GraySort on Apache Spark by Databricks

Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, Matei Zaharia
Databricks Inc.

This document describes our entry into the Gray Sort 100TB benchmark using Apache Spark.

Apache Spark

Apache Spark [1] is a general cluster compute engine for scalable data processing. It was originally developed by researchers at UC Berkeley AMPLab [2]. The engine is fault-tolerant and is designed to run on commodity hardware. It generalizes two stage Map/Reduce to support arbitrary DAGs of tasks and fast data sharing between operations.

Spark can work with a wide variety of storage systems, including Amazon S3, Hadoop HDFS, and any POSIX-compliant file system.

Although often used for in-memory computation, Spark is capable of handling workloads whose sizes are greater than the aggregate memory in a cluster, as demonstrated by this submission. Almost all Spark built-in operators, including sort and aggregation, automatically spill to local disks when the working set does not fit in memory.

Sorting in Spark

Overview

Similar to MapReduce, our program uses Spark to sort data in a two stage distributed environment:

(1) In the map stage, each map task reads in its own partition of data, in the form of a file from HDFS, and sorts each the file locally, and then writes the sorted output to local disks (not replicated). As it is writing the sorted output out, it also builds an index pointing to locations of the output file based on a given range partitioner so the reduce tasks can retrieve ranges of data quickly. The map stage is entirely bounded by the IO in reading from HDFS and and writing out the locally sorted files. There is no network usage at all in the map stage.
(2) In the reduce stage, each reduce task fetches its own range of data, as determined by the partitioner, from all the map outputs from daemon processes on other nodes. It then sorts these outputs using TimSort and writes the sorted output out to HDFS with the same replication factor as the input, i.e. the output files are two-way replicated. The reduce stage is mostly network bound, because reduce tasks need to fetch data over the network and also replicate the sorted output over the network.

Sorting Within a Partition

Spark's shuffle monitors the memory used by running tasks. If the memory usage is greater than a configurable threshold (spark.shuffle.memoryFraction - a percentage of the total heap size), it starts spilling data to disk to perform external sorts.

TimSort [3] is used as the in-memory sorting algorithm. When data does not fit in memory, Spark uses TimSort for in-memory sorting for partial runs, and then the standard priority queue based merging of the sorted runs. TimSort performs better when data is partially sorted. In the reduce stage, we also perform a full sort rather than simply merging partially sorted runs. Although it might be more efficient to do only a merge in the reduce stage, our program was bounded by mainly network I/O and thus we did not optimize the CPU time of sorting.

To better exploit cache locality and reduce data movement during sorting, we store each record in memory as a 16-byte record, where 10 bytes are used for tracking the 10 byte key (taking endianness and signedness into account), and another 4 bytes are used for tracking the position of the original 100-byte record.

Even though Spark can handle external sorting within a partition, in our experiment we made sure no spilling could happen to avoid the extra I/O incurred from external sorting.

Range Partitioner and Sampling

In the case of Daytona Sort, we sample the input to generate the range partitioner. Let P be the number of partitions. The program launches a Spark job to sample evenly on each partition. On each partition, we pick S (default value 79) random locations and collect these samples back to the driver. That is to say, in total we collect S * P samples. The driver node then sorts these S * P samples, and downsamples to find (P - 1) range boundaries for the range partitioner. As an example, if we have 28000 partitions of data, and 79 random locations on each partition, we collect 2212000 samples and the driver node will generate 27999 range boundaries.

We measured the sampling time and it took ~ 10 secs in wall clock time.

One thing to note is that Spark out-of-the-box uses binary search on the range boundaries to find the partition ID a record belongs to, because Spark is overly general and does not assume the input to its range partitioner is sorted. This would result in a trillion of binary search operations.

For the purpose of this benchmark, since input to the range partitioner is already sorted, we created a new partitioner implementation for Spark that simply compares the current record key with the range boundaries in ascending order, starting from the boundary from the previous key. This is effectively performing a sort-merge join between the sorted records and the range bounds. This amortizes the partition ID computation to O(1) per record.

Input Data

The input data is divided into fixed size partitions and stored in HDFS with a replication factor of two to avoid data loss due to the event of single node failure(s). The input data was generated using gensort provided by the benchmark committee. We use Spark to parallelize the data generation, running gensort on each node. Because gensort only writes data to a POSIX file system, we copy the gensort output into HDFS.

