NADSort

Qian Wang[1], Rong Gu[1], Yihua Huang[1], Reynold Xin[2], Wei Wu[3], Jun Song[3], Junluan Xia[3]

[1]*Nanjing University*, [2]*Databricks Inc.*, [3]*Alibaba Group Inc.*

{qian.wang, gurong}@smail.nju.edu.cn, yhuang@nju.edu.cn,
rxin@databricks.com, {wei.wu, songjun.sj, junluan.xjl}
@alibaba-inc.com

Abstract

In this paper, we present NADSort, a sorting system on top of the distributed computing platform Apache Spark [1]. We report performance results of NADSort for Daytona CloudSort and Indy CloudSort benchmarks. NADSort is able to complete the 100TB Daytona CloudSort in 2983.33 seconds on random non-skewed datasets at an average cost of $144.22 and 3057.67 seconds on skewed datasets at an average cost of $147.82, and complete Indy CloudSort in 2983.33 seconds at an average cost of $144.22.

1 Overview

We implement a sorting system named NADSort running on the Alibaba Cloud Elastic Compute Service (ECS) [2] to complete the CloudSort 100TB benchmark. We implemented our system based on the 2014 Databricks sort benchmark entry based on Apache Spark. To improve the performance and cost efficiency, a push-based shuffle mechanism (as opposed to the original pull-based shuffle) was introduced. In this push-based shuffle setup, the system sends intermediate data to the reducer side directly, rather than storing those data on the mapper side.

2 Architecture

In this section, we provide an overview of NADSort. There are four stages in NADSort: the sampling stage, the map stage, the shuffle stage and the reduce stage.

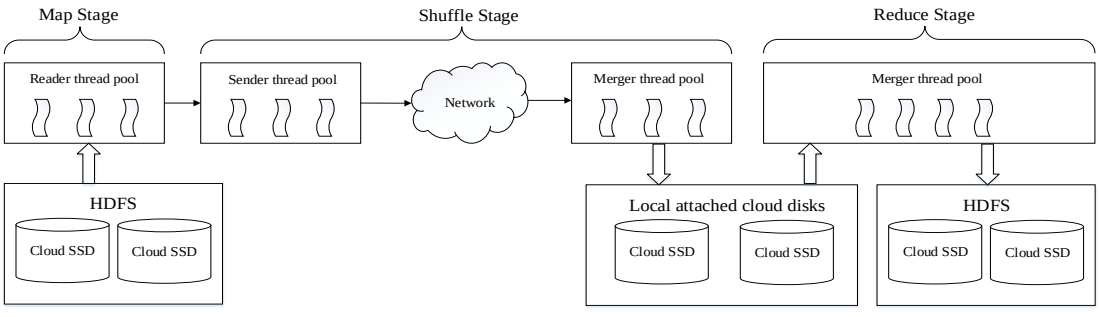

Figure1. Pipeline of NADSort

In the ***sampling*** stage, we sample records from the input data and generate the range boundaries for range partitioning using the sampled records. In the ***map*** stage, as shown in Figure 1, we maintain a Reader thread pool to read data from HDFS in file format and use Timsort [3] to sort data in each file locally. Then the ***shuffle*** stage partitions the sorted data into different parts (based on the range boundaries computed earlier) and sends the corresponding parts to the reducer side through the Sender thread pool. In the reducer side, there is a Merger thread pool that receives the data from different mappers, merges the data and writes the intermediate data to cloud disks mounted as local file system. In the ***reduce*** stage, there is a Merger thread pool that reads the data from local attached cloud disks, merges the data and then writes the final result to HDFS.

3 Implementation and Optimizations

In this section, we describe the implementation details of NADSort, including input data generation, sampling, map, push-based shuffle, and reduce.

3.1 Input Data Generation

We use gensort provided by the benchmark committee to generate input data in either random or skewed distribution. In order to balance the distribution of data on each node and to accelerate the generation of data, we adopt Apache Spark to parallelize the data generation, running gensort on each node. The input data is divided into partitions of the same size. The number of partitions matches the number of map tasks so that each map task processes one partition. We configure 200,000 partitions for 100TB dataset in both Daytona and Indy CloudSort. We use HDFS as a distributed file system to store the input and output data with the replica of one because Alibaba Cloud Block Storage can provide fault-tolerant. By using HDFS, we can get the location information of each partition, which allows us to achieve a better task locality.

