TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System

ALEXANDER RASMUSSEN, GEORGE PORTER, and MICHAEL CONLEY,
University of California, San Diego
HARSHA V. MADHYASTHA, University of California, Riverside
RADHIKA NIRANJAN MYSORE, University of California, San Diego
ALEXANDER PUCHER, Vienna University of Technology
AMIN VAHDAT, University of California, San Diego and Google, Inc.

We present TritonSort, a highly efficient, scalable sorting system. It is designed to process large datasets, and has been evaluated against as much as 100TB of input data spread across 832 disks in 52 nodes at a rate of 0.938TB/min. When evaluated against the annual Indy GraySort sorting benchmark, TritonSort is 66% better in absolute performance and has over six times the per-node throughput of the previous record holder. When evaluated against the 100TB Indy JouleSort benchmark, TritonSort sorted 9703 records/Joule. In this article, we describe the hardware and software architecture necessary to operate TritonSort at this level of efficiency. Through careful management of system resources to ensure cross-resource balance, we are able to sort data at approximately 80% of the disks' aggregate sequential write speed.

We believe the work holds a number of lessons for balanced system design and for scale-out architectures in general. While many interesting systems are able to scale linearly with additional servers, per-server performance can lag behind per-server capacity by more than an order of magnitude. Bridging the gap between high scalability and high performance would enable either significantly less expensive systems that are able to do the same work or provide the ability to address significantly larger problem sets with the same infrastructure.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Data-intensive computing, balanced systems, sorting, system optimization

ACM Reference Format:
Rasmussen, A., Porter, G., Conley, M., Madhyastha, H. V., Mysore, R. N., Pucher, A., and Vahdat, A. 2013. TritonSort: A balanced and energy-efficient large-scale sorting system. *ACM Trans. Comput. Syst.* 31, 1, Article 3 (February 2013), 28 pages.
DOI:http://dx.doi.org/10.1145/2427631.2427634

This work originally appeared at NSDI 2011 [Rasmussen et al. 2011].
This work was supported by the National Science Foundation, under MRI no. CNS-0923523 and no. CSR-1116079. This work was also supported by Cisco Systems.
Authors' addresses: A. Rasmussen, G. Porter (corresponding author), and M. Conley, Department of Computer Science, University of California San Diego, CA; email: gmporter@cs.ucsd.edu; H. V. Madhyastha, University of California Riverside, CA; R. N. Mysore, Department of Computer Science, University of California San Diego, CA; A. Pucher, Vienna University of Technology, Vienna, Austria; A. Vahdat, Department of Computer Science, University of California San Diego, CA and Google, Inc.

DOI:http://dx.doi.org/10.1145/2427631.2427634

1. INTRODUCTION

The need for large-scale computing is increasing, driven by search engines, social networks, location-based services, and biological and scientific applications. The value of these applications is defined by the quality and quantity of data over which they operate, resulting in very high I/O and storage requirements. These Data-Intensive Scalable Computing systems, or DISC systems [Bryant 2007], require searching and sorting large quantities of data spread across the network. Sorting forms the kernel of many data processing tasks in the datacenter, exercises computing, I/O, and storage resources, and is a key bottleneck for many large-scale systems.

Several new DISC software architectures have been developed, including MapReduce [Dean and Ghemawat 2004], the Google file system [Ghemawat et al. 2003], Hadoop [2011], and Dryad [Isard et al. 2007]. These systems 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 [YahooCluster 2008].

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 DISC sorting systems [Anderson and Tucek 2009] 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 [Anderson and Tucek 2009]. The largest known industrial Hadoop clusters achieve only 20Mbps of average bandwidth for large-scale data sorting on machines theoretically capable of supporting a factor of 100 more throughput.

In this work we present TritonSort, a highly efficient sorting system designed to sort large volumes of data across dozens of nodes. We have applied it to datasets as large as 100 terabytes spread across 832 disks in 52 nodes. The key to TritonSort's efficiency is its *balanced* software architecture, which is able to effectively make use of a large amount of colocated storage per node, ensuring that the disks are kept as utilized as possible. Our results show the benefit of our design: evaluating TritonSort against the "Indy" GraySort benchmark [SortBenchMark 2010] resulted in a system that was able to sort 100TB of input tuples in about 66% of the absolute time of the previous record-holder, but with four times fewer resources, resulting in an increase in per-node efficiency by over a factor of six.

It is important to note that our focus in building TritonSort is to highlight the efficiency gains that can be obtained in building systems that process significant amounts of data through balancing computation, storage, memory, and network. Systems such as Hadoop and Dryad further support data-level replication, transparent node failure, and a generalized computational model, all of which are not currently present in TritonSort. TritonSort sorts fixed-size 100-byte tuples, which is an aspect of the *sortbenchmark.org* sorting grand challenge problem. This challenge is designed to stress the I/O capabilities of the sorting system, and so while it does not map directly onto real datacenter workloads, it does serve as a reasonable proxy for them.

However, in presenting TritonSort's hardware and software architecture, we describe several lessons learned in its construction that we believe are generalizable to other data processing systems. For example, our design relies on a very high disk-to-node ratio as well as an explicit, application-level management of in-memory buffers to minimize disk seeks and thus increase read and write throughput. We choose buffer sizes to balance time spent processing multiple stages of our sort pipeline, and trade off the utilization of one resource for another.

Our experiences show that for a common datacenter workload, systems can be built with commodity hardware and open-source software that improve on per-node efficiency by an order of magnitude while still achieving scalability. This is particular evident with the high number of tuples sorted in the JouleSort [Rivoire et al. 2007] benchmark. Building such systems will either enable significantly less expensive systems to be able to do the same work or provide the ability to address significantly larger problem sets with the same infrastructure.

The primary contributions of this article are: (1) the selection of a balanced hardware platform tuned to support a large-scale sort application, (2) a sort application implemented on top of a staged, pipeline-oriented software runtime that supports performance tuning via selection of appropriate buffer sizes and quantities, (3) an examination of projected sort performance when bottlenecks are removed, and (4) a discussion of the experience gained in building and deploying this prototype at scale.

2. DESIGN CHALLENGES

In this article, we focus on designing systems that sort large datasets as an instance of the larger problem of building balanced systems. Here, we present our precise problem formulation, discuss the challenges involved, and outline the key insights underlying our approach.

2.1. Problem Formulation

We seek to design a system that sorts large volumes of input data. Based on the specification of the sort benchmark [SortBenchMark 2010], our input data comprises 100-byte tuples with a 10-byte key and 90-byte value. The keys and values of these tuples are uniformly generated at random, providing a uniform distribution across the keyspace. This requirement is part of the "challenge problem" nature of the sorting challenge, namely that it provides a clear (and solvable) specification to compare sorting systems as technology changes over time. We target deployments with input data on the order of tens to hundreds of TB of randomly generated tuples. The input data is stored as a collection of files on persistent storage. The goal of a sorting system is to transform this input data into an ordered set of output files, also stored on persistent storage, such that the concatenation of these output files in order constitutes the sorted version of the input data. In accordance with the rules of the sorting challenge, we delete the input data during our sorting run, after it is processed. This is because our cluster does not have enough disk capacity to hold the input data, necessary intermediate data, as well as the output data at once (since for 100TB, and given the size of our cluster, each disk could have to store 266GB of input data, and 266GB of output data. However, our disks are only 500GB in size). Our goal is to design and implement a sorting system that can sort datasets of the targeted size while achieving a favorable trade-off between speed, resource utilization, and cost.

2.2. The Challenge of Efficient Sorting

Sorting large datasets places stress on several resources in a cluster. First, storing tens to hundreds of TB of input and output data demands a large amount of storage capacity. Given the size of the data and modern commodity hard drive capacities, the data must be stored across several storage devices and almost certainly across many machines. Second, reading the input data and writing the output data across many disks simultaneously places load on both storage devices and I/O controllers. Third, since the tuples are distributed randomly across the input files, almost all of the large dataset to be sorted will have to be sent over the network. Finally, comparing tuples in order to sort them requires a nontrivial amount of compute power. This combination of demands makes designing a sorting system that efficiently utilizes all of these resources challenging.