Each partition of input is one HDFS file, occupying one HDFS block to ensure one replica of the partition is on a single node, and can be read entirely without triggering network I/O.

Output and Data Validation

In the reduce stage, each task writes one output file to HDFS, with a replication factor of 2.

We validate the data using valsort. Similar to the input data generation, we use a Spark program to parallelize the data validation itself. Each Spark task runs valsort on one partition of data (one output file copied from HDFS to local disks), and then the outputs are aggregated onto the driver node, and “valsort -s” is used to validate the checksum and global ordering.

Task Scheduling

Spark has a centralized scheduler that schedules tasks onto slots. On each machine, there are 32 slots. That is to say, for 28000 partitions on 206 worker nodes, it runs 5 waves of map tasks and 5 waves of reduce tasks.

Locality Scheduling

Spark’s scheduler uses a technique called Delay Scheduling [5] to schedule tasks based on the locality preference, i.e. it tries to schedule tasks onto nodes with local data. In this experiment, we increased the delay scheduling wait to a very large number so our sorting job gets 100% locality. Locality scheduling coupled with HDFS short-circuit local reads result in all tasks reading from local disks directly rather than going through the HDFS client. It also means the map stage does not have any network I/O.

Straggler Mitigation

Stragglers appear naturally on a large cluster. Stragglers are mitigated by the fact that we run more than 4 waves of tasks. Due to the two-way replication, slow nodes get scheduled less often for tasks. Spark also supports speculative task execution. For the purpose of this experiment, this feature was turned off.

System Configuration

We used 207 nodes (206 workers and 1 master) with the following configuration:

- i2.8xlarge Amazon EC2 instances
- Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz (32 virtual cores)
- 244 GB of RAM
- 8 x 800GB SSDs per node with NOOP scheduler and RAID 0 setup [4]
    - Once formatted with ext4, each node has 6TB capacity reported by “df -h”
    - In our testing, we can perform roughly 3GB/s mixed read/write in aggregate

- Network
    - All nodes put in a placement group in a VPC
    - Enhanced networking via single root I/O virtualization (SR-IOV)
    - In our testing of bandwidth between two random nodes in the cluster, iperf shows ~9.5Gbps
- Software configuration
    - Linux 3.10.53-56.140.amzn1.x86_64
    - OpenJDK 1.7 amzn-2.5.1.2.44.amzn1-x86_64 u65-b17
    - Apache Hadoop HDFS 2.4.1 with short-circuit local reads enabled
    - Apache Spark master branch (target for Spark 1.2.0 release)
    - Netty 4.0.23.Final with native Epoll transport

Results

- For the purpose of this benchmark, we use the same number of map partitions and reduce partitions, i.e. the number of input partitions equals the number of output partitions.
- 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.
- The system was up for more than 10 hours running various experiments.
- Runtime was measured using Linux time command. Each reported time represents one attempt.
- The map stage was able to saturate 3GB/s throughput per node, and the reduce stage sustained 1.1GB/s network.

Benchmark	**Skew**	**Data Size (Bytes)**	**P: Num Partitions**	**Time**	**Rate**
Daytona GraySort	non-skewed	100,000,000,011,000	29000	23m25.980s	4.27 TB/min
Daytona GraySort	skewed	100,000,000,008,000	28000	31m53.453s	3.14 TB/min

Acknowledgements

We would like to thank:

- The EC2 and EBS teams from Amazon Web Services
- Norman Maurer for reviewing our Netty-based network transport module code
- Aaron Davidson for his TimSort code implementation
- Min Zhou for various performance tuning & debugging help
- Andrew Wang for help identifying some performance issues with the OS kernel
- All our colleagues at Databricks and the Spark community in general for their support
- The Sort Benchmark committee members Chris Nyberg, Mehul Shah, and Naga Govindaraju for their help and support

References

[1] Apache Spark, http://spark.apache.org
[2] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012.
[3] TimSort. https://en.wikipedia.org/wiki/Timsort
[4] RAID Configuration.
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/raid-config.html
[5] M. Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. EuroSys 2010.