Scale-up vs Scale-out for Hadoop: Time to rethink?

Raja Appuswamy[*] Christos Gkantsidis, Dushyanth Narayanan,
Orion Hodson, and Antony Rowstron

Microsoft Research, Cambridge, UK

Abstract

In the last decade we have seen a huge deployment of cheap clusters to run data analytics workloads. The conventional wisdom in industry and academia is that scaling out using a cluster of commodity machines is better for these workloads than scaling up by adding more resources to a single server. Popular analytics infrastructures such as Hadoop are aimed at such a cluster scale-out environment.

Is this the right approach? Our measurements as well as other recent work shows that the majority of real-world analytic jobs process less than 100 GB of input, but popular infrastructures such as Hadoop/MapReduce were originally designed for petascale processing. We claim that a single "scale-up" server can process each of these jobs and do as well or better than a cluster in terms of performance, cost, power, and server density. We present an evaluation across 11 representative Hadoop jobs that shows scale-up to be competitive in all cases and significantly better in some cases, than scale-out. To achieve that performance, we describe several modifications to the Hadoop runtime that target scale-up configuration. These changes are transparent, do not require any changes to application code, and do not compromise scale-out performance; at the same time our evaluation shows that they do significantly improve Hadoop's scale-up performance.

1 Introduction

Data analytics, and in particular, MapReduce [14] and Hadoop [4] have become synonymous with the use of cheap commodity clusters using a distributed file system that utilizes cheap unreliable local disks. This is the standard scale-out thinking that has underpinned the infrastructure of many companies. Clearly large clusters of commodity servers are the most cost-effective way to process exabytes, petabytes, or multi-terabytes of data. Nevertheless, we ask: is it time to reconsider the scale-out versus scale-up question?

First, evidence suggests that the *majority* of analytics jobs do not process huge data sets. For example, at least two analytics production clusters (at Microsoft and Yahoo) have median job input sizes under 14 GB [16, 28], and 90% of jobs on a Facebook cluster have input sizes under 100 GB [2].

Second, hardware price trends are beginning to change performance points. Today's servers can affordably hold 100s of GB of DRAM and 32 cores on a quad socket motherboard with multiple high-bandwidth memory channels per socket. DRAM is now very cheap, with 16 GB DIMMs costing around $130, meaning 192 GB costs less than half the price of a dual-socket server and 512 GB costs 20% the price of a high-end quad-socket server. Storage bottlenecks can be removed by using SSDs or with a scalable storage back-end such as Amazon S3 [1] or Azure Storage [9, 31]. The commoditization of SSDs means that $2,000 can build a storage array with multiple GB/s of throughput. Thus a scale-up server can now have substantial CPU, memory, and storage I/O resources and at the same time avoid the communication overheads of a scale-out solution. Moore's law continues to improve many of these technologies, at least for the immediate future.

In this paper, we ask whether it is better to scale up using a well-provisioned single server or to scale out using a commodity cluster. For the world of analytics in general and Hadoop MapReduce in particular, this is an important question. Today the default assumption for Hadoop jobs is that scale-out is the only configuration

[*] Work done while on internship from Vrije Universiteit Amsterdam

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2523629

that matters. Scale-up performance is ignored and in fact Hadoop performs poorly in a scale-up scenario. In this paper we re-examine this question across a range of analytic workloads and using four metrics: *performance*, *cost*, *energy*, and *server density*.

This leads to a second question: how to achieve good scale-up performance, *without compromising scaling-out* for workloads that need it. Given the popularity of Hadoop and the rich ecosystem of technologies that have been built around it, we took the approach of transparently optimizing the Hadoop runtime for the scale-up case, without losing the ability to scale out.

We start by examining real-world analytics jobs, and argue that many jobs are sub tera-scale, and hence orders of magnitude smaller than the the peta-scale jobs that motivated the scale-out design of MapReduce and Hadoop. These are candidate jobs to run in a scale-up server. By default Hadoop performs poorly in a scale-up configurations. We describe simple, transparent optimizations that remove bottlenecks and improve both scale-out and scale-up performance. We present an evaluation that shows that, with these optimizations, scale-up is an extremely competitive option for sub tera-scale jobs. A scale-up server with 32 cores outperforms an 8-node scale-out configuration, also with 32 cores, on 9 out of 11 jobs and is within 11% for the other two. Larger scale-out clusters obviously improve performance but increase cost, power, and space usage. Compared to a 16-node scale-out cluster, a scale-up server provides better performance per dollar for all jobs. Moreover, scale-up performance per watt and per rack unit are *significantly better* for all jobs compared to either cluster.

Our results have implications both for data center provisioning and for software infrastructures. Broadly, we believe it is cost-effective for providers supporting "big data" analytic workloads to provision "big memory" servers (or a mix of big and small servers) with a view to running jobs entirely within a single server. Second, it is then important that the Hadoop infrastructure support both scale-up and scale-out efficiently and transparently to provide good performance for both scenarios.

The rest of this paper is organized as follows. Section 2 shows an analysis of job sizes from real-world MapReduce deployments that demonstrates that most jobs are under 100 GB in size. It then describes 11 example Hadoop jobs across a range of application domains that we use as concrete examples in this paper. Section 3 briefly describes the optimizations and tuning required to deliver good scale-up performance on Hadoop. Section 4 compares scale-up and scale-out for Hadoop for the 11 jobs on several metrics: performance, cost, power, and server density. Section 5 discusses some implications for analytics in the cloud as well as the crossover point between scale-up and scale-out. Section 6 describes related work, and Section 7 concludes the paper.

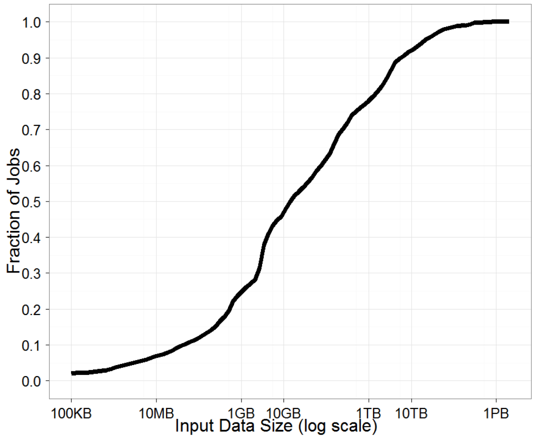


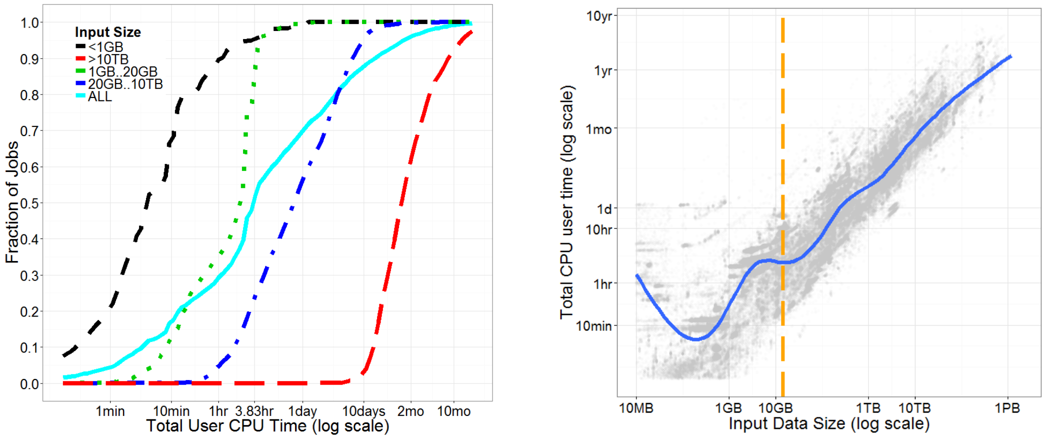
Figure 1: Distribution of input job sizes for a large analytics cluster

