Received: 15 May 2018 | Revised: 30 August 2019 | Accepted: 13 April 2020

DOI: 10.4218/etrij.2018-0265

ORIGINAL ARTICLE

ETRI Journal WILEY

Task failure resilience technique for improving the performance of MapReduce in Hadoop

Kavitha C[1] | **Anita X**[2]

[1]Department of Information and Communication Engineering, Anna University, Chennai, India

[2]Department of Computer Science and Engineering, Jerusalem College of Engineering, Chennai, India

Correspondence
Kavitha C, Research Scholar, Department of Information and Communication Engineering, Anna University, Chennai, India.
Email: kavitha4cse@gmail.com

MapReduce is a framework that can process huge datasets in parallel and distributed computing environments. However, a single machine failure during the runtime of MapReduce tasks can increase completion time by 50%. MapReduce handles task failures by restarting the failed task and re-computing all input data from scratch, regardless of how much data had already been processed. To solve this issue, we need the computed key-value pairs to persist in a storage system to avoid re-computing them during the restarting process. In this paper, the task failure resilience (TFR) technique is proposed, which allows the execution of a failed task to continue from the point it was interrupted without having to redo all the work. Amazon ElastiCache for Redis is used as a non-volatile cache for the key-value pairs. We measured the performance of TFR by running different Hadoop benchmarking suites. TFR was implemented using the Hadoop software framework, and the experimental results showed significant performance improvements when compared with the performance of the default Hadoop implementation.

KEYWORDS

Hadoop, in-memory, key-value pair, MapReduce, recovery, Redis cache, resilience, task failure

1 | INTRODUCTION

MapReduce is a framework for processing huge datasets in parallel and distributed computing environments [1]. It adopts a centralized architecture; one node acts as a master and all other nodes serve as workers [2]. The master node schedules the tasks, while the workers are responsible for performing the execution of the map and reduce tasks.

Fault tolerance is one of the most critical issues for MapReduce [3]. During the execution of a large volume of data, failure is a serious issue [4,5]. Some of the faults in a workflow environment include network failures, node crashes, memory leak, disk failures, out-of-disk space, and task failures [6]. In a MapReduce cluster, master failures are tolerated by setting up an effective standby master. Failed tasks are automatically rescheduled from scratch, which significantly increases task completion time. Suppose that the total execution time of task i on node j is T_{ij}. If task i encounters a failure at time t, then $t \ll T_{ij}$ means that the failed job has been redone effectively. If a failure occurred at the time when all jobs had been done, then $t \gg T_{ij}$; that is, redoing the failed job is computationally expensive as most of the jobs have already been done [7]. Development of an efficient fault tolerance mechanism in the MapReduce environment is an active research area. Moreover, current works rely on external storage facilities for storing the computed key-value pairs. However, this could increase the task latency because of the slower access time of external storage disks.

In this paper, we propose the TFR technique, which allows the MapReduce execution to continue from the state where the failure has occurred by recovering the processed bytes from the external source. The external source used here is Amazon ElastiCache for Redis. Compared with other in-memory data structure stores, Redis makes it very simple to manipulate complex data structures.

The main contributions of this paper are listed as follows:

- We modified the original MapReduce workflow by implementing the TFR algorithm in the standard map and reduce code, which makes it possible to implement fault tolerance.
- The following changes were done in the standard map and reduce code:
  a. A Hadoop application is integrated with Amazon ElastiCache for Redis.
  b. Whenever a mapper generates an intermediate key-value pair, the pair is sent along with the metadata to an external source.
  c. During the execution of the shuffle and sort phases in the reducer node, the generated key-value pairs are sent to the external source. Moreover, the mapper fetches the results of the shuffle and sort phases from the external source and executes the reduce tasks.
  d. If the execution fails, a retry attempt will skip the execution of the accomplished data recorded in the external source.
- We evaluated the proposed Hadoop framework using the HiBench Benchmarking Suite.

TFR is implemented using Hadoop, an Apache open-source software framework built to support data-intensive applications running on large commodity clusters. The Hadoop framework is designed mainly for the parallel and distributed processing of massive data residing in a cluster of commodity servers. To create a connection between Hadoop and Amazon ElastiCache for Redis, TFR uses the Jedis client code, which is a client library written in Java for Redis. TFR sends a request to the proxy server to store and retrieve data to and from the Redis cache. The Hadoop source code was modified by implementing TFR for the map phase and reduce phase.

The HiBench Benchmark Suite was used to analyze the behavior and performance improvement of TFR. This benchmark suit was used to perform a thorough comparison between the default Hadoop implementation and our proposed Hadoop application in terms of execution time. The benchmarks used in the experiment were WordCount, Sort, TeraSort, and PageRank. The experimental results showed significant performance improvements with TFR when compared with the performance of the default Hadoop implementation. The experimental data help in tuning the MapReduce applications.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 illustrates the basic execution flow of MapReduce. Section 4 describes the workings of Amazon ElastiCache for Redis. Section 5 presents the design and implementation of TFR. Section 6 discusses the results of the performance evaluation. Finally, Section 7 concludes this paper and describes the future work.

2 | RELATED WORKS

Many research studies have been conducted to improve the performance of MapReduce execution in different aspects. Some studies focused on implementing the job scheduling algorithms to increase the execution time of MapReduce tasks, while others focused on improving the execution of MapReduce from task failures by utilizing any fault-tolerance strategy. Moreover, some studies concentrated on optimizing the configuration settings to improve the execution flow of the MapReduce framework.

A resilient map task that uses checkpointing tactics was introduced in [3] to make a small change in the original MapReduce execution workflow and to gain a finer grained fault tolerance. A mapper informs the master node with the metadata of the spilled files that include the task ID, task retry ID, input task range, host location, and task size. The execution flow of this technique allows the reducers to shuffle the spilled files of a task from different task retry attempts. The technique used here creates a Java cache using HashMap, which requires main memory and creates more overhead.

According to a recent study [8], a single job failure can result in a 50% increase in total execution time. Moreover, applications may fail for a variety of reasons that we cannot count. To understand how an application will behave during faults in greater detail, we need to categorize the faults by using abstract models. These models help us to tolerate the faults. The Byzantine fault-tolerant model in [9] provides a fault tolerance framework to the Hadoop system. A simplistic solution is proposed here by executing the job more than once using the original Hadoop application. The map and reduce task is re-executed until the fault limit + 1 output match. The application executes the tasks many times to detect the arbitrary faults. The completion time of the task execution is sensitive to a slow-running task [10], as only one slow task is enough to cause a serious delay in the whole job completion. Hadoop tries to detect slow-running tasks and launches a checkpoint from them. The progress score of each task is monitored to determine the slow-running task. The progress score is simply the fraction of the key-value pairs for the map tasks and the completion of the copy, sort, and reduce phases for the reduce task.

Lin [11] proposed a library-level checkpointing approach that uses a library to create checkpoints. A drawback of this approach is that checkpoints cannot be created for certain shell scripts and system calls as the system files cannot be accessed by a library. This proposed approach is implemented in message passing interface (MPI)-based MapReduce data computing applications.

Quiané-Ruiz and others [12] proposed the RAFTing MapReduce, which piggybacks checkpoints on the task progress computation. A local checkpointing algorithm that allows a map task to store the metadata of the task progress on a local disk has been implemented. This work tries to create query metadata checkpointing to keep track of the mapping between the input data and the intermediate data.

