TritonSort: A Balanced Large-Scale Sorting System

Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha[†]
Radhika Niranjan Mysore, Alexander Pucher[*], Amin Vahdat
UC San Diego, UC Riverside[†], and Vienna University of Technology[*]

Abstract—**We present TritonSort, a highly efficient, scalable sorting system. It is designed to process large datasets, and has been evaluated against as much as 100 TB of input data spread across 832 disks in 52 nodes at a rate of 0.916 TB/min. When evaluated against the annual Indy GraySort sorting benchmark, TritonSort is 60% better in absolute performance and has over six times the per-node efficiency of the previous record holder. In this paper, 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 cheaper systems that are able to do the same work or provide the ability to address significantly larger problem sets with the same infrastructure.

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[8], 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 recently, including MapReduce[9], the Google File System[11], Hadoop[22], and Dryad[14]. 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 [23].

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[4] 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[4]. The largest known industrial Hadoop clusters achieve only 20 Mbps 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 data sets 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 co-located 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[19] resulted in a system that was able to sort 100TB of input tuples in about 60% 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. 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. Building such systems will either enable significantly cheaper 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 paper 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 paper, 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 [19], our input data comprises 100 byte tuples with a 10 byte key and 90 byte value. 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. Our goal is to design and implement a sorting system that can sort datasets of the targeted size while achieving a favorable tradeoff 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 non-trivial 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.

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 [1]. Since every additional read and write increases the time to sort, we seek to achieve exactly this lower bound to maximize system performance.

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

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

Network	
Type	Cost/port
1 Gbps Ethernet	$33
10 Gbps 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

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

Third, 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. To determine the best storage option given these requirements, we survey a range of hard disk options shown in Table 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). To allow 16 disks to operate at full streaming I/O throughput, we require storage controllers that are able to sustain at least 1600 MBps of streaming 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. 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. 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.

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

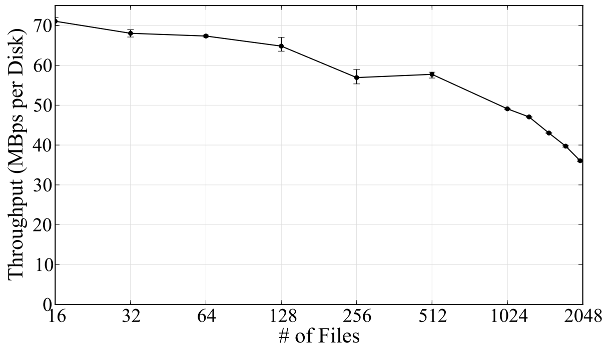

Figure 1: Performance of a Heaper-Merger sort implementation in microbenchmark on a 200GB per disk parallel external merge-sort as a function of the number of files merged per disk.

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 data set 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 1, 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 above 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 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 mergesort-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 100 TB 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 2 and 7 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.

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 $size_{LD}$ 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 paper, 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 non-uniform 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 2, 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.

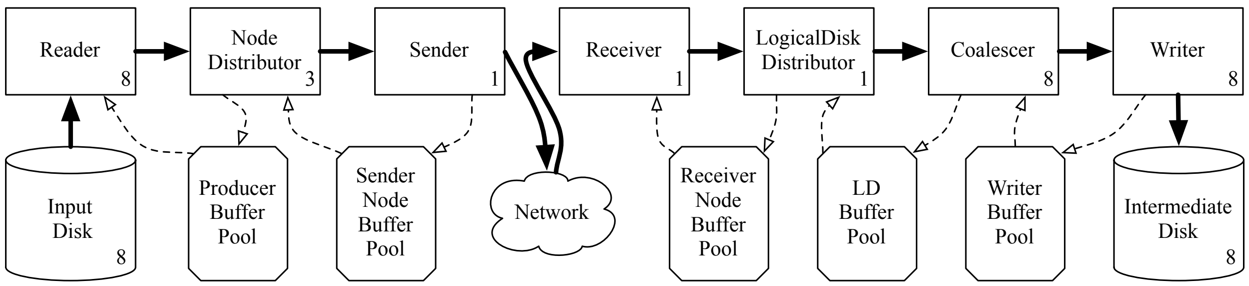

Figure 2: Block diagram of TritonSort's phase one architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

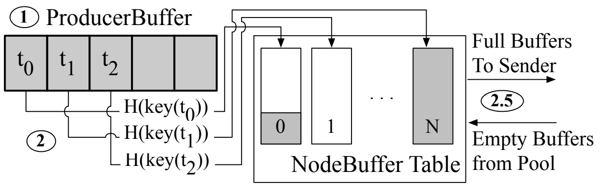

Figure 3: The NodeDistributor stage, responsible for partitioning tuples by destination node.

