.

Because we can't use blocked time analysis to understand the importance of CPU use, we instead examine CPU and disk utilization. Figure 5 plots the distribution across all tasks in the big data benchmark and TPC-DS benchmark of the CPU utilization compared to disk utilization. For this workload, the plot illustrates that on average, queries are more CPU bound than I/O bound. Hence, tasks are likely blocked waiting on computation to complete more often than they are blocked waiting for disk I/O. On clusters with more typical ratios of disk to CPU use, the disk utilization will be even lower relative to CPU utilization.

3.4 Sanity-checking our results against production traces

The fact that the disk is not the bottleneck in our workloads left us wondering whether our workloads were unusually CPU bound, which would mean that our results were not broadly representative. Unfortunately, production data analytics traces available to us do not include blocked time or disk utilization information. However, we were able to use aggregate statistics about the CPU and disk use of jobs to compare the I/O demands of our workloads to the I/O demands of larger production workloads. In particular, we measured the I/O demands of our workloads by measuring the ratio of data transferred to disk to non-idle CPU seconds:

$$\text{MB / CPU second} = \frac{\text{Total MB transferred to/from disk}}{\text{Total non-idle CPU seconds}}$$

This metric is imperfect because it looks at the CPU and disk requirements of the job as a whole, and does not account for variation in resource usage during a job. Nonetheless, it allows us to understand how the aggregate disk demands of our workloads compare to large-scale production workloads.

Using this metric, we found that the I/O demands of the three workloads we ran do not differ significantly from I/O demands of production workloads. Figure 6 illustrates that for the benchmarks and production workload we instrumented, the median MB / CPU second is less than 10. Figure 6 also illustrates results for the trace from Facebook's Hadoop cluster. The MB / CPU second metric is useful in comparing to Hadoop performance because it relies on the volume of data transferred to disk and the CPU milliseconds to characterize the job's demand on I/O, so abstracts away many inefficiencies in Hadoop's implementation (for example, it abstracts away the fact that a CPU-bound query may take much longer than the CPU milliseconds used due to inefficient resource use). The number of megabytes transferred to disk per CPU second is lower for the Facebook workload than for our workloads: the median is just 3MB/s, compared to a median of 9MB/s for the big data benchmark on-disk workload and 8MB/s for the TPC-DS workload.

We also examined aggregate statistics published about Microsoft's Cosmos cluster and Google's MapReduce cluster, shown in Tables 2 and 3. Unlike the Facebook trace, those statistics do not include a measurement of the CPU time spent by jobs, and instead quote the total time that tasks were running, aggregated across all jobs [23]. In computing the average rate at which tasks transfer data to and from disk, we assume that the input data is read once from disk, intermediate data is written once (by map tasks that generate the data) and read once (by reduce tasks that consume the data), and output data is written to disk three times (assuming the industry standard triply-replicated output data). As shown in Tables 2 and 3, based on this estimate, Google jobs transfer an average of 0.787 to 1.47 MB/s to disk, and Microsoft jobs transfer an average of 6.61 to 10.58 MB/s to disk. These aggregate numbers reflect an estimate of *average* I/O use so do not reflect tail behavior, do not include additional I/O that may have occurred (e.g., to spill intermediate data), and are not directly comparable to our results because unlike the CPU milliseconds, the total task time includes time when the task was blocked on network or disk I/O.[6] The takeaway from these results should not be the precise value of these aggregate metrics, but rather that sanity checking our results against production traces does not lead us to believe that production workloads have dramatically different

[6] For the Google traces, the aggregate numbers are skewed by the fact that, at the time, a few MapReduce jobs that consumed significant resources were also very CPU intensive (in particular, the final phase of the indexing pipeline involved significant computation). These jobs also performed some additional disk I/O from within the user-defined map and reduce functions [23]. We lack sufficient information to quantify these factors, so they are not included in our estimate of MB/s.

	Aug. '04	Mar. '06	Sep. '07
Map input data (TB) [24]	3,288	52,254	403,152
Map output data (TB) [24]	758	6,743	34,774
Reduce output data (TB) [24]	193	2,970	14,018
Task years used [24]	217	2,002	11,081
Total data transferred to/from disk (TB)	5383	74,650	514,754
Avg. MB transferred to/from disk per task second (MB/s)	.787	1.18	1.47
Avg. Mb sent over the network per task second (Mbps)	1.34	1.61	1.44

Table 2: Disk use for all MapReduce jobs run at Google.

