UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Achieving Efficient I/O with High-Performance Data Center Technologies

Permalink

https://escholarship.org/uc/item/56v9t2nk

Author

Conley, Michael Aaron

Publication Date

2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Achieving Efficient I/O with High-Performance Data Center Technologies

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Michael Aaron Conley

Committee in charge:

Professor George Porter, Co-Chair
Professor Amin Vahdat, Co-Chair
Professor Alin Deutsch
Professor Yeshaiahu Fainman
Professor Stefan Savage

2015

Copyright

Michael Aaron Conley, 2015

All rights reserved.

The Dissertation of Michael Aaron Conley is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2015

EPIGRAPH

I like things to happen, and if they don't happen I like to make them happen.

Winston Churchill

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

Though the work presented in this dissertation covers my last six years of graduate study, it represents the culmination of decades of study, formation, and personal growth, all of which would not be possible without the help of a large number of incredibly supportive people.

The importance of a PhD advisor is something that cannot be overstated. In my particular case, I am blessed with not one, but two advisors. I am incredibly grateful to Amin Vahdat, without whom I would not be completing this dissertation. Thanks to Amin, I was able to start on a particularly fruitful line of research in my first year and have been able to produce some amazing results. Amin's recent position at Google has also been a blessing, resulting in internships and opportunities for collaboration with Google's cloud computing team.

I need to give particular thanks to my advisor, George Porter, for picking me up, not with reluctance, but with great enthusiasm. George has been immensely helpful in driving our research. His ability to see the bigger picture and give enough direction to keep me from going astray without micromanaging is a rare talent that only the best leaders possess. George has also contributed to nearly every area of our work. From writing code, to administering systems, to writing grants, to collaborating with industry, to editing papers, and even giving one of my talks when I was sick, he has been incredibly helpful, and I am eternally grateful.

In addition to my advisors, I must thank my other committee members, Stefan Savage, Alin Deutsch, and Yeshaiahu Fainman. From working with George and Amin, I understand how jam-packed a university professor's schedule can be. These professors have taken the time to give feedback and support, and for this I am very grateful. Stefan in particular I must thank for continually hyping my work at the Center for Networked Systems research reviews, which has helped generate interest from industry partners.

I would also like to thank Geoff Voelker for his help time and time again. Geoff has on numerous occasions reviewed our paper drafts in the final days leading up to a submission. He also donated more than $10,000 in Amazon EC2 credits, which contributed to our success in breaking world records in high-speed sorting on EC2.

Though I have seen the TritonSort and Themis projects from their beginning to the present, I would be remiss if I didn't acknowledge the very large number of people who contributed to these projects. I must first and foremost thank Alex Rasmussen, who led the project from the beginning. Alex taught me a great deal about what it means to be a systems researcher. From the importance of proper design, to the elegance of well-written code, to the practice of meticulous note taking, to the value of proper testing and debugging infrastructures, Alex has been incredibly helpful in my formation as a researcher. The work in Chapter 1 would certainly not have been possible without his leadership, guidance, and incredible work ethic.

I must thank my co-authors, Harsha V. Madhyastha, Radhika Niranjan Mysore, and Alex Pucher for their contributions to the TritonSort project and our initial victories in the sort benchmark contest. The TritonSort software system required an enormous amount of development, and it would not have been possible without all of these individuals working tirelessly.

Additionally, I would like to thank my co-authors, Rishi Kapoor and Vinh The Lam, for their contributions to the Themis project. One of the primary criticisms of TritonSort was that it "only did sort". With Themis, we have a general-purpose MapReduce implementation that maintains the speed and efficiency benefits of TritonSort, and I must thank my co-authors for making this possible.

I must also thank our corporate sponsors over the lifetime of this project, Cisco, NetApp, FusionIO, Amazon, and Google, for their contributions, including hardware donations, funding, credits, and thoughtful collaboration. The partnership between

industry and academia in computer science is a rare blessing, and without it the work in this dissertation would not have been possible.

Academic research, especially at the scale presented in this work, is only possible with generous funding. I am particularly grateful to NSF for funding large portions of this work. In addition to simply funding the researchers who worked on these projects, building the cluster in Chapter 1 required substantial infrastructural costs. Without these grants, we would not have been able to operate at the scales necessary to break the sort records.

I would also like to thank the sort benchmark committee members, Chris Nyberg, Mehul Shah, and Naga Govindaraju for their feedback and certification of our numerous sort records. Any successful line of work must have a good sales pitch or hook, and our sort records have been just that. These records enabled us to generate hype and interest in this line of work, and for that I am incredibly grateful.

My choice to pursue a PhD in distributed systems was the result of a series of fortunate incidents in my life. I must thank K. Mani Chandy for getting me interested in the subject during my undergraduate career at Caltech. His project-based distributed systems course series sparked a true and genuine interest in the topic that led me to pursue it as a professional career. For his instruction, inside and outside the classroom, I am very grateful.

My interest in computer science in general is also due to a series of fortunate incidents, beginning with my first algebra class at the College of Marin community college when I was in grade school. This particular course required a graphing calculator, and I was blessed to obtain a TI-83, which included a form of the BASIC programming language. As I continued to take math classes at the community college, I became interested in programming on this calculator, and eventually the TI-89. I am grateful for my community college professors, who put up with a child in their classrooms.

Without their enthusiastic support, I would not have continued to pursue math, and as a consequence, would not have found my love of computer science.

I must also thank my friends, now too numerous to list, for supporting me throughout this difficult time period. From commiserating over paper rejections, to celebrating sort records, to putting up with me during moments of despair, to giving me a reason to take a break from work, you all have been incredibly supportive.

Lastly, and most importantly, I must thank my parents. Without your support over the many years it has taken, I certainly would not have made it to this point. Thank you for putting up with all the frustrated phone calls. I really could not have done it without you.

Chapter 1 includes material as it appears in Proceedings of the 8th Annual USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2011. Rasmussen, Alexander; Porter, George; Conley, Michael; Madhyastha, Harsha V.; Mysore, Radhika Niranjan; Pucher, Alexander; Vahdat, Amin. The dissertation author was among the primary authors of this paper.

Chapter 1 also includes material as it appears in Proceedings of the 3rd Annual ACM Symposium on Cloud Computing (SOCC) 2012. Rasmussen, Alexander; Conley, Michael; Kapoor, Rishi; Lam, Vinh The; Porter, George; Vahdat, Amin. The dissertation author was among the primary authors of this paper.

Chapter 2 and Chapter 3 include material that is submitted for publication as "Achieving Cost-efficient, Data-intensive Computing in the Cloud." Conley, Michael; Vahdat, Amin; Porter, George. The dissertation author was the primary author of this paper.

VITA

2009	Bachelor of Science, California Institute of Technology
2012	Master of Science, University of California, San Diego
2015	Doctor of Philosophy, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Achieving Efficient I/O with High-Performance Data Center Technologies

by

Michael Aaron Conley

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

Professor George Porter, Co-Chair
Professor Amin Vahdat, Co-Chair

Recently there has been a significant effort to build systems designed for large-scale data processing, or "big data." These systems are capable of scaling to thousands of nodes, and offer large amounts of aggregate processing throughput. However, there is a severe lack of attention paid to the efficiency of these systems, with individual hardware components operating at speeds as low as 3% of their available bandwidths. In light of this observation, we aim to demonstrate that efficient data-intensive computation is not only possible, but also results in high levels of overall performance.

In this work, we describe two highly efficient data processing systems, TritonSort

and Themis, built using 2009-era cluster technology. We evaluate the performance of these systems and use them to set world records in high-speed sorting. Next, we consider newer, faster hardware technologies that are not yet widely deployed. We give a detailed description of the design decisions and optimizations necessary for efficient data-intensive computation on these technologies. Finally, we apply these optimizations to large-scale data-processing applications running in the public cloud, and once again set world records in high-speed sorting. We present the details of our experience with the Amazon Web Services (AWS) cloud and also explore Google Cloud Platform.

Introduction

The need for large-scale data processing, or "big data," is increasing at a rapid pace. This need permeates not only traditional data industries, such as search engines, social networks, and data-mining operations, but also fields outside of technology, such as healthcare and retail. Technological advances in hard sciences, such as the rise of high-quality genomics data in biology, are creating huge data sets that must be processed efficiently in order to advance the state of science.

The Era of Data Processing Frameworks

To match the demand for data processing solutions, many large-scale software systems have recently been developed. The parallel programming framework, MapReduce [19], published by Google in 2004 took the data-processing world by storm. MapReduce offers a simple programming model that removes much of the burden of distributed systems programming and allows developers to quickly launch large-scale data processing jobs, all while running on cheap, commodity hardware. Shortly thereafter, Apache Hadoop [104], an open-source implementation of MapReduce, was released and made the technology accessible to companies and researchers around the world.

Despite the technological advances of the MapReduce framework, it has many inefficiencies. Its widespread adoption led to many cases of misuse, further reducing efficiency. As a concrete example, Yahoo! noted that some of its Hadoop users would write MapReduce programs that spawned other unrelated frameworks, simply to take

advantage of the Hadoop infrastructure [86].

To handle these inefficiencies and interface mismatches, a variety of alternative solutions were proposed. Dryad [42], for example, offers a more general-purpose dataflow processing framework. In addition, many researchers and companies began to build layers on top of these frameworks in order to fix the interface mismatch. Hive [84] provides a SQL database built on top of Hadoop MapReduce. Pig [29] provides a different interface that is more procedural than SQL, but not as low-level as MapReduce, again built on top of Hadoop.

While these frameworks and layers were being developed, the database community expressed skepticism. Very well-known database researchers [23] went so far as to publicly state that the MapReduce paradigm was "a major step backwards." While MapReduce has continued to be successful for more than a decade, many of the observations noted by these database researchers are spot on. For example, lack of indexes, lack of schemas, lack of data mining tools, and inefficient I/O mechanisms have all hindered the performance of MapReduce deployments.

The Era of Framework Improvements

The MapReduce community soon discovered for itself many of the issues the database community had pointed out. Data-mining and machine-learning applications written in MapReduce suffered from poor performance due to a lack of support of efficient iterative computation. Solutions like HaLoop [13], again built on top of Hadoop MapReduce, offer incremental performance improvements while maintaining the now-familiar Hadoop framework.

The MapReduce computational model makes heavy use of intermediate data files for checkpointing in order to provide fault tolerance. DeWitt and Stonebraker [23] note that this reliance on data storage can dramatically affect performance. Shortly thereafter,

the MapReduce community began to observe this very effect. Solutions like Sailfish [68] offer improvements to the Hadoop MapReduce framework for more efficient intermediate file processing.

The Era of New Designs

After years of attempting to solve every known problem with some flavor of MapReduce, the community finally decided that some problems are best solved with custom solutions. New frameworks like Pregel [57] solve graph computation problems without relying on the MapReduce programming model. GraphLab [56] is another system built for machine-learning systems using a graph-parallel programming model.

Other systems like Spark [107] have targeted a specific type of data processing application, namely one in which the data sets are small enough to fit in memory. In these cases, large degrees of efficiency can be recovered from abandoning frameworks like Hadoop MapReduce that are built for much larger data sets.

Efficiency Concerns

These three eras, occurring over the course of a single decade, point to concerns over the efficiency of large-scale data processing systems. Early systems were built simply to scale to the levels necessary to tackle large problems like web search. Emphasis was placed on the performance of the cluster as a whole, rather than on any one individual processing element.

A survey of large-scale sorting systems [6] reveals shocking levels of inefficiency, with some large cluster deployments failing to reach throughputs of even 3 MB/s per disk, and others failing to drive even 5 MB/s of application throughput over a 1 Gb/s network. Clearly there is substantial room for improvement.

Many of these efficiency concerns center around I/O patterns and proper utilization of I/O devices. Many existing systems use more than the minimum number of I/O

operations necessarily to perform a job. Extra I/O operations may be issued for fault tolerance purposes, but as we will show in Chapter 1, it may be more desirable to build a system using fewer I/O operations. In the particular case of sorting, the minimum number of I/O operations when the data size exceeds memory is two [2], and we call a system that meets this minimum a "2-IO" system. The remainder of this dissertation will show the performance and efficiency levels possible when 2-IO is specified as a system requirement, rather than simply being a theoretical optimum.

Thesis

Despite these trends and inefficiencies, we postulate that it is indeed possible to build highly-efficient large-scale data processing systems. In particular:

> It is possible to build and deploy large-scale, high-performance, efficient, 2-IO MapReduce systems across multiple generations of hardware, ranging from disks and 10 Gb/s networks, to flash and 40 Gb/s networks, and even to public cloud virtual machines.

In order to prove this assertion, we first build two highly efficient data processing systems, TritonSort and Themis, using 2009-era cluster technology. We describe the performance of these systems and how we use them to set world records in high-speed sorting in Chapter 1. Next, we consider newer, faster hardware technologies that are not yet widely deployed. We give a detailed description of the design decisions and optimizations necessary for efficient data-intensive computation on these technologies in Chapter 2. Finally, we apply these optimizations to large-scale data-processing applications running in the public cloud, and once again set world records in high-speed sorting. Chapter 3 details our experience with the Amazon Web Services (AWS) cloud, and Chapter 4 explores Google Cloud Platform.

Chapter 1

Efficient I/O-Bound Applications

In this chapter we present a discussion of efficiency in I/O-bound applications with sorting as a canonical problem. We restrict focus to a particular hardware configuration based on 2009-era technology, including magnetic hard disk drives (HDDs) and servers with smaller memories. These conditions necessitate certain design decisions for achieving efficient data processing. We will revisit these design decisions in the presence of different hardware configurations in Chapter 2.

To solve the problem of high-speed, efficient sorting, we describe two systems, TritonSort and Themis, that we have built and analyze their performance on large-scale workloads. We use TritonSort in particular to set several world-records in large-scale sorting which are also detailed in this chapter.

Finally, we discuss several of the critical design decisions for achieving high performance. As we will see in later chapters, these design decisions change as more modern, high-performance hardware becomes available.

1.1 Sorting as a Canonical Problem

Though the focus of this dissertation is data-intensive applications, there is no single, commonly held definition for what constitutes such an application. Jim Gray et al. categorize data-intensive processing as consisting of (1) interactive transactions, (2)

Table 1.1. A subset of the sorting benchmarks that measure sorting performance and efficiency.

Category	**Measures:**
GraySort	time required to sort 100 TB of data.
CloudSort	dollar cost to sort 100 TB in the cloud.
MinuteSort	amount of data sorted in 60 seconds.
JouleSort	energy required to sort 10 GB, 100 GB, 1 TB, or 100 TB.

mini-batch updates of a small working set of the total data, or (3) bulk data movements that processes the entire data set [12][1]. We limit our analysis to this third type of data-intensive processing, corresponding to jobs that touch most or all of the input data, with working sets similar in size to the entire data set size. In general, these types of workloads are largely I/O-bound, and in particular are typically throughput-limited. Gray suggested using sorting as a stand-in for general-purpose data processing and formed an annual competition to focus effort on building I/O-efficient data processing systems [81], which we discuss below.

1.1.1 Sort Benchmark

The contest proposed by Gray is currently divided into a number of categories to stress different aspects of data processing. A subset of these are shown in Table 1.1. Each of these categories is further divided into two subtypes: "Indy" and "Daytona." Entrants in the Indy variant can assume that the data consists of fixed-sized records, also called *tuples* in this work, with 10-byte keys and 90-byte values, and further that the values of the keys are uniformly distributed across the possible keyspace. In contrast, the Daytona variant of each of the sort categories must be general purpose, supporting variable-sized records and records drawn from a skewed key distribution. CloudSort further stipulates that input and output data must be stored on persistent, replicated storage.

[1] Originally published as "Anon et al. (1985)"

A central requirement of these sort benchmarks is that input and output data must reside on stable storage before and after the sort operation. This requirement turns what would normally be considered a computationally easy problem into a challenge due to the significant resource demands. While asymptotically efficient $O(n \log n)$ algorithms exist and can sort moderate amounts of data in seconds on modern hardware, getting that data to and from slow storage devices in the same time frame is tricky. Furthermore, larger data sizes may require a cluster of servers to sort, involving added network complexity and cost. Finally, the largest data sets are so big that even a large cluster won't have enough memory to hold the data set. This particular case requires multiple passes to and from storage, and is the focus of the bulk of our work.

1.1.2 The Challenge of Efficiency

Recent advances in large-scale data-processing systems have solved large-scale workloads by improved software scalability. For example, systems like MapReduce [19], the Google File System [31], Hadoop [104], and Dryad [42] are able to scale linearly with the number of nodes in the cluster, making it trivial to add new processing capability and storage capacity to an existing cluster by simply adding more nodes. This linear scalability is achieved in part by exposing parallel programming models to the user and by performing computation on data locally whenever possible. Hadoop clusters with thousands of nodes are now deployed in practice [105].

Despite this linear scaling behavior, per-node performance has lagged behind per-server capacity by more than an order of magnitude. A survey of several deployed large-scale sorting systems [6] found that the impressive results obtained by operating at high scale mask a typically low individual per-node efficiency, requiring a larger-than-needed scale to meet application requirements. For example, among these systems as much as 94% of available disk I/O and 33% CPU capacity remained idle [6]. When

this work began, the largest known industrial Hadoop clusters achieve only 20 Mb/s of average bandwidth for large-scale data sorting on machines theoretically capable of supporting a factor of 100 more throughput.

In addition to low resource utilization, existing systems may suffer from designs that perform unnecessary amounts of I/O operations. Fundamentally, every sorting system must perform at least two I/O operations per record when the amount of data exceeds the amount of memory in the cluster [2]. We refer to a system that meets this lower-bound as having the "2-IO" property. Any data processing system that does not have this property is doing more I/O than it needs to. Existing large-scale systems can incur additional I/O operations in exchange for simpler and more fine-grained fault tolerance. These features are important, but come at the cost of reduced performance and efficiency.

Throughout this dissertation, we will consider efficiency as measured by a variety of metrics. In this chapter, we will primarily be interested in the performance of a disk-based cluster described in the next section. These disks are capable of achieving a certain, relatively-low level of performance for sequential I/O. Therefore, we tend to measure efficiency as MB/s/disk in this chapter. While the GraySort benchmark itself measures absolute performance (seconds to sort 100 TB, or GB/min), other benchmarks measure records sorted per Joule of energy, or dollars per sort, lending themselves more naturally to this definition of efficiency.

Focusing on the efficiency of individual disks or servers also allows for better matching between different hardware components. Suppose, for example, that a server hosts a single disk capable of 100 MB/s of sequential I/O. An efficient system that can take advantage of most of this disk bandwidth will also be able to make the most of a 1 Gb/s (125 MB/s) network interface that might be attached to the server. An inefficient system will waste not only the disk bandwidth but also the network bandwidth in this case. Efficient software systems, on the other hand, allow system designers to build high

Table 1.2. Resource options considered for constructing a cluster for a balanced sorting system. These values are estimates as of January, 2010.

Storage			
Type	Capacity	R/W throughput	Price
7.2k-RPM	500 GB	90-100 MB/s	$200
15k-RPM	150 GB	150 MB/s	$290
SSD	64 GB	250 MB/s	$450

Network	
Type	Cost/port
1 Gb/s Ethernet	$33
10 Gb/s Ethernet	$480

Server	
Type	Cost
8 disks, 8 CPU cores	$5,050
8 disks, 16 CPU cores	$5,450
16 disks, 16 CPU cores	$7,550

performance hardware platforms with well-balanced configurations.

1.2 A Hardware Platform for Sorting

To determine the right hardware configuration for our application, we make the following observations about the sort workload. First, the application needs to read every byte of the input data and the size of the input is equal to that of the output. Since the "working set" is so large, it does not make sense to separate the cluster into computation-heavy and storage-heavy regions. Instead, we provision each server in the cluster with an equal amount of processing power and disks.

Second, almost all of the data needs to be exchanged between machines since input data is randomly distributed throughout the cluster and adjacent tuples in the sorted sequence must reside on the same machine. To balance the system, we need to ensure that this all-to-all shuffling of data can happen in parallel without network bandwidth

becoming a bottleneck. Since we focus on using commodity components, we use an Ethernet network fabric. Commodity Ethernet is available in a set of discrete bandwidth levels—1 Gb/s, 10 Gb/s, and 40 Gb/s—with cost increasing proportional to throughput (see Table 1.2). Given our choice of 7.2k-RPM disks for storage, a 1 Gb/s network can accommodate at most one disk per server without the network throttling disk I/O. Therefore, we settle on a 10 Gb/s network; 40 Gb/s Ethernet has yet to mature and hence is still cost prohibitive. To balance a 10 Gb/s network with disk I/O, we use a server that can host 16 disks. Based on the options available commercially for such a server, we use a server that hosts 16 disks and 8 CPU cores. The choice of 8 cores was driven by the available processor packaging: two physical quad-core CPUs. The larger the number of separate threads, the more stages that can be isolated from each other. In our experience, the actual speed of each of these cores was a secondary consideration.

Third, sorting demands both significant capacity and I/O requirements from storage since tens to hundreds of TB of data is to be stored and all the data is to be read and written twice. To determine the best storage option given these requirements, we survey a range of hard disk options shown in Table 1.2. We find that 7.2k-RPM SATA disks provide the most cost-effective option in terms of balancing $ per GB and $ per read/write MB/s. To allow 16 disks to operate at full speed, we require storage controllers that are able to sustain at least 1600 MB/s of bandwidth. Because of the PCI bus' bandwidth limitations, our hardware design necessitated two 8x PCI drive controllers, each supporting 8 disks.

The final design choice in provisioning our cluster is the amount of memory each server should have. The primary purpose of memory in our system is to enable large amounts of data buffering so that we can read from and write to the disk in large chunks. The larger these chunks become, the more data can be read or written before seeking is required. We initially provisioned each of our machines with 12 GB of memory; however,

during development we realized that 24 GB was required to provide sufficiently large writes, and so the machines were upgraded.

Our testbed consists of 52 HP ProLiant DL380 G6 servers, Each server has two quad-core Intel Xeon E5520 processors, clocked at 2.27 GHz, and 24 GB of RAM. Each server also hosts 16 2.5-inch 500 GB, 7200 RPM SATA hard drives, which we describe in detail below. Each machine is equipped with a 1 Gb/s on-board network card as well as a Myricom 10 Gb/s network card. Both network cards run unmodified Ethernet. All the machines in our testbed are inter-connected via a Cisco Nexus 5020 switch, which provides 10 Gb/s connectivity between all pairs.

Initially, all servers hosted Seagate Momentus 7200.4 hard drives, which are consumer-grade hard drives. We found during the course of evaluation that these drives have poor reliability at scale. In particular, the probability of a single hard drive failing during the course of a 52-node 100 TB sort is very high. We later upgraded the hard drives to HP Seagate MM0500EANCR drives. These are also 500 GB, 7200 RPM drives but are enterprise-grade. The enterprise-grade drives still fail but at a much lower rate, making large-scale evaluation possible.

1.3 TritonSort: I/O-efficient Sorting

We now describe TritonSort, our solution for high-performance, large-scale sorting. We evaluate TritonSort primarily in the context of the problem of sorting 100 TB of data. We present our entries in the 2010 and 2011 sorting contests described in Section 1.1.1, which resulted in several world records in large-scale sorting performance.

TritonSort is a distributed, staged, pipeline-oriented dataflow processing system. Figures 1.1 and 1.6 show the stages of a TritonSort program. Stages in TritonSort are organized in a directed graph (with cycles permitted). Each stage in TritonSort implements part of the data processing pipeline and either sources, sinks, or transmutes

data flowing through it.

Each stage is implemented by two types of logical entities—several *workers* and a single *WorkerTracker*. Each worker runs in its own thread and maintains its own local queue of pending work. We refer to the discrete pieces of data over which workers operate as *work units* or simply as *work*. The WorkerTracker is responsible for accepting work for its stage and assigning that work to workers by enqueueing the work onto the worker's work queue. In each phase, all the workers for all stages in that phase run in parallel.

Upon starting up, a worker initializes any required internal state and then waits for work. When work arrives, the worker executes a stage-specific *run()* method that implements the specific function of the stage, handling work in one of three ways. First, it can accept an individual work unit, execute the *run()* method over it, and then wait for new work. Second, it can accept a batch of work (up to a configurable size) that has been enqueued by the WorkerTracker for its stage. Lastly, it can keep its *run()* method active, polling for new work explicitly. TritonSort stages implement each of these methods, as described below. In the process of running, a stage can produce work for a downstream stage and optionally specify the worker to which that work should be directed. If a worker does not specify a destination worker, work units are assigned to workers round-robin.

In the process of executing its *run()* method, a worker can get buffers from, and return buffers to, a shared pool of buffers. This buffer pool can be shared among the workers of a single stage, but is typically shared between workers in pairs of stages with the upstream stage getting buffers from the pool and the downstream stage putting them back. When getting a buffer from a pool, a stage can specify whether or not it wants to block waiting for a buffer to become available if the pool is empty.

1.3.1 Sort Architecture

We implement sort in two phases. First, we perform distribution sort to partition the input data across L logical partitions evenly distributed across all nodes in the cluster. Each logical partition is stored in its own *logical disk*. All logical disks are of identical maximum size $size_{LD}$ and consist of files on the local file system.

The value of $size_{LD}$ is chosen such that logical disks from each physical disk can be read, sorted and written in parallel in the second phase, ensuring maximum resource utilization. Therefore, if the size of the input data is $size_{input}$, there are $L = \frac{size_{input}}{size_{LD}}$ logical disks in the system. In phase two, the tuples in each logical disk get sorted locally and written to an output file. This implementation satisfies our design goal of reading and writing each tuple twice.

To determine which logical disk holds which tuples, we logically partition the 10-byte key space into L even divisions. We logically order the logical disks such that the k^{th} logical disk holds tuples in the k^{th} division. Sorting each logical disk produces a collection of output files, each of which contains sorted tuples in a given partition. Hence, the ordered collection of output files represents the sorted version of the data. In this discussion, we assume that tuples' keys are distributed uniformly over the key range which ensures that each logical disk is approximately the same size.

To ensure that we can utilize as much read/write bandwidth as possible on each disk, we partition the disks on each node into two groups of equal size. Recall that servers in our hardware testbed described in Section 1.2 have 16 disks, so each group contains 8 disks. One group of disks holds input and output files; we refer to these disks as the input disks in phase one and as the output disks in phase two. The other group holds intermediate files; we refer to these disks as the intermediate disks. In phase one, input files are read from the input disks and intermediate files are written to the intermediate

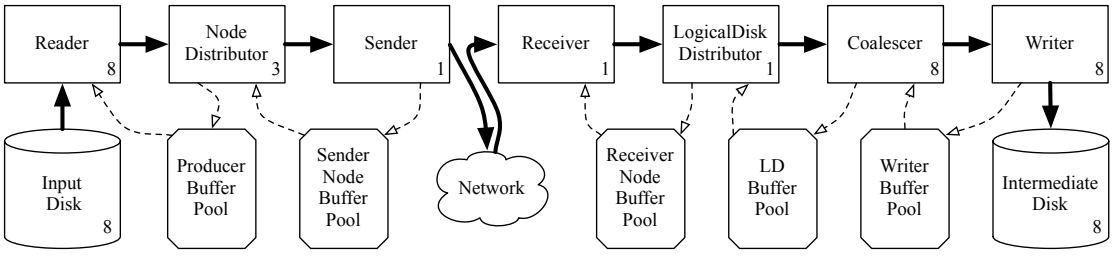


Figure 1.1. Block diagram of TritonSort's phase one architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

disks. In phase two, intermediate files are read from the intermediate disks and output files are written to the output disks. Thus, the same disk is never concurrently read from and written to, which prevents unnecessary seeking.

1.3.2 TritonSort Architecture: Phase One

Phase one of TritonSort, diagrammed in Figure 1.1, is responsible for reading input tuples off of the input disks, distributing those tuples over to the network to the nodes on which they belong, and storing them into the logical disks in which they belong.

Reader: Each Reader is assigned an input disk and is responsible for reading input data off of that disk. It does this by filling 80 MB ProducerBuffers with input data. We chose this size because it is large enough to obtain near sequential throughput from the disk.

NodeDistributor: A NodeDistributor (shown in Figure 1.2) receives a ProducerBuffer from a Reader and is responsible for partitioning the tuples in that buffer across the machines in the cluster. It maintains an internal data structure called a *NodeBuffer table*, which is an array of NodeBuffers, one for each of the nodes in the cluster. A NodeBuffer contains tuples belonging to the same destination machine. Its size was chosen to be the size of the ProducerBuffer divided by the number of nodes, and is approximately 1.6 MB

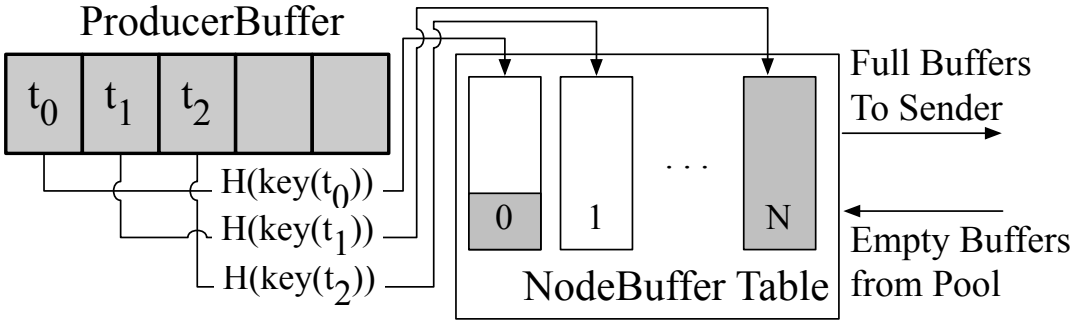


Figure 1.2. The NodeDistributor stage, responsible for partitioning tuples by destination node.

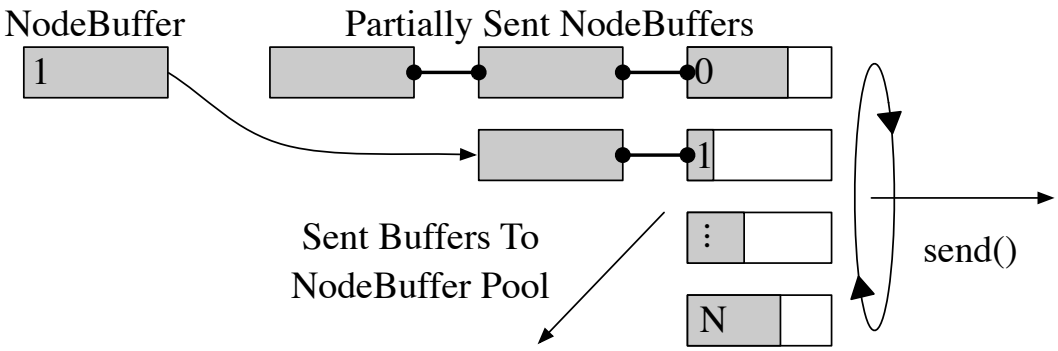


Figure 1.3. The Sender stage, responsible for sending data to other nodes.

in size for the scales we consider in this discussion.

The NodeDistributor scans the ProducerBuffer tuple by tuple. For each tuple, it computes a hash function $H(k)$ over the tuple's key k that maps the tuple to a unique host in the range $[0, N-1]$. It uses the NodeBuffer table to select a NodeBuffer corresponding to host $H(k)$ and appends the tuple to the end of that buffer. If that append operation causes the buffer to become full, the NodeDistributor removes the NodeBuffer from the NodeBuffer table and sends it downstream to the Sender stage. It then gets a new NodeBuffer from the NodeBuffer pool and inserts that buffer into the newly empty slot in the NodeBuffer table. Once the NodeDistributor is finished processing a ProducerBuffer, it returns that buffer back to the ProducerBuffer pool.

Sender: The Sender stage (shown in Figure 1.3) is responsible for taking NodeBuffers from the upstream NodeDistributor stage and transmitting them over the network to each of the other nodes in the cluster. Each Sender maintains a separate TCP socket per peer node in the cluster. The Sender stage can be implemented in a multi-threaded or a single-threaded manner. In the multi-threaded case, *N* Sender workers are instantiated in their own threads, one for each destination node. Each Sender worker simply issues a blocking *send()* call on each NodeBuffer it receives from the upstream NodeDistributor stage, sending tuples in the buffer to the appropriate destination node over the socket open to that node. When all the tuples in a buffer have been sent, the NodeBuffer is returned to its pool, and the next one is processed. For performance reasons, we choose a single-threaded Sender implementation. Here, the Sender interleaves the sending of data across all the destination nodes in small non-blocking chunks, so as to avoid the overhead of having to activate and deactivate individual threads for each send operation to each peer.

Unlike most other stages, which process a single unit of work during each invocation of their *run()* method, the Sender continuously processes NodeBuffers as it runs, receiving new work as it becomes available from the NodeDistributor stage. This is because the Sender must remain active to alternate between two tasks: accepting incoming NodeBuffers from upstage NodeDistributors, and sending data from accepted NodeBuffers downstream. To facilitate accepting incoming NodeBuffers, each Sender maintains a set of NodeBuffer lists, one for each destination host. Initially these lists are empty. The Sender appends each NodeBuffer it receives onto the list of NodeBuffers corresponding to the incoming NodeBuffer's destination node.

To send data across the network, the Sender loops through the elements in the set of NodeBuffer lists. If the list is non-empty, the Sender accesses the NodeBuffer at the head of the list, and sends a fixed-sized amount of data to the appropriate destination host

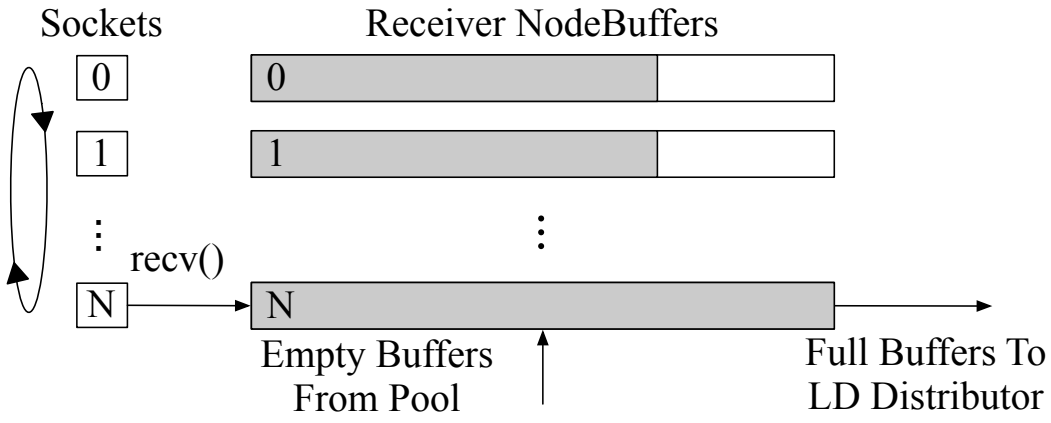


Figure 1.4. The Receiver stage, responsible for receiving data from other nodes' Sender stages.

using a non-blocking *send()* call. If the call succeeds and some amount of data was sent, then the NodeBuffer at the head of the list is updated to note the amount of its contents that have been successfully sent so far. If the *send()* call fails, because the TCP send buffer for that socket is full, that buffer is simply skipped and the Sender moves on to the next destination host. When all of the data from a particular NodeBuffer is successfully sent, the Sender returns that buffer back to its pool.

Receiver: The Receiver stage, shown in Figure 1.4, is responsible for receiving data from other nodes in the cluster, appending that data onto a set of NodeBuffers, and passing those NodeBuffers downstream to the LogicalDiskDistributor stage. In TritonSort, the Receiver stage is instantiated with a single worker. On starting up, the Receiver opens a server socket and accepts incoming connections from Sender workers on remote nodes. Its *run()* method begins by getting a set of NodeBuffers from a pool of such buffers, one for each source node. The Receiver then loops through each of the open sockets, reading up to 16KB of data at a time into the NodeBuffer for that source node using a non-blocking *recv()* call. If data is returned by that call, it is appended to the end of the NodeBuffer. If the append would exceed the size of the NodeBuffer, that buffer is sent

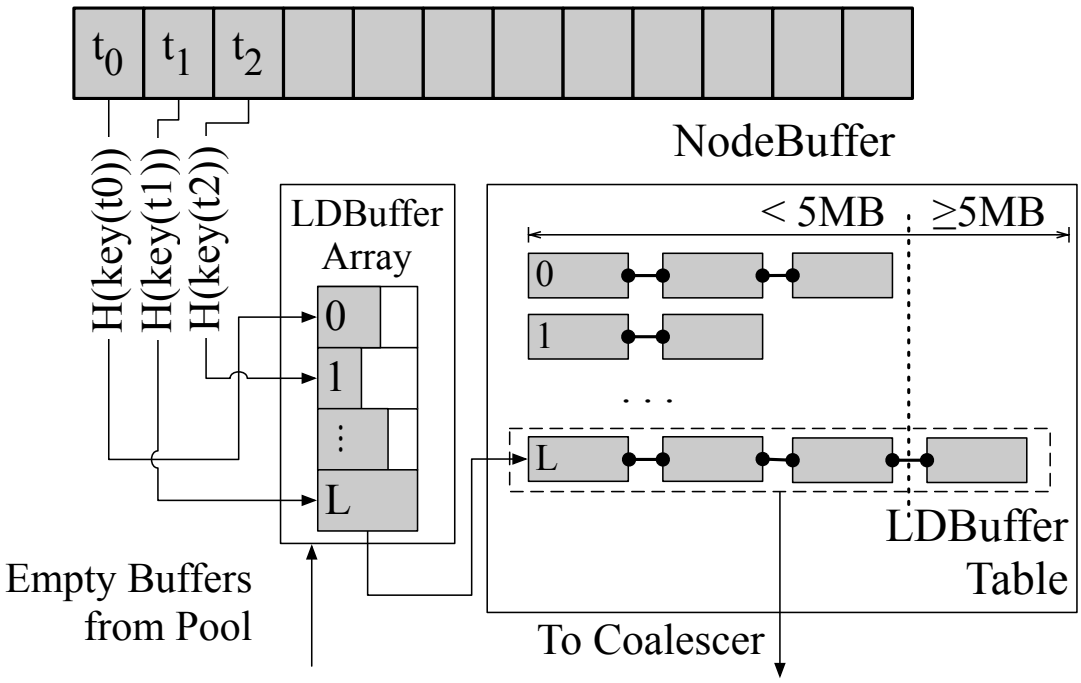


Figure 1.5. The LogicalDiskDistributor stage, responsible for distributing tuples across logical disks and buffering sufficient data to allow for large writes.

downstream to the LogicalDiskDistributor stage, and a new NodeBuffer is retrieved from the pool to replace the NodeBuffer that was sent.

LogicalDiskDistributor: The LogicalDiskDistributor stage, shown in Figure 1.5, receives NodeBuffers from the Receiver that contain tuples destined for logical disks on its node. LogicalDiskDistributors are responsible for distributing tuples to appropriate logical disks and sending groups of tuples destined for the same logical disk to the downstream Writer stage.

The LogicalDiskDistributor's design is driven by the need to buffer enough data to issue large writes and thereby minimize disk seeks and achieve high bandwidth. Internal to the LogicalDiskDistributor are two data structures: an array of LDBuffers, one per logical disk, and an LDBufferTable. An LDBuffer is a buffer of tuples destined to the

Algorithm 1. The LogicalDiskDistributor stage

```
 1: NodeBuffer ← getNewWork()
 2: {Drain NodeBuffer into the LDBufferArray}
 3: for all tuples t in NodeBuffer do
 4:    dst = H(key(t))
 5:    LDBufferArray[dst].append(t)
 6:    if LDBufferArray[dst].isFull() then
 7:       LDTable.insert(LDBufferArray[dst])
 8:       LDBufferArray[dst] = getEmptyLDBuffer()
 9:    end if
10: end for
11: {Send full LDBufferLists to the Coalescer}
12: for all physical disks d do
13:    while LDTable.sizeOfLongestList(d) ≥ 5 MB do
14:       ld ← LDTable.getLongestList(d)
15:       Coalescer.pushNewWork(ld)
16:    end while
17: end for
```

same logical disk. Each LDBuffer is 12,800 bytes long, which is the least common multiple of the tuple size (100 bytes) and the direct I/O write size dictated by the sectors of our disks (512 bytes). The LDBufferTable is an array of LDBuffer lists, one list per logical disk. Additionally, LogicalDiskDistributor maintains a pool of LDBuffers, containing 1.25 million LDBuffers, accounting for 20 of each machine's 24 GB of memory.

The operation of a LogicalDiskDistributor worker is described in Algorithm 1. In Line 1, a full NodeBuffer is pushed to the LogicalDiskDistributor by the Receiver. Lines 3-10 are responsible for draining that NodeBuffer tuple by tuple into an array of LDBuffers, indexed by the logical disk to which the tuple belongs. Lines 12-17 examine the LDBufferTable, looking for logical disk lists that have accumulated enough data to write out to disk. We buffer at least 5 MB of data for each logical disk before flushing that data to disk to prevent many small write requests from being issued if the pipeline temporarily stalls. When the minimum threshold of 5 MB is met for any particular

physical disk, the longest LDBuffer list for that disk is passed to the Coalescer stage on Line 15.

The original design of the LogicalDiskDistributor only used the LDBuffer array described above and used much larger LDBuffers (~10 MB each) rather than many small LDBuffers. The Coalescer stage (described below) did not exist; instead, the LogicalDiskDistributor transferred the larger LDBuffers directly to the Writer stage.

This design was abandoned due to its inefficient use of memory. Temporary imbalances in input distribution could cause LDBuffers for different logical disks to fill at different rates. This, in turn, could cause an LDBuffer to become full when many other LDBuffers in the array are only partially full. If an LDBuffer is not available to replace the full buffer, the system must block (either immediately or when an input tuple is destined for that buffer's logical disk) until an LDBuffer becomes available. One obvious solution to this problem is to allow partially full LDBuffers to be sent to the Writers at the cost of lower Writer throughput. This scheme introduced the further problem that the unused portions of the LDBuffers waiting to be written could not be used by the LogicalDiskDistributor. In an effort to reduce the amount of memory wasted in this way, we migrated to the current architecture, which allows small LDBuffers to be dynamically reallocated to different logical disks as the need arises. This comes at the cost of additional computational overhead and memory copies, but we deem this cost to be acceptable due to the small cost of memory copies relative to disk seeks.

Coalescer: The operation of the Coalescer stage is simple. A Coalescer will copy tuples from each LDBuffer in its input LDBuffer list into a WriterBuffer and pass that WriterBuffer to the Writer stage. It then returns the LDBuffers in the list to the LDBuffer pool.

Originally, the LogicalDiskDistributor stage did the work of the Coalescer stage.

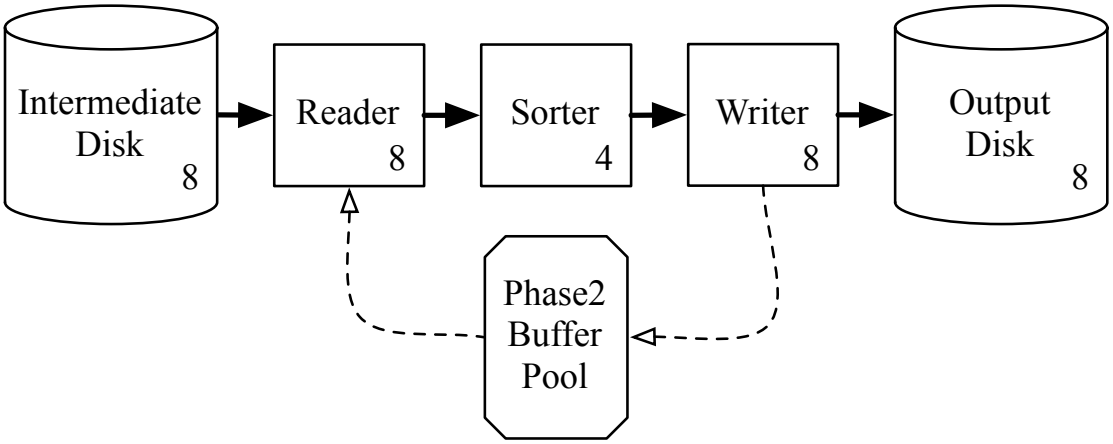


Figure 1.6. Block diagram of TritonSort's phase two architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

While optimizing the system, however, we realized that the non-trivial amount of time spent merging LDBuffers into a single WriterBuffer could be better spent processing additional NodeBuffers.

