The Memory Challenge in Reduce Phase of MapReduce Applications

Seyed Morteza Nabavinejad, Maziar Goudarzi, *Senior Member, IEEE,* and Shirin Mozaffari

***Abstract*—MapReduce has become a popular paradigm for Big Data processing. Each MapReduce Application has two phases: Map and Reduce. Each phase consist of several tasks in a defaulted sequence of processes. It is common place to determine the number of Map tasks equal to the number of data blocks in the input data. However, there is no specific rule for determining the number of Reduce tasks based on the amount of intermediate data generated by Map tasks or the specifications of machines that execute the tasks. Since the Reduce tasks bring the data into memory for processing, this may lead to inefficient execution of application and even application failure because of memory shortage or temporary consumption. In this work, we first evaluate this challenge and show its problematic significance. To address this challenge, we propose a Mnemonic approach. Mnemonic leverages a profiling mechanism to detect the application behavior regarding intermediate data generation. It first decides the amount of memory to be dedicated to each Reduce slot. Then it determines the number of Reduce tasks based on the gathered information through profiling and the decided size of memory for Reduce slots. Experimental results using PUMA benchmark suit indicates that our proposed memory-aware approach can 1) completely remove the likelihood of application failure due to out of memory error and 2) decrease the execution time of Reduce phase up to 58.27%, 79.36%, and 88.79% compared with Memory Oblivious, Fine Grain 1, and Fine Grain 2 approaches, respectively.**

***Index Terms*—MapReduce, Hadoop, Big Data Processing, Memory-awareness**

I. Introduction

MAPREDUCE [1] and its open source implementation, Hadoop [2], has emerged as manageable, scalable, and fault tolerant framework for processing big data. MapReduce has two phases: Map and Reduce. In the Map phase, the input data, in the form of data blocks, is processed by Map tasks to generate intermediate data. Each Map task processes one data block. After that, the intermediate data is handled by Reduce tasks of Reduce phase to deliver the final results. Reduce tasks bring the intermediate data chunks to memory to process it. Despite the Map phase, where the number of tasks is determined by the number of data blocks, in Reduce phase the number of tasks can be determined by user or cluster administrator. The ability to manage intermediate data, as well as determination of amount of Reduce tasks, significantly affects the performance. A large body of research has tried to improve the storage performance [3], [4] or use compression techniques in order to reduce the volume of intermediate data [5], [6]. The intermediate data movement time during shuffle phase is addressed in [7]–[9]. However, neither of them has addressed the memory limitation in Reduce phase, which can lead to job abortion, or proper number of Reduce tasks.

As described above, all the previous works has concentrated on storage or network regarding intermediate data or solely focused on ratio between slots and disregarded slots internal configuration. While saturation of storage IO operations or network bandwidth can lead to performance degradation of Reduce phase, the good news is that the MapReduce application would not be killed in such cases by the Hadoop framework and continues its execution although slowly. However, in case of out of memory error, the job is killed since the Reduce phase needs to bring large portions of intermediate data into memory for processing. If there is not enough space left in memory, the Reduce tasks and consequently the Reduce phase will fail which leads to job termination by Hadoop. This is a major difference between shortage of memory vs. other resources such as disk/network IO or CPU in MapReduce applications and makes it a significant challenge to conquer. Albeit similar to other resources if the memory becomes the bottleneck, one will face performance degradation even if the job is not killed. Moreover, according to study on a real world Hadoop cluster [10], one-third of all misconfiguration problems in cluster are related to memory mismanagement. This further emphasizes the importance of proper memory tuning to avoid problems and improve performance.

It is noteworthy that out of memory is not the only reason for failures of MapReduce jobs; there are also other factors such as disk failure, out of disk, and socket timeouts that might also lead to failure [4], [11]–[14]. Such causes are however fundamentally different from the focus of this work since they are imposed by *external* effects (e.g., by faults in case of disk failure and network timeouts, or lack of enough resources such as disk space), whereas in our case, the main cause of the failure is *internal* to the operation of the application; thus we consider such above factors as out of scope of this paper.

In this work, we propose and implement a new memory-aware technique to determine the number of Reduce tasks as well as reconfiguration of Reduce slots. The primary objective of this technique is avoiding job abortion due to lack of memory in Reduce phase. Furthermore, it aims to increase the performance of Reduce phase and consequently, decrease the execution time of applications. The key idea of this new technique, called Mnemonic, is to automate the configuration of Reduce slots in term of memory as well as determination of number of Reduce tasks. The first step of Mnemonic is to determine the memory size of each Reduce slot based on the available memory of machine that hosts the slots. At the same time, it estimates the volume of intermediate data that is

The authors are with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. Maziar Goudarzi is the corresponding author
E-mail: {mnabavi,shmozaffari}@ce.sharif.edu, goudarzi@sharif.edu

going to be generated through profiling. Finally, it determines the number of Reduce tasks based on the memory size of slots and estimated volume of intermediate data. We have implemented Mnemonic using Hadoop v1.2.1 and evaluated our proposed approach; we have used applications and data sets of representative PUMA MapReduce benchmark suit [15]. Our experimental results indicate that Mnemonic can completely omit the job failure related to lack of memory. It also can reduce the execution time of Reduce phase by up to 58.27%, 79.36%, and 88.79% compared with Memory Oblivious approach and two Fine Grain algorithms − see section III − respectively.