	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan
Input Data (PB) [11]	12.6	22.7	14.3	18.7	22.8	25.3	25.0	18.6	21.5
Intermediate Data (PB) [11]	0.66	1.22	0.67	0.76	0.73	0.86	0.68	0.72	1.99
Compute (years) [11]	49.1	88.0	51.6	60.6	73.0	84.1	88.4	96.2	79.5
Avg. MB read/written per task second	8.99	9.06	9.61	10.58	10.54	10.19	9.46	6.61	10.16
Avg. Mb shuffled per task second	3.41	3.52	3.29	3.18	2.54	2.59	1.95	1.90	6.35

Table 3: Disk use for a cluster with tens of thousands of machines, running Cosmos. Compute (years) describes the sum of runtimes across all tasks [12]. The data includes jobs from two days of each month; see [11] for details.

I/O requirements than the workloads we measure.

3.5 Why isn't disk I/O more important?

We were surprised at the results in §3.3 given the oft-quoted mantra that I/O is often the bottleneck, and also the fact that fundamentally, the computation done in data analytics job is often very simple. For example, queries 1a, 1b, and 1c in the big data benchmark select a filtered subset of a table. Given the simplicity of that computation, we would not have expected the query to be CPU bound. One reason for this result is that today's frameworks often store compressed data (in increasingly sophisticated formats, e.g. Parquet [1]), trading CPU time for I/O time. We found that if we instead ran queries on uncompressed data, most queries became I/O bound. A second reason that CPU time is large is an artifact of the decision to write Spark in Scala, which is based on Java: after being read from disk, data must be deserialized from a byte buffer to a Java object. Figure 7 illustrates the distribution of the total non-idle CPU time used by queries in the big data benchmark under 3 different scenarios: when input data, shuffle data, and output data are compressed and serialized; when input data, shuffle data, and output data are not deserialized but are decompressed; and when input and output data are stored as deserialized, in-memory objects (shuffle data must still be serialized in order to be sent over the network). The CDF illustrates that for some queries, as much as half of the CPU time is spent deserializing and decompressing data. This result is consistent with Figure 9 from the Spark paper, which illustrated that caching deserialized data significantly reduced job completion time relative to caching data that was still serialized.

Spark's relatively high CPU time may also stem from the fact that Spark was written Scala, as opposed to a lower-level language such at C++. For one query that we re-wrote in C++, we found that the CPU time reduced by a factor of more than 2×. Existing work has illustrated that writing analytics in C++ instead can significantly improve performance [22], and the fact that Google's MapReduce is written in C++ is an oft-quoted reason for its superior performance.

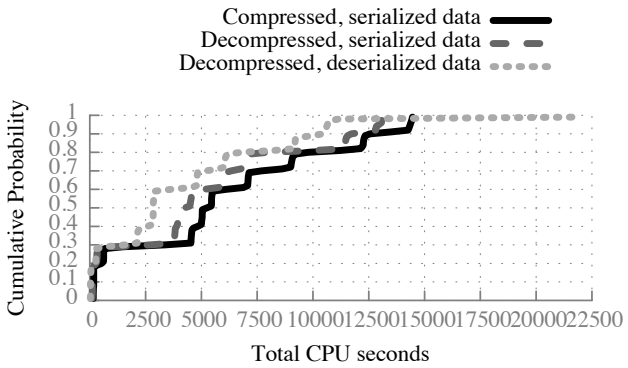


Figure 7: Comparison of total non-idle CPU milliseconds consumed by the big data benchmark workload, with and without compression and serialization.

3.6 Are these results inconsistent with past work?

Prior work has shown significant improvements as a result of storing input data for analytics workloads in memory. For example, Spark [51] was demonstrated to be 20× to 40× faster than Hadoop [51]. A close reading of that paper illustrates that much of that improvement came not from eliminating disk I/O, but from other improvements over Hadoop, including eliminating serialization time.

