Combining Mapper Outputs in Hadoop

Amardeep Singh Chawla[1], Michael Schliep[2], Sumit Raj[3]

Dept. of Computer Science
University of Minnesota
{chawl007[1],schli116[2],rajxx019[3]}@umn.edu

Abstract

Hadoop is a popular open source implementation of MapReduce programming paradigm. It is heavily used in software industry for batch processing of large amounts of data. It utilizes data-parallelism to compute job tasks in a parallel fashion enabling it to be fast, scalable and fault tolerant. However, the MapReduce framework makes extensive use of network bandwidth to shuffle intermediate data from mappers to reducers. This is more pronounced if the input data of the job is huge. We propose merging and combining of mapper outputs on the same node before the shuffling phase of the job begins. Combining data across mappers on the same node de-duplicates intermediate keys at each node level resulting in decrease in amount of data shuffled across the network. The experimental results validate that network bandwidth usage is decreased as a result of merging without significantly affecting job completion times.

I Introduction

In the present time, a number of organizations have now focused their business around the collection and analysis of substantial amount of data. Giant companies such as Facebook and Yahoo have taken up Hadoop [1, 3] for reliable, scalable, distributed computing and data storage.Apache Hadoop is an open source MapReduce Java framework for running jobs that computes large amounts of data in-parallel on large clusters of commodity hardware in a reliable and fault-tolerant way. It has its genesis as implementation of ideas of MapReduce and Google File System.

MapReduce [2] is a programming model that provides a scalable parallel data computation framework for large clusters. Hadoop, as an open-source implementation of MapReduce, allows programmers to process vast amount of data in the distributed fashion without having to face the complicated issues involved with distributed systems. It implements the MapReduce programming model in which the users provide the map and reduce functions; the underlying processing is handled by the hadoop framework.The java libraries of the framework detect and handle faults at the application layer. Hence deliver a highly available service to the client applications running on the cluster.

With portable digital devices and evolution made in the applications such as mobile web, have profoundly led to an explosion in the amount of data that needs to be processed. The sheer amount of generated data is stored for data analytics that have become increasingly important to businesses and several government organizations. Hadoop and other similar technologies like Dryad

[4] and Apache Tez [5] aim to cater such demands at large scale of computing. This has led us to the urgent need to make Hadoop more and more scalable.

As the input data becomes larger, the inherent model of MapReduce leads to the fact that large amount of data has to be shuffled from mappers to reducers. However, network bandwidth is limited resulting in shuffling phase of the job becoming the bottleneck for large input size. Hence, it is important to reduce the amount of data to be shuffled in order to make Hadoop more scalable. The Hadoop community has provided much functionality like compression and combiner options which hadoop users can optionally use to reduce the amount of shuffling data. The combiner removes duplicates from individual mapper outputs; thus reducing the size of shuffling data. We extend the combiner functionality to remove duplicates across all mappers scheduled on a node.

The remainder of the report is organized as follows. We provide review about related work in section II. In section III, we discuss about the motivation and describe the existing issues in Hadoop. We then present the solution and design with limitations in Section IV and implementation of our work in Section V. The experimental analysis is provided in Section VI. We conclude the paper in Section VII. Finally we mention about future work in Section VIII.

II Related Work

Hadoop performance is limited by the network bandwidth available. Hence it is quite beneficial to reduce the amount of data to be shuffled. During shuffling phase, the hadoop framework fetches the appropriate partition of the single output files of all the mappers, via HTTP. Hadoop provides compression options to the users to achieve reduction in shuffling data. If enabled, intermediate data is compressed before shuffling resulting in lesser network bandwidth usage. However, this technique adds the computational overload of compression at the sender node and decompression at the receiver node which affects job completion times.