2 Job sizes and example jobs

To better understand the size and resource requirements of typical analytics jobs, we analyzed 174,000 jobs submitted to a production analytics cluster in Microsoft in a single month in 2011. Figure 1 shows the CDF of input data sizes across these jobs. The median job input data set size was less than 14 GB, and 80% of the jobs had an input size under 1 TB. Thus although there are multi-terabyte and petabyte-scale jobs which would require a scale-out cluster, these are the minority.

The CPU requirements of sub tera-scale jobs are also correspondingly modest. Figure 2(a) shows the distribution of total CPU time across jobs. The median job requires 3.83 hr of CPU time, i.e. 7.2 min per core on 32 cores assuming perfect parallelism. Figure 2(b) shows the same data as a log-log scatter plot of CPU size and input data size. The solid blue line shows a spline fit to the data and the dashed vertical line is the median input size. Thus broadly, whether judged on input data size or CPU time, a large number of jobs would seem to fit comfortably within a "big memory" scale-up server with 32 cores and 512 GB of memory. Especially given the ever-decreasing price per gigabyte of DRAM, it seems worthwhile to consider whether such jobs are better fitted to a scale-up server rather than a scale-out cluster.

Of course, these job sizes are from a single cluster running a MapReduce like framework. However we believe our broad conclusions on job sizes are valid for MapReduce installations in general and Hadoop installations in particular. For example, Elmeleegy [16] analyzes Hadoop jobs from Yahoo's production clusters. Unfortunately, the median input data set size is not given but, from the paper we can estimate that the



(a) CDF of CPU time

(b) CPU time vs. input data size

Figure 2: Statistics of total CPU time for jobs from a large analytics cluster.

median job input size is less than 12.5 GB.[1] Ananthanarayanan et al. [2] show that Facebook jobs follow a power-law distribution with small jobs dominating; their graphs suggest that at least 90% of the jobs have input sizes under 100 GB. Chen et al. [12] present a detailed study of Hadoop workloads for Facebook as well as 5 Cloudera customers. Their graphs also show that a very small minority of jobs achieves terabyte scale or larger and the paper claims explicitly that “most jobs have input, shuffle, and output sizes in the MB to GB range.”

2.1 Experimental Workloads

The statistics presented so far are extracted from jobs running in production clusters: we are not able to compare their performance directly on scale-up versus scale-out infrastructures. In order to do so, we have collected a smaller set of jobs from a variety of sources. We focused on jobs for which both code and representative input datasets were available, and which are typical across a range of applications. We describe these jobs in the rest of the section, and use them as benchmarks in our evaluation in Section 4. Broadly the jobs we examine can be classified by application domain into log analysis, query-based data analytics, machine learning, and searching/indexing. Table 1 shows the amount of input, shuffle, and output data for each job.

Log processing A very common use of Hadoop and MapReduce is to process text logs. We use two such jobs from a real-world compute platform consisting of tens of thousands of servers.[2] Users issue tasks to the system that spawn processes on multiple servers which consume resources on each server. The system writes logs that capture statistics such as CPU, I/O and other resource utilization on these servers. Administrators can process these logs to extract both fine- and coarse-grained information about resource usage. In this paper we use two such jobs. The *FindUserUsage* job processes one such log to aggregate resource. The *ComputeIOVolumes* job processes two log files to combine process-level and task-level information, and hence find the amount of input and intermediate data I/O done by each task.

Analytic queries We use three analytical queries from a benchmark that was created for comparing the performance of Hadoop against a parallel DBMS [24]. These tasks mimic query processing on a cluster of HTML documents gathered by a web crawl. The input data to these tasks consists of two tables, each sharded across a large number of files. Both data sets consist of random records generated by a custom data generator.

Records in the *PageRank* table associate each unique URL with a page rank. The *UserVisits* table contains one record per user visit, which contains the source IP, the date of visit, ad revenue generated, user agent, search word used, URL visited, country code, and language code. We use three queries (converted into Hadoop jobs) based on these two tables in our benchmark suite.

[1] The paper states that input block sizes are usually 64 or 128 MB with one map task per block, that over 80% of the jobs finish in 10 minutes or less, and that 70% of *these* jobs very clearly use 100 or fewer mappers (Figure 2 in [16]). Therefore conservatively assuming 128 MB per block, 56% of the jobs have an input data set size of under 12.5 GB.

[2] The two jobs originally run in an internal Cosmos cluster. We rewritten them (from Scope [10]) to Hadoop and run them (on the same inputs) in our local clusters.

Job	Input	Shuffle	Output
Log processing			
FindUserUsage	41 GB	4 MB	36 KB
ComputeIOVolumes	94 GB	157 MB	30 MB
Analytic queries			
Select Task[1]	41 GB	0 KB	0 KB
Aggregate Task	70 GB	5 GB	51 MB
Join Task[2]	113 GB	10 GB	4.3 GB
TeraSort			
10 GB TeraSort	11 GB	11 GB	11 GB
50 GB TeraSort	54 GB	54 GB	54 GB
Pig Cogroup	5 GB	7 GB	131 GB
Mahout[3]			
k-Means	72 MB	N/A	N/A
Wikipedia	432 MB	N/A	N/A
Indexing	8.9 GB	34 GB	26 GB

[1] Select produces negligible intermediate and output data.
[2] Sum of input, output, and shuffle bytes across three stages.
[3] Mahout jobs iterate over many map-reduce stages and hence we only measure input data size.

Table 1: Summary of jobs used with input, shuffle, and output data sizes.

The *Select* task finds the top 1% of URLs by page rank from the PageRank table. The *Aggregate* task calculates the total ad revenue per source IP address, from the UserVisits table. The *Join* takes both data sets as input. It finds the source IP addresses that generates the most revenue within a particular date range and then computes the average page rank of all the pages visited by those IP addresses in that interval. It is computed in three phases, each of which is a MapReduce computation. We converted each phase into a separate Hadoop job in our benchmark suite.

