TritonSort 2014

Michael Conley[1], Amin Vahdat[2,1], George Porter[1]
[1]*UC San Diego,* [2]*Google*
{mconley,vahdat,gmporter}@cs.ucsd.edu
http://tritonsort.eng.ucsd.edu/

Abstract

We present TritonSort, a sorting system designed to maximize system resource utilization. We present the results for: Indy GraySort, Daytona GraySort, Indy MinuteSort, Indy CloudSort, and Daytona CloudSort.

1 Architecture

TritonSort runs a variant of the Themis MapReduce architecture to perform the sort benchmark. We use identity functions for `map()` and `reduce()` to leave the original data set unmodified. The shuffle phase inherent to MapReduce causes the data records to end up on disk in sorted order. For a detailed presentation of Themis MapReduce, please consult [3].

1.1 Handling Arbitrary Record Sizes

Themis MapReduce uses a header attached to each record that designates the record's key and value lengths. This header allows the system to correctly handle records of any size, including variably sized records. In the case of fixed sized records, this header is not materialized to I/O devices, which prevents extra disk and network overhead in our benchmark runs.

The actual sort algorithm used in the *sort-and-reduce* phase of Themis is dynamically chosen based on properties of the data set, as well as memory constraints. For fixed size records, we use a custom radix sort implementation when there is enough available system memory. For records of arbitrary size, Themis falls back to quick sort, and therefore satisfies the Daytona requirement of handling variably sized records.

1.2 Handling Data Skew

Themis runs a sampling step before every job to create a distribution model for the intermediate data. This distribution allows Themis to evenly partition map output data even in the face of data skew. Typically a sample rate between 0.1% and 1% is sufficient to create evenly sized partitions from skewed data sets.

In order for this sampling step to be efficient, each node computes partition boundaries from its local sample data. These partition boundaries are sent to a central coordinator node, which picks the median boundary key for each partition as the official partition boundary. This mechanism trades accuracy for performance; the median key may not yield the best partition sizes, but it circumvents the need to centralize the much larger set of sampled data on the coordinator node. In practice we have found this trade-off to be highly beneficial at scale.

Computing partition boundaries locally can be problematic if a data set's skew cannot be determined from the data on a single node. In this case, two nodes might create very different distribution models of the data set based on their locally sampled data. To combat this case, locally sampled data is randomly shuffled to give all nodes approximately the same view of the skewed data set. This shuffle leverages the all-to-all capabilities of the network and is therefore very efficient. In fact, the total time taken by the sampling step was no more than 81 seconds in any of our Daytona entries, with a minimum time of 34 seconds.

1.3 Handling Large Partitions

In the event that the sampling mechanism fails to create evenly sized partitions, Themis can still operate correctly, even if the map output partitions are larger than the size of memory. Any files that are larger than a user-configurable size at the end of the *map-and-shuffle* phase are marked as *large partitions*. Large partitions do not flow through the normal *sort-and-reduce* phase in Themis. Instead, a separate *split-and-merge* phase handles these corner cases after the bulk of the data has been completely processed.

Resource	i2.8xlarge	r3.4xlarge
Physical Processor	Intel Xeon E5-2670 v2	Intel Xeon E5-2670 v2
vCPU cores	32	16
Memory	244 GiB	122 GiB
Networking	10 Gb/s	2 Gb/s
Local Storage	8x 800 GB SSD	1x 320 GB SSD
Network Storage	-	8x 135 GiB EBS gp2

Table 1: EC2 configurations used

The *split* sub-phase of split-and-merge divides large partitions into reasonably sized *chunks* of records which can be sorted in memory before being written back to disk. This sub-phase involves one sequential read of each large partition, and one sequential write of each sorted chunk. The *merge* sub-phase reads the heads of each sorted chunk file into memory and merges them to create a stream of sorted records for the large partition. This stream of records is passed through the `reduce()` function before being written back to disk. The end result is an output file, possibly larger than the size of memory, that has been completely sorted and reduced. While this requires twice as many I/O operations as the typical sort-and-reduce phase, we rarely expect to use the split-and-merge phase, so its overhead is negligible.

One caveat is that the `reduce()` function cannot require all of its records to fit in memory, but this is not problematic for sort because the `reduce()` function simply passes records through unmodified.

