Elysium Journal
Engineering Research & Management

P-ISSN: 2347-4408
E-ISSN: 2347-4734

HADOOP MAP REDUCE ENHANCEMENT THROUGH IMPROVED AUGMENTED ALGORITHM

Mr.Ankit Kumar Kothari[1], Mrs. J. Vijayalakshmi[2]

[1]*Post Graduate Student, Department of Computer Applications, Sri Sairam Engineering College, Chennai, Tamilnadu, India.*

[2]*Associate Professor, Department of Computer Applications, Sri Sairam Engineering College, Chennai, Tamilnadu, India*

Abstract: ***The MapReduce model is implemented by using open-source software of Hadoop. A number of issues faced by Hadoop to achieve the best performance. A serialization barrier requires to achieve the best performance which delays the phase. Repetitive merges and disk access leads to leverage latest high speed interconnects. With increasing volume of datasets, an acceleration framework of Hadoop-A optimizes Hadoop to keep updates. To overcome the problem of repetition and disk access, a novel algorithm to merge data introduced in this paper. A full-pipeline is also proposed for overlapping of the shuffle, merge and reduce the phases. The disk access from the intermediate data efficiently reduced and the data movement also increased by proposed Hadoop-A.***

Key Terms: ***Hadoop, MapReduce, Hadoop-A algorithm, Pipeline algorithm, cloud computing***

1. INTRODUCTION

Organizations processes large amount data requires massive computation for extracting critical knowledge by MAPREDUCE technique. MAPREDUCE is an easy programming model for cloud computing [1]. Hadoop [2], currently maintained by the Apache Foundation and supported by leading IT companies such as Facebook and Yahoo!, is an open source software framework implementation of MapReduce.

The implementation of MapReduce framework is implemented by Hadoop in two types of components: They are Job Tracker and Task Trackers. The Job Tracker monitors the task trackers by providing commands to it. The commands received by Task Trackers and process the data in parallel through two main functions. They are map and reduce functions. The scheduling process of reduce tasks to Task Trackers is done by Job Tracker. Between the two phases [3], a Reduce Task needs to fetch a part of the intermediate output from all finished Map Tasks. This leads to the shuffling of intermediate data in segments from all Map Tasks to all Reduce Tasks. For many data-intensive MapReduce programs, data shuffling can lead to a significant number of disk operations, contending for the limited I/O bandwidth. This presents a severe problem of disk I/O contention in MapReduce programs.

Several algorithms are used out to improve the performance of Hadoop MapReduce framework. Condie et al. [4] opened up direct network channels between Map Tasks and Reduce tasks by using MapReduce Online architecture and the delivery of data is improved. It remains as a critical issue to examine the relationship of Hadoop MapReduce‟s three data processing phases, i.e., shuffle, merge, and reduce and their implication to the efficiency of Hadoop.

To ensure the correctness of MapReduce, no Reduce Tasks can start reducing data until all intermediate data have been merged together. This results in a serialization barrier that significantly delays the reduce operation of Reduce Tasks. More importantly, the current merge algorithm in Hadoop merges intermediate data segments from MapTasks when the number of available segments including those that are already merged goes over a threshold. These segments are spilled to local disk storage [5] when their total size is bigger than the available memory. The algorithm causes data segments to be merged repetitively and, therefore, multiple rounds of disk accesses of the same data.

To address these critical issues for Hadoop MapReduce framework, Hadoop-A has been designed for performance enhancement and protocol optimizations. Several enhancements are introduced:

1) A novel algorithm that enables Reduce Tasks to perform data merging without repetitive merges and extra disk accesses;

To combine the shuffle, merge, and reduce phases for Reduce Tasks, full pipeline is designed. The evaluation demonstrates that the algorithm is able to remove the serialization barrier and effectively overlap data merge and reduce operations for Hadoop Reduce Tasks. Overall, Hadoop-A is able to double the throughput of Hadoop data processing.

2. RELATED WORK

MapReduce is a programming model for large-scale arbitrary data processing. Ranger et al [6] utilized the advantage of evolution of multi core and multiprocessor systems to design Phoenix for shared-memory systems. In Phoenix, users writes simple parallel code for dynamic scheduling and partitioning of data without considering the complexity of thread creation.

Kaashoek et al [7] then used the composite structure for a new MapReduce library, which outperforms its simpler peers, including Phoenix.

Tiled-MapReduce designed by Chen et al [8] further improves the Phoenix by leveraging the tiling strategy that is

P-ISSN: 2347-4408
E-ISSN: 2347-4734

commonly used in complier community. It divides a large MapReduce job into multiple discrete subjobs and extends the Reduce phase to process partial map output.