Our major contributions in this work are as follows:

- We accentuate the impact of memory limitation on management of intermediate data in Reduce phase of MapReduce applications and show how memory limitation can lead to job failure or serious performance degradation.
- We investigate the slot configuration in Hadoop clusters in more details compared with previous approaches and configure each individual slot in term of memory size.
- We propose Mnemonic mechanism and implement it in Hadoop to determine number of Reduce tasks with respect to volume of intermediate data and memory size of Reduce slots. Experimental results indicate that Mnemonic can completely eliminate job failure due to memory limitation. Furthermore, it can increase the performance of applications compared with three rival approaches.

The rest of the paper is organized as follows. We have motivated our work in section I-A. The architecture of our new approach, Mnemonic, is presented in section II. Experimental results for evaluating the proposed approach are illustrated in section III. We review the related work in section IV. Finally, we conclude the paper and draw new directions for further research in section V.

A. Motivation

For a number of applications, the Reduce phase has a short execution time, while for many others it contributes significantly to total execution time. To quantify this, we have run nine applications from PUMA benchmark suit [15] with 1 GB to 6 GB input data size. The execution time breakdown of these applications is presented in Fig.1 (see Section 3 for specs of platform which results are obtained by). When the size of input data is increased from 1 GB (a) to 6 GB (b), for three applications (e.g., Classification, HistogramMovies, HistogramRatings) the share of Reduce phase in total execution time drops down dramatically. We can conclude that in these applications the execution time of Reduce phase for large scale jobs is negligible. However, for the rest of applications the execution time of Reduce phase grows up almost linearly with the volume of input data. We can also perceive from Fig.1 that the Reduce phase can contribute more than 50% to total execution time, and hence, it is important to reduce this time.

After showing the considerable share of Reduce phase in total execution time, we demonstrate the sensitivity of Reduce phase to available memory. Fig.2 illustrates average and maximum memory usage of each phase for different input sizes. While Map phase memory usage is effectively independent of input size, Reduce phase memory usage significantly changes with input size. Thus, tuning the memory size can improve the performance of Reduce phase. These observations can be intuitively justified as below: During transfer of input data to HDFS, input data is divided into equal size chunks (default size in Hadoop is 64MB) and each Map task processes one chunk at a time. Consequently, the Map phase always consumes the same amount of memory provided that the number of concurrent Map tasks, which is equal to the number of Map slots, is constant. However, in the Reduce phase, total number of Reduce tasks is set in benchmark configuration files, and hence, depending on the input data and the resulting intermediate data size, Reduce tasks need to bring different amounts of intermediate data into memory to process . Despite predetermined number of Map tasks and their set input data size, Reduce phase is not set and can be changed to better manage memory during processing and improve performance.

As described in Fig.1, for six out of nine applications, the execution time of Reduce phase contributes significantly to total execution time. We have run those applications eight times and each time we only changed the number of Reduce tasks to evaluate its effect on performance of Reduce phase. The machine we use in the experiments has 4 Reduce slots with 4 GB of RAM for each slot. The total memory available in the machine is 7 GB so the slots share the memory with each other. The depicted results in Fig.3 indicate that in a number of experiments, the Reduce phase cannot be completed and consequently the job fails, and hence, no point is plotted in the figure. We can also perceive from Fig.3 that execution time varies by the number of Reduce tasks. Up to this point, this implicates that number of Reduce tasks directly affects

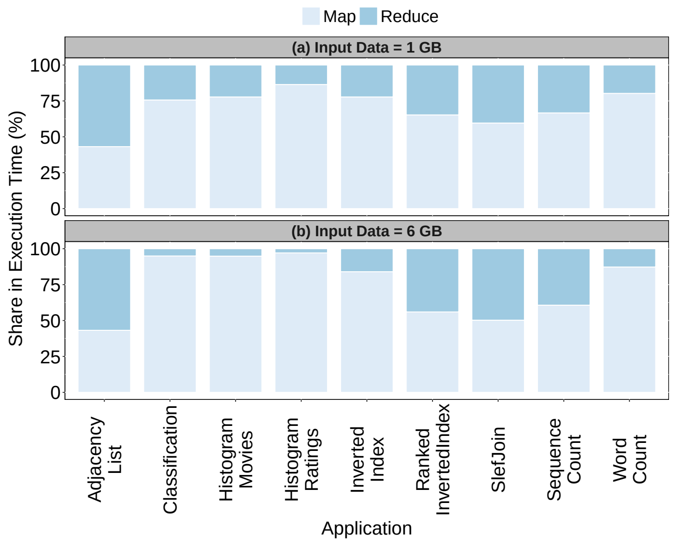


Fig. 1. Share of Map phase and Reduce phase in total execution time. In six out of nine applications, share of Reduce phase is significant and even surpasses the Map phase. By increasing the size of input data from 1GB to 6GB, this share is constant or even on the increase. Consequently, it worth trying decreases execution time of Reduce phase.