TeraSort TeraSort is a standard benchmark for data processing platforms. We use the Hadoop version of TeraSort. Despite its name, TeraSort can be used to sort different sizes of input data. In this paper we consider input sizes of 10 GB, 50 GB, and 100 GB.

Mahout machine learning tasks In recent years, Mahout has emerged as a popular framework to simplify the task of implementing scalable machine learning algorithms by building on the Hadoop ecosystem [5]. We used two standard machine learning algorithms implemented in Mahout in our benchmark suite.

The *Clustering* benchmark is based on k-means clustering, provided as part of the Mahout machine learning library [22]. This is an iterative machine-learning algorithm implemented as a series of map-reduce rounds. The input to the algorithm is a set of points represented as d-dimensional vectors, and an initial set of cluster centroids (usually chosen randomly). In each round, the mappers map each vector to the nearest centroid, and the reducers recompute the cluster centroid as the average of the vectors currently in the cluster. We use Mahout's k-means algorithm to implement a tag suggestion feature for Last.fm, a popular Internet radio site. The basic idea is to group related tags together so that the website can assist users in tagging items by suggesting relevant tags. We use the raw tag counts for the 100 most frequently occurring tags generated by Last.fm users as our input data set. The data set contains 950,000 records accounting for a total of 7 million tags assigned to over 20,000 artists.

The *Recommendation* benchmark is based on the recommendation algorithm also provided with Mahout. It is also iterative with multiple map-reduce rounds. We use this algorithm on the publicly available Wikipedia data set to compute articles that are similar to a given one. The output of the recommendation algorithm is a list of related articles that have similar outlinks as the given article and hence are related to it and should possibly be linked with it. The Wikipedia data set contains 130 million links, with 5 million sources and 3 million sinks. Due to the limited SSD space of our 8-machine cluster configuration (Section 4.1), we had to limit the maximum number of outlinks from any given page to 10, and, hence, our data set had 26 million links.

Pig Apache Pig is another platform that has gained widespread popularity in recent years as a platform that fosters the use of a high level language to perform large scale data analytics. We use a "co-group" query from a standard list of queries used as performance tests for Pig [6].[3] This takes two tables as input and computes the cross-product of the two tables. It differs to the other jobs in that the final output is much larger than either the input or shuffle data, since the cross-product is computed at the reducers.

Indexing Building an inverted index from text data was part of the original motivation for MapReduce [14], and inverted indices are commonly used by many search engines. We implemented a simple indexer as a Hadoop MapReduce job that takes a set of text files as input and outputs a full index that maps each unique word to a list of all occurrences of that word (specified as file name + byte offset).

[3] Other queries on this list essentially replicate the selection, aggregation, and projection tasks already included in our benchmark suite.

Optimization	Scale-up	Scale-out
SSD storage	Yes	Yes
Local FS for input	Yes	No
Optimize concurrency	Yes	Yes
Enable OOB heartbeat	Yes	Yes
Optimize heap size	Yes	Yes
Local FS for intermediate data	Yes	No
Unrestricted shuffle	Yes	No
RAMdisk for intermediate data	Yes	No

Table 2: Summary of Hadoop optimizations grouped as storage, concurrency, network, memory, and reduce-phase optimizations. All the optimizations apply to scale-up but only some to scale-out.

3 Optimizing for scale-up

In order to evaluate the relative merits of scale-up and scale-out for the jobs described in Section 2, we needed a software platform to run these jobs. The obvious candidate for scale-out is Hadoop; we decided to use Hadoop as our platform on a single scale-up server for three reasons. First, there are a large number of applications and infrastructures built on Hadoop, and thus there is a big incentive to stay within this ecosystem. Second, any scale-up platform will have to scale out as well, for applications that need to scale beyond a single big-memory machine. Finally, using the same base platform allows us an apples-to-apples performance comparison between the two configurations.

We first tuned and optimized Hadoop for good performance in the baseline, i.e., cluster scale-out, scenario. We then further optimized it to take advantage of features of the scale-up configuration, such as local file system storage and local RAMdisks. Table 2 lists the optimizations applied to Hadoop divided into five categories: storage, concurrency, network, memory, and reduce-phase optimizations. The use of the local file system and the reduce-phase optimizations are only applied to the scale-up configuration, because they are not applicable or detrimental to performance in the scale-out configuration. The concurrency and memory optimizations require tuning Hadoop parameters for the specific workload; all the other optimizations are workload-independent.

Here we describe each optimization. Section 4.4 has an experimental evaluation of the performance benefit of each optimization for the scale-up case.

Storage Our first step was to remove the storage bottleneck for both scale-up and scale-out configurations. In a default configuration with disks, Hadoop is I/O-bound and has low CPU utilization. We do not believe that this is a realistic configuration in a modern data center for either a scale-up or a scale-out system. Storage bottlenecks can easily be removed either by using SSDs or by using one of many scalable back-end solutions (SAN or NAS in the enterprise scenario, e.g. [23], or Amazon S3 / Windows Azure in the cloud scenario). In our experimental setup which is a small cluster we use SSDs for both the scale-up and the scale-out machines.

Even with SSDs, there is still substantial I/O overhead when using HDFS. While HDFS provides scalable and reliable data storage in a distributed Hadoop installation, we doubt its utility in scale-up installations for several reasons. A scalable storage back end can easily saturate the data ingest capacity of a scale-up compute node. Thus using the compute node itself to serve files via HDFS is unnecessary overhead. Alternatively, if data is stored locally on SSDs, modern file systems like ZFS and BTRFS already provide equivalent functionality to HDFS (e.g. check-summing) [8, 7], but without the associated throughput overheads, and can be retrofitted to work with a single-node Hadoop setup.

Hence for our scale-up configuration we store the inputs on SSDs and access them via the local file system. For the scale-out configuration, we also store the inputs on SSDs but we access them via HDFS. To improve performance of HDFS, we do not use replication.

Concurrency The next optimization adjusts the number of map and reduce tasks to be optimal for each job. The default way to run a Hadoop job on a single machine is to use the "pseudo-distributed" mode, which assigns a separate JVM to each task. We implemented a multi-threaded extension to Hadoop which allows us to run multiple map and reduce tasks as multiple threads within a single JVM. However, we found that when the number of tasks as well as the heap size for each tasks are well-tuned, there is no performance difference between the multi-threaded and pseudo-distributed mode.

Since our goal is to avoid unnecessary changes to Hadoop, we use the pseudo-distributed mode for the scale-up configuration and the normal distributed mode for the scale-out configuration. For the scale-out case, we tune the number of map and reducer slots *per job* (i.e. we experimented with various configurations per job and picked the best). For the scale-up case, we pick the optimal number of map and reduce slots and use it across all jobs. Per job, we also optimize the number of reducers used, in the same way that every expert user would have done.