Hadoop‟s MapReduce implementation enables a convenient and easy-to-use data processing framework. The characterization and analysis reveal a number of issues, including two functions. They were serialization and disk access. 1) The serialization Hadoop shuffle/merge and reduce phases are serialized in serialization, 2) repetitive merges and disk access. In this section, it provides an overview of the Hadoop MapReduce framework [9].

By meticulously reusing memory and threads, Tiled-MapReduce achieves considerable speed up over Phoenix. But our work is completely different from these works in three aspects. First, it aims to improve the Hadoop MapReduce that is designed for largescale clusters instead of for single machine with multicores. Second, our optimization strategy is to reduce the contention over disk I/O instead of cache and shared data structures.

3. OVERVIEW OF HADOOP MAPREDUCE FRAMEWORK

A key feature of the Hadoop MapReduce framework is its pipelined data processing. As shown in Fig.1 Hadoop contains three execution phases. They are map, shuffle/merge, and reduce. First, the Job Tracker receives user job and divides the input set into data splits. User data is organized as many records of <key,val> pairs in each split. The running and scheduling of map function includes the number of Task Trackers selected by Job Tracker. Each Task Tracker provides several Map Tasks each split. The conversion form original records into intermediate results is carried out by using mapping function. The intermediate results are termed as data records in the form of <key‟‟,val‟‟> pairs. The received data records are stored in MOF (Map Output File), one for each split of data. A MOF is organized into many data partitions, one per Reduce Task. Each data partition contains a set of data records. When a Map Task completes one data split, it is rescheduled to process the next split. Second, the Job Trackers selects a set of Task Trackers using available MOFs to run the Reduce Tasks. Task Trackers spawns several concurrent Reduce Tasks.

Each Reduce Task initiates the process by fetching a partition intended for Reduce Task from a MOF termed as segment. A segment in each MOF used for every Reduce Task hence, Reduce Task needs fetches such segments from all MOFs. Fetch operations lead to an all-to-all shuffle of data segments among all the Reduce Tasks.

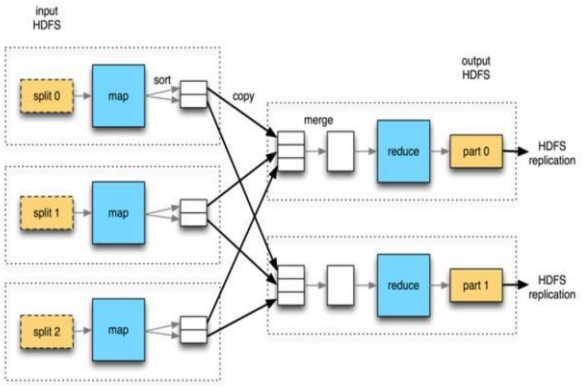

Fig. 1 Hadoop MapReduce Framework

The data segments are shuffled and merged based on the order of keys in the data records. The fetching and merging of remote segments spill the Reduce Task locally. The spill processes such as some segments are stored in local disk leads to alleviate memory pressure. The copy phase in hadoop contains two processes. They are shuffling and merging of data segments into Reduce Tasks.

Finally, the merged segments are loaded and processed by each Reduce Task loads using the reduce function. The results from the reduce function are then stored in Hadoop Distributed File System [10] .

Problem Identification:

- MapReduce results in Serialization barrier
- Delays the reduce phase
- Repetitive merge
- Multiple rounds of disk accesses of the same data

3.1 PROCESSING

Hadoop reach to pipeline the data processing. It is indeed able to do so, particularly for map and shuffle/merge phases. After a brief initialization period, a pool of concurrent MapTasks starts the map function on the first set of data splits. As soon as Map Output Files (MOFs) are generated from these splits, a pool of Reduce Tasks starts to fetch partitions from these MOFs.

P-ISSN: 2347-4408
E-ISSN: 2347-4734

contains an implicit serialization. At each Reduce Task, only until all its segments are available and merged, will the reduce phase start to process data segments [11] via the reduce function. This essentially enforce a serialization between the shuffle/merge phase and the reduce phase. When there are many segments to process it takes a significant amount of time for a Reduce Task to shuffle and merge them. As a result, the reduce phase will be significantly d e l a y e d . The analysis has revealed that this can increase the total execution.