Another functionality that hadoop provides to reduce network shuffling is that of *combiners*. Each map task reads input split files and produces key-value pairs in memory buffer. When the buffer reaches maximum size, its contents are spilled to disk. However, there are lots of duplicate intermediate keys in the spilled contents. Combiners remove duplicate keys before the spill, thus reducing the amount of write to disk. Since it reduces spill file size, it also reduces the amount of data to be shuffled across the network. The MapReduce framework runs combiners intelligently in order to bring down the amount of data that has to be written on to the disk and transfer it over the network in between the Mappers and Reducer.

Combiners provide a good way of reducing network traffic in hadoop cluster. The combiner does local aggregation of the intermediate mappers' outputs, that helps to bring down the amount of network traffic transferred from the Mapper to the Reducer. However, the latest stable release of hadoop does not leverage the fact that combiners can be used to remove duplicates across all mappers scheduled on a particular node. The combiner that hadoop provides removes duplicates per mapper but there still will be duplicates across mappers on a node which can be de-duplicated to further reduce the amount of shuffled data from each node.

III Problem statement and motivation

Hadoop's framework comprises of *JobTracker, TaskTracker, MapTask,* and *ReduceTask* as its four main component as shown in Figure 1. For executing the job, JobTracker assigns one TaskTracker per data-node and orchestrates TaskTracker to launch mapper and reducer tasks. Mapper and Reducer tasks are implemented in the MapReduce programming model for data processing. A map task reads an input split from Hadoop Distributed File System, executes the map function, and stores intermediate data as a single output map output file on to the local file system of the node executing the map task. A data shuffling phase is necessitated to transfer the intermediate data generated by the mapper to the reducers as their input. However, such data shuffling can cause a great volume of network traffic, imposing a serious impact over the efficiency of data analytics applications.

Hadoop intermediate data shuffling causes a large volume of network traffic. Every ReduceTask fetches data segments from all mappers, resulting in a network traffic pattern from all MapTasks to all Reduce Tasks, which grows in the order of O(N^2) assuming that MapTasks and Reduce Tasks are both a factor of N total tasks. In the recent findings from the study conducted by Yahoo!, the intermediate data shuffling from 5% of large jobs can consume more than 98% network bandwidth in a production cluster, and worse yet; Hadoop's performance degrades non-linearly with the increase of intermediate data size.

In our work, by merging and combining the mappers output at a node, we have reduced amount of data transfer during shuffle phase.

IV Solution and design

In the current state of an art, combiner function is applied on a single mapper. For every spill file of the mapper(each map task can have multiple spill files at the end of the task), first segments are merged then the combiner function is applied onto the merged segment. Then combiner output is written onto the single output file of the Mapper. The combining happens if number of spill files is more than the default value (in this case 3). As shown in Figure 1, once a single output file is written on the mapper, TaskTracker notifies JobTracker about the completion of MapTask through HeartBeat message. Reducer polls JobTracker for MapTasks that have completed. After getting their completion status from JobTracker, reducers read their input from the output file of the mappers on a data-node.

In our work, Combiner function applied across all mappers on the same node. As shown in Figure 2, Reducer reads its input from the output files of a single mapper(which has combined data of all mappers scheduled on that node) on a node.

IV.I Limitations of our design

In our implementation, the TaskTracker waits for the merge phase to finish before sending the status of map tasks as SUCCEEDED to JobTracker. However, if there is a straggler map task on a node, it can delay the shuffling of data of all map outputs from that node. Merging in phases would be a better