Our key design principle to ensure good resource utilization is to construct a balanced system—a system that drives all resources at as close to 100% utilization as possible. For any given application and workload, there will be an ideal configuration of hardware resources in keeping with the application's demands on these resources. In practice, the set of hardware configurations available is limited by the availability of components (one cannot currently, for example, buy a processor with exactly 13 cores), and so a configuration must be chosen that best meets the application's demands. Once that hardware configuration is determined, the application must be architected to suitably exploit the full capabilities of the deployed hardware. In the following section, we outline our considerations in designing such a balanced system, including our choice of a specific hardware and software architecture. We did not choose this platform with sorting in mind, and so we believe that our design generalizes to other DISC problems as well.

2.3. Design Considerations

Our system's design is motivated by three main considerations. First, we rely only on commodity hardware components. This is both to keep the costs of our system relatively low and to have our system be representative of today's data centers so that the lessons we learn can be applied to other applications with workload characteristics similar to those of sort. Hence, we do not make use of networking substrates such as Infiniband that provide high network bandwidth at high cost. Also, despite the recent emergence of Solid State Drives (SSDs) that provide higher I/O rates, we chose to use hard disks because they continue to provide the most affordable option for high capacity storage and streaming I/O. We believe that properly architected sorting software should not stress random I/O behavior, where SSDs currently excel.

Second, we focus our software architecture on minimizing disk seeks. In the particular hardware configuration we chose, the key bottleneck for sort among the various system resources is disk I/O bandwidth. Hence, the primary goal of the system is to enable all disks to operate continuously at peak bandwidth. The main challenge in sustaining peak disk bandwidth is to minimize the amount of time the disks spend seeking, since any time seeking is not spent transferring data.

Third, we choose to focus on hardware architectures whose total memory cannot contain the entire dataset. One possible implementation of sort is to read all the input data into memory, appropriately shuffle the data across machines in the cluster, sort the local in-memory data on each machine, and then write the sorted data to the local disks. Note that in this case, every tuple is read from and written to persistent storage exactly once. However, this implementation would require an amount of memory at least equal to the amount of input data; given that the cost per GB of RAM is over 70 times more than that of disks, such a design would significantly drive up costs and be infeasible for large input datasets. Although not the primary focus of this work, we do evaluate an entirely in-memory sort in Section 5.5.

Instead, we pursue an alternative implementation wherein every tuple is read and written multiple times from disk before the data is completely sorted. Storing intermediate results on disk makes the system's memory requirement far more modest. Sorting data on clusters that have less memory than the total amount of data to be sorted requires every input tuple to be read and written at least twice [Aggarwal and Vitter 1988]. Since every additional read and write increases the time to sort, we seek to achieve exactly this lower bound to maximize system performance. In TritonSort, half the disks in the cluster are devoted to intermediate data. Given that intermediate

Storage				
Type	Capacity	R/W throughput	Price (Jan 2010)	Price (Mar 2012)
7.2k-RPM	500 GB	90-100 MBps	\$200	\$159
15k-RPM	150 GB	150 MBps	\$290	\$165
SSD	64 GB	250 MBps	\$450 (SLC)	\$95 (MLC)

Network	
Type	Cost/port (Jan 2010)
1 Gbps Ethernet	\$33
10 Gbps Ethernet	\$480

Server	
Type	Cost (Jan 2010)
8 disks, 8 CPU cores	\$5,050
8 disks, 16 CPU cores	\$5,450
16 disks, 16 CPU cores	\$7,550

Fig. 1. Resource options considered for constructing a cluster for a balanced sorting system. These values are estimates as of January, 2010.

data is no longer needed after the sort is complete, devoting such a large proportion of storage to it might seem wasteful. However, our aim is to build the most balanced system possible, and so at this point we require that the intermediate storage pool be as large as the input/output storage pool. In unbalanced storage configurations where that property does not hold and that are I/O limited, then one or the other pool will be a bottleneck at runtime.

2.4. Hardware Architecture

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, sort 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. As mentioned before, we quickly ruled out the use of flash, even though it supports significantly higher I/O operations per second than disk, and has a higher sustained throughput (in the range of 2–4x the throughput of disk). However, the GB/\$ cost for an entirely flash-based approach was too excessive to pursue. We also ruled out the use of flash on the PCIe bus, since its high cost(over \$10/GB) would have necessitated a storage solution over \$1M. Thus, for this effort, we constrained our selection to disks. We first survey a range of hard disk options shown in Figure 1. We find that 7.2k-RPM SATA disks provide the most cost-effective option in terms of balancing \$ per GB and \$ per read/write MBps (assuming we can achieve streaming I/O). The most cost-effective direct-attach packaging option we had were servers with 16 disks per server. Since building our cluster in January of 2010, we have since updated this table to include recent pricing for the storage components (the server costs are difficult to directly compare since the original configurations are no longer available). Even here, in terms of total capacity and streaming I/O, the 7.2k-RPM drives provide the lowest cost option. Allowing 16 disks to operate at full streaming I/O throughput, we require storage controllers that are able to sustain at least 1600MBps of streaming bandwidth. Because of the PCI bus' bandwidth limitations, our hardware design necessitated two 8x PCI drive controllers, each supporting 8 disks.

Third, 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—1Gbps, 10Gbps, and 40Gbps—with cost increasing proportional to throughput (see Figure 1). Given our choice of 7.2k-RPM disks for storage, a 1Gbps network can accommodate at most one disk per server without the network throttling disk I/O. Therefore, we settle on a 10Gbps network; 40Gbps Ethernet has yet to mature and hence is still cost prohibitive. Our choice of 16 disks is in balance with a 10Gbps network interconnect. 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.

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 12GB of memory; however, during development we realized that 24GB was required to provide sufficiently large writes, and so the machines were upgraded. We discuss this addition when we present our architecture in Section 3. One of the key takeaways from our work is the important role that buffering plays in enabling high utilization of the network, disk, and CPU. Determining the appropriate amount of memory buffering is not straightforward and we leave to future work techniques that help automate this process.

2.5. Software Architecture

To maximize cluster resource utilization, we need to design an appropriate software architecture. We started with a a particular concrete starting point in terms of software: Debian Linux running the 2.6 kernel, the XFS file system, and an application written in C++. We will revisit and justify this starting point later in this article. There are a range of possible software architectures in keeping with our constraint of reading and writing every input tuple at most twice. The class of architectures upon which we focus share a similar basic structure. These architectures consist of two phases separated by a distributed barrier, so that all nodes must complete phase one before phase two begins. In the first phase, input data is read from disk and routed to the node upon which it will ultimately reside. Each node is responsible for storing a disjoint portion of the key space. When data arrives at its destination node, that node writes the data to its local disks. In the second phase, each node sorts the data on its local disks in parallel. At the end of the second phase, each node has a portion of the final sorted sequence stored on its local disks, and the sorted sequences stored on all nodes can be concatenated together to form the final sorted sequence.

There are several possible implementations of this general architecture, but any implementation contains at least a few basic software elements. These software elements include *Readers* that read data from on-disk files into in-memory buffers, *Writers* that write buffers to disk, *Distributors* that distribute a buffer's tuples across a set of logical divisions, and *Sorters* that sort buffers.

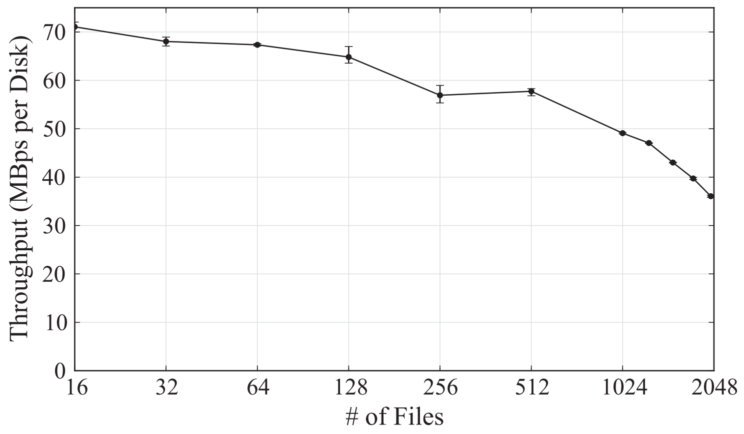


Fig. 2. Performance of a Heaper-Merger sort implementation in microbenchmark on a 200GB per disk parallel external merge-sort as a function of the number of files merged per disk.