Gu and others [13] proposed the SHadoop, an optimized version of Hadoop to improve the MapReduce performance in Hadoop clusters. SHadoop aims at improving the internal execution time of short jobs. The authors experimentally evaluated the scalability of SHadoop compared with that of the original version of Hadoop by scaling the data with respect to the nodes and by scaling the number of nodes with respect to the data.

The Hadoop++ method in [14] applies indexing on Hadoop without changing the original source code. The partition and indexing algorithm is added on top of the Hadoop interfaces. The authors built a new distributed database called HadoopDB, which is designed to utilize the fault-tolerant ability. The authors claimed that Hadoop++ runs 20 times faster than the original Hadoop. However, they failed to incorporate the Hadoop fault tolerant strategy and, thus, their method suffers from similar failure issues to those of the original version of Hadoop. As each type of these fault tolerance strategies and optimizations only pertains to a certain type of application, they lack general applicability. Our fault tolerance strategy is a generalized approach to improve the execution performance of MapReduce tasks.

3 | THE MAPREDUCE FRAMEWORK

MapReduce is a programming framework based on two fundamental pieces of code: a map function and a reduce function. It is capable of processing an enormous amount of data in parallel. The MapReduce model works in a master-slave architecture. In the map step [15], the master node takes a large problem input, divides it into smaller subproblems, and then distributes them to worker nodes. The worker nodes execute the tasks and handle the data movement between the mappers and reducers.

MapReduce tasks take key-value pairs as input. Typically, the input and output data are stored in the Hadoop Distributed File System (HDFS). The InputFormat class in MapReduce defines how these input files are to be split and read. It creates the InputSplit for MapReduce tasks. InputSplit is a logical representation of data. The input splits are divided into records, which are processed by the mappers. Each split has one map task. The number of maps is handled by the number of blocks in the input data, and the number of maps is determined by the InputFormat as follows:

$$\text{No\_of\_mappers} = \frac{\text{Input\_data\_size}}{\text{Split\_size}}. \tag{1}$$

Here, Split_size is based on the HDFS block size. The default size of an HDFS block is 64 MB and can be extended to 128 MB. In Hadoop, the user can define the Split_size and can control it based on the Input_data_size by setting the mapred.max.split.size property during job execution. Consider a block size of 100 MB and expect 1 TB of input data; then, the number of maps is calculated as

$$1\,\text{TB} = 1\,000\,000\,\text{MB},$$

$$\text{No\_of\_mappers} = \frac{1\,\text{TB}}{100\,\text{MB}} = \frac{1\,000\,000}{100} = 10\,000\,\text{mappers}. \tag{2}$$

The performance of the data extraction task is scaled by having many mappers running in parallel. Equation (2) shows that there are 10 000 input splits, and it can spawn 10 000 map jobs in total for an input size of 1 TB. These 10 000 map jobs are launched in parallel to convert the input splits into intermediate key-value pairs. Huge chunks of intermediate data will be produced by each of the mappers. The stream of these intermediate data generated by each mapper is buffered in memory and periodically stored on the local disk of the mapper. This output record is summarized before it is passed to the reducer. There are two phases in the reduce function: shuffle and sort. The output from the mappers is sorted and spawned as input to the reducers. The reducer takes the outputs of the mappers by contacting every mapper via a remote procedure call and processes them to produce the final result, which is stored in the HDFS. Figure 1 shows the execution flow of MapReduce, which is a framework for pa.

The execution flow of MapReduce is explained as follows:

1. The Hadoop job client submits the job copies and the jar files to the HDFS. The Java MR API is the JobClient class, which acts as an interface for the user job to interact with the cluster. Hadoop splits the input data into chunks, and the number of mappers is calculated using (1).
2. Each mapper takes one split at a time and is executed in parallel.
3. The input data are processed by these mappers, which generate the intermediate key-value pairs.
4. The mappers store their output in an in-memory buffer of about 100 MB by default. When the buffer reaches a

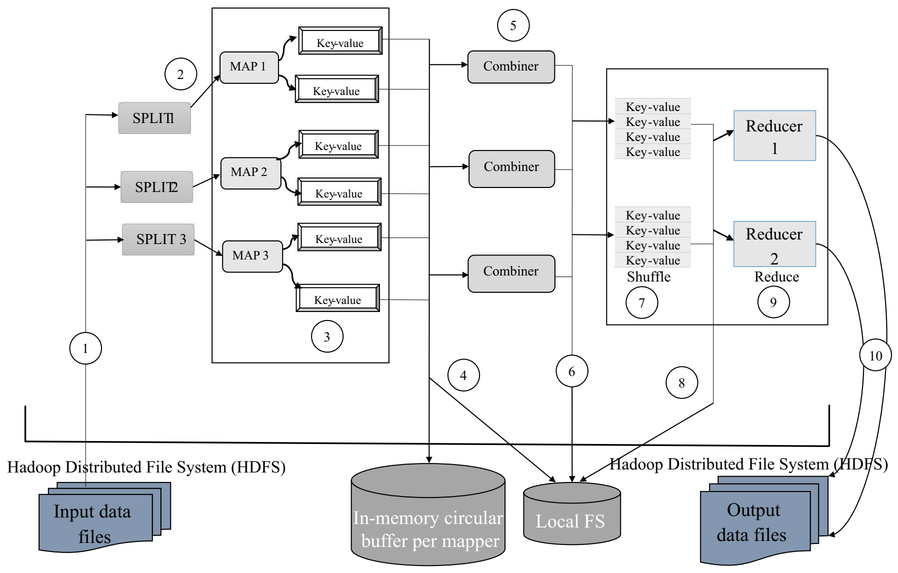

FIGURE 1 MapReduce execution flow [Colour figure can be viewed at wileyonlinelibrary.com]

certain threshold limit, the contents are spilled to the local disk of a machine on which the map task is running. Every time a buffer reaches the spill threshold, a new spill file is created. There are thus many spill files created as a result.

5. If the combiner function is enabled before the map outputs are written to the disk, all intermediate key-value pairs per mapper are merged into one output file.
6. The results of the combiner function are written to the local disk. All the output files of a map task are collected over the network and sent to the reducer nodes.
7. The input data to the reducer are shuffled on the reducer node. Then, they are sorted and grouped together by key.
8. The shuffled data are written to the local disk.
9. The output data from the shuffle and sort phases are provided as input to the reducers.
10. The output from the reducer nodes is written to the HDFS.

In the Hadoop pipeline framework, when a system failure occurs, the whole process of the above MapReduce execution flow is computed again. Even after the system failure, these intermediate key-value pairs are fed to another cluster of reducers. When a failure occurs in a task, the corresponding task is rescheduled to other reliable nodes, which start the execution from scratch after the recovery. The TFR technique is proposed to overcome this problem by eliminating the need to restart a failed task from scratch.

4 | AMAZON ELASTICACHE FOR REDIS

Redis is an open-source in-memory key-value store for use as a cache. It eliminates the need to access a disk. Using Amazon ElastiCache for Redis, we can add an in-memory layer to the Hadoop application design. A Redis cluster is created with six shards and 42.84 GB of memory using the AWS Management Console. Amazon ElastiCache supports high availability using Redis replication. The key-value pair is partitioned across six shards, and each shard consists of one read/write primary node and two read-only replica nodes. Each of these read replicas keeps a copy of the key-value pair from the primary shards. A Hadoop application writes only to the primary nodes. Read scalability is enhanced through the read replicas and protects against data loss.