Heartbeats Hadoop typically tracks task liveness through periodic heartbeats. When a task finishes much shorter than the heartbeat interval, the task tracker node will idle waiting for the next submission. "Out-of-band" heartbeats eliminate this waiting period and improve performance [13]. We enable them for both scale-up and scale-out configurations.

Heap size By default each Hadoop map and reduce task is run in a JVM with a 200 MB heap within which they allocate buffers for in-memory data. When the buffers are full, data is spilled to storage, adding overheads. We note that 200 MB per task leaves substantial amounts of memory unused on modern servers. By increasing the heap size for each JVM (and hence the working memory for each task), we improve performance. However too large a heap size causes garbage collection overheads, and wastes memory that could be used for other purposes (such as a RAMdisk). For the scale-out configurations, we found the optimal heap size for each job through trial and error. For the scale-up configuration we set a heap size of 4 GB per mapper/reducer task (where the maximum number of tasks is set to the number of processors) for all jobs.

Shuffle optimizations The next three optimizations speed up the shuffle (transferring data from mappers to reducers) on the scale-up configuration: they do not apply to scale-out configurations. First, we modified Hadoop so that shuffle data is transferred by writing and reading the local file system; the default is for reducers to copy the data from the mappers via http. However, we found that this still leads to underutilized storage bandwidth during the shuffle phase for our scale-up configuration, due to a restriction on the number of concurrent connections that is allowed. In a cluster, this is a reasonable throttling scheme to avoid a single node getting overloaded by copy requests. However it is unnecessary in a scale-up configuration with a fast local file system. Removing this limit substantially improves shuffle performance. Finally, we observed that the scale-up machine has substantial excess memory after configuring for optimal concurrency level and heap size. We use this excess memory as a RAMdisk to store intermediate data rather than using an SSD or disk based file system.

4 Evaluation

In this section, we will use the benchmarks and MapReduce jobs we described in Section 2 to perform an in-depth analysis of the performance of scale-up and scale-out Hadoop.

To understand the pros and cons of scaling up as opposed to scaling out, one needs to compare optimized implementations of both architectures side by side using several metrics (such as performance, performance/unit cost, and performance/unit energy) under a wide range of benchmarks. In this section we first describe our experimental setup. We then compare scale-up and scale-out across all our entire benchmark suite on several metrics. These results are based on applying all optimizations described in Section 3 to all the configurations.

	Workstation[1]	Server[1]
Base spec	E5520	4x E7-4820
- CPU	4 cores, HT, 2.3 GHz	32 cores, HT, 2.0 GHz
- Memory	12 GB	0 GB
- Disk	1 HDD (160 GB)	2 HDD (600 GB)
Base cost	$2130	$17380
SSD cost	$270[2]	$2160[2]
DRAM cost	none	$5440[2,3] (512GB)
Total cost	$2400	$24980
Power	154 W	877 W[4]

[1] Prices from Dell's website, `http://www.dell.com/`.
[2] Prices from `http://www.newegg.com`.
[3] 16 GB 240-Pin DDR3 1333 SDRAM.
[4] Measured with 512GB of RAM and all (HDD and SSD) drives connected.
[*] All processors are Intel Xeon

Table 3: Comparison of workstation (used for scale-out) and server (scale-up) configurations. All prices converted to US$ as of 8 August 2012.

We then look more closely at two of the jobs, FindUserUsage and TeraSort. For these two jobs we measure the individual phases — map, shuffle, and reduce — to understand better the scale-up/scale-out tradeoffs. We also measure the individual contributions of each of our optimization techniques on scale-up performance.

4.1 Hardware

The commodity scale-out cluster we used in this work consists of 16 data nodes and one name node. Each node is a Dell Precision T3500 Essential workstation with a 2.3 GHz E5520 Xeon quad-core processor (8 hyperthreads), 12 GB of memory, a 160 GB Western Digital Caviar Blue HDD, and a 32 GB Intel X25-E SSD. The HDD is used by the operating system; we use the SSD for our experiments (including HDFS and all input and output files). Each machine is also equipped with a 1Gbps NIC connected to a single switch, which can easily handle 1Gbps all-to-all communication. We used this cluster in two configurations: with 8 data nodes and with all 16 data nodes.

The scale-up machine is a 4-socket Dell PowerEdge 910 server with 4 8-core 2 GHz Intel Xeon E7-4820 processors for a total of 32 cores (64 hyperthreads). The server is also equipped with two RAID controllers. To this base configuration we added 512 GB of DRAM, two Hitachi UltraStar C10K600 HDDs (600 GB) in a RAID-1 configuration, and 8 240 GB Intel 520 Series SSDs in a RAID-0 configuration. The HDDs are used primarily for the OS image and the SSDs for Hadoop job data.

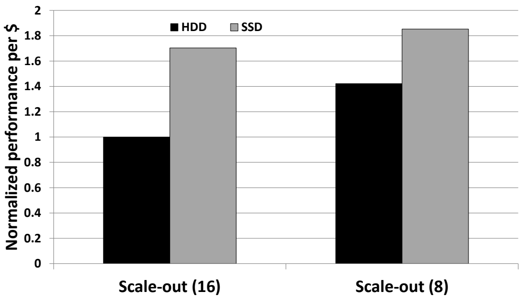


Figure 3: Performance/$ for scale-out running TeraSort with HDDs and SSDs, normalized to performance/$ of 16-node cluster with HDDs.

Table 3 summarizes the hardware configurations and their prices. The scale-up configuration costs around $25k,whereas the 8-machine and 16-machine scale-out configurations cost $19k and $38k respectively (at the time of acquisition), without counting the cost of the network switch that would be required in the scale-out case.

We use SSDs for both the scale-up and the scale-out server. This is based on our observation that without SSDs, many Hadoop jobs become disk-bound. Thus although SSDs add 13% to the base cost of the workstation-class machine in Table 3, they improve the overall performance/price ratio of the cluster. We verified this by running the 10 GB TeraSort benchmark on the 8- and 16-node configurations with and without SSDs. Figure 3 shows the performance/price ratio for the four configurations, normalized so that the 16-node cluster using HDDs has a value of 1. We see that SSDs do improve performance/$ for the scale-out configuration, and we use SSDs consistently for both scale-up and scale-out in our experiments.

Table 3 also shows the power draw under load of the two hardware platforms. We measured the power consumption offline using a stress test to ensure consistent results. We used the 10 GB TeraSort benchmark to generate load, since it uniformly stresses the resources of all the servers. We used a *Watts up? Pro* power meter which logs power usage statistics at a one second granularity in its internal memory during the job run, and derive the average power from this log. We ran this test both on the scale-up and the scale-out configuration. In the scale-out case, TeraSort loads all servers equally and thus we only measure power on a single server. We measured the power in both the 8-node and 16-node configurations and found no significant difference.

All configurations have all relevant optimizations enabled (Section 3 and Table 2).

