TritonSort

Alexander Rasmussen
University of California San Diego

Harsha V. Madhyastha
University of California San Diego

Radhika Niranjan Mysore
University of California San Diego

Michael Conley
University of California San Diego

Alexander Pucher
Vienna University of Technology

George Porter
University of California San Diego

Amin Vahdat
University of California San Diego

Abstract

We present TritonSort, a sorting system designed to maximize system resource utilization. We present the results of the Indy GraySort and MinuteSort benchmarks. TritonSort is able to sort 100 TB of 100-byte records at a rate of 0.582 TB/minute, and is able to sort 1014 GB of data in under 60 seconds.

1 Architecture

TritonSort is composed of a series of *tasks* that typically consist of some simple processing on a small piece of data. We call the small pieces of data that tasks process *work units*. Tasks are composed together into a directed graph so that workers from one task produce work units for workers in a subsequent task. In the GraySort version of TritonSort, tasks are further subdivided into two *phases*, with a distributed barrier between the first and second phase.

A fixed collection of *workers*, each of which is a thread, perform a given task. A single *worker tracker* for each task coordinates the passing of work units to and from the workers performing that task. For brevity, we will name workers based on the task that those workers perform; for example, workers performing the "read" task are simply called "readers".

When a worker is ready to send a work unit to the next stage, it can direct the next stage's tracker either to give the work unit to a particular worker or to give the work unit to the next worker that runs out of work to do.

When a task runs out of work to do, the task's workers shut down. When a task tracker is notified that an upstream task tracker has shut down, it waits until all outstanding work units have been processed and then begins shutting down itself. In this way, the entire graph is shut down at the end of the phase.

Tasks that are sources in the task graph do not receive work units from other tasks, but rather generate work units themselves. In TritonSort, these are tasks that receive from the network or read from a disk. Tasks that are sinks in the task graph do not produce work units for other tasks; these are tasks that write to a disk or transfer data over the network.

1.1 Disks

We subdivide the disks on each machine into equal numbers of input and output disks. In the first phase, input data is read sequentially from the input disks and intermediate data is written to the output disks. In phase two the disks' roles are reversed; intermediate data is read from the output disks and the final output is written to the input disks.

Each disk in the cluster stores tuples whose keys are in a given range. Physical disks are further subdivided into a number of *logical disks*, each of which is a file. Each logical disk on a given physical disk is responsible for its own disjoint sub-range of that physical disk's range. In this way, every tuple can be mapped to a destination logical disk according to that tuple's key.

We explore how the number of logical disks per physical disk is chosen in more detail in Section 2.4.

2 GraySort Architecture

TritonSort's architecture for the GraySort benchmark aims to sort large datasets by reading and writing each tuple exactly twice, which is the theoretical minimum I/O when the amount of memory in the system is less than the amount of data to be sorted [1]. The task pipeline is designed so that (ideally) all workers constantly have work units to process, thus maximizing utilization of disk I/O, network bandwidth, and CPU processing power.

2.1 Phase One: Distribute

The goal of TritonSort's first phase is to read all tuples from the input disks and transfer each tuple to the appropriate intermediate logical disk. The tasks that make up TritonSort's first phase are as follows:

Read: read from an input file into a collection of 80 MB in-memory *producer buffers*. When a producer buffer becomes full, pass it to the next stage. There is one reader for each input disk.

Spray Tuples: scan through a producer buffer, hashing each of its tuples to determine that tuple's destination logical disk. Copy each tuple into an in-memory *send buffer* appropriate to its destination. When a large enough group of send buffers become full, pass them to the next phase as a group. The number of tuple sprayers is variable, but is usually set to be equal to the number of readers.

Send: transmit a group of send buffers to the appropriate destination node. There is one sender responsible for communicating with each node, and tuple sprayers explicitly pass groups of send buffers to the appropriate sender.

Connect: establish connections with remote senders. When a connection is established, the connection's socket descriptor is passed to the first available receiver. That receiver reads from the connection until it closes, at which time it notifies the connector that the socket has been closed. A connector waits until all sockets are closed before tearing itself down.

Receive: receive send buffers into a collection of circular buffers, one per logical disk. There is one receiver per node, and that receiver is responsible for receiving all data from that node.

Write: repeatedly write data from the most full circular buffer to its corresponding logical disk. There is one writer per physical intermediate disk.

2.2 Phase Two: Sort

Once each tuple has been transferred to its appropriate logical disk, each logical disk must be sorted. The sorting of logical disks is done with three tasks:

Phase Two Read: read an entire logical disk into an 850MB in-memory buffer. There is one phase two reader per intermediate disk.

Sort: sort the tuples in a buffer using a variant of radix sort. The number of sorters is variable, and is currently set to half the number of phase two readers.

Phase Two Write: write a buffer to a file on the appropriate input disk. There is one phase two writer per input disk.

The tasks in both phases are logically connected to each other as shown in Figure 1.

2.3 MinuteSort Architecture

For the MinuteSort benchmark, we modify our architecture as follows. In the first phase, as before, we read the input data and spray 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.

