Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 408921, 7 pages
http://dx.doi.org/10.1155/2015/408921

Research Article

A Replication-Based Mechanism for Fault Tolerance in MapReduce Framework

Yang Liu and Wei Wei

College of Information Science and Engineering, Henan University of Technology, Zhengzhou 450001, China

Correspondence should be addressed to Wei Wei; nsyncw@126.com

Received 20 October 2014; Accepted 31 January 2015

Academic Editor: Hui-Huang Hsu

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. In cloud environment, node and task failure are no longer accidental but a common feature of large-scale systems. Current rescheduling-based fault tolerance method in MapReduce framework failed to fully consider the location of distributed data and the computation and storage overhead of rescheduling failure tasks. Thus, a single node failure will increase the completion time dramatically. In this paper, a replication-based mechanism is proposed, which takes both task and node failure into consideration. Experimental results show that, compared with default mechanism in Hadoop, our mechanism can significantly improve the performance at failure time, with more than 30% decreasing in execution time.

1. Introduction

MapReduce is an emerging programming paradigm that has gained more and more popularity due to its ability to support complex tasks execution in a scalable way [1, 2]. MapReduce is used for processing large data sets by parallelizing the processing on a large number of distributed nodes. Data is stored in splits that are processed by separate tasks. Processing is done in two phases: map and reduce. A MapReduce application is implemented in terms of two functions that correspond to these two phases. A map function processes input data expressed in terms of key-value pairs and produces an output also in the form of key-value pairs. A reduce function picks the output of the map functions and produces results. It is shown that many applications can be implemented using this programming model [1].

This popularity is also shown by the appearance of open-source implementations, like Hadoop that is now extensively adopted by Yahoo and many other companies [3]. Hadoop [4] is an open-source software framework implemented using Java and is designed to support data-intensive applications executed on large distributed systems. It is a project of the Apache Software Foundation and is a very popular software tool due, in part, to it being open-source. Yahoo! has contributed to about 80% of the main core of Hadoop, but many other large technology organizations have used or are currently using Hadoop, such as, Facebook, Twitter, LinkedIn, and others [5]. The Hadoop framework is comprised of many different projects, but two of the main ones are the Hadoop Distributed File System (HDFS) and MapReduce. Both the initial input and the final output of a Hadoop MapReduce application are normally stored in HDFS [3], which is similar to the Google File System [6].

However, in large-scale distributed computing environment such as cloud environment, node and task failure are no longer *accidental* but a common feature. Research discovered that failure has a significant impact on system performance in large-scale systems [4]. Every year in a cluster, 1% to 5% hard disks will be scrapped, up to 20 racks and 3 routers will go down, and servers will go down at least twice with 2% to 4% scrap rate each year. It shows that failure also occurs daily even in a distributed system with up to ten thousand super reliable servers (MTBF of 30 years) [5]. For the cloud environment consisting of a large number of inexpensive computers, node and task failure become a more frequent and wide spread problem, which must be handled by some

effective fault tolerance method. It is difficult to achieve 100% fault tolerance because there are many physical circumstances that just can not be planned for, but the goal of fault tolerance is to plan for all common failures [7]. In managing fault tolerance it is important to eliminate Single Points of Failure (SPOF), which are single elements of the system, that when they fail, they can bring down the whole system [8].

As a result, fault tolerance is as important in the design of the original MapReduce as in Hadoop. Specifically, a MapReduce job is a unit of work that consists of the input data, a map and a reduce function, and configuration information. Hadoop breaks the input data in splits. Each split is processed by a map task, which Hadoop prefers to run on one of the nodes where the split is stored (HDFS replicates the splits automatically for fault tolerance). Map tasks write their output to local disk, which is not fault tolerance. However, if the output is lost, as when the machine crashes, the map task is simply executed again on another computing node. The outputs of all map tasks are then merged and sorted by an operation called shuffle. This kind of rescheduling is inefficient and can be improved in many ways.

Since the MapReduce-based programs generate a lot of intermediate data, which is critical for completing the job, this paper views failover of intermediate data as a necessary component of MapReduce framework, specifically targeting and minimizing the effect of tasks and nodes failure on performance metrics such as job completion time. We propose new design techniques for a new fault tolerance mechanism, implement these techniques within Hadoop, and experimentally evaluate the prototype system.