4.2 Scale-up vs. scale-out

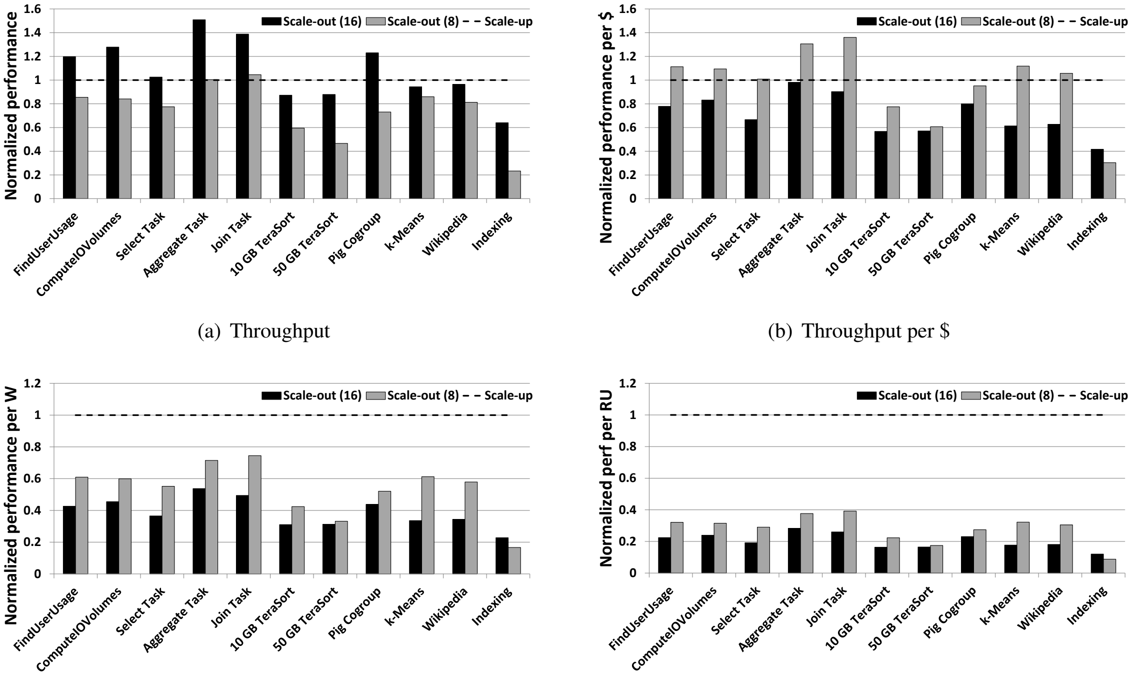
We ran all 11 jobs in all three configurations and measured their throughput, i.e. the inverse of job execution time. All results are the means of at least 4 runs of each job on each configuration.Figure 4(a) shows the results normalized so that the scale-up performance is always 1. We see that scale-up performs surprisingly well: better than the 8-machine cluster for all but two jobs and within 5% for those two.

When we double the cluster size from 8 to 16, scale-out performs better than scale-up for 6 jobs but scale-up is still significantly better for the other 5 jobs. In general, we find that scale-out works better for CPU-intensive tasks since there are more cores and more aggregate memory bandwidth. Scale-up works better for shuffle-intensive tasks since it has fast intermediate storage and no network bottleneck. Note that the Pig Cogroup job is CPU-intensive: a small amount of data is shuffled but a large cross-product is generated by the reducers.

Clearly adding more machines does improve performance; but at what cost? It is important to also consider performance per dollar. We derive performance per dollar by dividing raw performance by the capital/acquisition cost of the hardware (Table 3). In order to keep the analysis simple, here we do not consider other expenses like administration or cooling costs. In general we would expect these to be lower for a single machine.

Figure 4(b) shows the performance per dollar for the three configurations across all jobs, again normalized to set the scale-up platform at 1. Interestingly now the 8-node cluster does uniformly better than the 16-node cluster, showing that there are diminishing performance returns for scaling out even for these small clusters. Surprisingly, this is also true for the k-means and Wikipedia jobs whose core-computation is CPU-bound. The speed-up from 8 nodes to 16 is sublinear due to the overheads of frequent task management: every iteration of the algorithm involves setting up a new MapReduce round, which impacts the performance substantially. Scale-up is again competitive (though slightly worse) for map-intensive tasks and significantly better for the shuffle-intensive tasks, than either scale-out configuration.

Figure 4(c) shows normalized performance per watt, based on the power measurements reported in Table 3. This is an important metric as data centers are often power-limited; hence more performance per Watt means more computation for the same power budget. On this metric we see that scale-up is significantly better than either scale-out configuration across all jobs. We believe the results are pessimistic: they underestimate scale-out power by omitting the power consumed by a top-of-rack switch, and overestimate scale-up power by including a redundant power supply.



(a) Throughput

(b) Throughput per $

(c) Throughput per watt

(d) Throughput per rack unit

Figure 4: Scale-out performance on different metrics, normalized to scale-up performance for each of 11 jobs.

Finally, we look at server density. Like power, this is also an important consideration, as a higher server density means more computation for a given space budget. The scale-up machine is a "3U" form factor, i.e. it uses three rack units. Although our scale-out machines are in a workstation form factor, for this analysis we consider them to have a best-case 1U form factor. Figure 4(d) shows the performance per rack unit (RU). As with power, we see clearly that the scale-up machine outperforms scale-out across all jobs.

4.3 Phase-by-phase analysis

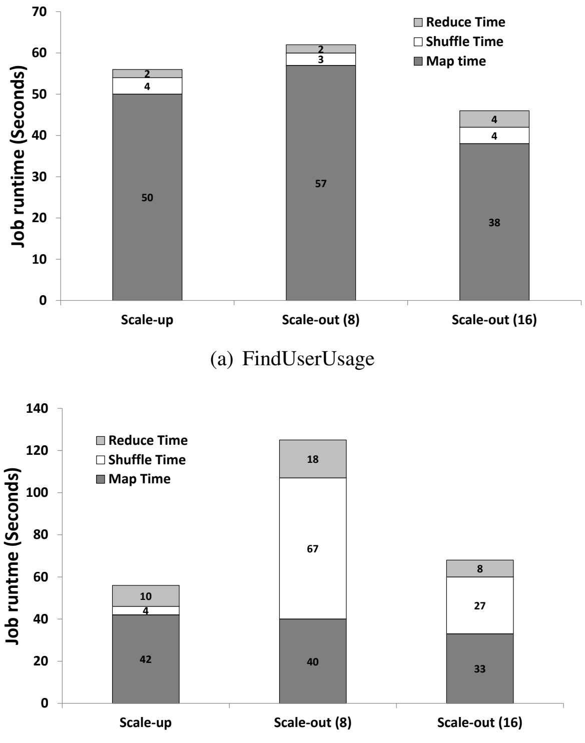
Broadly speaking, we expect map-intensive jobs to do relatively well for scale-out, and shuffle-intensive jobs to do well on scale-up. To validate this assumption, we choose two jobs: FindUserUsage, which is map-intensive, and the 10 GB TeraSort, which is shuffle-intensive. We then separated out the job execution times into map, shuffle, and reduce times. Since these can overlap we approximate them as follows: the map time is the time from the job start to the completion of the last map task; the shuffle time is the time interval from then to the completion of the last shuffle task; and the reduce time is the remaining time until the job is completed.