3.2 Sampling Stage

In order to reduce data skew in the reduce stage, we sample the input data to determine partition boundaries. Given that the input data is divided into P partitions, we randomly sample X records on each partition and collect these $P*X$ samples to the driver. We sort these samples and calculate the range boundaries. There are $Q-1$ range boundaries for the range partitioner (that is, we use Q reduce tasks in our benchmark). For 100TB CloudSort benchmark, with 200,000 partitions, we select 39 random locations on each partition. Thus, we collect 7,800,000 samples to sort, and the driver node will generate 39,999 range boundaries. While loading data, we use libaio to read data asynchronously. The sampling phase takes about 20 seconds in total.

3.3 Map Stage

Since we set each partition to 500MB, we have 200,000 partitions to deal with in map stage. For CloudSort, we apply for 395 VMs (394 workers and 1 master) and there are three slots on each VM. Therefore, we run 170 waves of map tasks. In each wave of map tasks, for each VM, we have three concurrent tasks to read data and sort the data. After that, the Sender thread will compute the shuffle write location for each datum according to the range boundaries and send the data to the reducer side. The reducer side has a Merger thread pool that are responsible for receiving data sent from

different mappers, merging the received data and writing them to attached disks.

3.4 Shuffle Stage

In a normal map-reduce model of Apache Spark, it uses pull-based shuffle, which means storing the shuffled data in the mapper side, and then in the reduce stage, the reducers pull the shuffled data from remote mapper nodes through network.

According to the previous analysis, pull-based shuffle has three negative effects on overall sorting performance. First, in pull-based shuffle, reducers need to fetch a number of blocks generated by dividing partitions using range boundaries. In our configurations (e.g., 200,000 partitions in map phase and 40,000 partitions in reduce phase), these blocks are usually very small, no more than 15KB in the majority of cases. Sending a large number of small blocks will negatively affect the network performance. Second, the intermediate data cannot be deleted before all of the reduce tasks are finished. Therefore, we need totally 300TB disk to store the input data, intermediate data and output data. Third, the pull-based shuffle incurs many random disk reads, which will reduce the speed of reading data.

Given the reasons above, we proposed the push-based shuffle to boost the execution and save disk spaces. In the shuffle stage, we maintain a Sender thread pool aggregating the intermediate data sent to the same reducer before they are sent. Only when the amount of aggregated data reaches a certain threshold, will the data be sent to the reducer. In this way, we achieve a much higher network transmission efficiency. The reducer side maintain a Merger thread pool to receive the intermediate data. We also set a threshold for the received data, and start to merge the received data after the threshold has been reached. Then we write the intermediate data to the local attached cloud disks as one block of a partition. Therefore, we can run the workload faster and more efficiently by adopting the push-based shuffle optimization.

3.5 Reduce Stage

After the shuffle stage, each partition contains many ordered blocks of intermediate data (which are stored in the reducer side). Since each block is ordered, a reduce task can perform an external merge of the all blocks in its partition to produce the output, and write the results to HDFS. Thus, even if the size of a partition is larger than the memory, our program can still run well. That is why we can handle either non-skewed or skewed data with the same partition configuration.

After a reduce task sorts all the intermediate data in its partition and writes the results back to the disk, it can then safely remove the intermediate data it has just sorted. Therefore, we do not need to maintain 3 copies of the data (one input, one output and one intermediate) in the test, as it would have been in the case of pull-based shuffle. Instead, the test only needs to use a little bit more than 200TB of data storage; specifically, the extra hard disk space is roughly equal to 100TB divided by the number of reducer waves (so as to hold the working set for all the parallel reduce tasks). Because each VM has four slots in reduce stage, we run 26 waves of reduce tasks for 40,000 partitions in the reduce stage.

4 Environment

We run NADSort in Alibaba Cloud Elastic Compute Service (ECS), a commercially available public

cloud with efficient computing service and flexile processing capacity. ECS provides three different billing methods, (1) Pay As You Go (which we used in our tests); (2) Monthly; (3) Yearly. Users can rent virtual machine resources in different ways according to their specific needs. CPU, memory and bandwidth can be upgraded at any time with business suspension time in a controlled range. ECS provides a highly reliable environment with service availability of 99.95%. Users can also choose to increase or decrease the computing resources according to the real-time demand of their own applications. Through the experimental analysis of cost performance, we chose the ecs.n1.large (Series Ⅱ with IO Optimization in China Mainland) instance, which is available on both the international website [4] and the Chinese website [5] of Alibaba Cloud.

