International Journal on Applications in Engineering and Technology
Volume 4: Issue 2: May 2018, pp 76 – 78 www.aetsjournal.com ISSN (Online) : 2455 - 0523

VIRTUAL SHUFFLING FOR EFFICIENT DATA MOVEMENT IN MAP REDUCE

DEEPIGA , M.VIJAYANANTHKUMAR , R.VETRIVENTHAN, SENTHIL KUMARAN

Abstract— Google's Map Reduce programing model serves for processing large data sets in a massively parallel manner. The model is stunningly simple, and it effectively supports parallelism. The programmer may abstract from the issues of distributed and parallel programming because it is the Map Reduce implementation that takes care of load balancing, network performance, fault tolerance, etc. To keep up with the increasing volume of datasets, it requires efficient I/O capability from the underlying computer systems to process and analyze data in two phases (mapping and reducing). Between these phases, Map Reduce requires a shuffling phase to globally exchange the intermediate data generated by the mapping phase. In this paper we do a brief survey of schemes developed and algorithms designed to enable efficient data movement between Map and Reduce phase. In addition we propose a novel approach for enabling efficient data movement and reducing data transfer by incorporating virtualization in existing Map Reduce framework. Hadoop is an open source implementation of Map Reduce.

Keywords— Map Output File , Apache Hadoop.

I. INTRODUCTION

Google's Map Reduce [1] programming model has become de facto model for large scale big data analytics. The programming model is based on the following, simple concepts: (i) iteration over the input; (ii) computation of key/value pairs from each piece of input; (iii) grouping of all intermediate values by key; (iv) iteration over the resulting groups; (v) reduction of each group . The model is stunningly simple, and it effectively supports parallelism. The efficiency of MapReduce's performance and scalability can directly affect our society's ability to mine knowledge out of raw data.

Apache Hadoop is an open source implementation of Map Reduce. It implements the MapReduce model by distributing user inputs as data splits across a large number of compute nodes. Hadoop uses a master program (called the Job Tracker) to command many Task Trackers (a.k.a slaves) and schedule map tasks (Map Tasks) and reduce tasks (Reduce Tasks) to the Task Trackers, A Hadoop program processes data through two main functions (map and reduce). Accordingly, the analytic functions are performed in two phases: mapping and reducing. In the mapping phase, the input dataset of a program is divided into many data splits. Each split is organized as many records of key and value (<k,v >) pairs. One Map Task is launched per data split to convert the original records into intermediate data in the form of many ordered < k,v > pairs. These ordered records are stored as a MOF (Map Output File) split. A MOF is organized into many data partitions, one per Reduce Task. In the reducing phase, each Reduce Task applies the reduce function to its own share of data partitions (a.k.a segments).

Between the mapping and reducing phases, a Reduce Task needs to fetch a segment of the intermediate output from all finished Map Tasks. Globally, this leads to a shuffling of intermediate data (in segments) from Map Tasks to Reduce Tasks. For data-intensive MapReduce programs such as Tera Sort, data shuffling can add a significant number of disks accesses, contending for the limited I/O bandwidth. The shuff ling of intermediate data competes for disk bandwidth with Map Tasks. This can significantly overload the disk subsystem. It results in a serious bottleneck along with the severe disk I/O contention in data-intensive MapReduce programs, which entails further research on efficient data shuffling techniques.

In our work, we undertake another effort to investigate the issue of disk I/O contention in data shuffling. As shown in **[Figure-1]** we take new perspective at data shuffling of MapReduce programs. In the default Hadoop implementation, intermediate data segments are pulled by Reduce Tasks in their entirety to local disks, and then merged before being reduced for final results.

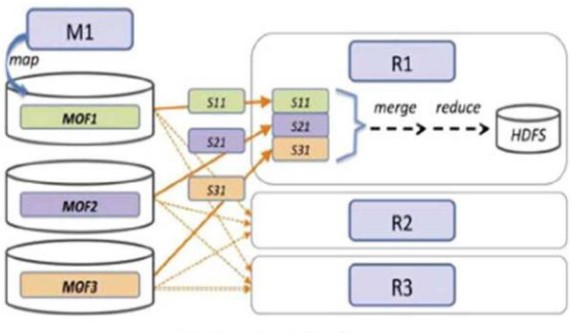

Figure 1 :

Deepiga , Master of Computer Applications , Meenaakshi Ramasamy Engineering College , Thathanur , Ariyalur Dt , Tamil Nadu .

M.Vijayananthkumar , Assistant professor/MCA , Meenaakshi Ramasamy Engineering College , Thathanur , Ariyalur Dt , Tamil Nadu .