Our initial implementation of TritonSort was designed as a distributed parallel external merge-sort. This architecture, which we will call the Heaper-Merger architecture, is structured as follows. In phase one, Readers read from the input files into buffers, which are sorted by Sorters. Each sorted buffer is then passed to a Distributor, which splits the buffer into a sorted chunk per node and sends each chunk to its corresponding node. Once received, these sorted chunks are heap-sorted by software elements called Heapers in batches and each resulting sorted batch is written to an intermediate file on disk. In the second phase, software elements called Mergers merge-sort the intermediate files on a given disk into a single sorted output file.

The problem with the Heaper-Merger architecture is that it does not scale well. In order to prevent the Heaper in phase one from becoming a bottleneck, the length of the sorted runs that the Heaper generates is usually fairly small, on the order of a few hundred megabytes. As a consequence, the number of intermediate files that the Merger must merge in phase two grows quickly as the size of the input data increases. This reduces the amount of data from each intermediate file that can be buffered at a time by the Merger and requires that the merger fetch additional data from files much more frequently, causing many additional seeks.

To demonstrate this problem, we implemented a simple Heaper-Merger sort module in microbenchmark. We chose to sort 200GB per disk in parallel across all the disks to simulate the system's performance during a 100TB sort. Each disk's 200GB dataset is partitioned among an increasingly large number of files. Each node's memory is divided such that each input file and each output file can be double-buffered. As shown in Figure 2, increasing the number of files being merged causes throughput to decrease dramatically as the number of files increases above 1000.

TritonSort uses an alternative architecture with similar software elements as detailed before and again involving two phases. We partition the input data into a set of logical partitions; with D physical disks and L logical partitions, each logical partition corresponds to a contiguous $\frac{1}{L}^{th}$ fraction of the key space and each physical disk hosts $\frac{L}{D}$ logical partitions. In the first phase, Readers pass buffers directly to Distributors. A Distributor maps the key of every tuple in its input buffer to its corresponding logical partition and sends that tuple over the network to the machine that hosts this logical partition. Tuples for a given logical partition are buffered in memory and written to disk in large chunks in order to seek as little as possible. In the second phase, each

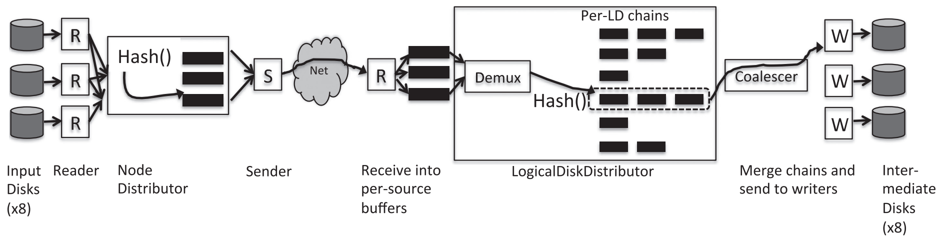


Fig. 3. Block diagram of TritonSort's phase one architecture.

logical partition is read into an in-memory buffer, that buffer is sorted, and the sorted buffer is written to disk. This scheme bypasses the seek limits of the earlier merge-sort-based approach. Also, by appropriately choosing the value of L, we can ensure that logical partitions can be read, sorted, and written in parallel in the second phase. Since our testbed nodes have 24GB of RAM, to ensure this condition we set the number of logical partitions per node to 2520 so that each logical partition contains less than 1GB of tuples when we sort 100TB on 52 nodes. We explain this architecture in more detail in the context of our implementation in the next section.

3. DESIGN AND IMPLEMENTATION

TritonSort is a distributed, staged, pipeline-oriented dataflow processing system. In this section, we describe TritonSort's design and motivate our design decisions for each stage in its processing pipeline.

3.1. Architecture Overview

Figures 3 and 8 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. Last, it can keep its *run()* method active, polling for new work explicitly. TritonSort stages implement each of these methods, as described shortly. 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.

3.2. 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 $size_{LD}$ (though not necessarily entirely full) 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 article, we assume that tuples' keys are distributed uniformly over the key range which ensures that each logical disk is approximately the same size; we discuss how TritonSort can be made to handle nonuniform key ranges in Section 6.1.

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 8 disks each. 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 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.

3.3. TritonSort Architecture: Phase One

Phase one of TritonSort, diagrammed in Figure 3, 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 80MB ProducerBuffers with input data. We chose this size because it is large enough to obtain near sequential throughput from the disk. The Reader stage produces up to 800MBps from its eight stages. However, each stage spends the vast majority of its time in the `iowait` state, waiting on the underlying disk. Thus we are able to multiplex all eight Reader stages onto a single CPU core.

NodeDistributor. A NodeDistributor (shown in Figure 4) 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

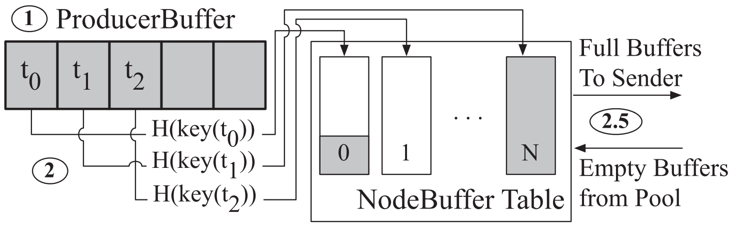


Fig. 4. The NodeDistributor stage, responsible for partitioning tuples by destination node.

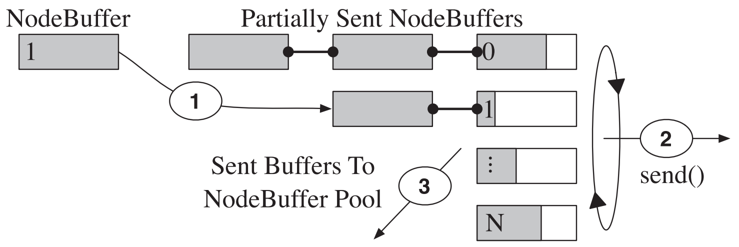


Fig. 5. The Sender stage, responsible for sending data to other nodes.

chosen to be the size of the ProducerBuffer divided by the number of nodes, and is approximately 1.6MB in size for the scales we consider in this article.

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. The throughput of a single instance of the NodeDistributor stage is about 300MBps, based on its two primary tasks of scanning memory in a linear manner, and hashing tuples. Thus, we require three of these stages, capable of handling 900MBps, to keep up with the Reader stages, which produce 800MBps.

Sender. The Sender stage (shown in Figure 5) 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. To keep up with the Reader stages, it must be able to send data at 800MBps, or about 6.4Gbps, to ensure that the Reader stages do not suffer from backpressure. Each Sender maintains a separate TCP socket per peer node in the cluster. The Sender stage can be implemented in a multithreaded or a single-threaded manner. In the multithreaded 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 reasons described in Section 4.1, we choose a single-threaded Sender implementation instead. Here, the Sender interleaves the sending of data across all the destination nodes

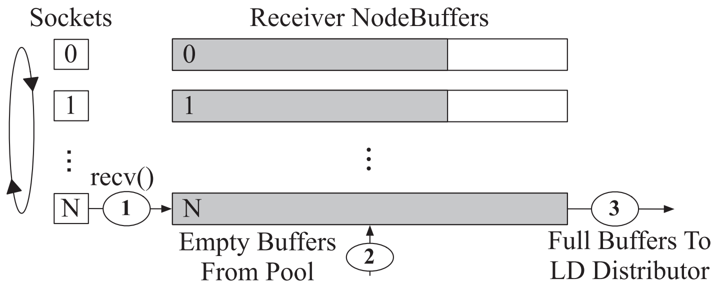


Fig. 6. The Receiver stage, responsible for receiving data from other nodes' Sender stages.

in small nonblocking 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 nonempty, the Sender accesses the NodeBuffer at the head of the list, and sends a fixed-sized amount of data to the appropriate destination host using a nonblocking *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 6, 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 nonblocking *recv()* call. This small socket read size is due to the rate-limiting fix that we explain in Section 4.1. 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 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 7, 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.