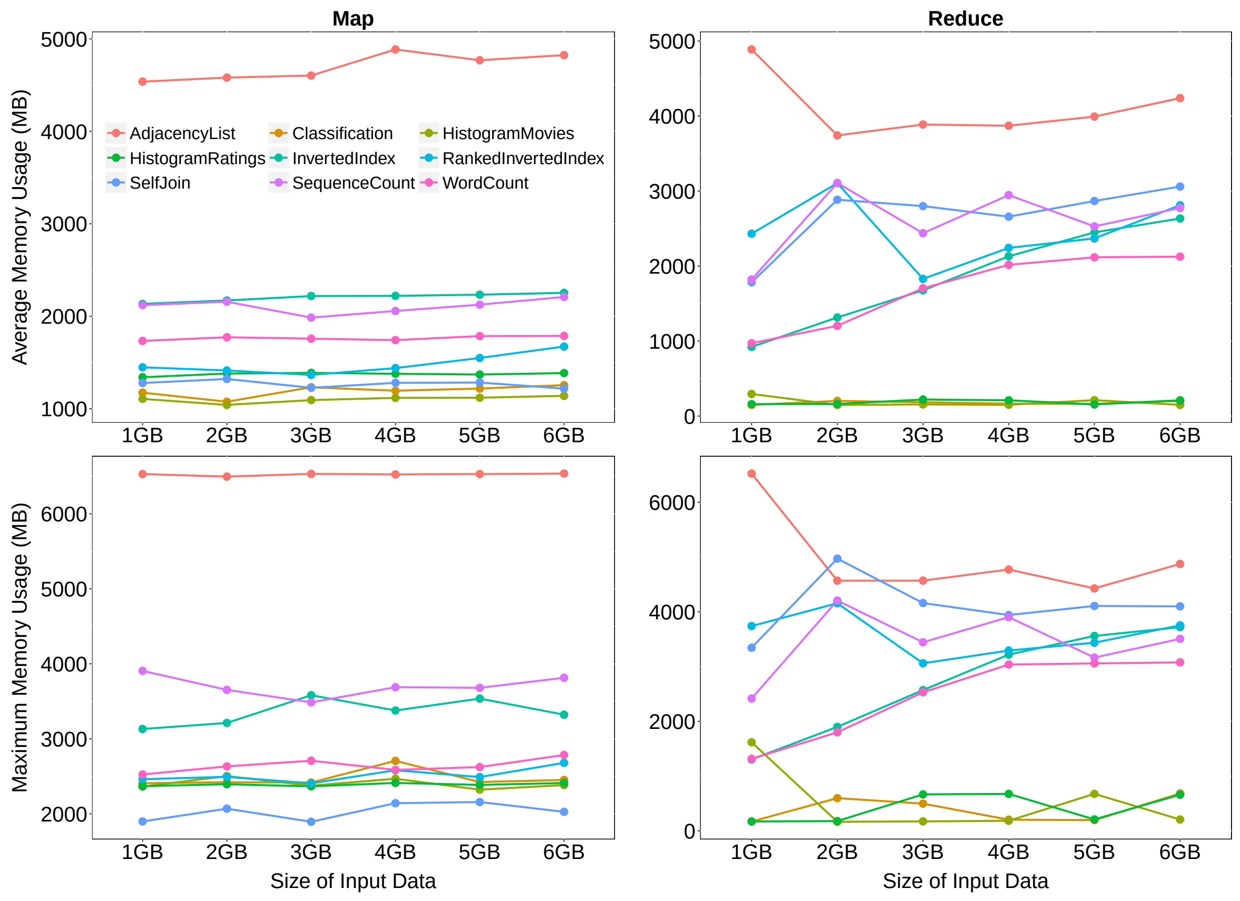


Fig. 2. Memory usage pattern of Map and Reduce phases for different sizes of input data. For Map phase (left column) both average and maximum memory usage are almost constant and independent of size of input data. In contrast, Reduce phase (right column) shows different memory usage under different input data. We conclude that tuning memory usage can affect performance of Reduce phase, while it might not be effective regarding Map phase.

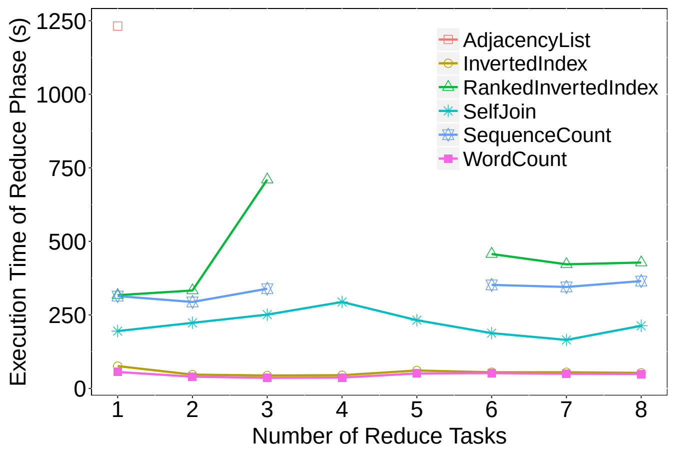


Fig. 3. Impact of number of Reduce tasks on Reduce phase performance. The points that are not plotted indicate job failure because of out of memory error in Reduce phase. We also can perceive the impact of number of reduce tasks on the performance. We see that different number of Reduce tasks yields different execution times.

the status of Reduce phase (finished/killed) as well as its performance.

Now, we plot the maximum memory usage of applications for different number of Reduce tasks and compare it against total memory of machine that has executed the applications. Comparing the results presented in Fig.4 with Fig.3, it clearly show that whenever the memory usage has approached the total memory, the application has encountered out of memory error and consequently failed. As an example, in AdjacencyList application we can see that except for one Reduce task, for other number of Reduce tasks the maximum memory usage has approached the total memory and consequently the job has failed. So, in Fig.3 only one point is plotted for this application. The same happens for RankedInvertedIndex and SequenceCount applications when they have 4 or 5 Reduce tasks and consequently they fail too. This can be justified as follows:

We mentioned earlier that number of Reduce slots of machine is four. Thus the machine can run four Reduce tasks simultaneously. When the number of Reduce tasks is four, all of them will be run. Consequently, all the intermediate data is being brought into memory for processing. In SequenceCount and RankedInvertedIndex applications, the amount of intermediate data is more than 6 GB. Since the machine's memory capacity is 7GB and a portion of it is reserved for OS and other applications functionalities, the out of memory error occurs and job fails. Almost the same happens in the case of five Reduce tasks. However, when the number of Reduce tasks is one, two, or three, we do not observe the same phenomena