The PACMan work described improvements in average job completion time of more than a factor of two as a result of caching input data [9], which, similar to Spark, seems to potentially contradict our results. The 2× improve-

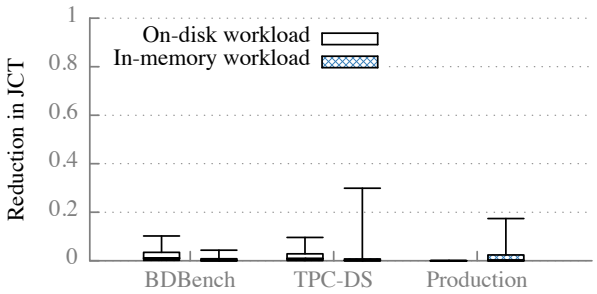


Figure 8: Improvement in job completion time as a result of eliminating all time blocked on network.

ments were shown for two workloads. The first workload was based on the Facebook trace, but because the Facebook trace does not include enough information to replay the exact computation used by the jobs, the authors used a mix of compute-free (jobs that do not perform any computation), sort, and word count jobs [12]. This synthetic workload was generated based on the assumption that analytics workloads are I/O bound, which was the prevailing mentality at the time. Our measurements suggest that jobs are not typically I/O bound, so this workload may not have been representative. The second workload was based on a Bing workload (rewritten to use Hive), where the $2\times$ improvement represented the difference in reading data from a serialized, on-disk format compared to reading de-serialized, in-memory data. As discussed in §3.5, serialization times can be significant, so the $2\times$ improvement likely came as much from eliminating serialization as it did from eliminating disk I/O.

3.7 Summary

We found that job runtime cannot improve by more than 19% as a result of optimizing disk I/O. To shed more light on this measurement,we compared resource utilization while tasks were running, and found that CPU utilization is typically close to 100% whereas median disk utilization is at most 25%. One reason for the relatively high use of CPU by the analytics workloads we studied is deserialization and compression; the shift towards more sophisticated serialization and compression formats has decreased the I/O and increased the CPU requirements of analytics frameworks. Because of high CPU times, optimizing hardware performance by using more disks, using flash storage, or storing serialized in-memory data will yield only modest improvements to job completion time; caching deserialized data has the potential for much larger improvements due to eliminating deserialization time.

Serialization and compression formats will inevitably evolve in the future, rendering the numbers presented in this paper obsolete. The takeaway from our measurements should be that CPU usage is currently much higher than disk use, and that detailed performance instrumentation like our blocked time analysis is critical to navigating the tradeoff between CPU and I/O time going forward.

4 How important is the network?

Researchers have used the argument that data-intensive application performance is closely tied to datacenter network performance to justify a wide variety of network optimizations [6, 14, 17–21, 27, 28, 41, 42, 48, 53]. We therefore expected to find that optimizing the network could yield significant improvements to job completion time.

4.1 How much time is spent blocked on network I/O?

To understand the importance of the network, we first use blocked time analysis to understand the largest possible improvement from optimizing the network. As shown in Figure 8, none of the workloads we studied could improve by a median of more than 2% as a result of optimizing network performance. We did not use especially high bandwidth machines in getting this result: the m2.4xlarge instances we used have a 1Gbps network link.

Our blocked time instrumentation for the network included time to read shuffle data over the network, and for on-disk workloads, the time to write output data to one local machine and two remote machines. Both of these times include disk use as well as network use, because disk and network are interlaced in a manner that makes them difficult to measure separately. As a result, 2% represents an upper bound on the possible improvement from network optimizations.

To shed more light on the network demands of the workloads we ran, Figure 5 plots the network utilization along with CPU and disk utilization. Consistent with the fact that blocked times are very low, median network utilization is lower than median disk utilization and much lower than median CPU utilization for all of the workloads we studied.

4.2 Sanity-checking our results against production traces

We were surprised at the low impact of the network on job completion time, given the large body of work targeted at improving network performance for analytics workloads. Similar to what we did in §3.3 to understand disk performance, we computed the network data sent per non-idle CPU second, to facilitate comparison to large-scale production workloads, as shown in Figure 9. Similar to what we found for disk I/O, the Facebook workload transfers less data over the network per CPU second than the workloads we ran. Thus, we expect that running our blocked time analysis on the Facebook workload would yield smaller potential improvements from network optimizations than for the workloads we ran. Tables 2 and 3 illustrate this metric for the Google and Microsoft traces, using the machine seconds or task seconds

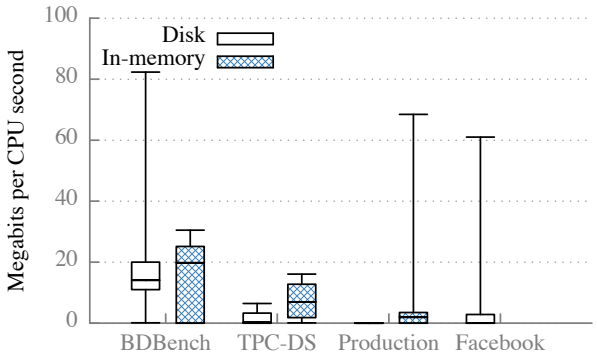


Figure 9: Megabits sent over the network per non-idle CPU second.