2. Related Work

MapReduce is a programming model and an associated implementation for processing and generating big data [9]. It is initially designed for parallel processing of big data using mass cheap server clusters and putting scalability and system availability on the prior position. Within Google Company, more than 20 PB of data is produced every day and 400 PB every month. Yahoo adopts Hadoop, an open-source MapReduce framework. Facebook uses it to process data and generate reports, while Amazon Company uses elastic MapReduce framework for large amount of data-intensive tasks [10]. MapReduce has obvious advantage over other programming models like MPI, and a single task failure does not affect the execution of other tasks because of the independence between mapper and reducer task. It has drawn a lot of attention for its benefits in simple programming, data distribution, and fault tolerance, which has been widely used in many areas, like data mining and machine learning.

Google Company pointed out that, in a cloud environment with averagely 268 nodes, each MapReduce job is accompanied with failure of five nodes [11]. On the other hand, large amount of data is usually accompanied with data inconsistency or even data loss, and incorrect data record will result in task failure or even failure of the entire job. MapReduce uses a rescheduling-based fault tolerance mechanism to ensure the correct execution of the failed task. Since the rescheduling failed to fully consider the location of distributed data, in the scenario of node failure, all the completed tasks on the failed node will start over, which shows severely low efficiency. If the failure detection timeout in Hadoop is set to 10 minutes (the default value), a single failure will cause at least 50% increase in completion time [12]. If each input split contains one bad record, the entire MapReduce job will have a 100% runtime overhead, which is not acceptable for those users with rigorous SLA requirements. So it clearly shows the need for effective algorithms that can reduce delays caused by these failures [13].

In [14], tests show that, in seven types of cluster with different MTBF (Mean Time between Failures), MapReduce job with three replicas can achieve better performance than that with one replica, because more replicas can reduce the chances of data migration when rescheduling jobs at failure time. Reference [15] discussed an alternative fault tolerance scheme, the state-based Stream MapReduce (SMR), which is suitable for handling continuous data streaming applications with real-time requirements, such as financial and stock data. The key feature is low-overhead deterministic execution which reduces the amount of persistently stored information. Reference [16] proposed a method to replicate intermediate data to the reducer, but this method will produce a large number of I/O operations, consume a lot of network bandwidth, and only support recovery for single node failure. Reference [17] proposed a method to improve performance of fault tolerance by replicating data copies. Reference [18] presented an intelligent scheduling system for web service, which considers both the requirements of different service requests and the circumstances of the computing infrastructure which consists of various resources. Reference [19] described the priority of fairness, efficiency, and the balance between benefit and fairness, respectively, and then recompiled the CloudSim and simulated three task scheduling algorithms on the basis of extended CloudSim, respectively.

We proposed a replication-based mechanism for fault tolerance in MapReduce framework, which can significantly reduce the average completion time of jobs. Unlike traditional fault tolerance mechanism, it will reschedule tasks on the failed node to another available node without starting over again but reconstruct intermediate results quickly from the checkpoint file. The preliminary experiments show that, under a failure condition, it outperforms default mechanism with more than 30% increasing in performance and incurs only up to 7% overhead.

3. Algorithms

3.1. MapReduce Programming Model. In MapReduce programming model, the calculation process is decomposed into two main phases, namely, the mapper stage and reducer stage. For one piece of input data, the reducer stage only starts when the mapper stage is completed. A MapReduce job includes M mapper tasks and R reducer tasks. In mapper stage, multiple mapper tasks run in parallel, and one mapper task will read an input split and perform a mapper function,