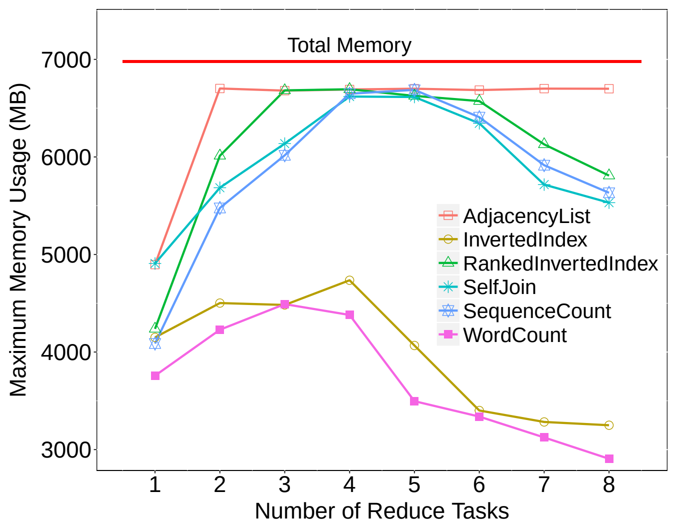


Fig. 4. Maximum memory usage of applications under different number of reduce tasks. Comparing this figure and Figure 3 illustrates that when memory usage approaches total memory and begins to exceed it, Reduce phase fails and consequently we face job failure.

for SequenceCount and RankedInvertedIndex. The reason is that for those numbers of Reduce tasks, the CPU becomes bottleneck and hence, the applications cannot bring all the intermediate data into the memory. So, no out of memory error occurs.

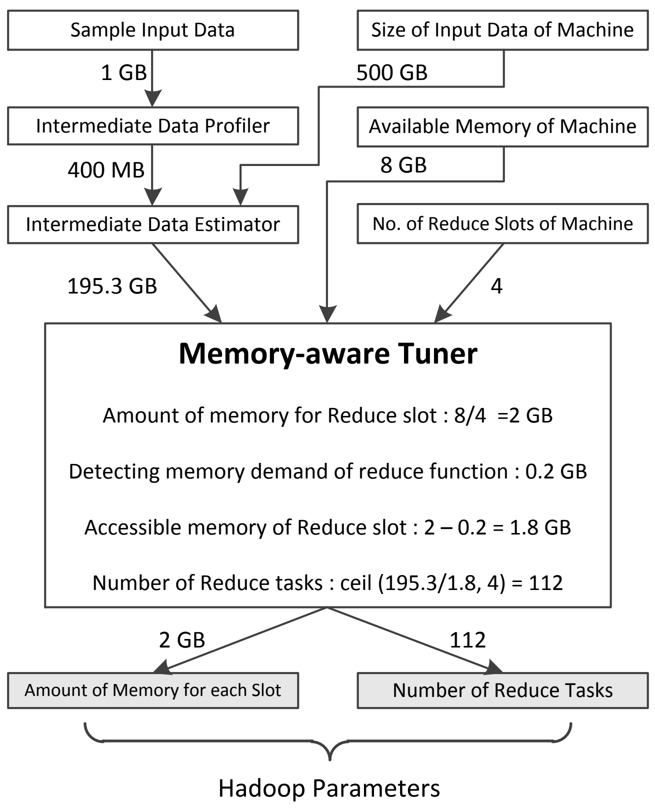

Sample Input Data
1 GB
Size of Input Data of Machine
500 GB
Intermediate Data Profiler
400 MB
Available Memory of Machine
8 GB
Intermediate Data Estimator
No. of Reduce Slots of Machine
195.3 GB
4
Memory-aware Tuner
Amount of memory for Reduce slot : 8/4 =2 GB
Detecting memory demand of reduce function : 0.2 GB
Accessible memory of Reduce slot : 2 – 0.2 = 1.8 GB
Number of Reduce tasks : ceil (195.3/1.8, 4) = 112
2 GB
112
Amount of Memory for each Slot
Number of Reduce Tasks
Hadoop Parameters

Fig. 5. Flow of Mnemonic memory tuning approach.

II. Mnemonic Architecture

According to the observations in section I-A, it is important to determine the number of Reduce tasks with respect to total memory in order to avoid job failure and simultaneously improve the performance of Reduce phase. Since the memory is distributed among the slots and tasks are also executed by those slots, the first step in a memory aware approach should be determining the size of memory dedicated to each slot. After that, the number of Reduce tasks should be determined accordingly. *Memory size of each slot* and *volume of intermediate data* are two important factors that should be considered when determining number of Reduce tasks. As previously mentioned, memory size of slots is important because they run the Reduce tasks. Furthermore, Reduce tasks process the intermediate data, so the volume of intermediate data is another decisive factor.

We have designed *Mnemonic* to tackle the problems related to memory in Reduce phase of MapReduce applications. Mnemonic is a memory-aware approach that aims to 1) completely eliminate the probability of job failure due to insufficient memory and 2) increase the performance of Reduce phase by determining the proper number of reduce tasks considering the amount of available memory. Fig.5 illustrates the architecture of Mnemonic by a walk through.