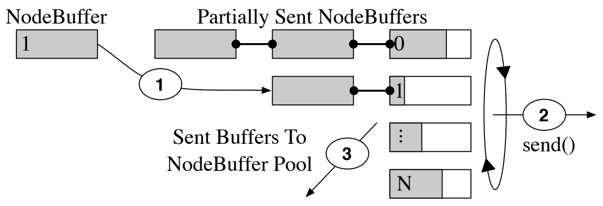

Figure 4: The Sender stage, responsible for sending data to other nodes.

NodeDistributor: A NodeDistributor (shown in Figure 3) 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 in size for the scales we consider in this paper.

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 4) 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 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 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.

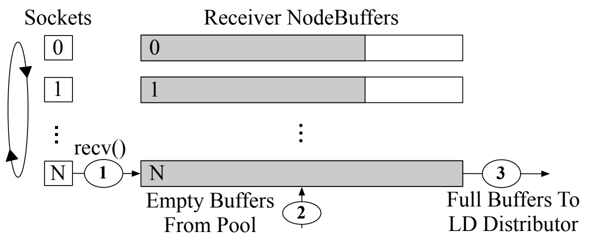

Figure 5: The Receiver stage, responsible for receiving data from other nodes' Sender stages.

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 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 5, 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. 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 6, 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.

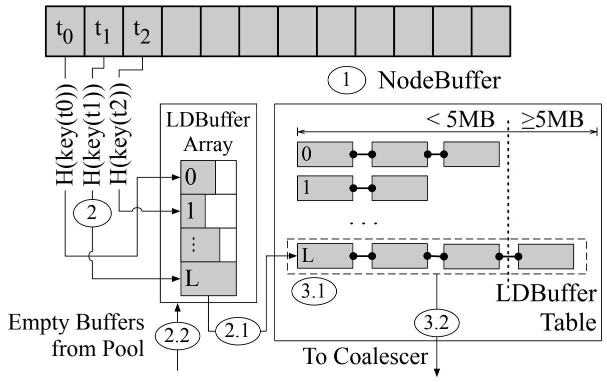

Figure 6: The LogicalDiskDistributor stage, responsible for distributing tuples across logical disks and buffering sufficient data to allow for large writes.

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 24 GB 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($d$) $\geq$ 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 3-10 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 (~10MB 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. 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.

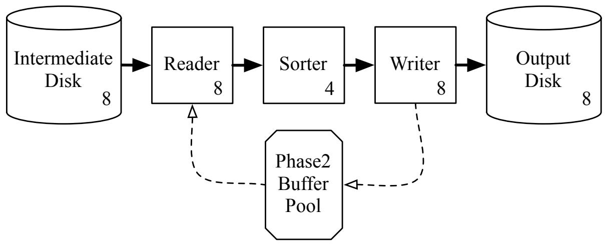

Figure 7: Block diagram of TritonSort's phase two architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

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. 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

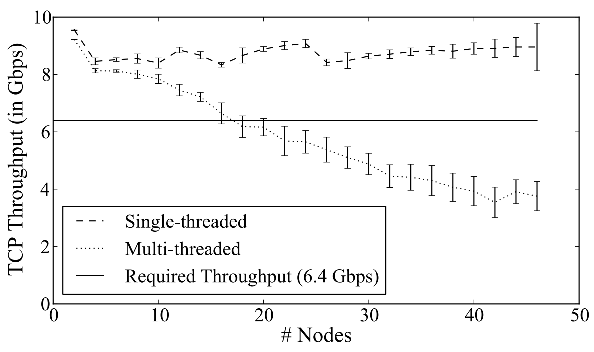

Figure 9: Comparing the scalability of single-threaded and multi-threaded Receiver implementations

(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 24 GB 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 amount of CPU time it requires to process a work unit. Figure 8 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.