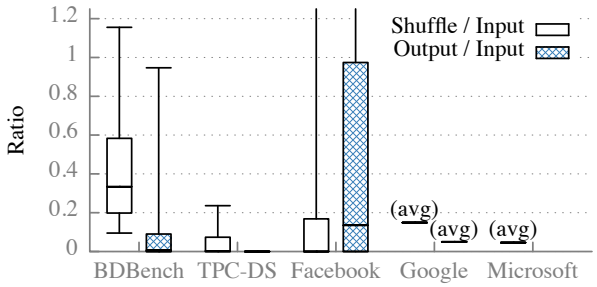


Figure 10: Ratio of shuffle bytes to input bytes and output bytes to input bytes. The median ratio of shuffled data to input data is less than 0.35 for all workloads, and the median ratio of output data to input data is less than 0.2 for all workloads.

in place of CPU milliseconds as with the disk results.[7] For Google, the average megabits sent over the network per machine second ranges from 1.34 to 1.61; Microsoft network use is higher (1.90-6.35 megabits are shuffled per task second) but still far below the network use seen in our workload.Thus, this sanity check does not lead us to believe that production workloads have dramatically different network requirements than the workloads we measure.

4.3 Why isn't the network more important?

One reason that network performance is relatively unimportant is that the amount of data sent over the network is often much less than the data transferred to disk, because analytics queries often shuffle and output much less data than they read. Figure 10 plots the ratio of shuffle data to input data and the ratio of output data to input data across our benchmarks, the Facebook trace (the production workload did not have sufficient instrumentation to capture these metrics), and for the Microsoft and Google aggregate data (averaged over all of the samples). Across all workloads, the amount of data shuffled is less than the amount of input data, by as much as a factor of 5-10, which is intuitive considering that data analysis often

[7]The Microsoft data does not include output size, so network data only includes shuffle data.

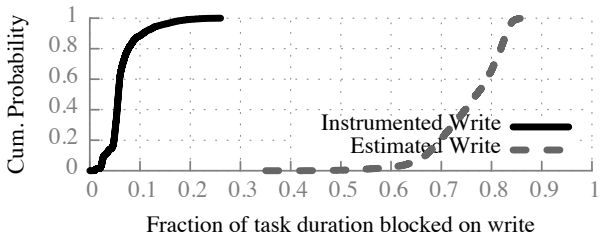


Figure 11: Cumulative distribution across tasks (weighted by task duration) of the fraction of task time spent writing output data, measured using our fine grained instrumentation and estimated using the technique employed by prior work [17]. The previously used metric far overestimates time spent writing output data.

centers around aggregating data to derive a business conclusion. In evaluating the efficacy of optimizing shuffle performance, many papers have used workloads with ratios of shuffled data to input data of 1 or more, which these results demonstrate is not widely representative.

4.4 Are these results inconsistent with past work?

Past work has reported high effects of the network on job completion time because the effect of the network has been estimated based on Hadoop traces rather than precisely measured. Many existing traces do not include sufficient metrics to understand the impact of the network. For example, Sinbad [17] asserted that writing output data can take a significant amount of time for analytics jobs, but used a heuristic to understand time to write output data: it defined the write phase of a task as the time from when the shuffle completes until the completion time of the task. This metric is an estimate, which was necessary because Hadoop does not log time blocked on output writes separately from time spent in computation. We instrumented Hadoop to log time spent writing output data and ran the big data benchmark (using Hive to convert SQL queries to Map Reduce jobs) and compared the result from the detailed instrumentation to the estimation previously used. Unfortunately, as shown in Figure 11, the previously used metric significantly overestimates time spent writing output data, meaning that the importance of the network was significantly overestimated.