Mnemonic uses profiling to collect information about volume of intermediate data generated by Map phase. It first executes the application for a sample input data and profiles the amount of intermediate data. By the use of gathered information in previous step and the size of input data for each machine in cluster that executes the application, Mnemonic estimates the amount of intermediate data that will be generated on each machine. Firstly our observations implicate that there is a linear relationship between the size of input data and volume of generated intermediate data in our employed applications. Fig.6 depicts this linear relationship, which we use to estimate the amount of generated intermediate data for large scale jobs with substantial input data. It should be noted that this *linear* relationship might not be applicable to all the applications. A second possible case is where there is a *deterministic* but *nonlinear* behavior; in such case, a once-off profiling to identify the pattern of generated intermediate data size would be enough. The linear estimator then can be replaced by this obtained pattern in the Intermediate Data Estimator module in Fig.5. The third possible pattern of size of intermediate data generation is a *non-deterministic behavior*. In this pattern, the amount of intermediate data depends completely on the input data. With change in input data, the pattern would dramatically change. Our Mnemonic approach can handle the first and second ones (linear and nonlinear behaviors), but would lose its effectiveness in case of non-deterministic behavior until equipped with a suitable prediction mechanism.

Similar to collected data-side information, Mnemonic needs information from machine-side too. It needs to have knowledge about the number of configured reduce slots on the machine and also the available memory of it. Memory-aware Tuner module in Fig.5 utilizes data-side and machine-side information and determines 1) amount of memory for each

Reduce slot and 2) number of Reduce tasks. Since it is common practice to set one Map slot and one Reduce slot per core [16], [17], each Reduce slot has one core which determines the computing capacity of it. Our experiments reveal that regarding memory size of Reduce slots, the best practice is to divide the memory among slots equally as long as the cores of machine are homogeneous.

In addition to *volume of intermediate data* and *memory size of reduce slots*, number of Reduce tasks also depends on *memory demand of Reduce function* of application. One simple way for determining the number of Reduce tasks is to divide the volume of intermediate data by the *amount of memory for Reduce slot* − Memory-aware Tuner in Fig.5. While this solution might seem very fast and easy, it ignores the memory demand of the Reduce function that processes the data. Since the Reduce task processes data in memory, discarding memory demand of the Reduce function will lead to memory overload and consequently job failure. Mnemonic considers this fact and identifies the memory demand of Reduce function to subtract it from memory size of Reduce slot to obtain the *accessible memory of Reduce slot* − see Memory-aware Tuner module in Fig.5.

After that, Mnemonic proceeds to specify the number of Reduce tasks. It first divides the volume of intermediate data by the accessible memory of Reduce slot. Then, it maps the obtained number to the smallest following integer, that is a multiple of number of Reduce slots of machine, using ceil function. The reason for a ceiling is to increase the parallelism of Reduce tasks execution. The Hadoop receives the determined number of Reduce tasks as well as the memory amount of each slot and sets its corresponding internal parameters with respect to them. To set the memory amount of each slot, Mnemonic changes the amount of *mapred.reduce.child.java.opts* parameter in *mapred-site.xml* file and overrides the default value for each individual slave node. To set the number of Reduce tasks, this should be applied in the configuration files of the application; Mnemonic applies the proper values per application in the PUMA benchmark suit.

The overheads of Mnemonic can be broken down into two parts: offline and online. The main overhead imposed by Mnemonic is the profiling phase,which is offline and once-off, where it wants to derive the pattern of intermediate data generation. Since in this phase only a small portion of input data is profiled, the overhead is negligible compared with total execution time of job. Moreover, this overhead is compensated by the improvement in performance of application due to proper memory management. For repeating jobs, which includes most of practical uses, it can be overlapped with the first run of the job to hide even this small overhead for the next runs. The other overheads of Mnemonic are (i) computing and (ii) setting the memory amount of each slot by changing the configuration options in configuration file of each slave. The computation is of O(1) but settings should be done per slave node, so time complexity of this part is O(n) where n is the number of slaves in Hadoop cluster. We can see that the online part of our technique is a light-weight mechanism with negligible performance overhead even at large datasets since it is independent from data size.

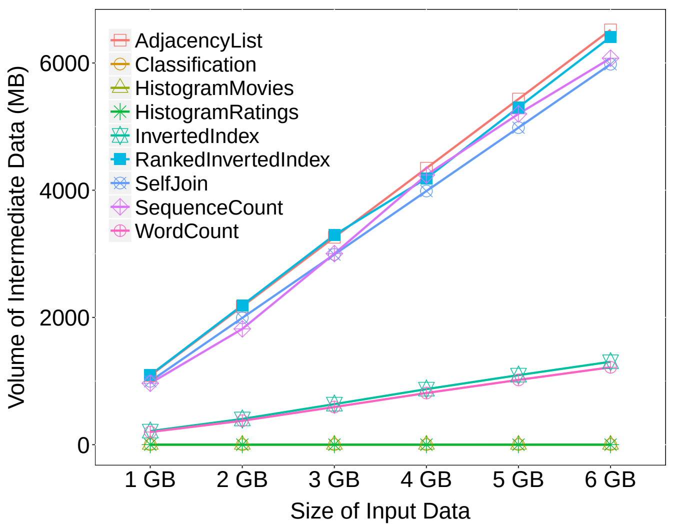


Fig. 6. Linear relationship between size of input data and volume of intermediate data. We estimate the amount of intermediate data for large input data according to this linear relationship. However, in some applications we see nonlinear relationship. Even in that case, once-off profiling can help find the pattern of relationship.

In this paragraph, we discuss the multi-job environments because many Hadoop clusters operate in such mode. While we have assumed an environment with single job running at each time, our approach is still applicable in a multi-tenant environment, such as clouds, where the (virtual) machines are not shared among jobs and every job has its own machines. Even if the machines are shared among jobs, we can consider the worst case scenario and adjust the parameters accordingly. In this situation, the only parameter that needs adjustment is the *memory demand for Reduce function* − see Fig.5. We can consider the worst case for it and then determine the amount of slot memory based on that. Then, for each job we can determine the number of Reduce tasks separately based on the slots memory and continue as before.