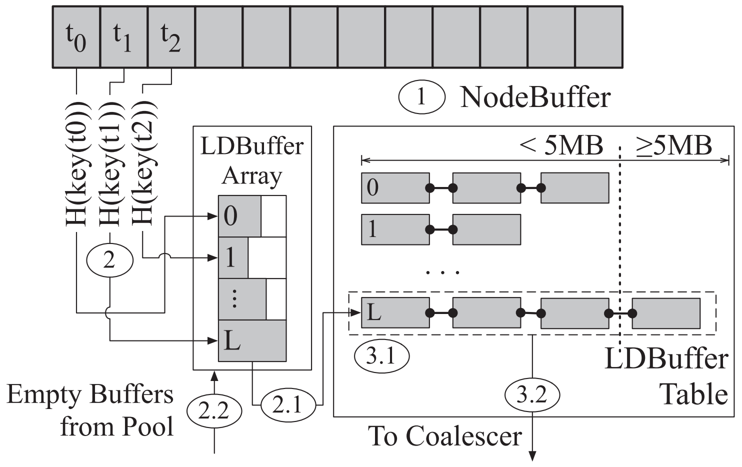


Fig. 7. The LogicalDiskDistributor stage, responsible for distributing tuples across logical disks and buffering sufficient data to allow for large writes.

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 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 (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 24GB of memory.