A second problem with past estimations of the importance of the network is that they have conflated inefficiencies in Hadoop with network performance problems. One commonly cited work quotes the percent of job time spent in shuffle, measured using a Facebook Hadoop workload [19]. We repeated this measurement using the Facebook trace, shown in Table 4. Previous measurements looked at the fraction of jobs that spent a certain percent of time in shuffle (i.e., the first two lines of Table 4); by this metric, 16% of jobs spent more than 75% of time shuffling data. We dug into this data and found that a typical job that spends more than 75% of time in its shuf-

Shuffle Dur.	$<25\%$	25-49%	50-74%	$>=75\%$
% of Jobs	46%	20%	18%	16%
% of time	91%	7%	2%	1%
% of bytes	83%	16%	1%	0.3%

Table 4: The percent of jobs, task time, and bytes in jobs that spent different fractions of time in shuffle for the Facebook workload.

fle phase takes tens of seconds to shuffle kilobytes of data, suggesting that framework overhead and not network performance is the bottleneck. We also found that only 1% of all jobs spend less than 4 seconds shuffling data, which suggests that the Hadoop shuffle includes fixed overheads to, for example, communicate with the master to determine where shuffle data is located. For such tasks, shuffle is likely bottlenecked on inefficiencies and fixed overheads in Hadoop, rather than by network bottlenecks.

Table 4 includes data not reported in prior work: for each bucket, we compute not only the percent of jobs that fall into that bucket, but also the percent of total time and percent of total bytes represented by jobs in that category. While 16% of jobs spend more than 75% of the time in shuffle, these jobs represent only 1% of the total bytes sent across the network, and 0.3% of the total time taken by all jobs. This further suggests that the jobs reported as spending a large fraction of time in shuffle are small jobs for which the shuffle time is dominated by framework overhead rather than by network performance.

4.5 Summary

We found that, for the three workloads we studied, network optimizations can only improve job completion time by a median of at most 2%. One reason network performance has little effect on job completion time is that the data transferred over the network is a subset of data transferred to disk, so jobs bottleneck on the disk before bottlenecking on the network. We found this to be true in a cluster with 1Gbps network links; in clusters with 10Gbps or 100Gbps networks, network performance will be even less important.

Past work has found much larger improvements from optimizing network performance for two reasons. First, some past work has focused only on workloads where shuffle data is equal to the amount of input data, which we demonstrated is not representative of typical workloads. Second, some past work relied on estimation to understand trace data, which led to misleading conclusions about the importance of the network.

5 The Role of Stragglers

A straggler is a task that takes much longer to complete than other tasks in the stage. Because the stage cannot complete until all of its tasks have completed, straggler tasks can significantly delay the completion time of the stage and, subsequently, the completion time of the higher-level query. Past work has demonstrated that stragglers can be a significant bottleneck; for example, stragglers were reported to delay average job completion time by 47% in Hadoop clusters and by 29% in Dryad clusters [8]. Existing work has characterized some stragglers as being caused by data skew, where a task reads more input data than other tasks in a job, but otherwise does not characterize what causes some tasks to take longer than others [11, 26, 29]. This inability to explain the cause of stragglers has typically led to mitigation strategies that replicate tasks, rather than strategies that attempt to understand and eliminate the root cause of long task runtimes [8, 10, 39, 52].

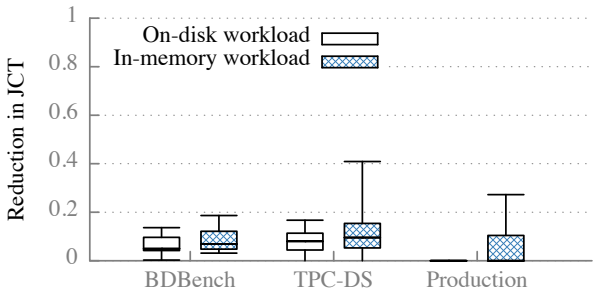


Figure 12: Potential improvement in job completion time as a result of eliminating all stragglers. Results are shown for the on-disk and in-memory versions of each benchmark workload.

5.1 How much do stragglers affect job completion time?

To understand the impact of stragglers, we adopt the approach from [8] and focus on a task's inverse progress rate: the time taken for a task divided by amount of input data read. Consistent with that work, we define the potential improvement from eliminating stragglers as the reduction in job completion time as a result of replacing the progress rate of every task that is slower than the median progress rate with the median progress rate. We use the methodology described in §2.3.2 to compute the new job completion time given the new task completion times. Figure 12 illustrates the improvement in job completion time as a result of eliminating stragglers in this fashion; the median improvement from eliminating stragglers is 5-10% for the big data benchmark and TPC-DS workloads, and lower for the production workloads, which had fewer stragglers.