Writer: The operation of the Writer stage is also quite simple. When a Coalescer pushes a WriterBuffer to it, the Writer worker will determine the logical disk corresponding to that WriterBuffer and write out the data using a blocking *write()* system call. When the write completes, the WriterBuffer is returned to the pool.

1.3.3 TritonSort Architecture: Phase Two

Once phase one completes, all of the tuples from the input dataset are stored in appropriate logical disks across the cluster's intermediate disks. In phase two, each of these unsorted logical disks is read into memory, sorted, and written out to an output disk. The pipeline is straightforward: Reader and Writer workers issue sequential I/O requests to the appropriate disk, and Sorter workers operate entirely in memory.

Reader: The phase two Reader stage is identical to the phase one Reader stage, except that it reads into a PhaseTwoBuffer, which is the size of a logical disk.

Sorter: The Sorter stage performs an in-memory sort on a PhaseTwoBuffer. A variety of sort algorithms can be used to implement this stage, however we selected the use of radix sort due to its speed. Radix sort requires additional memory overhead compared to an in-place sort like QuickSort, and so the sizes of our logical disks have to be sized appropriately so that enough Reader–Sorter–Writer pipelines can operate in parallel. Our version of radix sort first scans the buffer, constructing a set of structures containing a pointer to each tuple's key and a pointer to the tuple itself. These structures are then sorted by key. Once the structures have been sorted, they are used to rearrange the tuples in the buffer in-place. This reduces the memory overhead for each Sorter substantially at the cost of additional memory copies.

Writer: The phase two Writer writes a PhaseTwoBuffer sequentially to a file on an output disk. As in phase one, each Writer is responsible for writes to a single output disk.

Because the phase two pipeline operates at the granularity of a logical disk, we can operate several of these pipelines in parallel, limited by either the number of cores in each system (we can't have more pipelines than cores without sacrificing performance because the Sorter is CPU-bound), the amount of memory in the system (each pipeline requires at least three times the size of a logical disk to be able to read, sort, and write in parallel), or the throughput of the disks. In our case, the limiting factor is the output disk bandwidth. To host one phase two pipeline per input disk requires storing 24 logical disks in memory at a time. To accomplish this, we set $size_{LD}$ to 850 MB, using most of the 24 GB of RAM available on each node and allowing for additional memory required by the operating system. To sort 850 MB logical disks fast enough to not block the Reader

Table 1.3. Median stage runtimes for a 52-node, 100TB sort, excluding the amount of time spent waiting for buffers.

Worker Type	Input (MB)	Runtime (ms)	Workers	Worker Throughput (MB/s)	Total Throughput (MB/s)
Reader	81.92	958.48	8	85	683
NodeDistributor	81.92	263.54	3	310	932
LogicalDiskDistributor	1.65	2.42	1	683	683
Coalescer	10.60	4.56	8	2,324	18,593
Writer	10.60	141.07	8	75	601
Phase two Reader	762.95	8,238	8	92	740
Phase two Sorter	762.95	2,802	4	272	1089
Phase two Writer	762.95	8,512	8	89	717

Table 1.4. Effect of increasing speed of intermediate disks on a two node, 500GB sort

Intermediate Disk Speed (RPM)	Logical Disks Per Physical Disk	Phase 1 Speed (MB/s)	Phase 1 Bottleneck Stage	Average Write Size (MB)
7200	315	69.81	Writer	12.6
7200	158	77.89	Writer	14.0
15000	158	79.73	LogicalDiskDistributor	5.02

and Writer stages, we find that four Sorters suffice.

1.3.4 Stage and Buffer Sizing

One of the major requirements for operating TritonSort at near disk speed is ensuring cross-stage balance. Each stage has an intrinsic execution time, either based on the speed of the device to which it interfaces (e.g., disks or network links), or based on the amount of CPU time it requires to process a work unit. Table 1.3 shows the speed and performance of each stage in the pipeline. In our implementation, we are limited by the speed of the Writer stage in both phases one and two.

1.4 Evaluation of TritonSort

We now evaluate TritonSort's performance and scalability under various hardware configurations. Our testbed is the same one described in Section 1.2. Each hard drive is configured with a single XFS partition. Each XFS partition is configured with a single allocation group to prevent file fragmentation across allocation groups, and is mounted with the `noatime`, `attr2`, `nobarrier`, and `noquota` flags set. The servers run Linux 2.6.35.1, and our implementation of TritonSort is written in C++.

1.4.1 Examining Changes in Balance

We first examine the effect of changing the cluster's configuration to support more memory or faster disks. Due to budgetary constraints, we could not evaluate these hardware configurations at scale.

In the first experiment, we replaced the 500 GB, 7,200 RPM disks that are used as the intermediate disks in phase one and the input disks in phase two with 146 GB, 15,000 RPM disks. The reduced capacity of the drives necessitated running an experiment with a smaller input data set. To allow space for the logical disks to be pre-allocated on the intermediate disks without overrunning the disks' capacity, we decreased the number of logical disks per physical disk by a factor of two. This doubles the amount of data in each logical disk, but the experiment's input data set is small enough that the amount of data per logical disk does not overflow the logical disk's maximum size.

Phase one throughput in these experiments is slightly lower than in subsequent experiments because the 30-35 seconds it takes to write the last few bytes of each logical disk at the end of the phase is roughly 10% of the total runtime due to the relatively small dataset size.

The results of this experiment are shown in Table 1.4. We first examine the effect

Table 1.5. Effect of increasing the amount of memory per node on a two node, 2 TB sort.

RAM Per Node (GB)	Phase 1 Throughput (MB/s)	Average Write Size (MB)
24	73.53	12.43
48	76.45	19.21

of decreasing the number of logical disks without increasing disk speed. Decreasing the number of logical disks increases the average length of LDBuffer chains formed by the LogicalDiskDistributor; note that most of the time, full WriterBuffers (14 MB) are written to the disks. In addition, halving the number of logical disks decreases the number of external cylinders that the logical disks occupy, decreasing maximal seek latency. These two factors combine together to net a significant (11%) increase in phase one throughput.

The performance gained by writing to 15,000 RPM disks in phase one is much less pronounced. The main reason for this is that the increase in write speed causes the Writers to become fast enough that the LogicalDiskDistributor exposes itself as the bottleneck stage. The lack of back-pressure from the writer also causes the LogicalDiskDistributor to emit smaller chains of buffers, creating smaller writes.

In the next experiment, we doubled the RAM in two of the machines in our cluster and adjusted TritonSort's memory allocation by doubling the size of each WriterBuffer (from 14 MB to 28 MB) and using the remaining memory (22 GB) to create additional LDBuffers. As shown in Table 1.5, increasing the amount of memory allows for the creation of longer chains of LDBuffers in the LogicalDiskDistributor, which in turn causes write sizes to increase. The increase in write size is not linear in the amount of RAM added; this is likely because we are approaching the point past which larger writes will not dramatically improve write throughput.

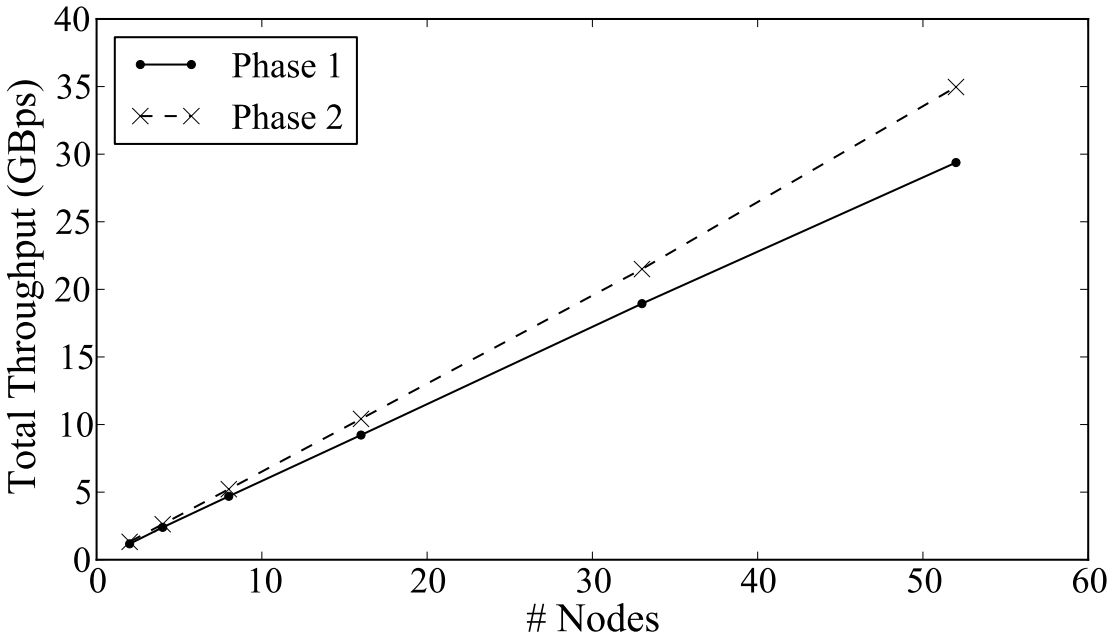


Figure 1.7. Throughput when sorting 1 TB per node as the number of nodes increases

1.4.2 TritonSort Scalability

Figure 1.7 shows TritonSort's total throughput when sorting 1 TB per node as the number of nodes increases from 2 to 48. Phase two exhibits practically linear scaling, which is expected since each node performs phase two in isolation. Phase one's scalability is also nearly linear; the slight degradation in its performance at large scales is likely due to network variance that becomes more pronounced as the number of nodes increases.

1.5 Sort Benchmark Results

We now present the results, shown in Table 1.6, of our entry into the 2010 and 2010 Sort Benchmark contests. The contest, described in detail in Setion 1.1.1, consists of a number of different benchmarks measuring a variety of quantities, ranging from raw performance to energy efficiency.

To set these records in sorting performance and efficiency, we ran TritonSort

Table 1.6. Submitted benchmark results for 2010 and 2011.

Year	Benchmark	Variant	Data Size	Nodes	Quantity of Interest
2010	GraySort	Indy	100 TB	47	10318 seconds
	MinuteSort	Indy	1014 GB	52	57.9 seconds median
2011	GraySort	Indy	100 TB	52	6395 seconds
	GraySort	Daytona	100 TB	52	8274 seconds
	MinuteSort	Indy	1353 GB	66	59.2 seconds median
	JouleSort	Indy	100 TB	52	9704 records/Joule
	JouleSort	Daytona	100 TB	52	7595 records/Joule

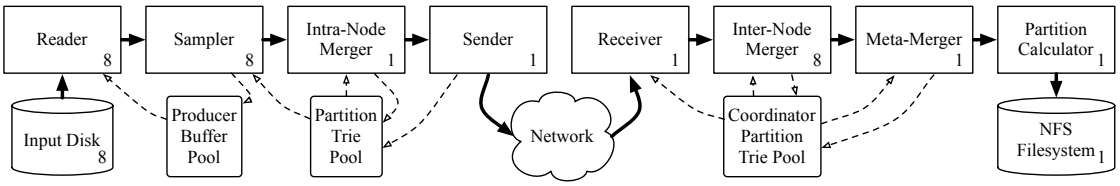


Figure 1.8. Architecture pipeline for phase Zero

on the testbed described in Section 1.2. In the case of the 2011 Indy MinuteSort, we upgraded our switch to the Cisco Nexus 5596UP, which has more ports. We were therefore able to add more servers, resulting in the 66 node sort benchmark result.

We now discuss two of the more interesting benchmark submissions, our 2011 Daytona GraySort record and our 2010 and 2011 Indy MinuteSort records. The discussion of the energy-efficient JouleSort is outside the scope of this work. For a detailed discussion of JouleSort, please consult our written reports [70, 72].

1.5.1 Daytona GraySort

For the 'Daytona' variant of GraySort, the input data does not necessarily follow a uniform key distribution. To prevent our system from becoming unbalanced, we need to construct a hash function that will ensure that tuples read from the input are spread across the nodes in our system evenly. Thus before we can begin sorting, we have to sample the input data to construct an empirical hash function based on that input data. We call the mechanism that performs this sampling *phase zero* because it runs before

phase one. The stages that make up phase zero are interconnected as shown in Figure 1.8.

We chose to use the well known approach of reading a subset of the input data (sampled evenly throughout the entire input) to determine this distribution. This process works as follows. The input data is spread across N nodes. At the start of phase zero, each node opens its input file and reads some number of 80 MB buffers' worth of data from each file. The number of buffers used depends on the amount of data sampled from each disk; for our experiments, we chose to sample at least 1 GB of data from each node, which means that we read two buffers from each disk. The keys of the tuples in these buffers are then summarized by recording their values in a fixed-depth, fixed-fanout full partition trie.

We choose a partition trie with a depth of three and a fanout of 256. Every path from the root to a non-root node in the partition trie represents a possible key value; for example, the key whose first three bytes are 234, 119, and 6 would correspond to the node that is the 6^{th} child of the 119^{th} child of the 234^{th} child of the root. Every node in the trie maintains a *sample count* indicating how many tuples were recorded with keys equal to that node's key. Keys that are less than three bytes long will be recorded as samples in interior nodes of the trie; keys that are three bytes long or longer will be recorded at the trie's leaves.

Each reader records its sample values in a separate partition trie. Partition tries from multiple readers are merged together into a single trie. Tries are merged together simply by adding their sample counts at each node.

Once a node's partition tries have been merged into a single trie, that trie is sent to a single designated node, called the coordinator. The coordinator merges the partition tries from each node together into a single partition trie, and then uses this combined partition trie (which contains a summary of sampling information across all nodes) to figure out how to split the key space across partitions such that each node receives a

roughly equal division of the input data set.

To do this, the coordinator determines a target partition size, which it calculates as the total number of samples divided by the total number of partitions. The total number of partitions is equal to the number of logical disks per physical disk multiplied by the number of physical disks in the cluster.

Once it has computed the target partition size, the coordinator does a pre-order traversal of the trie. As it does this traversal, it keeps track of the current partition and the number of samples allocated to that partition so far. At each node, it sets that node's partition to the current partition and adds that node's sample count to the total number of samples seen so far. If the number of samples seen so far meets or exceeds the target partition size, the current partition is incremented and the number of samples seen is reset.

We found in practice that this greedy allocation of nodes to partitions could potentially starve later partitions of samples if many previous partitions' sample counts slightly exceeded the target sample count, hence taking more than their fair share of samples. To mitigate this problem, we slightly adjusted the above algorithm to re-adjust the target partition size based on how much "slack" the previous partition had. For example, if the target partition size was 10 and the number of samples greedily allocated to it was 12, the target size of the next partition is set to 8. While this introduces minor imbalances in sample allocation, we found that this produces extremely uniform partitions in practice without starving partitions of tuples.

Once the coordinator has computed the partition assignments for each node in the trie, it writes the trie as a file on an NFS filesystem shared by all the nodes. At the start of phase one (described below), each node will read this trie from NFS and use it to drive its hash function. The trie is used by the hash function by simply traversing the trie based on the first three bytes of the key and returning the partition number at the appropriate node.

In practice, phase zero takes between 15 and 30 seconds to execute at scale.

1.5.2 MinuteSort

The MinuteSort benchmark measures the total amount of data that can be sorted in less than 60 seconds. This strict time limit includes start-up and shutdown time, effectively measuring what work can be done given a hard deadline. Because the time frame is so short, only a relatively small amount of data can be sorted. In fact, the data size is often so small that it can fit in the aggregate memory of a moderately sized cluster. Therefore, a more efficient design than the one given in Section 1.3 can be built.

For ease of implementation, we aim to keep most of TritonSort's design unchanged. A key insight is that our design is already very efficient. It simply performs an extra round of storage I/O that is not necessary when the data can fit in memory. Therefore, we modify the first phase to hold intermediate data in memory rather than writing to disk. In this case, the LogicalDiskDistributor creates LDBuffers that are the size of an entire logical disk, and no coalescing occurs. In the second phase, rather than reading logical disks from stable storage, we simply take the in-memory logical disks produced in the first phase and sort them before writing to disk.

This modification to TritonSort causes the first phase to perform only reads, and the second phase to perform only writes. Therefore, we use all 16 disks for reading in the first phase and writing in the second phase, yielding extra I/O performance.

1.6 Themis: I/O-efficient MapReduce

We now describe Themis, a high-performance MapReduce framework. MapReduce [19] is a parallel programming framework in which a program is decomposed into two functions, `map` and `reduce`. The `map` function takes data in the form of key/value pairs and transforms them into different key/value pairs. The `reduce` function takes

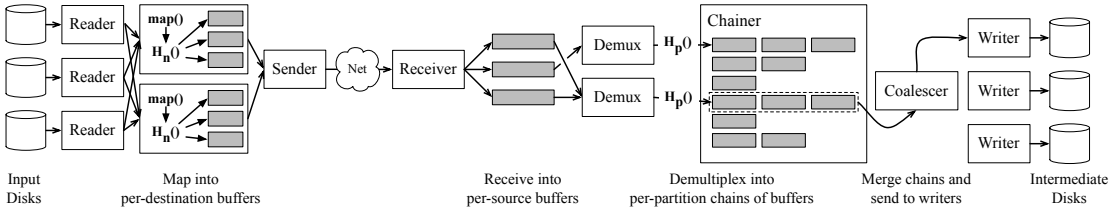


Figure 1.9. Stages of Phase One (Map/Shuffle) in Themis

groups of pairs with the same key and applies some aggregation-style computation to them, typically producing a smaller final result. These functions are written in a serial, single-threaded manner, but are intended to run on large distributed clusters. It is the job of the framework to shuttle data to and from the appropriate computational sites in the cluster. This data movement, also called the *shuffle phase*, is typically complicated and is a significant barrier to high performance MapReduce-style computation. Themis is a solution to this problem and provides an efficient, high-speed shuffle.

Themis builds upon the efficient sorting work in the TritonSort architecture described in Section 1.3. As such, Themis reuses several core runtime components that were used to build the TritonSort [71] sorting system. Like TritonSort, Themis is written as a sequence of *phases*, each of which consists of a directed dataflow graph of *stages* connected by FIFO queues. Each stage consists of a number of *workers*, each running as a separate thread.

1.6.1 MapReduce Overview

Unlike existing MapReduce systems, which execute `map` and `reduce` tasks concurrently in waves, Themis implements the MapReduce programming model in three phases of operation, summarized in Table 1.7. Phase zero, described in Section 1.6.4, is responsible for sampling input data to determine the distribution of record sizes as well as the distribution of keys. These distributions are used by subsequent phases to minimize partitioning skew. Phase one, described in Section 1.6.2, is responsible for applying the

Table 1.7. Themis's three stage architecture.

Phase	Description	Required?
0	Skew Mitigation	Optional
1	`map()` and shuffle	Required
2	sort and `reduce()`	Required

`map` function to each input record, and routing its output to an appropriate partition in the cluster. This is the equivalent of existing systems' map and shuffle phases. Phase two, described in Section 1.6.3, is responsible for sorting and applying the `reduce` function to each of the intermediate partitions produced in phase one. At the end of phase two, the MapReduce job is complete.

Phase one reads each input record and writes each intermediate record exactly once. Phase two reads each intermediate partition and writes its corresponding output partition exactly once. Thus, Themis maintains TritonSort's 2-IO property defined in Section 1.1.

1.6.2 Phase One: Map and Shuffle

Phase one is responsible for implementing both the `map` operation as well as shuffling records to their appropriate intermediate partition. Each node in parallel implements the stage graph pipeline shown in Figure 1.9.

The **Reader** stage reads records from an input disk and sends them to the **Mapper** stage, which applies the `map` function to each record. As the `map` function produces intermediate records, each record's key is hashed to determine the node to which it should be sent and placed in a per-destination buffer that is given to the sender when it is full. The **Sender** sends data to remote nodes using a round-robin loop of short, non-blocking `send()` calls. We call the Reader to Sender part of the pipeline the "producer-side" pipeline.

The **Receiver** stage receives records from remote nodes over TCP using a round-

robin loop of short, non-blocking `recv()` calls. We implemented a version of this stage that uses `select()` to avoid unnecessary polling, but found that its performance was too unpredictable to reliably receive all-to-all traffic at 10Gbps. The receiver places incoming records into a set of small per-source buffers, and sends those buffers to the Demux stage when they become full.

The **Demux** stage is responsible for assigning records to partitions. It hashes each record's key to determine the partition to which it should be written, and appends the record to a small per-partition buffer. When that buffer becomes full, it is emitted to the **Chainer** stage, which links buffers for each partition into separate *chains*. When chains exceed a pre-configured length, which we set to 4.5 MB to avoid doing small writes, it emits them to the **Coalescer** stage. The **Coalescer** stage merges chains together into a single large buffer that it sends to the **Writer** stage, which appends buffers to the appropriate partition file. The combination of the Chainer and Coalescer stages allows buffer memory in front of the Writer stage to be allocated to partitions in a highly dynamic and fine-grained way. We call the Receiver to Writer part of the pipeline the "consumer-side" pipeline.

A key requirement of the consumer-side pipeline is to perform large, contiguous writes to disk to minimize seeks and provide high disk bandwidth. We now describe a node-wide, application-driven disk scheduler that Themis uses to ensure that writes are large.

Each writer induces back-pressure on chainers, which causes the per-partition chains to get longer. In this way, data gets buffered within the chainer. This buffering can grow very large–to over 10GB on a machine with 24GB of memory. The longer a chain becomes, the larger the corresponding write will be. We limit the size of a chain to 14MB, to prevent very large writes from restricting pipelining. The large writes afforded by this scheduler allow Themis to write at nearly the sequential speed of the disk. Our earlier

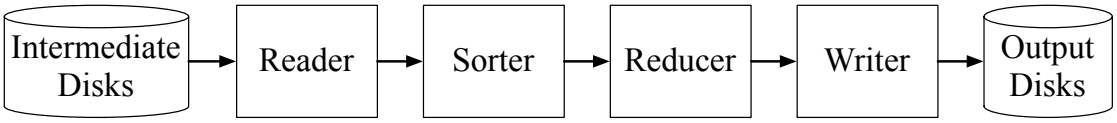


Figure 1.10. Stages of Phase Two (Sort/Reduce) in Themis

work in high performance sorting [71] provides a detailed evaluation of the relationship between write sizes and system throughput.

Signaling back-pressure between the chainer and the writer stage is done by means of *write tokens*. The presence of a write token for a writer indicates that it can accept additional buffers. When the writer receives work, it removes its token, and when it finishes, it returns the token. Tokens are also used to prevent the queues between the chainer and writer stages from growing without bound.

Relationship to TritonSort

The phase one pipeline in Themis looks very similar to TritonSort's phase one pipeline. This is intentional. In fact, the partitioning work performed by the Mapper is analogous to the NodeDistributor stage in TritonSort. The Demux and Chainer together perform a function that is similar to the LogicalDiskDistributor stage in TritonSort. Our insight is that `map` and shuffle can be accomplished simply by applying the `map` function after reading the data in the first phase of a large-scale sort operation.

1.6.3 Phase Two: Sort and Reduce

By the end of phase one, the map function has been applied to each input record, and the records have been grouped into partitions and stored on the appropriate node so that all records with the same key are stored in a single partition. In phase two, each partition must be sorted by key, and the `reduce` function must be applied to groups of records with the same key. The stages that implement phase two are shown in Figure 1.10.

There is no network communication in phase two, so nodes process their partitions

independently. Entire partitions are read into memory at once by the **Reader** stage. A **Sorter** stage sorts these partitions by key, keeping the result in memory. The **Reducer** stage applies the `reduce` function to all records sharing a key. Reduced records are sent to the **Writer**, which writes them to disk.

All records with a single key must be stored in the same partition for the `reduce` function to produce correct output. As a result, partitioning skew can cause some partitions to be significantly larger than others. Themis's memory management system allows phase two to process partitions that approach the size of main memory, and its optional skew mitigation phase can reduce partitioning skew without user intervention. We describe skew mitigation in Section 1.6.4.

A key feature of Themis's sorter stage is that it can select which sort algorithm is used to sort a buffer on a buffer-by-buffer basis. There is a pluggable *sort strategy* interface that lets developers use different sorting algorithms; currently quicksort and radix sort are implemented. Each sort strategy calculates the amount of scratch space it needs to sort the given buffer, depending on the buffer's contents and the sort algorithm's space complexity. For both quicksort and radix sort, this computation is deterministic. In Themis, radix sort is chosen if the keys are all the same size and the required scratch space is under a configurable threshold; otherwise, quicksort is used.

Relationship to TritonSort

Like with the phase one pipeline, phase two looks very similar to the second phase of TritonSort. In fact, it is simply the result of applying the `reduce` function after the sort operation, but before writing data to disk. This similarity to a previously measured high-performance sorting system is key to the performance of Themis, which we will describe in Section 1.7.

1.6.4 Phase Zero: Skew Mitigation

To satisfy the 2-IO property, Themis must ensure that every partition can be sorted in memory, since an out-of-core sort would induce additional I/Os. In addition, to support parallelism, partitions must be small enough that several partitions can be processed in parallel. Phase zero is responsible for choosing the number of partitions, and selecting a partitioning function to keep each partition roughly the same size. This task is complicated by the fact that the data to be partitioned is generated by the `map` function. Thus, even if the distribution of input data is known, the distribution of intermediate data may not be known. This phase is optional: if the user has knowledge of the intermediate data's distribution, they can specify a custom partitioning function, similar to techniques used in Hadoop.

Phase zero approximates the distribution of intermediate data by applying the `map` function to a subset of the input. If the data is homoscedastic, then a small prefix of the input is sufficient to approximate the intermediate distribution. Otherwise, more input data will need to be sampled, or phase two's performance will decrease. DeWitt et al. [22] formalize the number of samples needed to achieve a given skew with high probability; typically we sample 1 GB per node of input data for nodes supporting 8 TB of input. The correctness of phase two only depends on partitions being smaller than main memory. Since our target partition size is less than 5% of main memory, this means that a substantial sampling error would have to occur to cause job failure. So although sampling does impose additional I/O over the 2-IO limit, we note that it is a small and constant overhead.

Once each node is done sampling, it transmits its sample information to a central coordinator. The coordinator uses these samples to generate a partition function, which is then re-distributed back to each node.

Mechanism

On each node, Themis applies the `map` operation to a prefix of the records in each input file stored on that node. As the `map` function produces records, the node records information about the intermediate data, such as how much larger or smaller it is than the input and the number of records generated. It also stores information about each intermediate key and the associated record's size. This information varies based on the sampling policy. Once the node is done sampling, it sends that metadata to the coordinator.

The coordinator merges the metadata from each of the nodes to estimate the intermediate data size. It then uses this size, and the desired partition size, to compute the number of partitions. Then, it performs a streaming merge-sort on the samples from each node. Once all the sampled data is sorted, partition boundaries are calculated based on the desired partition sizes. The result is a list of "boundary keys" that define the edges of each partition. This list is broadcast back to each node, and forms the basis of the partitioning function used in phase one.

The choice of sampling policy depends on requirements from the user. Themis supports the following sampling policies:

(1) Range partitioning: For MapReduce jobs in which the ultimate output of all the reducers must be totally ordered (e.g., sort), Themis employs a range partitioning sampling policy. In this policy, the entire key for each sampled record is sent to the coordinator. A downside of this policy is that very large keys can limit the amount of data that can be sampled because there is only a limited amount of space to buffer sampled records.

(2) Hash partitioning: For situations in which total ordering of `reduce` function output is not required, Themis employs hash partitioning. In this scheme, a hash of the key is sampled, instead of the keys themselves. This has the advantage of supporting very large keys, and allowing Themis to use reservoir sampling [88], which samples data in constant space in one pass over its input. This enables more data to be sampled with a fixed amount of buffer. This approach also works well for input data that is already partially or completely sorted because adjacent keys are likely to be placed in different partitions, which spreads the data across the cluster.

1.7 Evaluation of Themis

We evaluate Themis through benchmarks of several different MapReduce jobs on both synthetic and real-world data sets. A summary of our results are as follows:

- Themis is highly performant on a wide variety of MapReduce jobs, and outperforms Hadoop by 3x - 16x on a variety of common jobs.
- Themis can achieve nearly the sequential speed of the disks for I/O-bound jobs, which is approximately the same rate as TritonSort's record-setting performance.
- Themis's memory subsystem is flexible, and is able to handle large amounts of data skew while ensuring efficient operation.

1.7.1 Workloads and evaluation overview

We evaluate Themis on the testbed described in Section 1.2, with the upgraded Cisco Nexus 5096 switch. All servers run Linux 2.6.32. Our implementation of Themis is written in C++ and is compiled with g++ 4.6.2.

To evaluate Themis at scale, we often have to rely on large synthetically-generated data sets, due to the logistics of obtaining and storing freely-available, large data sets. All

Table 1.8. A description and table of abbreviations for the MapReduce jobs evaluated in this section. Data sizes take into account 8 bytes of metadata per record for key and value sizes.

Job Name	Description	Data Size		
		Input	Interm-ediate	Output
Sort-100G	Uniformly-random sort, 100GB per node	2.16TB	2.16TB	2.16TB
Sort-500G	Uniformly-random sort, 500GB per node	10.8TB	10.8TB	10.8TB
Sort-1T	Uniformly-random sort, 1TB per node	21.6TB	21.6TB	21.6TB
Sort-1.75T	Uniformly-random sort, 1.75TB per node	37.8TB	37.8TB	37.8TB
Pareto-1M	Sort Pareto distribution, $\alpha = 1.5$, $x_0 = 100$ (1MB max key/value size)	10TB	10TB	10TB
Pareto-100M	Sort Pareto distribution, $\alpha = 1.5$, $x_0 = 100$ (100MB max key/value size)	10TB	10TB	10TB
Pareto-500M	Sort Pareto distribution, $\alpha = 1.5$, $x_0 = 100$ (500MB max key/value size)	10TB	10TB	10TB
CloudBurst	CloudBurst (two nodes, UW data)	971.3MB	68.98GB	517.9MB
PageRank-U	PageRank (synthetic uniform graph, 25M vertices, 50K random edges per vertex)	1TB	4TB	1TB
PageRank-PL	PageRank (synthetic graph with power-law vertex in-degree, 250M vertices)	934.7GB	3.715TB	934.7GB
PageRank-WEX	PageRank on WEX page graph	1.585GB	5.824GB	2.349GB
WordCount	Count words in text of WEX	8.22GB	27.74GB	812MB
n-Gram	Count 5-grams in text of WEX	8.22GB	68.63GB	49.72GB
Click-Sessions	Session extraction from 2TB of synthetic click logs	2TB	2TB	8.948GB

synthetic data sets are evaluated on 20 cluster nodes. Non-synthetic data sets are small enough to be evaluated on a single node.

All input and output data is stored on local disks without using any distributed filesystem and without replication. The integration of Themis with storage systems like HDFS is the subject of future work.

We evaluate Themis's performance on several different MapReduce jobs. A summary of these jobs is given in Table 1.8, and each job is described in more detail below.

Sort : Large-scale sorting is a useful measurement of the performance of MapReduce and of data processing systems in general. During a sort job, all cluster nodes are reading from disks, writing to disks, and doing an all-to-all network transfer simultaneously. Sorting also measures the performance of MapReduce independent of the computational complexity of the `map` and `reduce` functions themselves, since both `map` and `reduce` functions are effectively no-ops. We study the effects of both increased data density and skew on the system using sort due to the convenience with which input data that meets desired specifications can be generated. We generate skewed data with a Pareto distribution. The record size in generated datasets is limited by a fixed maximum, which is a parameter given to the job.

WordCount : Word count is a canonical MapReduce job. Given a collection of words, word count's `map` function emits `<word, 1>` records for each word. Word count's `reduce` function sums the occurrences of each word and emits a single `<word, N>` record, where N is the number of times the word occurred in the original collection.

We evaluate WordCount on the 2012-05-05 version of the Freebase Wikipedia Extraction (WEX) [94], a processed dump of the English version of Wikipedia. The

complete WEX dump is approximately 62GB uncompressed, and contains both XML and text versions of each page. We run word count on the text portion of the WEX data set, which is approximately 8.2GB uncompressed.

n-Gram Count : An extension of word count, n-gram count counts the number of times each group of *n* words appears in a text corpus. For example, given "The quick brown fox jumped over the lazy dog", 3-gram count would count the number of occurrences of "The quick brown", "quick brown fox", "brown fox jumped", etc. We also evaluate n-gram count on the text portion of the WEX data set.

PageRank : PageRank is a graph algorithm that is widely used by search engines to rank web pages. Each node in the graph is given an initial rank. Rank propagates through the graph by each vertex contributing a fraction of its rank evenly to each of its neighbors.

PageRank's `map` function is given a `<vertex ID, adjacency list of vertex IDs|initial rank>` pair for each vertex in the graph. It emits `<adjacent vertex ID, rank contribution>` pairs for each adjacent vertex ID, and also re-emits the adjacency list so that the graph can be reconstructed. PageRank's `reduce` function adds the rank contributions for each vertex to compute that vertex's rank, and emits the vertex's existing adjacency list and new rank.

We evaluate PageRank with three different kinds of graphs. The first (PageRank-U) is a 25M vertex synthetically-generated graph where each vertex has an edge to every other vertex with a small, constant probability. Each vertex has an expected degree of 5,000. The second (PageRank-PL) is a 250M vertex synthetically-generated graph where vertex in-degree follows a power law distribution with values between 100 and 10,000. This simulates a more realistic page graph where a relatively small number of pages are linked to frequently. The third (PageRank-WEX) is a graph derived from page links in

the XML portion of the WEX data set; it is approximately 1.5GB uncompressed and has 5.3M vertices.

CloudBurst : CloudBurst [62] is a MapReduce implementation of the RMAP [80] algorithm for short-read gene alignment, which aligns a large collection of small "query" DNA sequences called *reads* with a known "reference" genome. CloudBurst performs this alignment using a standard technique called *seed-and-extend*. Both query and reference sequences are passed to the `map` function and emitted as a series of fixed-size *seeds*. The `map` function emits seeds as sequence of `<seed, seed metadata>` pairs, where the seed metadata contains information such as the seed's location in its parent sequence, whether that parent sequence was a query or a reference, and the characters in the sequence immediately before and after the seed.

CloudBurst's `reduce` function examines pairs of query and reference strings with the same seed. For each pair, it computes a similarity score of the DNA characters on either side of the seed using the Landau-Vishkin algorithm for approximate string matching. The `reduce` function emits all query/reference pairs with a similarity score above a configured threshold.

We evaluate CloudBurst on the lakewash_combined_v2 data set from University of Washington [40], which we pre-process using a slightly modified version of the CloudBurst input loader used in Hadoop.

Click Log Analysis : Another popular MapReduce job is analysis of click logs. Abstractly, click logs can be viewed as a collection of `<user ID, timestamp|URL>` pairs indicating which page a user loaded at which time. We chose to evaluate one particular type of log analysis task, *session tracking*. In this task, we seek to identify disjoint ranges of timestamps at least some number of seconds apart. For each such range of times-

tamps, we output <user ID, start timestamp|end timestamp|start URL|end URL> pairs.

The map function is a pass-through; it simply groups records by user ID. The reduce function does a linear scan through records for a given user ID and reconstructs sessions. For efficiency, it assumes that these records are sorted in ascending order by timestamp. We describe the implications of this assumption in the next section.

1.7.2 Job Implementation Details

In this section, we briefly describe some of the implementation details necessary for running our collection of example jobs at maximum efficiency.

Combiners : A common technique for improving the performance of MapReduce jobs is employing a *combiner*. For example, word count can emit a single <word, *k*> pair instead of *k* <word, 1> pairs. Themis supports the use of combiner functions. We opted to implement combiners within the mapper stage on a job-by-job basis rather than adding an additional stage. Despite what conventional wisdom would suggest, we found that combiners actually decreased our performance in many cases because the computational overhead of manipulating large data structures was enough to make the mapper compute-bound. The large size of these data structures is partially due to our decision to run the combiner over an entire job's intermediate data rather than a small portion thereof to maximize its effectiveness.

In some cases, however, a small data structure that takes advantage of the semantics of the data provides a significant performance increase. For example, our word count MapReduce job uses a combiner that maintains a counter for the top 25 words in the English language. The combiner updates the appropriate counter whenever it encounters one of these words rather than creating an intermediate record for it. At the

end of phase one, intermediate records are created for each of these popular words based on the counter values.

Improving Performance for Small Records : The `map` functions in our first implementations of word count and n-gram count emitted `<word/n-gram, 1>` pairs. Our implementations of these `map` functions emit `<hash(word), 1|word>` pairs instead because the resulting intermediate partitions are easier to sort quickly because the keys are all small and the same size.

Secondary Keys : A naïve implementation of the session extraction job sorts records for a given user ID by timestamp in the `reduce` function. We avoid performing two sorts by allowing the Sorter stage to use the first few bytes of the value, called a *secondary key*, to break ties when sorting. For example, in the session extraction job the secondary key is the record's timestamp.

1.7.3 Performance

We evaluate the performance of Themis in two ways. First, we compare performance of the benchmark applications to the cluster's hardware limits. Second, we compare the performance of Themis to that of Hadoop on two benchmark applications.

Performance Relative to Disk Speeds

The performance of Themis on the benchmark MapReduce jobs is shown in Figure 1.11. Performance is measured in terms of *MB/s/disk* in order to provide a relative comparison to the hardware limitations of the cluster. The 7200 RPM drives in the cluster are capable of approximately 90 MB/s/disk of sequential write bandwidth, which is shown as a dotted line in the figure. A job running at 90 MB/s/disk is processing data as fast as it can be written to the disks.

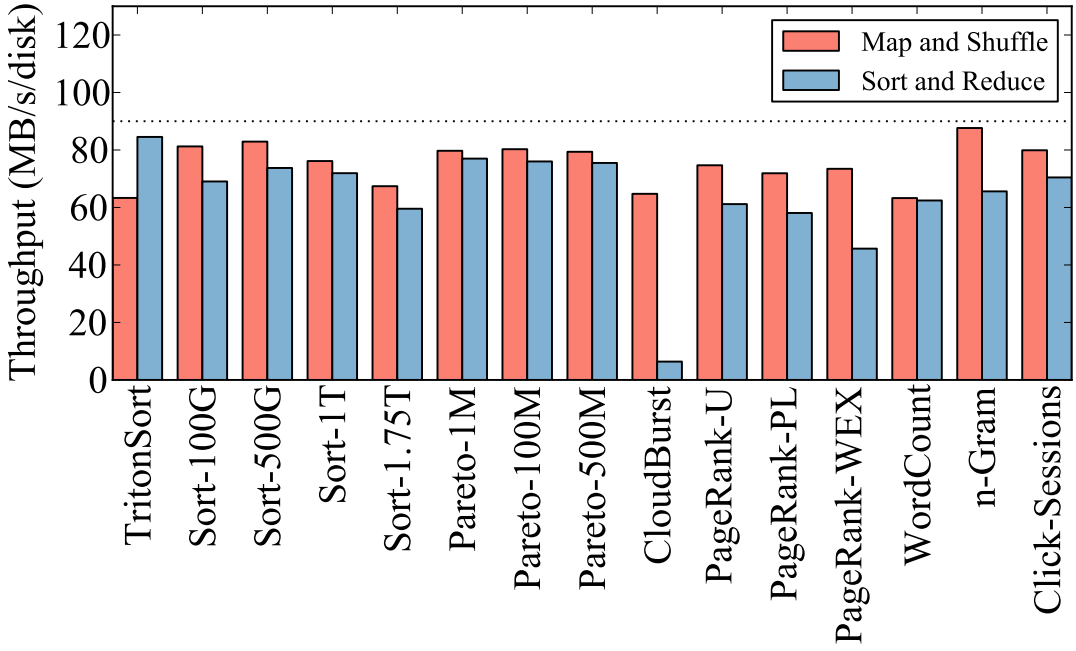


Figure 1.11. Performance of evaluated MapReduce jobs. Maximum sequential disk throughput of approximately 90 MB/s is shown as a dotted line. Our TritonSort record from 2011 is shown on the left for comparison.

Most of the benchmark applications run at near maximum speed in both phases. CloudBurst's poor performance in phase two is due to the computationally intensive nature of its `reduce` function, which is unable to process records fast enough to saturate the disks. More CPU cores are needed to drive computationally intensive applications such as CloudBurst at maximum speed in both phases. Notice however that CloudBurst is still able to take advantage of our architecture in phase one.

We have included TritonSort's performance on the Indy 100TB sort benchmark for reference. TritonSort's 2011 Indy variant runs a much simpler code base than Themis. We highlight the fact that Themis's additional complexity and flexibility does not impact its ability to perform well on a variety of workloads. Our improved performance in phase one relative to TritonSort at scale is due to a variety of internal improvements and optimizations made to the codebase in the intervening period, as well as the improved

Table 1.9. Performance comparison of Hadoop and Themis.

Application	Running Time		Improvement Over Hadoop
	Hadoop	Themis	
Sort-500G	28881s	1789s	16.14x
CloudBurst	2878s	944s	3.05x

memory utilization provided by moving from buffer pools to dynamic memory management. Performance degradation in phase two relative to TritonSort is mainly due to additional CPU and memory pressure introduced by the Reducer stage.

Comparison with Hadoop

We evaluate Hadoop version 1.0.3 on the Sort-500G and CloudBurst applications. We started with a configuration based on the configuration used by Yahoo! for their 2009 Hadoop sort record [81]. We optimized Hadoop as best we could, but found it difficult to get it to run many large parallel transfers without having our nodes blacklisted for running out of memory.

The total running times for both Hadoop and Themis are given in Table 1.9. I/O-bound jobs such as sort are able to take full advantage of our architecture, which explains why Themis is more than a factor of 16 faster. As explained above, CloudBurst is fundamentally compute-bound, but the performance benefits of the 2-IO property allow the Themis implementation of CloudBurst to outperform the Hadoop implementation by a factor of 3.

1.7.4 Skew Mitigation

Next, we evaluate Themis's ability to handle skew by observing the sizes of the intermediate data partitions created in phase one. Figure 1.12 shows the partition sizes produced by Themis on the evaluated applications. The error bars denoting the 95% confidence intervals are small, indicating that all partitions are nearly equal in size.

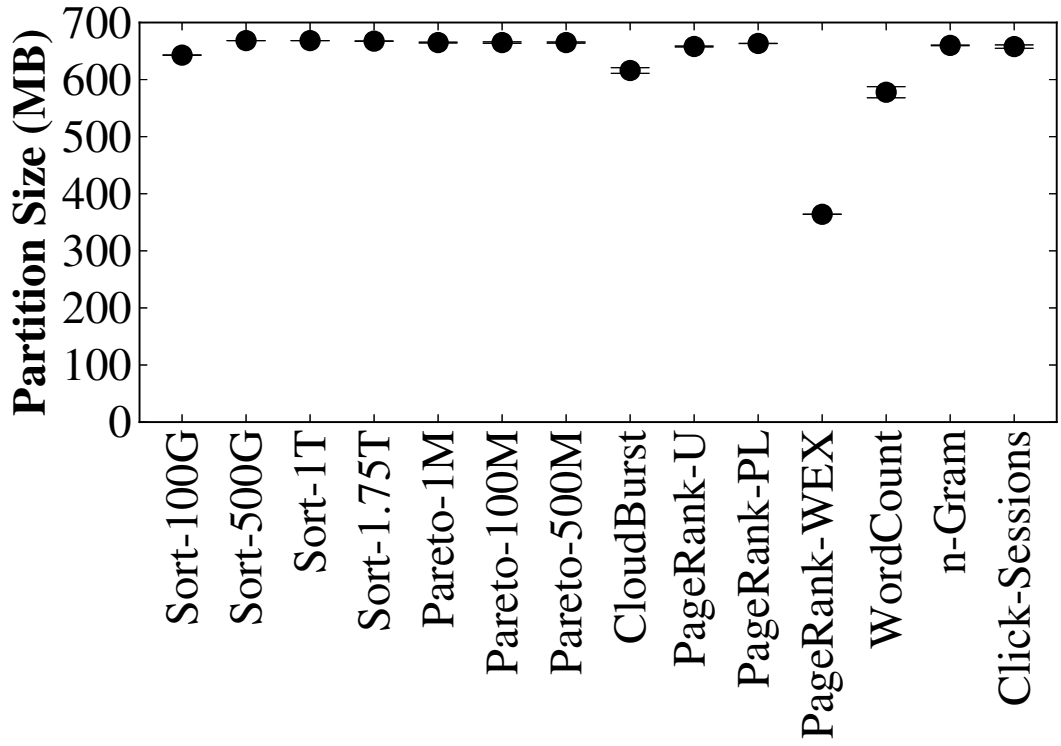


Figure 1.12. Partition sizes for various Themis jobs. Error bars denoting the 95% confidence intervals are hard to see due to even partitioning.