3.2 REPETITIVE MERGES A N D DISK ACCESS

Hadoop ReduceTasks merge data segments when the number of segments or their total size goes over a threshold. A newly merged segment has to be spilled to local disks due to memory pressure. However, the current merge algorithm in Hadoop often leads to repetitive merges, thus extra disk access. It uses a very small threshold parameter. A ReduceTask fetches its data segments and arranges them in the order of their size. When the number of data segments reaches threshold the smallest three segments are merged. Under memory pressure, this will incur disk access. The resulting segment is inserted back into the heap based on its relative size.

When more segments arrive the threshold is reached again. It is then necessary to merge another set of segments. This again causes additional disk access, let alone the need to read segments back if they have been stored on local disks. As even more segments arrive, a previously merged segment will be grouped into another set and merged again. Altogether, this means repetitive merges and disk access, causing degraded performance for Hadoop.

4. PROPOSED FRAMEWORK

To address both the issues in Hadoop as mentioned, the algorithm has been described that avoids repeated merges and then details the construction of a new pipeline to eliminate the serialization barrier.

4.1 IMPROVED AUGMENTED ALGORITHM

Hadoop appeal to repetitive merges because of limited memory compared to the size of data. For each remotely completed MOF, each ReduceTask invokes request to query the partition length, pull the entire data, and store locally in memory or on disk. This incurs many memory loads/stores and/or disk I/O operations.

The algorithm has been designed that can merge all data partitions exactly once and, at the same time, stay levitated above local disks. The key idea is to leave data on remote disks until it is time to merge the intended data records.

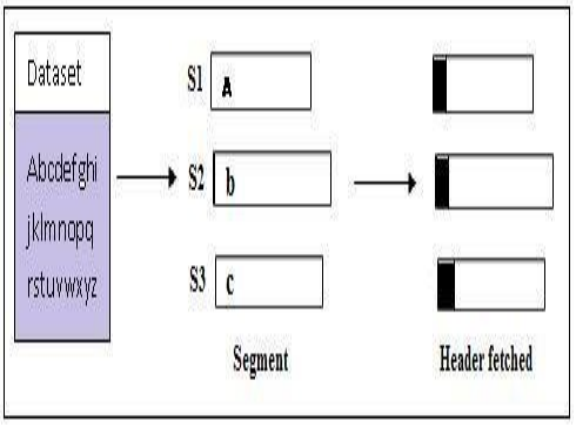

Fig. 2 Header Fetching

Therefore, an alternative merge algorithm is critical for H a d o o p t o m i t i g a t e t h e i m p a c t o f r e p e t i t i v e merges and extra disk accesses as shown in Fig 2. Three remote segments S1, S2, and S3 are to be fetched and merged.

Instead of fetching them to local disks, new algorithm only fetches a small header from each segment. Each header is especially constructed to contain partition length, offset, and the first pair of <key,val>.

These <key,val> pairs are sufficient to construct a priority queue (PQ) to organize these segments. More records after the first <key,val> pair can be fetched as allowed by the available memory. Because it fetches only a small amount of data per segment, this algorithm does not have to store or merge segments onto local disks.

Elysium Journal
Engineering Research & Management

P-ISSN: 2347-4408
E-ISSN: 2347-4734

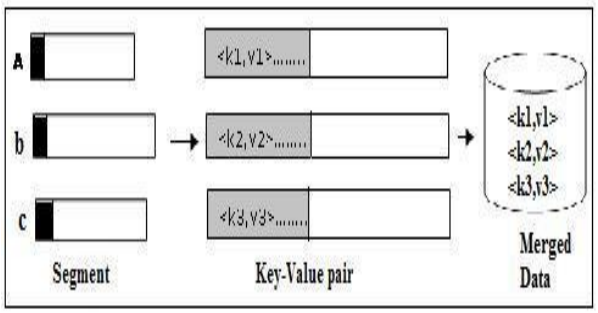

Fig.3 P r i o r i t y Queue Setup

Relocated accordingly. Concurrent data fetching and merging continues until all records are

2. Instead of merging segments when the number of segments is over a threshold, it keeps building up the PQ until all headers arrive and are integrated.

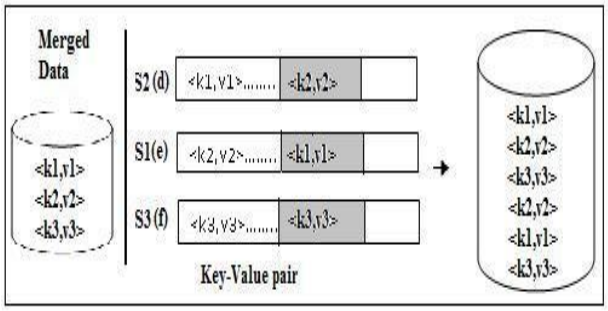

Fig.4 C o n c u r r e n t Fetching and Merging

As soon as the PQ has been set up, the merge phase starts. The leading <key,val> pair will be the beginning point of merge operations for individual segments as shown in Fig. 3.