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) ≥ 5MB do
14:         ld ← LDTable.getLongestList(d)
15:         Coalescer.pushNewWork(ld)
16:     end while
17: end for
```

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 to 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 to 17 examine

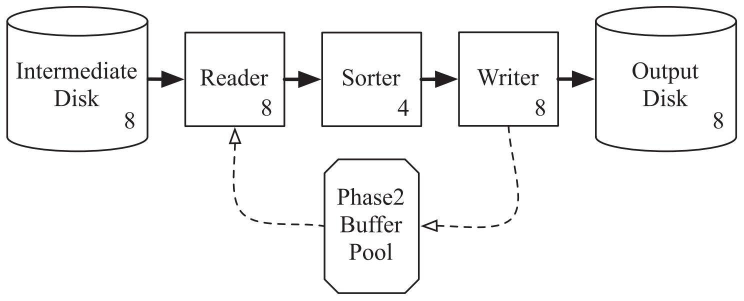


Fig. 8. Block diagram of TritonSort's phase two architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

the LDBufferTable, looking for logical disk lists that have accumulated enough data to write out to disk. We buffer at least 5MB 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 5MB 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 before and used much larger LDBuffers (~10MB each) rather than many small LDBuffers. The Coalescer stage (described in the following text) 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. While optimizing the system, however, we realized that the nontrivial 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.

We initially considered an asynchronous I/O (AIO) implementation of the Writer stage, in which the Writer instances would issue *write()* requests that immediately return. Later, when the disks finish writing the data, they would signal the completion back to the caller who would return the buffer to the buffer pool. Unfortunately, the implementation of AIO on our particular version of Linux does not ensure that the interrupts are delivered to the core that issued the *write()* operation, which would hurt inter-stage performance isolation. Furthermore, the AIO support did not appear to work properly, in that the calling thread was blocked until each individual AIO call completed. Since we have only eight Writer instances, and since each *write()* operation runs for a relatively long time, it was not a problem multiplexing multiple Writer stages on a single hyperthread.

3.4. 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, streaming 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 (which used to contain the input data, before it was deleted during sorting). 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 850MB, using most of the 24GB of RAM available on each node and allowing for additional memory required by the operating system. To sort 850MB logical disks fast enough to not block the Reader and Writer stages, we find that four Sorters suffice.

3.5. 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

Worker Type	Size Of Input (MB)	Runtime (ms)	# Workers	Throughput (in MBps)	Total Throughput (in MBps)
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

Fig. 9. Median stage runtimes for a 52-node, 100TB sort, excluding the amount of time spent waiting for buffers.

Component	Quantity	Description
CPU	2	Intel Xeon E5520 (4-core), 2.2 GHz
PCI	2	PCIe gen 2.0 x4 slots (one per RAID controller)
PCI	1	PCIe gen 2.0 x8 slot (per Ethernet controller)
Memory	6	2 GB DDR3-1333 Mhz
Ethernet	1	Myricom 10Gbps dual-port Ethernet
RAID Controller	2	HP Smart Array p410 SAS/SATA
Disk	16	HP 500 GB SATA 7,200 RPM 2.5”

Fig. 10. Hardware implementation of each node in the cluster used to evaluate TritonSort.

amount of CPU time it requires to process a work unit. Figure 9 shows the speed and performance of each stage in the pipeline. Shown is the worker type (described earlier), the average size of each buffer it processes, the runtime to process each buffer, the number of worker instances included in our deployment, and finally the throughput (per stage and in aggregate across all instances of that stage type, respectfully). The bottleneck stage can be determined by looking at the minimum value in the rightmost column. In our implementation, we are limited by the speed of the Writer stage in both phases one and two.

3.6. Hardware Implementation

Figure 10 describes the resulting hardware configuration of the cluster used to evaluate TritonSort. This configuration corresponds to the design decisions described in Sections 2.4 and 2.3.

4. OPTIMIZATIONS

In implementing the TritonSort architecture, we learned that several nonobvious optimizations were necessary to meet our desired goal of driving every disk at full utilization. Here, we present the key takeaways from our experience. In each case, we believe these lessons generalize to a wide variety of DISC systems.

4.1. Network

For TritonSort to operate at the aggregate sequential streaming bandwidth of all of its disks, the network must be able to sustain the read throughput of eight disks while data is being shuffled among nodes in the first phase. Since the 7.2k-RPM disks we use deliver at most 100MBps of sequential read throughput (Table I), the network must be able to sustain 6.4Gbps of all-pairs bandwidth, irrespective of the number of nodes in the cluster.

It is well-known that sustaining high-bandwidth flows in datacenter networks, especially all-to-all patterns, is a significant challenge. In-network reasons for this

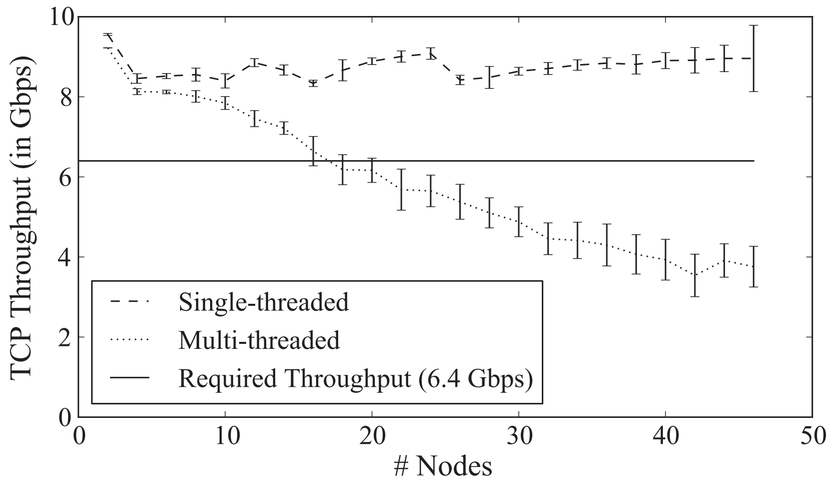


Fig. 11. Comparing the scalability of single-threaded and multithreaded Receiver implementations.

include commodity datacenter network hardware, incast, queue buildup, and buffer pressure [Alizadeh et al. 2010], and in-host reasons include overhead in the socket and network stack, as well as thread scheduling policies to ensure that data is fairly injected into the network. A major challenge in TritonSort is supporting a large number of open connections while ensuring that downstream stages do not stall waiting for work. We found that the primary reason for this was data not going into the network due to endhost network stack overhead, due to unfairness in the thread scheduler, causing starvation of individual flows (and thus downstream pipeline stalls).

Initially, we chose a straightforward multithreaded design for the Sender and Receiver stages in which there were N Senders and N Receivers, one for each TritonSort node. In this design, each Sender issues blocking *send()* calls on a NodeBuffer until it is sent. Likewise, on the destination node, each Receiver repeatedly issues blocking *recv()* calls until a NodeBuffer has been received. Because the number of CPU hyperthreads on each of our nodes is typically much smaller than $2N$, we pinned all Senders' threads to a single hyperthread and all Receivers' threads to a single separate hyperthread.

Figure 11 shows that this multithreaded approach does not scale well with the number of nodes, dropping below 4Gbps at scale. This poor performance is due to thread scheduling overheads at the end hosts. TCP receive buffers can fill up much faster than individual threads can be scheduled to sink the incoming data, especially given the latency of activating and deactivating multiple threads responsible for draining each of these sockets. The Receiver stage must clear out each of its buffers sufficiently fast that data does not unnecessarily queue. Since there are 52 such buffers, a Receiver must visit and clear a receive buffer in just over 20 μs. A Receiver worker thread cannot drain the socket, block, go to sleep, and get woken up again fast enough to service buffers at this rate.

To circumvent this problem we implemented a single-threaded, nonblocking Receiver that scans through each socket in round-robin order, copying out any available data and storing it in a NodeBuffer during each pass through the array of open sockets. This implementation is able to clear each socket's Receiver buffer faster than the arrival rate of incoming data. Figure 11 shows that this design scales well as the cluster grows. One limitation of this design is that for a single-threaded Sender and receive, the scale of the cluster is limited to the number of sockets that a single thread can fill

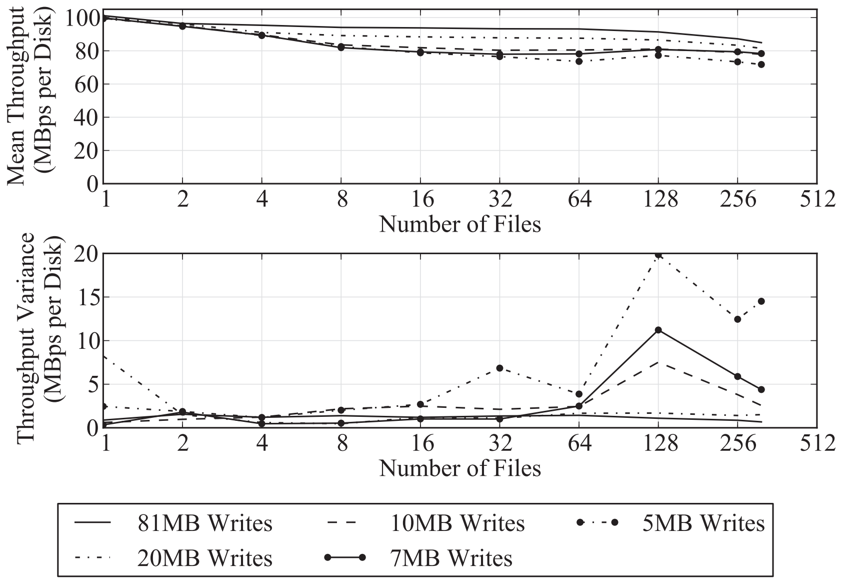


Fig. 12. Microbenchmark indicating the ideal disk throughput as a function of write size.

or drain without blocking. If we were to build a much larger cluster (e.g., 512 nodes), we would need to assign multiple hyperthreads to the Sender and Receiver stages.

4.2. Minimizing Disk Seeks

Key to making the TritonSort pipeline efficient is minimizing the total amount of time spent performing disk seeks, both while writing data in phase one and while reading that data in phase two. As individual write sizes get smaller, the throughput drops, since the disk must occasionally seek between individual write operations. Figure 12 shows disk write throughput measured by a synthetic workload generator writing to a configurable set of files with different write sizes. Ideally, the Writer would receive WriterBuffers large enough that it can write them out at close to the sequential rate of the disk, for example, 80MB. However, the amount of available memory limits TritonSort's write sizes. Since the tuple space is uniformly distributed across the logical disks, the LogicalDiskDistributor will fill its LDBuffers at approximately a uniform rate. Buffering 80MB worth of tuples for a given logical disk before writing to disk would cause the buffers associated with all of the other logical disks to become approximately as full. This would mandate significantly higher memory needs than what is available in our hardware architecture. Hence, the LogicalDiskDistributor stage must emit smaller WriterBuffers, and it must interleave writes to different logical disks.

4.3. The Importance of File Layout

The physical layout of individual logical disk files plays a strong role in trading off performance between the phase one Writer and the phase two Reader. One strategy is to append to the logical disk files in a log-structured manner, in which a WriterBuffer for one logical disk is immediately appended after the WriterBuffer for a different logical disk. This is possible if the logical disks' blocks are allocated on demand. It has the advantage of making the phase one Writer highly performant, since it minimizes seeks and leads to near-sequential write performance. On the other hand, when a phase two Reader begins reading a particular logical disk, the underlying physical disk will need to seek frequently to read each of the WriterBuffers making up the logical disk.

