Indy Gray Sort and Indy Minute Sort

Dasheng Jiang
Baidu Inc. & Peking University
July, 2014

Overview

We have built a sorting system to improve performance of Indy Minute Sort and Indy Gray Sort on a large cluster. The reported results are:

Indy Minute Sort, sort 7TB in 56.69s on 993 machines
Indy Gray Sort, sort 100TB in 716.10s on 982 machines

System Configuration

Machines: 993 nodes for Minute Sort and 982 machines for Gray Sort, one as master and the rest as slaves
Operating System: Red Hat Enterprise Linux AS release 4 (Nahant Update 3)
Processors: 2 * Intel Xeon(R) E5-2450, 2.10GHz, 32 cores
Memory: 192GB, 1333MHZ
Disks: 8 * 3TB, 7200rpm SATA
Network: 10Gb/s, full duplex, 3:1 subscription
JDK: jdk6u45

Indy vs. Daytona

Indy and Daytona are two very different kinds of benchmark on the large cluster. Former system tends to treat Indy as a special case of Daytona and using the same code and configuration with little modification. However, the less restricted Indy sort is actually an attempt to archive the theoretical upper limit of the hardware, whatever the bottleneck might be. While Daytona need a system providing fault tolerance which will degrade the performance inevitably.

We would rather explore the hardware limit as the target cluster contains thousands of machines with large memory, so we aim at Indy instead of Daytona. We find TritonSort[1] was a well-designed system with main focus on disk performance and we extend their experiments from 52 machines to thousands.

Architecture

TritonSort[1] was the past winner which archived near optimal performance on a single rack. In our system we use the architecture of TritonSort with small modifications. We will briefly introduce the entire pipeline of the workflow. Please consult [1,2] for detailed design choice.

TritonSort splits the sorting process into two phases. Phase one reads the input, sends records over network and writes on the receiver nodes. Node distributor ensures data volume is approximately even among all receivers. Logical Disk Distributor ensures each output file is small enough to be processed in memory in phase two. Coalescer ensures each write request is large enough to guarantee disk performance.

The file written on Intermediate Disk is a logical unsorted version of output file. Phase two simply reads the logical file, sorts it and writes to local disk.

The pictures below are the pipeline of phase one & phase two in TritonSort. Only the thread number may differ from the original version.

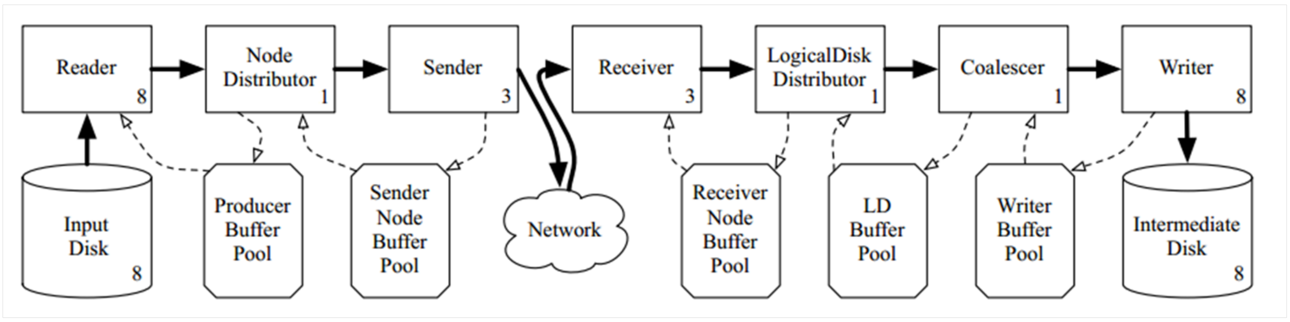

Figure 1: Pipeline of phase one in TritonSort Architecture, from [1]

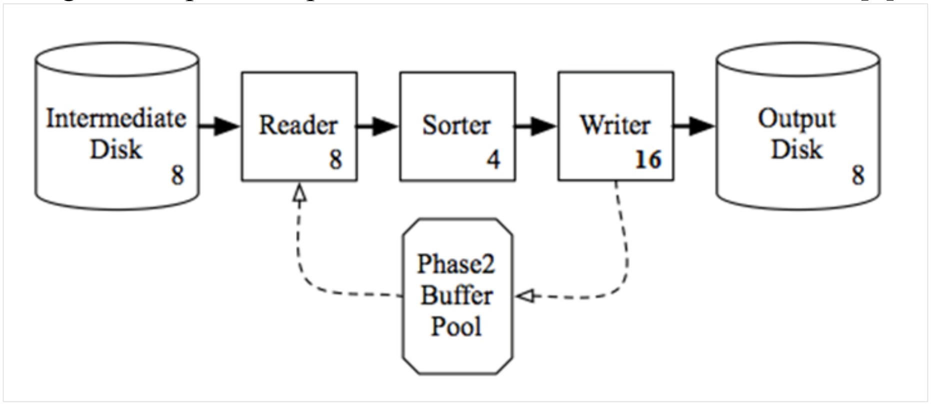

Figure 2: Pipeline of phase two in TritonSort Architecture, from [1]

We have several small modifications to the original architecture. A key difference between TritonSort and our system is the number of machines which indicates we have far more available memory. In such scenario, we add application level cache to ensure that no intermediate data are written to local disk in the Minute Sort. This removes possible bottleneck in the Writer stage of phase one and Reader stage of phase two. In Gray Sort, only a portion of the data are cached.

With more intermediate data being cached, disk IO is less likely to become the bottleneck in phase one. So in our configuration, there are no reason to separate disks into two sets as in TritonSort. The Intermediate Disk in the pictures above are equivalent to Input/Output Disk.