We have enabled the automatic failover functionality of Amazon ElastiCache on our Redis cluster. To improve the fault tolerance, we provision the primary node and read replicas in multiple availability zones (multi-AZ). Consider the case of a Redis replication group with a primary node in AZ-a and read replicas in AZ-b and AZ-c. If the primary node fails, the read replica is promoted as the primary cluster. Amazon ElastiCache for Redis creates a new replica in Az-a. When the entire cluster fails, all the nodes in the failed cluster are recreated and provisioned in the same AZ as that of the original nodes. The data in the primary node and read replicas can be backed up if any failure occurs.

Partitioning the heavy load of key-value pairs over a greater number of Redis nodes reduces the access bottlenecks. Proxy-assisted partitioning is implemented, which allows sending requests to a proxy that speaks to the Redis instance. Redis with a cluster mode-enabled state has a configuration end point that knows all the primary nodes and the end points of the nodes in the cluster. When a Hadoop application writes or reads the key-value pairs, Redis can determine which shard the key belongs to and which end point to use in that shard. A Redis cluster has 16,384 hash slots. To map keys to hash slots, Redis computes the hash slot of a key using the following formula: CRC16 (key) % 16 384, where "%" is the modulus operator. Every shard in a cluster is responsible for a subset of hash slots.

- Shard s1 = slots 0–2730
- Shard s2 = slots 2731–5461
- Shard s3 = slots 5462–8192
- Shard s4 = slots 8193–10 923
- Shard s5 = slots 10 924–13 654
- Shard s6 = slots 13 655–16 383

Redis can flush the cache in the background using the redis-cli FLUSH command after the corresponding map-reduce task is completed.

5 | IMPLEMENTATION OF TFR

To resolve the above mentioned issues in Sections 1 and 2, we designed a new MapReduce prototype system and implemented it on the basis of the Amazon ElastiCache for Redis for a faster recovery during the task failures. In this section, the conceptual design of TFR is outlined. In the original version of Hadoop, if a task execution fails, the whole task will be executed again. This is because the MapReduce framework does not keep track of the task progress after a task failure. The main goal of TFR is to recover the processed bytes at a faster rate to continue the execution from the state where the failure has occurred. In [16], the intermediate data from the map and reduce tasks are stored sequentially in files. To manage these intermediate files, the authors implemented a distributed message management system that aggregates the messages effectively.

A natural solution to recover the processed intermediate data during a task failure is to save the ongoing computation to some stable storage. Traditionally, for the storage of a massive amount of data, companies have two choices: vertical and horizontal scaling [17]. Vertical scaling involves adding more RAM modules and CPUs to a single large machine. Horizontal scaling involves splitting the data into shards and storing them over multiple machines in a distributed manner. Data storage using cloud computing is a conceivable option for many small to huge business organizations that use big data technology.

We need to leverage cloud computing solutions to address big data problems. Amazon ElastiCache for Redis [18] easily deploys a cache environment that accelerates the high-volume application workload. It caches the data and provides data retrieval in submilliseconds. The two well-known distributed memory caching systems are Memcached and the Redis protocol-compliant cache engine software. We created a Hadoop application that uses Amazon ElastiCache for Redis to recover all the processed data. It lightens the burden associated with heavy request loads and increases the overall performance. There are two main implementation techniques available in the Redis ElastiCache data store [19].

- A client application needs to select the right Redis instance to read or write the data.
- A client application should send the request to a proxy server that can communicate using the Redis protocol. This protocol, in turn, sends the request to the right Redis node.

TFR substitutes Amazon ElastiCache for Redis as the in-memory key-value store for both the in-memory circular buffer of the mapper and the local file system. The input of the map task and the output of the reduce task reside in the HDFS. TFR utilizes Amazon ElastiCache for Redis only for storing the intermediate key-value pairs. TFR is fast because it requires only a few sub-milliseconds to collect all the saved data required. When a mapper creates the intermediate key-value pairs, which are quite large, TFR stores them along with their corresponding timestamp in the Redis cache. Several changes are required to utilize TFR.

Figure 2 shows the execution flow of the TFR MapReduce workflow.

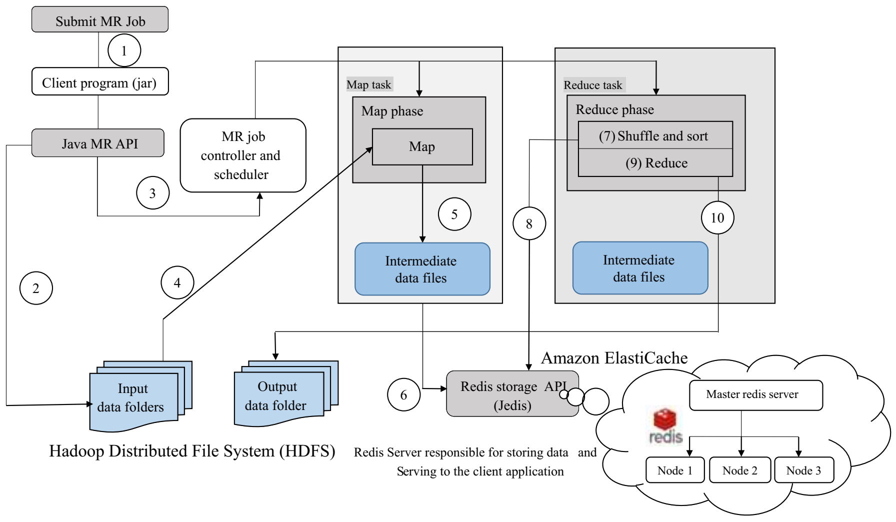

FIGURE 2 Proposed MapReduce execution flow

1. A client submits a job with the Hadoop jar command and copies the jar files to the HDFS.
2. The Java MR API is the JobClient class, which acts as an interface for the user job to interact with the cluster. The class creates tasks on the basis of the file splits (blocks).
3. It submits the task to the MR job controller, which is the application master that assigns the job to the mappers and reducers.
4. The map phase is executed by reading the data from the HDFS.
5. Each map task generates the intermediate key-value pairs.
6. The results of the mapper are stored in the Redis cache using the Jedis client API.
7. The results of the map task are shuffled and sorted in the reducer node by retrieving them from the Redis cache.
8. The output data from the shuffle and sort phases are stored in the Redis cache through the Jedis client code.
9. These output data can be retrieved from the Redis cache to provide as input to the reducers.
10. The final reduced results are written to the HDFS.

We save the ongoing computation of the map and reduce phases in the Redis cache as follows:

1. We download the Jedis client, which contains all the logic for connecting to the nodes of Amazon ElastiCache for Redis.
2. The MapReduce program of the Hadoop application is modified so that it periodically updates the total processed bytes and the timestamp to the Redis cache by making a request to the proxy server. Read and write operations are done through this request. Redis can handle up to a million requests per second per cache node. Using this auto discovery, the Hadoop application program connects to all the nodes in the cluster without any intervention.

TFR is implemented by modifying the source code of the original version of Hadoop. We modified the following classes: OutputFormat, RecordWriter, Mapper, Reducer, and Driver. All these classes were modified to enable them to recover the processed data to the Redis cache and to create a Jedis connection with Hadoop.

OutputFormat code: The OutputFormat class was modified to enable it to establish and verify the input job configuration and to take a list of Redis instance nodes as a CSV structure and a Redis hash key to write all the output.