We use Block storage to provide high reliability, flexibility, high performance, low latency and data block level random storage for ECS. Block storage supports automatic replication of user data that can tolerate hardware failures. Block storage can be mounted to the ECS instance, and the user can format, create file systems, and persist data to block storage. We choose SSD Cloud Disk as the persistent storage for this test, because it has better throughput, IOPS and lower latency than General Cloud Disk and Ultra Cloud Disk. The input and output are resided on SSD Cloud Disks, the non-ephemeral storage with a reliability no less than 99.9999999%. We used 395 VMs (394 workers and 1 master) with the configuration described in Table 1 and Table 2. We did not mount any SSD Cloud Disks to the master. The prices of ECS and block storage in Table 1 are based on the international website of Alibaba Cloud [4].

Table 1. Hardware

Item	Master	worker
Resource	ecs.n1.large	ecs.n1.large
CPU	Haswell E5-2680 v3	Haswell E5-2680 v3
vCPU Cores	4	4
Memory	8GB	8GB
Cost	$0.271/hour	$0.271/hour
Network	iperf shows that the bandwidth between two random nodes in the cluster is about 1.2Gbps	
System Storage	40GB Ultra Cloud Disk	40GB Ultra Cloud Disk
Cost	$0.008/hour	$0.008/hour
Data Storage	--	4*135GB SSD Cloud Disk with software RAID 0 setup [6]
Cost	--	$0.0003 /GB/hour
Total Cost	$0.279/hour	$0.441/hour

Table 2. Software

OS	CentOS Linux release 7.2.1511 with kernel optimization
JDK	1.8.0._91 Hotspot
Apache Hadoop	Modified based on 2.7.2
Apache Spark	Modified based on 1.5.2
Netty	4.0.33

5 Benchmark Results

- Compression was turned off for all parts, including input, output, and network.
- File system buffer cache was dropped before each run and writes are flushed in each run.
- Runtime was measured using Linux time command.

We present our NADSort results for both Daytona and Indy CloudSort benchmarks in Table 3 (based on the prices on the international website [4] of Alibaba Cloud). We use an Apache Spark program to parallelize the data validation itself. Each Apache Spark task runs valsort on one partition of data and "valsort -s" is used to validate the checksum and global ordering. As shown in Table 3, NADSort is able to complete the 100TB Daytona CloudSort in 2983.33 seconds on random non-skewed datasets at an average cost of $144.22 and 3057.67 seconds on skewed datasets at an average cost of $147.82, and complete Indy CloudSort in 2983.33 seconds at an average cost of $144.22.

Table 3: 100TB CloudSort results

Item	Indy CloudSort	Daytona CloudSort	Daytona CloudSort (skewed)
VMs	395	395	395
Input Size	100TB	100TB	100TB
Records	100,000,000,000,000	100,000,000,000,000	100,000,000,000,000
Checksum	746a51007040ea07ed	746a51007040ea07ed	746a50ec9293190d87
Duplicate Keys	0	0	30285373571
Trial 1	2961s	2961s	3023s
Trial 2	3027s	3027s	3030s
Trial 3	2962s	2962s	3120s
Average Time	2983.33s	2983.33s	3057.67s
Average Cost($)	$144.22	$144.22	$147.82

In addition, one can also purchase Alibaba Cloud Elastic Compute Service (ECS) from its Chinese website [5], which is actually less expensive: the hourly cost of ECS instance (ecs.n1.large) and SSD Cloud Disk would be CNY 1.77 and CNY 0.0014 respectively. Based on those prices (and the BOC Exchange Rate in August 31, 2016 [7]), NADSort would be able to complete the 100TB Daytona CloudSort on random non-skewed datasets at an average cost of $124.86 (CNY 835.39) and on skewed datasets at an average cost of $127.97 (CNY 856.21), and complete Indy CloudSort at an average cost of $124.86 (CNY 835.39).

References

[1] Apache Spark. http://spark.apache.org/.
[2] Alibaba Cloud Elastic Compute Service. https://www.aliyun.com/product/ecs.
[3] Timsort. https://en.wikipedia.org/wiki/Timsort.
[4] Price of ECS and Block Storage on Alibaba Cloud international website. https://intl.aliyun.com/product/ecs#pricing.
[5] Price of ECS and Block Storage on Alibaba Cloud Chinese website. https://cn.aliyun.com/price/product#/ecs/detail.
[6] RAID. https://en.wikipedia.org/wiki/Mdadm#RAID_configurations.
[7] BOC Exchange Rate. http://www.boc.cn/sourcedb/whpj/enindex.html.