Mr.R.Vetriventhan , BE , MTech , MISTE , Head of the department , Department of MCA , Meenaakshi Ramasamy Engineering College , Thathanur , Ariyalur Dt , Tamil Nadu .

Mr.Senthil Kumaran , ME Phd , Managing Director , Meenaakshi Ramasamy Engineering College , Thathanur , Ariyalur Dt , Tamil Nadu.

Because the physical movement of entire segments across disks, we refer to this strategy as physical shuffling. We propose a virtual shuffling strategy to enable efficient shuffling for MapReduce programs. **[Figure-2]** shows the general idea. Instead of moving entire segments to local disks before starting the reduce function, virtual shuffling allows a Reduce Task to fetch only a minimal set of segment attributes and create a virtual segment table that records the actual locations of remote segments. Once the global segment table is fully constructed, the Reduce Task can start the final merge and generate data for the reduce function. When the reduce function starts, the majority of segments still reside on remote disks. In another word, segments will not be brought in locally through the global segment table until they are needed by the reduce function. Virtual shuffling delays the actual movement of data until the Reduce Task requests data.

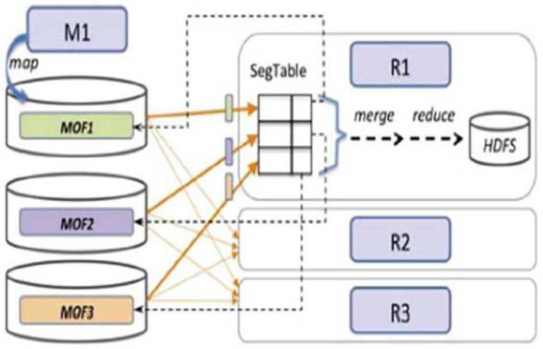

Figure: 2. Virtual shuffling

Compared to physical shuffling, when the reduce function demands more data input, virtual shuffling employs near demand merging to fetch data in small blocks into memory, merge and send them directly to the reduce function. In doing so, virtual shuffling greatly reduces the number of disk accesses of physical shuffling, and enables efficient data movement. Such optimizations in I/O and program execution also benefit MapReduce programs in terms of power consumption and energy savings.

II. PREVIOUS WORK:

Condie et al. has proposed a MapReduce online architecture to open up direct network channels between Map Tasks and Reduce Tasks and speed up the delivery of data from Map Tasks to Reduce Tasks. While their work reduces job completion time and improves system utilization, it cannot accommodate a gigantic dataset that does not fit in memory, and also complicates the fault tolerance handling of Hadoop tasks. It often falls back to spilling data to the disks for large data sets

Kim et al. [4] improved the performance of MapReduce by reducing redundant I/O in the software architecture. But it did not study the I/O issue caused by data shuffling between Map Tasks and Reduce Tasks.

III. PROPOSED SYSTEM

In order to overcome the disk I/O problem of physical shuffling, virtual shuffling technique is used. For that it needs to address three important issues: (1) how to scalably represent intermediate data segments in a virtual manner, (2) how to minimize the impact of actual shuffling of data; and (3) how to dynamically coordinate and balance data shuffling and merging without degrading the performance.

1) Three-Level Segment Table

To manage many intermediate data segments produced by Map Tasks, we design a three-level segment table to organize them in a scalable manner. The hierarchical table includes three kinds of directories: the Segment Table Directory, the Segment Middle Directory, and the Segment Global Directory **[Figure-3]** shows the organization and relationship among these three levels.