An alternative approach is to greedily allocate all of the blocks for each of the logical disks at start time, ensuring that all of a logical disk's blocks are physically contiguous on the underlying disk. This can be accomplished with the *fallocate()* system call, which provides a hint to the file system to preallocate blocks. In this scheme, interleaved writes of WriterBuffers for different logical disks will require seeking, since two subsequent writes to different logical disks will need to write to different contiguous regions on the disk. However, in phase two, the Reader will be able to sequentially read an entire logical disk with minimal seeking. We also use fallocate() on input and output files so that phase one Readers and phase two Writers seek as little as possible.

The location of output files on the output disks also has a dramatic effect on phase two's performance. If we do not delete the input files before starting phase two, the output files are allocated space on the interior cylinders of the disk. When evaluating phase two's performance on a 100TB sort, we found that we could write to the interior cylinders of the disk at an average rate of 64MBps. When we deleted the input files before phase two began, ensuring that the output files would be written to the exterior cylinders of the disk, this rate jumped to 84MBps. For the evaluations in Section 5, we delete the input files before starting phase two. For reference, the fastest we have been able to write to the disks in microbenchmark has been approximately 90MBps.

4.4. CPU Scheduling

Modern operating systems support a wide variety of static and dynamic CPU scheduling approaches, and there has been considerable research into scheduling disciplines for data processing systems. We put a significant amount of effort into isolating stages from one another by setting the processor affinities of worker threads explicitly, but we eventually discovered that using the default Linux scheduler results in a steady-state performance that is only about 5% worse than any custom scheduling policy we devised. In our evaluation, we use our custom scheduling policy unless otherwise specified.

4.5. Pipeline Demand Feedback

Initially, TritonSort was entirely "push"-based, meaning that a worker only processed work when it was pushed to it from a preceding stage. While simple to design, certain stages perform suboptimally when they are unable to send feedback back in the pipeline as to what work they are capable of doing. For example, the throughput of the Writer stage in phase one is limited by the latency of writes to the intermediate disks, which is governed by the sizes of WriterBuffers sent to it as well as the physical layout of logical disks (due to the effects of seek and rotational delay). In its naïve implementation, the LogicalDiskDistributor sends work to the Writer stage based on which of its LDBuffer lists is longest with no regard to how lightly or heavily loaded the Writers themselves are. This can result in an imbalance of work across Writers, with some Writers idle and others struggling to process a long queue of work. This imbalance can destabilize the whole pipeline and lower total throughput.

One possible approach that we could have used would have been to add a *writer scheduler*. This scheduler could have served as a load balancer, intercepting work units from the pipeline and distributing them to Writer instances based on their observed queue lengths. In our case, we chose a different approach in which we introduce a layer of indirection between the LogicalDiskDistributor and the Writers. This indirection layer must communicate information about the sizes of Writers' work queues to upstream stages. We do this by creating a pool of *write tokens*. Every write token is assigned a single "parent" Writer. We assign parent Writers in round-robin order to tokens as the tokens are created and create a number of tokens equal to the number of WriterBuffers. When the LogicalDiskDistributor has buffered enough LDBuffers so that one or more of its logical disks is above the minimum write threshold (5MB), the LogicalDiskDistributor will query the write token pool, passing it a set of Writers for which it has enough data. If a write token is available for one of the specified Writers in the set, the pool will return that token, otherwise it will signal that no tokens are available. The LogicalDiskDistributor is required to pass a token for the target Writer along with its LDBuffer list to the next stage. This simple mechanism prevents any Writer's work queue from growing longer than its "fair share" of the available WriterBuffers and provides reverse feedback in the pipeline without adding any new architectural features.

5. EVALUATION

We now evaluate TritonSort's performance and scalability under various hardware configurations.

5.1. Evaluation Environment

We evaluated TritonSort on a 52-node cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24GB of memory, and sixteen 500GB 7,200 RPM 2.5" SATA drives. 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. Each server has two HP P410 drive controllers with 512MB on-board cache, as well as a Myricom 10Gbps network interface. We use a 52-port Cisco Nexus 5020 datacenter switch for the all experiments except for MinuteSort, which uses a Cisco Nexus 5596UP switch. The servers run Linux 2.6.35.1, and our implementation of TritonSort is written in C++. TritonSort consists of 25,078 lines of C++ code, and 19,167 lines of Python.

5.2. Comparison to Previous GraySort Recordholders

The 100TB Indy GraySort benchmark was introduced in 2009, and hence there are few systems against which we can compare TritonSort's performance. The most recent holder of the Indy GraySort benchmark, DEMSort [Rahn et al. 2009], sorted slightly over 100TB of data on 195 nodes at a rate of 564GB per minute. TritonSort currently sorts 100TB of data on 52 nodes at a rate of 938GB per minute, a factor of six improvement in per-node efficiency. Each DEMSort node contained four disks, for a total of 780 disks, whereas each TritonSort node contains 16 disks, for a total of 832 disks. So the six per-node improvement for TritonSort comes at the expense of about 7% more disks compared to DEMSort.

5.3. Examining Changes in Balance

We next 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, and so we carry out a more limited evaluation.

In the first experiment, we replaced the 500GB, 7200RPM disks that are used as the intermediate disks in phase one and the input disks in phase two with 146GB, 15000RPM disks. The reduced capacity of the drives necessitated running an experiment with a smaller input dataset. To allow space for the logical disks to be preallocated 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 dataset is small

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

Fig. 13. Effect of increasing speed of intermediate disks on a two-node, 500GB sort.

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

Fig. 14. Effect of increasing the amount of memory per node on a two-node, 2TB sort.

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 Figure 13. We first examine the effect 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 (14MB) 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 15000 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. One side-effect of this is that the LogicalDiskDistributor cannot populate WriterBuffers as fast as they become available, so it reverts to a pathological case in which it always is able to successfully retrieve a write token and hence continuously writes minimally filled (5MB) buffers. Creating a LogicalDiskDistributor stage that dynamically adjusts its write size based on write token retrieval success rate is the subject of future work.

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 14MB to 28MB) and using the remaining memory (22GB) to create additional LDBuffers. As shown in Figure 14, 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. Although larger write sizes mean that larger writes are amortized across individual disk seeks, the resulting increase in total system performance has diminishing, nonlinear returns.

5.4. TritonSort Scalability

Figure 15 shows TritonSort's total throughput when sorting 1TB 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

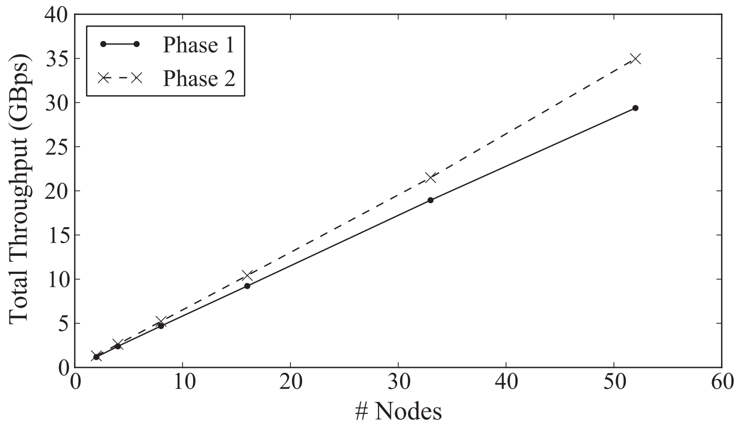


Fig. 15. Throughput when sorting 1TB per node as the number of nodes increases.

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.

5.5. MinuteSort: An In-Memory Sort Implementation

For the MinuteSort benchmark, we modify our architecture as follows. In the first phase, as before, we read the input data and distribute tuples across machines based on the logical disk to which the tuple maps. However, logical disks are maintained in memory instead of being written to disk immediately. In phase two (once all input tuples have been transferred to their appropriate logical disks), the in-memory logical disks are directly passed to workers that sort them. These sorters in turn pass sorted logical disks to writers to be written to disk. Hence, logical disks are still written to disk but are not written until after they have been sorted. This enables us to make use of 16 Writer stages, since we can separate reads and writes to disk in time (versus separating those operations by partitioning the disks into input and intermediate disks in the case of out-of-memory sorting described before). The goal of MinuteSort is to sort as much data as possible in under one minute, and thus the evaluation metric is "GB per node."

Running TritonSort in its MinuteSort configuration on 66 nodes resulted in 20.5GB per node for a total of 1353GB of data. We performed 15 consecutive trials. For these trials, TritonSort's median elapsed time was 59.2 seconds, with a maximum time of 61.7 seconds, a minimum time of 57.7 seconds, and an average time of 59.2 seconds. All times were rounded to the nearest tenth of a second. Only 3 of the 15 consecutive trials had completion times longer than 60 seconds. Although MinuteSort and JouleSort (described in the following section) test against a different number of nodes than Indy and Daytona GraySort, their results can be qualitatively compared, given that the scalabilty we have observed is nearly linear across the range of nodes that we test against.

5.6. JouleSort: A Measure of Energy Efficiency