RecordWriter: We modified this class to enable it to write out the data to the Redis cache and to handle the connection to the Redis server via the Jedis client. The intermediate key-value pairs are written to the Redis instance by making a request to a proxy server and provide an even distribution of all key-value pairs across all Redis nodes. A constructor is created to store the hash key to write to the Redis instance.

Mapper code: This class was modified to enable it to store the generated key-value pairs along with the timestamp of the Redis instance by making a request to the proxy server.

Reducer code: This class was modified to enable it to retrieve the mapper results from the right Redis instance and to store the shuffled and sorted results to the Redis instance by making a request to the proxy server.

Driver class: This code was modified to enable it to write out the data to the Redis cache.

5.1 | TFR for the map phase

Algorithm 1 outlines the pseudo-code of TFR for recovering the intermediate key-value pairs from the map task failures. Each mapper is executed in parallel and stores the output as a key-value pair (see lines 1-4). TFR skips the storing of the intermediate data to a file system and sends them directly to the external source. The total processed bytes from each mapper are periodically updated in the Redis cache using the Jedis client. We first connect to the Redis server through the Jedis Java code. Lines 5 and 6 in Algorithm 1 are used to obtain the Jedis connection pool. Once the Jedis connection is established, it saves the data to the Redis cache (see lines 7 and 8).

Algorithm 1 TFR for the map phase

```
Input: (Key, Value)
Begin
[1]: for each m in mapper
[2]:   Execute maptask (Object Key, Text Value)
[3]:         /*user-defined code to run the mapper class*/
[4]:           context.writeMapOutput (key1,    value1);
[5]:           Jedis jedis = new jedis ();
[6]:       jedis. connect ();
[7]:             writeMapOutput (Text key1, Text
               value1):
[8]:             jedis.set (hashKey, timestamp,    key1.toString
             (), value1.toString ());
[9]:   if fail.isMapper ()
[10]:        jedis.get (hashKey, timestamp,   key1.toString (),
               value1.toString ());
End
```

In this work, the total processed bytes along with the hashKey and timestamp are stored in the Redis cache by sending a request to a proxy server, which then sends the request directly to the right Redis instance. Similar to the Java hash map, the Redis hash map is a key between the string keys and string values. The intermediate key-value pairs from each mapper are distributed equally across all the Redis instance. Hence, hashKey can be used as a key to identify the right key-value pair of a corresponding mapper. Redis offers a timestamp facility to record the CPU statistics, event logs, and times of occurrences of a particular event. The < timestamp, key, value > triplet denotes the observation recorded at a point in time. Generally, data arrive in an increasing timestamp order. Data are read from the Redis cache by specifying the time window. When a failure occurs, the execution begins from the state where the failure has occurred. Redis saves the database needed to restore to that state and provides a much more recent copy of the saved data using the timestamp to continue the execution by skipping it from the already processed bytes (see line 10 in Algorithm 1).

5.2 | TFR for the reduce phase

Algorithm 2 outlines the pseudo-code of TFR for recovering the processed bytes from the failed reduce task. The reduce phase occurs after the map phase.

Algorithm 2 TFR for the reduce phase

```
Input: (K1, V1)
Output: (K3, V3)
Begin
[1]: Execute shufflesort (Reducer reducerID,
        MapOutputCollector<Key1, Value1>)
[2]:  Jedis jedis = new jedis ();
[3]:  jedis. connect ();
[4]:        jedis.get (hashKey, timestamp,  key1.toString (),
        value1.toString ());
[5]:    scheduleshuffle= new shuffle<Key1,
        Value1>(reducerID,MapOutputCollector, hashKey,
        key1, value1);
[6]:  /*user-defined code to run the shuffle class*/
[7]:      context.writeShuffleOutput (key2, value2);
[8]:      writeshuffleOutput (Text key2, Text value2):
[9]:        jedis.set (hashKey, timestamp, key2.toString  (),
        value2.toString  ());
[10]: if fail.isShuffle_SortPhaseFailure ()
[11]:     jedis.get (hashKey, timestamp, key2.toString (),
        value2.toString ());
[12]: Execute reduceTask (Object Key, Text  Value)
[13]:           /*user-defined code for executing  the  reducer
        class*/
[14]:  context.writeReducerOutput (key3, value3);
[15]:           write the reducer output to the HDFS as key-
                value pairs
[16]: if fail.isreduceTask ()
[17]:          jedis.get (hashKey, timestamp,   key3.toString
        (), value3.toString ());

End
```

First, we fetch the map output bytes from the Redis cache (see line 4 in Algorithm 2). In the shuffle and sort phases, data from the mappers are grouped and sorted together by key. The intermediate data from the mappers are transferred to one or more reducers. Then, the TFR writes the shuffled and sorted key-value pairs to the Redis cache (see lines 7 and 8). The sorted output is given as input to the reducer nodes. If the failure occurs during the execution of the shuffle phase, the last saved bytes of the failed shuffled task are retrieved using the Jedis "get" method by attaching the timestamp to retrieve the data recorded at the time of occurrence of that event (see lines 10 and 11). Each reducer node obtains all the values associated with the same key. The result of the reducer is stored in the HDFS.

6 | PERFORMANCE EVALUATION

In this section, we analyze the performance of TFR. Hadoop was used as the baseline and our prototype of TFR was based on Hadoop 2.6.5. In this work, we used Hadoop 2.6.5 version to apply TFR on top of the original Hadoop interfaces. TFR can be implemented in any higher version of Hadoop to resolve the task failures efficiently. Most of the previous studies have implemented the fault tolerance strategy in Hadoop 1.x versions [2,9,12,13]. For a cluster setup, we ran our experiments on a 10-node cluster, where we dedicated one node to act as a master node and all other nodes to run as slave nodes in a virtual environment. We used Proxmox, which is an open-source virtual environment that is based on the Debian Linux distribution. Each node has an Intel processor clocked at 2.40 GHz, and it is equipped with 1.5 GB RAM and 200 GB HDDs. We deployed Redis 2.8.22 on Amazon ElastiCache. The Redis API such as the Jedis code was used as an interface between the Hadoop application and Amazon ElastiCache for Redis to store and retrieve data from the Hadoop application to the Redis cache and vice versa. The capacity of the Redis cache was 42.84 GB, and the node type was m4.xlarge (for reference, see Table 1). We improved the performance of the Hadoop application by performing a faster recovery of the intermediate data from this high-throughput and low-latency in-memory data store.

We performed a number of experiments separately to evaluate Algorithms 1 and 2. The HiBench Benchmarking Suite was used to assess the performance of the Hadoop framework in terms of execution time for MapReduce using the WordCount, Sort, TeraSort, and PageRank benchmarks. The said benchmark suite was installed and configured in all nodes. We ran our experiment by varying the sizes of the datasets. Table 1 shows the benchmarking environment used to test the performance of TFR. The experiments were designed and tested to measure the following aspects:

- The performance of TFR in terms of execution time.
- The performance of TFR in terms of throughput and latency in recovering the data from Amazon ElastiCache for Redis.

