A NEW CHECKPOINT AND ROLLBACK FOR HIGH AVAILABILITY OF MAPREDUCE COMPUTING

By

NIKHIL PENDEM

Bachelor of Technology in Computer Science

Kakatiya University

Warangal, Telangana, India

2011

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2015

A NEW CHECKPOINT AND ROLLBACK FOR HIGH AVAILABILITY OF MAPREDUCE COMPUTING

Thesis Approved:

Dr. Nohpill Park

Thesis Chair and Adviser

Dr. Eric Chan Tin

Committee Member

Dr. Johnson P. Thomas

Committee Member

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Nohpill Park for the continuous support of my thesis study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Eric Chan Tin Dr. Johnson P. Thomas, for their encouragement, insightful comments.

I would like to thank my family for the support they provided me through my entire life and in particular, I must acknowledge my best friend, Sridhar Vemula, without whose encouragement, support and editing assistance, I would not have finished this thesis.

Name: NIKHIL PENDEM

Date of Degree: JULY, 2015

Title of Study: A NEW CHECKPOINT AND ROLLBACK FOR HIGH AVAILABILITY OF MAPREDUCE COMPUTING

Major Field: COMPUTER SCIENCE

Abstract: MapReduce is a programming model and an associated implementation for processing and generating large data sets, so called big data. A MapReduce job usually splits the input data-set into independent chunks which are processed by the maptasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. If an error occurs in a name node other name node will take over the failed node and continues its execution. Other than data node failure, if an error occurs during the program execution itself then there must be a detection and recovery steps to correct the error.

A solution for this problem is to implement the checkpoint and rollback mechanism in the system. When memory error occurs in the MapReduce program then execution in all the data nodes will be stopped and it starts all over from the starting phase in hadoop. The proposed methodology is to detect the heap space error [10] and provide a recovery operations by employing a new checkpoint and recovery process. In order to realize this, a new phase based checkpoint and rollback is proposed versus the hadoop default configuration. Once an error occurs in hadoop, the memory size required by the program is raised then the configuration file setting is modified and then a checkpoint is set and from there next phases will be executed. In this way, the entire already completed phases are not needed to be re-executed. From the experimental results, the hadoop availability is increased to 53.22% compared to the default hadoop configuration thereby decreasing the running time of the application.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF GRAPHS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

MapReduce is a new paradigm of applications computing to run on top of very large commodity clusters. However, it is pointed out that a system composed of 10 thousand super reliable servers can watch one fail per day [1]. Fault-tolerance is an important aspect in large clusters because the probability of node failures increases with the growing number of computing nodes exponentially. This is confirmed by a 9-year [2] study of node failures in large computing clusters. Moreover, large datasets themselves are flawed, and contain data inconsistencies and missing values (bad records) [3] to mention a few. This may, in turn, cause a task or even an entire application to crash. The impact of task and node failures may be considerable in terms of performance as well as reliability or availability.

MapReduce makes task and node failures invisible to users it automatically reschedules failed tasks — due to a task or node failure — to available and optimal nodes. However, re-computing failed tasks from the scratch as exercised in current default hadoop configuration can significantly decrease the performance of long-running applications [4] — especially for applications composed of several MapReduce jobs — by propagating and adding up delays. A common solution is to checkpoint and save the state of ongoing computation on stable storage and resume the computation from the last and known-safe checkpoint in case of failures. However, checkpointing ongoing computation in MapReduce is challenging for several reasons:

(1) Checkpointing techniques require the system to replicate intermediate results on stable storage. This can significantly decrease performance as MapReduce jobs often produce large amounts of intermediate results [1].

(2) Keeping and sustaining checkpoints information on a stable storage generally requires heavy use of network bandwidth, which is a the most known bottlenecking resource in MapReduce systems [1].

(3) Recovering tasks from failures intermediate results from stable storage commit both network and I/O resources heavily resuming [1].

Therefore, utilizing a straight-forward usage of traditional checkpointing techniques [5], [6]would significantly decrease the performance of MapReduce jobs.

There is a few resilient map task checkpointing tactics(ReCT) such that map tasks create checkpoints on the fly, making it possible to implement fault tolerance strategies behind the task operations.

One of the existing checkpoint and rollback scheme is ReCT and it features the following capabilities [7]:

1) Enhanced Map Task Checkpointing: Whenever a mapper generates an output spill, the mapper sends meta data of this spill to the master. Reducers shuffle spill files created by mappers rather than per-map output files. When a task attempt fails, a retry attempt will skip finished input ranges.

2) Resilient Checkpoint Creation: Finer-grained checkpoint creation policy is included in ReCT. Users can tell ReCT to create checkpoints at periodic intervals, or every time after certain amount of input data is processed.