3. The algorithm merges the available <key,val> pairs in the same way as is done in Hadoop. When the PQ is completely established, the root of the PQ is the first <key,val> pair among all segments.

It extracts the root pair as the first <key,val> in the final merged data. Then, It updates the order of PQ based on the first <key,val> pairs of all segments. The next root will be the first <key,val> among all remaining segments. It will be extracted again and stored to the final merged data. When the available data records in a segment are depleted, the algorithm can fetch the next set of records to resume the merge operation. In fact, the algorithm always ensures that the fetching of upcoming records happens concurrently with the merging of available records

As shown in Fig. 4, the headers of all three segments are safely merged; more data records are fetched, and the merge points are

4. Fig. 5 shows a possible state of the three segments when their merge complete. Since the merge data have the final order for all records, it can safely deliver the available merged. All <key,val> records are merged exactly once and stored as part of the merged results.

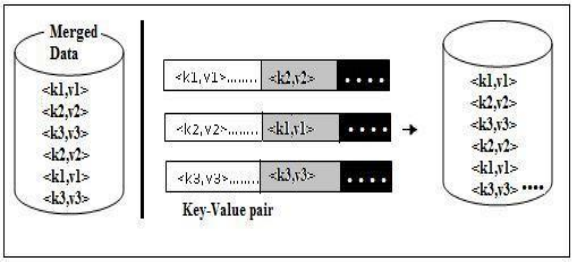

Fig.5 To w a r d s Completion

data to the Reduce Task where it is then consumed by the reduce function.

4.2 PIPELINED SHUFFLE, MERGE AND REDUCE ALGORITHM

Besides avoiding repetitive merges, the algorithm removes the serialization barrier between merge and reduce. As described earlier, the merged data have <key,val> pairs ordered in their final order and can be delivered to the Reduce Task as soon as they are available. Thus, the reduce phase no longer has to wait until the end of the merge phase.

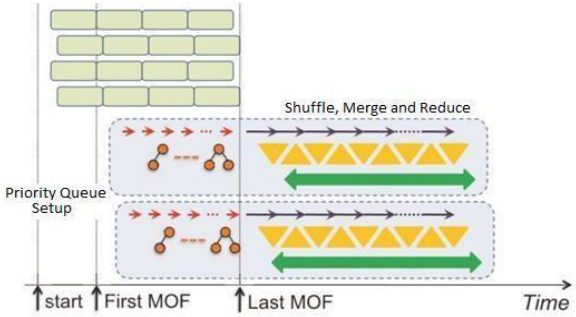

Fig. 6 Pipelined Algorithm

In view of the possibility to closely couple the shuffle, merge, and reduce phases, they can form a full pipeline as shown in Fig.6.

Steps:

1. In this pipeline, MapTasks map data split as soon as they can. When the first MOF is available, Reduce Tasks fetch the headers and build up the PQ.

2. These activities are pipelined. Header fetching and PQ setup are pipelined and overlapped with the map function, but they are very lightweight, compared to shuffle and merge operations.

3. As soon as the last MOF is available, completed PQs are constructed. The full pipeline of shuffle, merge and reduce then starts.

Advantages:

P-ISSN: 2347-4408
E-ISSN: 2347-4734

- Hadoop-A enables Reduce Task to perform data merging without repetitive merges
- It is concern to no extra disk accesses
- Overlapping shuffle, merge and reduce phases for ReduceTask
- The data movement and throughput is improved by using Hadoop-A. One may notice that there is still a serialization between the availability of the last MOF and the beginning of this pipeline. This is inevitable in order for Hadoop to conform to the correctness of the MapReduce programming model. Simply stated, before all <key,val> pairs are available, it is erroneous to send any <key,val> pair to the reduce function (for final results) because its relative order with future <key,val> pairs is yet to be decided. Therefore, our pipeline is able to shuffle, merge, and reduce data records as soon as all MOFs are available. This eliminates the previous serialization barrier in Hadoop and allows intermediate results to be reduced as soon as possible for final results.