We ran different benchmark programs several times on Hadoop, and the measurement results were averaged. Each experiment was repeated five times and the testbed conditions were fixed to ensure the low variability and reproducibility of our results. For each data size, the WordCount, Sort, TeraSort, and PageRank benchmarks were executed five times. We calculated the coefficient of variation (CV) for the execution times e based on the standard deviation and mean,

$$\text{Coefficient of variation CV} = \frac{\text{Standard deviation}}{\text{mean}} = \frac{\sigma}{\mu}, \quad (3)$$

$$\text{Standard deviation } \sigma = \sqrt{\frac{1}{n}\left[\sum_{x=1}^{n} e_x - \mu^2\right]}, \quad (4)$$

$$\text{Mean } \mu = \frac{1}{n}\sum_{x=1}^{n} e_x. \quad (5)$$

Figure 3A shows the execution time of WordCount with an input size ranging from 5 GB to 25 GB in both TFR and Hadoop 2.6.5. With the increasing sizes of the datasets from 5 GB to 25 GB, the execution time significantly increased from 150 s to 1,620 s for TFR. Table 2 summarizes the performance improvement rate of TFR over that of Hadoop 2.6.5 on running WordCount. We quantified the measurement of the dispersion of a set of execution times. A low standard deviation value indicates that the set of execution times measured from repeating the experiments five times tended to be close to the mean. Figures 3B, 4B, 5B, and 6B show the standard deviation of Hadoop 2.6.5 and the execution time of TFR for the WordCount, Sort, TeraSort, and PageRank benchmarks. Figures 4A, 5A, and 6A show the performance comparison results for running the Sort, TeraSort, and PageRank benchmarks on Hadoop 2.6.5 and TFR. Tables 3, 4, and 5 summarize the performance improvement rate of TFR over that of Hadoop 2.6.5 on running the Sort, TeraSort and PageRank benchmarks. From the experiment results, we can see that the performance improvements varied for different benchmarks and that TFR performed better than the original Hadoop. Figure 7 shows the running time of the map, shuffle, and reduce phases for the WordCount, Sort, TeraSort, and PageRank benchmarks. The execution time of the task can be tracked by starting the Job History Server Web UI, which contains the information for each job such as the total run time and the run time of each phase. MapReduce often faces failures under various conditions. TFR is executed in a controlled testing environment with the injection of known faults. Errors and exceptions are added to the TFR application logic to achieve the fault tolerance of the system. It is a fault injection technique, where the source code of Hadoop is modified to inject simulated faults into a system. The error states are observed and termed as failures.

TABLE 1 Hardware and software configurations

No. of Nodes	10
CPU	Intel Xeon 12 cores
No. of Cores per CPU	1
No. of CPUs per node	1
Memory	16 GB
RAM	DDR4
Hard drive	2 TB Seagate Barracuda
Network	2 Gigabit Ethernet NIC
Operating System	Linux 5.2
JVM	JDK 2.0
Hadoop Version	Hadoop 2.6.5
Redis	Redis 2.8.22

Whenever the map output file is shuffled by the reducers, if there is not enough memory left, the file cannot be shuffled to the memory buffer and, instead, a local disk file will be created. If a task failure occurs, it is necessary to restart the failing task and recompute all the input data from scratch, which increases the overall execution time of the job. TFR allows the execution of a restarted task to continue from the point it was interrupted, without having to redo all the work from scratch. As in [1], the map output buffer size is adjusted to increase the performance of the MapReduce execution. This feature is completely avoided in TFR because of its use of the Amazon ElastiCache for Redis. Figure 8 shows the performance comparison results between TFR and Hadoop 2.6.5 under both failure and non-failure conditions for the WordCount benchmark, and these were obtained by combining Algorithms 1 and 2. We observed that the execution time of an input job for TFR with failures was slightly higher than that for TFR with no failures. Hadoop 2.6.5 performed badly under failures and without failures.

In recent years, several techniques have been developed to improve the performance of the MapReduce workflow. Among those techniques, some are specific to a particular type of applications such as reduce, skew, join, combiner, group, and aggregate [20–23], which will introduce an extra overhead and a negative impact on the computing cluster. The MapReduce workflow in TFR runs faster under the failure conditions because of the faster recovery of the intermediate data. We observed that Redis storage is much faster than the main memory and hard drives.

We examined the throughput and latency of our proposed TFR technique in retrieving the required intermediate data from the Amazon ElastiCache for Redis through the Jedis client API. To test this, we needed the Redis benchmark utility. Figure 9 shows the performance result of Redis on the GET/SET operations. This performance was evaluated using the Redis benchmarking tool [24]. The SET and GET commands were used to store and retrieve, respectively, the intermediate data between the Hadoop application and Amazon ElastiCache for Redis through the Jedis client request. During the executions of the map and reduce phases, the results of the map and shuffle tasks were written to the Redis cache.

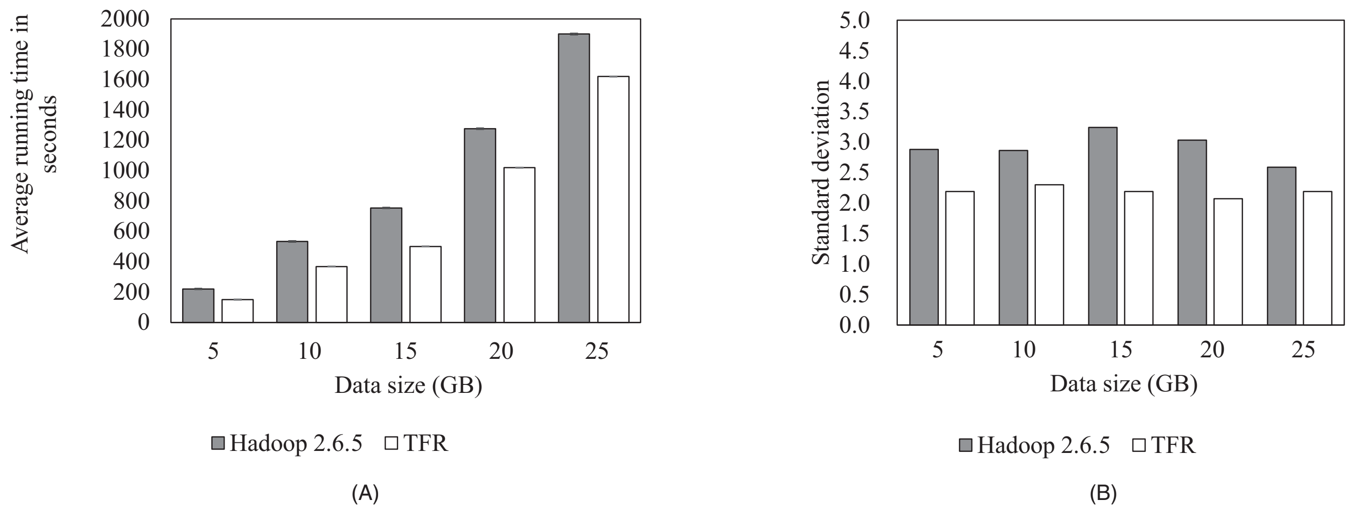

FIGURE 3 (A) Running times of the WordCount benchmark on Hadoop 2.6.5 and TFR. (B) Quantification measure of the standard deviation of Hadoop 2.6.5 and the execution time of TFR for the WordCount benchmark

TABLE 2 Performance improvement rate of TFR over that of Hadoop 2.6.5 on running the WordCount benchmark

	5 GB		10 GB		15 GB		20 GB		25 GB	
Input	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**
Average execution time (s)	220	150	533	368	754	500	1276	1019	1900	1620
Standard deviation	2.88	2.19	2.86	2.30	3.24	2.19	3.033	2.07	2.58	2.19
CV (%)	1.311	1.45	0.53	0.62	0.429	0.43	0.23	0.20	0.136	0.13
Improvement rate (%)	31.82		30.96		33.69		20.14		14.74	