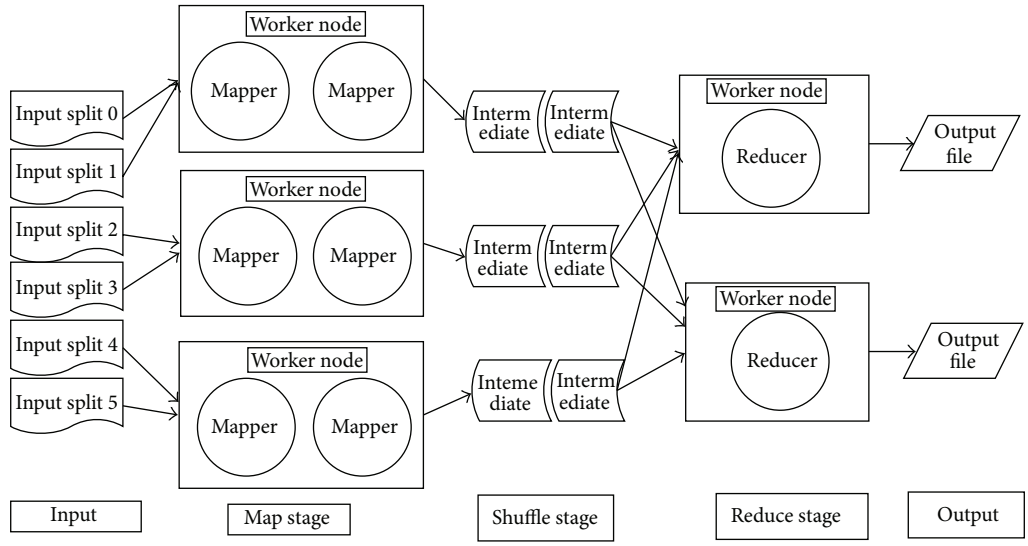

Figure 1: Execution process of MapReduce model.

where mapper tasks are independent of each other. Mapper tasks will produce a large number of intermediate results in local storage. Before reducer function is called, the system will classify the generated intermediate result and shuffle result with the same key to reducers. A reducer task will execute a reduce function and generates an output file, and eventually a MapReduce job will generate R output files which can be merged to get the final result. When programming, developers need to write a mapper and a reducer function:

$$\begin{aligned} &\text{map: } (\text{input\_data}) \longrightarrow \left\{\left(\text{key}_j, \text{value}_j\right) \mid j = 1, \ldots, k\right\} \\ &\text{reduce: } \left(\text{key}, \left[\text{value}_1, \ldots, \text{value}_m\right]\right) \longrightarrow \left(\text{key}, \text{final\_value}\right). \end{aligned} \tag{1}$$

The MapReduce model is shown in Figure 1.

Node and task failure are prone to happen during a MapReduce job execution process. When a node fails, MapReduce will move all mapper tasks on the failed node to available nodes. This kind of rescheduling method is simple but often introduces a lot of time cost; thus for users with high responsiveness requirements, its negative influence is not acceptable.

3.2. Improved Rescheduling Algorithm. In the paper, a replication-based mechanism in MapReduce framework is proposed, which uses a checkpoint-based active replication method to provide better performance. Both node and task failures are considered and the introduced delay is significantly decreased; thus the overall performance is improved.

In our mechanism, two kinds of files are introduced, namely, the local checkpoint file and the global index file. These files are created before the execution of one mapper task. The local checkpoint file is responsible for recording the progress of current task, avoiding reexecution from the beginning in case of task failure. If local task failure happened, one node can continue failed task using information in local checkpoint file. And the global index file is responsible for recording the characteristics of the current job, thereby helping in reconstructing the intermediate results across nodes to reduce reexecution time. The global index file can be implemented as a checkpoint file saved in HDFS and can be accessible in case of node failure.

The algorithm is divided into two parts: one part works on master node and the other works on worker nodes. In addition, as the master node is important, it is necessary to maintain multiple fully consistent "hot backup," to ensure seamless migration when a fault occurs. The two parts of algorithm are shown as follows.

The Algorithm on Master Node

(1) The master node preassigns mapper and reducer task to different worker nodes.

(2) Choose K replicas for each worker node.

(3) Wait for results from all worker nodes.

  (a) If all results are received, merge these results and mark job as completed.

  (b) Or go to 3 and keep waiting.

(4) Periodically send probing packets to all worker nodes.

  (a) If all worker nodes respond, then go to 4 and keep probing.

  (b) Or if one node does not respond in given time interval, then mark the node as failed.

    (i) Get the worker ID and all unfinished tasks on the node.

    (ii) Put all unfinished mapper tasks into global queue, and reschedule them on available replica nodes.

(iii) If there are failed tasks on the failed node, then reschedule these tasks on replica nodes which have intermediate results, without reexecuting these mapper tasks.

(c) If one node has finished all tasks, then reassign other unexecuted tasks to the node.