5.2 Are these results inconsistent with prior work?

Some prior work has reported the effect of stragglers to be similar to what we found in our study. For example, based on a Facebook trace, Mantri described a median improvement of 15% as a result of eliminating all stragglers. Mantri had a larger overall improvements when deployed in production that stemmed not only from eliminating stragglers, but also from eliminating costly

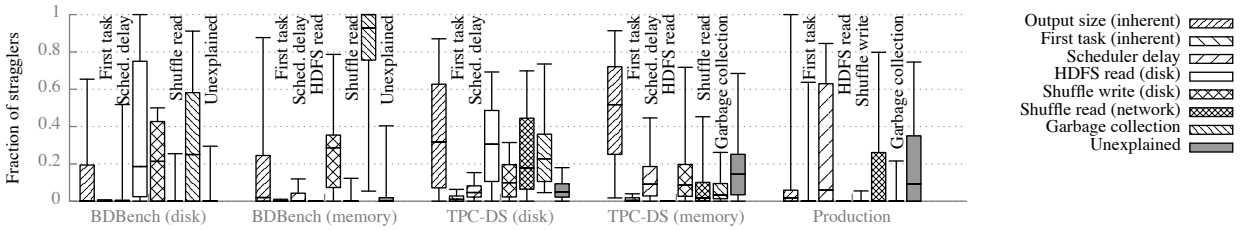


Figure 13: Our improved instrumentation allowed us to explain the causes behind most of the stragglers in the workloads we instrumented. This plot shows the distribution across all queries of the fraction of the query's stragglers that can be attributed to a particular cause.

recomputations [11].

Other work has reported larger potential improvements from eliminating stragglers: Dolly, for example, uses the same metric we used to understand the impact of stragglers, and found that eliminating stragglers could reduce job completion time by 47% for a Facebook trace and 29% for a Bing trace [8]. This difference may be due to the larger cluster size or greater heterogeneity in the studied traces. However, the difference can also be partially attributed to framework differences. For example, Spark has much lower task launch overhead than Hadoop, so Spark often breaks jobs into many more tasks, which can reduce the impact of stragglers [39]. Stragglers would have had a much larger impact for the workloads we ran if all of the tasks in each job had run as a single wave; in this case, the median improvement from eliminating stragglers would have been 23-41%. We also found that stragglers have become less important as Spark has matured. When running the same benchmark queries with an older version of Spark, many stragglers were caused by poor disk performance when writing shuffle data. Once this problem was fixed, the importance of stragglers decreased. These improvements to Spark may make stragglers less of an issue than previously observed, even on large-scale clusters.

5.3 Why do stragglers occur?

Prior work investigating straggler causes has largely relied on traces with coarse grained instrumentation; as a result, this work has been able to attribute stragglers to data skew and high resource utilization [11, 49], but otherwise has been unable to explain why stragglers occur. Our instrumentation allows us to describe the cause of more than 60% of stragglers in 75% of the queries we ran.

In examining the cause of stragglers, we follow previous work and define a straggler as a task with inverse progress rate greater than $1.5\times$ the median inverse progress rate for the stage. Unlike the previous subsection, we do not look at all tasks that take longer than the median progress rate, in order to focus on situations where there was a significant anomaly in a task's execution.

Many stragglers can be explained by the fact that the straggler task spends an unusually long amount of time in a particular part of task execution. We characterize a straggler as caused by X if it would not have been considered a straggler had X taken zero time for all of the tasks in the stage. We use this methodology to attribute stragglers to scheduler delay (time taken by the scheduler to ship the task to a worker and to process the task completion message), HDFS disk read time, shuffle write time, shuffle read time, and Java's garbage collection (which can be measured using Java's `GarbageCollectorMXBean` interface).

We attribute stragglers to two additional causes that require different methodologies. First, we attribute stragglers to output skew by computing the progress rate based on the amount of output data processed by the task instead of the amount of input data, and consider a straggler caused by output skew if the task is a straggler based on input progress rate but not based on output progress rate. Second, we find that some stragglers can be explained by the fact that they were among the first tasks in a stage to run on a particular machine. This effect may be caused by Java's just-in-time compilation: Java runtimes (e.g., the HotSpot JVM [32]) optimize code that has been executed more than a threshold number of times. We consider stragglers to be caused by the fact that they were a "first task" if they began executing before any other tasks in the stage completed on the same machine, and if they are no longer considered stragglers if compared to other "first tasks."