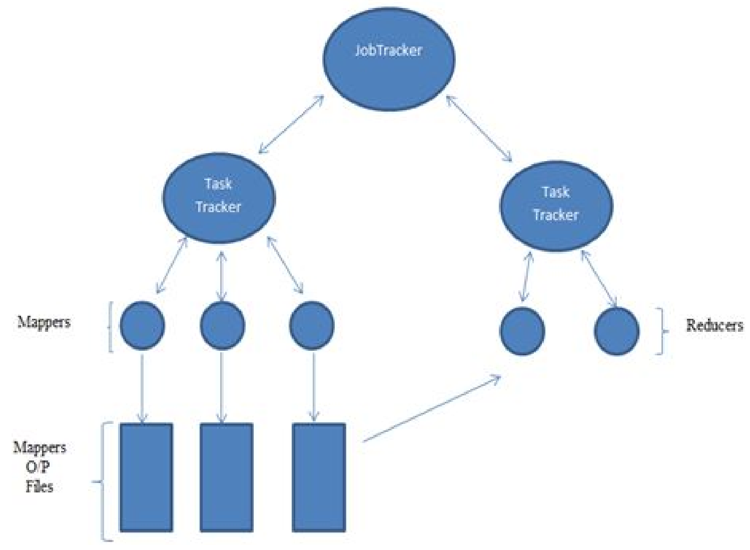

Figure 1: Hadoop's Component.

approach since it will allow partial shuffling of data in presence of a straggler. Partial shuffling is beneficial because the reducers can then proceed with reduce function on a larger subset of mapper outputs.

However, merge in phases would add significant implementation complexity. Also, it should be noted that in absence of stragglers, merging in phases can actually increase network bandwidth usage.

V Implementation

With an aim to combine data across mappers so as to remove duplicates among all mapper outputs, we have opted for a simple design. The TaskTracker keeps report of all map tasks running on a node. Once all map tasks have succeeded, the TaskTracker initiates the merge (along with combiner) of all map outputs. The merge phase removes duplicates across all mappers on a node.

The TaskTracker has to report the status of all tasks to the JobTracker through the heartbeat mechanism. Merge phase starts only when all map tasks at a node have succeeded. However, if TaskTracker sends the status of map tasks as succeeded during the merge phase, reducers will pull intermediate data before the merge is finished. This fails our aim of reducing shuffling data. Therefore, to prevent reducers from reading data before the completion of merge phase, TaskTracker lies to JobTracker by sending the reports of map tasks as incomplete until the merge is complete. Once the merge is complete, the actual status of all map tasks (succeeded) is sent to JobTracker, thus allowing reducers to start shuffling.

In our implementation, until the merge completes, the status of map tasks send to JobTracker is COMMIT_PENDING. This prevents the reducers from initiating shuffling until merge completion. Once merge is successful, the status of all map tasks is sent as SUCCEEDED, thus enabling shuffling to occur.

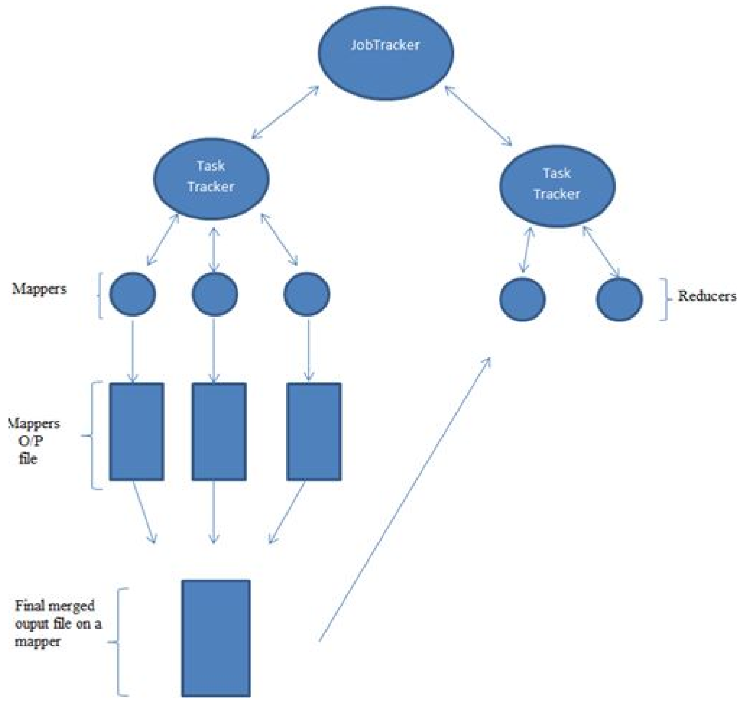

Figure 2: Proposed System Model.