Besides, the intermediate data are evenly distributed among all disks (not only machines) in TritonSort. In larger cluster with thousands of disk, performance could vary remarkably. We choose to evenly distribute data among machines. But the amount of data read/written on each disk could vary depending on its performance. In the Write stage of phase one & two, we choose disk adaptively for the write request.

Implementation

Basic Settings

Since Indy doesn't require fault tolerance, we use one replication for both input and output files. And we treat disks as JBOD.

We have one input file on each disk with approximate equal size. The size of output file is about 250MB each, so there are 28,000 output files in Minute Sort and 400,000 in Gray Sort.

The method we use to distribute data on each machine (Node Distributor) is equivalent to SimplePartitioner in Hadoop. Logical Disk Distributor use similar method and the size of logical file is the same as output file.

Our system doesn't handle system/disk failure and we have to remove a few machines due to such failure or other performance consideration.

Sorting Algorithm

Although the sorting performance is a neglectable part in the overall performance, a better sorting algorithm could be helpful in CPU intensive scenario. We use a hybrid sorting algorithm which sorts the first 4-byte of key using radix sort (LSD) in the first pass. Then we apply quick sort on the partially sorted results.

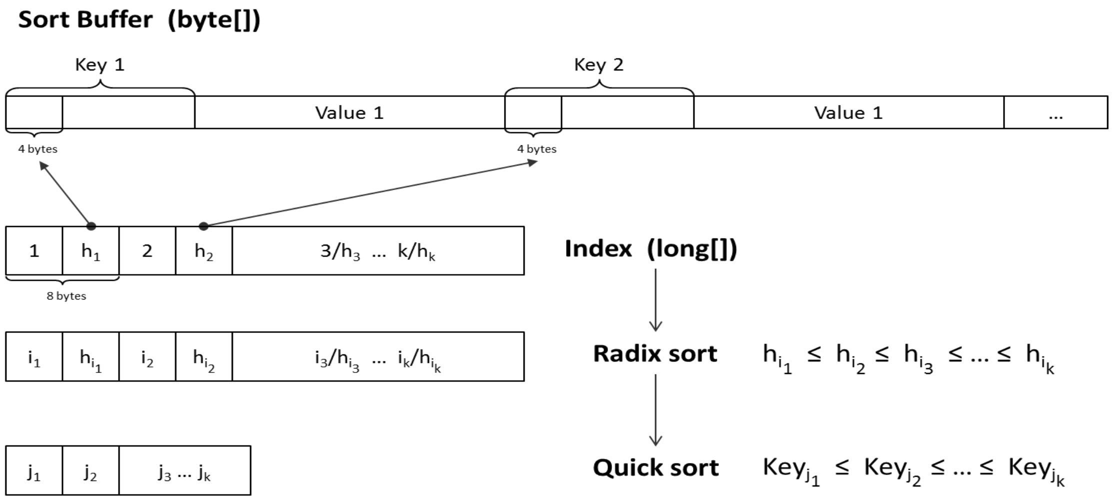

Figure 3: Hybrid Sorting Algorithm

We optimize the algorithm to reduce the memory footprint. Instead of sorting pointers, we directly sort the 4-byte key with index. The algorithm costs 360ms CPU time to sort 256MB data in our environment.

Although single thread can match the speed of pipeline, we use 4 threads in Gray Sort and 16 threads in Minute Sort to reduce 1 ~ 2 seconds in filling up the writer pipeline.

Network Transfer

We use ZeroMQ [3] as our all-to-all network transfer library. The transfer rate is more than 800MB/s on a 20-node cluster, and 360MB/s on the large cluster due to subscription.

It works well on the 20-node cluster, but it breaks down due to on large cluster. We observe nearly half the runs failed because of package loss. And the loss rate is around one in one million packages. We track the lost packages and find that they are continuous and belong to one or two socket pairs in a short period. Our best guess would be a temporarily disconnection of TCP which cause the problem. We don't have conclusive answer since it cannot be reproduced on a small cluster. We have to add an ACK mechanism to guarantee the correctness. For the time being, we cache all data on the Sender even though only little portion need to be resent.

Bottleneck

The bandwidth of Writer in phase one can surpass 400MB/s when we test on the small cluster without cache. Therefore, with cache applied, subscription makes network an obvious bottleneck in phase one. We also find the fastest sender completes in 22s in Minute Sort while the slowest needs 32s. The difference is beyond our expectation and requires further exploration.

In phase two, Writer is the bottleneck. The cost is not only the waiting time of disk IO but also the CPU time spent on random read in the memory. We reduce the total time by using two writer threads for each disk. And the straggler is under control because the maximum time difference of the fastest disk and the slowest disk on the same machine is the time of writing two files. The machine level difference is also steady and less than phase one.

Results

We use Linux drop cache and then warm up[1] the JVM before each run. The time is measured by the “time” command and includes all processes time on master and slaves. (1TB = 1e12 bytes)

	Indy Minute Sort	Indy Gray Sort
Data Volumn (TB)	7	100
Runs	15	5
Media time (s)	56.69	716.10
Min time (s)	51.28	710.29
Max time (s)	85.737	725.51
Checksum	826284e18bd1ea31c	746a51007040ea07ed
Duplicate keys	0	0

Acknowledgements

I would like to thank YangYi from Baidu Infrastructure Department for supervising me during my internship in Baidu. I won't be able to accomplish the experiments without his help in providing the platform and offering technical support. I’d also like to thank all the members in the Hadoop team of Baidu for their kind support.

Reference

[1] TritonSort, past winner, http://sortbenchmark.org/2011_06_tritonsort.pdf
[2] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In NSDI, 2011
[3] ZeroMQ, zeromq.org

[1] We start a JVM and terminate it to ensure all libraries required are from memory.