Figure 5(a) shows the results for FindUserUsage and Figure 5(b) the results for TeraSort. We see that as expected, runtime FindUserUsage is completely dominated by map time. Thus scale-up and 8-node scale-out have similar performance, since they both have 32 cores. The 16-node scale-out on the other hand benefits from twice as many cores. The scaling is not linear as Hadoop jobs also have task startup costs: "map time" is overlapped both with startup and with shuffle and reduce.

TeraSort is clearly shuffle-dominated on the 8-node cluster. For the 16-node cluster shuffle and map time appear approximately equal; however this is an artifact of our methodology where the overlap between the two is counted as "map time". In reality both shuffle and map time are reduced as the larger cluster has more cores as well as more bisection bandwidth, but the overall runtime remains shuffle-dominated. In the scale-up case we see that the runtime is clearly map-dominated, and the shuffle phase is extremely efficient.

4.4 Effect of optimizations

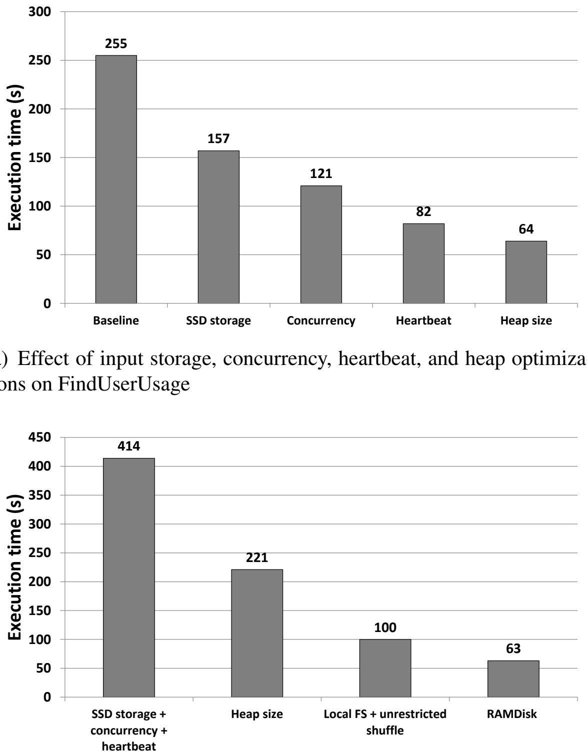
The results in the previous section used Hadoop setups that were fully optimized, i.e. all the relevant optimizations described in Section 3 were applied. In this section we look at the effect of each optimization individually. The aim is to understand the effect of the optimizations on scale-up performance, compared to a vanilla Hadoop running in pseudo-distributed mode.



(a) FindUserUsage

(b) 10 GB TeraSort

Figure 5: Runtime for different phases with FindUserUsage and 10 GB TeraSort



(a) Effect of input storage, concurrency, heartbeat, and heap optimizations on FindUserUsage

(b) Effect of memory and shuffle optimizations on 10 GB TeraSort

Figure 6: Effect of different optimizations on FindUserUsage and 10 GB TeraSort

We examine two jobs from our benchmark suite. FindUserUsage is map-intensive with little intermediate data and a small reduce phase. We use it to measure all optimizations except those that improve the shuffle phase, since the shuffle phase in FindUserUsage is very small. TeraSort is memory- and shuffle- intensive. Thus we use it to measure the effect of the memory and shuffle-phase optimizations. In all cases we use the scale-up server described previously.

Figure 6(a) shows the effect on FindUserUsage's execution time of successively moving from disk-based HDFS to an SSD-based local file system; of optimizing the number of mappers and reducers; of removing out-of-band heartbeats; and of optimizing the heap size. Each has a significant impact, with a total performance improvement of 4x.

Figure 6(b) shows the effect on execution time of 10 GB TeraSort starting from a baseline where the storage, concurrency, and heartbeat optimizations have already been applied. The heap size optimization has a significant effect, as input and intermediate data buffers are spilled less frequently to the file system. Moving to the file system based rather than http based shuffle and unthrottling the shuffle has an even bigger effect. Finally, using a RAMdisk for that intermediate data improves performance even further. For TeraSort, heap size optimization improves performance by 2x and shuffle-phase optimizations by almost 3.5x, for a total performance improvement of 7x.

5 Discussion

In Section 4 we evaluated 11 read-world jobs to show that scale-up is competitive on performance and performance/$, and superior on performance per watt and per rack unit. Here we consider implications for cloud computing, and discuss the limits of scale-up.

5.1 Scale up vs. scale-out in the cloud

Our analysis so far was based on a private cluster and scale-up machine. As many analytic jobs including Hadoop jobs are now moving to the cloud, it is worth asking, how will our results apply in the cloud scenario? The key difference from a private cluster is that in the cloud, a scalable storage back end, such as S3 or Azure Storage, is likely to be used for input data (instead of HDFS), and the compute nodes, at least today, are unlikely to use SSD storage. Intermediate data will be stored in memory or local disk.

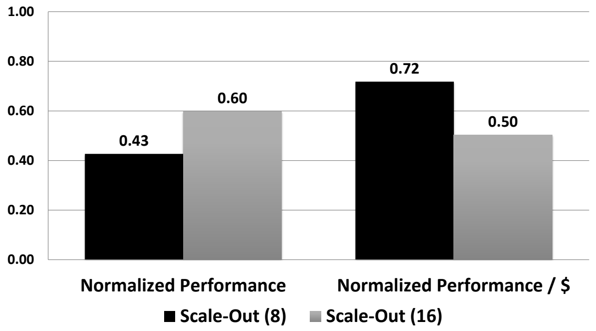


Figure 7: Relative throughput and throughput/$ of cloud and single-node scale-up with a scalable storage back-end. Results are normalized with respect to the scale-up throughput and throughput/$.

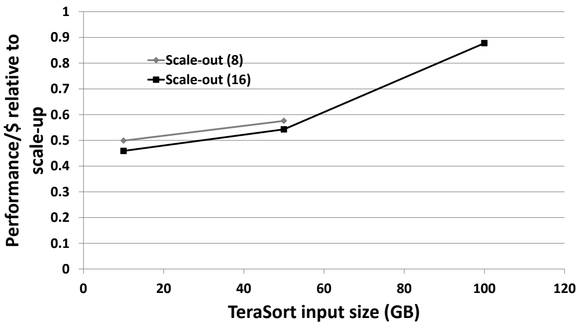


Figure 8: TeraSort performance/$ normalized to that of scale-up as a function of input size

We re-ran the FindUserUsage job using Azure compute nodes to scale out and the Azure Storage back end to hold input data. We used extra-large instances (8 cores, 16 GB of RAM) as being the closest to our scale-out workstation machines. At the time we ran these experiments, cloud VM instances corresponding to our scale-up machine were not available. Hence we used our scale-up machine but with data being read over the (10 Gbps) network rather than from SSD. (We assume here that a cloud scale-up VM is connected by a 10 Gbps network link to scalable storage.) Figure 7 shows that the scale-out performance in the cloud is worse than scale-up for both 8 and 16 nodes. We also observe that the scale-out cloud performance is 62% and 64% of the scale-out performance in our private cluster for 8 and 16 nodes respectively, due to contention for shared network bandwidth with other cloud users. On the other hand scale-up performance is almost unchanged (92% of the original).