System Architecture:

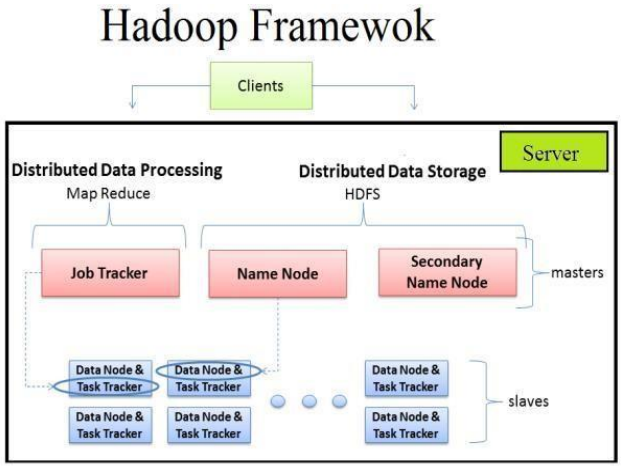

Fig.7 H a d o o p -A Framework same as such Hadoop

5. DISCUSSION AND RESULTS

Hadoop TeraSort and WordCount programs with different data sizes and numbers of slave nodes are run. It chooses the data size per split as 256MB. Each slave has 8 MapTasks and 4 ReduceTasks. The performance shows comparison between Hadoop-A and Hadoop for TeraSort and WordCount programs.

The performance evaluation of Map and Reduce Tasks is estimated as percentage of completion for with respect to the progress of time during execution. Hadoop -A increases the total execution time of TeraSort program by 47% compared to Hadoop Framework. The small size of intermediate data and data movement in Hadoop-A leads to less benefit inWordCount. Using Hadoop –A, MapTasks of TeraSort completed much faster, when the percentage of completion goes over 50%. Hadoop- A performed light weight operations in MapTasks. They are fetching headers and setting up PQ. Hence, resources such as disk bandwidth for MapTasks are left. After the data merging is over, Hadoop reports the progress of ReduceTasks. The reporting process also implemented in Hadoop-A. The progress of ReduceTasks is slow since Hadoop-A waits until the completion of last MOF. The percentage jumps quickly for TeraSort and WordCount, respectively followed by reporting process in Hadoop- A.