1.4 Data Replication

Themis supports the replication of input and output files across nodes in the cluster. If replication is enabled, a configurable number of copies of the `reduce()` output data for each partition are sent to different nodes in the cluster to be written to disk. In this way, the input and output data sets are capable of surviving a single node failure.

2 Environment

We run all benchmarks on Amazon's Elastic Compute Cloud (EC2) public cloud infrastructure [2]. For GraySort and MinuteSort, we use the `i2.8xlarge` virtual machine instance type described in Table 1. For CloudSort, we use the `r3.4xlarge` instance type in tandem with Amazon's Elastic Block Storage (EBS) persistent network-attached storage system [1].

We launch all instances in the same placement group to improve the odds of getting full bisection network bandwidth. Additionally, the `r3.4xlarge` instances are launched in "EBS-optimized" mode to improve bandwidth to the EBS storage service.

In general, servers and networking resources in the public cloud are shared resources. The "Enhanced Networking" feature of these instances attempts to provide some network isolation using a single placement group, although the network itself is still a shared resource.

All virtual machines run the Linux operating system, although versions vary depending on the benchmark. Our Indy GraySort and Indy MinuteSort benchmarks run Linux version 3.10.48 and a custom Amazon Machine Image (AMI) based on ami-76817c1e, which is version 2014.03.2 of the HVM flavor of Amazon Linux. Daytona GraySort and both CloudSort benchmarks run Linux version 3.10.53 and a custom AMI based on ami-51736438, which is version 2014.03.0 of the HVM Amazon Linux distribution.

Each storage device, be it local SSD or EBS volume, is configured with a single XFS partition. Each XFS partition is mounted with the `noatime`, and `discard` mount options.

All EBS volumes are 135 GiB General Purpose SSD (gp2) volumes. We attach eight volumes to each VM in the CloudSort benchmark to maximize throughput to EBS. Input and output files are spread evenly across EBS volumes with each file containing whole records, and we read and write to files on different EBS volumes simultaneously.

Amazon automatically replicates EBS volumes across availability zones, which provides data durability. EBS volumes are designed for 5 nines of availability, which makes them an ideal choice for persistent input and output data storage.

In all the benchmarks that follow, scale was limited only by the number of resources available on Amazon EC2 and EBS.

3 Benchmarks

TritonSort 2014 uses the same cluster coordination scripts used in TritonSort 2011 [4]. In all benchmarks, we measure elapsed running time from the moment that the central coordinator script instructs the nodes to

Benchmark	# Nodes	Time	Rate	Checksum
Indy GraySort	178	888 s	6.76 TB/min	`746a51007040ea07ed`
Daytona GraySort	186	1378 s	4.35 TB/min	`746a51007040ea07ed`
Daytona GraySort (Skewed)	186	1943 s	3.09 TB/min	`746a50ec9293190d87`

Table 2: 100TB GraySort results

Benchmark	# Nodes	Data Size	Median Time	Checksum
Indy MinuteSort	178	4094 GB	58.8 s	`4c41ae646f1bb3bff`

Table 3: MinuteSort results

launch Themis to the moment that the last node reports to the central coordinator that its Themis binary has completed. This gives an elapsed time measurement external from Themis itself, in compliance with the benchmark rules.

For all benchmarks, we run the **valsort** application on input and output data sets to verify that 1) the final output is sorted, and 2) the input and output checksums match.

While we did experience failures while running on Amazon EC2, they were infrequent enough where we also experienced large stretches of time, on the order of hours, with no failures, thus satisfying the Daytona requirement of being able to run continuously for an hour without failure. Because of this, we were able to use the same set of virtual machines for the both the uniform and skewed Daytona GraySort runs, as well as all 3 of our CloudSort entries.

3.1 GraySort

The results of our 100TB GraySort benchmarks are shown in Table 2. Our Indy benchmark sorts 100 TB of data in 888 seconds for a total rate of 6.76 TB/min on 178 `i2.8xlarge` virtual machines. Our Daytona benchmark runs in 1378 seconds on 186 `i2.8xlarge` instances for a total rate of 4.35 TB/min. In compliance with benchmark rules, we also run a 100 TB skewed sort, which completes in 1943 seconds, which is 41% slower than our uniform sort time.

We disable sampling, large partition handling, and replication in the Indy GraySort benchmark. This benchmark uses a predetermined partitioning function that assumes a uniform data distribution. The Daytona variant samples 0.4% of the data to construct partition boundaries. The large partition handling routine checks for large partitions at the end of the sort-and-reduce phase. Finally we enable twofold replication of input and output data sets to survive single node failures.