This is unsurprising for applications with uniform data, such as sort. However, Themis also achieves even partitioning on very skewed data sets, such as Pareto-distributed sort, PageRank, and WordCount. PageRank-WEX has fairly small partitions relative to the other jobs because its intermediate data size is not large enough for phase zero to create an integer number of partitions with the desired size.

1.7.5 Write Sizes

One of primary goals of phase one is to do large writes to each partition to avoid unnecessary disk seeks. Figure 1.13 shows the median write sizes of the various jobs we evaluated. For jobs like Sort and n-Gram where the `map` function is extremely simple and mappers can map data as fast as readers can read it, data buffers up in the Chainer

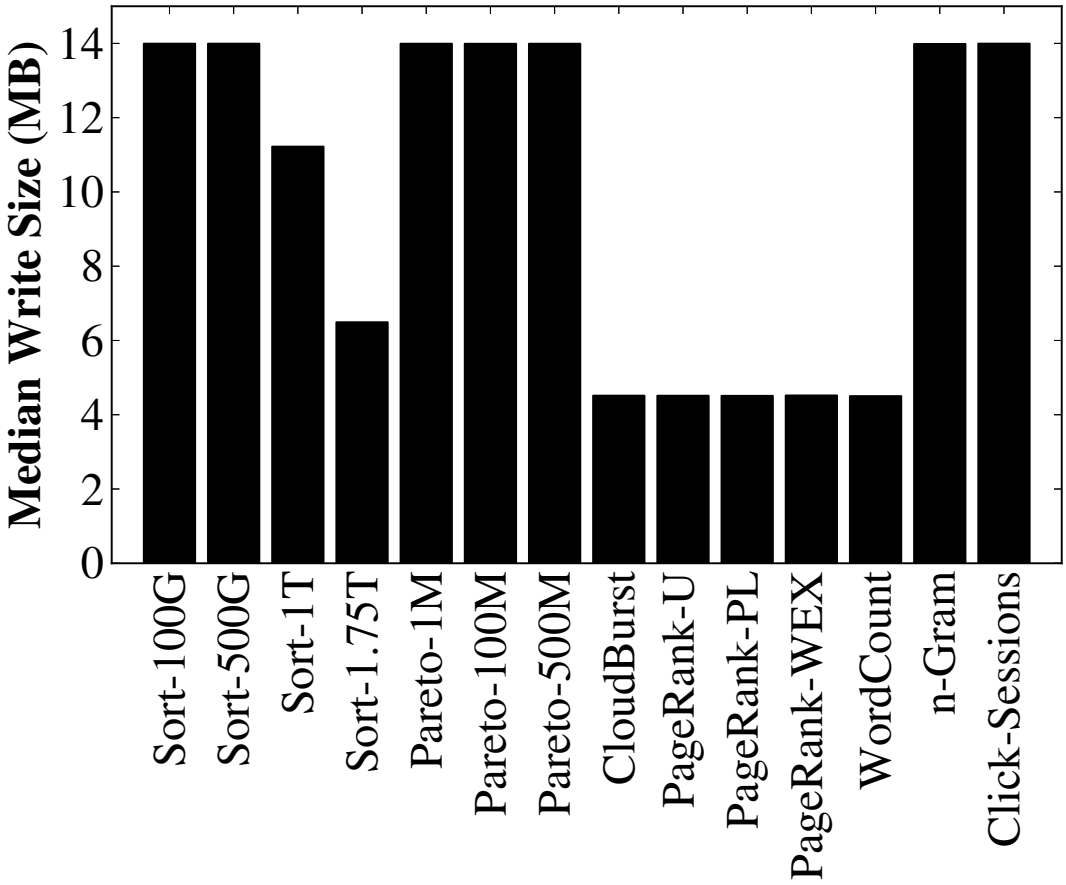


Figure 1.13. Median write sizes for various Themis jobs

stage and all writes are large. As the amount of intermediate data per node grows, the size of a chain that can be buffered for a given partition decreases, which fundamentally limits the size of a write. For example, Sort-1.75T writes data to 2832 partitions, which means that its average chain length is not expected to be longer than about 5 MB given a receiver memory quota of 14GB; note, however, that the mean write size is above this minimum value, indicating that the writer is able to take advantage of temporary burstiness in activity for certain partitions. If the stages before the Writer stage cannot quite saturate it (such as in WordCount, CloudBurst and PageRank), chains remain fairly small. Here the minimum chain size of 4.5 MB ensures that writes are still reasonably large. In the case of PageRank-WEX, the data size is too small to cause the chains to

ever become very large.

1.8 Bridging the Gap Between Software and Hardware

The systems described thus far in this chapter required years of development to reach the desired levels of performance. A large portion of this effort was spent bridging the gap between the software and the hardware. We note that we acquired the hardware testbed described in Section 1.2 in the very early stages of this project, so we were able to design the software systems to match the performance levels afforded by our testbed. In general it may not always be possible to design systems this way, but if the target hardware platform is available, designing the software to suit it can be a very successful development strategy.

As alluded to in Section 1.2, a good way to evaluate a hardware platform is to consider the relative performance levels of each piece of hardware. I/O devices in particular lend themselves to this strategy, since these devices tend to have well-understood bandwidth levels and are often bottlenecks in a large-scale deployment.

As an illustration of this point, the hard disk drives in our cluster are capable of approximately 90 MB/s of sequential read or write bandwidth. Since each server has 16 drives, our servers support 1440 MB/s of read-only or write-only storage bandwidth, or 720 MB/s of read/write bandwidth. Our 10 Gb/s network interfaces have a theoretical maximum performance of 1250 MB/s, although in reality performance will be slightly lower. Therefore, our hard disks will be a bottleneck in a read/write workload like TritonSort or Themis. Network can become the bottleneck in read-only or write-only workloads, such as the first phase of MinuteSort (Section 1.5.2).

These bandwidth levels are orders of magnitude slower than a CPU executing simple instructions. While real-world workloads can become CPU-bound, many data-intensive applications will be I/O-bound. Thus the techniques in this work are critical to

achieving good performance.

If an application is I/O-bound, it is absolutely critical that the software can take full advantage of the hardware. In other words, if a disk is capable of 90 MB/s, then the software ought to be able to issue reads or writes at a rate of at least 90 MB/s. Any software-imposed slowdown is simply wasted performance and therefore wasted money, energy, or time.

We now consider several features of our hardware configuration that require special attention in software to achieve high levels of performance.

1.8.1 Hard Disk Drives

An enormous amount of effort in the development of TritonSort and Themis was spent optimizing the performance of magnetic hard disk drives (HDDs). These drives have a mechanical component that requires significant delays (on the order of milliseconds) whenever the read or write address changes significantly. This behavior, called *seeking*, can be fatal to an application that is trying to squeeze every ounce of performance from a hard disk. A workload that issues sequential I/O operations will not lose performance due to seek overheads. In practice, an application that issues large, random I/O operations can still achieve near-maximal throughput. The time to service a large I/O will dominate any time spent seeking. In the case of our disks, "large" turns out to be on the order of 10 MB.

The simplest way to ensure large I/O operations is to store data in large memory buffers. However, as we will see in Section 1.8.3, it is not always possible to allocate large memory buffers, especially if many of them are needed, as in the case of the LogicalDiskDistributor and Demux in TritonSort and Themis, respectively.

To solve this problem, we created a *write chaining* mechanism, implemented by the LogicalDiskDistributor (Section 1.3) and the Chainer (Section 1.6). In both of these

cases, data records are stored in lists small buffers until a list becomes large. At this point, the large list is combined into a single large buffer, which is then written to disk. By ensuring lists are large, we ensure that our random writes are also large. However, by using small buffers to store the actual data, we can get away with allocating far less total memory, since a very small fraction of our allocated memory regions are empty or partially empty. In other words, the fraction of allocated bytes that hold useful data is much higher using write chaining than without. This property is absolutely critical to issuing large random writes in the presence of low memory, and is a major contributing factor to the success of TritonSort and Themis.

In addition to ensuring reads and writes are large, another important technique is reducing operating system overheads. We found that bypassing the file buffer cache in the operating system using the `O_DIRECT` I/O mode improved performance significantly for both reads and writes to disk. This I/O mode imposes alignment limitations on memory addresses, memory region lengths, I/O sizes, and file sizes that are more of an annoyance than anything. Nevertheless, developing around these restrictions was a significant burden. In particular, transitioning from supporting only 100-byte records to records of arbitrary size required significant application modification to support direct I/O for reading and writing.

1.8.2 10 Gb/s Networking

As described above, the networking interface can also become a bottleneck if not properly utilized. A major concern in TritonSort and Themis is the ability to send and receive data at a significant fraction of line rate using an all-to-all communication pattern that is typical of data shuffle applications like MapReduce.

As alluded to in Section 1.3, there are a number of different issues one has to overcome in implementing a networking subsystem. The first is ensuring fairness

in an all-to-all communication pattern. The initial implementation of TritonSort used application-enforced rate limiting in combination with TCP to ensure that each transfer proceeded at the appropriate rate. This solution has limited applicability in practice, but works well in the evaluation in this chapter.

Another issue, in addition to fairness in the network, is how to schedule the I/O operations with limited overhead. An initial implementation included one thread per network peer using a loop of blocking `send()` and `recv()` calls. It was quickly discovered that this approach did not scale, so a single-threaded implementation that visited each connection in round-robin order was chosen. We found this implementation sufficient for achieving the desired levels of performance on our 10 Gb/s networking, but as we will see in the next chapter, this solution is far from optimal.

1.8.3 Low Memory Conditions

As mentioned above, the absence of abundant memory can be a significant challenge for efficient systems design. The write chaining mechanism is an example of a sophisticated solution to a very simple problem. Namely the hardware platform does not have enough memory to run the application with the desired configuration parameters. A typical solution to this problem is *swapping*, where the operating system will temporary store memory pages on disk until they are needed. While this solution prevents a system crash, it eliminates any chance of achieving acceptable levels of performance. In particular, the disks that would be necessary to swap to are already running at maximum speed, thanks to our careful attention to I/O performance and efficiency. Any extra reads or writes would not only reduce available application bandwidth, but would also interfere with our sequential or near-sequential I/O pattern, reducing performance further. Therefore, we disable swap in all of our evaluations.

With swap disabled, we run into the issue of application crashes due to over-

allocation. TritonSort solved this issue by carving the available memory into fixed sized pools of buffers, as described in Section 1.3. While this method worked well for 100-byte records, it is inflexible and inefficient when records of arbitrary size are used. We therefore abandoned this allocation technique and created a centralized memory allocator that had visibility into the entire application pipeline. This allocator was able to satisfy memory requests of arbitrary size and prioritize requests in the system to prevent starvation. In Themis we refer to this allocation style as constraint-based allocation [69]. This allocation scheme worked, but was unnecessarily complicated and experienced performance issues. Another scheme, called quota-based allocation, was introduced to solve this problem. Quota-based allocation essentially operated like buffer pools by using a configuration parameter, the *quota*, to limit memory usage in portions of the system. However, memory request sizes are allowed to be arbitrary, solving the flexibility issue.

Solving the application-level memory allocation problem still isn't enough however. Allowing allocations of arbitrary size can introduce memory fragmentation in the `malloc` implementation. We experimented with alternative memory allocation libraries, including TCMalloc [75] and jemalloc [27]. At different points in our development, we used different combination of these libraries and the standard `malloc` for different phases in Themis MapReduce. The current version of Themis uses TCMalloc for the entire program, and this setting was chosen based on empirical performance measurements.

All of the issues touched upon in this section were caused by the underlying hardware platform described in Section 1.2. With a different hardware platform, it is likely we would have experienced a host of different issues. The remainder of this dissertation is primarily concerned with this point, and investigates alternative, faster hardware technologies.

1.9 Acknowledgements

Chapter 1 includes material as it appears in Proceedings of the 8th Annual USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2011. Rasmussen, Alexander; Porter, George; Conley, Michael; Madhyastha, Harsha V.; Mysore, Radhika Niranjan; Pucher, Alexander; Vahdat, Amin. The dissertation author was among the primary authors of this paper.

Chapter 1 also includes material as it appears in Proceedings of the 3rd Annual ACM Symposium on Cloud Computing (SOCC) 2012. Rasmussen, Alexander; Conley, Michael; Kapoor, Rishi; Lam, Vinh The; Porter, George; Vahdat, Amin. The dissertation author was among the primary authors of this paper.

Chapter 2

Next Generation Clusters

We now turn our attention to next generation cluster technologies. The work in the previous chapter focused on magnetic hard disk drives. These drives are cheap and offer relatively high storage capacities, but they are typically very slow compared to other resources like CPU or memory. In fact, the write chaining optimization described in the previous chapter was necessary in part because these drives are so slow. It is therefore critical to get every bit of available bandwidth from the devices.

While we do not believe disks will go away, the current trend in data centers and cloud computing is moving towards nonvolatile memories to replace disks for many applications. In particular, flash memory, in the form of a solid state drive, is an affordable, high-performance storage solution. In this chapter, we will consider a variety of different flash configurations, including both traditional SATA-based SSDs, and newer PCIe-attached flash memories, and their implications on I/O-efficient system design.

To match the storage bandwidth provided by flash memory, next generation clusters must make use of fast networking technologies, such as InfiniBand or 40 Gb/s Ethernet. As in the case of high-speed storage, efficient systems must be able to drive application throughput at the network line rate. Further, the application's networking subsystem must be efficient in order to transfer data at high speeds.

The design of such systems is complicated further by non-uniform memory access

(NUMA) based hardware platforms, which may be required to get a balanced mix of hardware resources in a single server. These systems break the multi-core abstraction present in high-performance servers because an application's performance will vary depending on which core it runs on. In other words, the application designer must take even greater care to ensure performance does not degrade.

In this chapter, we describe several next-generation cluster hardware platforms. We then describe in detail a number of upgrades to the Themis framework described in Chapter 1 in order to achieve high levels of performance on these hardware platforms. Next, we give an evaluation of Themis on these clusters using these upgrades. We conclude with a discussion of the important themes of this work and their implications for future work.

2.1 Hardware Platforms

Before we can describe the application features necessary to support next generation clusters, it first required to explain what exactly we mean by *next generation.* For the purpose of this work, a **next generation cluster** is one in which the hardware technologies employed are readily available, but are not yet commonplace. A good metric here is to consider how many years from the present time it will take for these technologies to be ubiquitous in large-scale production clusters. While this metric is somewhat unclear because it involves an unknown future, it can certainly be estimated.

In this chapter, we will consider a next generation cluster to be one that is approximately 5-10 years ahead of its time. As a concrete example, one of the clusters will consider, Gordon (Section 2.1.1), was announced in late 2011, and contains SSD-based storage at a large-scale. As of this writing, three and a half years later, large-scale SSD-based deployments are available in Amazon's Elastic Cloud Computing (EC2) service. The PCIe-attached flash devices we discuss in Section 2.1.2, however, are not

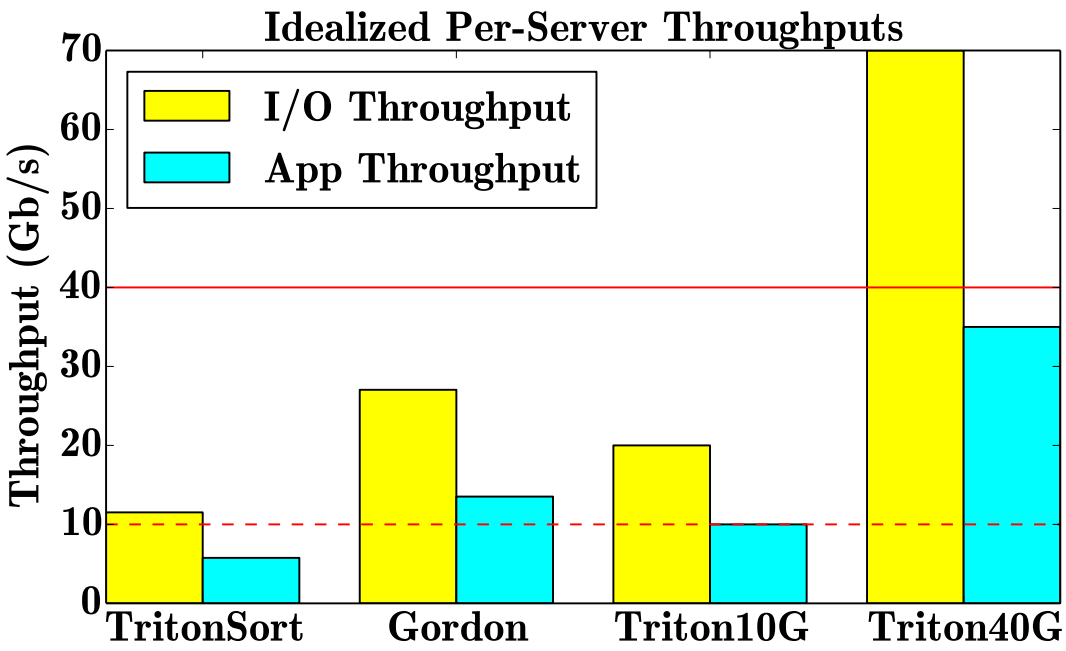


Figure 2.1. Comparison of hardware platforms and their performance levels. Application throughput is approximately half of the maximum I/O bandwidth due to the read/write nature of the application. The cluster described in Chapter 1 is shown for comparison.

widely deployed in clusters today, and are probably still a few years out from universal adoption.

We now give a detailed description of the next generation clusters available to us in this work. A summary of their maximum performance levels is shown in Figure 2.1.

2.1.1 The Gordon Supercomputer

The Gordon supercomputer is a large scientific computing cluster available at the San Diego Supercomputer Center (SDSC) [1]. It consists of two types of nodes: 1024 compute nodes and 64 I/O nodes. We list the full specifications in Table 2.1. The I/O nodes each contain 16 SSDs that are each 300 GB in capacity. This configuration yields a total of 1024 SSDs, one per compute node. In fact, the default configuration of the cluster has each compute node remotely mount one of the SSDs on the I/O nodes.

Table 2.1. System specification for the Gordon supercomputer.

	Compute Node	I/O Node
Nodes	1024	64
CPU Type	2x Intel EM64T Xeon	2x Intel X5650
CPU Generation	Sandy Bridge	Westmere
CPU Cores	16	12
Core Speed	2.6 GHz	2.67 GHz
Memory	64 GB	48 GB
Memory Speed	85 GB/s	64 GB/s
Local Storage	-	16x 300 GB SSD
Storage Speed	-	4.3 GB/s read, 3.4 GB/s write
Network	4x QDR Infiniband (40 Gb/s)	4x QDR Infiniband (40 Gb/s)

The compute nodes and I/O nodes are interconnected with a 4x4x4 3D torus network topology, which provides high bandwidth between nearby nodes. Each switch hosts 16 compute nodes and one I/O node, which are connected via 4x QDR Infiniband. Switches are connected to each other with three 4x QDR Infiniband links. This network topology is not ideal for applications like TritonSort and Themis, which perform all-to-all network transfers. Nevertheless, the resource is available to us, so we choose to evaluate it in this work.

Being a scientific supercomputer, Gordon's intended use case is data processing for scientists who have large data sets. The supercomputer attaches to a large 4 PB Lustre-based, shared, parallel file system that is constructed from hard disk drives. Users store large data sets in this shared file system. When they wish to process the data, they request time on Gordon in the form of compute nodes. Next, they download their data sets to the SSDs as a form of high-speed *scratch* storage. After the data processing job is complete, interesting results are uploaded back to the Lustre file system.

Because we are interested primarily in stressing the performance of the flash in this work, we typically do not use the Lustre-based file system, although we will consider it briefly in our evaluation. In our case, we consider the SSDs not as scratch space, but as

Table 2.2. The different configurations for a compute node on Gordon.

	Default	BigflashR	Bigflash16
SSDs	1	16	16
Capacity	300 GB	4.8 TB	4.8 TB
Exposed	single disk	RAID0 array	separate disks

Table 2.3. The Triton10G cluster.

Nodes	4
CPU	Intel Xeon E3-1230
Cores	4
Hyperthreads	8
Core Speed	3.3 GHz
Memory	32 GB
Storage	FusionIO ioDrive2 Duo
Storage Speed	3 GB/s read, 2.5 GB/s write
Capacity	2.4 TB
Network	40 Gb/s Ethernet

the main storage devices attached to the compute nodes. For this purpose, one SSD per node is usually not sufficient. Gordon also allows users to request *Bigflash* configurations. A Bigflash node on Gordon remotely attaches to 16 SSDs, rather than just one. These can be exposed either as a RAID0 array, or as individual disks, as shown in Table 2.2. As we will see in Section 2.3.3, the choice of node configuration is incredibly important for achieving high performance.

2.1.2 Triton10G

In addition to the SATA-based SSDs used in Gordon, we are interested in the performance implications of even higher performance flash devices. In this section we consider FusionIO [74] flash devices, which attach via PCI-Express. This use of a fast bus attachment enables the devices to reach speeds not possible with a typical SATA connection.

The first platform we consider is the Triton10G cluster, shown in Table 2.3. Each

Table 2.4. The Triton40G cluster.

Nodes	2
CPU	4x Intel Xeon E5-4610
Cores	24
Hyperthreads	48
Core Speed	2.4 GHz
Memory	128 GB
Storage	3x FusionIO ioDrive2 Duo, 1x FusionIO ioDrive2
Storage Speed	10.5 GB/s read, 8.8 GB/s write
Capacity	8.4 TB
Network	40 Gb/s Ethernet

FusionIO ioDrive2 Duo is exposed as two separate block devices that can be accessed independently. These devices are capable of 3 GB/s of read bandwidth and 2.5 GB/s of write bandwidth when both block devices are used. Therefore, it is possible to read data from one device, process it, and write it back to the other device at a rate of 1.25 GB/s, or 10 Gb/s, hence the "10G" in the name. Additionally, it is worth noting that these devices support millions of IOPS. While we do not consider IOPS-bound workloads here, this is still a very impressive level of performance.

We note that although the nodes have 40 Gb/s network interfaces and are connected to a 40 Gb/s switch, they do not have enough storage bandwidth to achieve 40 Gb/s of throughput in a balanced software configuration. An application like Themis or TritonSort, which reads data, transfers it over the network, and writes data simultaneously will be storage-limited in this particular configuration to roughly 10 Gb/s of application throughput.

2.1.3 Triton40G

As described above, the Triton10G cluster has far more network bandwidth than storage bandwidth. The Triton40G cluster, shown in Table 2.4, solves this imbalance. In particular, each server hosts enough PCIe slots to contain four FusionIO ioDrive2 Duos.

However, due to limited resources, we only have enough FusionIO ioDrive2 Duos to put three in each server. We fill the fourth slot with a standard FusionIO ioDrive2, which is essentially half of a Duo. More accurately, a Duo is two ioDrive2s packaged together in the same physical device. Therefore, these servers have enough storage bandwidth to nearly match the network bandwidth available from the 40 Gb/s Ethernet.

We note that in order to build such a server, we had to use a non-uniform memory access (NUMA) multi-socket configuration. As we will see in Section 2.5, this architecture presents a significant challenge to achieving high performance.

These three next generation clusters will set the context for the features and optimizations that will be described in the following sections.

2.2 Compute and Memory Optimizations

In Chapter 1, we described the TritonSort and Themis frameworks in the context of a particular disk-based cluster. Because these disks were very slow relative to flash-based SSDs, the software framework did not need to be terribly efficient with its CPU-bound operations. Because each disk is capable of reading or writing at about 90 MB/s, the framework only needs to run at a speed of 720 MB/s on any individual server in order to drive the disks at 100% utilization.

However, the PCIe-attached flash drives described in Section 2.1.2 are capable of speeds far in excess of 720 MB/s. Therefore, the framework needs to be upgraded to support such speeds without running out of available CPU cycles. We now describe some of these upgrades and their implications on efficient systems design.

2.2.1 Efficient Data Format Handling

As part of its general-purpose MapReduce computational model, Themis has far more complicated record handling mechanics than its predecessor, TritonSort. A

ByteStreamConverter is a type of worker, or thread, in Themis that formats arbitrary streams of bytes into structured records, as illustrated in Figure 2.2. This machinery is necessary to translate between operations that function over records, e.g. the `map` function, and operations that function over byte streams, e.g. reading a file from disk. Furthermore, a user may want to apply data formatting functions to records read from disk, and the ByteStreamConverter is an appropriate place to do such work.

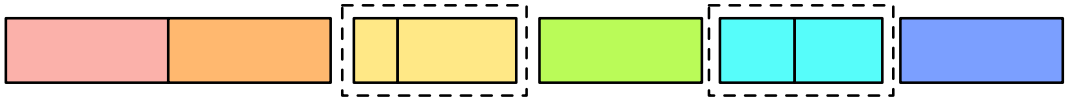
Early implementations of the ByteStreamConverter copied data record-by-record into memory regions consisting of only whole records. This functionality allows disks to be read in fixed size increments, e.g. multiples of a flash page size., without having to worry about ending the read in the middle of a record. However, the record-by-record copy dramatically increases the CPU and memory usage of the system. One optimization we made was to re-use as much of existing memory regions as possible. Rather than copy every record, only the incomplete fragments of records on either end of a memory region are copied to a separate region, as illustrated in Figure 2.2d. This simple change improved the processing throughput of the ByteStreamConverter on the Triton10G cluster from 1180 MB/s to 2664 MB/s, a 125% increase.

Another optimization was made to the ByteStreamConverter placed after the Receiver in the shuffle part of the *map and shuffle* phase, shown in Figure 2.3a. In fact, it was noted that this converter was entirely unnecessary as long as Mapper output buffers consisted entirely of complete records. In this case, before we transmit map output records across the network, we first send metadata including the number of bytes we intend to send. This metadata allows the Receiver to allocate memory regions of exactly the right number of bytes, guaranteeing they will include only whole records. With this simple change, we can entirely eliminate the ByteStreamConverter from the receive-side of the first phase, as shown in Figure 2.3b. The reduction in CPU usage by eliminating this thread improved end-to-end performance by about 15% on the Triton10G cluster.

(a) Contiguous data records residing in a file on disk.

(b) Reader threads read data into fixed size memory regions based on the properties of the storage devices. This can fragment records across memory regions.

(c) ByteStreamConverters read record metadata and reattach fragments in such a way that memory regions contain only whole records, facilitating record processing such as a `map` function.



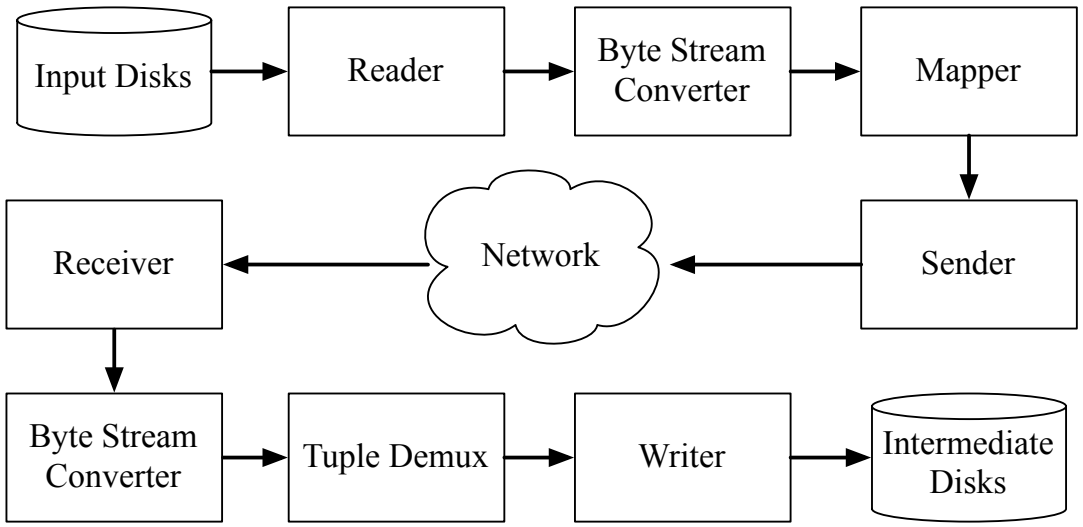
(d) Optimization for ByteStreamConverters that reduces memory allocations and CPU usage. The portions of memory regions that contain whole records are passed through unmodified. Only the fragments of records on either end of a memory region are copied. These copies are highlighted with a dotted-box.

Figure 2.2. Graphical representation of the functionality of a ByteStreamConverter. Colored rectangles indicate data records that might be split between memory regions.

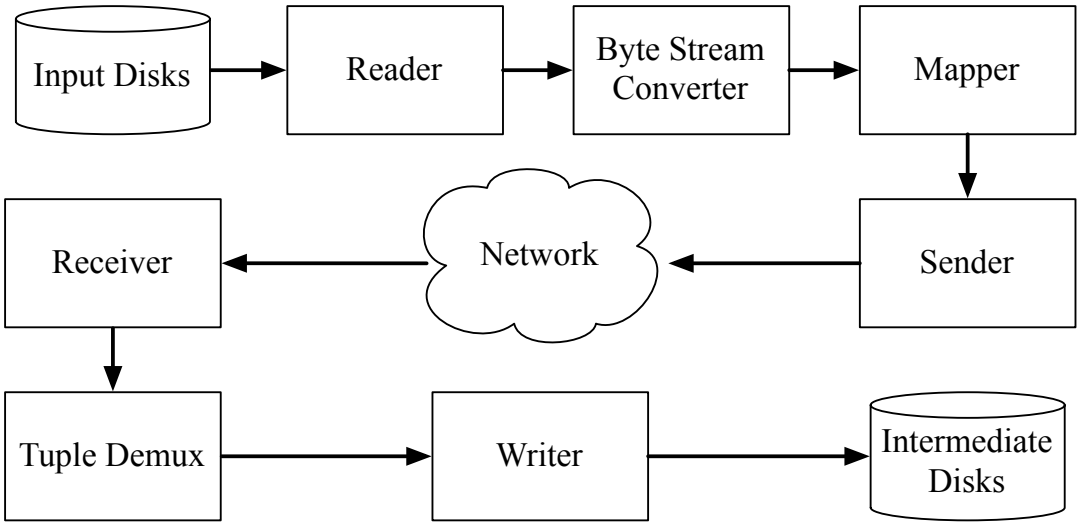
2.2.2 Write Chaining

The write chaining mechanism described in Section 1.8.1 and illustrated in Figure 1.5 is critical for obtaining high performance on hard disk drives under low-memory conditions. This is due to the inherent seek latency in hard drives that causes small, random I/O operations to have low performance. Write chaining essentially allows for large, random I/O operations, which approximate the performance of sequential I/O, even in the presence of low memory. In fact, we found in Chapter 1 that write chains of about 10 MB get near maximal performance on hard disk drives.

However, flash memory does not possess the same mechanical seek latency



(a) Pipeline diagram of phase one, the map and shuffle phase, with a ByteStreamConverter after the Receiver.



(b) Optimized pipeline that removes the ByteStreamConverter from the receiver side. Intelligent memory-allocation in the receiver and a very small amount of metadata enables the elimination of a thread from the system.

Figure 2.3. The map and shuffle phase with and without a ByteStreamConverter after the Receiver.

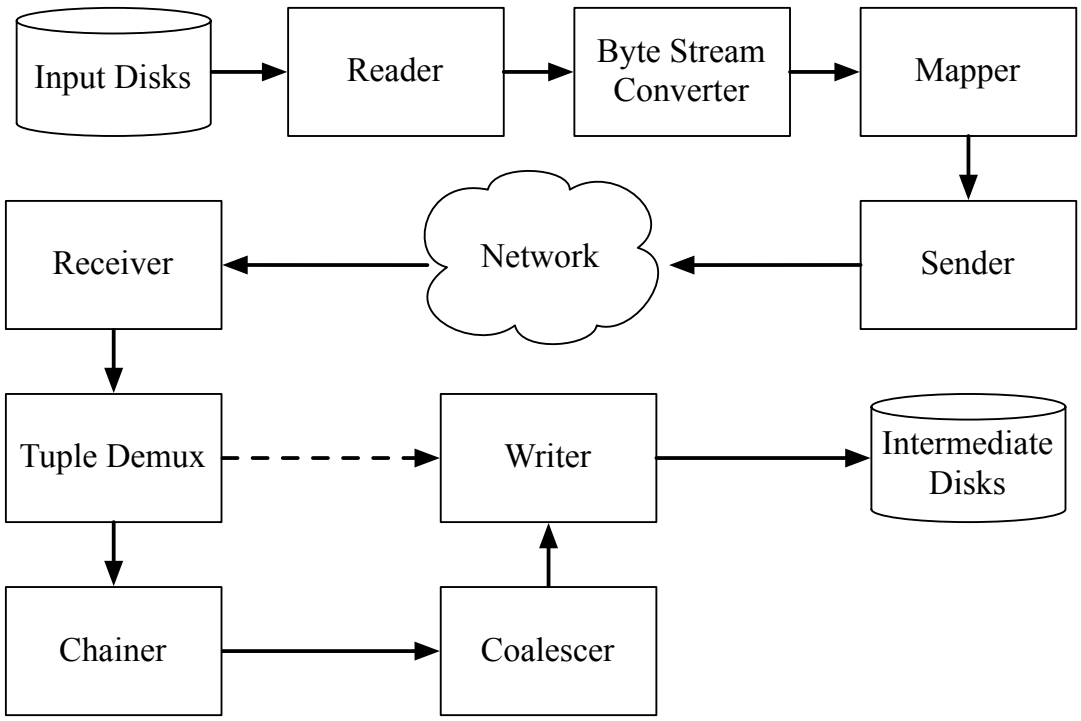


Figure 2.4. Write chaining is performed by the Chainer and Coalescer. By moving data records directly from the TupleDemux to the Writer (dotted arrow), we can eliminate two threads from the system, thereby reducing CPU usage and increasing performance.

properties that plague hard disk drives. In fact, many flash devices are optimized for high levels of random IOPS. A consequence of this physical property is that maximum bandwidth can be obtained with random I/O operations that are much smaller in size than on hard disk drives. The FusionIO ioDrive2 Duo devices used in the Triton10G and Triton40G clusters can achieve good performance using random I/O operations that are as small as 4 KB. However, the best levels of performance can be achieved with I/O sizes on the order of 1 MB, about one order of magnitude smaller than that which we observed for hard disk drives.

Since the random writes in TritonSort and Themis no longer must be so large on flash devices, the write chaining mechanism loses much of its advantage. In fact, we found that the extra CPU work done by write chaining outweighed its benefits for flash

devices. In order to quantify the performance lost by using write chaining, we configure the map and shuffle phase to run up to a specific point, called a *sink*. After the sink, data record processing ceases and memory regions are simply freed. We noticed that placing a sink directly after the receiver-side ByteStreamConverter caused the Triton10G cluster to run at a rate of 1030 MB/s. Placing a sink after the Chainer reduced performance levels to 850 MB/s, or about a 17% reduction. Running the entire pipeline resulted in a throughput of 630 MB/s, a further reduction of 36%. We cannot remove the TupleDemux because it serves a critical function for application correctness. However, the Chainer and Coalescer can certainly be eliminated, and help recover some of this performance loss. Therefore, we remove these threads when using flash devices, as illustrated in Figure 2.4.

2.2.3 Flexible Memory Allocation

As described in Section 1.8.3, the choice of application-level memory allocator significantly affects performance. While the constraint-based allocator worked well at times for the cluster evaluated in Chapter 1, it has too much overhead to be useful for faster cluster technologies like flash memories and high-speed networking employed in the Gordon, Triton10G, and Triton40G clusters.

As part of the upgrades for running on these clusters, we implemented quota-based allocation in all parts of Themis. This allocation scheme is both flexible and highly efficient. Using quota-based allocation increases the number of configuration parameters, which can complicate application tuning. However, the efficiency benefits outweigh the additional configuration complexity. In particular, this change improved the performance of the sort and reduce phase on Triton10G from 716 MB/s to 800 MB/s, which is about 12%.

2.3 Optimizations for Flash-Based Storage

We now describe the optimizations necessary to support flash-based storage technologies in Themis. We note that upon running Themis for the first time on the Triton10G cluster, we were able to reach speeds of 630 MB/s in the map and shuffle phase, and 708 MB/s in the sort and reduce phase. These speeds are comparable to those achievable on the cluster described in Chapter 1. However, the Triton10G cluster's flash-based storage devices are capable of much greater speeds, as seen in Figure 2.1.

While the compute and memory optimizations described in the previous section do help, they simply remove the artificial restrictions preventing us from getting good performance. Further care must be taken to actually achieve great performance on high-speed flash devices. In this section, we give a detailed breakdown of all the changes necessary to run high performance applications on flash-based storage.

2.3.1 Direct I/O

One important feature for achieving high performance on storage devices of all types is direct I/O. Direct I/O is a mechanism for bypassing the operating system's buffer cache, which caches in memory blocks that reside on disk. Applications that touch data records multiple times can benefit substantially from the buffer cache. However, applications like TritonSort and Themis, which process the entire data set before touching a single record twice get essentially no benefit from this operating system feature when running on data sets that are much larger than the available memory. On the contrary, the buffer cache can actually increase overhead and reduce performance.

Direct I/O is implemented in Linux through the `O_DIRECT` flag, which is incredibly complicated and not terribly well designed [85]. In particular, it requires very specific alignment requirements for I/O operations and associated memory regions. The criteria

for performing a successful direct I/O are:

1. The I/O size must be a multiple of the disk's sector size (typically 512 B or 4 KB).
2. The I/O must begin at an offset in the file that is a multiple of the sector size.
3. The memory region to be read into or written from must have a virtual address that is a multiple of the sector size.

These requirements have several consequences for building an efficient large-scale system, some of which are implementation details, while others are significant design challenges. The first, and easiest to handle is that memory regions must be aligned to the sector size, as dictated by the third criterion above. A memory region returned by an operation such as `malloc()` or `new[]` can have an arbitrary virtual address. Since the memory region we use for reading and writing must be aligned to the sector size, we must allocate more memory than we actually need and perform the I/O starting from the first offset into the memory region that is sector aligned. In particular, we must be willing to waste $S-1$ bytes if S is the sector size. For small buffers, this overhead can be significant. In practice, buffers are often large enough where this overhead is negligible, so this is mostly an implementation detail.

A second implementation detail, imposed by the first two criteria, is that if we are ever prevented from performing a direct I/O to a file, then we will likely never be able to issue another direct I/O to that file. For example, if we are halfway done writing a file and must perform a write that is not a multiple of the sector size, we cannot issue a direct write. From this point onward, the position of writing in the file will not be a multiple of the sector size, so even if a perfectly valid direct write is created (correct size and memory address), it cannot be issued until the file position again becomes a multiple of the sector size. This can be remedied by issuing non-aligned I/O operations until all

factors again permit direct I/O. In practice, however, an application will likely abandon direct I/O after the first non-direct operation.

The largest challenge we faced, however, was correctly handling direct I/O in the presence of variably sized records. For fixed size records, I/O sizes can be made to be a multiple of both the record size and the sector size in order to permit direct IO. For example, the write chains in the original TritonSort implementation were made up of 12,800 B buffers because this is the least common multiple of 512 B (the sector size) and 100 B (the record size). When records are arbitrarily sized, this technique doesn't work.

In Themis, we solved this problem by placing functionality into the Coalescer that guaranteed that coalesced write chains were always aligned for direct I/O. Because the Coalescer already performs a memory copy, there is essentially no overhead in guaranteeing that the copied buffer is aligned. Further, the Coalescer operates only on chains of buffers, not on individual records, so fragmenting records in order to align writes is not a problem. The only issue that can occur is if multiple Coalescer threads interleave computation on buffers destined for the same file on disk. In this case, fragments of records can become interleaved, leading to data corruption. In Themis, we solved this problem by restricting the Coalescer to a single thread, which was fast enough for the cluster described in Chapter 1 and was therefore not a problem.

However, when moving to flash, we eliminate the write chaining as described in Section 2.2.2. Thus there is no Coalescer thread that can perform write alignment. The only option is to perform the write alignment inside the TupleDemux. There are two problems with this solution. The first is that the TupleDemux operates on a record-by-record basis, and aligned write buffers must permit fragments of whole records. Therefore, we had to upgrade the entire record-processing mechanism inside of Themis to permit operations over fragments of records.

The second problem with aligning writes inside of the TupleDemux presents a

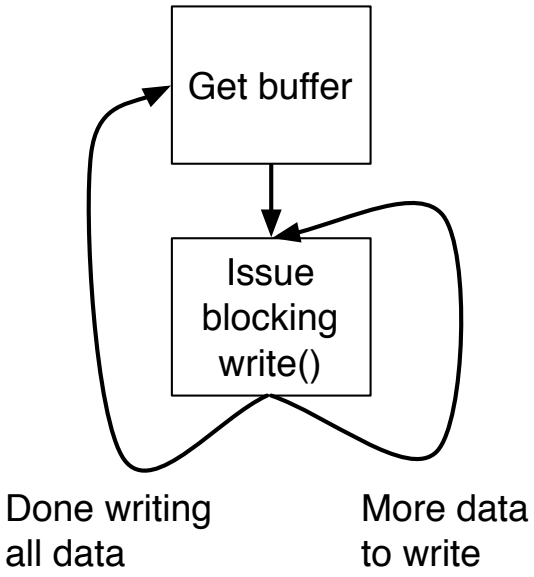
far greater challenge. Unlike the Coalescer, the TupleDemux is often not fast enough to allow for a single thread. In order to provide enough application throughput to drive the flash at full speed, we must use multiple TupleDemux threads. Therefore, fragments of records destined for the same file can be interleaved, resulting in the data corruption mentioned earlier. In order to solve this problem, we must force each thread to be responsible for mutually exclusive sets of files. Since each TupleDemux processes its records serially, this solution will prevent record fragments from interleaving. However, it requires substantial changes to the partitioning framework described in Section 1.6.4. In particular, rather than simply partitioning records by node in Mapper threads, records must be partitioned by the particular TupleDemux thread that will be serving them on the remote node. This change affects a large portion of the framework, and required significant effort to implement.

With all of these features in place (and some more to be described in the following sections), direct I/O enables the Triton10G cluster to run at roughly 1200 MB/s in the map and shuffle phase, which is nearly the maximal write bandwidth afforded by the cluster.