Thus the relative performance of scale-up to scale-out is likely to improve for cloud scenarios; however we need to consider the relative pricing of the corresponding instances either. We note that Amazon has recently announced its High-Memory Cluster Instance for EC2[4] with 16 cores, 244 GiB of memory, and 10 Gbps networking for $3.50/hr. We estimate that a VM instance corresponding to our scale-up configuration (32 cores, 512 GB, and 10 Gbps) would cost at most twice as much as such this, and the same performance as our scale-up server reading data over a 10 Gbps link. Similarly, our scale-out machines can be mapped to M1 extra-large instances, which currently cost $0.52/hr.

At current EC2 prices, this would give us a scale-up pricing of $7/hr, compared to a scale-out pricing of $4.16/hr for 8 nodes and $8.32/hr for 16 nodes. This gives us the scale-up configuration 38% better performance/$ than the 8-node scale-out, and 50% better performance/$ than the 16-node scale-out (Figure 7). Compared with the corresponding measurements for FindUserUsage on our private cluster (Figure 4(b)) these are a relative improvement for scale-up. Thus, although we have not evaluated the cloud performance across all benchmarks, we believe this indicates that the price/performance tradeoffs in the cloud will be similar or better to those for a private cluster.

There are three factors that we believe will drive down the price of scale-up cloud instances compared to scale-out. The first two are the savings in power and rack space. The last one is the trend towards commoditization of high-bandwidth networking hardware (40 Gbps is already available and 100 Gbps on the horizon), which will make it more cost-effective to add bandwidth to a single scale-up node than to each of many scale-out nodes. We thus expect that cloud "scale-up" instances (which are already being offered for high-end applications such as HPC and large in-memory databases) will become an increasingly cost-effective way to run sub tera-scale Hadoop jobs with our optimizations to Hadoop.

5.2 Limitations to scale-up

Our results show that scale-up is more cost-effective for many real-world jobs which today use scale-out. However, clearly there is a job size beyond which scale-out becomes a better option. This "cross-over" point is job-specific. In order to get some understanding of this crossover point, we use TeraSort as an example since its job size is easily parametrized.

Figure 8 shows the results, again normalized to set the values for scale-up at 1. All three of our test configurations are able to sort up to 50 GB of data, with the scale-up configuration providing by far the best performance per dollar. However the 8-node cluster cannot sort 100 GB of data without running out of storage, and the scale-up machine cannot sort significantly more

[4] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html

than 100 GB without running out of memory. At 100 GB, scale-up still provides the best performance/$, but the 16-node cluster is close at 88% of scale-up.

These results tell us two things. First, even with "big memory", the scale-up configuration can become memory-bound for large jobs. However we expect this point to shift upwards as DRAM prices continue to fall and multiple terabytes of DRAM per machine become feasible. Second, for TeraSort, scale-out is competitive at around the 100 GB mark with current hardware.

More generally, while it is feasible to have a mix of large and small machines (and cloud providers already provide a mix of instance sizes), it is not desirable to maintain two versions of each application. By making all our changes transparently "under the hood" of Hadoop, we allow the decision of scale-up versus scale-out to be made transparently to the application.

6 Related Work

One of the motivations for this work was the observation that most analytic job sizes are well within the 512 GB that is feasible on a standard "scale-up" server today. We collected information about job sizes internally, where we found median job sizes to be less than 14 GB, but our conclusions are also supported by studies on a range of real-world Hadoop installations including Yahoo [16], Facebook [2, 12], and Cloudera [12].

Piccolo [25] is an in-memory distributed key-value store, aimed at applications that need low-latency fine-grained random access to state. Resilient Distributed Datasets (RDDs) [33] similarly offer a distributed memory like abstraction in the Spark system, but are aimed at task-parallel jobs, especially iterative machine learning jobs such as the Mahout jobs considered in this paper. Both of these are "in-memory scale-out" solutions: they remove the disk I/O bottleneck by keeping data in memory. However they still suffer from the network bottleneck of fetching remote data or shuffling data in a MapReduce computation. Our contribution is to show that scale-up rather than scale-out is a competitive option even for task-parallel jobs (both iterative and non-iterative) and can be done with transparent optimizations that maintain app-compatibility with Hadoop.

Phoenix [26, 32, 30], Metis [20], and Tiled-MapReduce [11] are in-memory multi-core (i.e. scale-up) optimized MapReduce libraries. They demonstrate that a carefully engineered MapReduce library can be competitive with a shared-memory multi-threaded implementation. In this paper we make a similar observation about multi-threaded vs. MapReduce in the Hadoop context. However our distinct contribution is that we provide good scale-up performance transparently for Hadoop jobs; and we evaluate the tradeoffs of scale-up vs. scale-out by looking at job sizes as well as performance, dollar cost, power, and server density.

GraphChi [18] also advocates processing big data in a single machine. They focus on graph computations, and argue that, by carefully scheduling disk operations and computations, even commodity machines can process very large graphs.

Michael et al. [21] studied the problem of scale-up vs scale-out for an interactive application (query processing in web search). They find scale-out to have a better performance per price ratio than scale-up. Observe, however, the differences in the context compared to our work (answering user queries versus data analytics respectively), and in the underlying hardware (IBM BladeCenter vs commodity PCs). They also find that running scale-out in a box gives better performance than using multi-threading.

In previous work [28] we showed that certain machine learning algorithms do not fit well within a MapReduce framework and hence both accuracy and performance were improved by running them as shared-memory programs on a single scale-up server. However this approach means that each algorithm be implemented once for a multi-threaded shared-memory model and again for MapReduce if scale-out is also desired. Hence in this paper we demonstrate how scale-up can be done transparently for Hadoop applications without sacrificing the potential for scale-out and without a custom shared-memory implementation. We believe that while custom multi-threaded implementations might be necessary for certain algorithms, they are expensive in terms of human effort and notoriously hard to implement correctly. Transparent scale-up using Hadoop is applicable for a much broader range of applications which are already written to use Hadoop MapReduce.