3) Comprehensive Evaluation: The word count program has been used as the most simple and pure application of MapReduce, to comprehensively analyze behaviors, overhead and performance improvement under task failures in ReCT [7]. Realistic data and applications are also adopted in this evaluation to better understand both pros and cons of ReCT [7].

In this work, we implement the checkpoint and recovery methods to fix the memory errors such as Java heap space [10] in the MapReduce program. The implementation is different from ReCT, the proposed methodology will create checkpoints based on phase level while ReCT create checkpoints at periodic intervals, or every time after certain amount of input data is processed. If an error occurs in the program we increase the memory size required by the program and perform the recovery operation. Thus we decrease the running time of MapReduce application by not executing the already completed phases. In this way, the performance of MapReduce application will be increased by achieving higher Hadoop availability as compared to the existing Hadoop architecture.

CHAPTER II

REVIEW OF LITERATURE

The existing checkpoint mechanism in default Hadoop architecture will be reviewed in this chapter.

2.1 Checkpoint and rollback process in Hadoop

Checkpointing is a vital of keeping up and enduring filesystem metadata in HDFS. It's pivotal for productive NameNode recovery and restart, and is an essential marker of overall cluster health.

At an abnormal state, the NameNode's essential obligation is storing the HDFS namespace. This implies things like the directory tree, file permissions consents, and the mapping of file to block IDs. It's vital that this metadata are securely continued to stable storage for fault tolerance.

Normally, this filesystem metadata is put away in two distinct constructs [8]:

1. fsimage 2. edit log

The fsimage is a file that represents to a point-in-time snapshot of the filesystem's metadata. On the other hand, while the fsimage document format is exceptionally productive to read, its unsatisfactory for making little incremental upgrades like renaming a single file. Therefore, as opposed to composing another fsimage every time the namespace is altered, the NameNode rather records the modifying operation in the edit log for durability. Along these lines, if the NameNode crashes, it can restore its state by first stacking the fsimage then replaying every one of the operations in the edit log to make up most recent state of the namesystem. The edit log embodies a progression of files, called alter log portions, that together speak to all the namesystem changes made since the creation of the fsimage.

When checkpoint has to create?

An average edit ranges from 10s to 100s of bytes [8], however over time edits can accumulate to become unwieldy. Several issues can emerge from these vast edit logs. In extreme cases, it can fill up all the available disk capacity on a node, a huge edit log can significantly defer NameNode startup as the NameNode reapplies all the edits. This is the place checkpointing comes in.

Checkpointing is a process that takes a fsimage and edit log and compacts them into another fsimage. In this way, as opposed to replaying a possibly unbounded edit log, the NameNode can load the last in-memory state directly from the fsimage. This is a much more effective operation and diminishes NameNode startup time.

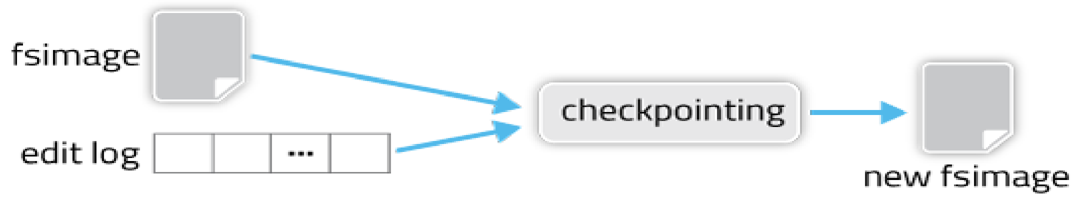


Fig. 2.1 Checkpoint creates a new fsimage from an old fsimage and edit log

During a checkpoint, the namesystem also needs to restrict concurrent access from other users. So, rather than pausing the active NameNode to perform a checkpoint, HDFS defers it to either the Secondary NameNode or Standby NameNode, depending on whether NameNode high-availability is configured.

2.2 Checkpoint with a Standby NameNode

High Availability(HA) Name Node is to add support for deploying two Name Nodes in an active/passive configuration. This is a common configuration for highly-available distributed systems, and HDFS's architecture lends itself well to this design. Even in a non-HA configuration, HDFS already requires both a Name Node and another node with similar hardware specs which performs checkpointing operations for the Name Node. The design of the HA Name Node is such that the passive Name Node is capable of performing this checkpointing role, thus requiring no additional Hadoop server machines beyond what HDFS already requires.

The standby NameNode maintains a relatively up-to-date version of the namespace by periodically replaying the new edits written to the shared edits directory by the active NameNode. As a result, checkpointing is as simple as checking if either of the two preconditions are met, saving the namespace to a new fsimage, then transferring the new fsimage to the active NameNode via HTTP.