Where do we store the merged output? The thread that TaskTracker starts for the merge reads the output data of all mappers, combines them, removes duplicates and spills the result to the output location of one of the mappers scheduled at that node. It should be noted that the original map outputs have to be deleted to maintain correctness of the system. However, it is also necessary to have an output file for each map task to prevent reducer exception. Therefore, we place dummy files (which do not have any data) at the mapper output locations. Only one of the mapper output locations has the merged data.

It should be noted that the merge operation is done partition-wise. That is, for a given partition P $\in$ [0 , R-1] where R is the number of reducer tasks, records present in all the mappers output on a single node, are merged and written onto the partition P of the final output file. For example, partition zero across all map outputs is merged to form partition zero of the merged output. Final output file is created on the output path of a mapper present at *index 0* of the mapTasks list at TaskTracker. *mapTasks* list contain objects of mapTasks that are *TaskInProgress*.
In our implementation, the partition-wise merging is done by a single thread in a loop. However, partition-wise merging in different threads parallelly will be better in terms of time required for the merging phase.

VI Evaluation

VI.I Environment

We evaluated our implementation with Hadoop 1.2.1 on four virtual machines with a single Intel(R) Xe-on(R) CPU E3-1220 V2 @ 3.10GHz CPU and 1 GB of memory. One limitation of our evaluation is that all the virtual machines reside on the same physical host. Due to this we are not able to observer the effects of our implementation on the overall completion time. Even with this limitation we still show the data sent over the network is less than vanilla and we can infer the job completion time will reduce based on the latency of the network.

To work around the limitation on speculative execution and our approach we have disabled it in your Hadoop cluster.

VI.II Experiment

We ran a simple word count example with a combiner function. Word count is acceptable because most production use cases for the combiner function are to combine counts in log files or references in documents. Other uses of the combiner function are usually with small output from the mappers and the results would be similar.

The input for our word count evaluation is comprised of Wikipedia[6] featured articles and Project Gutenberg[7] top books in the past 30 days. The Wikipedia articles are more representative of small to medium inputs(28 to 319) kilobytes and the Project Gutenberg books of medium to larger inputs(138 to 3244) kilobytes.

VI.III Result

Here we see the real impact of our system on shuffled data. If Figure 3. we show the ratio of records shuffled compared to the ideal case. The ideal case being the minimum shuffle of only the output records. You see our solution less than the median number of records of vanilla Hadoop. This appears to be consistent for any size of output records.