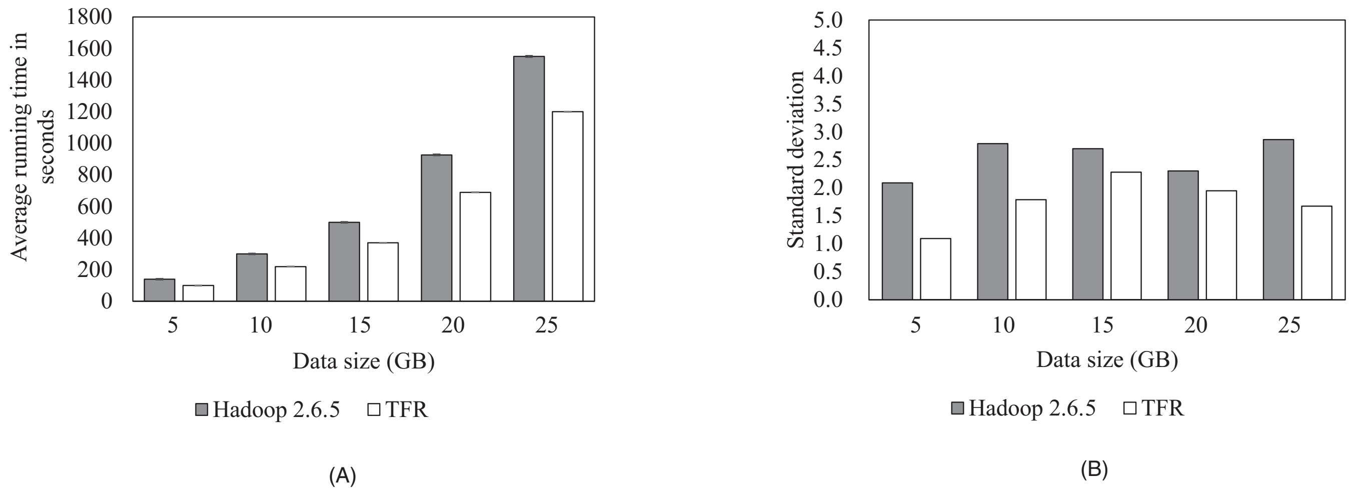

FIGURE 4 (A) Running times of the Sort benchmark on Hadoop 2.6.5 and TFR. (B) Quantification measure of the standard deviation of Hadoop 2.6.5 and the execution time of TFR for the Sort benchmark

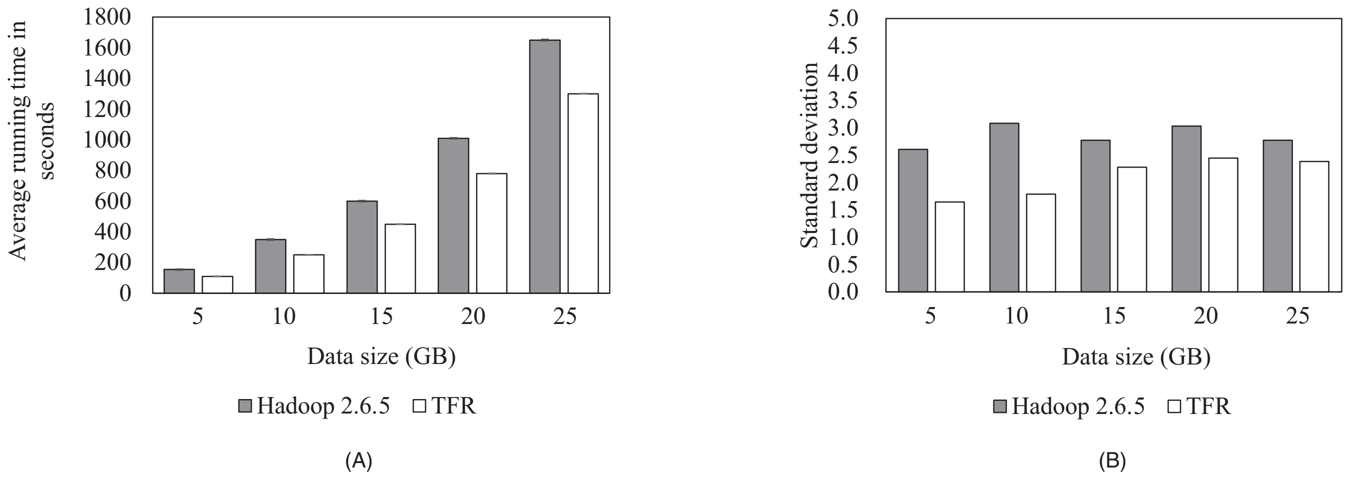

FIGURE 5 (A) Running times of the TeraSort benchmark on Hadoop 2.6.5 and TFR. (B) Quantification measure of the standard deviation of Hadoop 2.6.5 and the execution time of TFR for the TeraSort benchmark

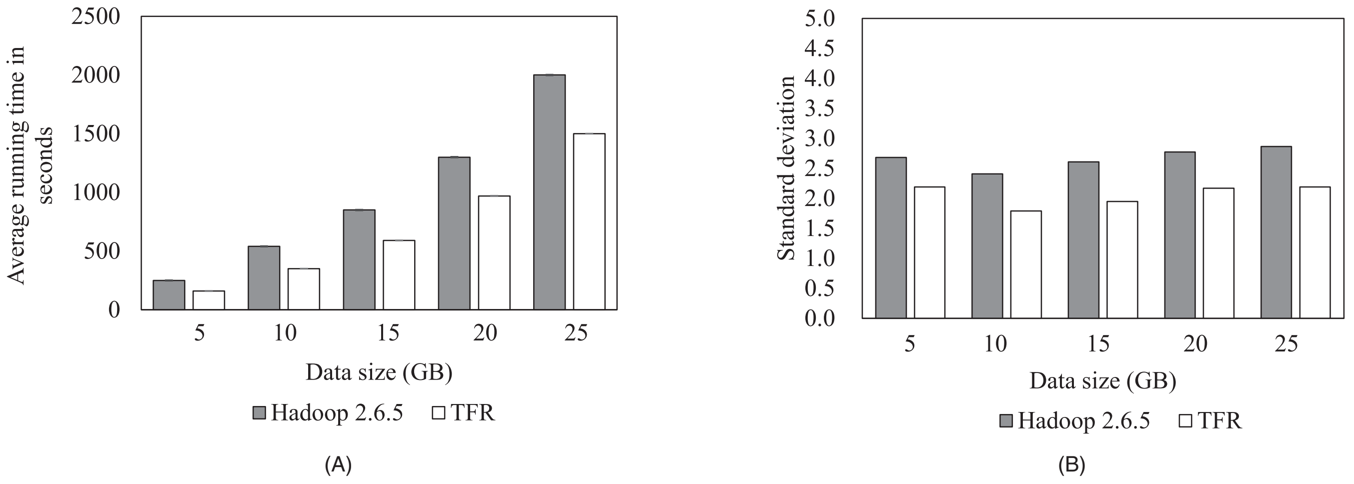

FIGURE 6 (A) Running times of the PageRank benchmark on Hadoop 2.6.5 and TFR. (B) Quantification measure of the standard deviation of Hadoop 2.6.5 and the execution time of the TFR for the PageRank benchmark

TABLE 3 Performance improvement rate of TFR over that of Hadoop 2.6.5 on running the Sort benchmark