For each of these causes, Figure 13 plots the fraction of stragglers in each query explained by that cause. The distribution arises from differences in straggler causes across queries; for example, for the on-disk big data benchmark, in some jobs, all stragglers are explained by the time to read data from HDFS, whereas in other jobs, most stragglers can be attributed to garbage collection.

The graph does not point to any one dominant of stragglers, but rather illustrates that straggler causes vary across workloads and even within queries for a particular workload. However, common patterns are that garbage collection can cause most of the stragglers for some queries, and many stragglers can be attributed to long times spent reading to or writing from disk (this is not inconsistent with our earlier results showing a 19% median improvement from eliminating disk: the fact that some straggler tasks are caused by long times spent

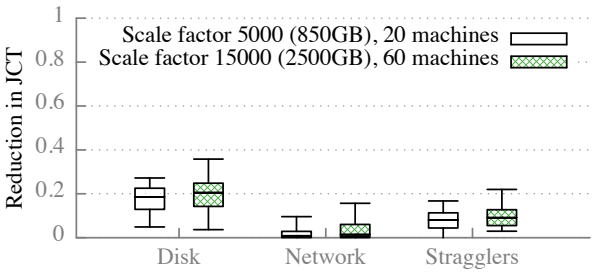


Figure 14: Improvement in job completion time as a result of eliminating disk I/O, eliminating network I/O, and eliminating stragglers, each shown for two different trials of the on-disk TPC-DS workload using different scale factors and cluster sizes.

blocked on disk does not necessarily imply that overall, eliminating time blocked on disk would yield a large improvement in job completion time). Another takeaway is that many stragglers are caused by inherent factors like output size and running before code has been jitted, so cannot be alleviated by straggler mitigation techniques.

5.4 Improving performance by understanding stragglers

Understanding the root cause behind stragglers provides ways to improve performance by mitigating the underlying cause. In our early experiments, investigating straggler causes led us to find that the default file system on the EC2 instances we used, ext3, performs poorly for workloads with large numbers of parallel reads and writes, leading to stragglers. By changing the filesystem to ext4, we fixed stragglers and reduced median task time, reducing query runtime for queries in the big data benchmark by $17-58\%$. Many of the other stragglers we observed could potentially be reduced by targeting the underlying cause, for example by allocating fewer objects (to target GC stragglers) or consolidating map output data into fewer files (to target shuffle write stragglers). Understanding these causes allows for going beyond duplicating tasks to mitigate stragglers.

6 How Does Scale Affect Results?

The results in this paper have focused on one cluster size for each of the benchmarks run. To understand how our results might change in a much larger cluster, we ran the TPC-DS on-disk workload on a cluster with three times as many machines and using three times more input data. Figure 14 compares our key results on the larger cluster to the results from the 20-machine cluster described in the remainder of this paper, and illustrates that the potential improvements from eliminating disk I/O, eliminating network I/O, and eliminating stragglers on the larger cluster is comparable to the corresponding improvements on the smaller cluster.

7 Conclusion

This paper undertook a detailed performance study of three workloads, and found that for those workloads, jobs are often bottlenecked on CPU and not I/O, network performance has little impact on job completion time, and many straggler causes can be identified and fixed. These findings should not be taken as the last word on performance of analytics frameworks: our study focuses on a small set of workloads, and represents only one snapshot in time. As data-analytics frameworks evolve, we expect bottlenecks to evolve as well. As a result, the takeaway from this work should be the importance of instrumenting systems for blocked time analysis, so that researchers and practitioners alike can understand how best to focus performance improvements. Looking forward, we argue that systems should be built with performance understandability as a first-class concern. Obscuring performance factors sometimes seems like a necessary cost of implementing new and more complex optimizations, but inevitably makes understanding how to optimize performance in the future much more difficult.

8 Acknowledgments

We are thankful to many Databricks employees who made the inclusion of production data possible: Aaron Davidson, Ali Ghodsi, Lucian Popa, Ahir Reddy, Ion Stoica, Patrick Wendell, Reynold Xin, and especially Andy Konwinski. We are also thankful to Michael Armbrust, for helping to configure and debug SparkSQL; to Yanpei Chen, Radhika Mittal, Amy Ousterhout, Aurojit Panda, and Shivaram Venkataraman, for providing helpful feedback on drafts of this paper; and to Mosharaf Chowdhury, for helping to analyze Facebook trace data. Finally, we thank our shepherd, Venugopalan Ramasubramanian, for helping to shape the final version of the paper.