III. Evaluation

We use six applications and their corresponding input data set from PUMA benchmark suit to evaluate our proposed approach. The algorithms and implementations of benchmark applications are publicly available from PUMA suit [15]. We have used the same implementation without any change. The applications are:

- *Adjacency-List*: for each vertex of a graph, produces the list of its neighbors. The results can be used in algorithms such as PageRank.
- *Inverted-Index*: generates the mapping of word to document for a set of documents and their constituent words.
- *Ranked-Inverted-Index*: for a list of words and number of their repetitions in documents, generates the decreasing list of documents that given words appears in them.
- *Self-Join*: gets the association of k fields and generates it for k+1 field.
- *Sequence-Count*: finds the number of all individual sets of three successive words per document.

- *Word-Count*: calculates the number of appearance of each word in a set of documents

Hadoop version 1.2.1 is used as the MapReduce framework on a machine with four cores and 7 GB of RAM. While we have used Hadoop version 1 in our work, Mnemonic is applicable to newer version of Hadoop, Yarn [18], too. The only difference between Hadoop version 1 and Yarn, concerning our work, would be the name of parameters that Mnemonic needs to modify, but the main concept remains the same. The operating system of machine is Ubuntu 12.04. The size of input data of all the applications is 6 GB from data sets in PUMA suit [15]. The PUMA data sets are publicly available similar to PUMA applications. Data sets are accessible in different sizes and one can download a portion or all of them. We have used a portion of data but we have not applied any kind of filtering on them and used the raw data intact, and this is the same setup used for all previous observations. We compare Mnemonic with three rival approaches:

- *Memory Oblivious*: this approach is neither aware of memory size of Reduce slots nor memory demand of Reduce tasks. It uses an inefficient profiling mechanism that changes the number of Reduce tasks from one to eight and chooses the one with the best performance.
- *Fine Grain 1 and Fine Grain 2*: the main concept of Fine Grain approach is to increase the number of Reduce tasks to decrease chance of job failure. The higher the number of Reduce tasks, the lower their input data size. This lower input size leads to lower memory demand when executing in parallel. Therefore, this lower memory demand decreases the possibility of job failure due to lack of memory. While this approach tries to avoid job failure, it does not pay attention to job performance. We see the effect of performance ignorance of this approach in the following figures. The number of Reduce tasks in this approach is constant and equals to 64 and 128 for Fine Grain 1 and Fine Grain 2 respectively.

Fig.7 depicts the execution time of Reduce phase by different approaches. As can be seen, our proposed Mnemonic approach can surpass all the other ones in all the applications. The average reduction of execution time compared with Memory Oblivious, Fine Grain 1 and Fine Grain 2 is 19.8%, 42.58%, and 62.78% and the maximum is 58.27%, 79.36%, and 88.79% respectively. These results also indicate that Fine Grain approach sacrifices performance to avoid memory shortage. The performance of this approach is usually the worst. Even Memory Oblivious yields better performance in a number of applications. The source of such poor performance is the overhead of task creation and termination. When the number of Reduce tasks increase, the negative impact of this overhead on performance becomes more significant. The value of profiling is the other conclusion that can be derived from Fig.7. Comparing Fine Grain approach with Mnemonic and Memory Oblivious indicates that while profiling imposes overhead, it can improve the performance if used wisely.

After evaluating the effect of different approaches on performance, in the following we discuss the results in more details. Fig.8 illustrates the maximum memory usage, average memory

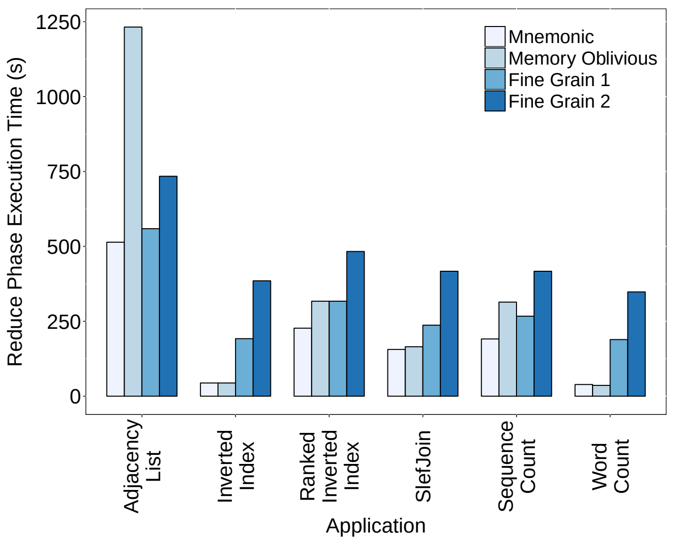


Fig. 7. Execution time of Reduce phase under different approaches. Mnemonic can always yield the best performance since it considers volume of intermediate data as well as memory size of slots when deciding on the number of Reduce tasks.

usage, and average CPU utilization of Reduce phase. It also presents number of Reduce tasks decided and created by each approach.

If we take a closer look to the number of Reduce tasks in Fig.8(a), we can see that while they are constant in both Fine Grain cases, they vary in Memory Oblivious and Mnemonic. The amount of variation in Memory Oblivious is low since it only changes the number between one and eight. However, for Mnemonic the amount of variation is higher (from 4 to 40). It shows that Mnemonic effectively leverages the different behavior of applications in terms of intermediate data and decides on the number of Reduce tasks based on that. The average utilization of CPU − see Fig.8(b) − for Fine Grain is higher than others while its performance is lower. It confirms that the overhead of Reduce tasks creation and termination significantly wastes a portion of CPU time. Furthermore, the low CPUutilization of Memory Oblivious demonstrates its weakness in using the resources efficiently. The Mnemonic stands between two approaches and can effectively use the CPU to achieve high performance.