The Algorithm on Worker Node

(1) Check the type of the given task.

(a) If it is a mapper task, check whether it is a new task or a reexecution of failed task.

(i) If it is a new task, initialize and execute it.

(ii) If it is a reexecution of local failed task, then get its progress from the local checkpoint file and continue execution.

(iii) If it is a reexecution of the failed task from other nodes, then read global index file for the task and rapidly reconstruct intermediate result using information in global index file.

(b) If it is a reducer task, check whether it is a new task, a reexecution of the failed task, or unexecuted task from other nodes.

(i) If it is a new task, initialize and execute it.

(ii) If it is a reexecution of the failed task from other nodes, read intermediate data from the local disk and execute it.

(iii) If it is a new assigned unexecuted task from other nodes, then read intermediate data from a given worker node and execute it.

(2) Create a local checkpoint file and a global index file for the given mapper task.

(3) Start the mapper task.

(a) When memory buffer of the mapper node is full, dump intermediate data into local files. After dumping finished, record the position of the input stream and mapper ID (Position, mapper_ID) into local checkpoint file.

(b) According to the location distribution of input key-value pairs which contributes to output key-value pairs, two different strategies are employed to record these distributions in the global index file.

(i) For input key-value pairs producing output, record their position (T1, offset) into global index file, which means only pairs in these offsets need to be processed in reexecution. Here T1 means this is the type 1 record.

(ii) Or, for input pairs which give no output, record the range as (T2, offset1, offset2), which means the input pairs between offset1 and offset2 have no output and can be skipped in reexecution. T2 means this is the type 2 record.

(4) When a mapper task finished, shuffle and send the intermediate results to corresponding reducer nodes. Copy the intermediate data needed by local reducer task to replica nodes, then notify the completion of mapper task to master node, and delete the local checkpoint file and the global index file.

In our algorithm, when a task failure occurs, the corresponding node simply reads the checkpoint file saved in the local disk, restore task status to the checkpoint, and reload the intermediate results generated before failure, so reexecution is avoided.

When a node failure occurs, the scheduler on the master node is responsible for rescheduling the interrupted mapper tasks to available replica nodes, which can quickly construct the intermediate results of failed tasks using the global index file, to reduce the reexecution time greatly.

Note if tasks and nodes failed before checkpoint, the progress will continue from the recent checkpoint. And if the action of saving checkpoint failed, the progress will start from the recent checkpoint again. The simple failover strategy is of low cost and is effective in distributed computing environment. The frequency of saving checkpoint should be carefully chosen: a high frequency will provide lower cost for failover and higher checkpoint saving cost for normal running, while a low frequency is in the opposite case. After careful adjusting, the frequency value in our experiment is set to one checkpoint per 105 key-value pairs.

If node failure happens in the reducer stage, the tasks on the failed node are rescheduled to any available replica nodes. The needed intermediate results have been copied to the replica node when mapper tasks finished; thus there is no need to repeat the mapper tasks on the failed node, so that overall completion time of the MapReduce job is greatly reduced.

3.3. System Analysis. For a given mapper task m, the question is, for a given input range in an input split, how to properly place the kind of records (T_1 or T_2) in global index file, to minimize storage overhead. For the given input range, set the total number of key-value pairs as N_m and the number of output key-value pairs as N'_m. The size of type 1 and type 2 records is L_1 and L_2:

$$L_2 = 2L_1. \tag{2}$$

Set $S_{m,1}$ and $S_{m,2}$ as the storage overhead in type 1 and type 2 record in the input range. Set V_m as the count of input subranges which give no output, and the O_ith input key-value pair produces the ith output key-value pairs. Consider

$$V_m = \sum_{i=2}^{N'_m} \begin{cases} 0, & O_i - O_{i-1} = 1 \\ 1, & O_i - O_{i-1} > 1. \end{cases} \tag{3}$$

We have

$$S_{m,1} = L_1 N'_m$$
$$S_{m,2} = L_2 V_m = 2L_1 V_m. \quad (4)$$

Set R_m as the decision we make; when $R_m = 1$, we store type 1 record to global index file, or if $R_m = 2$, we use type 2 record. Then we can decide which kind of record to use according to equation as follows:

$$R_m = \begin{cases} 1, & N'_m < 2V_m \\ 2, & N'_m \geq 2V_m. \end{cases} \quad (5)$$