	5 GB		10 GB		15 GB		20 GB		25 GB	
Input	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**
Average execution time (s)	140	100	300	220	500	370	927	690	1550	1200
Standard deviation	2.09	1.09	2.79	1.78	2.70	2.28	2.30	1.94	2.86	1.67
CV (%)	1.48	1.09	0.92	0.81	0.53	0.61	0.24	0.52	0.18	0.13
Improvement rate (%)	28.57		26.67		26.00		25.57		22.58	

TABLE 4 Performance improvement rate of TFR over that of Hadoop 2.6.5 on running the TeraSort benchmark

	5 GB		10 GB		15 GB		20 GB		25 GB	
Input	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**	**Hadoop**	**TFR**
Average execution time (s)	155	110	350	250	600	450	1010	780	1650	1300
Standard deviation	2.60	1.64	3.08	1.78	2.77	2.28	3.03	2.44	2.77	2.38
CV (%)	1.67	1.48	0.88	0.71	0.46	0.50	0.30	0.31	0.14	0.18
Improvement rate (%)	29.03		28.57		25.00		22.77		21.21	

TABLE 5 Performance improvement rate of TFR over that of Hadoop 2.6.5 on running the PageRank benchmark

	5 GB		10 GB		15 GB		20 GB		25 GB	
Input	Hadoop	TFR	Hadoop	TFR	Hadoop	TFR	Hadoop	TFR	Hadoop	TFR
Average execution time (s)	250	160	540	350	850	590	1,300	970	2000	1500
Standard deviation	2.68	2.19	2.40	1.78	2.60	1.94	2.77	2.16	2.86	2.19
CV (%)	1.07	1.36	0.44	0.51	0.30	0.33	0.21	0.22	0.143	0.146
Improvement rate (%)	36		35.19		30.58		25.38		25	

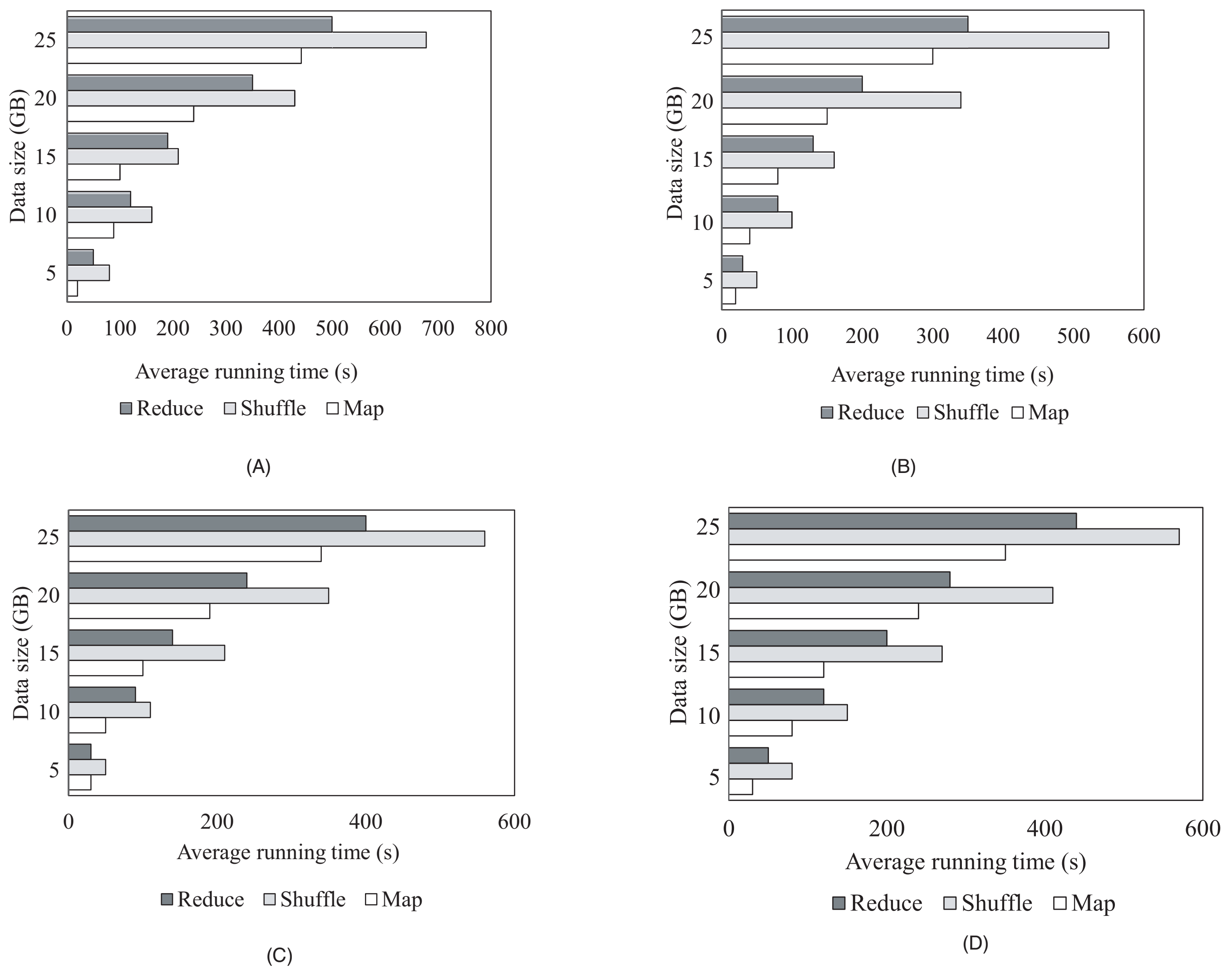

FIGURE 7 Running times of the map, shuffle, and reduce phases for the (A) WordCount, (B) Sort, (C) TeraSort, and (D) PageRank benchmarks

7 | CONCLUSIONS AND FUTURE WORK

A new task failure resiliency method has been proposed and implemented, which performs better than the standard Hadoop. The intermediate data from the map and shuffle phases are backed up to an in-memory data store for error recovery. The TFR technique is implemented on the basis of Hadoop 2.6.5, a popular open-source implementation of MapReduce. It was evaluated to determine the overhead and effectiveness of all features included in it. TFR outperformed Hadoop 2.6.5 in different scenarios including conditions with no failures and a diverse density of failures.

In the future, we will recover the worker node failures by scheduling the least reliable node to another healthier node and will seek further improvements to the node failure recovering strategy. We will also concentrate on more possible optimizations to further improve the MapReduce performance and