4 Optimizations

In implementing the TritonSort architecture, we learned that several non-obvious 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 100 MBps of sequential read throughput (Table 1), the network must be able to sustain 6.4 Gbps 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. Reasons for this include commodity datacenter network hardware, incast, queue buildup, and buffer pressure[2]. Since we could not employ a strategy like that presented in [2] to provide fair but high bandwidth flow rates among the senders, we chose instead to artificially rate limit each flow at the Sender stage to its calculated fair share by forcing the sockets to be receive window limited. This works for TritonSort because 1) each machine sends and receives at approximately the same rate, 2) all the nodes share the same RTT since they are interconnected by a single switch, and 3) our switch does not impose an oversubscription factor. In this case, each Sender should ideally send at a rate of $(6.4/N)$ Gbps, or 123 Mbps with a cluster of 52 nodes. Given that our network offers approximately $100\mu sec$ RTTs, a receiver window size of $8 - 16$ KB ensures that the flows will not impose queue buildup or buffer pressure on other flows.

Initially, we chose a straightforward multi-threaded 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 9 shows that this multi-threaded approach does not scale well with the number of nodes, dropping below 4 Gbps at scale. This poor performance is due to thread scheduling overheads at the end hosts. 16 KB TCP receive buffers fill up much faster than connections that are not window-limited. At the rate of 123 MBps, a 16 KB buffer will fill up in just over 1 ms, causing the Sender to stop sending. Thus, the Receiver stage must clear out each of its buffers at that rate. 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, non-blocking 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 9 shows that this design scales well as the cluster grows.

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

Figure 8: Median stage runtimes for a 52-node, 100TB sort, excluding the amount of time spent waiting for buffers.

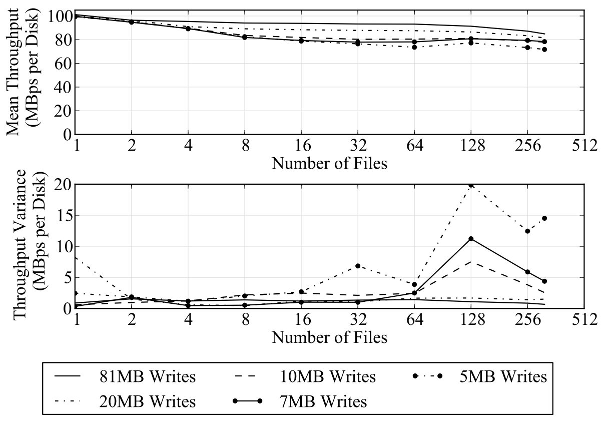

Figure 10: Microbenchmark indicating the ideal disk throughput as a function of write size

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 10 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, e.g., 80 MB. 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 80 MB 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 pre-allocate 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 100 TB sort, we found that we could write to the interior cylinders of the disk at an average rate of 64 MBps. 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 84 MBps.

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 90 MBps.

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 sub-optimally 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.

To address this problem, we must effectively 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.

4.6 System Call Behavior

In the construction of any large system, there are always idiosyncrasies in performance that must be identified and corrected. For example, we noticed that the sizes of arguments to Linux `write()` system calls had a dramatic impact on their latency; issuing many small writes per buffer often yielded more performance than issuing a single large write. One would imagine that providing more information about the application's intended behavior to the operating system would result in better management of underlying resources and latency but in this case, the opposite seems to be true. While we are still unsure of the cause of this behavior, it illustrates that the performance characteristics of operating system services can be unpredictable and counter-intuitive.

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), 24 GB of memory, and 16 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 10 Gbps network interface. The network interconnect we use is a 52-port Cisco Nexus 5020 datacenter switch. The servers run Linux 2.6.35.1, and our implementation of TritonSort is written in C++.

5.2 Comparison to Alternatives

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 [18], sorted slightly over 100TB of data on 195 nodes at a rate of 564 GB per minute. TritonSort currently sorts 100TB of data on 52 nodes at a rate of 916 GB per minute, a factor of six improvement in per-node efficiency.