We observed no duplicate keys in the uniform data set. However, we observed 30285373571 duplicate keys in the skewed data set.

3.2 MinuteSort

Our Indy MinuteSort results are summarized in Table 3. We sort 4094 GB of data on 178 `i2.8xlarge` instances. We perform 15 consecutive trials and report a median elapsed time of 58.8 seconds, with a maximum time of 59.8 seconds and a minimum time of 57.7 seconds, for an average of 58.7 seconds.

Because MinuteSort uses a much smaller data set than GraySort, we can keep data in memory during the shuffle phase. In this case, we do not spill data to disk between the two phases of Themis. Instead, shuffled map output records are stored in memory on the reducer nodes until the map-and-shuffle phase completes. At this time, in-memory partitions are sorted and then reduced, before being written to output disks.

As with Indy GraySort, we disable sampling, large partition handling, and data replication in the Indy MinuteSort benchmark.

We observed no duplicate keys in the MinuteSort data set.

3.3 CloudSort

Our results for both Indy and Daytona CloudSort are summarized in Table 4. We sort 100 TB of data in three consecutive trials with elapsed times 3093.92 seconds, 2913.92 seconds, and 2934.45 seconds, which yields our benchmark submission average time of 2980.76 seconds.

We use 330 instances of the `r3.4xlarge` type, which has an on-demand price of $1.400 per hour. However, we launch these instances in "EBS-optimized" mode, which increases their hourly cost to $1.500. Thus the dollar costs of the VMs alone comes to $425.41, $400.66, and $403.49.

We attach eight 135 GiB gp2 EBS volumes to each instance, yielding an overly conservative 382.7 TB of persistent network attached storage. These volumes have a price of $0.10 per GiB per month. We compute the prorated cost using a 30-day month, which brings the EBS costs to $42.54, $40.07, and $40.35. This gives total per-sort costs of $467.95, $440.73, and $443.84. Thus we

Benchmark	# Nodes	Average Time	Average Cost	Checksum
Indy CloudSort	330	2980.76 s	$450.84	`746a51007040ea07ed`
Daytona CloudSort	330	2980.76 s	$450.84	`746a51007040ea07ed`
Daytona CloudSort (Skewed)	328	2944.34 s	$442.63	`746a50ec9293190d87`

Table 4: 100TB CloudSort results

have an average total cost of $450.84.

Because we enter in the Daytona category, we enable sampling and large partition handling. All Cloudsort attempts sample 0.3% of the input data set. The EBS persistent storage system allows our input and output data sets to survive node failures without explicitly replicating data at the application level. Therefore, we do not employ any kind of application-level replication or RAID for CloudSort. In compliance with Daytona benchmark rules, we run the 100 TB skewed data set on 328 `r3.4xlarge` instances in 2944.34 seconds, which yields a cost of $442.63, which is actually less than the cost of sorting the uniform data set.

We observed no duplicate keys in any of the the uniform data sets. However, we observed 30285373571 duplicate keys in the skewed data set.

4 Acknowledgments

We would like to thank Alexander Rasmussen for his significant contributions to the TritonSort and Themis projects. Additionally we would like to thank Praveen Gujar, Chad Schmutzer, Ann Merrihew, Amy Hogenhurt, and Jonathan Fritz at Amazon.com for provisioning EC2 resources for our benchmark runs. We would also like to thank Rahul Pathak, also from Amazon.com, for both providing EC2 credits and helping us acquire cluster resources. Finally we'd like to thank the sort benchmark organizers, Mehul Shah, Chris Nyberg, and Naga Govindaraju. This project was sponsored in part by the Amazon Educational Grant program, by the UCSD Center for Networked Systems, and by grants from the National Science Foundation (CNS-#116079 and CNS-0964395). This project was supported by a donation from FusionIO.

References

[1] Amazon Elastic Block Store (Amazon EBS). `http://aws.amazon.com/ebs/`.

[2] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`.

[3] Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. Themis: An I/O-Efficient MapReduce. In *3rd ACM Symposium on Cloud Computing (SOCC)*, October 2012.

[4] TritonSort 2011. `http://sortbenchmark.org/2011_06_tritonsort.pdf`.