A key motivation to building a balanced sorting system is improving per-node efficiency. A potential effect of improved efficiency is lowering the energy requirements to complete a given task, and in this section we describe a quantitative study of the use of energy in TritonSort. A new sorting category of the annual GraySort challenge is the 100TB JouleSort category. The evaluation metric in this categoy is "records sorted per Joule." TritonSort is the first system to set the records sorted per Joule benchmark at 100TB data sizes, and so we do not have a particular point of comparison to know how much better we do as compared to the state-of-the-art in data centers.

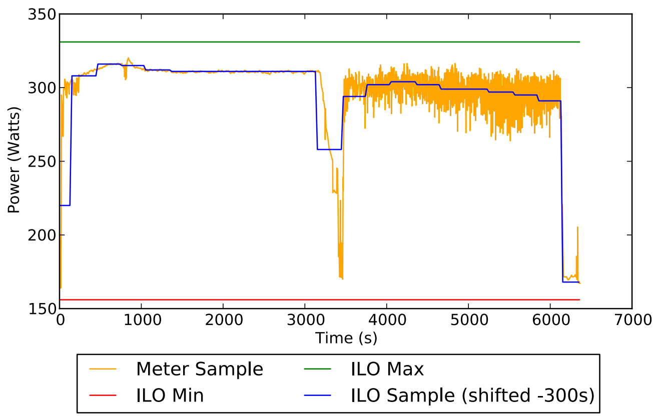
For the 100TB Indy JouleSort benchmark, we ran on 52 nodes and one experiment head node, all of which are HP DL380G6 servers. We employed two different methodologies for measuring the power consumed by the cluster during each sorting run. First, we relied on the built-in meter included in each server. Each server comes equipped with an embedded power measurement system as part of its Integrated Lights-Out (ILO) system. Unfortunately, the ILO cannot be queried more than once every 15–20 seconds because the onboard service processor is quite slow and a separate SSL connection must be set up and torn down for each query issued to the meter.

To cross-check and provide sufficiently fine-grained measurements, we next attached a WattsUp [WattsUpMeter 2011] power meter to a machine at random for each of our trials. We measured the observed power draw throughout the run from that representative server and extrapolated its power to account for the power of the experiment servers. We also performed a similar measurement on the experimental control node; since the control node is not doing anything particularly intensive (monitoring the power of each machine and recording experiment time), its power consumption is relatively low. In practice, we found that the average draw for the control node was 134 Watts.

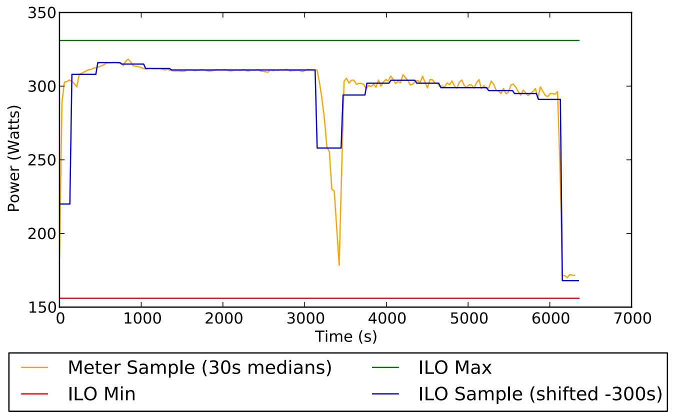
We also had to account for the power used by our Cisco 5596UP datacenter switch. To do this, we plugged that switch into an Avocent PM 3000V PDU [AvocentPDU 2011] during our sorting run. This PDU has remote access through a command-line interface and SNMP. For each outlet, it keeps track of the minimum, maximum, and present power draw. We used the maximum power draw reported after the run was completed over the lifetime of that switch's operation as an estimate of the instantaneous power draw. This overestimates the value somewhat, but made the power calculations easier. In fact, there was only a small difference between the values observed during the run and the maximum value observed since the switch was plugged into the PDU, and so the effect on precision was small. We ran a total of five trials to measure the energy required to perform each sorting run.

When evaluated against the Indy GraySort workload, TritonSort sorted an average of 9703 records per Joule with a standard deviation of 351 records per Joule. To put this in context, FAWNSort [Fawnsort 2010], the most energy-efficient sorting system for the 10^8-byte sort benchmark, sorted data at 44,900 records per Joule. FAWNSort is based on low-power Atom processors and flash SSDs. While TritonSort had a lower record per Joule rate, its ability to sort such a large amount of data over a modest number of nodes provides it with significant energy savings compared to previous systems. In addition to measuring this particular sort run, we also want to determine how the energy requirement varies with the number of nodes. Although we were unable to rerun additional energy studies with different numbers of nodes, we note that the scalability of TritonSort is almost linear (see Figure 15), and so we can estimate the energy draw as $E(N) \approx N{*}E_{server} + E_{switch}$, where N is the number of nodes, E_{server} is the average energy used by a single server, and E_{switch} is the energy used by the switch. In the experiments we performed in our testbed, E_{server} = 2.764 MJ for 100TB Daytona sort, E_{server} = 2.109 MJ for 100TB Indy sort, and E_{switch} = 8.332 MJ in both cases.

To determine how accurate the measurements we were receiving from the nodes' ILO systems were, we also recorded the ILO power measurements from each of the nodes at 15-second intervals. We found that the power measured by the ILO system lags that measured by the WattsUp meter by exactly five minutes. Figure 16 shows both the maximum, minimum, and "present" power reported by the ILO and the power reported by the WattsUp meter during an Indy JouleSort run.



(a) Raw power measurements



(b) Median of each 30 seconds of measurement data

Fig. 16. Power consumed by TritonSort on a representative node during an Indy GraySort run.

In these runs you can see that there is a sharp reduction in power usage about halfway through the sort run. This is a result of the barrier between phases one and two. Due to the natural variation in node performance, some nodes finish phase one earlier than others, and so their power usage is reduced. However, none of the nodes can start phase two until all nodes are done with phase one, which results in the gap visible in Figure 16.

The WattsUp meter's data is more variable during phase two; we suspect that this is due to the fact that the CPU is far more active during phase two than it is during the other phases. However, if we look at the median power reported by the WattsUp meter during each 30-second interval throughout the run, we notice that the WattsUp meter's measurements track the ILO's measurements quite closely.

6. FUTURE WORK

In this section, we discuss our system and present directions for future work.

6.1. Supporting More General Sorting

Two assumptions that we make in our design are that tuples are uniform in size, and that they are uniformly and identically distributed across the input files. TritonSort can be extended to support nonuniform tuple sizes by extending the tuple data structure to keep key and value lengths. The most major modification that this will necessitate will be supporting the in-memory sort of keys in phase two, which will require modifications to the phase two Sorter stage. To support the nonuniform distribution of keys across input files, we plan to implement a new phase that will operate before TritonSort begins in which a random small subset of the input data is scanned, determining a histogram of the key distribution. Using this empirical distribution, we will determine a hash function that spreads tuples across nodes as uniformly as possible.

6.2. Automated Performance Tuning

In the current TritonSort prototype, the sizes of individual buffers, the number of buffers of each type, and the number of workers implementing each stage are determined manually. Key to supporting more general hardware configurations and more general DISC applications is the ability to determine these quantities automatically and dynamically. This automatic selection will need to be performed both statically at design time, and dynamically during runtime based on observed conditions. A stage's performance on synthetic data in isolation provides a good upper bound on its real performance and makes choosing between different implementations easier, but any such synthetic analysis does not take runtime conditions such as CPU scheduling and cache contention into account. Therefore, some manner of online learning algorithm will likely be necessary for the system to determine a good configuration at scale.

6.3. Incorporating SSDs into TritonSort

To achieve nearly sequential speed throughput to the disks, writes must be large. However, limited per-node memory capacity and high memory cost makes it hard to allocate more than 25MB of memory to each WriterBuffer. Here, we discuss a possible use of SSDs to provide high write speeds with much smaller buffers.

If we were to add three 80GB SSDs to each machine, we could set up a pipeline in which these SSDs are divided between the eight Writers, so that each Writer has 30GB of SSD space. The LogicalDiskDistributor passes data for each logical disk to the Writer stage in small chunks, where Writers write them to the SSDs. Assuming 315 logical disks per Writer, this gives each logical disk 95MB of space on the SSD. Because the SSD can handle such a large number of IOPS, there is no penalty for small writes as there is with standard hard drives. Once 80MB of data is written to a single logical disk on the SSDs, the Writer initiates a *sendfilev()* system call that causes a sequential DMA transfer of that data from the SSD to the appropriate intermediate disk. This should lower our memory requirements to 24GB, while permitting extremely large writes. This approach relies on two features: significant PCI bandwidth to support parallel transfers to the SSDs, and an SSD array present in the node able to provide high streaming bandwidth to the SSDs; we will need such an array to simultaneously support over 640MBps of parallel writes and 640MBps of parallel reads to fully utilize the disks.

7. RELATED WORK

The design of TritonSort is based on a long history of disk-to-disk sorting systems and database techniques. In this section, we highlight this related work.

Disk-to-disk sorting and dataflow systems. Good benchmarks are critical for improvements within a particular area, and this is especially true for sorting systems.

The Datamation sorting benchmark [Anon et al. 1985] 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. Over the years, numerous authors have reported the performance of their sorting systems against this benchmark, and we benefit from their insights [Arpaci-Dusseau et al. 1997; Hadoop 2011; Kuszmaul 2007; Nyberg et al. 1995, 1997; Rahn et al. 2009; Wyllie 2005].

NOWSort [Arpaci-Dusseau et al. 1997] 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 [Rahn et al. 2009]. An evaluation of NOWSort done in 1998 [Arpaci-Dusseau et al. 1998] found that its performance was limited by I/O bus bandwidth and poor instruction locality. Modern PCI buses and multicore processors have largely eliminated these concerns; in practice, TritonSort is bottlenecked by disk bandwidth.