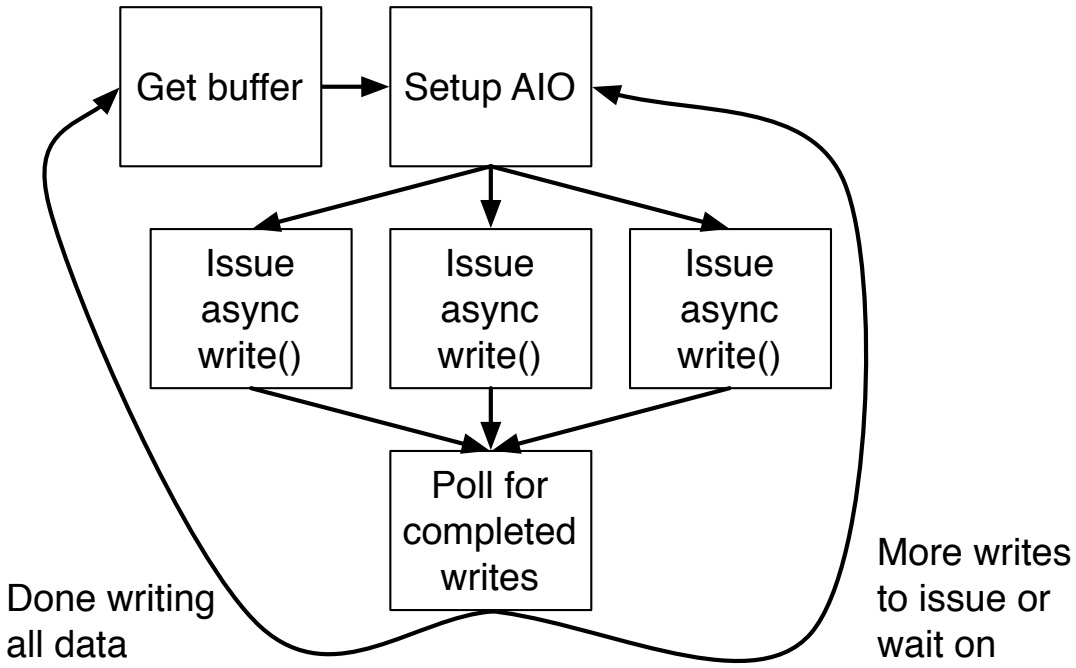
2.3.2 Asynchronous I/O

While flash-based memories have an advantage over hard disk drives in that they can handle random I/O patterns with relative ease, they tend to have their own idiosyncrasies that depend on how the flash device is manufactured. Many flash devices are built with an inherent level of parallelism. This parallelism means that the device can service multiple I/O operations simultaneously. As a consequence, it is often impossible to achieve full performance using a single thread issuing serial `write()` calls, as illustrated in Figure 2.5a.

In contrast, this single-threaded behavior the is preferred I/O pattern for a hard



(a) A representation of a single-threaded synchronous I/O implementation.



(b) A single application threaded issuing multiple parallel I/O operations using an asynchronous I/O library.

Figure 2.5. Illustration of the fundamental differences between synchronous I/O and asynchronous I/O.

disk drive. In fact, a hard disk drive's performance will suffer in any multi-threaded implementation because even if each thread issues sequential I/O operations, their interleaving will constitute a random I/O pattern. In this sense, hard disk drives and flash memories are polar opposites in terms of the style in which an application must issue its I/O calls.

A naive solution to this problem for flash is to simply run enough threads to saturate the bandwidth of the SSD. However, this solution can be inefficient due to CPU overheads, context switching, and additional memory usage.

A more efficient and more flexible solution is the use of an asynchronous I/O library. Asynchronous I/O allows a single application thread to issue many streams of I/O operations in parallel, as shown in Figure 2.5b. The implementation is left up to the library. In this work, we consider two different libraries that are popular in Linux. The first, POSIX AIO, is a portable library that implements asynchronous I/O with threads. Despite its use of threads, it can still be more efficient than an ad hoc application-level solution. The second library we consider is Linux Asynchronous I/O. This non-portable, poorly-documented, partially-implemented solution takes advantage of the internals of the Linux operating system to provide the best levels of performance.

As an illustration of the performance benefits attainable with asynchronous I/O, we consider the performance of a random write workload on the BigflashR configuration on the Gordon supercomputer. Using synchronous I/O, we are only able to write at maximum speed of around 1200 MB/s. However, the use of POSIX AIO enables the write speed to reach upwards of 1800 MB/s, which is a 50% performance improvement. In both cases, we use 16 MB write sizes. However, the asynchronous implementation issues eight writes simultaneously, which improves the performance on the flash devices. The large write size here is due to the fact that the BigflashR has 16 SSDs in a RAID0 configuration. Therefore, a 16 MB write is striped across the SSDs in 1 MB stripes. This

smaller size is more in line with the findings in Section 2.2.2.

2.3.3 Garbage Collection

Flash memory differs substantially from hard disk drives, not only in terms of performance and I/O pattern, but also in terms of data layout. A hard disk drive lays out data in tracks. The operating system hides this information from the application by simply exporting the device as if it contained a linear sequence of blocks. If a sector on the disk fails, the disk can choose to remap the sector transparently, but in normal operation, an application accessing a logical block will access a fixed sector or set of sectors on disk for all subsequent reads and writes.

Flash devices, on the other hand, continuously remap writes to their pages. This is an artifact of the physical properties of the flash memory that require a full erase operation before a page can be rewritten. These erase operations are expensive, and a typical solution is to write to a different page and transparently remap the pages so the operating system believes it has written over the existing piece of data. However, this solution quickly uses up all spare pages on the flash device. At this point, pages containing invalid data must be erased in order to be re-used. Such an operation, termed *garbage collection*, is very expensive and can substantially reduce the performance of concurrent I/O operations.

One particular cause of garbage collection in flash is the deletion of data by the user. In particular, running a large-scale data-intensive application, such as TritonSort or Themis, leaves a large amount of intermediate data that needs to be deleted between successive runs. Because garbage collection occurs in the SSD firmware, also called the *flash translation layer* or FTL, the operating system has no visibility into this operation. Therefore, a file delete operation given by a user will not trigger garbage collection until some later point in the future when subsequent write operations use up the available spare

pages. The direct consequence of this is reduced performance at unpredictable times when running data-intensive applications backed by flash storage devices.

To remedy the situation, operating systems designers and flash drive manufacturers came up with a solution called TRIM. The TRIM command allows the operating system to inform the SSD that a logical block will no longer be used, for example as the result of a file delete operation. The SSD can then react by immediately erasing deleted blocks, preventing a future garbage collection event that can impact performance. This solution is elegant in that the operating system can reduce the garbage collection penalty while still viewing the SSD as a mostly opaque black-box device. However, it requires support from the operating system, the file system, and the SSD in order to work. As an example, Linux operating system versions older than 3.2 will not support TRIM. File systems like xfs must be new enough to support TRIM via options like `discard`.

Features like RAID further complicate the usage of TRIM. In a RAID0 configuration, logical data blocks are striped across multiple physical devices. This striping can occur either at the software layer in the operating system, or at the hardware layer (e.g. a RAID controller card). This provides a significant challenge for correctly implementing the TRIM command.

In particular, the BigflashR configuration in the Gordon supercomputer does not have access to TRIM for its RAID0 array of SSDs. This is hugely problematic for performance, since garbage collection events will be frequent. Further, since these garbage collection events reduce the performance of a single SSD for the duration of the event, a stripe written to an SSD that is garbage collecting will suffer a performance loss. This performance loss will translate to the entire write operation, since all stripes must be written for the write to be complete.

The problem becomes even worse as the number of SSDs in the RAID0 array increases. Since the SSDs are performing garbage collection events independently,

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	svctm	%util
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sdb	0.00	0.00	0.00	75.40	0.00	36.20	983.26	6.74	86.46	5.56	41.92
sdd	0.00	0.00	0.00	73.00	0.00	35.00	981.92	6.95	88.99	5.24	38.28
sdc	0.00	0.00	0.00	74.60	0.00	35.80	982.82	6.14	78.22	5.43	40.52
sde	0.00	0.00	0.00	74.60	0.00	35.80	982.82	5.38	68.15	5.09	37.96
sdf	0.00	0.00	0.00	72.20	0.00	34.60	981.45	7.82	101.04	6.26	45.22
sdg	0.00	0.00	0.00	73.40	0.00	35.20	982.15	6.80	87.05	5.51	40.44
sdh	0.00	0.00	0.00	76.20	0.00	36.60	983.69	7.00	89.90	5.33	40.60
sdi	0.00	0.00	0.00	75.40	0.00	36.20	983.26	6.25	79.92	5.37	40.52
sdj	0.00	0.60	0.00	73.20	0.00	34.50	965.39	19.37	257.18	11.45	83.78
sdk	0.00	0.00	0.00	76.60	0.00	36.05	963.84	6.06	75.69	5.37	41.12
sdl	0.00	0.00	0.00	74.60	0.00	35.80	982.82	6.12	78.01	5.51	41.12
sdm	0.00	0.00	0.00	74.40	0.00	35.70	982.71	6.35	81.03	5.55	41.26
sdo	0.00	0.00	0.00	73.80	0.00	35.40	982.37	7.40	95.24	5.73	42.26
sdn	0.00	0.20	0.00	73.20	0.00	34.80	973.69	24.66	331.95	13.36	97.78
sdp	0.00	0.00	0.00	75.60	0.00	36.30	983.37	6.69	85.79	5.62	42.52
sdq	0.00	0.00	0.00	76.20	0.00	36.60	983.69	6.45	82.70	5.59	42.58
md1	0.00	0.00	0.00	1200.80	0.00	598.46	1020.69	0.00	0.00	0.00	0.00

Figure 2.6. Two of the devices in the RAID0 array, sdj and sdn, are performing garbage collection, and suffer dramatically higher latencies and queue lengths. As a result, the utilization of every other SSD in the array drops to match the performance levels of sdj and sdn.

the probability that at least one SSD is garbage collecting at any given time becomes relatively high, greatly reducing the performance of the RAID0 array. An illustration of this phenomenon occurring on the BigflashR node running Themis is shown in Figure 2.6. Here, two SSDs are garbage collecting, and the performance of the entire array suffers as a result.

To attempt to solve this problem, we asked the Gordon administrators to reconfigure one of the BigflashR nodes as Bigflash16, effectively breaking the RAID0 array and exposing each device individually. This configuration still does not support TRIM, but does increase performance significantly. In fact, the end-to-end throughput measured on Bigflash16 is 1030 MB/s in the map and shuffle phase and 1170 MB/s in the sort and reduce phase. When compared to BigflashR, which supports 890 MB/s and 820 MB/s in these phases respectively, we see a performance improvement of 15% to 42% simply from exposing SSDs individually instead of in a RAID0 array.

2.3.4 CPU Power and Frequency Scaling

Modern CPUs achieve high levels of energy efficient through frequency scaling techniques. In particular, CPU cores are set to run at slower speeds until a computationally demanding task comes along. At this point, the CPU clock speed increases to match the demands placed on the cores. This mechanism works well for traditionally CPU-bound workloads, but dramatically reduces performance of high-speed flash devices like the FusionIO ioDrive2 Duos in the Triton10G and Triton40G clusters.

In particular, we found that with frequency scaling enabled (the default), read performance to the ioDrive2 can drop as low as 374 MB/s. Because there is no computationally demanding task, the CPU speed is not raised to the levels necessary to saturate the flash device. However, by simply increasing CPU load by copying data with the `cat` program, we can trigger an increased CPU frequency that raises the read performance of the flash to 1400 MB/s. We therefore configure the system BIOS to disable frequency scaling.

2.4 Optimizations for High-Speed Networking

The techniques previously described enable high levels of performance on flash-based solid state drives. However, in order for these performance improvements to manifest in end-to-end application performance, we must also improve the performance of the network. We now describe several features necessary for improved networking performance.

2.4.1 Multiple Network Interfaces

Typical server configurations for large-scale data-intensive clusters utilize a single high performance network link. For example, a server may have a 10 Gb/s network interface card attached to appropriate networking infrastructure. If more than 10 Gb/s of

throughput is desired, the server will typically be configured with a higher performance link, such as 40 Gb/s Ethernet.

The Gordon supercomputer, however, exposes its network links as two separate interfaces in Linux, `ib0` and `ib1`. We measured the network performance with `netperf` and found that 6 Gb/s of TCP traffic was possible using just one interface, but the use of both interfaces improved speeds to 10 Gb/s. In an effort to drive more application throughput, we therefore augment the Themis framework to support multiple network interfaces. This requires, in particular, storing multiple network addresses for each node in the cluster, and multiplexing data transfers between them. While mostly a matter of implementation, this change required a substantial amount of effort.

2.4.2 IPoIB: IP over InfiniBand

TritonSort and Themis are written using TCP over IP sockets. The cluster described in Chapter 1, the Triton10G cluster, and the Triton40G cluster all have Ethernet network interconnects, which work well for this particular use case. The Gordon supercomputer, however, uses an InfiniBand interconnect. This network fabric uses a *verbs* protocol that is substantially different from IP. However, rewriting Themis to make full use of InfiniBand verbs is complicated. We therefore choose to run IPoIB, which is IP over InfiniBand, in order to use the same socket interface already in place.

IPoIB has significant performance issues due to the interface mismatch between IP and InfiniBand. In particular, even though the 4x QDR is capable of 40 Gb/s of throughput, or 32 Gb/s of goodput, TCP flows using traditional IP sockets run much slower. We found that, despite netperf achieving 10 Gb/s using, Themis's performance was much worse.

Fortunately, a solution exists to this problem. Sockets Direct Protocol [16] is a more efficient implementation of the socket interface using InfiniBand. In terms of usage,

the user simply needs to inform the linker that Sockets Direct Protocol will be used, and the socket interface will be transparently replaced with a much faster implementation. The performance of netperf under this implementation increased from 10 Gb/s to 21.5 Gb/s, a 115% improvement. Using this library, the network performance of Themis on Gordon increased to the point where the network was no longer the bottleneck. In particular, the map and shuffle phase was able to run at 8 Gb/s, or roughly the speed we could write to the flash devices at the time of measurement.

2.4.3 Multi-Threaded Networking

As alluded to Section 1.8.2, TritonSort's networking subsystem was initially implemented with multiple threads. Due to scaling concerns, we moved to a single-threaded implementation. A single thread works for speeds around 10 Gb/s. However, when we move to 40 Gb/s networking, a single thread cannot service the sockets fast enough and we essentially become CPU-bound by the speed of the Sender or Receiver thread.

To address this issue, we upgraded Themis with the ability to support an arbitrary number of Sender and Receiver threads. Sockets are multiplexed across the threads in such a way that each thread handles an equal number of network connections. This allows the us to remove the CPU-boundedness from our networking subsystem.

We can evaluate the performance of multiple Sender and Receiver threads even on a single node. We measure the performance of one of the Triton40G nodes sending data to itself, which does not actually involve the network. We observe that a single Sender thread and a single Receiver thread can only achieve a throughput of 24 Gb/s even when the network is not involved. However, three Sender threads and 2 Receiver threads can achieve 40 Gb/s. Therefore we conclude that multi-threaded networking is key to the success of high speed networks like 40 Gb/s Ethernet.

2.5 Non Uniform Memory Access (NUMA)

As described earlier, the Triton10G cluster does not have enough storage bandwidth to match its networking bandwidth. The Triton40G cluster was an attempt to solve this by hosting four FusionIO flash drives per server instead of one. However, finding a server configuration that has enough PCI slots, CPU power, and memory to service four flash devices and one high-speed network card is difficult.

As of the time we purchased the Triton40G cluster, the only configuration available that suited our needs was a multi-socket Sandy Bridge configuration. This configuration runs four Intel Xeon E5-4610 processors in a non-uniform memory access (NUMA) configuration. In this configuration, there are four NUMA domains, one per processor. Memory accesses within the same domain are fast, but there is a penalty for accessing memory from a different NUMA domain. In NUMA terms, the *distance* between components in different NUMA domains on these servers is twice as large as the distance between components in the same domain.

In addition to just CPU and memory, PCI-Express devices can also belong to NUMA domains. As we will see, this presents a significant challenge for achieving high performance with FusionIO flash devices or 40 Gb/s network cards.

2.5.1 Interrupt Request Handling

When an I/O device has data for the CPU, it sends an interrupt request, or IRQ. This IRQ is handled by a program called an interrupt request handler. In a system with uniform memory access, the choice of which core to run the IRQ handler on is irrelevant. However, in a NUMA architecture, this core assignment for IRQ handlers is significant. In fact, without a proper assignment of IRQ handlers, it is impossible to run the FusionIO ioDrive2 devices on the Triton40G cluster at full performance.

In order to correctly assign IRQ handlers, it is necessary to know the hardware topology of each server. Program suites like `hwloc` can be used to determine the rough assignment of PCI-Express devices to NUMA domains. With this information, we have a chance of correctly assigning IRQ handlers. However, the usable PCI-Express slots on the Triton40G servers are not evenly distributed between the four NUMA domains. In particular, we observed that two of the ioDrive2 Duos and the regular ioDrive2 were all placed in a single NUMA domain, complicating the assignment of IRQ handlers to cores within that domain.

We found the ioDrive2 Duos to be particularly finicky with regards to the IRQ assignment. In particular, the write performance for a device varies between 880 MB/s and 1204 MB/s depending on how IRQ assignments are configured.

The performance of the 40 Gb/s NIC is also affected by IRQ settings. We found that in our servers, the 40 Gb/s NIC was attached to the second NUMA domain. Placing the IRQs on the CPU responsible for this domain yields a transfer rate of about 38 Gb/s for three TCP flows. In contrast, placing the IRQs on the wrong NUMA domain dropped the transfer rate to 29 Gb/s to 33.5 Gb/s, depending on which domain was chosen.

2.5.2 Core Assignment

In addition to assigning interrupt request handlers to the appropriate NUMA domains, it is also necessary to run application threads on cores within specific NUMA domains. For example, if a thread is issuing `read()` or `write()` calls to a specific device, it ought to also run on the same NUMA domain to prevent performance loss. Similarly, a thread that is transferring data over the 40 Gb/s network interface should run on the same NUMA domain as the network card.

In the networking example above, even if we place the IRQs on the correct NUMA domain, the performance will still suffer if the application thread runs on the

wrong NUMA domain. For example, while 38 Gb/s is possible with IRQs set to the correct domain, an application thread that runs outside that domain can see performance as low as 30 Gb/s, which is a reduction of more than 20%.

The situation is complicated further when many high-performance devices exist on a single server. As mentioned earlier, the devices in the Triton40G cluster are not evenly distributed between the NUMA domains. This means that placing all application threads and IRQs on the correct NUMA domain for the device can actually lead to resource contention that reduces performance. In particular, application processing threads that are not issuing I/O operations should be placed on other NUMA domains to prevent contention with the I/O threads.

2.6 Application-Level Benchmarks

When debugging performance issues on high-speed I/O devices, it is often useful to look at benchmark results rather than end-to-end application performance. However, translating the results from programs like `dd` or `netperf` back to application performance is not always possible due to complicated usage patterns for these devices. We therefore choose to implement a set of application-level benchmark tools that re-use as much of the components of Themis as possible. These tools have been invaluable in discovering the optimizations described in this chapter. We now describe each tool in detail.

2.6.1 The DiskBench Microbenchmark

DiskBench, shown in Figure 2.7, is a pipelined application that reuses many of the components of Themis MapReduce (Section 1.6). The goal of DiskBench is to isolate the storage subsystem of the *map and shuffle* phase. As such, data records are read from disk, but no `map()` function is applied. Records are randomly assigned to partitions on the same node, and are written back to local disk without involving a network shuffle.

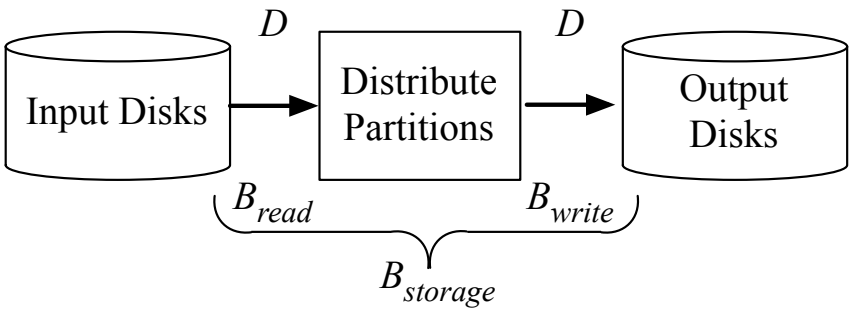


Figure 2.7. The *DiskBench* storage microbenchmark runs locally on a single node without involving the network.

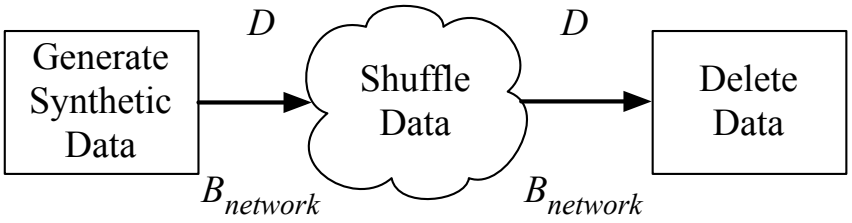


Figure 2.8. The *NetBench* network microbenchmark measures network scalability and performance using synthetic input data.

2.6.2 The NetBench Microbenchmark

Similar to DiskBench, NetBench (Figure 2.8) is a pipeline-oriented application derived from Themis. Following the analogy, NetBench aims to isolate the network subsystem from the *map and shuffle* phase. Synthetic data records are generated in-memory and shuffled over the network to remote nodes, which simply delete the data records. NetBench operates entirely in memory and does not touch local disk.

2.7 Evaluation

Now that we have described all the optimizations necessary to get good performance on next generation clusters, we turn our attention to small-scale and large-scale

evaluations of sorting on these clusters. We note that due to constraints on resource availability, not all of the optimizations were enabled in each experiment given in this section. Nevertheless, these evaluations represent a significant milestone in the evolution of Themis. As we will show, performance has dramatically improved from the original implementation described in Chapter 1.

2.7.1 Gordon Small-Scale Experiments

We now give a small-scale performance evaluation of the server configurations available on the Gordon supercomputer. Our results are summarized in Figure 2.9.

BigflashR

Gordon exposes only 4 of its compute nodes as Bigflash nodes. Three of these are configured with the RAID0 array of BigflashR, and the other is configured with separate SSDs in the Bigflash16 configuration.

First, we measure the performance of the three BigflashR nodes. We find that the map and shuffle phase of sort runs at a per-server rate of 1073 MB/s and the sort and reduce phase runs at a rate of 982 MB/s. In both phases, Themis is storage-bound. We note that these rates are the best rates observed. Performance varies randomly due to the garbage collection in the flash described in Section 2.3.3.

Bigflash16

As alluded to earlier in this chapter, the RAID0 array used in the BigflashR configuration suffers greater performance losses from garbage collection than if the SSDs were not configured in a RAID array. Unfortunately, there is only a single Bigflash16 node to evaluate. We evaluate the performance of the Bigflash16 node by running a sort with all four nodes, the three BigflashR nodes and the one Bigflash16 node. We note that in this experiment, the read performance of the Bigflash16 node was 1030 MB/s

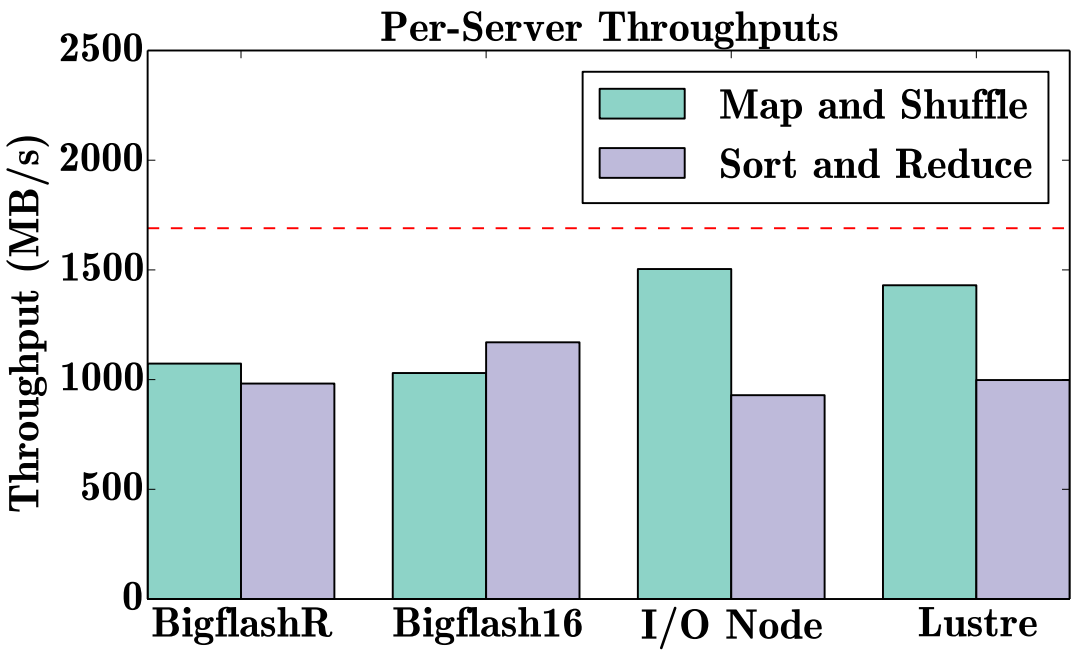


Figure 2.9. Small-scale performance evaluation of the offerings available on the Gordon supercomputer. The maximum performance afforded by the flash devices is denoted with a dashed line.

and 1170 MB/s in the map and shuffle phase and sort and reduce phase respectively. In contrast, the BigflashR servers performed at 890 MB/s and 820 MB/s in each phase.

I/O Node

After running experiments on the Bigflash nodes for some time, we determined that the best performance levels we could reach were far below what the flash devices ought to be capable of. Comparing the performance numbers above to the specifications given in Table 2.1 reveals significant performance loss. In particular, the 3.4 GB/s of write bandwidth ought to yield approximately 1.7 GB/s of application throughput using the flash for both reading and writing. However, the closest we can get using the Bigflash16 node is about 1.2 GB/s, which is roughly 70% of the expected performance.

In order to remedy this situation, we run directly on the I/O nodes, rather than

mounting the flash remotely through the compute nodes. Running computation on the I/O node breaks the abstraction in the supercomputer and incurs significant administrative costs. However, the promise of good performance was enough for the system administrators to give us one I/O node for experimentation.

Running a sort operation on the single I/O node yields a application throughput of 1504 MB/s in the map and shuffle phase and 929 MB/s in the sort and reduce phase. The rate of the first phase in particular is substantially higher in the I/O node and approaches the maximum performance levels available with the flash devices. The second phase, however, is still fairly slow. This is due to the reduced computational power in the I/O node. As shown in Table 2.1, the I/O nodes have only 12 CPU cores, while the compute nodes have 16. In fact, Themis is CPU-bound in both phases on the I/O node, indicating that while the locally-attached flash is indeed faster, the I/O nodes do not have enough computational power to match the increased I/O speed.

Lustre File System

For comparison, we also measure the performance of the I/O node accessing the Lustre parallel file system. This system is disk-based, rather than flash-based and supports a large aggregate bandwidth. In fact, we measured the map and shuffle phase running at 1430 MB/s and the sort and reduce phase at 998 MB/s. These numbers are very similar to running directly on the flash. In particular, we observe that this configuration is also CPU-bound in both phases.

2.7.2 Gordon Large-Scale Experiments

We now turn our attention to running a large-scale sort operation on Gordon. In particular, we wish to sort 100 TB of data as described in Section 1.1.1. Given the small-scale experiments above, the best option is to run on all 64 I/O nodes on Gordon.

Together, these nodes have 307.2 TB of flash storage, which is just enough to hold the input, intermediate and output data sets for the 100 TB sort.

Unfortunately, due to resource provisioning issues, we were not able to obtain access to all 64 I/O nodes. Instead, we were only given access to 37 of the I/O nodes. This smaller set of nodes does not have enough storage capacity to hold the data for the sort operation even if we delete intermediate data as we go. We therefore were unable to run a 100 TB sort on the flash devices on Gordon.

Instead, we attempt to run a 100 TB sort using the disk-based Lustre parallel file system measured earlier. Based on small-scale tests, this configuration ought to have roughly the same levels of performance as running on flash. Unfortunately, the performance of the Lustre file system is unreliable due to the fact that it is a shared resource between many supercomputers. We noticed that read time varied dramatically by file, with some files taking much longer to read than the average. We were not able to complete the 100 TB sort operation before our access to the I/O nodes was revoked.

2.7.3 Triton10G

Next, we measure the performance of the Triton10G cluster. This cluster is much smaller than Gordon and does not have the capacity to run a 100 TB sort. However, the fact that we own and administer the cluster means that we do not have to worry about contention from other users.

First, we run a small-scale sort operation on the Triton10G cluster without any of the upgrades described in this chapter in order to get baseline. As mentioned in Section 2.3, the map and shuffle phase runs at 630 MB/s and the sort and reduce phase runs at 708 MB/s, which are speeds roughly comparable to Themis running on the disk-based cluster described in Chapter 1.

After implementing the upgrades in this chapter, we observe 1193 MB/s in the

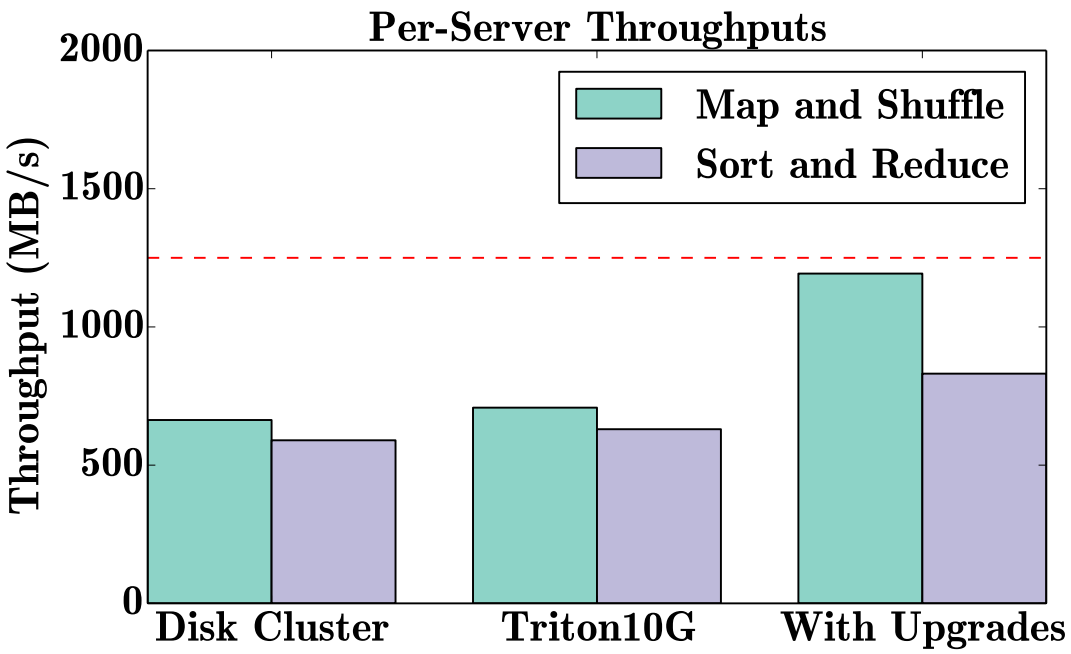


Figure 2.10. Performance evaluation of the Triton10G cluster with and without upgrades for next generation hardware. For reference, we also show the performance of a 500 GB sort on the disk-based cluster described in Chapter 1.

map and shuffle phase and 831 MB/s in the sort and reduce phase. The map and shuffle phase is bound by the speed of writing to the flash device, and the sort and reduce phase is CPU-bound. In particular, we have improved the performance of Themis by 89% in the first phase and 17% in the second phase. The results are summarized in Figure 2.10.

We note that the CPU-boundedness problem experienced by the sort and reduce phase is inherent to the Triton10G cluster. If we compare the Triton10G specifications in Table 2.3 to the cluster that Themis was originally designed on, given in Table 1.2, we see that we are aiming to roughly double the performance of the framework with half as many CPU cores. It is therefore not unreasonable to expect the Triton10G cluster to be CPU-bound. In fact, this is precisely the reason why the Triton40G cluster has so many cores. After our experience with Triton10G, we wanted to make sure the system does not become CPU-bound due to limited CPU cores.

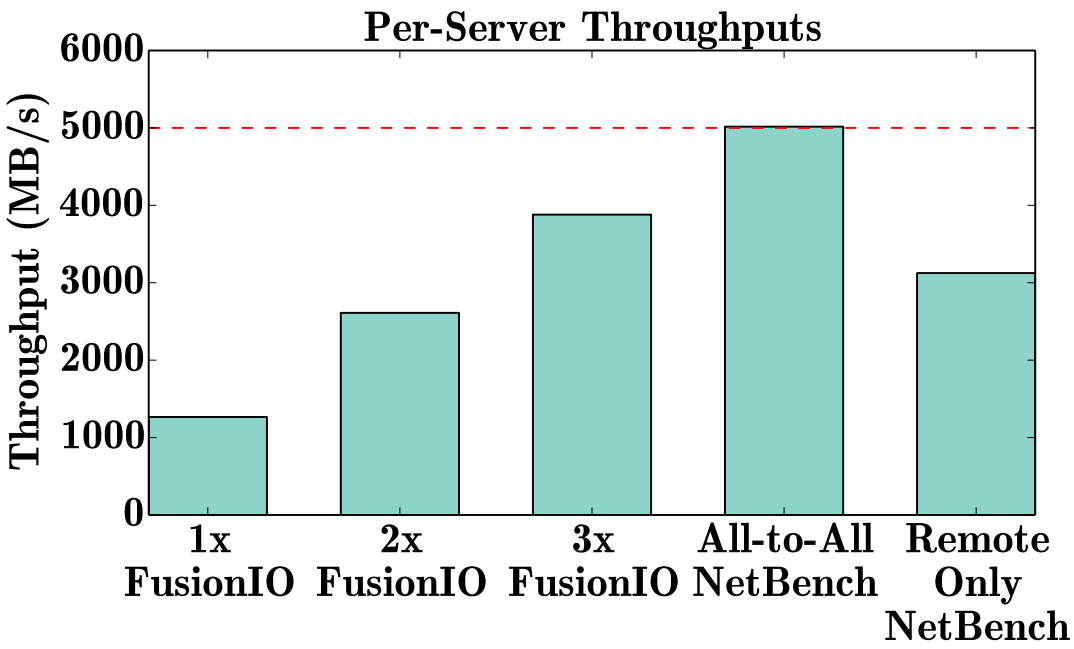


Figure 2.11. Performance evaluation of the Triton40G cluster. We show the performance of DiskBench using one, two, and three FusionIO ioDrive2 Duo devices. We also show the performance of NetBench measured both in all-to-all and remote-only modes.

2.7.4 Triton40G

Finally, we measure the performance of the Triton40G cluster. When we purchased this cluster, many of the optimizations described in this chapter were already implemented. We therefore expected to see very high levels of performance. However, the best performance levels we could observe were 1600 MB/s in the map and shuffle phase and 1957 MB/s in the sort and reduce phase. Here, the map and shuffle phase is network-bound, while the sort and reduce phase is CPU-bound.

Given the specifications of this cluster, we expect to see performance levels of 40 Gb/s, or 5000 MB/s. Because we are so far way from optimal, we choose to use a different strategy for optimizing performance. Rather than continue to add application-level features, we decide to benchmark the individual I/O subsystems within Themis. We use the DiskBench and NetBench tools described in Section 2.6.

The performance of DiskBench and NetBench as measured on the Triton40G cluster are given in Figure 2.11. We note in particular that DiskBench can achieve the full bandwidth available on three FusionIO ioDrive2 Duo devices. In terms of networking, NetBench can achieve 40 Gb/s when configured in the all-to-all mode of the shuffle phase. However, we note that in this case, half of the data is directed back to the sending node without actually involving the network hardware. When we measure the performance of only sending data to the remote node, performance drops to 25 Gb/s, indicating that there are some networking issues with running Themis on Triton40G.

2.8 Lessons

We end this chapter with a discussion of the lessons we learned evaluating next generation clusters with Themis. The results in the previous section are bittersweet. They paint a picture of next generation clusters as resources that have the potential to dramatically improve the performance and per-server efficiency of large-scale data processing applications. However, this potential is very difficult to realize. In particular, we were neither able to run a 100 TB sort on the Gordon supercomputer, nor we were able to achieve 40 Gb/s of per-server sorting throughput on Triton40G. Even the Triton10G cluster, which is the simplest of the three, cannot realize 10 Gb/s of application throughput due to lack of CPU cores in the second phase of the sort.

However, it is important to consider the context in which we give these results. Recall from Section 2.1 that we are working with hardware that is 5-10 years ahead of its time. From this perspective, the results in this chapter represent a huge success. In fact we were able to accomplish quite a lot with hardware that is not commonplace even as of this writing.

Further, the optimizations and insights in this chapter turn out to be absolutely critical for achieving good performance on existing high-speed hardware configurations,

as we will see in Chapter 3. By discovering all of these necessary optimizations early, we set ourselves up for the high quality work that will be presented towards the end of this dissertation.

In addition, the work in this chapter confirms the key insight of Chapter 1. Namely, in order to build high-performance systems that are highly efficient at handling I/O, it is absolutely necessary to consider the properties of the underlying hardware. Whether the storage devices are disks, commodity flash-based SSDs, or high-speed PCI-Express flash devices, the application must be aware of the properties of the devices so that it can achieve peak levels of storage performance. Similarly, the properties of the network must also be known in order to achieve good performance in an application that shuffles data like MapReduce. Even the server architecture itself must be carefully considered, as we have shown in Section 2.5.

As a final note, we point out that ultimately a lack of available resources prevented us from running a successful 100 TB using the work in this chapter. In particular, we could not get access to enough I/O nodes on Gordon to run such a sort. There are a couple solutions to this problem. The first is to build our own cluster, much like we did in Chapter 1. However, this can be very costly. A better solution is to find an existing large-scale infrastructure that will have fewer resource provisioning issues than the supercomputer. Fortunately, such a solution – cloud computing – exists and is the subject of the next chapter of this dissertation.

2.9 Acknowledgements

Chapter 2 includes material that is submitted for publication as “Achieving Cost-efficient, Data-intensive Computing in the Cloud.” Conley, Michael; Vahdat, Amin; Porter, George. The dissertation author was the primary author of this paper.

Chapter 3

Cost-Efficient Data-Intensive Computing in Amazon Web Services

We now analyze cost-efficient data-intensive computing in the cloud. Throughout this chapter, we focus on Amazon Web Services (AWS) as an exemplary public cloud provider. We use Themis, described in Chapter 1, with the upgrades for newer hardware technologies, described in Chapter 2, as our evaluation framework.

We show that through various optimizations and design decisions, it is possible to run high-performance, cost-efficient computation in the public cloud. As a result of these optimizations, we set several world records in the 2014 sorting competition, mirroring the success in the 2010 and 2011 competitions described in Chapter 1.

3.1 Introduction

Cloud providers such as Amazon Web Services (AWS) [10], Google Cloud Platform [30] and Microsoft Azure [11] offer nearly instantaneous access to configurable compute and storage resources that can grow and shrink in response to application demands, making them ideal for supporting large-scale data processing tasks. Yet supporting the demands of modern Internet sites requires not just raw scalability, but also cost- and resource-efficient operation: it is critical to minimize the resource budget

necessary to complete a particular amount of work, or conversely to maximize the amount of work possible given a particular resource budget.

Minimizing cloud costs requires choosing a particular combination of resources tailored to a given application and workload. There have been several measurement studies of the performance of cloud resources [53, 60, 90], and several efforts aimed at automatically selecting a configuration of cloud resources suited to a given workload [39, 44, 95]. This is no easy task, as the diversity within public cloud platforms has rapidly accelerated over the past half decade. For example, as of this writing, Amazon offers 47 different types of VMs, differing in the number of virtual CPU cores, the amount of memory, the type and number of local storage devices, the availability of GPU processors, and the available bandwidth to other VMs in the cluster. The above-mentioned provisioning tools have shown promise, especially for resources such as CPU time and memory space, which can be precisely divided across tenant VMs located on the same hypervisor. On the other hand, shared resources, such as network bandwidth and storage, have proven to be a much bigger challenge [32, 92].

Recently, providers have begun introducing I/O-virtualization at the storage and network layers to enhance performance. Cloud nodes increasingly have access to high-speed flash-based solid state drives (SSDs), which can be virtualized by the hypervisor across multiple guest VMs. These virtualized SSDs can provide high throughput and thousands of IOPS to multiple tenants. Likewise, the data center network fabric is also virtualized, enabling guest VMs to access a "slice" of resources from the network through technologies such as SR-IOV [82]. These virtualized networks enable throughputs and latencies previously unattainable with then-available network technologies. The result is that VMs have access to significantly higher bandwidths than before, e.g., 10 Gb/s VM-to-VM.

These advances in virtualized I/O have the potential to improve efficiency, thereby

reducing the number of resources a user needs. Because users pay only for the resources they use, greater efficiency leads to lower costs. However, choosing the right set of resources in this environment is harder than ever, given that the configuration space is now even larger than before. Further, as the size of the cluster increases, overall cluster utilization and efficiency can drop, requiring more VMs to meet performance targets and driving up overall cost [18]. Thus an understanding of the scaling behavior of virtualized cloud network and storage resources is key to achieving cost-efficiency in any large deployment.

In this chapter, we present a systematic measurement of the scaling properties of recently-introduced virtualized network and storage resources within the AWS public cloud. Our aim is to determine the optimal price points for configuring clusters for data-intensive applications, specifically applications that are I/O-bound. We deploy Themis [69], our in-house implementation of MapReduce, as a case study of I/O-bound data processing applications under a variety of efficiency and data durability assumptions. We give a large-scale evaluation of our methodology using jobs drawn from the annual 100 TB "GraySort" sorting competition [81].

We find that despite newly-introduced I/O virtualization functionality, AWS clusters still have scalability limitations, leading to larger cluster sizes than would be otherwise predicted from the performance of small numbers of nodes. We further find that the choice of cloud resources at scale differs significantly from predicted configurations measured at smaller scale. Thus the actual deployment cost shifts dramatically from estimates based on small-scale tests.

We further show that, by measuring performance at scale, it is possible to provision highly efficient clusters withing the AWS public cloud. As a demonstration of this point, we deploy Themis MapReduce to an AWS cluster consisting of 100s of nodes and 100s of terabytes of virtualized SSD storage, and set three new world records in the

Table 3.1. Four example EC2 instance types with various CPU, memory, storage, and network capabilities. Some types use flash devices(⋆) rather than disk.

Type	vCPU	RAM	Storage	Net.
`m1.small`	1	1.7 GB	160 GB	Low
`m3.xlarge`	4	15 GB	80 GB⋆	High
`hs1.8xlarge`	16	117 GB	49 TB	10G
`i2.8xlarge`	32	244 GB	6.4 TB⋆	10G

GraySort competition at very low cost. We compare our sorting results to other record winners, and find several commonalities between the winning entries, further supporting the results of this work.

The contributions described in this chapter are:

1. A systematic methodology for measuring the I/O capabilities of high-performance VMs in the public cloud via application-level benchmarks.
2. A measurement of the current AWS offerings at scale, focusing on virtualized I/O.
3. A large-scale evaluation of cost-efficient sorting on 100s of nodes and 100s of terabytes of data informed by this measurement methodology.
4. Three new world records in sorting speed and cost-efficiency based on our evaluation results.

3.2 Background

We now present a brief overview of Amazon Web Services (AWS)'s I/O resources, and then describe our application model.

3.2.1 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is a cloud computing service that provides access to on-demand VMs, termed *instances*, at an hourly cost. There are many types of

instances available, each with a particular mixture of virtual CPU cores (vCPU), memory, local storage, and network bandwidth. Table 3.1 lists a few examples.

VM instances are located in *availability zones*, which are placed across a variety of geographically distributed regions. VMs within the same region are engineered to provide low-latency and high-bandwidth network access to each other. The cost of individual VMs varies by instance type, as well as over time, as new hardware is deployed within AWS. In this work, we only consider "on-demand" pricing, representing the cost to reserve and keep instances during a given job. Finally, although the cloud offers the abstraction of unlimited computing and storage resources, in reality the number of resources in a given availability zone is limited. This complicates cluster provisioning because the most economical cluster for a given job might not be available when the user needs it. In our experience, launching even 100 VMs of a specific type required two weeks of back and forth communication with engineers within Amazon. Even then, we were only permitted to allocate the virtual machines in a short window of a few hours.

3.2.2 Virtualized I/O

Recent advances in I/O-virtualization technology have made the cloud an attractive platform for data-intensive computing. Here we discuss three types of virtualized I/O available in the cloud.

Virtualized Storage

In 2012, Amazon introduced the first EC2 VM with solid-state storage devices. Prior to this, all VM types available on EC2 ran either on disk or persistent network-attached storage. Over the next two years, more and more VMs with SSDs became available. By mid 2014, Amazon began highlighting its SSD offerings, relegating the disk-based VMs to the "Previous Generation" of VMs. Other cloud providers have

followed suit in the race for newer and faster storage technologies. Google recently added a local SSD offering to its Compute Engine [33] cloud. Microsoft Azure's new G-series VMs include large amounts of local SSD storage [11].

