GraySort and MinuteSort at Yahoo on Hadoop 0.23

Thomas Graves
Yahoo!
May, 2013

The Apache Hadoop[1] software library is an open source framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to run on commodity hardware and detect and handle failures without relying on the hardware to deliver high availability. Hadoop provides a distributed file system (HDFS), a framework for application scheduling and cluster resource management (YARN), and a map/reduce[2] implementation for parallel processing of large data sets. A core aspect of map/reduce is it performs a distributed sort so we used Hadoop TeraSort to run the GraySort and MinuteSort benchmarks.

1. Hadoop Overview

Hadoop HDFS is a distributed file system that runs on commodity hardware, is highly fault-tolerant, provides high throughput access to application data, and is suitable for applications that have large data sets.

HDFS has a master/slave architecture. The master is the NameNode, which manages the file system namespace and regulates access to files by clients. The slaves are the DataNodes, generally one per node, which manage storage attached to the nodes that they run on. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

HDFS supports a traditional hierarchical file organization. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The current replication policy in HDFS uses hardware rack awareness. Generally the nodes in a cluster are spread across multiple racks, so a simple policy is to place replicas on unique racks. This prevents losing data if an entire rack fails and allows use of bandwidth from multiple racks when reading data. For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack.

Hadoop YARN is a resource manager and distributed application framework. It has a master/slave architecture where the ResourceManager is the master and the per-node slaves are the NodeManagers. YARN usually runs on the same cluster as Hadoop HDFS with NodeManagers running on the same nodes as the Datanodes.

The ResourceManager manages the cluster resources among the applications in the system. It has two main parts, a scheduler and an applications manager. The scheduler is responsible for partitioning the cluster resources among the various applications, queues, etc. The applications manager is responsible for accepting application submission, launching the first container for executing the application specific ApplicationMaster, and provides services for restarting the application on failures.

The NodeManager is responsible for reporting the node resources to the ResourceManager and managing the containers that are running on that node.
Each type of application that runs on YARN needs to have an ApplicationMaster.

The ApplicationMaster is responsible for negotiating resources with the ResourceManager, coordinating with the NodeManagers to start the allocated containers, tracking the container status, and monitoring the progress of the application.

Hadoop MapReduce is an application that runs on Hadoop YARN that is an implementation of map/reduce[2] for parallel processing of large data sets. Hadoop MapReduce usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically the input and output are stored in a file-system.

2. **TeraSort**

TeraSort is a standard map/reduce sort, except for a custom partitioner that uses a sorted list of *N-1* sampled keys that define the key range for each reduce. In particular, all keys such that *sample[i-1] <= key < sample[i]* are sent to reduce *i*. This guarantees that the output of reduce *i* are all less than the output of reduce *i+1*. This allows us to have multiple output files and the concatenation of files will be the same as one ordered output file.

The Hadoop TeraSort map/reduce program was used to run GraySort and MinuteSort benchmarks, using Hadoop HDFS to store the input and output.

The input data was generated with gensort version 1.5. For GraySort, the data was 102.5TB in size, spread across 1025 files each with 100,000,000,000 bytes. For the Indy MinuteSort, the data was 1612.22GB (1612223312700 Bytes) in size, spread across 1001 files each with 1610612700 bytes. For Daytona MinuteSort, the data was 1497.86 GB (1497869841679 Bytes) in size, spread across 920 files each with 1610612700 bytes. Both skewed and non-skewed data were generated as needed.

The data was uploaded into Hadoop HDFS with a replication factor of 3 to ensure it persists in the event of a single node failure. Hadoop HDFS splits the files up into blocks and spreads them randomly across the cluster nodes.

TeraSort was then run. It starts by sampling a subset of the input to determine the partitioning. The partitions are used to determine which range of keys go to which reducer. It does the sampling by reading a configurable number of sample records (defaults to 100,000) from a configurable number of locations (defaults to 10) in the file. The sampling is done from the client node where the TeraSort job is launched. The locations it reads the samples from are spread evenly across the input. In order to determine those locations, it first calculates the input splits of the file. The input splits are based on the block size or a user specified minimum split size. Once it determines the locations to read from, it then uses a thread per sample location to read the samples in parallel. Each thread reads the specified number of samples divided by the specified number of locations records. The sample keys are then sorted with QuickSort and down sampled to find even split points for the partitions. It then writes the partitions to a partition file in HDFS that will later be read by the maps.

For GraySort the sample size was 300,000 keys and the number of locations was 250, for the Daytona MinuteSort the sample size was 300,000 keys and the number of locations was 185, and for the Indy MinuteSort the sample size was 100,000 keys and the number of locations was 10.

For GraySort, it took 23670ms to compute the base input splits and 1789ms to read the samples and compute the partitions. For Daytona MinuteSort, it took 1285ms to compute the base input splits and 755ms to read the samples and compute the partitions.

The maps then run in parallel across the cluster. It uses the input splits to determine how many maps to run and the data each map will read. Hadoop MapReduce tries to schedule each map on the same node as the block(s) it is reading from HDFS. If it cannot get node locality, it then tries to get rack locality (a node within the same rack). If it cannot get rack locality then it chooses any node. In our case, the input data had a replication factor of 3 so it had 3 nodes to try to place each map to get node local data locality.

For GraySort, the job used 31775 maps, for the Indy MinuteSort the job used 6006 maps and for the Daytona MinuteSort the job used 5580 maps.