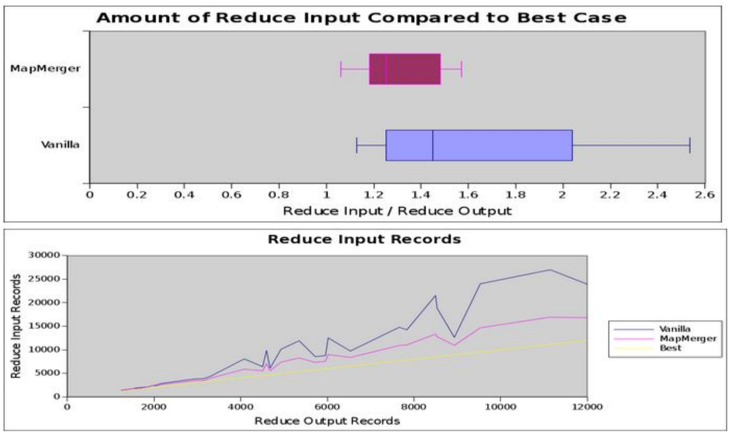

Figure 3: Reduce input records compared to reduce output records

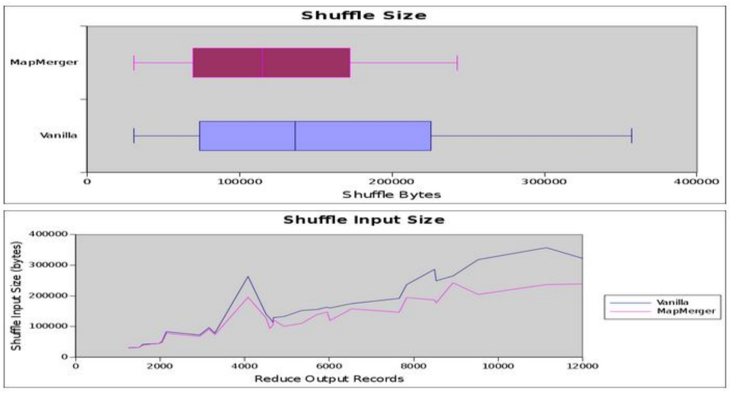

Figure 4: Amount of network traffic during shuffle phase.

To represent this in more concrete fashion we provided the same results in the form of amount of traffic sent over the network during the shuffle phase in Figure 4. We see similar results for the median network traffic. The lower chart makes it apparent that even though the gap between vanilla and our system fluctuates it increases as the output size grows.

It is not enough for our system to only reduce the network traffic. We also must show we are comparable in performance to vanilla Hadoop. Figure 5 shows we are perfectly acceptable in performance, requiring only a few hundred milliseconds longer to complete. The anomaly near inputs size of 2000 records is explained in Figure 6. A side effect of our solution is it takes longer for the Map task to complete but is shorter during the Reduce task. This input displays our current limitation of waiting on stragglers that will be address with selective merging. At input size of 2000 records our solution spends more time during the Map phase and the vanilla spends that time during the Reduce phase. This shows one node received significantly more occurrences or a word. With our previously mention limitation our solution is not able to carry on and that single straggler due to skewed data delays the node. You can see this is the issue because the CPU time is the same and the delay must be in the waiting on a single Mapper before starting to transfer the output of the others on the same node to the Reducers.

VII Conclusion

Hadoop MapReduce is a open source software framework meant for scalable, distributed and data-intensive computation. The Hadoop framework is fault tolerant, reliable, and stores petabytes of data. Framework incurs huge network traffic during shuffle phase. In our work, we reduce amount of data getting transferred from mappers to reducers by merging outputs of mappers scheduled on the same node. Thereby applications running in environments with constrained network bandwidth, will comparatively perform better than Vanilla hadoop.

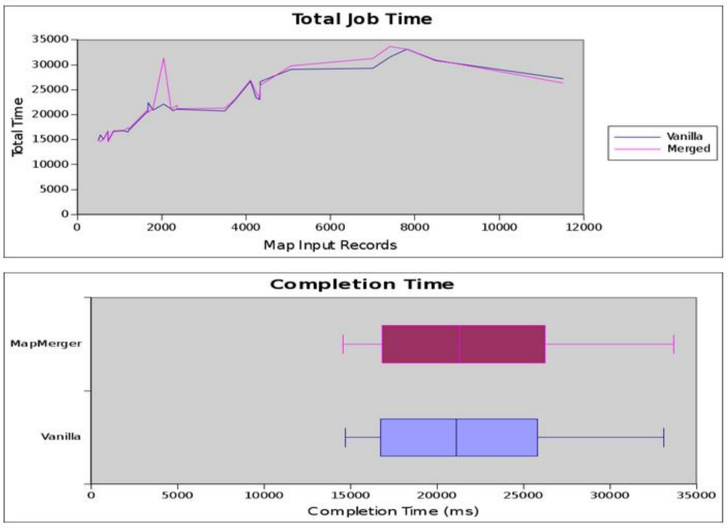

Figure 5: Overall Performance.