6. EXPERIMENTAL RESULTS

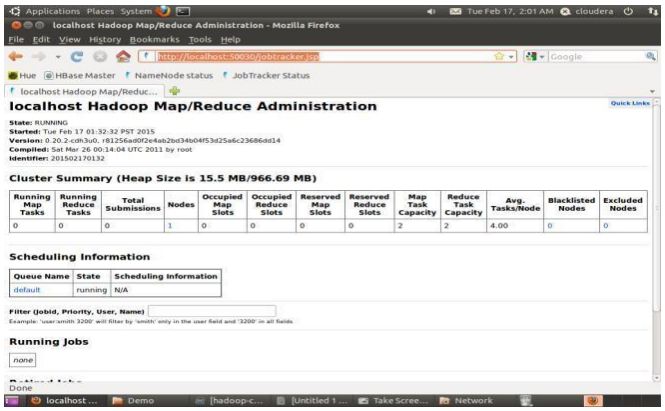

Fig.8 H a d o o p MapReduce Administration

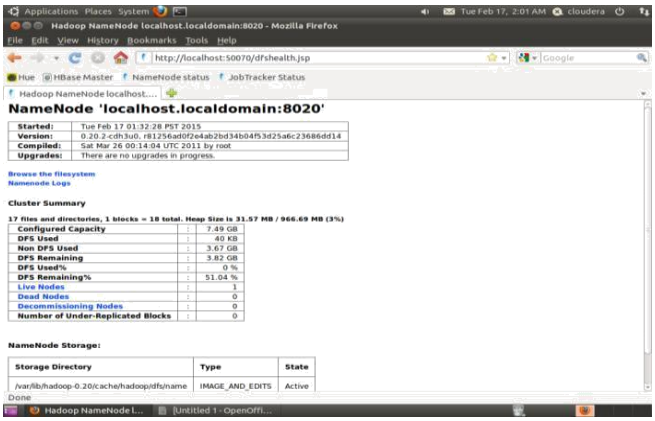

Fig.9 N a m e N o d e Information

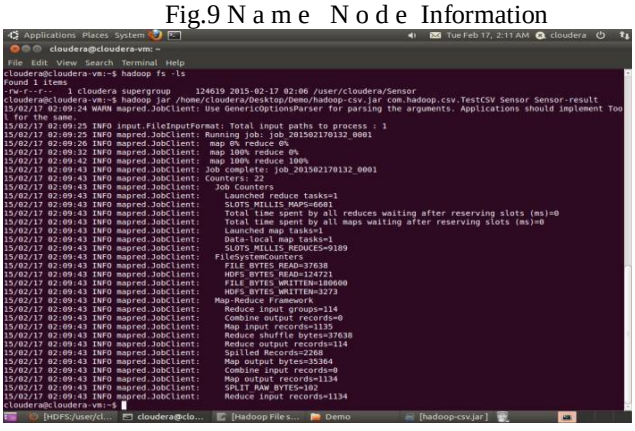

Fig.10 Hadoop Data Processing

P-ISSN: 2347-4408
E-ISSN: 2347-4734

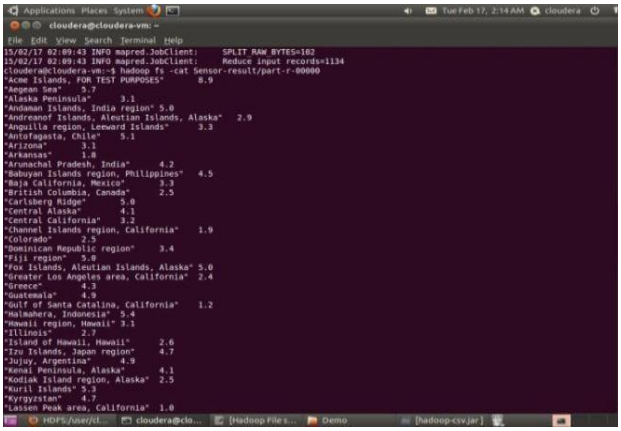

Fig.11 Hadoop Data Generation Framework

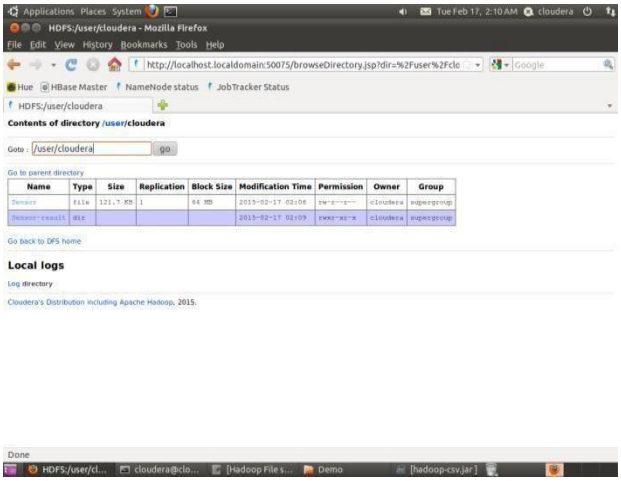

Fig.12 Data Generated after MapReduce Phase

7. CONCLUSION

The design and architecture of Hadoop‟s MapReduce framework in great detail has been examined. Particularly, the analysis has been focused on data processing inside Reduce Tasks. It reveals that there are several critical issues faced by the existing Hadoop implementation, including its merge algorithm, its pipeline of shuffle, merge, and reduce phases. Hadoop-A has been designed and implemented as an extensible acceleration framework, and evaluated algorithm that can merge data without touching disks and designing a full pipeline of shuffle, merge, and reduce phases for Reduce Tasks, It has been successfully accomplished an accelerated Hadoop framework, Hadoop-A. Because of the use of evaluated algorithm, it can significantly reduce disk accesses during Hadoop‟s shuffling and merging phases, thereby speeding up data movement.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107-113, 2008.

[2] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[3] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: an in-depth study," *Proceedings of the VLDB Endowment*, vol. 3, pp. 472-483, 2010.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *NSDI*, 2010, p. 20.

[5] A. MateiZaharia, A. Joseph, and I. RandyKatz, "Improving mapreduce performance in heterogeneous environments," 2010.

[6] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, pp. 13-24.

[7] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for multicore architectures," in *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[8] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 523-534.

[9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of network and computer applications*, vol. 23, pp. 187-200, 2000.

[10] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: locality-aware resource allocation for MapReduce in a cloud," in *Proceedings of 2011 International Conference or High Performance Computing, Networking, Storage and Analysis*, 2011, p. 58.

[11] X. Zhang, C. Liu, S. Nepal, S. Pandey, and J. Chen, "A privacy leakage upper bound constraint-based approach for cost-effective privacy preserving of intermediate data sets in cloud," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, pp. 1192-1202, 2013.