The tradeoff between low-power cores and a smaller number of server-grade cores was extensively studied by Reddi et al. [27] in the context of web search, and by Andersen et al. [3] in the context of a key-value storage system. Those works study the problem in different context and reach opposite conclusions. Andersen et al. [3] argues that scale-out is more cost and power effective than scale-up (for key-value storage systems). Reddi et al. [27] reach similar conclusions to us (although for a different context). Similarly, recent work [19] shows that for TPC-H queries, a cluster of low-power Atom processors is not cost-effective compared to a traditional Xeon processor. In general, the scale-up versus scale-out tradeoff is well-known in the parallel database community [15]. A key observation is that the correct choice of scale-up versus scale-out is workload-specific. However in the MapReduce world the conventional wisdom is that scale-out is the only interesting option. We challenge this conventional wisdom by showing that scale-up is in fact competitive on performance and cost, and superior on power and density, for a range of MapReduce applications.

7 Conclusions and Future Work

In this paper, we showed that, contrary to conventional wisdom, analytic jobs — in particular Hadoop MapReduce jobs — are often better served by a scale-up server than a scale-out cluster. We presented a series of transparent optimizations that allow Hadoop to deliver good scale-up performance, and evaluated our claims against a diverse set of Hadoop jobs.

Our results have implications for the way Hadoop and analytics clusters in general are provisioned, with scale-up servers being a better option for many jobs whether in a private cluster or in the cloud. Broadly the results suggest that a scale-up machine (or instance) running a scale-up optimized Hadoop should be considered as an option, rather than always using a cluster of small machines (or instances).

This observation raises two questions: when should a job run with scale-up rather than scale-out; and for jobs larger than even the largest scale-up machine, should we scale them out with a few large machines or with many small ones? The correct decision depends on job size, job characteristics, and pricing and we need an automated way to predict the best architecture and configuration for a given job. We are currently working on such a predictive mechanism based on input job sizes and static analysis of the application code. Moreover, the co-existence of scale-up and scale-out machines in a cluster also complicates the management of the cluster, e.g. the design of the scheduler [17, 29].

Our results also imply that software infrastructures such as Hadoop must in future be designed for good scale-up as well as scale-out performance. The optimizations presented in this paper provide a good initial starting point for improving scale-up performance.

Acknowledgments

We thank the reviewers, and our shepherd Anthony Joseph, who provided valuable feedback and advice.

References

[1] *Amazon Simple Storage Service (Amazon S3)*. `http://aws.amazon.com/s3/`. Accessed: 08/09/2011.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. “PACMan: Coordinated Memory Caching for Parallel Jobs”. *NSDI*. 2012.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. “FAWN: A Fast Array of Wimpy Nodes”. *Proceedings of SOSP*. 2009.

[4] *Apache Hadoop*. `http://hadoop.apache.org/`. Accessed: 08/09/2011.

[5] Apache Mahout. `http://mahout.apache.org/`. Accessed: 02/07/2013.

[6] Apache Pig Wiki. `http://wiki.apache.org/pig/PigPerformance`. Accessed: 02/07/2013.

[7] M. Bierman and L. Grimmer. *How I Use the Advanced Capabilities of Btrfs*. `http://www.oracle.com/technetwork/articles/servers-storage-admin/advanced-btrfs-1734952.html`. Accessed: 02/07/2013. 2012.

[8] J. Bonwick. *ZFS End-to-End Data Integrity*. `https://blogs.oracle.com/bonwick/entry/zfs_end_to_end_data`. Accessed: 02/07/2013. 2005.

[9] B. Calder et al. “Windows Azure Storage: a highly available cloud storage service with strong consistency”. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. ACM, 2011, pp. 143–157.

[10] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. “SCOPE: easy and efficient parallel processing of massive data sets”. *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1265–1276.

[11] R. Chen, H. Chen, and B. Zang. “Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling”. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT ’10. ACM, 2010.

[12] Y. Chen, S. Alspaugh, and R. H. Katz. “Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads”. *PVLDB* 5.12 (2012), pp. 1802–1813.

[13] Cloudera. *Tips and Guidelines: Improving Performance*. `http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/4.2.0/CDH4-Installation-Guide/cdh4ig_topic_11_6.html`. Accessed: 02/07/2013.

[14] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. *OSDI*. 2004.

[15] D. DeWitt and J. Gray. “Parallel Database Systems: The Future of High Performance Database Systems”. *Communications of the ACM* 35.6 (1992), pp. 85–98.

[16] K. Elmeleegy. “Piranha: Optimizing Short Jobs In Hadoop”. *VLDB*. 2013.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’11. USENIX, 2011.

[18] A. Kyrola, G. Blelloch, and C. Guestrin. “GraphChi: large-scale graph computation on just a PC”. *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI’12. USENIX Association, 2012, pp. 31–46.

[19] W. Lang, J. M. Patel, and S. Shankar. “Wimpy Node Clusters: What About Non-Wimpy Workloads?” *Workshop on Data Management on New Hardware (DaMon)*. 2010.

[20] Y. Mao, R. Morris, and F. Kaashoek. *Optimizing MapReduce for Multicore Architectures*. Tech. rep. MIT-CSAIL-TR-2010-020. MIT CSAIL, 2010.

[21] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. “Scale-up x Scale-out: A Case Study using Nutch/Lucene”. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IPDPS’07. IEEE, 2007, pp. 1–8.

[22] S. Owen, R. Anil, T. Dunning, and E. Friedman. *Mahout in Action*. Manning Publications Co., 2011.

[23] Panasas. *Accelerating and Simplifying Apache Hadoop with Panasas ActiveStor*. `http://www.panasas.com/sites/default/files/uploads/docs/hadoop_wp_lr_1096.pdf`. Accessed: 02/07/2013.

[24] 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”. *SIGMOD ’09: Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 165–178.

[25] R. Power and J. Li. “Piccolo: Building Fast, Distributed Programs with Partitioned Tables”. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[26] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. “Evaluating MapReduce for Multi-core and Multiprocessor Systems”. *HPCA*. 2007.

[27] V. J. Reddi, B. C. Lee, T. M. Chilimbi, and K. Vaid. “Web search using mobile cores: Quantifying and mitigating the price of efficiency”. *Proc. 37th International Symposium on Computer Architecture (37th ISCA’10)*. 2010, pp. 314–325.

[28] A. Rowstron, D. Narayanan, A. Donnelly, G. O’Shea, and A. Douglas. “Nobody ever got fired for using Hadoop”. *Workshop on Hot Topics in Cloud Data Processing (HotCDP)*. 2012.

[29] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys’13. ACM, 2013, pp. 351–364.

[30] J. Talbot, R. M. Yoo, and C. Kozyrakis. “Phoenix++: Modular MapReduce for Shared-Memory Systems”. *Second International Workshop on MapReduce and its Applications (MAPREDUCE)*. 2011.

[31] *Windows Azure Storage*. `http://www.microsoft.com/windowsazure/features/storage/`. Accessed: 08/09/2011.

[32] R. M. Yoo, A. Romano, and C. Kozyrakis. “Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System”. *IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[33] 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”. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.