4. Experimental Results

We validate our algorithm on Hadoop cluster, which is implemented as a patch to Hadoop. The comparison is measured from the performance and the overhead aspect. The performance of the algorithm is evaluated by delay. The implementation of the algorithm is based on Hadoop 0.20.1, Java 1.6, and HDFS file system with data block size of 256 MB. The underlying infrastructure is a 20-node HP blade cluster and each node is equipped with quad-core Xeon 2.6 GHz CPU, 8 G memory, 320 G hard drive, and two pieces of Gigabit NICs. Each node is configured to hold four Xen virtual machines; thus it has 80 virtual nodes. And we schedule 40 nodes for Hadoop cluster with default fault tolerance mechanism and 40 nodes for cluster with our mechanism. For each cluster, one node is deployed as the master node and the remaining 39 nodes are deployed as the worker node. A single worker node can simultaneously run two mapper tasks and one reducer task. In the experiment we run a typical filter job, which is to find out certain entries from huge amounts of data. This kind of job is computationally intensive and has less intermediate results. We use 1.2 million English web pages with an average page size of 1 MB for test. By adjusting the split size, one mapper handles an average of approximately 120 M input data, and each node is assigned with an average of about 250 mapper tasks.

According to the distribution of query words in input data, there are three kinds of MapReduce jobs used in the experiment: the aggregated job, the sparse job, and the mixed job. In an aggregated job, the locations of query words are gathered in the target data; in a sparse job, the locations of the query words are more dispersed; in a hybrid job, the above two situations coexist.

The processing time of MapReduce job is usually influenced by factors as below: (1) the size of the data set, where each mapper task will deal with a subset of data (split); (2) the computation performance of worker nodes; (3) MapReduce job type; (4) the number of bad records in each split, where each record will lead to restarting of task; (5) the number of failed worker nodes. By adjusting these factors, the effectiveness and the overhead of given fault tolerance algorithms can be evaluated and compared in the same criterion.

In scenario with only task failure, the comparison of job completion time of two mechanisms is shown in Figure 2, with our mechanism marked as RFM (replication-based fault tolerance mechanism). The x-axis is the task error probability in the form of error tasks number per 100 tasks, with the y-axis being the total completion time. There is no upper limit on the errors number of each mapper task. It can be observed that, along with the increasing of probability of errors, the execution time of the job with REF is significantly decreased compared to that of default mechanism.

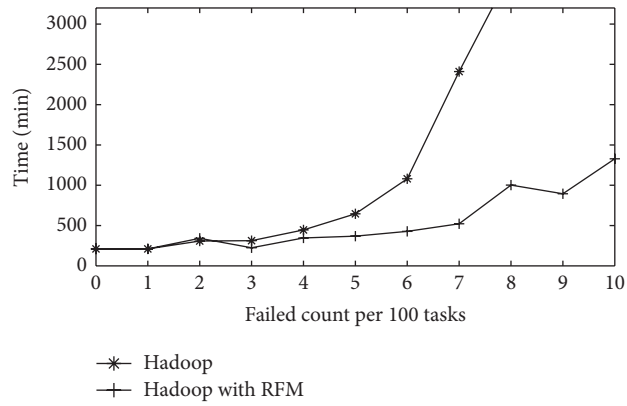

Figure 2: Comparison of execution time with task failure.

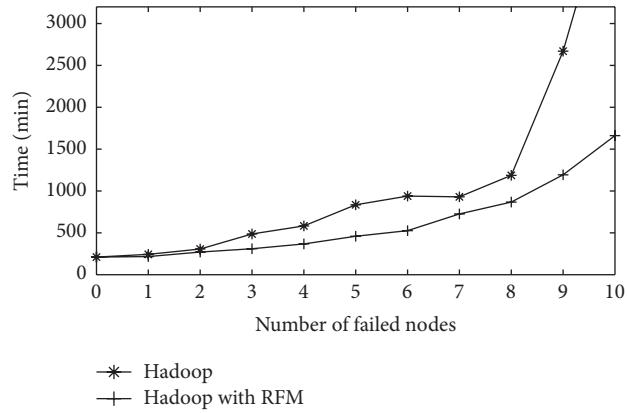

Figure 3: Comparison of execution time with node failure.