This research is supported in part by a Hertz Foundation Fellowship, NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

References

[1] Apache Parquet. `http://parquet.incubator.apache.org/`.

[2] Common Crawl. `http://commoncrawl.org/`.

[3] Databricks. `http://databricks.com/`.

[4] Spark SQL. `https://spark.apache.org/sql/`.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In Proc. SOSP, 2003.

[6] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proc. NSDI, 2010.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In Proc. EuroSys, 2011.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In Proc. NSDI, 2013.

[9] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In Proc. NSDI, 2012.

[10] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In Proc. NSDI, 2014.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In Proc. OSDI, 2010.

[12] G. Ananthanarayayan. Personal Communication, February 2015.

[13] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[14] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In Proc. SIGCOMM, 2011.

[15] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In Proc. SOSP, 2004.

[16] D. Borthakur. Facebook has the world's largest Hadoop cluster! `http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html`, May 2010.

[17] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-intensive Clusters. In Proc. SIGCOMM, 2013.

[18] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In Proc. HotNets, 2012.

[19] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In Proc. SIGCOMM, 2011.

[20] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In Proc. SIGCOMM, 2014.

[21] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In Proc. NSDI, 2012.

[22] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. CoRR, 2014.

[23] J. Dean. Personal Communication, February 2015.

[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. CACM, 51(1):107–113, Jan. 2008.

[25] J. Erickson, M. Kornacker, and D. Kumar. New SQL Choices in the Apache Hadoop Ecosystem: Why Impala Continues to Lead. `http://goo.gl/evDBfy`, 2014.

[26] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In Proc. ICDE, pages 522–533, 2012.

[27] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In Proc. OSDI, 2012.

[28] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In Proc. NSDI, 2013.

[29] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In Proc. SIGMOD, pages 25–36, 2012.

[30] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Reliable, Memory Speed Storage for Cluster Computing Frameworks. In Proc. SoCC, 2014.

[31] K. O'Dell. How-to: Select the Right Hardware for Your New Hadoop Cluster. `http://goo.gl/INds4t`, August 2013.

[32] Oracle. The Java HotSpot Performance Engine Architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[33] K. Ousterhout. Display filesystem read statistics with each task. https://issues.apache.org/jira/browse/SPARK-1683.

[34] K. Ousterhout. Shuffle read bytes are reported incorrectly for stages with multiple shuffle dependencies. https://issues.apache.org/jira/browse/SPARK-2571.

[35] K. Ousterhout. Shuffle write time does not include time to open shuffle files. https://issues.apache.org/jira/browse/SPARK-3570.

[36] K. Ousterhout. Shuffle write time is incorrect for sort-based shuffle. https://issues.apache.org/jira/browse/SPARK-5762.

[37] K. Ousterhout. Spark big data benchmark and TPC-DS workload traces. http://eecs.berkeley.edu/~keo/traces.

[38] K. Ousterhout. Time to cleanup spilled shuffle files not included in shuffle write time. https://issues.apache.org/jira/browse/SPARK-5845.

[39] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In Proc. HotOS, 2013.

[40] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In Proc. SIGMOD, 2009.

[41] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network in Cloud Computing. In Proc. SIGCOMM, 2012.

[42] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In Proc. NSDI, 2012.

[43] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An I/O-efficient MapReduce. In Proc. SoCC, 2012.

[44] K. Sakellis. Track local bytes read for shuffles - update UI. https://issues.apache.org/jira/browse/SPARK-5645.

[45] Transaction Processing Performance Council (TPC). TPC Benchmark DS Standard Specification. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf, 2012.

[46] UC Berkeley AmpLab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, February 2014.

[47] A. Wang and C. McCabe. In-memory Caching in HDFS: Lower Latency, Same Great Taste. In Presented at Hadoop Summit, 2014.

[48] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In Proc. SIGCOMM, 2012.

[49] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In Proc. SoCC, 2014.

[50] L. Yi, K. Wei, S. Huang, and J. Dai. Hadoop Benchmark Suite (HiBench). https://github.com/intel-hadoop/HiBench, 2012.

[51] 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 Proc. NSDI, 2012.

[52] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In Proc. OSDI, 2008.

[53] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In Proc. NSDI, 2012.