Considering the memory usage pattern of Memory Oblivious and Fine Grain in Fig.8(c) and Fig.8(d), we conclude that their behavior with respect to memory is as the opposite of their CPU behavior. For example, the memory utilization of Fine Grain is lower than Memory oblivious since the size of Reduce tasks data is smaller. So it cannot utilize the accessible memory effectively to lessen the execution time of Reduce phase. The Mnemonic again stands between other approaches.

The conclusion derived from memory and CPU usage information is that Mnemonic is successful in improving performance of Reduce phase because it can leverage available resources simultaneously. Unlike Mnemonic, the two other approaches lack effective use of either memory (Fine Grain) or CPU (Memory Oblivious), and hence, they cannot reach the performance of Mnemonic.

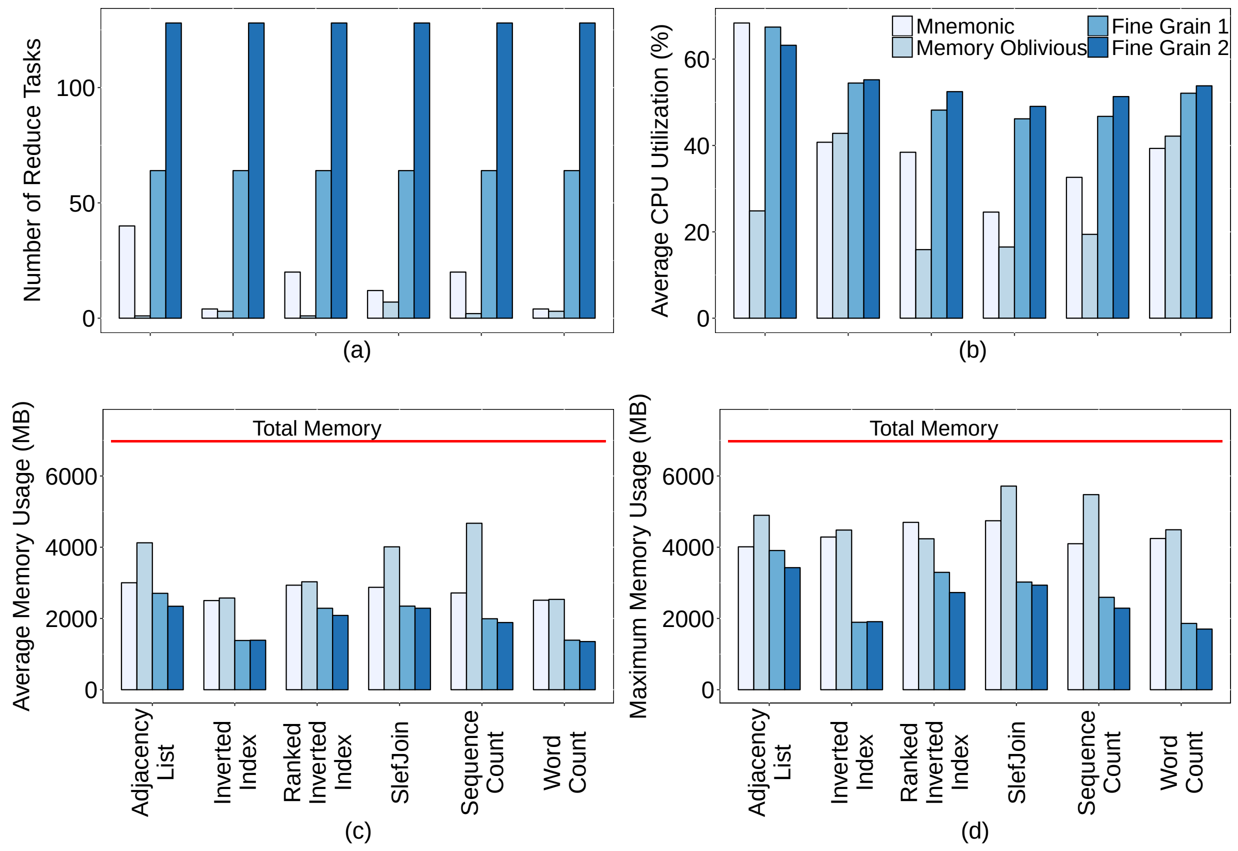


Fig. 8. The results in more details. (a) indicates the number of Reduce tasks by each approach. While fine grain approaches always have the same number of Reduce tasks for every job, Mnemonic chooses different number of tasks based on application characteristics. The average CPU utilization is shown in (b). The average memory usage and maximum memory usage is illustrated in (c) and (d) respectively.

IV. Related Work

An Important direction for enhancing the performance of MapReduce applications is considering intermediate data. Since the Reduce phase processes the intermediate data to generate final result, any technique that can improve the management of intermediate data will lead to better performance in Reduce phase. Hadoop stores the intermediate data on local disk of nodes temporary prior to being processed by Reduce tasks. Hence, several techniques are proposed to improve the performance of storage system for intermediate data [3], [4], [19]. Two common policies for storing/replicating intermediate data in Hadoop are locally stored (LS) and Distributed File System (DFS). The former stores the intermediate data on the same node that has generated it, while the latter distributes it on several nodes. Reference [3] studies the impact of aforementioned policies on job reliability and energy consumption. In the presence of LS approach, machine failure leads to intermediate data lost. Consequently, all the previous tasks should be re-executed to generate it again. So, [4] prefers DFS rather than LS and proposes BlobSeer as the storage layer to realize DFS. Compressing the intermediate data in order to minimize its volume and consequently improve the storage I/O performance is proposed by [5], [6], [20]. Reducing movement time of intermediate data during shuffle phase [7]–[9] can increase the performance of MapReduce applications. Hence, approaches such as ShuffleWatcher [9] and JVM bypassing [7] are introduced to address it. Since none of the above works has addressed the memory issue of intermediate data, our work can be assumed as complementary to them.