In scenario with only node failure, the comparison of MapReduce job completion times is shown in Figure 3. The x-axis is the number of failed nodes, with the y-axis being the total completion time. It can be observed that, along with more failed nodes, our mechanism can significantly decrease the reexecution time. It is because the default mechanism of Hadoop will start all failed task from the beginning on replica nodes, while our mechanism gets more tasks finished in the same period of time by continuing the mapper tasks quickly and efficiently.

In scenario with both task and node failure, the comparison of job completion times is shown in Figure 4. The x-axis is the probability of task failure and node failure, in the form of failed tasks/nodes count per 100 tasks/nodes, with y-axis as the total completion time. It is shown that, under the joint influence of task and node failure, the relation between the job execution time and the failure probability is nearly linear

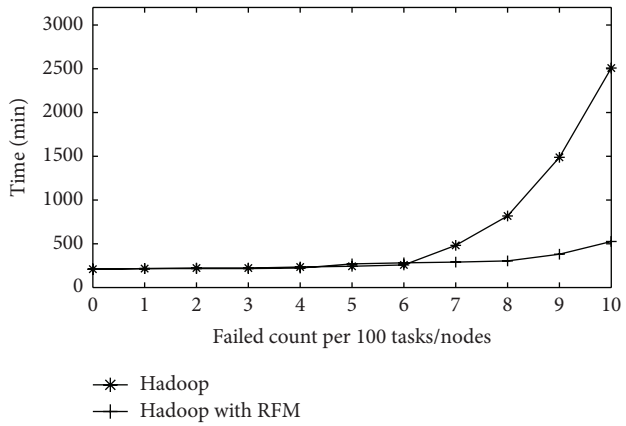

Figure 4: Comparison of execution time with task and node failure.

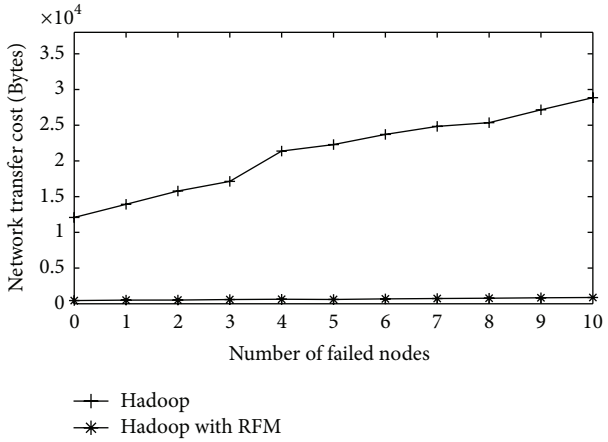

Figure 5: Comparison of average network overhead with node failure.

in our mechanism, and the overhead is mainly introduced by task migration and restoring.

Moreover, the overhead introduced by our mechanism is low. In case of task failure, the network overhead can be ignored since there is no extra network transferring. In case of node failure, the extra network overhead of our mechanism is mainly from the active replication of the global index file. Figure 5 shows the network overhead which is introduced by node failure. The x-axis is the number of failure nodes, and the y-axis is the average network overhead. It is shown that, compared with the network overhead of default mechanism in Hadoop, the network overhead of our mechanism is significantly low.

In the task failure scenario, the storage overhead is the size of the local checkpoint file, which is negligible due to the limited position information stored in it. Figure 6 shows the comparison of storage overhead of three different types of MapReduce job, under the scenario of 10 failed nodes. It is shown that the increased storage overhead of our mechanism is mainly from global index file, which is low compared with original intermediate results.

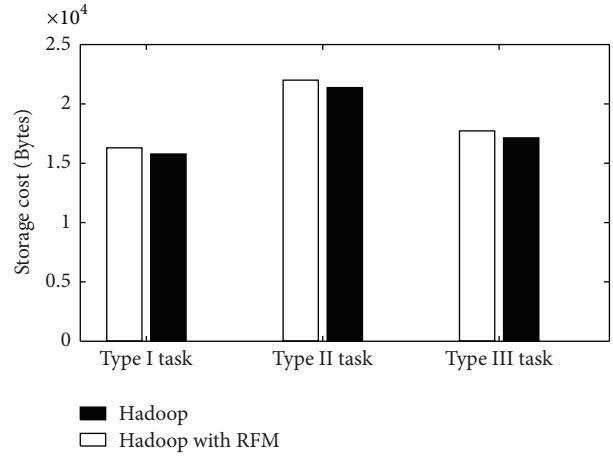

Figure 6: Comparison of average storage overhead with node failure.