Each map reads its input data, sorts the data, and then partitions it per reducer. The TeraSort TotalOrderPartitioner was used for GraySort and the Daytona MinuteSort. It uses the partition file written to HDFS based on the sampling to build a two level trie that quickly indexes into the list of sample keys based on the first two bytes of the key. If more then one partition maps to a leaf in the trie, it iterates through those partitions and compares the entire key to that partition key to find the exact match. The TeraSort SimplePartitioner was used in the Indy MinuteSort benchmark. The SimplePartitioner is a total order partitioner that assigns keys based on their first 3 bytes. It assumes a flat distribution.

Once enough of the maps have completed (a user specified percent), the reducers launch and fetch the data from the maps. For GraySort the job used 10,000 reducers, for the Indy Sort the job used 2600 reducers, and for the Daytona MinuteSort the job used 2790 reducers. Each reducer generates an output file in HDFS. The output was sync'd to disk as required by the benchmark and for the Daytona benchmarks the output had a replication factor of 3 to meet the rule that the output must persist in the event of a single node failure. For the Indy MinuteSort benchmark the output had a replication factor of 1 to reduce any overhead in writing the extra replications.

The output was validated using both valsort version 1.5 and Hadoop TeraValidate. TeraValidate ensures that the output is globally sorted. It creates one map per a file in the output directory and each map ensures that each key is less than or equal to the previous one. The map also generates records with the first and last keys of the file and the reduce ensures that the first key of file *i* is greater that the last key of file *i-1*. Any problems are reported as output of the reduce with the keys that are out of order. valsort was run on each of the output files using the –o option, the summary files were then concatenated together and then valsort –s was run on the concatenated file. Both valsort and TeraValidate generated the same checksums. TeraChecksum was used on the input to verify the checksum of the input matched the output. TeraChecksum computes the 128 bit sum of the CRC32 of each key/value pair. Each map computes the sum of its input and emits a single 128 bit sum. There is a single reduce that adds the sums from each map.

3. Hardware and Operating System

- Approximately 2100 nodes for GraySort and 2200 nodes for MinuteSort
- System: Dell R720xd, 2 x Xeon E5-2630 2.30GHz, 62.3GB / 64GB 1333MHz DDR3, 12 x 3TB SATA
- Processors: 2 x Xeon E5-2630 2.30GHz, 7.2GT QPI (HT enabled, 12 cores, 24 threads) - Sandy Bridge-EP C2, 64-bit, 6-core, 32nm, L3: 15MB
- OS: RHEL Server 6.3, Linux 2.6.32-279.19.1.el6.YAHOO.20130104.x86_64 x86_64, 64-bit
- Network: eth0 (bnx2x): 10Gb/s <full-duplex>
- 40 nodes/rack 160Gbps rack to spine. 2.5:1 subscription.
- Oracle JDK 1.7 (u17) - 64 bit

4. Software and Configuration

The version of Hadoop used was Hadoop 0.23.7. Hadoop 0.23.7 is an early branch of the Hadoop 2.X line that Yahoo has used to stabilize YARN. It is available for download at hadoop.apache.org.

The configuration used in this benchmark is close to what Yahoo uses in production with the exception that Hadoop security was turned off. Compression was also turned off on all parts (input, intermediate, and output). We modified job parameters to change some of the time outs, sort and spill configurations, and map/reduce memory sizes. The only source code modification made for GraySort was to the ExampleDriver.java to allow TeraChecksum to be easily called. For MinuteSort, some minor Hadoop source code modifications were made to remove extra logging messages and change some hardcoded wait/retry periods. The TeraSort suite was repackaged to be the only thing in the jar.

Between the GraySort runs, Linux drop caches was used on all the hosts to flush anything out of memory. I also alternated between the non-skewed and skewed data which should have also flushed the other data set from main memory. For the MinuteSort runs, the benchmark was run 15 consecutive times alternating between TeraSort and TeraChecksum of a very large amount of data (154.6TB) in order to flush any data from the TeraSort out of memory. The median time of the 15 runs of TeraSort was reported.

Each run was measured on the client side using the Linux ‘time’ utility.

5. Results

According to the rules, the results are reported such that a terabyte = 10^{12} bytes, gigabyte = 10^{9} bytes. For the Daytona benchmarks, the skewed data had to be sorted in no more than twice the elapsed time of the non-skewed data.

The only benchmark that made the official benchmark entry this year was the GraySort. We missed the deadline on submitting the MinuteSort results.

Benchmark	**Data Type**	**Amount of Data Sorted**	**Time**	**Rate**	**Duplicate Keys**
Daytona GraySort	non-skewed	102.5 TB	72 minutes 8.053 seconds	1.42 TB per minute	0
Daytona GraySort	skewed	102.5 TB	117 minutes 48.261 seconds	0.87 TB per minute	31867643140
Indy MinuteSort	non-skewed	1612.22 GB	58.027 seconds		0
Daytona MinuteSort	non-skewed	1497.86 GB	59.223 seconds		0
Daytona MinuteSort	skewed	1497.86 GB	1 minute 27.242 seconds		0

6. Acknowledgements

I'd like to thank Balaji Narayanan and Rajiv Chittajallu from the Yahoo Grid Operations team for setting up the hardware and providing support while these benchmarks were being run. I'd also like to thank Nathan Roberts, Bobby Evans, Kihwal Lee, and Jason Lowe from the Yahoo Hadoop development team for providing input.

7. References

7.1. Apache. Hadoop. http://hadoop.apache.org/.

7.2. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on largeclusters. In Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.

More info at http://sortbenchmark.org/ and http://sortbenchmark.org/FAQ-2013.html