Because offered bandwidth is so high and access times are so low, significant effort is required to support these devices in a virtualized environment at full speed. If the hypervisor spends too much time processing I/O on shared devices, performance will suffer. Recent virtualization technologies, such as Single Root I/O Virtualization (SR-IOV), enable providers to expose a high-speed I/O device as many smaller, virtualized devices [82]. With SR-IOV, the hypervisor is out of the data path, enabling faster guest VM access to these devices.

Virtualized Network

Today, high-speed networks are common in public cloud platforms. EC2 has offered VMs connected to a 10 Gb/s network as early as 2010, although these VMs were primarily targeted at scientific cluster computing. More recently, 10 Gb/s networks have been rolled out to VM types targeting more general workloads. While achieving maximum network performance is difficult on dedicated hardware, virtualization adds another level of complexity that needs to be addressed for achieving efficiency. As in the case of storage, technologies such as SR-IOV can reduce virtualization overheads and make the most of the high speed network. In a shared environment, SR-IOV can be used to slice the 10 Gb/s interface so each VM receives a portion of the bandwidth. In the case of a single guest VM, eliminating overhead makes 10 Gb/s transfer speeds possible.

Amazon offers SR-IOV through a feature called *enhanced networking*. Though not all VMs support enhanced networking, a large portion of the newer VMs can access the feature. These include not only the VMs that support 10 Gb/s, but also their smaller counterparts, which are likely carved up from larger instance types using SR-IOV to

efficiently share a single 10 Gb/s NIC.

Enhanced networking also enables VMs to launch in a *placement group*. Placement groups instruct EC2 to provision VMs strategically in the network to increase bisection bandwidth. Given that oversubscription is common in large data center networks [37], placement groups play an important role in delivering high performance to the user.

Network-Attached Storage

A third type of virtualized I/O, network-attached storage, is a common way to implement persistent storage in cloud environments. The local storage devices described above are typically erased after a VM shuts down or migrates. To store persistent data, users are directed to separate storage services, such as Amazon Simple Storage Service (S3) or Amazon Elastic Block Store (EBS). These services are accessed remotely by a variety of interfaces. For example, S3 supports a RESTful API and can be accessed via HTTP, while EBS is exposed as a standard block device. When evaluating persistent storage in this work, we consider EBS because its interface is similar to a local storage device, thereby supporting unmodified applications. To access EBS, a user simply attaches a volume to a running instance. Volumes can be created with near arbitrary size and IOPS requirements, backed either by disks or SSDs.

Achieving high performance on persistent, network-attached storage brings its own complexities. On the back-end, the storage service must be provisioned with enough storage devices to suit users' needs and also have an efficient way of carving them up into volumes. Typically these storage services are also replicated, resulting in additional complexity. On the client's side, an application wants to issue an optimal pattern of I/O's while simultaneously knowing nothing about storage system's internal characteristics or preferred I/O patterns. Finally, congestion in the network or interference from co-located

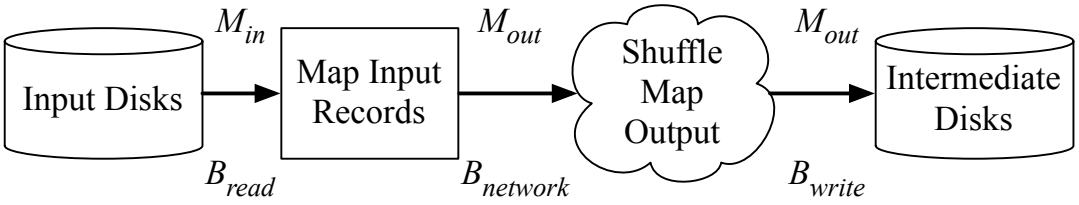


Figure 3.1. Themis phase 1: `map()` and shuffle.

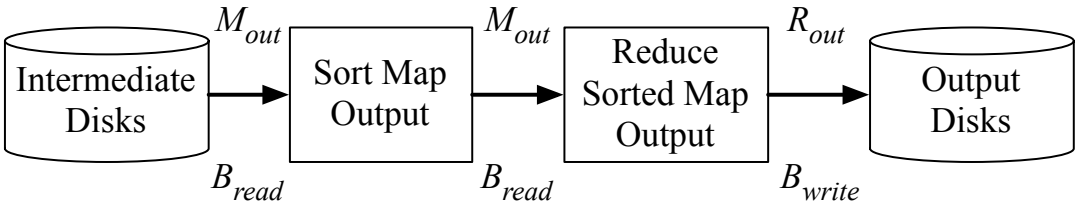


Figure 3.2. Themis phase 2: sort and `reduce()`.

VMs can reduce the performance observed by certain VMs in an unpredictable way.

3.2.3 Application Models

In this work, we focus on the performance of I/O-bound jobs and deploy Themis, our in-house MapReduce [69, 71]. Themis implements MapReduce as a two pass, pipelined algorithm. In its first *map and shuffle* pass (Figure 3.1), Themis reads input data from disk into small, in-memory buffers. It then applies the `map()` function to records in these buffers, and the resulting map output, or *intermediate*, data is divided into *partitions*. Unlike traditional MapReduce systems, which write intermediate data to local disk, Themis streams intermediate data buffers over the network to remote nodes before writing to partition files on the remote node's local disks. This implementation eschews traditional task-level fault tolerance in favor of improved I/O performance.

In the second *sort and reduce* pass (Figure 3.2), Themis reads entire intermediate partitions from local disk into memory. It then sorts these partitions and applies the `reduce()` function. Finally, the resulting records are written to output partition files on

local disk. In the rare event that partitions do not fit in memory, a separate mechanism handles these overly large partitions.

We now model the performance of Themis MapReduce under several assumptions about I/O efficiency and data durability.

2-IO

Because Themis eschews traditional task-level fault tolerance, it exhibits the 2-IO property [69], which states that each record is read from and written to storage devices exactly twice. In this work, we consider data sorting as our motivating application. For external sorting, Themis achieves the theoretical minimum number of I/O operations [2]. This property not only makes Themis efficient, but it also yields a very simple computational model. When we restrict our focus to I/O-bound applications, the **processing time** of the *map and shuffle* phase can be modeled as:

$$T_1 = \max\left(\frac{M_{in}}{B_{read}}, \frac{M_{out}}{B_{network}}, \frac{M_{out}}{B_{write}}\right) \tag{3.1}$$

where M_{in} and M_{out} represent the per-node map input and output data sizes, and B_{read}, B_{write}, and $B_{network}$ represent the per-node storage and network bandwidths. For clarity, we have labeled these variables in Figures 3.1 and 3.2. In the particular case of sorting, map input and output are the same, and if we ensure that storage read and write bandwidths are the same, we are left with:

$$T_1 = \max\left(\frac{D}{B_{storage}}, \frac{D}{B_{network}}\right) \tag{3.2}$$

where D is data size to be sorted per node. Next we compute the processing time of *sort and reduce* phase. Because this phase involves only local computation, storage is the only I/O bottleneck:

$$T_2 = \max\left(\frac{M_{out}}{B_{read}}, \frac{R_{out}}{B_{write}}\right) \tag{3.3}$$

where R_{out} is the reduce output data size. Again in the case of sort, this is equal to D, the per-node data size, so the processing time is:

$$T_2 = \frac{D}{B_{storage}} \tag{3.4}$$

In practice, it may not be the case that read and write bandwidths are equal, in which case we have:

$$B_{storage} = \min\left(B_{read}, B_{write}\right) \tag{3.5}$$

Therefore the final processing time of the sort is:

$$T = T_1 + T_2 = \max\left(\frac{D}{B_{storage}}, \frac{D}{B_{network}}\right) + \frac{D}{B_{storage}} \tag{3.6}$$

Finally, we account for the VM's hourly cost C_{hourly} to compute the total dollar cost of the sort:

$$C = C_{hourly}\left[\max\left(\frac{D}{B_{storage}}, \frac{D}{B_{network}}\right) + \frac{D}{B_{storage}}\right] \tag{3.7}$$

Application-Level Replication

The 2-IO model discussed above represents the upper-bound of cost-efficiency and performance for I/O-bound applications. In practice, storing exactly one copy of the data dramatically reduces durability. We now consider the case where the application makes a remote replica of each output file for improved data durability.

We augment the *sort and reduce* phase with output replication as shown in

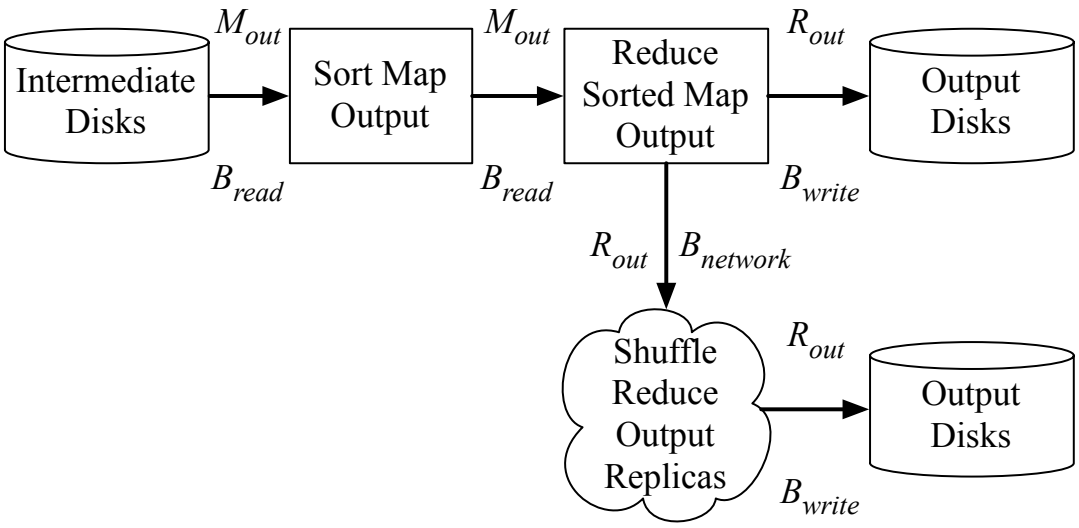


Figure 3.3. Sort and `reduce()` with Application-Level Replication.

Figure 3.3. In addition to writing output partitions to local output disks, the system creates a replica of each output file on a remote node's local output disks. This incurs an extra network transfer and disk write for each output partition file. This online replication affects the total processing time of the *sort and reduce* phase:

$$T_2 = \max\left(\frac{M_{out}}{B_{read}}, \frac{R_{out}}{B_{network}}, \frac{2R_{out}}{B_{write}}\right) \tag{3.8}$$

In the case of sort, this becomes:

$$T_2 = \max\left(\frac{D}{B_{read}}, \frac{D}{B_{network}}, \frac{2D}{B_{write}}\right) \tag{3.9}$$

Notice there is now an asymmetry in the storage bandwidth requirements between the *map and shuffle* phase (Equation 3.2) and the *sort and reduce* phase (Equation 3.9). This asymmetry will necessitate storage configuration changes, as we will see in Section 3.4.2.

Infrastructure-Level Replication

Implementing Application-Level Replication as described in Section 3.2.3 adds significant complexity and cost. Cloud providers typically offer infrastructural services to reduce the burden on application developers.

To illustrate the use of Infrastructure-Level Replication, we consider running Themis MapReduce on Amazon EC2 using the EBS storage service described in Section 3.2.2 for input and output data, and local disks for intermediate data only. The time for the *map and shuffle* phase becomes:

$$T_1 = \max\left(\frac{M_{in}}{B_{readEBS}}, \frac{M_{out}}{B_{network}}, \frac{M_{out}}{B_{write}}\right) \tag{3.10}$$

Similarly, the time for *sort and reduce* is:

$$T_2 = \max\left(\frac{M_{out}}{B_{read}}, \frac{R_{out}}{B_{writeEBS}}\right) \tag{3.11}$$

We consider the performance and cost implications of these three models in the following sections. Section 3.3 thoroughly explores the 2-IO model, while Section 3.4 describes a large-scale evaluation of all three models.

3.3 Profiling AWS Storage and Networking

We now turn our attention to choosing a cluster configuration on EC2 for I/O-bound applications. As we will show, it is not simply enough to know the VM specifications. The scaling behavior of each VM must be taken into account.

To this end, we design a series of experiments to estimate the performance of I/O-bound jobs on EC2. First, we measure the per-VM bandwidth of local storage devices (Section 3.3.2). This approximates the performance of instance types where

the network is not the bottleneck ($B_{network} = \infty$ in our models). Next, we measure the network performance of each instance type (Section 3.3.3). Together, these metrics give a performance estimate that accounts for either bottleneck, but assumes that network performance scales perfectly. Then, we measure the actual scaling behavior of the network at the largest cluster sizes that we can reasonably allocate to get a more realistic performance estimate. Finally, we combine the above results with the published hourly costs of each instance type to select the most cost-effective instance type for carrying out a large-scale 100 TB sort job under the 2-IO model described in Section 3.2.3.

The data we use in this analysis comes from a pair of custom-built microbenchmark tools: (1) *DiskBench*, which measures the overall throughput of the storage subsystem within a single node, and (2) *NetBench*, which measures network performance by synthetically generating data without involving local storage. These tools are described in detail in Section 2.6.

3.3.1 Measurement Limitations

A common concern when conducting measurements of the public cloud is variance. Resource sharing between customers, either co-located on the same machine or utilizing the same network, increases variance and makes measurement more difficult. Getting a completely fair assessment of the performance of the cloud is complicated by diurnal workload patterns that necessitate measuring at different times of day. Jobs launched during the work week cause different days of the week to experience different performance levels as well. Less-frequent, periodic jobs may even lead to changes based on week of the month or month of the year.

In addition to user-created variance, the infrastructure of the cloud itself is constantly changing, meaning that any attempted measurement is just a snapshot of the cloud in its current state. For example, in the time between the experiments in this work and

the current writing, Amazon has added 10 new instance types to EC2, all of which can alter the performance of the shared network that connects them. Variance can even exist between different data centers belonging to the same provider. Different data centers may contain I/O devices with different performance characteristics, as Schad et al. [76] have shown.

While we acknowledge the amount of variance that exists in the public cloud, we admit that our ability to quantify variance is limited. Despite partial support from Amazon's educational grant program, the experiments described in this work totaled more than $50,000 in AWS costs, and so we were not able to continue studying AWS in enough detail to account for these forms of variance.

Furthermore, in many of the more interesting cases, it is often not possible to allocate a large number of on-demand VMs. The large-scale evaluations in Section 3.4 were only possible after weeks of back-and-forth communication with AWS engineers. When we were finally able to allocate the VMs, we were instructed to decommission them after only a few hours, proving further measurement impossible. For these two reasons, a comprehensive study of variance in the cloud is not presented in this work.

3.3.2 Local Storage Microbenchmarks

We begin our measurement study by profiling the local storage available on each EC2 VM type with DiskBench, a tool described in Section 2.6.1. Because local storage devices are often faster than network-attached storage, these measurements are typically an upper-bound on storage performance. We revisit the choice of local versus remote storage in Section 3.3.4.

In the measurements that follow, we configure DiskBench to use half of a node's local disks for reading and the other half for writing when more than one device is available. This configuration is typically ideal for local storage devices, and is in fact the

configuration used in our earlier experience with high speed sorting [71]. As a result, the bandwidths reported by DiskBench measure a simultaneous read/write workload, and in many cases are approximately half of the bandwidth available in read-only or write-only workloads.

Experimental Design

We begin by running DiskBench on each of the VM types offered by AWS. For each type, we instantiate two to three VMs in the `us-east-1a` availability zone, and we run DiskBench on each of those instances three times. From these six to nine data points, we compute the average per-node storage bandwidth, $B_{storage}$, measured in megabytes per second (MB/s). We run DiskBench on multiple instances to account for natural variance in performance between VMs.

Analysis

The results of DiskBench are shown in Figure 3.4. We report the mean storage bandwidth across the measured data points, as well as the offered per-VM storage capacity. Recall that storage bandwidth as measured by DiskBench is a read/write workload that approximates half of the read-only or write-only bandwidth of the devices. We have used vertical bars to group VM instance types into regions based on the number of instances needed to sort 100 TB of input data; the rightmost region represents instance types needing fewer than 100 instances. The middle region represents types needing between 100 and 1,000 instances. Finally, the leftmost region represents instance types needing more than 1,000 instances. We highlight these regions because provisioning a large number of instances is not always possible. For example, we found that even with the help of Amazon's engineers, we were only able to allocate at most 186 instances of `i2.8xlarge` in a single availability zone. Furthermore, as we will show, network performance can degrade significantly with larger clusters.

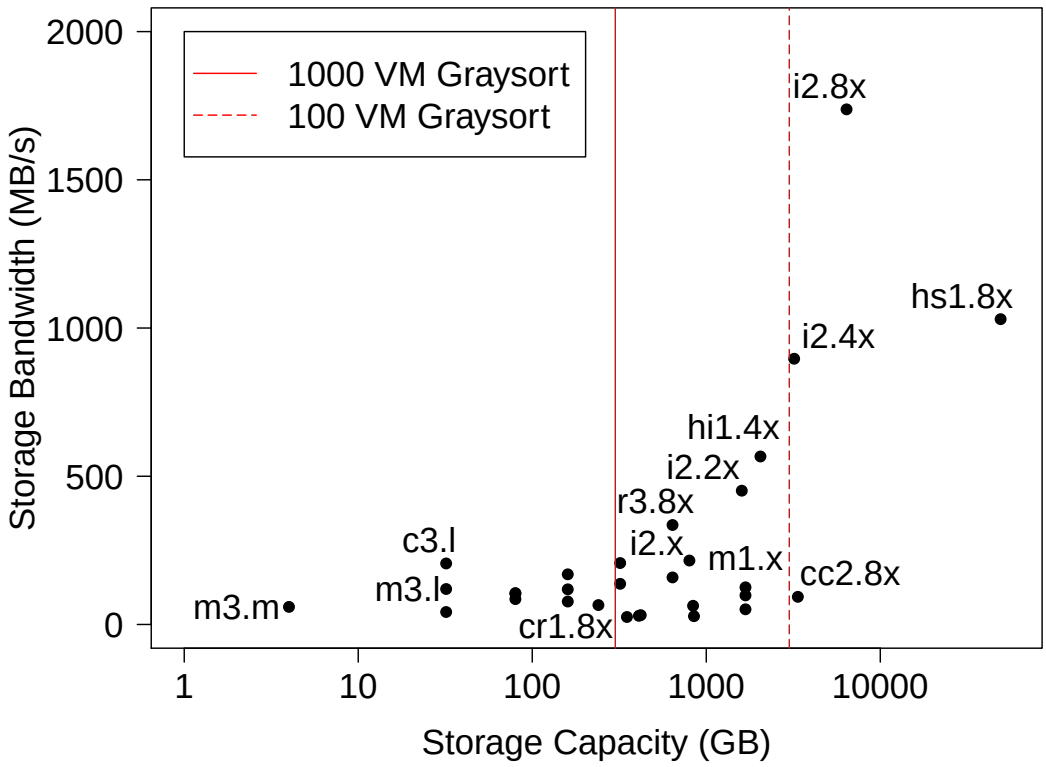


Figure 3.4. Storage performance of EC2 VM reported by *DiskBench*. Vertical lines cluster VM types into those requiring more than 100 or 1,000 instances to sort 100 TB.

In Figure 3.4 we have labeled some of the more interesting instance types. Many of these are on the upper right-hand side of the figure and represent a candidate set of instance types which deliver both high storage performance and host enough local storage to meet the capacity requirements of a 100 TB sort with a small cluster. The highest performing instance type in the sub-100 VM region is `i2.8xlarge`, which contains eight 800 GB SSDs and offers 1.7 GB/s of simultaneous read/write bandwidth as measured by DiskBench. The `i2.4xlarge` instance type has half the number of SSDs, with half as much storage bandwidth as a result. Another interesting instance type is `hs1.8xlarge`, which provides the highest density of storage using HDDs instead of SSDs. The `hs1.8xlarge` instance type includes 24 local HDDs and supports 1.06 GB/s

Table 3.2. Estimated dollar cost of sorting 100 TB on a subset of EC2 instance types based solely on local storage performance.

Instance	Min. nodes required for 100TB sort	Cost	
		Sort ($)	Hourly ($/hr)
c3.large	9,375	28	0.105
m3.large	9,375	65	0.14
m3.medium	75,000	66	0.07
m1.xlarge	179	155	0.35
i2.4xlarge	94	211	3.41
i2.8xlarge	47	218	6.82
hs1.8xlarge	7	248	4.60
cr1.8xlarge	1,250	2,966	3.50

of read/write bandwidth. Because of its high storage density, only seven instances are needed to meet the capacity needs of a 100 TB sort operation.

Estimating the dollar cost of sorting: We next use the results of DiskBench in conjunction with the listed AWS hourly cost to predict the total dollar cost of running a 100 TB 2-IO sort using Themis. Here, we consider only local storage performance ($B_{network} = \infty$), and apply the results from Figure 3.4 to Equation 3.7 to estimate the total cost of sorting 100 TB.

A subset of these results, shown in Table 3.2, is presented in ascending order using this sort cost metric to rank instance types. Note that each configuration has its own storage capacity limitations, and to highlight the impact this capacity limitation has on overall resource utilization, we also include the number of nodes necessary to meet the capacity requirements of a 100 TB sort. Specifically, the cluster must be capable of storing 300 TB between the input, intermediate, and output data sets. However, it is important to note that under the assumption of perfect scalability, the total dollar cost is independent of the number of VMs used. To see this, consider that using twice as many VMs cuts job execution time in half, resulting in exactly the same dollar cost.

From Table 3.2, we see that `c3.large` is the most economical, with a per-sort

cost of $28. However, each VM only has 32GB of storage, so 9,375 instances are required to hold the necessary 300 TB of data. Next are the `m3.large` and `m3.medium` instance types, with a sort cost of approximately $65. Again, scaling to meet capacity requirements is a significant challenge. In fact, it is not until the `m1` instance types that clusters of $O(100)$ nodes will suffice. The first instance types with $O(10)$ node cluster sizes are the `i2` types, which are built with arrays of SSDs. A 100 TB sort can be completed with just 47 `i2.8xlarge` instances at a cost of $218. For reference, the most expensive instance type is `cr1.8xlarge`, a memory-optimized 32-core instance type with two 120GB SSDs, on which a 100 TB sort would cost $2,966, a factor of over 100x more expensive than the cheapest instance type. It is worth noting that two instance types might have hourly costs that are an order of magnitude apart, but the total cost to the user may be very similar, e.g., `m1.xlarge` and `i2.4xlarge`.

Summary : Measuring VM storage bandwidth provides great insight into the total cost of a large-scale data-intensive application. Many high-performance VM configurations can deliver reasonable costs using a small number of nodes.

3.3.3 Network Microbenchmarks

Next, we measure the performance and scalability of the AWS networking infrastructure. We focus on the subset of instance types that have relatively high performance and high storage capacity as measured in Section 3.3.2. We perform our measurements using the NetBench tool described in Section 2.6.2.

Experimental Design

We perform two experiments to measure the AWS networking infrastructure. The first experiment determines the baseline network bandwidth of each instance type. For each VM type, we allocate a cluster of two nodes in the `us-east-1a` availability

zone. On each of these clusters, we run NetBench three times. From these three data points, we compute the average observed network bandwidth, $B_{network}$, which we report in the unconventional unit of megabytes per second (MB/s) for easy comparison with the results of DiskBench. This measurement represents the ideal scaling behavior of the network. When available, we enable the enhanced networking feature and allocate nodes in a single placement group, and we use two parallel TCP connections between nodes to maximize the bandwidth of the high speed VMs.

The second experiment assesses the scaling behavior of the network in a candidate set of VM types. For each type, we allocate increasingly large clusters in the `us-east-1a` availability zone in the following way. We first create a cluster of size two. We then create a cluster of size four by allocating two new VMs and adding them to the existing cluster. Next we create a cluster of size eight by adding four new VMs. We repeat this process until we reach the end of the experiment. For each cluster size, we run NetBench once, and measure the all-to-all network bandwidth as observed by the slowest node to complete the benchmark.

We note that the largest measured cluster size varies by instance type. In many cases, limits imposed by AWS prevented larger study. For some of the more expensive VMs, we cap the maximum cluster size due to limited funds. We do not use placement groups in this experiment because doing so alters the natural scaling behavior of network and limits cluster sizes. Placement groups also work best when all VMs launch at the same time. This launch pattern is neither representative of elastically scaling applications, nor is it applicable to our experiment setup. Additionally, we use a single TCP connection between VMs because using multiple TCP connections reduces performance at larger cluster sizes, and we are ultimately interested in the performance at scale.

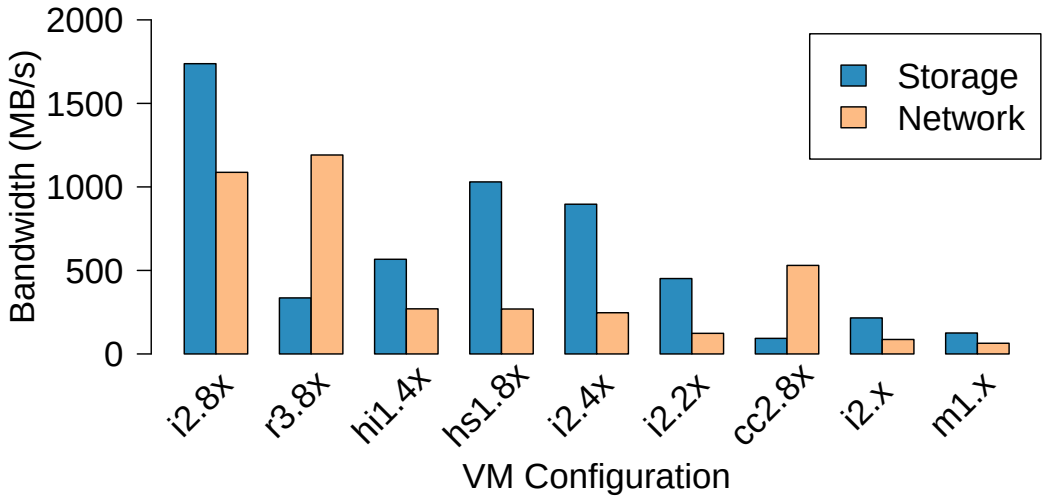


Figure 3.5. Comparison between storage and network performance of each VM instance type.

Analysis

The ideal network performance of a select subset of instance types measured in the first experiment is shown in Figure 3.5. For comparison, we also show the storage performance measured in Section 3.3.2. For many instance types, the storage and network bandwidths are mismatched. Equation 3.2 (Section 3.2.3) suggests that we want equal amounts of storage and network bandwidth for the *map and shuffle* phase of sort, but this is often not achieved. For example, the network bandwidth of `i2.8xlarge` is only 63% of its measured storage bandwidth. This mismatch reduces the end-to-end performance of an application that must use both storage and network I/O, resulting in underutilized resources.

Figure 3.6 shows the network scaling behavior measured in the second experiment. We present the data as a fraction of the baseline bandwidth measured in the first experiment. This comparison is not perfect because the experiments were run on different sets of VMs on different days during a two week period and at different times of day.

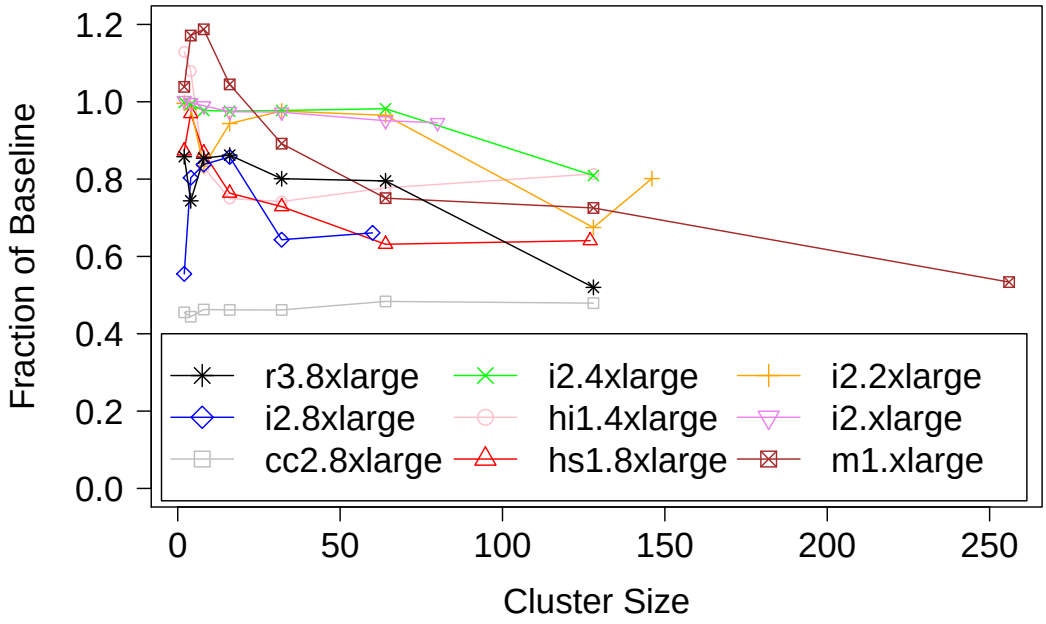


Figure 3.6. Network performance scalability displayed as a fraction of the baseline network performance given in Figure 3.5.

This perhaps explains how `m1.xlarge` and `hi1.4xlarge` reach speeds that are 20% faster than the baseline at small cluster sizes.

However, the main takeaway is that performance degrades significantly as more nodes are added to the cluster. In eight of the nine VM types measured, performance drops below 80% of baseline during the experiment. One instance type, `cc2.8xlarge`, shows consistently poor performance. We speculate this type resides in a highly congested portion of the network and can only achieve high performance when placement groups are enabled.

The dollar cost of sorting revisited: Finally, we use the results of DiskBench and NetBench to predict the total monetary cost of running a 100 TB 2-IO sort operation on each of the VM instance types. We apply the measured bandwidths to Equation 3.7 (Section 3.2.3) to determine the total dollar cost.

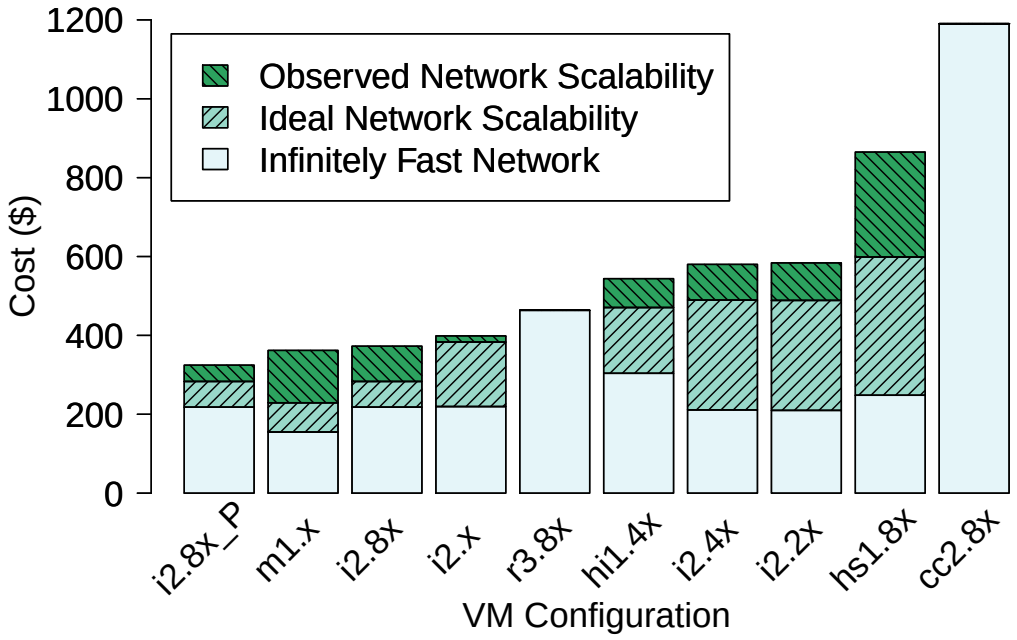


Figure 3.7. Estimated cost of sorting 100 TB on a subset of EC2 VM types, under various network performance assumptions.

This overall cost prediction is shown in Figure 3.7. For the selected instance types, we show (1) the overall cost assuming that the network is *not* the bottleneck, (2) the cost assuming that the offered network bandwidth scales in an ideal manner, and (3) the cost based on the observed scale-out networking performance. The results show that the lowest-cost instance type for sort is `m1.xlarge`, at $362 per sort followed closely by `i2.8xlarge` and `i2.xlarge`. Interestingly, while the ideal network scalability cost of `i2.8xlarge` is larger than `m1.xlarge`, `i2.8xlarge` has better actual network scaling properties, resulting in very similar overall dollar costs. However, the `i2.8xlarge` instance type supports placement groups, which if employed actually result in a lower overall cost than `m1.xlarge`. We represent this configuration as `i2.8x_P`, with an estimated cost of $325, which is $37 cheaper than `m1.xlarge`.

Summary : Networking performance, particularly at scale, must be accounted for when estimating cost. Poor scaling performance can significantly drive up costs. Better network isolation, e.g. placement groups, can substantially reduce costs. In the case of sort, network isolation results in a savings of $37, or about 10%.

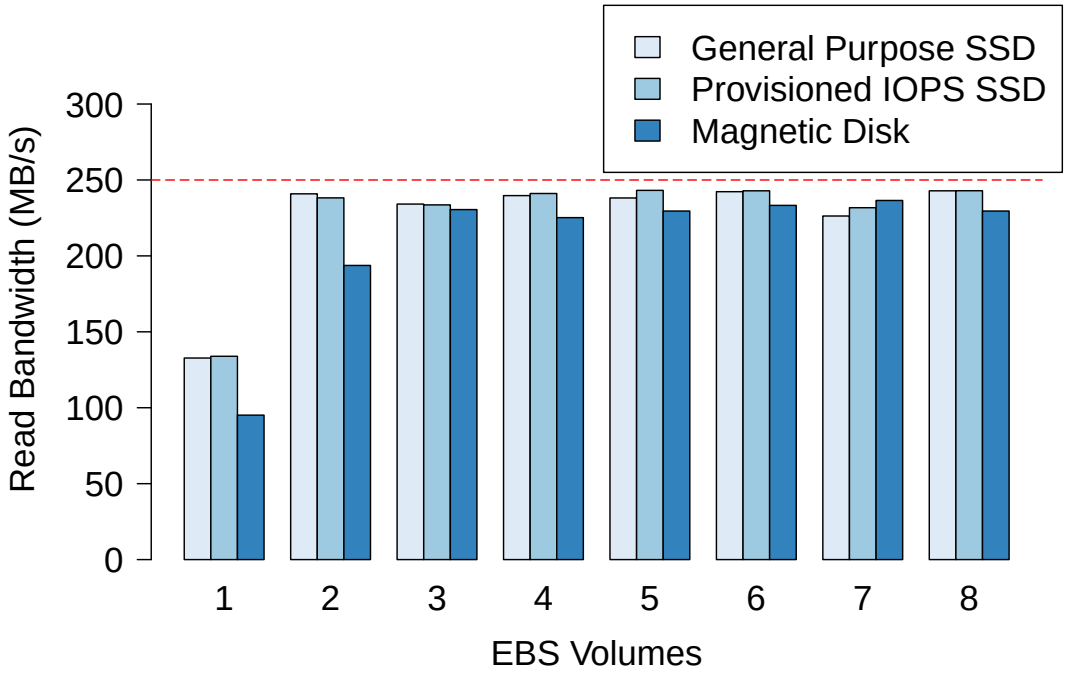
3.3.4 Persistent Storage Microbenchmarks

We now turn our attention to persistent network-attached storage. While local storage devices typically have higher performance, many cloud deployments will want input and output data sets to persist across VM resets and migrations. We now consider the performance properties of Elastic Block Store (EBS), a persistent network-attached storage service offered by AWS.

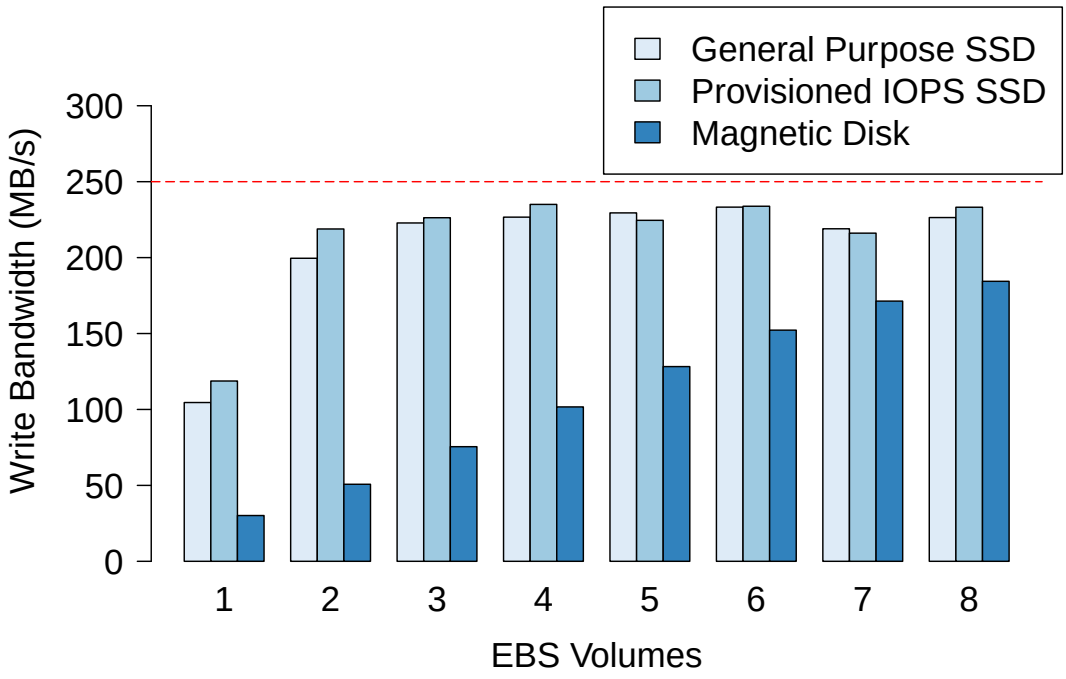
Experimental Design

To measure the performance of EBS, we allocate three `i2.4xlarge` instances in `us-east-1a` with the enhanced networking and EBS-optimization features enabled. At the time of the experiment, `i2.4xlarge` was one of the few VM types supporting an EBS throughput of up to 250 MB/s. As of this writing, Amazon offers new classes of instance types, `c4` and `d4`, with speeds of up to 500 MB/s. EBS offers three types of storage volumes: magnetic disk, general purpose SSDs, and IOPS-provisioned SSDs. For each type, we create and attach eight 215 GB EBS volumes to each of the three `i2.4xlarge` instances. We then run DiskBench, and vary the number of EBS volumes used.

We configure DiskBench to run in read-only and write-only modes, but not in the read/write mode described in Section 2.6.1. This more closely resembles an actual EBS-backed application, which will read input data from persistent storage, process it for some period of time using local per-VM storage, and then write output data back to



(a) EBS read performance



(b) EBS write performance.

Figure 3.8. EBS performance observed by `i2.4xlarge`. The maximum advertised performance is shown with a dashed line.

persistent storage. This usage pattern directly corresponds to the Infrastructure-Level Replication model described in Section 3.2.3.

We run each combination of EBS volume type, number of EBS volumes, and DiskBench mode three times on each of the three nodes to get an average bandwidth measurement.

Analysis

Figure 3.8a shows the read-only DiskBench results, and Figure 3.8b shows the write-only results. There are four key takeaways. First, a single EBS volume cannot saturate the link between the VM and EBS. Bandwidth increases as more volumes are added up to the 250 MB/s limit. Second, near maximal read performance can be achieved using as few as three volumes of any type. Third, near maximal write performance can be achieved using three or four SSD-based volumes. Finally, the magnetic disk volume type cannot achieve maximal write performance with even eight volumes.

These results are promising in that EBS-optimized instances can actually achieve the maximal read or write bandwidth using the SSD volume types. However, these maximum speeds are quite low relative to the performance of local, per-VM storage. For example, the `i2.4xlarge` instance measured in Section 3.3.2 is capable of nearly 900 MB/s of read/write bandwidth to its local storage devices, as shown in Figure 3.4. As such, EBS bandwidth is likely to be a bottleneck in the Infrastructure-Level Replication model (Equations 3.10 and 3.11) and will shift the cost analysis quite a bit from that derived in Section 3.3.3.

Summary : Persistent storage systems built from SSDs can deliver reasonable levels of storage performance. However, local, per-VM storage provides far higher levels of performance, so persistent storage will likely be a bottleneck.

3.4 Evaluation

Thus far we have measured the I/O performance and scalability of several cloud offerings in AWS in the context of the 2-IO model described in Section 3.2.3. We now present a large-scale evaluation of 2-IO, as well as the other models presented in Section 3.2, Application-Level Replication and Infrastructure-Level Replication. We consider the problem of sorting 100 TB and measure the performance and cost in each case. Each of these evaluations corresponds to one of a larger number of established large-scale sorting benchmarks [81], and thus represents a realistic problem that one might want to solve using the public cloud.

3.4.1 2-IO

We evaluate the performance and cost of 2-IO by sorting a 100 TB data set that consists of one trillion 100-byte key-value pairs. Each pair consists of a 10-byte key and a 90-byte value. Keys are uniformly distributed across the space of 256^{10} possible keys.

Experiment Setup: We allocate 178 on-demand instances of the `i2.8xlarge` VM instance type. All instances belong to a single placement group in the `us-east-1a` availability zone. We use local, per-VM SSDs for input, intermediate, and output data sets.

Before running the sort application, we run the DiskBench and NetBench microbenchmarks on the cluster to get a baseline performance measurement, and also to decommission VMs with faulty or slow hardware. DiskBench reports read/write storage bandwidth at 1515 MB/s for the slowest VM, which is 87% of the bandwidth measured in Section 3.3.2. NetBench yields a network bandwidth of 879 MB/s which is 81% of the ideal bandwidth measured in Section 3.3.3. We note that this experiment was conducted on a different day than the microbenchmarks described in Section 3.3, and therefore may

Table 3.3. Our 100 TB Indy GraySort entry. Past and current record holders are shown for comparison.

System Name	Cluster Size	Sort Speed (TB/min)	Per-node Speed (MB/s)	Total Cost ($)
Themis	178	6.76	633	299.45
Hadoop	2,100	1.42	11	?
Baidu	982	8.38	142	?

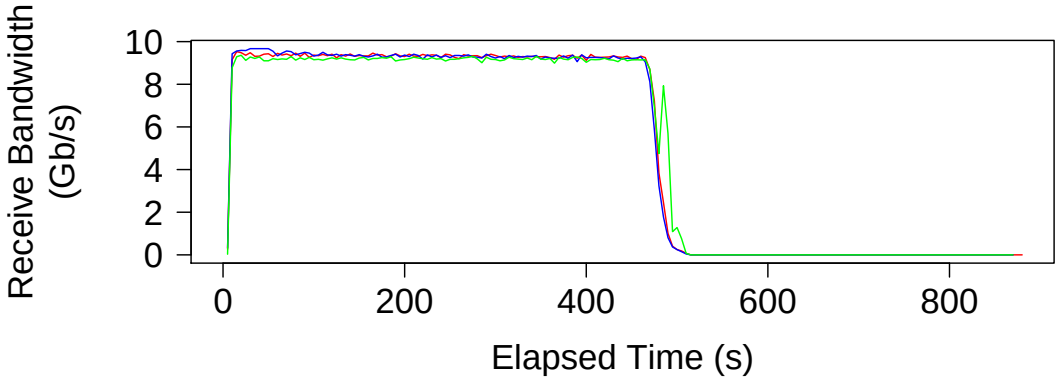
have somewhat different performance characteristics.

As in Section 2.6.1 we configure Themis to use four of the eight local SSDs for input and output files, and the remaining four SSDs for intermediate files.