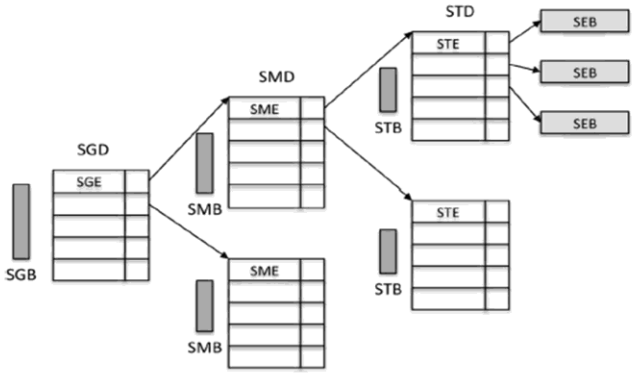

Figure: 3. Three level segment table

At the completion of a Map Task, its data segment is not physically fetched all at once by a Reduce Task. Instead, a Segment Table Entry (STE) is created at the lowest level-Segment Table Directory (STD) to represent the segment in a virtual manner. The STE includes several attributes of the segment such as its total length, its source Map Task, as well as its physical location on the remote disk. Many STDs are organized into a Segment Middle Directory (SMD), in which each entry (SME) represents an STD. Many SMDs in turn are organized as a Segment Global Directory (SGD) with each entry (SGE) referring to an SMD. Within a Reduce Task, we allocate Segment Entry Buffers (SEBs) for each STE. SEBs are used to store partial blocks of data for incoming segments. In addition, three kinds of memory buffers are used as temporary merging buffers. For example, an SGD will merge the data from its constituent SMDs to the SGD Buffer (SGB); an SMD will merge data from its STDs to its SMD Buffer (SMB); and an STD will merge data from its constituent SEBs to its STD Buffer (STB). With this three-level hierarchical table, if there were memory pressure, we can keep only a few active STDs and their ancestral SMDs and SGDs in memory, while other STDs are temporarily stored on disk.

2) Near-Demand Merging

With near demand merging, we wait until it is clear which segments are needed by the reduce function of Reduce Tasks. Near-demand merging does not really wait until the last moment to fetch data. Instead, it works as part of virtual shuffling to form a pipeline of fetching, merging and reducing data segments, with an emphasis on hiding the cost of data transfer over the network.

[Figure-4] shows the operation of near demand merging. When a Reduce Task needs to reduce some data, it initiates a data request to the segment table, which in turn triggers the fetching of data blocks (which contain more intermediate<k,v > pairs from Map Tasks) from remote segments.

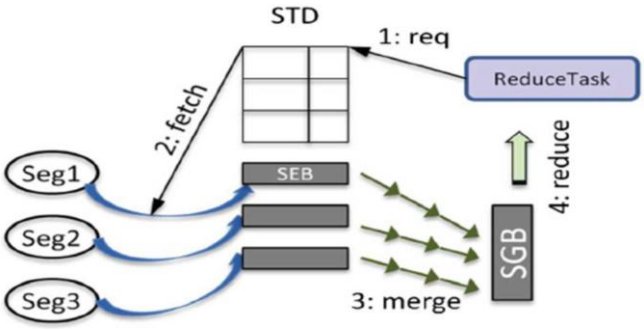

Figure: 4. Near demand merging

These blocks will then be buffered at SEBs. Based on the virtual segment table, these < k,v> pairs in SEBs will then be merged through STBs, SMBs, and finally into SGBs. The ordered <k,v > pairs in SGBs are ready to be reduced by the reduce function.

3) Dynamic and Balanced Subtrees

With a hierarchical segment table, all virtual segments are essentially organized into a merging tree in which the leaves are the SEBs. Instead of activating all leaves, we organize the whole tree as many subtrees, each composed of an STD and its SEBs. At any time, only a limited number of subtrees are actively merging data. As shown in **[Figure-5]** two subtrees are currently active in merging its SEBs into STBs. The merged data will be further merged to SMB s and/or SGBs. To balance the merging progressat different subtrees, previously active subtrees will be deactivated to allow other subtrees to make progress.

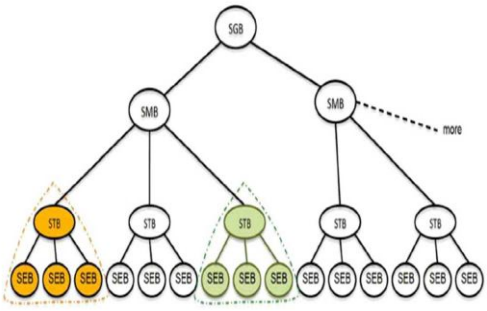

Figure: 5.Dynamic and balanced subtrees

IV. CONCLUSION:

Virtual shuffling significantly relieves the disk I/O contention problem and speeds up data movement in Map Reduce programs. Thus execution time is reduced. In addition, it reduces power consumption of Map Reduce programs.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in Proc. 6thSymp. Oper. Syst. Design mplementation (OSDI), Dec. 2004, pp. 137–150.

[2] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, MapReduce online,"

[3] S.-G. Kim, H. Han, H. Jung, H. Eom, and H. Y. Yeom, "Harnessing input redundancy in a MapReduce framework," in Proc.ACMSymp. Appl. Comput. (SAC'10), 2010, pp. 362–366. [Online]. Available: http://doi.acm.org/10.1145/ 1774088.1774167

[4] X. Yang, Z. Yu, M. Li, and X. Li, "Mammoth: Autonomic data processing framework for scientific state transition applications," in Proc. ACM Cloud Autonomic Comput. Conf. (CAC'13), 2013, pp. 13:1