An important aspect of efficient disk I/O is taking advantage of the multiple disks that might be attached to a node. A common approach to increasing disk bandwidth is to stripe input data across all disks in a node or, more generally, across all disks in a cluster. AlphaSort [Nyberg et al. 1995] and NOWSort [Arpaci-Dusseau et al. 1997] both employ this method to increase disk bandwidth. NOWSort additionally adjusts the sizes of the stripes on different disks proportional to their speed. Aggarwal and Vitter show that any external sort must do a minimum of two reads and two writes of the data to be sorted [Aggarwal and Vitter 1988]. We have designed TritonSort to ensure that exactly two reads and two writes are issued on each tuple during the entirety of its execution.

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 [Amdahl 1970] and Gray and Putzolu's "five-minute rule" [Gray and Putzolu 1987] for trading off memory and I/O capacity. These guidelines have been reevaluated and refreshed as hardware capabilities have increased.

TritonSort's staged, pipelined dataflow architecture is inspired in part by SEDA [Welsh et al. 2001], a staged, event-driven software architecture that decouples worker stages by interposing queues between them. Other DISC systems such as Dryad [Isard et al. 2007] export a similar model, although Dryad has fault-tolerance and data redundancy capabilities that TritonSort does not currently implement.

Parallel databases. Some of the first systems to explore parallelizing pipelined dataflow systems were parallel databases. One of the first parallel database systems to be deployed on a shared-nothing cluster was the Gamma database machine [DeWitt et al. 1990]. To maximize throughput, Gamma employs *horizontal partitioning*, also called *declustering*. Declustering involves spreading the rows of a relational table across many disks so that queries can be optimized to take advantage of the aggregate bandwidth of all disks in the cluster. Gamma provides three different strategies for declustering a table that can be specified when it is created: round-robin, hash, and range-partitioned. When planning a query, Gamma takes this distribution information into account and tries to execute as much of the query plan without moving data between nodes as possible. In the case of TritonSort, each tuple is moved exactly twice. Our sampling approach mimics Gamma's split table abstraction.

An alternative to the split table approach that focuses on hiding the details of network I/O from operators is known as the exchange operator and was developed as part of the Volcano parallel database system [Graefe 1994]. From the query optimizer's perspective, the exchange operator is just another operator. Any notion of data partitioning and flow control are encapsulated within this operator, allowing all other operators to be implemented and executed without regard to parallelism. Abstractly, the exchange operator exports the traditional iterator interface to other operators. It is typically implemented as a pair of operators, one that produces tuples and one that consumes them. The producer operator pulls data from its child operator(s) and pushes that data to the consumer operator. Each exchange producer for a given operator is connected to every corresponding exchange consumer for that operator in the cluster.

The Fault-Tolerant Load Balancing Exchange (FLuX) [Shah et al. 2003] provides an exchange operator that adjusts to performance faults and can also provide content-sensitive partitioning. While FLuX provides both a fault-tolerance and a load balancing component, its load balancing component is most relevant to our discussion of TritonSort. FLuX is designed for Continuous Query (CQ) environments, where data is continually arriving from some source and must be processed in real time. As such, it cannot afford to have the producer side of the exchange operator block waiting for consumers. Its high-level goal, like RiverDQ's [Arpaci-Dusseau 2003], is to keep the rate at which consumers process data as balanced as possible by minimizing *imbalances*, which can be caused by higher-than-desired skew in the data or performance faults. To deal with short-term imbalances, FLuX places a *transient skew buffer* between the producer side of the FLuX operator and its child operator(s). The transient skew buffer is essentially a buffer that allows FLuX to reorder tuples when transmitting them. If the transient skew buffer has available space, the FLuX producer calls `next` on its child operator and inserts it into the buffer. It then sends tuples to an unblocked consumer, if such a consumer exists. This allows FLuX to service unblocked consumers until the consumers that are blocked due to the imbalance become unblocked.

8. CONCLUSIONS

In this work, we describe the hardware and software architecture necessary to build TritonSort, a highly efficient, pipelined, stage-driven sorting system designed to sort tens to hundreds of TB of data. This efficiency was obtained by selecting a distributed partitioned sort algorithm which ensures that each data item is read and written exactly twice, which is the theoretical lower bound for out-of-core sort implementations. We coupled this algorithm with careful management of system resources to ensure cross-resource balance. The result is that we are able to sort tens of GB of data per node per minute, resulting in 938GB/min across only 52 nodes.

In addition to demonstrating the performance of sort on a particular balanced hardware platform, we also evaluated against two different hardware configurations: one with greater per-node memory capacity, and another with faster disks. We believe the work holds a number of lessons for balanced system design and for scale-out architectures in general, and will help inform the construction of more balanced data processing systems that will bridge the gap between scalability and per-node efficiency. The key lessons we learned in constructing TritonSort include: (1) the importance of partitioning disks based on their expected I/O workload (e.g., sustained read or write), (2) the importance of fine-grained buffering at the application layer to take advantage of application semantics of expected data movement, (3) the high impact of thread scheduling on network performance, and (4) the importance of sustaining high throughput to deliver low-energy workloads.

As compute and disk technology evolves, we believe the techniques presented in this work can be used to construct hardware and software instances that ensure that the system's bottleneck is as close as possible to the underlying capability of the hardware.

ACKNOWLEDGMENTS

We'd like to acknowledge Stefan Savage and Andrew Birrell, as well as the anonymous NSDI 2011 reviewers, for their feedback and suggestions.

REFERENCES

Aggarwal, A. and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Comm. ACM 31*, 9, 1116–1127.

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., S Engupta, S., and Srid-Haran, M. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference*.

Amdahl, G. 1970. Storage and I/O parameters and system potential. In *Proceedings of the IEEE Computer Group Conference.*

Anderson, E. and Tucek, J. 2009. Efficiency matters! In *Proceedings of the Workshop on Hot Topics in Storage and File Systems (HotStorage'09)*.

Anon, Bitton, D., Brown, M., Catell, R., Ceri, S., et al. 1985. A measure of transaction processing power. *J. Datamation*.

Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Culler, D. E., Hellerstein, J. M., and Patterson, D. A. 1997. High-Performance sorting on networks of workstations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Arpaci-Dusseau, R., Arpaci-Dusseau, A., Culler, D., Hellerstein, J., and Patterson, D. 1998. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 90–101.

Arpaci-Dusseau, R. H. 2003. Run-Time adaptation in river. *ACM Trans. Comput. Syst. 21*, 1.

AvocentPDU. 2011. Avocent PM3000V PDU. http://www.avocent.com/Products/Category/Power_Distribution_Units/PM1000_2000_3000_PDUs.aspx.

Bryant, R. E. 2007. Data-Intensive supercomputing: The case for DISC. Tech. rep. CMU-CS-07-128, Carnegie Mellon University.

Dean, J. and Ghemawat, S. 2004. MapReduce: Simpli?ed data processing on large clusters. In *Proceedings of the ACM USENIX Symposium on Operating Systems Design and Implementation*.

Dewitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I., and Rasmussen, R. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Engin. 2*, 1.

Fawnsort. 2010. FAWNSort: Energy-Ef?cient sorting of 10GB. http://sortbenchmark.org/fawnsort_2010.pdf.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.

Graefe, G. 1994. Volcano-An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Engin. 6*, 1.

Gray, J. and Putzolu, G. R. 1987. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Hadoop 2011. Apache hadoop. http://hadoop.apache.org/.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the SIGOPS European Conference on Computer Systems*.

Kuszmaul, B. C. 2007. Kuszmaul, B. C. 2007. TeraByte TokuSampleSort. http://sortbenchmark.org/tokutera.pdf.

Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. 1995. Alphasort: A cache-sensitive parallel external sort. In *Proceedings of the International Conference on Very Large Databases*.

Nyberg, C., Koester, C., and Gray, J. 1997. NSort: A parallel sorting program for NUMA and SMP machines. http://www.ordinal.com/white/whitepaper.html.

Rahn, M., Sanders, P., Singler, J., and Kieritz, T. 2009. DEMSort – Distributed external memory sort. http://sortbenchmark.org/demsort.pdf.

Rasmussen, A., Porter, G., Conley, M., Madhyastha, H. V., Mysore, R. N., Pucher, A., and Vahdat, A. 2011. TritonSort: A balanced, large-scale sorting system. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Rivoire, S., Shah, M. A., Ranganathan, P., and Kozyrakis, C. 2007. Joulesort: A balanced energy-efficiency benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 365–376.

Shah, M., Hellerstein, J., and Franklin, M. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the International Conference on Data Engineering*.

SortBenchMark 2010. Sort benchmark home page. http://sortbenchmark.org/.

WattsUpMeter 2011. WattsUp power meters. http://www.wattsupmeters.com.

Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the SIGOPS Symposium on Operating Systems Principles*.

Wyllie, J. 2005. Sorting on a cluster attached to a storage-area network. http://sortbenchmark.org/2005_SCS_Wyllie.pdf.

YahooCluster. 2008. Scaling Hadoop to 4000 nodes at Yahoo! http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

Received May 2011; revised April 2012; accepted August 2012