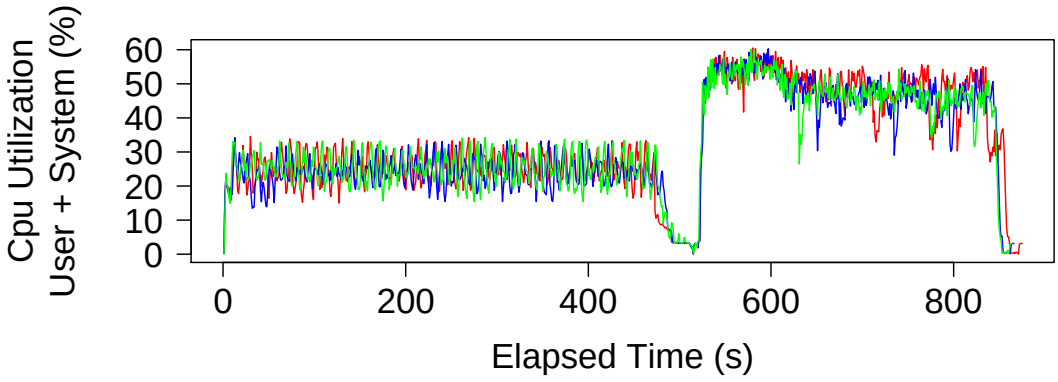
Results: The 100 TB 2-IO sort completes in 888 seconds and requires $299.45. To better understand the bottlenecks and limitations of this particular job, we collect system-level performance metrics using `sar`, `iostat`, and `vnstat` [83, 89]. Using these measurements, we find that during the approximately 500 seconds required to complete the map and shuffle phase, Themis is network-bound. Figure 3.9a shows the network utilization for three randomly chosen servers as a function of time. The 10 Gb/s network is almost fully utilized, and as a result, the CPU and SSDs are only lightly utilized, as shown in Figures 3.9b and 3.9c.

The sort and reduce phase, which begins immediately after the map and shuffle phase completes, is I/O-bound by the local SSDs. Because no network transfer occurs in this phase, Themis can fully utilize the available storage bandwidth, and Figure 3.9c shows that the disk write bandwidth approaches the limitations of the underlying hardware. Multiple sorting threads allow CPU usage to increase considerably. However the overall system does not become CPU-limited, as illustrated in Figure 3.9b.

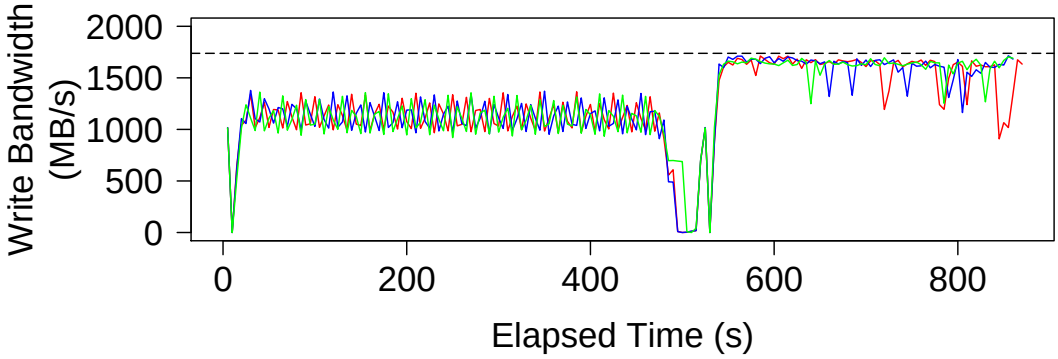
Because the sort job is I/O-limited, the final cost ($299.45) closely resembles the estimated cost given Section 3.3.3 for `i2.8xlarge` with placement groups ($325). We



(a) Network receive throughput.



(b) CPU utilization.



(c) Disk write bandwidth. The maximum hardware speed is denoted by a dashed line.

Figure 3.9. System-level metrics collected on 3 of the 178 nodes running the 100 TB 2-IO sort, which shifts from being network-limited to being SSD-limited at $t \approx 500s$.

conclude that the methodology in Section 3.3 can predict the cost of I/O-bound jobs with reasonable accuracy.

Sort Benchmark: While the analysis thus far has been focused on cost-efficiency, raw performance is also a highly-desired feature. Our 100 TB 2-IO sort conforms to the guidelines of the Indy GraySort 100 TB sort benchmark [81], and achieves an overall throughput of 6.76 TB/min. Our sort is nearly five times faster than the prior year's Indy GraySort record [35] (see Table 3.3), while still costing less than $300.

We attribute this result to both the methodology in this work, and also to our Themis MapReduce framework. It is important, however, to note that it is not simply our codebase that yields high performance. In fact, our Indy GraySort speed was surpassed by Baidu [45] by more than 20% using a system derived from TritonSort [70, 71], which also exhibits 2-IO. Thus the 2-IO model of computation has powerful implications for performance as well as cost-efficiency.

3.4.2 Application-Level Replication

Next we evaluate Application-Level Replication on the same 100 TB data set described in Section 3.4.1. We run a variant of Themis that supports output replication as illustrated in Figure 3.3. This particular configuration conforms to the Daytona GraySort benchmark specification [81].

Experiment Setup: We allocate 186 on-demand instances of `i2.8xlarge`. As before, we launch all instances in a single placement group. However, due to insufficient capacity in `us-east-1a`, we use the `us-east-1d` availability zone.

As alluded to in Section 3.2.3, the storage requirement asymmetry in Application-Level Replication necessitates a slight change in the configuration of Themis. Here we use five of the eight SSDs for input and output files and the remaining three for

Table 3.4. 100 TB Daytona GraySort results.

System Name	Cluster Size	Sort Speed (TB/min)	Per-node Speed (MB/s)	Total Cost ($)
Themis	186	4.35	390	485.56
Spark	207	4.27	344	551.36
Hadoop	2,100	1.42	11	?

intermediate files. This configuration more evenly balances the storage and network requirements of the MapReduce job.

Results: Sorting 100 TB with Application-Level Replication requires 1,378 seconds and results in a total cost of $485.56. While a comparison between this result and the 2-IO result in Section 3.4.1 is not completely fair due to different sets of resources used in different availability zones on different dates, it is nevertheless interesting to note that the improved data durability increases the cost of the sort from $299.45 measured in Section 3.4.1 by $186.11.

Sort Benchmark: The performance of our Application-Level Replication surpassed the prior year's record-holder by more than 3x, as seen in Table 3.4, setting the 100 TB Daytona GraySort record. Apache Spark, run by Databricks, submitted a benchmark result [100, 99, 98] that was slightly slower than ours, although our results are close enough to be considered a tie. However, our system is slightly more resource-efficient, resulting in a cost savings of $66, or about 12%.

We note that both results for this sort benchmark use the `i2.8xlarge` VM type on Amazon EC2, despite there being no requirement to use EC2 at all. While we cannot speculate as to what methodology Apache Spark used to determine the use of `i2.8xlarge`, we can say that the fact that both teams submitted records using this virtual machine validates the conclusions drawn in Section 3.3.

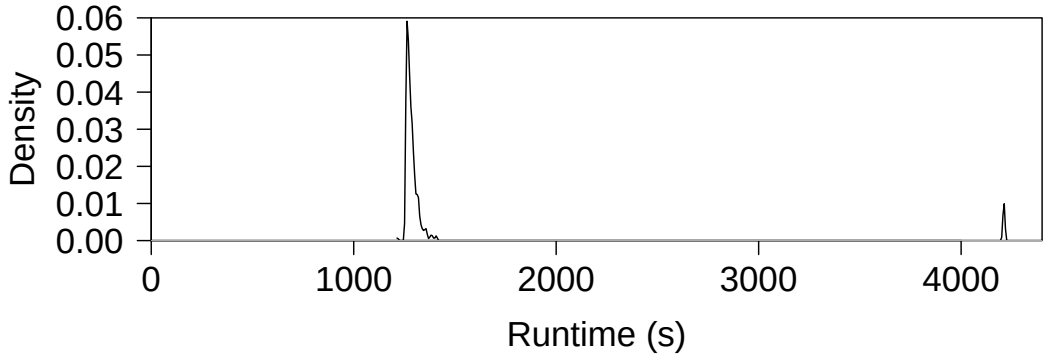


Figure 3.10. Bimodal elapsed times of reading 100 TB from EBS as seen by a cluster of 326 `c3.4xlarge` VMs.

3.4.3 Infrastructure-Level Replication

Finally, we evaluate Infrastructure-Level replication on the same 100 TB data set. This time we run the 2-IO implementation of Themis but replace the input and output storage devices with EBS volumes, which provide the desired replication properties. This configuration meets the specifications for the Indy and Daytona CloudSort benchmarks [81]. Incidentally, CloudSort directly measures cost, rather than absolute performance as measured in the GraySort benchmarks, and is more in-line with the spirit of this work.

Preliminary Results: While the analysis in Section 3.3 suggests `i2.8xlarge` for sorting on local disks, the use of EBS changes the cost analysis substantially. Our measurements indicate the cheapest VM type is `c3.4xlarge`. Therefore we allocate 326 `c3.4xlarge` VMs in a single placement group in the `us-east-1a` availability zone and attach to each four 161 GB general purpose SSD EBS volumes. Unfortunately, this configuration experiences significant variance in read performance. Figure 3.10 shows a probability distribution function of runtimes across the 1,304 EBS volumes experienced when reading 100 TB from EBS. Approximately 95% of the nodes complete

Table 3.5. 100 TB Indy and Daytona CloudSort results.

System Name	Cluster Size	Sort Time (s)	Per-node Speed (MB/s)	Total Cost ($)
Themis	330	2981	102	450.84

in under 1,400 seconds, but the remaining nodes take three times longer. This long-tailed distribution makes `c3.4xlarge` an ineffective choice for Infrastructure-Level Replication at scale.

Experiment Setup: The next best option after `c3.4xlarge` is `r3.4xlarge`, which is 60% more expensive and offers approximately the same projected performance. We allocate 330 `r3.4xlarge` instances in a single placement group in the `us-east-1c` availability zone. We use a different zone because, as stated earlier in this work, it is often not possible to allocate a large number of instances in a particular zone. To each instance we attach eight 145 GB[1] general purpose EBS volumes. We use EBS for input and output data and local SSD for intermediate data, as suggested in Sections 3.2.3 and 3.3.4.

Results: We run Infrastructure-Level Replication three times and get completion times of 3094, 2914, and 2934 seconds, yielding an average completion time of 2,981 seconds and an average cost of $450.84 (Table 3.5). The first point to note is the total runtime, which includes two full rounds of I/O to EBS, is around 3000 seconds. When we compare this to a single round of I/O on `c3.4xlarge`, shown in Figure 3.10 to be more than 4000 seconds on a cluster of comparable size, we conclude that `r3.4xlarge` does not experience the same long-tailed behavior we see in `c3.4xlarge`. Because EBS is a black-box storage service, we can only guess as to the cause of this behavior. One hypothesis is that the network connecting `c3.4xlarge` to EBS is more congested,

[1] Actually 135 GiB. The EBS API uses GiB (2^{30}) rather than GB.

and thus more variable, than that of `r3.4xlarge`. It may also be possible that the `us-east-1c` availability zone itself experiences better EBS performance at scale.

Another interesting point is that the per-VM throughput is nearly half of the maximum 250 MB/s throughput to EBS. This indicates that each phase of the sort is running at near-optimal EBS speeds. In fact, Section 3.3.4 pins the ideal read and write bandwidths at 243 and 226 MB/s, respectively. This suggests an ideal end-to-end throughput of 117 MB/s, so our sort speed is 87% of optimal.

Sort Benchmark: The Infrastructure-Level-Application sort set the world record for both Indy and Daytona CloudSort. Because CloudSort was recently introduced, we do not have prior records to compare against. Further, losing submissions are not published. We can, however, compare to our Daytona GraySort record. We note that although far slower than Daytona GraySort in absolute speed, our CloudSort record actually sorts 100 TB about $35, or about 8%, cheaper with even stronger durability requirements.

3.5 Small-Scale Evaluation

In addition to the large-scale evaluations presented in Section 3.4, we also perform an evaluation at a much smaller scale to highlight the levels of performance possible if attention is given to efficient I/O.

Here, we sort 4094 GB of data (about 4% of the data size in Section 3.4) with the goal of completing the sort in under one minute. As we saw in Chapter 1, this small-scale sort permits more a efficient data-processing style due to relatively large amounts of available memory. In particular, since the data size can fit in the aggregate memory of the cluster, we modify Themis MapReduce in a similar fashion to the MinuteSort application in Section 1.5.2.

This variant of Themis MapReduce uses large allocations in the Demux, on

the order of 500 MB. These memory regions are not written to disk, but are rather passed immediately to Sorter threads at the start of the second phase. This modification eliminates one full round of disk I/O, and also permits the use of all storage devices for reading or writing, since the first phase is read-only and the second phase is write-only.

Additionally, we disable data sampling because the sampling step typically takes on the order of 30 seconds, which is half of our 60 second deadline. This configuration qualifies our sort for the Indy MinuteSort benchmark [81].

3.5.1 Results

We use the same 178 instances of `i2.8xlarge` allocated in the 2-IO experiment in Section 3.4.1. We perform 15 consecutive trials and report a median elapsed time of 58.8 seconds, with a maximum time of 59.8 seconds and a minimum time of 57.7 seconds, for an average of 58.7 seconds. Therefore, we sort 4094 GB in under a minute.

While our system sorts almost three times as much data as the prior year's Indy MinuteSort record of 1470 GB, we were bested by Baidu [45], which sorted an impressive 7 TB of data in under 60 seconds. We note that, as with our 100 TB 2-IO sort, Baidu used an implementation to TritonSort, our earlier work described in Chapter 1.

3.6 Conclusions

High-speed flash storage and 10 Gb/s virtualized networks supporting SR-IOV have enabled high performance data-intensive computing on public cloud platforms, and yet achieving efficiency remains challenging for these workloads. In this chapter, we present a systematic methodology for measuring the I/O capabilities of high-performance VMs, and extensively measure these features within EC2. We find that expected costs rise dramatically due to poor network scaling, altering the optimal choice of VM configurations. By provisioning based on performance measurements at scale, we demonstrate

highly efficient sorting on EC2 and set three new world records at very low cost.

3.7 Acknowledgements

Chapter 3 includes material that is submitted for publication as "Achieving Cost-efficient, Data-intensive Computing in the Cloud." Conley, Michael; Vahdat, Amin; Porter, George. The dissertation author was the primary author of this paper.

Chapter 4

Measuring Google Cloud Platform

As a final piece of work, we measure and analyze the performance of the Google Cloud Platform service. While our analysis of Amazon Web Services was thorough and comprehensive, this measurement will be exploratory in nature. We aim to make some simple comparisons between the Google and Amazon cloud providers with the goal of determining whether or not the results and methodology described in Chapter 3 generalize to other providers.

We first describe our efforts getting Themis to run on the Google Cloud Platform and discuss some potential issues. We then measure the variability of I/O resources in Google's cloud and compare with our findings on Amazon. Next, we run several small-scale sort operations to estimate the costs of running a 100 TB sort, much like we did in Chapter 3. Finally, we discuss a major issue we encountered with the local SSD storage on Google Cloud Platform.

4.1 Introduction

While Amazon Web Services [10], detailed in Chapter 3, has consistently been a leader in the cloud computing space, other providers also offer a wide variety of services that can be useful to cloud customers. Google Cloud Platform [30] is a collection of services offered by Google that provide various models of cloud computing. A

particularly well-known service is Google App Engine, which provides a dedicated API for creating scalable web-based applications. App Engine follows the platform-as-a-service model and requires users to conform to its APIs, rather than running arbitrary software.

More recently, Google introduced Compute Engine, which follows the infrastructure-as-a-service model and, much like Amazon EC2, allows users to run arbitrary pieces of software in a virtual machine. Because of its similarity to Amazon EC2, and because it permits us to run unmodified applications, we will focus on Google Compute Engine in this chapter. In particular, we are interested in the performance of I/O-intensive applications like Themis (Chapter 1).

4.2 Google Compute Engine

At a high level, Google Compute Engine is a service that allows users to rent virtual machines at pennies or dollars per hour, in order to run arbitrary computation at a large scale. Like Amazon's Elastic Compute Cloud, Google Compute Engine supports a large number of virtual machine types. There are 18 different types to choose from, and are divided into four categories that are nominally called Standard, High Memory, High CPU, and Shared-core, with 16 of the 18 types belonging to the first three categories. Unlike EC2, these categories appear to be more of a classification of the memory levels of the systems than of their intended use cases. As an illustration, a subset of virtual machine types are listed in Table 4.1.

From Table 4.1, we observe a few key facts. First, resources in Google Compute Engine are more loosely couple than in Amazon EC2. In particular, CPU cores appear to be more or less an independent parameter. Users may select 1, 2, 4, 8, 16 or 32 CPU cores for their virtual machine. Second, memory scales linearly with the number of cores. For example, `n1-standard-8` has exactly eight times as much memory as `n1-standard-1`.

Table 4.1. Five example Compute Engine machine types with various CPU and memory capabilities.

Type	vCPU	Memory (GB)	Cost ($/hr)
`n1-standard-1`	1	3.75	0.05
`n1-standard-8`	8	30	0.40
`n1-highcpu-32`	32	28.8	1.216
`n1-standard-32`	32	120	1.60
`n1-highmem-32`	32	208	2.016

Similarly, the 32-core variant has 32 times as much memory. Third, the three classes of instances correspond to thee different memory levels, rather than different intended use cases. Here, the High CPU class really means that the virtual machine has low levels of memory. Similarly, The Standard class has moderate amounts of memory, and the High Memory class has large amounts of memory. Fourth price, varies linear with the number of cores, and approximately linearly over the three levels of memory.

The final and most striking observation is that Compute Engine virtual machines do not have any local storage. This appears to be a design feature of Google's cloud service. Users that wish to store data on disks are encouraged to use Persistent Disks, which are network-attached storage devices akin to Amazon's Elastic Block Store (EBS) persistent network-attached storage service. Google's Persistent Disks can be created from either magnetic hard disk drives, or flash-based solid state drives. The performance of Persistent Disks is directly proportional to the size of the disk, up to a certain per-core or per-VM limit. Furthermore, writes to Persistent Disks count against the network egress limit for a given virtual machine, meaning that an application that writes data and sends data over the network may experience reduced performance.

4.2.1 Local SSDs

Google has recently added the ability to allocate virtual machines with locally-attached SSDs in addition to remote-attached Persistent Disks. Unlike Persistent Disks,

local SSD capacity is fixed at 375 GB, and between zero and four SSDs may be configured for a single virtual machine. Local SSDs are exposed through either a SCSI interface or NVMe interface, although operating system support is required to use the NVMe interface.

Google's documentation suggests using at least 4 vCPU cores per local SSD, implying that configurations with large amounts of local storage will need large amounts of CPU to drive the performance of the devices.

Local SSDs increase the hourly cost of a virtual machine by a small, fixed amount per SSD used. As of this writing, the estimated hourly cost per device is $0.113, so configuring a `n1-standard-32` virtual machine with four local disks increases the hourly cost from $1.60 (Table 4.1) to $2.052, which is an increase of 28%.

4.2.2 Network Placement

In Section 3.2, we noted how Amazon offers placement groups in EC2 to provide better networking guarantees by allocating groups of virtual machines such that bisection bandwidth is high. In particular, we showed this feature improved performance substantially for the `i2.8xlarge` VM type.

Google Compute Engine does not offer such a feature. This could be by design, or it could simply be a matter of time before this feature is offered. In any case, there is currently no way for users to specify stronger networking requirements. As we will see, this may affect performance on Google Compute Engine.

4.3 Variance in Google Compute Engine

We are primarily interested in determining if the lack of networking placement, described previously, impacts performance on Google Compute Engine. However, because we do not have a placement feature, we cannot make a direct comparison as we

did with Amazon EC2.

In order to better answer this question, we choose to measure the variance of five clusters with identical specifications. Because we launch these clusters separately, we postulate that observed variance in the network can be attributed to the lack of better network placement.

Ideally, all clusters would be launched simultaneously. Unfortunately, resource provisioning limits within Google Compute Engine prevent us from launching more than one cluster at a time. We therefore admit that, like with our Amazon measurements, time of day may impact our results.

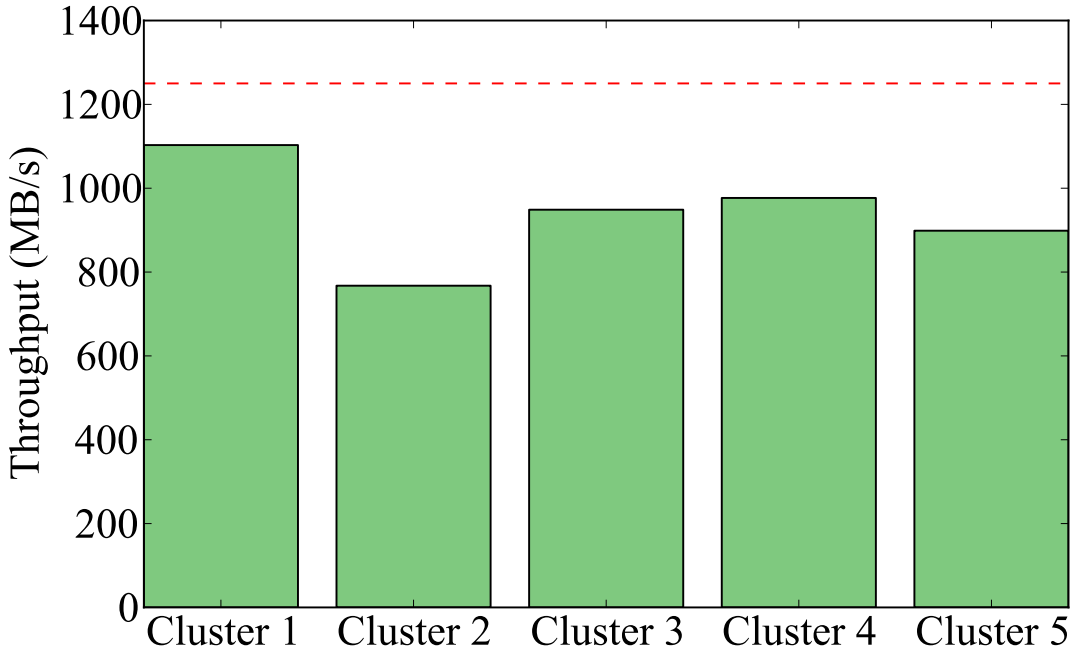
4.3.1 Experiment Setup

Each cluster consists of 10 virtual machines of the `n1-standard-8` type. In order to tailor the result to I/O-bound applications, we restrict focus to the virtual machines capable of providing large amounts of local storage by configuring each virtual machine with the maximum of four local SSDs.

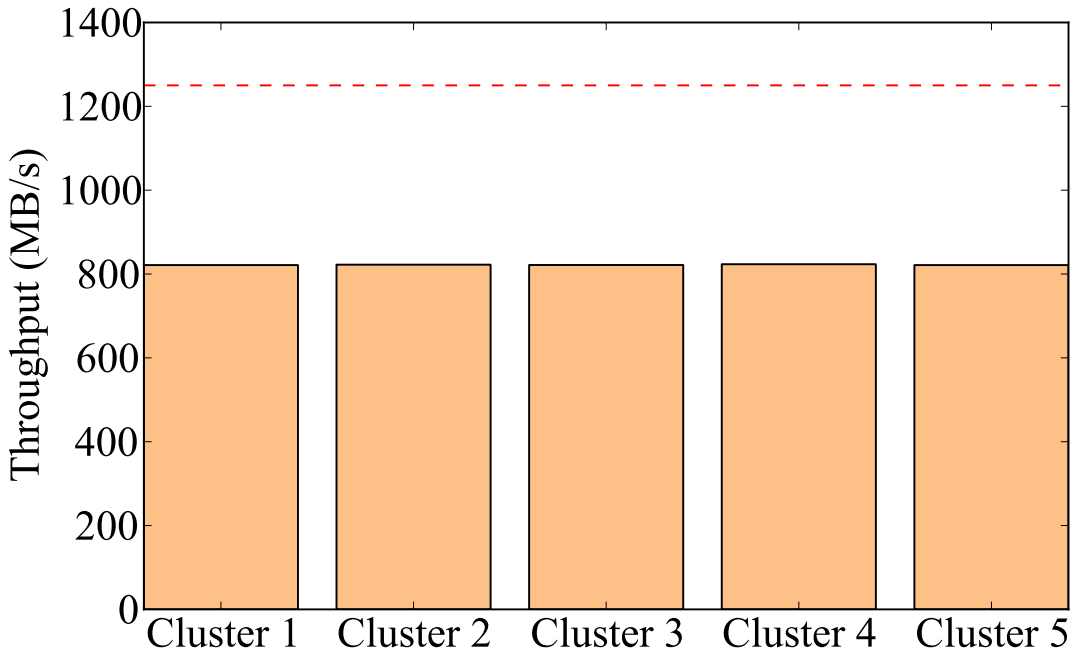
On each cluster, we first run some preliminary tests to make sure the hardware is not faulty, and then we measure the storage and networking performance using DiskBench and NetBench (Section 2.6. For each cluster, we run each of these benchmarks three times using data sizes that require approximately three minutes per benchmark invocation. We compute the average over these three benchmark runs for each resource and for each cluster.

4.3.2 Results

Figure 4.1a shows the networking performance for the five clusters. The average all-to-all network throughput of the clusters is 939 MB/s, with a standard deviation of 109 MB/s. In particular, the slowest cluster has a network that runs at 767 MB/s, and the



(a) Network performance.



(b) Storage performance.

Figure 4.1. Network and storage performance for five identically configured clusters of 10 nodes of the `n1-standard-8` virtual machine, each configured with four local SSDs.

fastest has a network capable of 1103 MB/s, which is 44% faster.

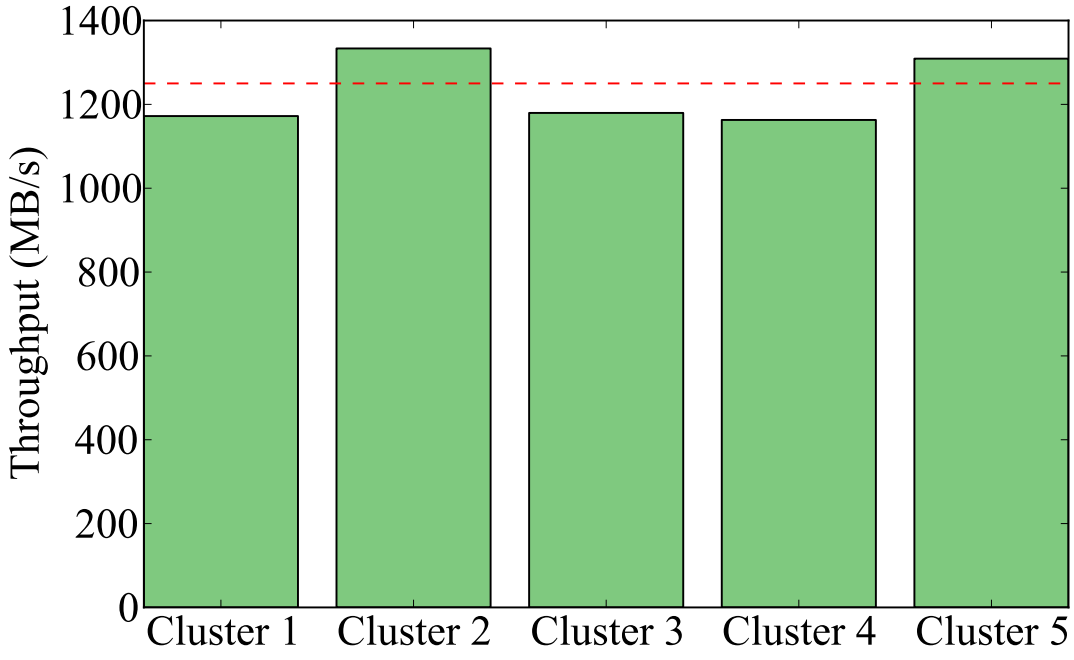
For reference, we also compare the storage performance across the local SSDs in the clusters, which is shown in Figure 4.1b. We note that there is essentially no variance in the storage performance of the SSDs in these clusters. We report an average DiskBench throughput of 822 MB/s with a standard deviation of less than 1 MB/s.

4.3.3 Different Cluster Configurations

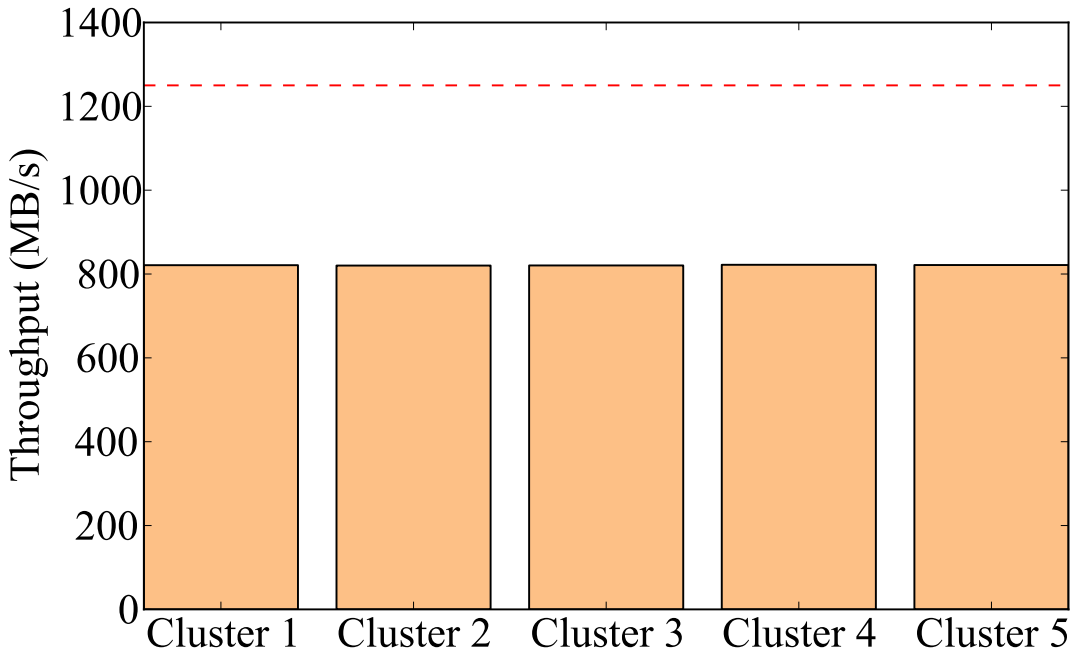
Our experience with Amazon EC2 in Chapter 2 shows performance and variance differs dramatically as the virtual machine configuration changes. In order to check if this occurs in Google Compute Engine, we repeat the previous experiment using the `n1-highmem-32` virtual machine type, again with four local SSDs per virtual machine and 10 virtual machines per cluster.

We choose to measure `n1-highmem-32` because it has the largest number of compute and memory resources out of all of the virtual machine types. On Amazon EC2, we found that such configurations were often implemented as dedicated virtual machines, which eliminates contention from other users. While we cannot make this claim for Google Compute Engine, we postulate that is more likely that contention is lower on `n1-highmem-32` than on `n1-standard-8`, which could reduce variance or improve performance.

The NetBench results from this experiment are shown in Figure 4.2a. The clusters have an average network bandwidth of 1231 MB/s, with a standard deviation of 74 MB/s. In particular, these clusters are both faster and more consistent than the `n1-standard-8` clusters. However, the variance is still substantial. The slowest cluster has a network bandwidth of 1163 MB/s, while the fastest runs at 1333 MB/s, which is about 15% faster. We also note that two of the clusters have network bandwidths larger than 1250 MB/s, or 10 Gb/s, indicating that, unlike Amazon, Google's networks are capable of exceeding



(a) Network performance.



(b) Storage performance.

Figure 4.2. Network and storage performance for five identically configured clusters of 10 nodes of the `n1-highmem-32` virtual machine, each configured with four local SSDs.

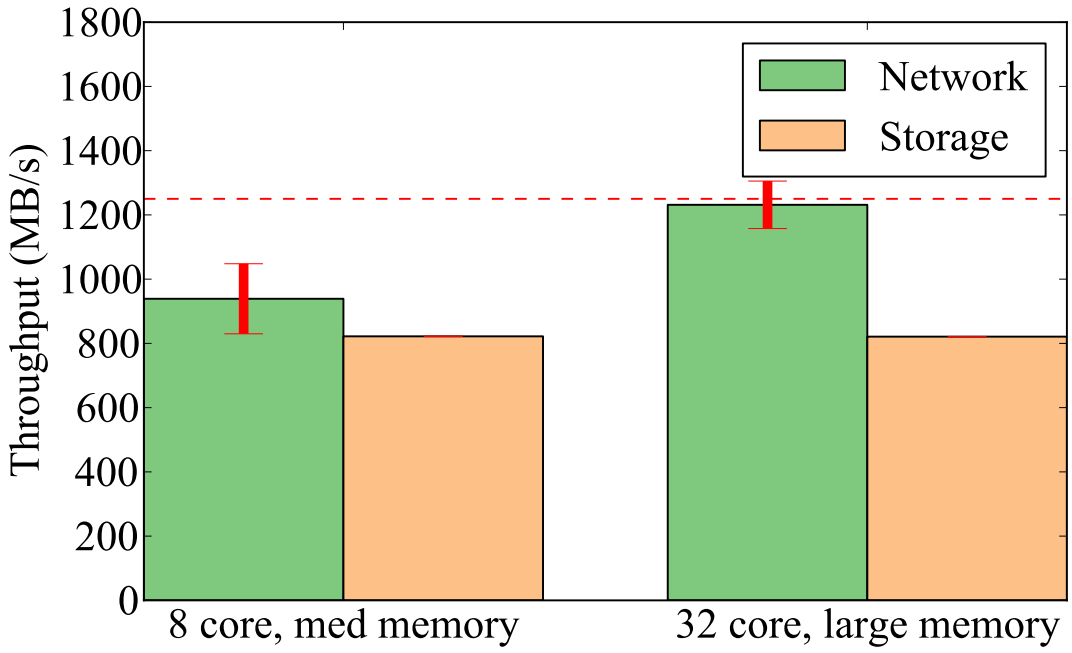


Figure 4.3. Summary of networking and storage performance of `n1-standard-8` and `n1-highmem-32`. Error bars show one standard deviation.

10 Gb/s.

Again for comparison we show the DiskBench results in Figure 4.2b. The average storage bandwidth is 821 MB/s, with a standard deviation of less than 1 MB/s. In particular, the difference in storage performance between `n1-standard-8` and `n1-highmem-32` is negligible, and is consistently about 820 MB/s.

4.3.4 Summary

These results are summarized in Figure 4.3. While these measurements are hardly comprehensive, we can postulate some theories as to what might be happening. Cloud virtual machines are typically implemented by taking large servers and carving them up into smaller configurations. In particular, the provider can dynamically choose to carve up a server into a small number of large virtual machines, or a large number of small virtual machines.

Based on Google Compute Engine's offerings, it may be the case that both the `n1-standard-8` and `n1-highmem-32` virtual machines are carved up from the same set of physical servers. These servers might have 32 or 64 cores, and at least 208 GB of memory. While we do not expect every physical server in Compute Engine to have local SSDs, we can certainly be sure that every physical server used in these experiments hosts at least four local SSDs, since all virtual machines have access to these SSDs. The fact that storage performance is the same across both virtual machine types supports the hypothesis that the underlying physical servers might be the same.

Differences in the network performance can be due to a variety of factors. If it is the case that both virtual machines use the same underlying physical servers, the `n1-standard-8` virtual machines may experience more contention from co-located users because the physical server can be carved up into more virtual machines, which can all share the same physical network interface. This contention can reduce performance and increase variance, consistent with our results.

Regardless of whether or not the underlying physical servers have the same configuration, it can also be the case that Google Compute Engine's internal allocation policy uses different sets of physical servers for different virtual machine configurations. In this case, there can be a natural difference in performance and variance because data center becomes loosely segregated, with different virtual machine configurations accessing different portions of the physical network, leading to heterogeneous behavior and performance across the data center's network.

4.3.5 Network Placement

Google Compute Engine does not have a feature analogous to placement groups in Amazon EC2. The existence of such a feature might improve the network performance and variance issues measured here. In particular, virtual machines could be placed to

Table 4.2. The eight instance types involved in the sorting experiment.

Type	vCPU	Memory (GB)	Cost ($/hr)
n1-highmem-32	32	208	2.016
n1-standard-32	32	120	1.60
n1-highcpu-32	32	28.8	1.216
n1-highmem-16	16	104	1.008
n1-standard-16	16	60	0.80
n1-highcpu-16	16	14.4	0.608
n1-highmem-8	8	52	0.504
n1-standard-8	8	30	0.40

reduce contention in the network from other users, improving performance.

4.4 Sorting on Google Compute Engine

We now describe a small-scale analysis of sorting on Google Compute Engine. Rather than perform a comprehensive large-scale evaluation as we did in Chapter 3, we instead measure a small number of virtual machine types at a small-scale, and extrapolate our results. We reiterate that the intention of this work is simply to get a baseline comparison between Amazon EC2 and Google Compute Engine, in terms of the performance, efficiency and cost metrics considered in the previous chapter.

4.4.1 Experiment Setup

We measure eight instance types shown in Table 4.2. We choose these particular instance types because they support enough CPU and memory to drive four local SSDs at near-maximal capacity. Intuitively, we expect that using more virtual machines hosting fewer SSDs will suffer performance losses due to poor network scaling behavior, as observed in Chapter 3. Therefore, we only consider these eight virtual machines with four local SSDs attached to each.

For each instance type, we instantiate a cluster of 10 nodes. We run all experiments on the same cluster, and we run each experiment three times to account for natural

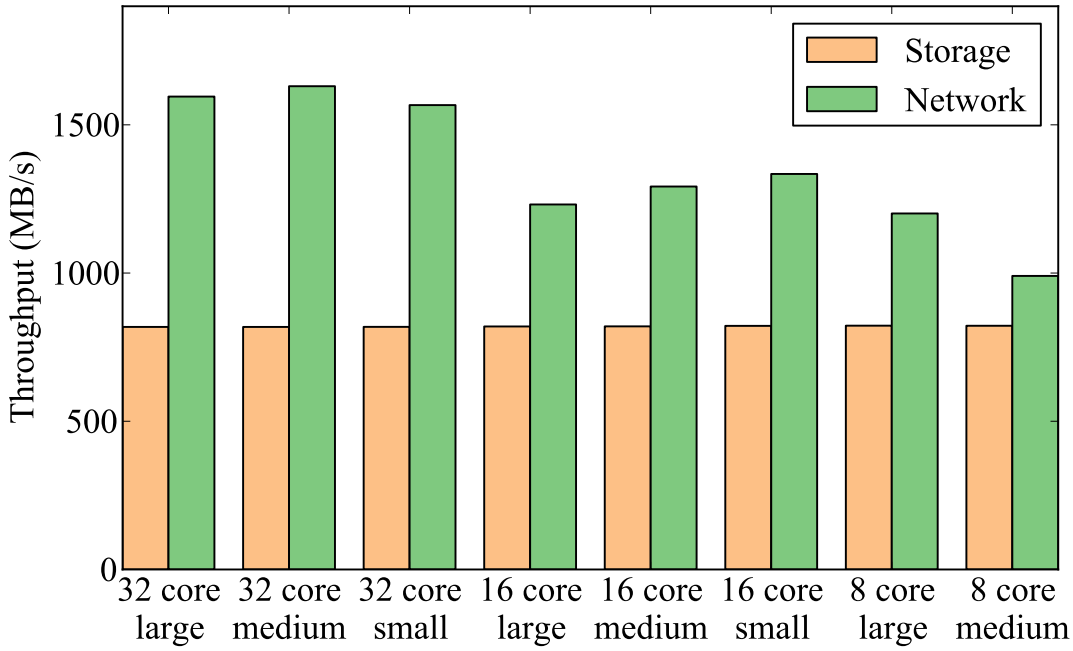


Figure 4.4. DiskBench and NetBench measurements across the eight instance types.

variance within a cluster. In the data that follows, we sometimes refer to the virtual machine types by their number of cores and memory sizes instead of the full machine type name. For example, `n1-highmem-32` is a 32 core virtual machine with a large memory size, while `n1-highcpu-16` is a 16 core virtual small with a small memory size.

4.4.2 Benchmarks

Before running the sort application, we first run the DiskBench and NetBench microbenchmarks to get a baseline performance assessment. We run each benchmark three times on data sizes that take three to five minutes to process. We compute the average bandwidth across all runs, and the results are shown in Figure 4.4. In all cases, the storage bandwidth is less than the network bandwidth, often by a large margin. We therefore predict that if the sort application is I/O-bound, it will in fact be storage-bound on these virtual machines at least at a small scale.

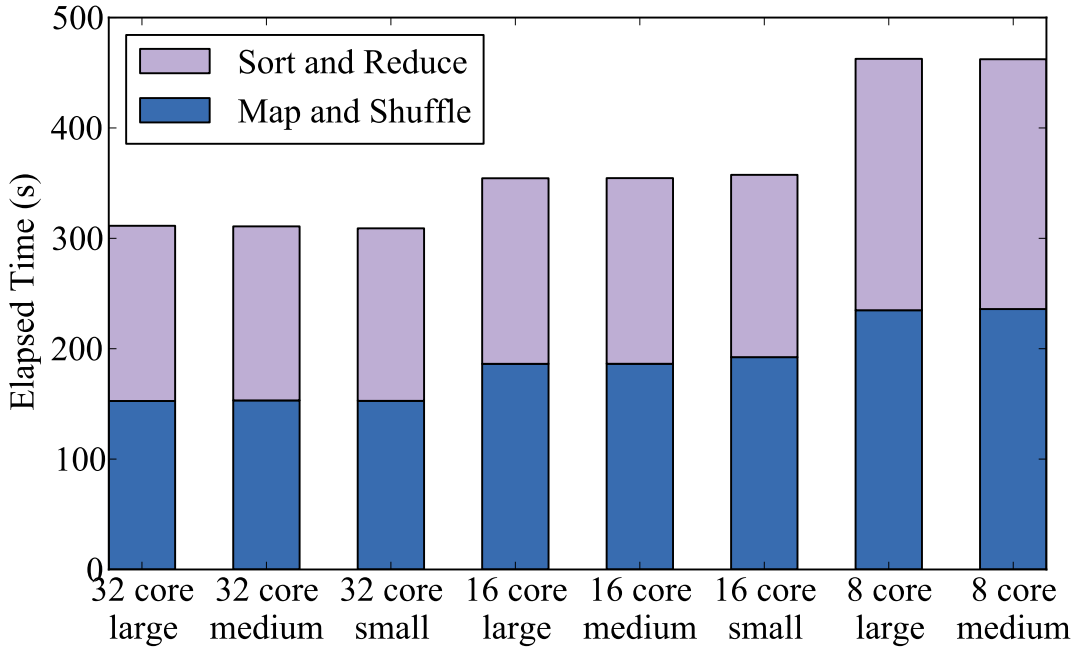


Figure 4.5. Running time for a 1.2 TB sort operation on 10 nodes across the eight instance types.

4.4.3 Sort

Next we run a 1.2 TB sort operation on each cluster of 10 nodes. This results in a data size of 120 GB per node, or 60 GB per input disk. Given that each SSD has a capacity of 375 GB, this results input files that take up 16% of the disk's capacity. We run this sort three times, and for each cluster we consider the run with the median completion time in the measurements that follow.

The running time for the two phases of the sort operation is shown in Figure 4.5. There are two interesting observations to make. First, the running time is strongly correlated with the number of cores. While not linear in the number of cores, all machines of a specific core count have roughly the same levels of sorting performance. Second, the running times of the map and shuffle and sort and reduce phases is approximately equal in all cases. If it is the case that the sort is I/O-bound, this observation makes sense

because storage bandwidth will be the bottleneck in both phases and each phase should run at approximately the same rate.

To put these results in perspective, we note that the 2014 Indy GraySort record for sorting 100 TB is 716 seconds using 982 servers. If the results of this experiment scale, the 32-core configurations are capable of sorting 100 TB twice as fast as the current world record using about 830 VMs. However, efficiently deploying sort at this scale is likely to be a challenge, as demonstrated in Chapter 3.