5. Conclusions

The paper proposed a replication-based mechanism for fault tolerance in MapReduce framework, which is fully implemented and tested on Hadoop. Experimental results show the runtime performance can be improved by more than 30% in Hadoop; thus our mechanism is suitable for multiple types of MapReduce job and can greatly reduce the overall completion time under the condition of task and node failures.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This paper is supported by the National Natural and Science Foundation of China (nos. 61003052, 61103007), Natural Science Research Plan of the Education Department of Henan Province (nos. 2010A520008, 13A413001, and 14A520018), Henan Provincial Key Scientific and Technological Plan (no. 102102210025), Program for New Century Excellent Talents of Ministry of Education of China (no. NCET-12-0692), Natural Science Research Plan of Henan University of Technology (2014JCYJ04), and Doctor Foundation of Henan University of Technology (nos. 2012BS011, 2013BS003).

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processingon large clusters," in *Proceedings of the 6th Symposiumon Operating Systems Design & Implementation*, vol. 3, pp. 102–111, 2004.

[2] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan and Claypool Publishers, 2009.

[3] T. White, *Hadoop: The Definitive Guide*, O'Reilly, 2009.

[4] Apache Hadoop, http://hadoop.apache.org.

[5] D. Borthakur, The Hadoop Distributed File System: Architecture and Design, http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.

[6] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[7] B. Selic, "Fault tolerance techniques for distributed systems," http://zh.scribd.com/doc/37243421/Fault-Tolerance-Techniques-for-Distributed-Systems.

[8] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop high availability through metadata replication," in *Proceedings of the 1st International Workshop on Cloud Data Management (CloudDB '09)*, pp. 37–44, ACM, November 2009.

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

[10] Amazon Elastic Mapreduce, http://aws.amazon.com/elasticmapreduce/.

[11] J. Dean, "Experiences with MapReduce, an abstraction for large-scale computation," in *Proceedings of the Internet Conference on Parallel Architectures and Computation Techniques (PACT '06)*, vol. 8, pp. 5–10, Seattle, Wash, USA, 2006.

[12] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "On availability of intermediatedata in cloud computations," in *Proceedings of the The USENIX Workshop onHot Topics in Operating Systems (HotOS '09)*, vol. 5, pp. 32–38, Ascona, Switzerland, 2009.

[13] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "RAFTing MapReduce: fast recovery on the RAFT," in *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE '11)*, vol. 3, pp. 589–600, IEEE, Hannover, Germany, April 2011.

[14] J. Hui, Q. Kan, S. Xian-He, and L. Ying, "Performance under failures of mapreduce applications," in *IEEE/ACM International Symposium on Cluster Cloud and Grid Computing*, pp. 608–609, Newport Beach, Calif, USA, 2011.

[15] A. Martin, T. Knauth, S. Creutz et al., "Low-overhead fault tolerance for high-throughput data processing systems," in *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS '11)*, pp. 689–699, Minneapolis, Minn, USA, June 2011.

[16] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, vol. 1, pp. 181–192, Indianapolis, Ind, USA, June 2010.

[17] Q. Zheng, "Improving mapreduce fault tolerance in the cloud," in *Proceedings of the IEEE International Symposium on Parallel Distributed Processing Workshops and Phd Forum (IPDPSW '10)*, vol. 1, pp. 1–6, New York, NY, USA, 2010.

[18] J. Liu, X. G. Luo, B. N. Li, X. M. Zhang, and F. Zhang, "An intelligent job scheduling system for web service in cloud computing," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 11, no. 12, pp. 2956–2961, 2013.

[19] S. Hong, S.-p. Chen, J. Chen, and G. Kai, "Research and simulation of task scheduling algorithm in cloud computing," *Telkomnika*, vol. 11, no. 11, pp. 1923–1931, 2013.