Intermediate Disk Speed (RPM)	Logical Disks Per Physical Disk	Phase 1 Throughput (MBps)	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

Table 2: Effect of increasing speed of intermediate disks on a two node, 500GB sort

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. Evaluating the performance benefits of SSDs is the subject of future work.

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 data set. 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 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 2. 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.

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

Table 3: Effect of increasing the amount of memory per node on a two node, 2TB sort

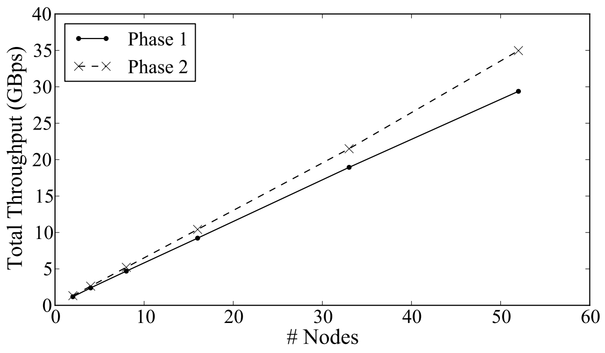

Figure 11: Throughput when sorting 1 TB per node as the number of nodes increases

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 Table 3, 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.

5.4 TritonSort Scalability

Figure 11 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.

6 Discussion and 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 non-uniform 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 non-uniform 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 setup a pipeline in which these SSDs are divided between the eight Writers, so that each Writer has 30 GB 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 95 MB 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 80 MB 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 24 GB, 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 640 MBps of parallel writes and 640 MBps of parallel reads to fully utilize the disks.

7 Related Work

The Datamation sorting benchmark[5] 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, and we benefit from their insights[18, 15, 21, 6, 17, 16]. 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" [13] for trading off memory and I/O capacity. These guidelines have been re-evaluated and refreshed as hardware capabilities have increased.

NOWSort[6] 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[18]. An evaluation of NOWSort done in 1998[7] 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.

TritonSort's staged, pipelined dataflow architecture is inspired in part by SEDA[20], a staged, event-driven software architecture that decouples worker stages by interposing queues between them. Other DISC systems such as Dryad [14] export a similar model, although

Dryad has fault-tolerance and data redundancy capabilities that TritonSort does not currently implement.

We are further informed by lessons learned from parallel database systems. Gamma[10] 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[12] in that it abstracts data partitioning, flow control, parallelism and data distribution from the rest of the system.

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. Through careful management of system resources to ensure cross-resource balance, we are able to sort tens of GB of data per node per minute, resulting in 916 GB/min across only 52 nodes. 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.

9 Acknowledgments

This project was supported by NSF's Center for Integrated Access Networks and NSF MRI #CNS-0923523. We'd like to thank Cisco Systems for their support of this work. We'd like to acknowledge Stefan Savage for providing valuable feedback concerning network optimizations, and thank our shepherd Andrew Birrell and the anonymous reviewers for their feedback and suggestions.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 1988.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] G. Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, 1970.

[4] E. Anderson and J. Tucek. Efficiency matters! In *HotStorage*, 2009.

[5] Anon et al. A Measure of Transaction Processing Power. *Datamation*, 1985.

[6] 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.

[7] 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.

[8] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, CMU, 2007.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 1990.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[12] G. Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 1994.

[13] 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.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

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

[16] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A cache-sensitive parallel external sort. In *VLDB*, 1995.

[17] C. Nyberg, C. Koester, and J. Gray. NSort: a Parallel Sorting Program for NUMA and SMP Machines, 1997.

[18] M. Rahn, P. Sanders, J. Singler, and T. Kieritz. DEMSort – Distributed External Memory Sort, 2009. http://sortbenchmark.org/demsort.pdf.

[19] Sort Benchmark Home Page. http://sortbenchmark.org/.

[20] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[21] J. Wyllie. Sorting on a Cluster Attached to a Storage-Area Network, 2005. http://sortbenchmark.org/2005_SCS_Wyllie.pdf.

[22] Apache hadoop. http://hadoop.apache.org/.

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