A more likely deployment would use a larger data size per node, up to the capacity of the available SSDs, in order to use a smaller cluster size. Such an experiment would have results more comparable to those in Chapter 3. The 100 TB 2-IO sort measured in Section 3.4 ran in 888 seconds on 178 VMs. If we were to run a 100 TB sort on Compute Engine using three times as much data per node as measured here, we would approach the capacity of the input and output SSDs. Such a cluster would sort data three times slower using three times fewer VMs. This extrapolation yields an expected running time of about 930 seconds using about 280 VMs. We note that while the running time is similar to our result on Amazon EC2, the cluster size is much larger. This is due to the fact that the local SSDs on Google Compute Engine are slower than those on the `i2.8xlarge` EC2 instance type.

To determine whether or not the sorts are indeed I/O-bound, we compare the throughput of each phase of the sort with the expected throughput of a truly I/O-bound phase, based on the DiskBench measurements. The results are shown in Figure 4.6.

We see that the 32-core virtual machines sort data at roughly the speed estimated by our benchmarks, indicating that they are indeed I/O-bound. However, the 16-core virtual machines sort data at a slightly slower rate. This difference is even more pronounced for the 8-core virtual machines. In fact, we can confirm by log analysis that these configurations are CPU-bound.

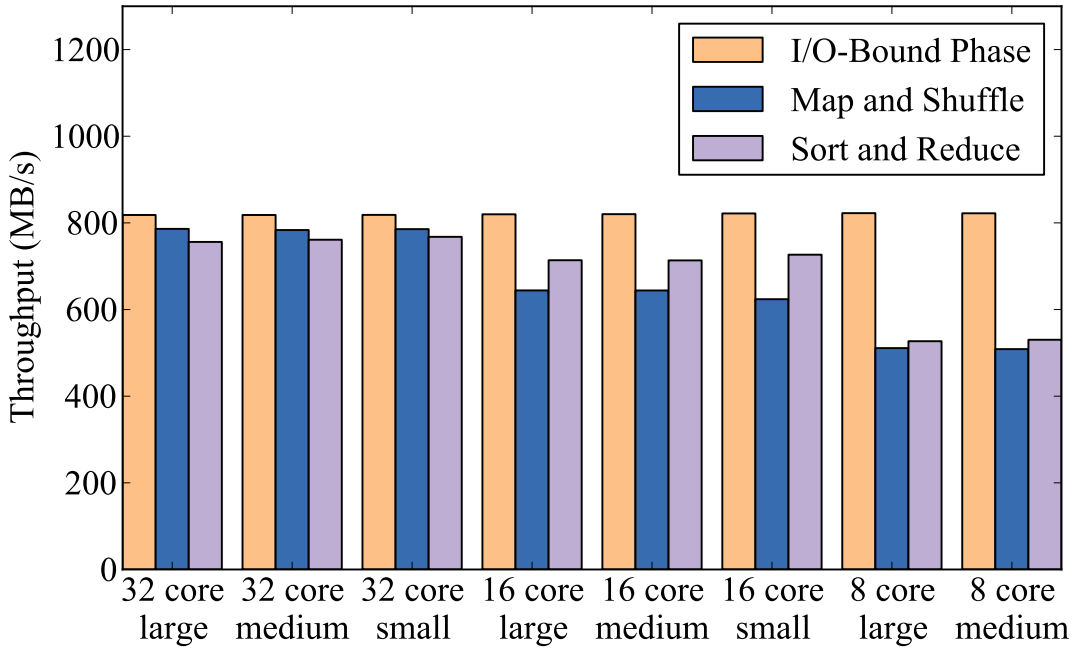


Figure 4.6. Phase bandwidths for sorting across the eight instance types. The expected bandwidth of an I/O-bound phase is shown for comparison.

4.4.4 Estimating the Cost of Sorting

We can apply the same analysis in Chapter 3 to estimate the cost of sorting 100 TB of data. However, the analysis in this chapter differs in two ways from that in the previous chapter. First, we do not have large-scale measurements to assess the scaling behavior of the network. Second, we do have small-scale sorting measurements, which are more accurate than our benchmark since the sort is CPU-bound in some cases.

We also note that the per-hour cost of each virtual machine is $0.452 more than reported in Table 4.2. Google Compute Engine's pricing model adds approximately $0.113 onto the hourly cost a virtual machine for each locally-attached SSD.

Our estimated sorting costs are shown in Figure 4.7. Here we can clearly see the cost increases for the 16-core and 8-core configurations due to being CPU-bound. However, generally speaking the smaller core and smaller memory configurations are

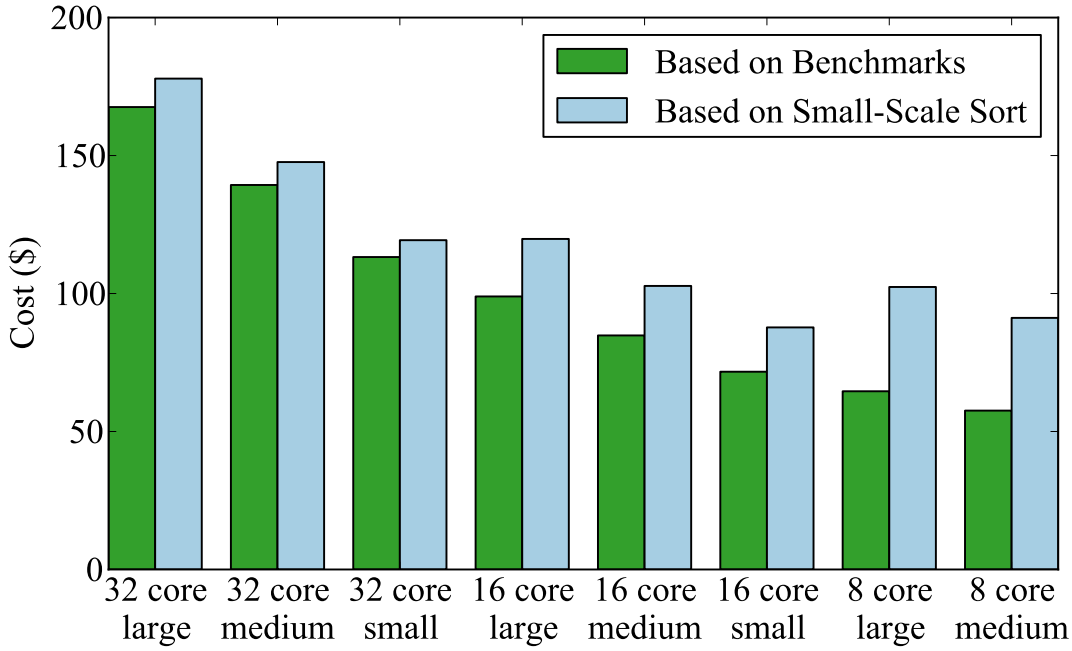


Figure 4.7. Estimated cost of sorting 100 TB across eight instance types. The expected cost of an I/O-bound sort is shown for comparison.

cheaper in terms of the total cost of the sort. In fact, the `n1-highcpu-16` (16-core, small memory) configuration has the lowest estimated cost at roughly $88 per 100 TB sort.

We note that network scaling factors make come into play when running a large-scale 100 TB sort operation. The 32-core VM types have the best per-VM performance levels, resulting in fewer VMs required for a 100 TB sort. If scaling is an issue, the `n1-highcpu-32` (32-core, small memory) configuration can sort 100 TB for approximately $119.

4.4.5 Comparison to Amazon EC2

The cheapest EC2 virtual machine type measured in Chapter 3 is `i2.8xlarge` with placement groups, with an estimated cost of $325 per 100 TB sort. In contrast, the cheapest Google Compute Engine machine type, `n1-highcpu-16`, has an estimated cost of $88 per 100 TB sort, which is about 27% of the cost on EC2. Even the most expensive

VM type measured on Compute Engine, `n1-highmem-32`, has an estimated cost of $178, which is still 55% of the cost of the cheapest VM type on EC2.

However, we note that this comparison is not entirely fair because we do not consider the network scaling properties of Google Compute Engine. The measurements in this chapter are more akin to the ideal network scalability assumption in Chapter 3. Under this assumption, the cheapest EC2 virtual machine type measured is `m1.xlarge`, with an approximate sort cost of $229, which is still more expensive than the most expensive virtual machine on Google Compute Engine.

We can therefore conclude that unless scaling behavior on Compute Engine is substantially worse than on EC2, it will be significantly cheaper to sort large amounts of data using Compute Engine. This conclusion, while somewhat weak, is the best we can do short of running a large-scale comprehensive measurement of Compute Engine.

4.5 Local SSD Issues

In the process of porting Themis MapReduce to Google Compute Engine, we encountered an issue with the correctness of certain workloads on the local SSDs measured in the previous section. We experienced data corruption that occurs randomly, with high probability, if the following three conditions are satisfied:

1. I/O operations are performed asynchronously using the Linux Asynchronous I/O library `libaio`.

2. Direct I/O is used via the `O_DIRECT` flag.

3. Files are pre-allocated to large sizes using `fallocate()`.

In this case, we noticed occasional 4 KiB chunks of zeroes towards the end of files in Themis. These blocks of zeroes are infrequent, but with enough files, the probability

of at least one zero block becomes significant. We observed zero blocks in intermediate files in the map and shuffle phase of Themis, as well as in the output files in the sort and reduce phase. However, the likelihood of observing these blocks in the sort and reduce phase is much higher than in the map and shuffle phase.

While we do not know exactly what causes this issue, and we cannot rule out a bug within Themis MapReduce, we note that we did not experience this bug on Amazon EC2 or on our own clusters, even with the same three conditions described above.

There are several possible causes for this issue. The first is that the Google Compute Engine hypervisor may be faulty. The second is that the SSDs themselves may be faulty. The third is that the `libaio` library, which is self-described as not fully implemented, may be faulty. Finally, it may be the case that there is a subtle interaction between the application, the asynchronous I/O library, the operating system, the file system, the hypervisor, and the SSDs that is causing the issue.

We note that removing any one of the three conditions appears to fix the bug, although we cannot rule out the possibility that the likelihood of occurrence simply drops low enough so that it cannot be easily observed. In particular, file pre-allocation, while useful for performance on HDDs, does not appear to improve performance on Google Compute Engine's SSDs. Therefore, we can disable pre-allocation and circumvent this issue.

4.6 Conclusions

Google Compute Engine offers highly-configurable virtual machines and is an attractive alternative to providers such as Amazon EC2. In this chapter, we describe the fundamental differences between Compute Engine and EC2. We then perform a small-scale analysis of Compute Engine to see how it compares to EC2. Extrapolating this analysis suggests that Compute Engine can be substantially cheaper than EC2 for

running large-scale sorting operations.

Chapter 5

Related Works

Thus far we have given a detailed description of building efficient data-intensive systems and how to run them on a variety of hardware platforms. We now describe some of the more relevant related works. In particular, we first look at the space of sorting and MapReduce systems. Next, we consider measurement of the cloud. Finally, we give a detailed description of skew, and how existing techniques can be used to solve efficiency problems that arise when we measure workloads other than the uniform sort workload that is so prevalent in this dissertation.

5.1 Sorting

The Datamation sorting benchmark[12] initially measured the elapsed time to sort one million records from disk to disk. As hardware has improved, the number of records has grown to its current level of 100TB as described in Section 1.1.1. Over the years, numerous authors have reported the performance of their sorting systems, and we benefit from their insights[67, 49, 97, 8, 64, 63]. We differ from previous sort benchmark holders in that we focus on maximizing both aggregate throughput and per-node efficiency.

Achieving per-resource balance in a large-scale data processing system is the subject of a large volume of previous research dating back at least as far as 1970. Among the more well-known guidelines for building such systems are the Amdahl/Case rules of

thumb for building balanced systems [3] and Gray and Putzolu's "five-minute rule" [36] for trading off memory and I/O capacity. These guidelines have been re-evaluated and refreshed as hardware capabilities have increased.

NOWSort[8] was the first of the aforementioned sorting systems to run on a shared-nothing cluster. NOWSort employs a two-phase pipeline that generates multiple sorted runs in the first phase and merges them together in the second phase, a technique shared by DEMSort[67]. An evaluation of NOWSort done in 1998[9] found that its performance was limited by I/O bus bandwidth and poor instruction locality. Modern PCI buses and multi-core processors have largely eliminated these concerns; in practice, TritonSort is bottlenecked by disk bandwidth.

TritonSort's staged, pipelined dataflow architecture is inspired in part by SEDA[93], a staged, event-driven software architecture that decouples worker stages by interposing queues between them. Other data-intensive systems such as Dryad [42] export a similar model, although Dryad has fault-tolerance and data redundancy capabilities that TritonSort does not implement. The modifications to Themis described in Chapter 3 implement some data redundancy features, but not to the degree that an enterprise-grade system requires.

We are further informed by lessons learned from parallel database systems. Gamma[20] was one of the first parallel database systems to be deployed on a shared-nothing cluster. To maximize throughput, Gamma employs horizontal partitioning to allow separable queries to be performed across many nodes in parallel, an approach that is similar in many respects to our use of logical disks. TritonSort's sender-receiver pair is similar to the exchange operator first introduced by Volcano[34] in that it abstracts data partitioning, flow control, parallelism and data distribution from the rest of the system.

5.2 MapReduce

There is a large continuum of fault tolerance options between task-level restart and job-level restart, including distributed transactions [66], checkpointing and rollback [26], lineage-based recovery [107] and process-pairs replication [79]. Each fault tolerance approach introduces its own overheads and has its own complexities and limitations. With Themis, we choose to focus our efforts on creating a MapReduce system model that is able to handle large real-world data sets while utilizing the resources of an existing cluster as much as possible.

Recovery-Oriented Computing (ROC) [73, 14] is a research vision that focuses on efficient recovery from failure, rather than focusing exclusively on failure avoidance. This is helpful in environments where failure is inevitable, such as data centers. The design of task-level fault tolerance in existing MapReduce implementations shares similar goals with the ROC project.

Sailfish [68] aims to mitigate partitioning skew in MapReduce by choosing the number of reduce tasks and intermediate data partitioning dynamically at runtime. It chooses these values using an index constructed on intermediate data. Sailfish and Themis represent two design points in a space with the similar goal of improving MapReduce's performance through more efficient disk I/O.

Several efforts aim to improve MapReduce's efficiency and performance. Some focus on runtime changes to better handle common patterns like job iteration [13], while others have extended the programming model to handle incremental updates [55, 66]. Work on new MapReduce scheduling disciplines [108] has improved cluster utilization at a map- or reduce-task granularity by minimizing the time that a node waits for work. Tenzing [15], a SQL implementation built atop the MapReduce framework at Google, relaxes or removes the restriction that intermediate data be sorted by key in certain

situations to improve performance.

Massively parallel processing (MPP) databases often perform aggregation in memory to eliminate unnecessary I/O if the output of that aggregation does not need to be sorted. Themis could skip an entire read and write pass by pipelining intermediate data through the `reduce` function directly if the `reduce` function was known to be commutative and associative. We chose not to do so to keep Themis's operational model equivalent to the model presented in the original MapReduce paper. This model is implemented, however, in the MinuteSort application described in Section 1.5.2.

Characterizing input data in both centralized and distributed contexts has been studied extensively in the database systems community [58, 59, 38], but many of the algorithms studied in this context assume that records have a fixed size and are hence hard to adapt to variably-sized, skewed records. Themis's skew mitigation techniques bear strong resemblance to techniques used in MPP shared-nothing database systems [21].

5.3 Cloud Computing

While many previous works have studied performance in the public cloud, we note that our work is unique in that it has the following three aspects:

- We measure clusters composed of 100s of VMs
- We measure VMs offering high-performance virtualized storage and network devices
- We measure workloads making use of 100s of terabytes of cloud-based storage

We now discuss several related studies in cloud computing.

Measurement: Many have measured the public cloud's potential as a platform for scientific computing. Walker [90] compared Amazon Elastic Compute Cloud (EC2)

to a state-of-the-art high-performance computing (HPC) cluster. Mehrotra et al. [60] performed a similar study four years later with NASA HPC workloads. Both came to the same conclusion that the network in the public cloud simply is not fast enough for HPC workloads.

Others have identified this problem of poor I/O performance and have studied the impact of virtualization on I/O resources. Wang and Ng [92] measure a wide variety of networking performance metrics on EC2 and find significantly more variance in EC2 than in a privately owned cluster. Ghoshal et al. [32] study storage I/O and find that EC2 VMs have lower performance and higher variability than a private cloud designed for scientific computation.

Variability in the cloud extends to CPU and memory resources as well. Schad et al. [76] measure the variability of a wide variety of VM resources and find that among other things, heterogeneity in the underlying server hardware dramatically increases performance variance. Two VMs of the same type may run on different processor generations with different performance profiles.

In a somewhat different line of study, Li et al. [53] measure inter-cloud variance, that is, the difference in performance between cloud providers. They compare Amazon EC2, Microsoft Azure, Google AppEngine and RackSpace CloudServers across a variety of dimensions and find that each cloud provider has its own performance profile that is substantially different from the others, further complicating the choice of resource configuration in the public cloud.

Configuration: One goal of measuring the cloud is optimal, automatic cluster configuration. Herodotou et al. [39] describe Elasticizer, a system that profiles Hadoop MapReduce jobs and picks an optimal job configuration on EC2. Wieder et al. [95] construct a similar system, Conductor, that combines multiple cloud services and local

servers in a single deployment.

Scheduling around deadlines in shared clusters is another common line of work. ARIA [87] is a scheduler for Hadoop that meets deadlines by creating an analytical model of MapReduce and solving for the appropriate number of map and reduce slots. Jockey [28] is a similar system for more general data-parallel applications. Bazaar [44] translates these efforts to the cloud by transforming the typical resource-centric cloud API to a job-centric API whereby users request job deadlines rather than collections of VMs. In this model, the cloud provider applies the job profile to an analytical model to compute the cheapest way to meet the job's deadline.

Scale: In the public cloud, users are often presented with a choice of whether to use a larger number of slow, cheap VMs or a smaller number of fast, expensive VMs. The choice to scale out or scale up often depends on the technology available. Michael et al. [61] compared a scale-up SMP server to a scale-out cluster of blades and found the scale-out configuration to be more cost effective. Half a decade later, Appuswamy et al. [7] revisited this question in the context of Hadoop and found the opposite to be true: that a single scale-up server is more cost-effective than a larger scale-out configuration.

While the relative costs of either approach change over time, scale-out configurations must be cautious to avoid excessive variance. Dean and Barroso [18] study tail latency in Web services at Google and demonstrate very long-tailed latency distributions in production data centers. They specifically call developers to build *tail tolerance* into their systems to avoid performance loss. Xu et al. [103] take a pragmatic approach and develop a system to screen for and remove outlier VMs in the long tail.

At the same time, Cockcroft [17] demonstrates how Netflix takes advantage of scale-up VMs on EC2 to reduce costs while substantially simplifying cluster configuration. Cockcroft relies on newer SSD-based VMs, indicating that available hardware

drives the choice of whether to scale out or scale up. Of course, the software must also be capable of taking advantage of scale-up. Sevilla et al. [78] describe an optimization to MapReduce that alleviates I/O bottlenecks in scale-up configurations.

Modern, large-scale data-processing systems are now being designed to eliminate as much extraneous I/O as possible. Spark [107] is highly optimized for iterative and interactive workloads that can take advantage of small working sets and large memories. While this target workload is different, the spirit of this work is the same as ours. In fact, Databricks, using Apache Spark, set a world record for sorting using the same AWS VM configuration we derive in Chapter 3 [100, 99, 98]. In the same sorting contest, Baidu [45] set a record using an implementation of TritonSort [71], further highlighting the need for efficient I/O processing.

5.4 Skew in Parallel Databases

The TritonSort and Themis architectures described in Chapter 1 were primarily tested on the uniform sort workload described in Section 1.1.1. While these systems have mechanisms for handling other workloads, name the sampling phase termed *phase zero*, a variety of performance issues can occur when considering "real" workloads.

In particular, we made significant changes to the sampling mechanism described in Section 1.6.4 in order to handle a particular workload that is *heteroscedastic*, meaning that the record distribution changes over the course of the input file. This is just one example of skew hampering the performance of general purpose data processing systems. We now give a detailed description of skew mitigation techniques that can be useful for handling these corner cases. We first describe techniques from the database community, and then we describe techniques for MapReduce-like systems.

5.4.1 Background

The database community has been operating at the heart of large data processing systems for many decades. Relational databases are responsible for finding the optimal way to translate a user's query into a sequence of operators that read and transform data stored in persistent relations to produce the desired result. Efficient software design gives rise to upward scalability, but such architectures quickly become expensive. A more practical solution is to use a scale-out architecture that parallelizes the relational database software [21].

One of the most complex operators in a database query is the join operator. A join takes tuples from several relations and combines them together, typically by some filtering predicate such as attribute equality. Since a join must in the worst case look at every pair of tuples, it is an obvious target for optimization. Parallel databases compound this problem by requiring an even partitioning of the join workload in order to get parallel speedup. If join partitions are incorrectly computed, the join will be slow, which impacts the overall query response time. The rest of this section will describe the impact of skew on the space of parallel join algorithms.

5.4.2 Parallel Join Algorithms

The simplest join algorithm is a *nested loop join*. The nested loop join operates by comparing every tuple in one relation, the *outer relation*, with every tuple of the other relation, the *inner relation*. This is analogous to a pair of nested `for` loops, where the *inner* loop executes inside the *outer* loop. As in the case of the `for` loops, the nested loop join compares tuples one by one. It therefore makes a quadratic number of comparisons. More importantly, it requires multiple scans if the inner relation is too large to fit in memory.

Researchers quickly determined that nested loop join was not terribly efficient

and began to look at other types of joins. Schneider and DeWitt [77] compared the performance of four popular join algorithms in a shared-nothing multiprocessor environment, namely the Gamma database system. There are essentially two popular categories of join algorithms beyond nested loop. The first type, termed *sort-merge join*, relies on merging sorted runs, and the second type, *hash join*, uses hash functions to speed up comparisons. It should be noted that this use of a hash function is distinct from *hash partitioning*, in which relations are partitioned by hashing the join attribute. Partitioning is orthogonal to the join type. For example, a sort-merge join could conceivably use hash partitioning to divide the relations. This partitioning can be avoided if the relations are already partitioned on the join attributes, but this cannot be assumed in the general case.

The sort-merge join algorithm begins with an initial partitioning of the relations across the cluster by the join attributes. Tuples are written to a temporary file as they are received across the network. After all tuples have been redistributed, each file is sorted. Finally, the join result is efficiently computed via a merge of the sorted partition files for each relation. Joins can be computed locally because the relations have been redistributed on the join attributes.

In the same work, Schneider and DeWitt survey three types of hash join algorithms. The first is a *simple hash join* algorithm that begins by redistributing relations on the join attributes. As tuples are received at their destination sites, a hash table is constructed from the inner relation using a second hash function. The tuples from the outer relation probe the hash table using this second hash function to compute the join result locally. Note that a hash join algorithm such as simple hash join will only work if the join is an *equijoin*, i.e. a join with an equality condition on join attributes. While there has been work on non-equijoins [24], most of the literature focuses on equijoins because they are common in practice.

The second type of hash algorithm is the *GRACE hash join* algorithm. We will

Table 5.1. Summary of skew in parallel join algorithms.

Skew Type		Description	Point of Manifestation
TPS	Tuple Placement Skew	Initial relation partitioning imbalanced	Before the query starts
SS	Selectivity Skew	Query only selects certain tuples	After Select operator is applied
RS	Redistribution Skew	Relation partitioned unevenly on join attribute	After tuples are redistributed
JPS	Join Product Skew	Join output volume differs between partitions	After local joins are computed

discuss this algorithm in further detail in the following sections, but a quick description of the algorithm is as follows. The relations are partitioned into *buckets* where the number of buckets is much greater than the number of nodes in the cluster. Buckets are partitioned across the cluster in the initial *bucket-forming* stage. Next, in the *bucket-joining* stage, corresponding buckets from each relation are joined locally using a hash method similar to the simple hash join.

The *hybrid hash join* is a combination of simple hash join and GRACE hash join. Hybrid hash join operates like GRACE hash join, except the first bucket is treated separately. Instead of writing the first bucket back to stable storage, an in-memory hash table is constructed from the inner relation and probed by the outer relation as in the simple hash join. Thus the joining of the first bucket is overlapped with the bucket-forming stage for slightly increased performance. The hybrid hash join outperforms the other joins in most situations

5.4.3 Types of Skew

While Schneider and DeWitt [77] characterized several parallel join types, their analysis of the effects of skew was limited. Two years later, Walton, Dale, and Jenevein

[91] constructed a taxonomy of the various types of join skew. They note that there are really two categories of skew with which to be concerned. The first, *attribute value skew*, or *AVS*, is intrinsic to the relations. AVS means that the relations themselves are skewed, for example, with some values occurring more frequently than others. The second broad category of skew is *partition skew*. Partition skew is caused by a join algorithm that poorly partitions the join workload and therefore loses some parallel speedup. Partition skew is possible even in the absence of AVS, so it is not enough to simply know the AVS properties of the joining relations.

Partition skew can be mitigated by using the proper algorithm, so Walton, Dale, and Jenevein focus on this category of skew. They further subdivide it into four separate types of skew shown in Table 5.1. *Tuple Placement Skew*, or *TPS*, occurs when the initial partitioning of a relation across the cluster is skewed. In this case, some nodes have more tuples than others, which causes an imbalance in scan times.

The second type of partition skew is *Selectivity Skew*, or *SS*. SS occurs when the number of selected tuples differs across nodes. An example of SS is a join involving an additional range predicate where the relations are range-partitioned across the cluster. The partitions that cover the selection predicate have many more candidate tuples than those that either partially cover or do not cover the selection predicate.

Redistribution Skew, or *RS*, is the improper redistribution of tuples across the cluster. When RS occurs, different nodes in the cluster hold different amounts of tuples to be joined. A bad hash function, or one that is not properly tuned for the join attribute distribution, can cause RS. Researchers tend to focus on solving RS since it is a direct property of the join algorithm.

The last type of partition skew is *Join Product Skew*, or *JPS*. JPS occurs when the number of matching tuples in the join differs at each node. JPS can be present even in the absence of RS, when all nodes have the same number of tuples before the join.

5.4.4 Solutions

Now that we have introduced several categories of parallel joins and join skew types, we can begin to discuss skew-resistant join algorithms. These algorithms can be generally classified by two broad categories [41]. *Skew resolution* techniques recognize and react to skewed parallel execution. On the other hand, *skew avoidance* techniques take proactive measures to prevent skew from occurring at all. As we will see, each algorithm focuses on solving skew in a particular situation.

Bucket Tuning

Kitsuregawa, Nakayama and Takagi [46] identified a potential skew issue in the single-node versions of GRACE hash join and hybrid hash join. Both of these join algorithms partition the joining relations up into buckets such that each bucket should fit entirely in memory. This avoids bucket overflow, and thus extra I/Os, in the case where all buckets are evenly sized. The problem with this approach is that buckets are statically chosen based on the properties of the joining relations. It can be the case that an upstream operator or a selection predicate applied to the join causes some buckets to be significantly larger than others. If this bucket size skew is severe enough, some buckets may overflow to disk which greatly reduces join performance.

Their solution, called *bucket tuning*, partitions the relations into a very large number of very small buckets, with the goal being that every bucket fits in memory regardless of bucket size skew. Since database systems perform I/O operations at the page level, this method can be inefficient if a bucket is smaller than a page. The bucket tuning strategy addresses this by combining very small buckets into larger buckets so that every bucket is at least one page large. The trick here is that buckets are combined *after* relations are partitioned and selection predicates are applied. The application of bucket tuning to hybrid hash join is called *dynamic hybrid GRACE hash join*, which chooses

buckets dynamically rather than statically.

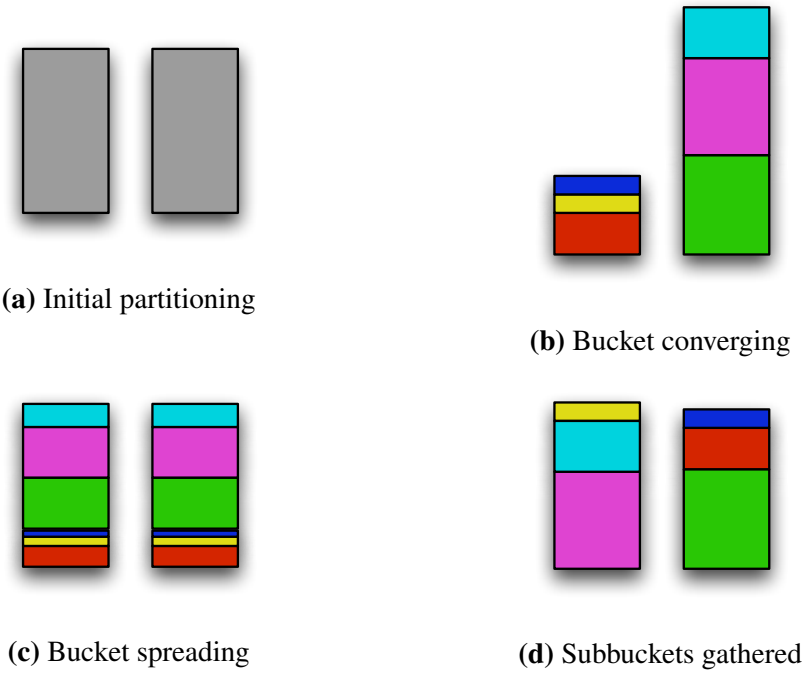
Kitsuregawa, Nakayama and Takagi compare dynamic hybrid GRACE hash join to hybrid hash join under three distributions of join attributes: triangular, Zipf, and uniform. They use the triangular and Zipf distributions to demonstrate the performance of the algorithm under skew. The uniform distribution represents a baseline comparison for the ideal case of no skew. They compute the number of I/Os analytically as a performance metric. Under the skewed distributions, dynamic hybrid GRACE hash join performs better than hybrid hash join. Under the uniform distribution, dynamic hybrid GRACE hash join and hybrid hash join are nearly identical in performance. The dynamic hybrid GRACE hash join algorithm represents an improvement over hybrid hash join, and all future work considers bucket tuning as an essential component of a bucketing hash join algorithm.

Bucket Spreading

While the bucket tuning solution discussed earlier was originally proposed as a single-node algorithm, it can easily be extended to the parallel versions of GRACE hash join and hybrid hash join. Kitsuregawa and Ogawa [47] describe a parallel version of GRACE hash join with bucket tuning. In this algorithm, each node in the cluster holds a subset of buckets after partitioning and performs bucket tuning independently of the other nodes in the cluster.

The parallel GRACE hash join above has the property that tuples destined for a given bucket are read in parallel from the relation partitions and then converge on a single node. Kitsuregawa and Ogawa call this style of parallel join *bucket converging*. In contrast, a *bucket spreading* algorithm will scatter buckets across the cluster by further repartitioning the buckets into *subbuckets*, which are simply bucket fragments.

A bucket converging algorithm suffers from RS. Even though GRACE hash join



(a) Initial partitioning

(b) Bucket converging

(c) Bucket spreading

(d) Subbuckets gathered

Figure 5.1. Relations are initially partitioned on two nodes (a). Bucket converging (b) statically assigns buckets and may create uneven bucket volumes. Bucket spreading evenly divides buckets into subbuckets (c) which can then be gathered into whole buckets evenly (d).

with bucket tuning will create buckets that all fit in memory, there might be a significant difference in the total volume of buckets on each node. The bucket spreading technique combats RS by assigning buckets to nodes *after* bucket sizes are known. Buckets are evenly spread across nodes as subbuckets in the bucket forming stage. A coordinator node then performs bucket tuning and computes an optimal assignment of whole buckets to nodes. Finally, the subbuckets are gathered and joined in the bucket joining stage. Figure 5.1 illustrates the differences between bucket converging and bucket spreading on two nodes.

Bucket spreading can be tricky to implement because it requires subbuckets to be evenly spread, or *flattened*, across the cluster without prior knowledge of how many tuples for each bucket a given node will produce. Kitsuregawa and Ogawa solve this by

utilizing an intelligent omega network, which is a network topology consisting of many 2x2 crossbar switches that can either be crossed or straight. Details are given in [47]. The switches in this network maintain counters that can be used to reassign ports in response to skewed traffic. The network effectively takes skewed input and produces evenly partitioned output.

In the same work, Kitsuregawa and Ogawa evaluate both the bucket flattening mechanism of the omega network and the ability of bucket spreading to cope with skewed tuples. To evaluate bucket flattening, tuples are randomly assigned to buckets using a uniform distribution. Even though the bucket assignment is uniform, there is still small variation in bucket sizes. Using the mean standard deviation of bucket sizes as a metric, they find that without flattening this deviation increases linearly as the number of tuples increases. However, with the bucket flattening omega network, the deviation remains constant as the number of tuples increases, indicating that buckets can be effectively spread evenly across the cluster.

They compare bucket spreading to bucket converging by considering GRACE hash join on Zipf distributions of varying degrees of skew. They use the maximum number of I/Os on any individual node as the performance metric. In a low-skew environment, bucket spreading and bucket converging require roughly the same number of I/Os. However, as skew increases, the bucket converging strategy's I/O count greatly increases while bucket spreading strategy's I/O count increases more slowly. The reason for this increase is that a bucket converging algorithm will yield one node with a significantly higher volume of tuples than the others. This node will require many more I/Os to process all of its tuples. Bucket spreading, on the other hand, evenly distributes the bucket volume across the cluster, and therefore is a highly effective strategy for reducing the performance penalties caused by a Zipf distribution.

Schneider and DeWitt [77] also describe a parallel GRACE hash join algorithm

using a technique related to bucket spreading. In their algorithm, a hash function, as opposed to an intelligent omega network, is used to spread buckets across the cluster. Their goal, however, is not to ensure even bucket spreading, but to gain maximum I/O throughput by spreading the bucket across all disks in the cluster. Indeed it is impossible to guarantee even bucket spreading using a hash function. Consider, for example, the case where all values are the same. In this case, a hash function will assign all tuples to the same subbucket. Nevertheless, the hashing technique has the advantage that buckets no longer need to be collected on a single node for joining. Each node can perform a local join on its subbuckets because matching tuples will map to the same subbucket in both relations.

Partition Tuning

While the bucket tuning technique can be thought of as a way to tune bucket sizes on a given node, the bucket spreading technique can be viewed as a way to tune partition sizes across the cluster. Hua and Lee [41] describe three algorithms using this general idea of *partition tuning*. Two of their algorithms are *skew avoidance* algorithms, meaning they prevent skew from occurring at all. They also include a *skew resolution* algorithm that initially permits skew but then later corrects it.

The first skew avoidance algorithm with partition tuning is the *tuple interleaving parallel hash join*. Tuple interleaving parallel hash join is effectively identical to bucket spreading [47] except that the bucket flattening step is done in software rather than in an omega network. Buckets can be flattened by sending tuples to nodes in round robin order, thereby interleaving them across the nodes in the cluster. Hua and Lee state that processors are fast enough to do this interleaving in software, although it can also be done in hardware components such as specialized CPUs.

The second algorithm is a skew resolution algorithm called *adaptive load bal-*

ancing parallel hash join. As the name indicates, this algorithm adapts to skewed data by redistributing tuples across the cluster after partitioning. Whole buckets are initially hash-distributed to nodes without using any kind of spreading algorithm. After all buckets have been distributed, each node selects a subset of buckets that is close to its fair share of the data and then reports bucket information to a coordinator node. The coordinator node uses global information to compute an optimal reshuffling of excess buckets to equalize data across the cluster. After excess buckets have been reshuffled, each node independently performs bucket tuning and computes joins locally.

Adaptive load balancing parallel hash join has the property that its overhead, i.e. the amount of data is reshuffled, is proportional to the amount of skew in the joining relations. Under mild skew, adaptive load balancing parallel hash join will redistribute only a small number of buckets. Tuple interleaving parallel hash join, on the other hand, requires an all-to-all shuffle of data while buckets are gathered. It therefore pays the full shuffle overhead even under mild skew.

The last algorithm proposed by Hua and Lee is a skew avoidance algorithm called *extended adaptive load balancing parallel hash join*. This algorithm takes the adaptive load balancing parallel hash join a step further by computing the optimal bucket assignment earlier. Relations are initially partitioned into buckets that are written back to local disk without any network transfer. Bucket information is then sent to the coordinator, which computes an optimal bucket assignment. Finally, buckets are distributed according to this assignment and bucket tuning and local joins are performed. This algorithm effectively has no network overhead because buckets are transferred exactly once, so it works well under high skew. However, extended adaptive load balancing parallel hash join still has significant disk I/O overhead since an extra round of reads and writes is required for all tuples before any network transfer can begin.

Hua and Lee model the three algorithms analytically and evaluate them on a

reasonable assignment of parameters for a parallel database system. Because the most skewed node is always the bottleneck, they assume a skew model where all nodes except for one have the same amount of data after partitioning, and the one node has more data after partitioning. As the data distribution varies from no skew, i.e. uniform, to full skew, where one node has all the data, they find that no single algorithm always wins. In the absence of skew, vanilla GRACE hash join performs the best, although it only slightly beats the next best algorithm. This is likely due to the overheads associated with the other algorithms. Under mild skew, adaptive load balancing parallel hash join beats GRACE hash join slightly. As expected, adaptive load balancing parallel hash join redistributes only a small amount of data under mild skew and still manages to equalize partitions, preventing bucket overflow. Under heavy skew, tuple interleaving parallel hash join and extended adaptive load balancing parallel hash join greatly outperform the other algorithms and are both equally good under the analytical model. The model assumes that different parts of the computation within each stage can be overlapped and that disk I/O dominates the running time. While extended adaptive load balancing parallel hash join has a much smaller network footprint, this has no impact on query response time since the network is not the bottleneck given the assumed system parameters.

In the context of the skew taxonomy provided in Section 5.4.3, these three algorithms all focus on solving RS. Any hash-based algorithm that splits relations into buckets has the potential for RS. Tuple interleaving parallel hash join handles RS by breaking the partitioning into two steps. The first step creates small sub-buckets on each node with interleaving, and the second step redistributes buckets evenly across the cluster. Adaptive load balancing parallel hash join mitigates RS by fixing skewed partitions after the fact using bucket redistribution. Extended adaptive load balancing parallel hash join uses a preprocessing bucketing step to compute optimal partitions, so RS never has a chance to occur.

Practical Skew Handling

In 1992, DeWitt et al. [25] proposed a new set of skew-resistant join algorithms. Unlike the previously discussed join algorithms, which were evaluated using analytical models and simulations, these new algorithms were actually implemented on the Gamma parallel database. Additionally, the algorithms are a marked departure from the previously discussed hash join algorithms. Instead of hash partitioning, these algorithms use sampling and range partitioning to avoid skew. They sample whole pages of tuples randomly from the inner relation. The samples are used to construct an approximate distribution of join values from which an even range partitioning can be computed.

The first algorithmic technique discussed by DeWitt et al. is *range partitioning*. The algorithm samples random tuples from the inner relation and constructs an approximate distribution of join values. Next it builds disjoint ranges from this distribution that evenly partition the sampled join values. Tuples are partitioned according to which range they fall in, and then nodes join their assigned tuples locally using standard hash join techniques.

Range partitioning by itself cannot create even partitions in the case when a common value is repeated many times. Consider, for example, the case where all tuples in one relation have the same value on the join attribute. Here, range partitioning will create a single partition containing all tuples. To combat this scenario, DeWitt et al. use a technique called *subset replicate*.

The key insight behind subset replicate is that it is possible to split a set of repeated values up into subsets, as long as all corresponding tuples in the other relation are copied to all partitions. Consider the example above, where all tuples in one relation have the same value on the join attribute. Suppose the other relation contains only a small number of tuples with this join attribute. The large number of repeated values in the

first relation can be split into separate partitions, and the small number of corresponding tuples in the second relation can be copied to all of these partitions. Local joins can be computed on each partition, and their union will be the correct join output.

The simplest implementation of subset replicate splits a large set of repeated values into subsets of equal size. Since the first and last of the resulting partitions may also contain other values, uniform subsets may not actually create uniform partitions. Instead, a technique called *weighted range partitioning* is used. With weighted range partitioning, subset sizes are weighted by how large the resulting partitions will be, so a non-uniform collection of subsets will be used to create a uniform set of resulting partitions.

DeWitt et al. also provide a technique for achieving finer granularity partitions called *virtual processor partitioning*. This technique creates many more partitions than processors and statically assigns them to physical processors in round-robin fashion. This is very similar to GRACE hash join with bucket tuning [46], which creates many more buckets than nodes in order to spread tuples more evenly.

Virtual processor partitioning can alternatively use a dynamic assignment of partitions to processors. DeWitt et al. use the LPT scheduling algorithm to dynamically compute the assignment as an alternative to round robin.

They evaluate four algorithms: range partitioning, weighted range partitioning, virtual processors with round robin, and virtual processors with processor scheduling on the Gamma database system. Each algorithm implements subset replicate in its internal data structures. They compare to hybrid hash join as a good baseline. The hybrid hash join has the best performance on unskewed data due to low overhead, but its performance quickly degrades. Weighted range partitioning works quite well for mild skew, and virtual processing with round robin works well for moderate to heavy skew.

Because they only sample the inner relation, the proposed algorithms do not work

well when the outer relation is skewed. In this case, hybrid hash join outperforms the other algorithms due to lower overheads. The other algorithms effectively sample the wrong relation and do not get a chance to learn about the skewed data.

An interesting result is that the performance of the algorithm with the widest range of applicability, virtual processing with round robin, is roughly independent of the number of tuples sampled. DeWitt et al. state that the performance gain due to more accurate sampling is offset by the performance loss of actually sampling the data.

In the context of the skew taxonomy, range partitioning with sampling is a technique that mitigates RS. An algorithm that samples can more accurately create even partitions. Weighted range partitioning is an improvement that further reduces RS. Virtual processor processing can also help to create even partitions. However, DeWitt et al. motivate it as a solution to Join Product Skew, or JPS, as discussed in Section 5.4.3. If there is a mild to moderate amount of skew in both relations, even weighted range partitioning might not be able to separate the repeated values into enough partitions. In this case, there aren't enough repeated values to cause the subset replicate mechanism to spread the values across all nodes. However, a small number of repeated values in both relations can in the worst case cause a quadratic blowup in the magnitude of the join result. This is, by definition, JPS, and it can be solved by forcing the small number of repeated values to be spread across all nodes in the cluster with virtual processing partitioning. Even if there are only a few virtual processors per physical processor, there will still be enough partitions to evenly spread the data across the cluster.

Skew Handling in Sort-Merge Join

All of the algorithms discussed so far have been hash join algorithms. As mentioned in Section 5.4.2, there are other types of joins. Li, Gao, and Snodgrass [54] present several refinements to the sort-merge join style algorithm that improve skew

resistance. Unlike many of the works discussed so far, this work does not focus on parallel algorithms, but rather focuses on the skew resistance techniques themselves.

Li, Gao, and Snodgrass are quick to mention that typical hash join algorithms suffer from bucket overflow in the presence of skew. While vanilla sort-merge join also suffers from skew, it has desirable performance properties when more than two relations are involved because the intermediate result relations are already sorted and can therefore skip the sort step. They are primarily interested in making sort-merge join skew-resistant so they can take advantage of these performance properties.

An implementation of vanilla sort-merge join may require tuple rereads in the presence of skew. This is essentially the same problem faced by nested-loop joins. If a value is repeated in both relations, its corresponding tuples in the inner relation will have to be reread once for each matching tuple in the outer relation. Depending on the implementation, the I/O cost can be enormous.

A typical optimization to the above problem is block orientation. Li, Gao, and Snodgrass present a block oriented algorithm called *R-1*. R-1 reads a block of tuples at a time from disk. If the inner relation is skewed and the skew is contained within a block, no extra I/Os are required since the repeated values already exist in memory. If the skew crosses block boundaries, all inner relation blocks containing tuples for a particular value may need to be *reread* for each matching tuple in the outer relation if. These rereads will be required in the case where the older blocks have been evicted due to memory pressure.