VIII Future Work

The future of this work should focus on selectively merging the output of Mappers. There may not be a need to merge the output of all the Mappers on a single node. The current approaches has a couple of defined issues. Specifically straggler mitigation does not work as intended in MapReduce. This is observered by the single Mapper slowing down the shuffle for all Mappers on the affected node. With an approach similar to SkewTune[8] we should be able to alleviate the straggler problem of our design. What SkewTune offers is a method to measure and predict the issue we saw with the skewed input. Using this information we would be able to shuffle all Mappers except for the straggler and improve our tail performance. Selective Merging would allow us to also look at the benefits of merging and combining mappers' output across nodes in the same rack. This could help reduce inter-rack network traffic which is normally not as performant as intra-rack networks. There are other works dealing with solutions[9, 10] to cluster network performance that may make this a non-issue. Apache Tez is the next step for Hadoop. It is an implantation of a system to execute Directed Acyclic Graph(DAG) of tasks. This is the future of MapReduce in Hadoop 2. Moving this work to a DAG model of MapReduce should be simple and doing so would help the open source community continue their excellence in Big Data computing.

References

[1] Apache Hadoop Project. *http://hadoop.apache.org/*

[2] J. Dean and S. Ghemawat *Mapreduce: simplified data processing on large clusters.* In Proceedings of the 6th conference on Symposium on Opeart-

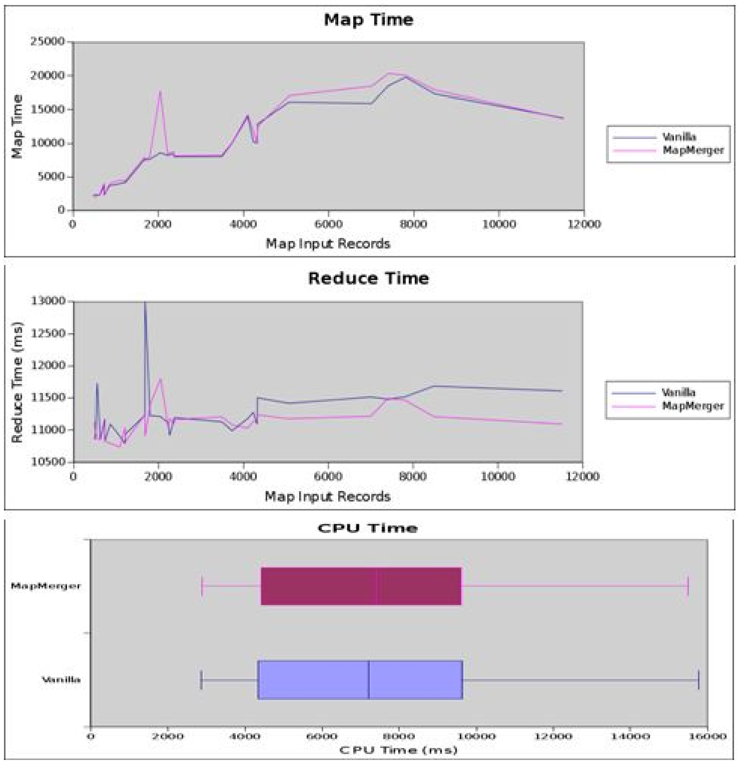

Figure 6: Comparison of Map vs Reduce time.

ing Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[3] Yahoo's Hadoop Blog. *http://developer.yahoo.com/hadoop/*

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks.* European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007.

[5] Apache Tez. *http://hortonworks.com/hadoop/tez/*

[6] Wikipedia: Featured Articles. *http://en.wikipedia.org/wiki/Wikipedia:Featured_articles/*

[7] Project gutenberg. *http://www.gutenberg.org/browse/scores/top#books-last30/*

[8] YongChul Kwon, Magdalena Balazinska, Bill Howe, Jerome Rolia. *SkewTune: Mitigating Skew in MapReduce Applications.* In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). ACM, New York, NY, USA, pages 25-36.

[9] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica. *Managing data transfers in computer clusters with orchestra.* SIGCOMM Comput. Commun. Rev. 41, pages 98-109, August 2011.

[10] Mosharaf Chowdhury, Srikanth Kandula, Ion Stoica. *Leveraging Endpoint Flexibility in Data-Intensive Clusters.* SIGCOMM Comput. Commun. Rev. 43, pages 231-242, August 2013.