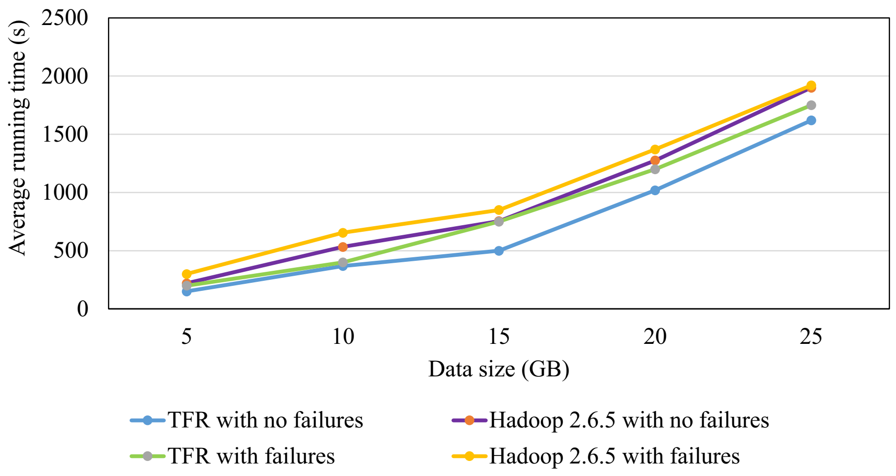

FIGURE 8 Performance measure of TFR and Hadoop 2.6.5 in terms of failures

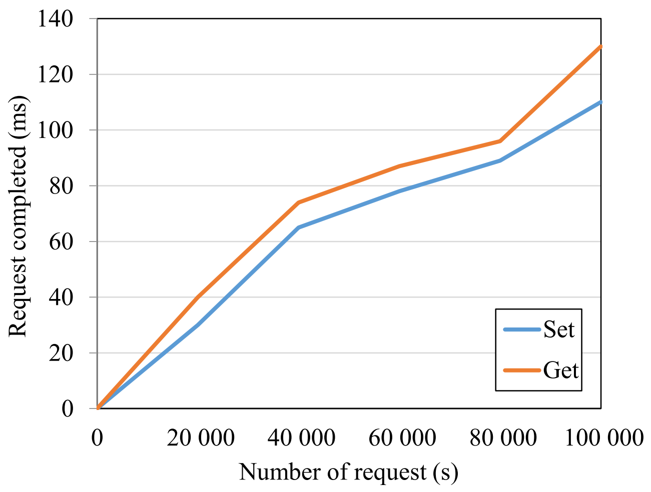

FIGURE 9 Performance benchmark of TFR in terms of Redis storage

reduce the impact of a possible data security breach. The weakness of the data security mechanism obstructs the development and use of Hadoop. Hadoop and HDFS have no security model against storage servers. Accordingly, to make the Hadoop platform more secure for enterprises, we need to propose new security models in the future.

ORCID

Kavitha C https://orcid.org/0000-0001-7034-2848

REFERENCES

1. H. Jin et al., *Performance under Failures of MapReduce Applications*, in Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (Newport Beach, CA, USA), May 2011, pp. 608–609.
2. H. Herodotou, *Hadoop performance models*. arXiv:1106.0940, 2011, 1–19.
3. H. Wang et al., *BeTL: MapReduce checkpoint tactics beneath the task level*, IEEE Trans. Services Comput. **9** (2016), no. 1, 84–95.
4. M. Isard et al., *Dryad: Distributed data parallel programs from sequential building blocks*, in Proc. ACMSIGOPS, Eur. Conf. Comput. Syst. (Lisbon Portugal), Mar. 2007, pp. 59–72.
5. J. Dean, *Experiences with MapReduce, An abstraction for large-scale computation*, in Proc. Int. Conf. Parallel Architectures Compilation Techn. (Seattle, WA, USA), Sept. 2006, p. 1.
6. K. Plankensteiner et al., Fault Detection, Prevention and Recovery in Current Grid Workflow Systems, Grid and Services Evolution, Springer, 2009, pp. 1–13. https://doi.org/10.1007/978-0-387-85966-8_9.
7. Y. Chen et al., *aHDFS: An Erasure-Coded Data Archival System for Hadoop Clusters*, IEEE Trans. Parallel Distrib. Syst. **28** (2017), no. 11, 3060–3073.
8. Q. Zheng, *Improving MapReduce Fault Tolerance in the Cloud*, in Proc. IEEE Int. Symp. Parallel Distrib. Process. (Atlanta, GA, USA), May 2010, pp. 1–6.
9. P. Costa et al., *Byzantine Fault-Tolerant MapReduce: Faults are not just crashes*, in Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (Athens, Greece), 2011, 32–39.
10. P. Hu and W. Dai, *Enhancing Fault Tolerance Based on Hadoop Cluster*, Int. J. Database Theor. Appl. **7** (2014), no. 1, 37–48.
11. J. Lin et al., *Modeling and Designing Fault-Tolerance Mechanisms for MPI-Based MapReduce Data Computing Framework*, in Proc. IEEE Int. Conf. Big Data Comput. Service Applicat. (Redwood City, CA, USA), 2015, pp. 176–183.
12. J.-A. Quiané-Ruiz et al., *RAFTing MapReduce: Fast Recovery on the RAFT*, in Proc. IEEE Int. Conf. Data Eng. (Hannover, Germany), Apr. 2011, pp. 589–600.
13. R. Gu et al., *SHadoop: Improving mapreduce performance by optimizing job execution mechanism in Hadoop Clusters*, J. Parallel Distrib. Comput. **74** (2014), no. 3, 2166–2179.
14. J. Dittrich et al., *Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)*, Proc. VLDB Endowment **3** (2010), no. 1, 515–529.
15. https://data-flair.training/blogs/hadoop-mapper-in-mapreduce/.
16. H. Jianfeng et al., *KVBTree: A Key/Value Based Storage Structure for Large-Scale Electric Power Data*, in Proc. Int. Conf. Adv. Cloud Big Data (Chengdu, China), Aug. 2016, pp. 133–137.
17. M. Zaharia et al., *Improving MapReduce performance in heterogeneous environments*, in Proc. USENIX Conf. Operat. Syst. Design Implementation (San Diego, CA, USA), Dec. 2008, pp. 29–49.

18. AWS, *What Is Amazon ElastiCache for Redis?*, https://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/WhatIs.html.
19. 8K Miles, *Billion Messages – Art of Architecting scalable ElastiCache Redis tier*, Sept. 2014, https://8kmiles.com/blog/billion-messages-art-of-architecting-scalable-elasticache-Redis-tier.
20. L. Chen et al., *MRSIM: Mitigating Reducer Skew in MapReduce*, in Proc. Int. Conf. Adv. Inf. Netw. Applicat. Workshops (Taipei, Taiwan), Mar. 2017, pp. 379–384.
21. C. B. Walton, A. G. Dale, and R. M. Jenevein, *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins*, in Proc. Int. Conf. Very Large Data Bases (Barcelona, Spain), 1991, pp. 537–548.
22. S. Acharya, P. B. Gibbons, and V. Poosala, *Congressional samples for approximate answering of group-by queries*, ACM SIGMOD Record. ACM **29** (2000), no. 2, 487–498.
23. A. Shatdal and J. F. Naughton, *Adaptive Parallel Aggregation Algorithms*, ACM SIGMOD Record. ACM **24** (1995), no. 2, 104–114.
24. Redis, *How fast is Redis?*, https://Redis.io/topics/benchmarks.

AUTHOR BIOGRAPHIES

Kavitha C received her BE degree in computer science and engineering from Agni College of Technology, Chennai, India, in 2013 and her ME degree in computer science and engineering from Saveetha Engineering College, Chennai, India, in 2015. She is pursuing her PhD in information and communication engineering at Anna University, Chennai, India. Her research interests include big data an alytics, machine learning, and cloud computing.

Anita X received her BE degree in computer science and engineering from Madurai Kamaraj University, Madurai, India, in 2003 and her ME degree in computer science and engineering from Anna University, Chennai, India, in 2008. She has obtained her PhD in the field of network security. She is presently working as an associate profes sor in the Department of Computer Science and Engineering, Jerusalem College of Engineering, Chennai, India. Her research interests include data analytics, image processing, cloud computing, sensor networks, and network security. Under her guidance, one scholar had obtained a PhD and there are approximately four other research scholars pursuing their PhD. She has 20 publications in national/international journals and conferences.