Here we note that in a modern system with large amounts of memory, the operating system may be able to keep old blocks in a buffer cache and prevent the rereads from touching disk. However, Li, Gao, and Snodgrass are primarily interested in special purpose database systems that tend to manage memory from within the application. The database system can use application-specific knowledge to make better use of memory than a traditional operating system using LRU replacement. Thus it is useful to consider

application-level memory management techniques such as block rereading.

An alternate implementation of R-1 operates on multiple sorted runs per relation instead of a single sorted run. This algorithm is called *R-n*. R-n overlaps the join operation with the last phase of the merge-sort for extra efficiency. This efficiency comes at the cost of a trickier implementation. R-n also has the possibility of incurring more random reads since skewed tuples may be spread across multiple sorted runs.

R-1 and R-n require block rereads for every matching tuple in the outer relation. An improvement over R-1 is *BR-1*, which joins every tuple in a block in the inner relation to the entire block in the outer relation. This strategy is analogous to loop tiling optimizations used to improve cache performance [96]. BR-1 will only incur inner relation rereads every time a new *block* in the outer relation is read, rather than once per *tuple*. The *BR-n* algorithm similarly extends BR-1 to handle multiple sorted runs.

An improvement on BR-n is *BR-S-n*, which does block rereads but makes smarter use of memory. When skew in both relations is detected, the previously joined values in the memory-resident blocks are discarded and the tuples corresponding to the current join value are shifted to the top of memory. This has the effect of fitting more of the joining tuples in memory, which reduces the number of reread I/Os.

In the same work, Li, Gao, and Snodgrass discuss a different strategy called *spooled caching* that makes good use of memory. In the *SC-1* algorithm, tuples that satisfy selection predicates, but may or may not actually be joined, are stored in an in-memory cache. This possibly prevents rereads since a tuple is only placed in the cache if it has the possibility of satisfying the join condition. If the number of such tuples is small, the cache may be able to hold all of them. On the other hand, if the cache overflows it is spooled to disk, so rereads are still required under heavy skew. In this case, only those tuples that can satisfy the join condition will be reread, so this is still an improvement over rereading everything. As an optimization, tuples from the inner

relation are immediately joined with the current block of the outer relation, and if skew is detected in both relations, those tuples are added to the cache. As one might expect, the *SC-n* variant implements spooled caching for multiple sorted runs. It is more complicated and requires caching if skew is detected in any one of the sorted runs in the outer relation.

The last algorithm is *BR-NC-n* which is similar to BR-n but uses a cache that is *not* spooled to disk. Instead, when the cache fills up, blocks are simply reread as in BR-n. This algorithm has the slight advantage over BR-n in that if the number of skewed tuples is small enough to fit in the cache, but larger than a single block, no rereads will be required. Compared to SC-n, BR-NC-n must reread more tuples since it rereads from the actual relations. However, BR-NC-n does not need to flush cache blocks to disk.

Li, Gao, and Snodgrass evaluate their algorithms on the TimeIt [48] database prototyping system. They measure performance under two types of skew which represent extremes over of the spectrum of distributions. The first type, *smooth skew*, is a lightly skewed distribution where some join values have two tuples and the rest have one. The second type, *chunky skew*, has a single join value with a large number of tuples, and the rest of the values have a single tuple. They use the same relation for both sides of the join, so skew is present in both the inner and outer relations.

The experiment for smooth skew shows that SC-n, BR-NC-n and BR-S-n are all good algorithms, with SC-n being slightly better than the others. R-n and BR-n are both a little worse than the others, although the slowest algorithm is only 11% slower than the fastest. This difference is likely due to the fact that R-n and BR-n will have to perform rereads if one tuple is in one block and the other tuple is in the next block. The other algorithms use memory tricks to avoid paying this I/O cost. The *-1* algorithms that operate on a single sorted run from each relation are uniformly worse than their multiple-run *-N* counterparts due to the overlapping of the merge and join steps, so they are not evaluated.

The results are more startling for chunky skew. In this case, all of the multiple-run algorithms are roughly the same except for R-n, which is several times worse than the others. When a single join value has many tuples, R-n requires multiple rereads for each of the tuples in the corresponding tuples in the outer relation. In the presence of chunky skew, these rereads dominate the running time of the algorithm.

In the absence of skew, all of the algorithms are roughly the same. This indicates that the bookkeeping overhead of the more sophisticated algorithms is negligible. While it would be interesting to see a comparison of the memory usages of each algorithm, the authors fix the memory size at 16MB and each algorithm uses all of the available memory. Differences in algorithmic performance might manifest on systems with larger memories, but this is outside the scope of [54].

While the above algorithms are not parallel join algorithms, we can still glean some key insights. The algorithms essentially focus on solving a problem related to JPS. When the join product is very large, additional I/Os may be required in systems without adequate memory. These I/Os can dominate the computational time of the join if care is not taken when selecting the join algorithm. Li, Gao, and Snodgrass show that it is feasible to design a sort-merge join algorithm that avoids most of these I/Os in the presence of skew in both relations.

Partial Redistribution Partial Duplication

While many skew-resistant join algorithms have been discussed so far, Xu et al. [102] from Teradata published a paper in 2008 that states that these algorithms are too difficult to implement correctly in practice. As a result, parallel database software simply does not handle skewed data well. They discuss a simple algorithmic modification that mitigates skew and is practical to implement in real software.

Xu et al. characterize two types of parallel hash join strategies. The first type,

called *redistribution*, involves redistributing tuples in the joining relations based on a hash of the join attribute. This is very similar to the simple hash join algorithm presented by Schneider and DeWitt [77] and discussed in Section 5.4.2. As mentioned before, redistribution suffers a performance penalty when there is intrinsic skew in the join attribute of one or both relations. In this case, some nodes will receive more data than others. Adding more nodes does not help much, and in fact increases the degree to which a *hot node* is overloaded relative to the other nodes.

The second strategy, *duplication*, works well when one relation is much smaller than the other. With duplication, the smaller relation is copied to all processing nodes, which then can perform local joins between this copy and their portion of the larger relation. Duplication does not suffer from hot spots in the same way redistribution does, since the relations are never partitioned on the join attribute. However, significant network and storage I/O costs are required if neither relation is very small.

Redistribution and duplication can be combined into a hybrid algorithm called *partial redistribution partial duplication*. This algorithm attempts to gain the benefits of both techniques by redistributing some tuples and duplicating others. Skewed tuples, i.e. a set of tuples with the same join attribute, are stored locally at partition time. The tuples in the other relation with the corresponding join attribute are duplicated to all nodes. Non-skewed tuples are redistributed using a standard hash function. After this partitioning, each node computes three joins and unions them together: redistributed tuples with redistributed tuples, locally skewed tuples with duplicated tuples, and duplicated tuples with locally skewed tuples.

Xu et al. evaluate partial redistribution partial duplication on a cluster of 10 nodes, each hosting 8 virtual processing units. They compute a join on two relations and artificially set the join attribute in one of the relations so that a large fraction of the tuples contain the same join attribute. As this fraction varies from 0% to 40%, the

traditional redistribution algorithms suffers linear slowdown, while partial redistribution partial duplication's performance remains constant.

The baseline redistribution algorithm is related to the simple hash join and GRACE hash join algorithms. All three of these algorithms suffer from RS. In order to avoid redistributing too many tuples to a given node, Xu et al. apply fine-grained duplication to only those problematic tuples. Because the skewed tuples are not redistributed, the initial tuple placement determines which nodes must join these tuples, so a vanilla implementation of partial redistribution partial duplication can suffer from TPS. To avoid TPS, the algorithm is modified to randomly redistribute the skewed tuples, thus spreading them evenly across the cluster at the cost of slightly higher network and disk I/O. Because the corresponding tuples in the other relation are duplicated everywhere, this modified algorithm is correct and avoids hot spots caused by TPS.

Outer Join Skew Optimization

All of the previously discussed algorithms focus on inner joins. Xu and Kostamaa [101] solve the issue of skew in outer joins, which are prevalent in business intelligence tools.

Outer join skew manifests itself in the computation of multiple joins. Even if the first join does not suffer from any skew problems and its output is evenly distributed across the cluster, outer join skew can be a problem. If the subsequent join attribute happens to be the same as the first join attribute, any dangling rows will contain `NULL`s on this join attribute. Since the subsequent join is partitioned on its join attribute, these `NULL`s will be redistributed to the same node. This node will then contain many more tuples than the other nodes and will bottleneck the parallel join.

The algorithm given by Xu and Kostamaa in this work, called *Outer Join Skew Optimization*, or *OJSO*, effectively handles skew caused by outer joins. OJSO treats

tuples with NULLs as a special case. A tuple containing a NULL on the join attribute is saved locally, while all other tuples are redistributed as normal. Local joins are computed only on the redistributed tuples. The final result is the union of the joined output with the locally saved tuples containing NULLs. OJSO's output is correct because tuples containing NULLs cannot join with other tuples by definition, so they need not be redistributed.

Xu and Kostamaa evaluate OJSO on a cluster of 8 nodes where each node hosts 2 virtual processing units. They measure the execution time of a three-way join and vary the fraction of dangling rows, i.e. tuples containing NULLs, from 0% to 70%. They find that OJSO's performance is constant as outer join skew increases, whereas a conventional outer join algorithm slows down linearly with the amount of outer join skew.

Since all previously described algorithms have been inner join algorithms, it is a little challenging to compare OJSO with the others. The technique of treating some tuples as a special case is related to Xu et al.'s treatment of skewed tuples in partial redistribution partial duplication [102]. In the partial redistribution partial duplication algorithm, skewed tuples are saved locally and their corresponding tuples in the other relation are duplicated, so a join can be computed. While OJSO does not require duplication, it still uses the technique of saving some tuples locally rather than redistributing them.

At its heart, the problem of outer join skew is a form of RS that manifests in joins computed after the first. In this case, the NULLs are unevenly redistributed across the cluster, causing hot spots. However, one can also view this as a variant of JPS, since NULLs effectively do not join with anything so any partition receiving NULLs will have a different join product size than a partition that does not receive NULLs.

5.5 Skew in MapReduce Systems

Next, we describe several techniques for mitigating skew in MapReduce-like frameworks. In addition to providing a parallel computation framework, MapReduce

integrates with a distributed file system such as Google File System [31], or the Hadoop Distributed File System, HDFS [104]. These storage solutions differ from a traditional relational database in that they offer access to unstructured files, rather than adhering to schemas. Relational databases also offer access to indexes that are absent in MapReduce systems. Finally, since the `map` and `reduce` functions are *User Defined Functions*, or *UDF*s, automatic optimizations are nontrivial. These differences necessitate skew-resistant solutions that are slightly different from the parallel join solutions presented earlier. This section will survey skew mitigation techniques in MapReduce systems and how they relate to each other and the previously discussed parallel join techniques

5.5.1 Types of Skew

MapReduce clusters are typically built from large numbers of unreliable commodity components. The degree of hardware unreliability in itself is a type of skew that MapReduce tackled from its inception. For example, if a node in the cluster has a faulty disk drive, it may write `map` output to disk at a much slower rate than its healthy counterparts. If the job is partitioned evenly across nodes, this slow node will take much longer to accomplish its task and may become a bottleneck for the entire job. Dean and Ghemawat identified these slow nodes as *stragglers* [19].

A typical strategy for handling skew caused by stragglers is *speculative execution* [19, 104]. A *backup copy* of a long-running task is speculatively executed on another node before the original task finishes. If the backup copy finishes first, the original task is killed, and vice versa. The hope with this strategy is that the task was slow because of faulty hardware, and so the backup on the newly selected node will finish before the original task because it will *not* have faulty hardware with high probability.

In addition to hardware-related skew, MapReduce can also exhibit skew in the data or computation. Skew manifests itself differently depending on the phase of the

MapReduce job. Kwon et al. [51] identified three types of *map-skew*. The first is a type of computational skew called *expensive record skew*. When expensive record skew is present, some records take significantly longer to process than others. A good example of this is a MapReduce implementation of the PageRank [65] graph analysis algorithm. In PageRank, there are two kinds of records: small contribution records and huge graph structure records. The presence or absence of these structural records in an input partition can greatly skew the processing time for an individual `map` task.

Another type of map skew is *heterogeneous map skew*. A heterogeneous map reads multiple input sources in a single task. An example of a job with heterogeneous maps is CloudBurst [62], a DNA alignment algorithm modeled on RMAP [80] that attempts to align a set of reads to a reference genome. The `map` logic treats reads and reference records differently, yielding bimodal performance characteristics. In this case, it can be difficult to create evenly partitioned `map` tasks without application-specific knowledge.

A third type of map skew is *non-homomorphic map skew*, which occurs when the `map` function must operate on a group of records rather than processing them one-by-one in a streaming fashion. These `map` functions performs `reduce`-like logic. Some clustering algorithms used in scientific computing fall into this category.

Kwon et al. also identified two types of reduce skew. The first is *partitioning skew*, which is akin to RS in parallel databases. Under partitioning skew, the intermediate files that result from the `map` output are unevenly partitioned across the `reduce` tasks in the cluster. This can be caused by a bad hash function that unevenly partitions the `map` output. Even with a good hash function, however, duplicate keys can cause some partitions to be larger than others since the semantics of `reduce` dictate that all records of a given key must be present in the same intermediate file. Unfortunately it is generally impossible to predict the intermediate key distribution without at least partially running

the `map` function because `map` is a UDF and is entirely application-specific.

The second type of reduce skew is *expensive record skew*. This skew is more severe than its `map` counterpart due to the fact that an invocation of `reduce` operates on collections of records rather than individual records. If the `reduce` function must perform any comparisons between the values associated with a key, the processing time will be super-linear in the size of the record, creating a significant source of computational skew. The degree to which this and other types of skew affect a MapReduce job depends of course on the particular application and input data.

5.5.2 Solutions

In Section 5.5.1 we discussed several types of skew that occur in MapReduce systems. Now we will present solutions proposed by the MapReduce community that tackle these various types of skew.

LATE

While speculative execution as discussed above addresses the problem of stragglers, it does have some shortcomings. Speculative execution, as implemented in Hadoop [104], causes a task to be duplicated toward the end of the job if its measured progress is a below some threshold. There are several problems with this approach. The first is that a task might be intrinsically slow, so a backup copy may not actually help reduce the job's completion time. A second problem is that outlier tasks are not identified until they have already run for a long period of time. By the time a task is identified as slow, it may have already wasted significant cluster resources.

Zaharia et al. [108] present an improved scheduler for Hadoop called *Longest Approximate Time to End*, or *LATE*. LATE uses a more intelligent speculative backup mechanism than the stock Hadoop scheduler. In particular, it is designed to handle the

case of clusters containing heterogeneous hardware configurations. Hadoop's default backup mechanism schedules backup copies of tasks that are some amount slower than the average currently running task. In a heterogeneous environment, any task running on older hardware will be considered too far below the average and will be speculatively executed, leading to significant resource waste. Zaharia et al. state that heterogeneous environments are actually the common case as older hardware is incrementally replaced, or in a virtualized environment where customers compete for shared resources.

LATE, like Hadoop, uses heuristics to determine when tasks should be speculatively executed. However, Hadoop's mechanism is based on a `progress score`, which varies from 0 to 1 and roughly corresponds to the fraction of the task that has been completed. Since the progress score increases over time, Hadoop can only schedule backups if a task fails to make progress long enough for it to be noticed as slower than average. More concretely, Hadoop will consider a task as slow if its `progress score` is less than 0.2 below the average. LATE, on the other hand, uses a `progress rate` metric, which is defined as `progress score / T` where `T` is the elapsed time of the task. Using the progress rate metric allows LATE to notice immediately when a task is progressing slower than it should be. In particular, LATE estimates the task's time to completion as `(1 - progress score) / progress rate`, and then uses this metric to decide which tasks to backup first. The intuition is that the task that will finish furthest in the future has the most potential for the speculative backup to overtake the original task and improve the job's overall completion time.

Zaharia et al. evaluate LATE both on a large cluster of Amazon EC2 nodes and on a small, local testbed. They compare LATE with Hadoop with and without speculative execution. They test their configuration using sort, grep and word count as application benchmarks. They observe cluster heterogeneity, and therefore hardware-related skew, by measuring the number of virtual machines per physical machine. LATE is up to a

factor of two better than Hadoop with speculative execution. In some cases, speculative execution actually decreases the performance of Hadoop.

Mantri

While LATE offers a significant improvement over plain speculative execution, it does not go far enough to solve the problem. If an outlier task occurs early on in the job, LATE will not be able to detect it because speculative execution does not occur until the end of the job. Furthermore, the only course of action LATE can take is speculative backup, which may not be the most efficient response, especially if the problem is not directly related to the slow node.

Ananthanarayanan et al. [5] solve the problems of speculative execution with a system called *Mantri*. Mantri is an intelligent outlier-response system that actively monitors tasks in a cluster and takes action dependent on the identified cause of slowdowns. Rather than using a one-size-fits-all approach, Mantri performs a cost-benefit analysis at the task level to determine whether to take action or not. It also acts early, which prevents early outliers from slipping through the cracks.

Mantri uses two methods of task-level restarts. The first, called *kill and restart*, kills a task that is identified as an outlier and restarts it on a different node. Kill and restart has the benefit of not requiring an extra task slot, but requires that the restart must actually save time with high probability. Another method is *duplicate*, which schedules a backup copy much like speculative execution. The duplicate method uses the original task as a safety net in case the backup copy does not actually save time. In this case, Mantri will notice that the backup is also slow and will kill it off. Any progress made by the original task is maintained because it was duplicated and not killed.

When scheduling tasks, Mantri uses network-aware placement to try to prevent hot spots in the network. The main network-bottleneck that needs to be avoided is the

shuffle phase that occurs before a `reduce` task can begin. Since Mantri knows about `map` outputs, it can intelligently assign `reduce` tasks to racks in order to prevent downlink congestion on racks hosting `reduce` tasks.

Another problem caused by hardware failures is recomputation. When a task's output is used by a later task, such as a `reduce` task using `map` output, it can be the case that the node storing this output fails after the first task completes but before the second tasks starts. In this case, the intermediate data files are lost and must be recomputed. Mantri reduces the penalty of recomputation by selectively replicating intermediate data if the probability of machine failure causes the cost of recomputation to exceed the cost replication. Mantri uses failure history over a long period of time to compute this probability. Additionally, if a machine does fail, all of the tasks that had output stored on the machine are proactively recomputed to reduce delay when these files are eventually requested later in the job.

The final component of Mantri is a scheduler for long-running tasks. Unlike MapReduce with speculative execution, which will schedule backups of long running tasks, Mantri will leave long tasks alone as long as they are making progress at a sufficient rate based on their input size. Mantri also schedules long tasks early, which bounds the overall completion time of the job.

Ananthanarayanan et al. evaluate Mantri on a Bing cluster consisting of thousands of servers. They compare jobs running with Mantri to prior versions of the jobs running before Mantri was enabled using a simulator. They also evaluate Mantri on some benchmark applications. They evaluate Mantri's outlier mitigation strategies and compare them to Hadoop, Dryad, and LATE. They find that Mantri significantly improves the completion time of actual jobs and is noticeably better than the other outlier mitigation algorithms.

Compared to previous works, Mantri is a very sophisticated solution to the

problem of hardware-skew in MapReduce systems. Speculative execution is about the simplest possible solution, and its effectiveness is limited. LATE is more sophisticated, but can still only take one possible action. Mantri, on the other hand, uses cost-benefit analysis to determine which response will likely be the most effective. As a result, it outperforms the other systems.

While Mantri is pitched as a solution to the problem of outliers caused by hardware, it actually is intelligent enough to cope with partition skew. Mantri's ability to take task input size into account when determining outliers allows it to tolerate skewed partition sizes. Consider, for example, a `reduce` task with an abnormally large partition size. Basic speculative execution will identify this task as a straggler and will schedule a backup. While the task technically is a straggler, the backup has no hope of overtaking the original since the task is inherently slow due to data partitioning skew. Mantri will leave the task alone, which is the correct course of action. While Mantri does not proactively fix partitioning skew, it does not perform poorly in the face of such skew.

SkewReduce

While the above systems focus on hardware-related skew, there are several other types of skew that impact MapReduce systems. One such type of skew is *computational skew*. An application exhibits computational skew if some records take longer to process than others. Such skew can manifest even the data is evenly partitioned.

Kwon et al. [50] solve a particular type of computational skew that arises in scientific computing using a system called *SkewReduce*. Many scientific computing algorithms perform some kind of feature extraction using clustering on multidimensional data. Kwon et al. cite the example of searching astronomical imaging data for galaxies. The amount of work required to recognize a galaxy depends on how close the data points are to each other. In this sense, two partitions that contain the same number of points

may have vastly different processing times if one is sparse and the other is dense.

SkewReduce is a feature-extraction framework built on top of MapReduce. Like MapReduce, SkewReduce allows users to write application logic as if it were to be executed on a single processor. SkewReduce then automatically executes this logic in parallel to eliminate the burden of writing parallel code. The SkewReduce API consists of three functions: `process`, `merge`, and `finalize`. The `process` function takes as input a set of data points and outputs a set of features, along with any data points that are no longer needed. Next, `merge` combines feature sets from previous `process` or `merge` executions to create a new one and possibly throws out more data points that are no longer needed. Lastly, the `finalize` function takes the fully merged set of features and applies it to any data points if needed, for example by labeling points with their cluster information. These functions are implemented as MapReduce jobs in Hadoop.

In order to efficiently handle computational skew, SkewReduce applies two user defined cost functions to a small sample of the input data. The user is required to supply functions that estimate the cost of `process` and `merge`. The optimizer then takes the sample data and creates an even partitioning in a greedy fashion by repeatedly splitting the most expensive partition. Each partition initially starts as a `process` job but becomes a `merge` job when split. The optimizer uses the cost functions to find the optimal splitting point, and also to determine if splitting a partition improves execution time. Partitions are split until every partition fits in memory and splitting does not further reduce execution time.

Kwon et al. evaluate SkewReduce on an 8 node Hadoop cluster where each node also serves HDFS. They use the LPT scheduling algorithm and give cost functions that compute the sum of squared frequencies over a histogram of the sampled data. They compare SkewReduce's optimizer to a query plan using uniform partitioning with varying partition granularity. For reference, they also manually tune a query plan by hand.

SkewReduce's optimizer beats all other query plans, and beats the uniform query plans by more than a factor of 2. They also find that a sample rate of 1% is sufficient to reduce computational skew and increase performance.

SkewReduce is unlike other skew-resistant solutions in that it focuses on computational skew, whereas most other solutions focus on hardware-related skew or data skew. SkewReduce's ability to cope with skew stems from its use of input data sampling, a popular technique used by DeWitt et al. [25] that was discussed in Section 5.4.4. SkewReduce is somewhat unique in that it does not apply the actual `process` and `merge` functions to the samples, but rather uses the user defined cost functions to more efficiently probe the space of partitions.

The scientific computing problems that SkewReduce tackles have unusual MapReduce implementations that exhibit non-homomorphic skew as classified by Kwon et al. [51]. Essentially, a `map` function that does clustering does not stream records, but rather computes a result based on large collections of records. The properties of these records relative to others in the collection is the central cause of computational skew. In particular, how close or far one data point is from another in a multidimensional space determines how long such a `map` function will take to execute.

Scarlett

Another type of skew that is somewhere between hardware-related skew and data skew is *content popularity skew*. Ananthanarayanan et al. [4] address popularity skew with *Scarlett*, which is an augmentation to existing distributed file systems such as Google File System or HDFS. Content popularity skew occurs when certain files on distributed storage are more popular than other files and are accessed more frequently. It is not quite data skew, since any individual MapReduce job may not be skewed. However, the collection of all jobs executing simultaneously on a cluster may access some pieces of

data more than others. Similarly, it is not quite hardware-related skew since all machines may be equally fast, but some machines will become hot spots simply due to increased data demand. Such skew arises in practice in log processing, for example, when some logs are interesting and the rest are not.

Scarlett uses the temporal locality of data to predict popularity skew. Every 12 hours, the prior 24 hours of concurrent accesses to a given file is used to update its predicted popularity. Scarlett then adjusts the replication factor on a file-by-file basis to reduce contention from future concurrent accesses. In order to prevent too many tasks from being scheduled a single rack, Scarlett departs from existing replication schemes and spreads new replicas evenly across racks in the cluster. In particular, the new replica is created on the least loaded machine on the least loaded rack. This heuristic prevents popular files from reducing the effective availability of other files stored on other machines in the same rack.

Increasing the replication factor for a file causes extra network traffic that can interfere with running jobs and cause even more contention. To prevent such contention when a large number of replicas need to be added, Scarlett begins by adding a small number of replicas and exponentially increases the number of new replicas as new source racks become available. In addition, Scarlett uses compression to trade spare computational resources for extra network bandwidth.

Ananthanarayanan et al. evaluate Scarlett in two environments. They implement Scarlett as an extension of HDFS in Hadoop and run some benchmark Hadoop jobs driven by trace data from a Dryad cluster. Additionally, they run the actual trace data through a Dryad simulator to confirm Scarlett's ability to improve real workloads. They find that Scarlett speeds up the median Hadoop job by more than 20%, and it speeds up the median Dryad job by more than 12%. Additionally, they measure the network overhead imposed by Scarlett to be less than 1% of the total traffic.

Since Scarlett solves popularity skew, which is rather unique in nature, it is difficult to compare to other systems. Scarlett is most similar to techniques that improve read locality in MapReduce, such as delay scheduling [106] and Quincy [43]. These systems both focus on the locality and fairness. While Scarlett is primarily concerned with skew mitigation, it does so by increasing data locality. In terms of the skew already identified, popularity skew is related to the straggler problem caused by faulty hardware. A storage server in a distributed file system that contains a hot block will effectively be overloaded and will offer lower per-task throughput than a server that only serves cold blocks.

When combined with systems such as speculative execution or the kill-and-restart semantics of Mantri [5], tasks that read data from one of these hot storage serves will be considered slow and the system may try to schedule a backup. In this case, backups will actually reduce overall job performance since new tasks must compete for limited read bandwidth. The correct course of action when a server is overloaded due to content popularity skew is not to start new tasks, but to fix the root problem by increasing replication in the distributed storage system. Increasing replication across the board is far too expensive to be useful in practice, so systems like Scarlett represent a good compromise. Hot files will benefit from increased replication, whereas cold files will maintain the minimum number of replicas to satisfy a given availability guarantee.

SkewTune

Handling data skew and computational skew in vanilla MapReduce is a challenging task since the system is driven entirely by user defined functions. The user must have expert knowledge of the application workload and then must design an ad-hoc partitioning method to mitigate skew. Kwon et al. [52] solve this with an alternative MapReduce implementation called *SkewTune*. SkewTune effectively solves the issues

of data and computational skew in many circumstances and does so without altering the MapReduce API. It therefore enables users to run unmodified MapReduce programs and achieve the performance benefits of ad-hoc partitioning without all of the costs.

SkewTune solves data skew and computational skew by repartitioning straggler tasks into smaller tasks that can be spread across the cluster. If the `map` and `reduce` functions operate independently on individual records and key groups respectively, repartitioning a task's input will guarantee the output stays the same. Put another way, tasks can be safely repartitioned as long as `map` and `reduce` are *pure functions* without side effects. Furthermore, by using range partitioning as the repartitioning mechanism, SkewTune leaves the ordering unchanged as well, so the final output is identical to execution without SkewTune.

After all tasks have been scheduled, SkewTune begins monitoring for tasks to repartition. When a slot frees up, SkewTune determines the task with the longest remaining time. It carves up the task's remaining input into many small disjoint intervals. It then assigns some of these intervals to a task to run on the free slot. SkewTune uses estimates of remaining time to guess when other tasks will finish and prepares intervals for them to process as soon as they free up. In this way, a task is effectively repartitioned across the entire cluster, which allows for maximal slot utilization.

Kwon et al. evaluate an implementation of SkewTune in Hadoop on a 20 node cluster where each node also functions as an HDFS storage server. They evaluate SkewTune on three applications: inverted index, PageRank, and CloudBurst. They find that SkewTune achieves high performance even in the presence of poor configuration settings. If, for example, the user asks for too few reduce tasks, SkewTune can compensate by repartitioning tasks. SkewTune also reacts to heterogeneous map tasks in CloudBurst by splitting tasks that process the more expensive reference data set.

Unlike the systems examined so far, SkewTune does not solve stragglers by

scheduling backups [19, 108, 5], or by killing and restarting tasks [5]. Rather, SkewTune splits long-running tasks into multiple pieces while saving the partially computed output. SkewTune is therefore able to avoid wasted computation entirely while spreading skewed tasks evenly across the cluster.

It is unsurprising, given the authors, that SkewTune solves the types of skew mentioned above. Kwon et al. [51] earlier categorized the types of skew that occur in MapReduce systems. SkewTune directly attacks the problems of expensive record map skew and heterogeneous map skew, as well as partitioning reduce skew and expensive record reduce skew. The fifth type of skew identified in [51], non-homomorphic map skew, is not solved by SkewTune. In fact, SkewTune will not operate correctly if the `map` function is non-homomorphic. To see why, consider a `map` function that operates on a collection of records rather than on individual records. In this case there is no obvious way to split a `map` task's input and still retain the correct output since some output may be dependent upon the interaction of one record in the first split and another record in the second split. Any attempt to repartition a `map` task could therefore change the job's output, so SkewTune cannot be used as a drop-in replacement for MapReduce in this case.

The techniques employed by Kwon et al. are similar in nature to the parallel join techniques proposed by Hua and Lee [41]. Specifically, the adaptive load balancing parallel hash join algorithm reacts to skewed partitions by redistributing buckets across the cluster to create an even partitioning. Kwon et al. take this a step further by not only redistributing tasks, but also by splitting them into smaller pieces for load balancing.

SkewTune, like speculative execution [19] and LATE [108], takes action towards the end of a job. Kwon et al. are primarily focused on making sure every slot in the cluster has a task to process, and acting towards the end of the job satisfies this condition. However, as discussed by Ananthanarayanan et al. [5], such a system will miss outlier

tasks that are scheduled towards the beginning of the job. It might be possible to adapt SkewTune to act earlier and possibly reduce job completion time using the lessons learned from Mantri, although this is not discussed in [52].

A Study of Skew in MapReduce

In Section 5.5.1 we listed several types of skew in MapReduce identified by Kwon et al. [51]. In their paper, the authors discuss five "best practices" for building MapReduce applications that are resistant to skew. Here we will discuss these best practices and how they relate to the solutions presented so far.

The first piece of advice given by Kwon et al. is to avoid the default hash-partitioning map output scheme in Hadoop. Instead, users are directed to use range partitioning or some application-specific ad-hoc partitioning scheme. It is incredibly difficult to create a skew-resistant algorithm using hash partitioning [46, 47, 41, 25, 102, 101, 52]. SkewTune [52] uses range partitioning to achieve a more even data split and to facilitate repartitioning. Kwon et al. mention that ad-hoc partitioning schemes may be required in the case of a *holistic* `reduce` function, which is a `reduce` function that buffers records in-memory and makes comparisons across key groups and is therefore dependent on the data distribution. An example is CloudBurst's `reduce` function, which compares the values from key groups in the reference and read data sets. An ad-hoc partitioning function might be able to use application-specific logic to limit the degree of skew, although this requires significant domain expertise from the user.

A second suggestion is to experiment with different partitioning schemes either between runs or dynamically during a run. In the first case, a production job might have smaller debug runs, or even large production runs with accompanying log data. The logs from these previous runs can inform the framework how it ought to adjust the partitioning function to create more even partitions. In the second case, previous log data may not be

available, but if the partitions can be adjusted dynamically then it may still be feasible to evenly distribute the data. From the partition tuning work discussed earlier [41], the adaptive load balancing parallel hash join algorithm can react to a bad partitioning by redistributing buckets from overloaded nodes to underloaded nodes. In the context of MapReduce, SkewTune [52] effectively implements this solution by repartitioning large partitions dynamically without any previous log data.

The third solution proposed by Kwon et al. is the use of a `combiner` function. A `combiner` implements `reduce` logic on the map output file before data is shuffled across the network. This has the effect of reducing network traffic and consequently shrinks the sizes of all reduce partitions. By reducing the size of the data, a `combiner` reduces the severity of skew and improves performance. Kwon et al. are quick to mention that a traditional `combiner` task will use extra CPU and disk resources, so it is best to implement `combiner` logic inside the `map` function. It should be noted, however, that it may not be feasible to use a `combiner` if the `reduce` function is not commutative and associative since the `combiner` will be applied to a collection of unrelated partition fragments as opposed to one whole partition.

While there is no direct analogue of a `combiner` for parallel join algorithms, most database optimizers apply selection and projection operations as early as possible in order to reduce the amount of data passed to the join operator. The rationale for this optimization is essentially the same as the reason for using a `combiner`. The SC-1 and SC-n algorithms proposed by Li, Gao, and Snodgrass in their work on sort-merge joins [54] make use of a cache of tuples that satisfy additional join predicates. The cache effectively allows the algorithm to skip over uninteresting tuples when joining new tuples with old tuples. This optimization has the same spirit as a `combiner` in that the size of the data is reduced early on in order to avoid additional overhead and reduce the impact of skew later in the algorithm.

The fourth recommended practice is a preprocessing step that is applied before a long-running job actually executes in production. This is especially useful if it is known in advance that a job will be executed multiple times. The data can be repartitioned offline to eliminate skew in the actual job. If offline processing is not feasible, it is still possible to implement preprocessing just before the actual job begins. SkewReduce [50], effectively implements this preprocessing step by searching the partition space for an even partitioning and query plan. It uses cost functions estimate the actual running time of the job and creates an even partitioning plan based on these estimates.

While not strictly a MapReduce system, Scarlett [4] implements this preprocessing in the distributed file system itself. Scarlett uses predicted file popularity metrics to dynamically adjust the replication factors of popular files. When a MapReduce job eventually runs on the data, its partitioning will not only be even, but each partition will have greater data locality than the same job running without Scarlett. Put another way, the loss of data locality experienced by a system that does *not* run Scarlett effectively causes skew by requiring that some tasks fetch data over congested network links or from overloaded file servers.

In the context of parallel joins, the extended adaptive load balancing parallel hash join algorithm proposed by Hua and Lee [41] implements a form of preprocessing by having all nodes send locally computed bucket information to a coordinator node. The coordinator uses this global information to construct an even assignment of buckets to nodes, which eliminates skew.

It should be noted that if a parallel database can know ahead of time which attribute a relation will most likely be joined on, the parallel database can partition the relation according to this join attribute when the data is loaded into the database. This partitioning occurs well in advance of any join computations and allows the query planner to simply use local relation partitions as join partitions. Since the relation is already

partitioned on the join attribute, the partitions are guaranteed to be evenly spread across the cluster thereby reducing skew.

The final "best practice" given by Kwon et al. is to rewrite the application to be independent of the data distribution. This advice is relevant in the cases of non-homomorphic `map` functions or holistic `reduce` functions. If an application has either of these functions, it will be incredibly difficult to eliminate skew. Unfortunately, such applications are usually quite complicated, and designing a skew-resistant variant places an enormous burden on the user. SkewReduce [50] and SkewTune [52] get part of the way there by attempting to tackle these problems from within the framework. The heart of the problem however is in the application implementation, so it may not be feasible to completely eliminate skew.

Chapter 6

Conclusions

In this work, we have considered the problem of building efficient large-scale data processing systems. We focus not only on scalability and high performance, but also on efficiency metrics such as performance per server and performance per I/O device. We hope that the findings and principles in this work will inspire others to build efficiency into their systems as a primary design concern.

The contributions of this work are:

1. We describe and evaluate two highly efficient data processing systems. TritonSort is capable of sorting a particular benchmark data set at record-breaking speeds. Themis is an efficient MapReduce implementation derived from TritonSort. Both systems exhibit the 2-IO property, which states that data records are read and written exactly twice, leading to highly efficient use of I/O devices.

2. We run Themis on a variety of next-generation server configurations, including high performance hardware devices such as flash-based solid state drives, PCI-Express attached flash devices, and 40 Gb/s Ethernet. Based on our experience with these hardware platforms, we describe and implement a series of optimizations that enable Themis to make efficient use of these devices.

3. We run Themis on the public cloud as an application of these optimizations. In

particular, we focus on high-performance virtual machines with access to flash-based solid state drives. We present a detailed analysis of the scalability properties of Amazon Web Services (AWS). Based on this analysis, we set several world-records in high-speed sorting. We then explore another cloud provider, Google Cloud Platform, as a generalization of our work.

Bibliography

[1] SDSC Gordon User Guide. http://portal.xsede.org/sdsc-gordon.

[2] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9), Sept. 1988.

[3] G. Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, 1970.

[4] 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 *EuroSys*, 2011.

[5] 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 *OSDI*, 2010.

[6] E. Anderson and J. Tucek. Efficiency Matters! In *HotStorage*, 2009.

[7] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *SoCC*, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[8] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD*, 1997.

[9] R. Arpaci-Dusseau, A. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *HPCA*, pages 90–101, 1998.

[10] Amazon Web Services. http://aws.amazon.com/.

[11] Microsoft Azure. http://azure.microsoft.com/.

[12] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman,

M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, Apr. 1985.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB*, 2010.

[14] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI*, 2004.

[15] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL Implementation On The MapReduce Framework. In *Proc. VLDB Endowment*, 2011.

[16] G. Coates. Infiniband HOWTO: SDP. https://pkg-ofed.alioth.debian.org/howto/infiniband-howto.html.

[17] A. Cockcroft. Benchmarking high performance I/O with SSD for Cassandra on AWS. http://techblog.netflix.com/2012/07/benchmarking-high-performance-io-with.html/.

[18] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[20] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 1990.

[21] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[22] D. DeWitt, J. Naughton, and D. Schneider. Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting. In *PDIS*, 1991.

[23] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. *The Database Column*, 1, 2008.

[24] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *VLDB*, 1991.

[25] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, 1992.

[26] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM CSUR*, 34(3), Sept. 2002.

[27] J. Evans. jemalloc. http://www.canonware.com/jemalloc/.

[28] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, pages 99–112, New York, NY, USA, 2012. ACM.

[29] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce : The Pig Experience. In *VLDB*, 2009.

[30] Google Cloud Platform. http://cloud.google.com/.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[32] D. Ghoshal, R. S. Canon, and L. Ramakrishnan. I/O performance of virtualized cloud environments. In *DataCloud-SC*, pages 71–80, New York, NY, USA, 2011. ACM.

[33] Google Compute Engine. http://cloud.google.com/compute/.

[34] G. Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 1994.

[35] T. Graves. GraySort and MinuteSort at Yahoo on Hadoop 0.23. http://sortbenchmark.org/Yahoo2013Sort.pdf.

[36] J. Gray and G. R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD*, 1987.

[37] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, pages 51–62, New York, NY, USA, 2009. ACM.

[38] M. Hadjieleftheriou, J. Byers, and G. Kollios. Robust Sketching and Aggregation of Distributed Data Streams. Technical Report 2005-011, Boston University, 2005.

[39] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SoCC*, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[40] B. Howe. lakewash_combined_v2.genes.nucleotide. https://dada.cs.washington.edu/research/projects/db-data-L1_bu/escience_datasets/seq_alignment/.

[41] K. A. Hua and C. Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *VLDB*, 1991.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[43] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[44] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SoCC*, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

[45] D. Jiang. Indy Gray Sort and Indy Minute Sort. http://sortbenchmark.org/BaiduSort2014.pdf.

[46] M. Kitsuregawa, M. Nakayama, and M. Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *VLDB*, 1989.

[47] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In *VLDB*, 1990.

[48] R. N. Kline and M. D. Soo. The TIMEIT Temporal Database Testbed, 1998. www.cs.auc.dk/TimeCenter/software.htm.

[49] B. C. Kuszmaul. TeraByte TokuSampleSort, 2007. http://sortbenchmark.org/tokutera.pdf.

[50] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SoCC*, 2010.

[51] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. The 5th Open Cirrus Summit, 2011.

[52] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *SIGMOD*, 2012.

[53] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *IMC*, pages 1–14, New York, NY, USA, 2010. ACM.

[54] W. Li, D. Gao, and R. T. Snodgrass. Skew Handling Techniques in Sort-Merge Join. In *SIGMOD*, 2002.

[55] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, pages 51–62, New York, NY, USA, 2010. ACM.

[56] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

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

[58] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*, 1999.

[59] J. P. McDermott, G. J. Babu, J. C. Liechty, and D. K. Lin. Data Skeletons: Simultaneous Estimation of Multiple Quantiles for Massive Streaming Datasets with Applications to Density Estimation. *Statistics and Computing*, 17(4), Dec. 2007.

[60] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *ScienceCloud*, pages 41–50, New York, NY, USA, 2012. ACM.

[61] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using Nutch/Lucene. In *IPDPS*, pages 1–8. IEEE, 2007.

[62] Michael C Schatz. CloudBurst: Highly Sensitive Read Mapping with MapReduce. *Bioinformatics*, 25(11):1363–9, 2009.

[63] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A cache-sensitive parallel external sort. In *VLDB*, 1995.

[64] C. Nyberg, C. Koester, and J. Gray. NSort: a Parallel Sorting Program for NUMA and SMP Machines, 1997.

[65] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report SIDL-WP-1999-0120. Stanford InfoLab, 1999.

[66] D. Peng and F. Dabek. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.

[67] M. Rahn, P. Sanders, J. Singler, and T. Kieritz. DEMSort – Distributed External Memory Sort, 2009. http://sortbenchmark.org/demsort.pdf.

[68] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *SoCC*, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

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

[70] A. Rasmussen, M. Conley, G. Porter, and A. Vahdat. TritonSort 2011. http://sortbenchmark.org/2011_06_tritonsort.pdf.

[71] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A balanced large-scale sorting system. In *NSDI*, 2011.

[72] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced and energy-efficient large-scale sorting system. *ACM Transactions on Computer Systems (TOCS)*, 31(1):3, 2013.

[73] Recovery-Oriented Computing. http://roc.cs.berkeley.edu/.

[74] SanDirect. Fusion-io SSD. http://www.sandirect.com/data-storage/flash/fusion-io-ssd.

[75] P. M. Sanjay Ghemawat. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[76] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, Sept. 2010.

[77] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *SIGMOD*, 1989.

[78] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn. SupMR: Circumventing disk and memory bandwidth bottlenecks for scale-up MapReduce. In *LSPP*, 2014.

[79] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[80] A. D. Smith and W. Chung. The RMAP Software for Short-Read Mapping. http://rulai.cshl.edu/rmap/.

[81] Sort Benchmark. http://sortbenchmark.org/.

[82] PCI-SIG Single Root IOV. http://www.pcisig.com/specifications/iov/single_root.

[83] SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/.

[84] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.

[85] L. Torvalds. O_DIRECT performance impact on 2.4.18. http://yarchive.net/comp/linux/o_direct.html.

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

[87] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic resource inference and allocation for MapReduce environments. In *ICAC*, pages 235–244, New York, NY, USA, 2011. ACM.

[88] J. S. Vitter. Random Sampling with a Reservoir. *ACM TOMS*, 11(1), Mar. 1985.

[89] vnStat - a network traffic monitor for Linux and BSD. http://humdi.net/vnstat/.

[90] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, Oct. 2008.

[91] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, 1991.

[92] G. Wang and T. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, pages 1–9. IEEE, 2010.

[93] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[94] Freebase Wikipedia Extraction (WEX). http://wiki.freebase.com/wiki/WEX.

[95] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *NSDI*, pages 367–381, 2012.

[96] M. Wolfe. More Iteration Space Tiling. In *Supercomputing*, 1989.

[97] J. Wyllie. Sorting on a Cluster Attached to a Storage-Area Network, 2005. http://sortbenchmark.org/2005_SCS_Wyllie.pdf.

[98] R. Xin. Spark officially sets a new record in large-scale sorting. http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html.

[99] R. Xin. Spark the fastest open source engine for sorting a petabyte. http://databricks.com/blog/2014/10/10/spark-petabyte-sort.html.

[100] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia. GraySort on Apache Spark by Databricks. http://sortbenchmark.org/ApacheSpark2014.pdf.

[101] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS. In *VLDB*, 2009.

[102] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling Data Skew in Parallel Joins in Shared-Nothing Systems. In *SIGMOD*, 2008.

[103] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

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

[105] Scaling Hadoop to 4000 nodes at Yahoo! http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

[106] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[107] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

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