V. Conclusion

Memory has an important role in the performance of Reduce phase in many MapReduce applications. It not only can degrade the performance, but also can lead to job failure due to lack of memory. So, if an approach considers memory correctly in the process of decision making about Reduce slots configuration as well as number of Reduce tasks, it can achieve high performance. Our memory aware approach, Mnemonic, considers this fact and achieves high performance compared with Memory Oblivious and Fine Grain approaches. Our major contributions in this approach are 1) accentuating the impact of memory on intermediate data management, 2) investigating the slot configuration and configure memory size of each slot, and 3) setting the number of Reduce tasks as well as memory size of Reduce slots properly to eliminate job failure and increase the performance of applications. The experimental results indicate that while Mnemonic can outperform other approaches, it still cannot fully utilize the available computing resource e.g., memory and CPU. This warrants further work and analysis for more efficient approaches that can achieve higher performance.

Acknowledgment

This research is supported by grant number G930826 from Sharif University of Technology. We are grateful for their support.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache hadoop," https://hadoop.apache.org/, accessed May 28th 2016.

[3] J.-C. Lin, F.-Y. Leu, and Y.-P. Chen, "Analyzing job completion reliability and job energy consumption for a heterogeneous mapreduce cluster under different intermediate-data replication policies," *The Journal of Supercomputing*, vol. 71, no. 5, pp. 1657–1677, 2014.

[4] D. Moise, T.-T.-L. Trieu, L. Bougé, and G. Antoniu, "Optimizing intermediate data management in mapreduce computations," in *Proceedings of the first international workshop on cloud computing platforms*. ACM, 2011, pp. 1–7.

[5] Z. Xue, G. Shen, J. Li, Q. Xu, Y. Zhang, and J. Shao, "Compression-aware i/o performance analysis for big data clustering," in *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. ACM, 2012, pp. 45–52.

[6] G. Ruan, H. Zhang, and B. Plale, "Exploiting mapreduce and data compression for data-intensive applications," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*. ACM, 2013, pp. 1–8.

[7] Y. Wang, C. Xu, X. Li, and W. Yu, "Jvm-bypass for efficient hadoop shuffling," in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2013, pp. 569–578.

[8] W. Yu, Y. Wang, X. Que, and C. Xu, "Virtual shuffling for efficient data movement in mapreduce," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 556–568, 2015.

[9] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proceedings of the USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 1–12.

[10] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, July 2013.

[11] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders, "Failure scenario as a service (fsaas) for hadoop clusters," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*. ACM, 2012, pp. 1–6.

[12] F. Dinu and T. Ng, "Understanding the effects and implications of compute node related failures in hadoop," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 187–198.

[13] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, "An empirical study on quality issues of production big data platform," in *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 17–26.

[14] X. Chen, C.-D. Lu, and K. Pattabiraman, "Failure analysis of jobs in compute clouds: A google cluster case study," in *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2014, pp. 167–177.

[15] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012.

[16] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-aware scheduling of mapreduce jobs for big data applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10, pp. 2720–2733, 2015.

[17] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni, "Optimizing power and performance trade-offs of mapreduce job processing with heterogeneous multi-core processors," in *IEEE 7th International Conference on Cloud Computing (CLOUD)*. IEEE, 2014, pp. 240–247.

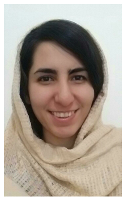
[18] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, pp. 1–16.

[19] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 287–300.

[20] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency," in *Proceedings of the first ACM SIGCOMM workshop on Green networking*. ACM, 2010, pp. 23–28.

Seyed Morteza Nabavinejad is a Ph.D. candidate at the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He received the B.Sc. degree in Computer Engineering from Ferdowsi University of Mashhad and the M.Sc. in Computer Architecture from Sharif University of Technology in 2011 and 2013, respectively. He is currently working at Energy Aware Systems (EASY) Laboratory under supervision of Dr. Maziar Goudarzi. His research interests include big data processing, cloud computing, green computing, and energy aware datacenters.

Maziar Goudarzi is an Assistant Professor at the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He received the B.Sc., M.Sc., and Ph.D. degrees in Computer Engineering from Sharif University of Technology in 1996, 1998, and 2005, respectively. Before joining Sharif University of Technology as a faculty member in September 2009, he was a Research Associate Professor at Kyushu University, Japan from 2006 to 2008, and then a member of research staff at University College Cork, Ireland in 2009. His current research interests include architectures for large-scale computing systems, green computing, hardwaresoftware codesign, and reconfigurable computing. Dr. Goudarzi has won two best paper awards, published several papers in reputable conferences and journals, and served as member of technical program committees of a number of IEEE, ACM, and IFIP conferences including ICCD, ASP-DAC, ISQED, ASQED, EUC, and IEDEC among others.



Shirin Mozaffari Shirin Mozaffari is a senior student in the Department of Computer Engineering at Sharif University of Technology, Iran. She is a research assistant in the Energy Aware System Laboratory under supervision of Dr. Goudarzi. Her research interests include cloud computing, big data Processing and resource allocation..