2.4 Logical Disks

An operator specifies the number of logical disks for each physical disk before running TritonSort. The number of logical disks per physical disk is determined differently depending on the architecture. For GraySort, the number of logical disks is chosen such that three logical disks for each physical disk can be resident in memory at the same time during phase two; in this way, each worker in phase two will always have a logical disk to process at any given time. For MinuteSort, the number of logical disks is chosen such that all logical disks for a given node can be resident in that node's memory simultaneously; this is necessary because the logical disks are not written to disk before they are sorted.

3 Testbed Setup

Our testbed consisted of 52 HP ProLiant DL380 G6 servers, although we use different numbers of servers for different benchmarks. Each server has two quad-core Intel Xeon E5520 processors, clocked at 2.27 GHz, and 24 GB of RAM. Each server also hosts 16 2.5-inch 500 GB, 7200 RPM SATA hard drives. 40 of the machines use HP Seagate MM0500EANCR drives that are enterprise-grade and therefore have a much higher reliability. The remaining machines use Seagate Momentus 7200.4 drives, which are consumer-grade.

Each machine is equipped with a 1Gbps on-board network card as well as a Myricom 10Gbps network card. Both network cards run unmodified Ethernet. All the machines in our testbed are inter-connected via a Cisco Nexus 5020 switch, which provides 10 Gbps connectivity between all pairs.

All machines are running version 2.6.32.8 of the Linux operating system. All disks use the ext4 filesystem.

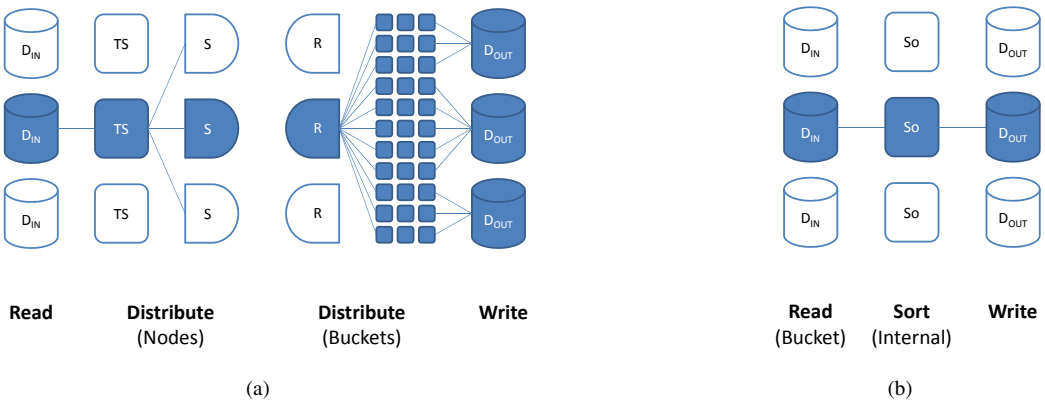

Figure 1: Architecture pipeline for (a) phase one and (b) phase two.

4 Experimental Setup

TritonSort is bootstrapped by a pair of shell scripts. The user executes a central shell script (called the coordinator) that is responsible for starting TritonSort on all cluster nodes. This script is assumed to run on the same subnet as the cluster nodes. The coordinator starts a script called a monitor as a daemon on each cluster node. After a monitor finishes initializing its internal state, it opens a TCP pipe to the coordinator and announces that the node it is monitoring is ready to run. The monitor then waits for a UDP broadcast packet. Once the coordinator has established connections to all monitors and all monitors are ready, the coordinator sends a UDP broadcast indicating that the experiment is ready to begin. Once a monitor receives the UDP broadcast, it starts the TritonSort instance for its node immediately.

Each monitor checks the status of its TritonSort instance every tenth of a second. While its TritonSort instance is still running, the monitor transmits a keep-alive message to the coordinator via its established TCP pipe every second. Once the TritonSort instance has finished, the monitor notifies the coordinator that it is done.

The coordinator and monitor scripts are not strictly part of TritonSort, and were built primarily for experimental convenience. For our MinuteSort runs, where strict timing is essential, we measure the elapsed time of the sort as the time between when the coordinator sends the UDP broadcast to start the experiment and when the last monitor reports to the coordinator that it is done. This ensures that the measured time encompasses the starting up and shutting down of all TritonSort instances. For GraySort, we choose to underestimate our performance by a few seconds to make logging simple, and time the duration between when the monitor starts at the beginning of phase one and when it stops at the end of phase two.

5 Evaluation

In this section, we present the results of our sort benchmark runs.

5.1 Minute Sort

We ran TritonSort in its MinuteSort configuration on 52 nodes with 19.5 GB per node for a total of 1014 GB of data. TritonSort was configured to use 12 logical disks per physical disk for a total of 192 logical disks. We performed 15 consecutive trials. For these trials, TritonSort's median elapsed time was 57.9 seconds, with a maximum time of 59.9 seconds, a minimum time of 55.9 seconds, and an average time of 57.9 seconds. All times were rounded to the nearest tenth of a second.

5.2 GraySort

We ran GraySort on 100 TB (1,000,000,000,000 100-byte records) of data with 47 nodes. Each physical disk stores 315 logical disks.

We verified that the resulting output's checksum matches the input's checksum. We sorted the data in 10318 seconds, for a total rate of approximately 0.582 TB per minute.

References

[1] AGGARWAL, A., AND VITTER, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM* (1988).