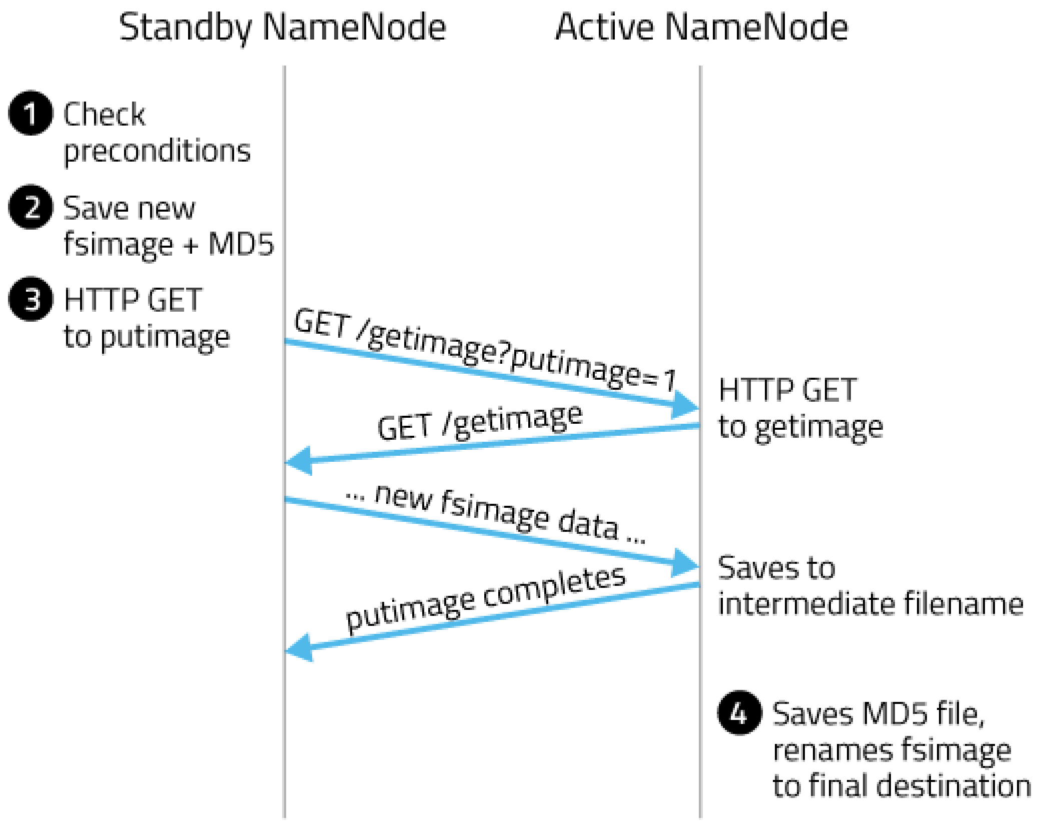


Fig. 2.2 Checkpoint with NameNode configured

Here, Standby NameNode is abbreviated as SbNN and Active NameNode as ANN [8]:

- SbNN checks whether either of the two preconditions are met: elapsed time since the last checkpoint or number of accumulated edits.
- SbNN saves its namespace to an a new fsimage with the intermediate name fsimage.ckpt_, where txid is the transaction ID of the most recent edit log transaction. Then, the SbNN writes an MD5 file for the fsimage, and renames the fsimage to fsimage_. While this is taking place, most other SbNN operations are blocked. This means administrative operations like NameNode failover or accessing parts of the SbNN's are blocked. Routine HDFS client operations like reading, and writing files are unaffected as these operations are serviced by the ANN.

- SbNN sends an HTTP GET to the active NN’s GetImageServlet at /getimage?putimage=1. The URL parameters also have the transaction ID of the new fsimage and the SbNN’s hostname and HTTP port.
- The active NN’s servlet uses the information in the GET request to in turn do its own GET back to the SbNN’s GetImageServlet. Similar to the standby, it first saves the new fsimage with the intermediate name fsimage.ckpt_, creates the MD5 file for the fsimage, and then renames the new fsimage to fsimage_.

CHAPTER III

PROPOSED METHODOLOGY

In the proposed methodology, a new checkpoint and rollback process in a different manner will be implemented. In MapReduce program if there is any error occured then the program will be suspended in the middle of execution. Since this is not a node failure no other node takes over the remaining program for execution.

It is particularly targeted at memory errors such as java heap space error [10] in the MapReduce program. Memory errors may be a serious issue in distributed applications, exhausting their performance [9], regardless of the capability that they keep running on platforms with automatic memory recovery as exercised in Java Virtual Machine (JVM). A memory error in a modern JVM will cause an expensive restart of the JVM.

OBJECTIVE

The goal of this work is to increase the Hadoop availability. To achieve this a new phase-based checkpoint and rollback method is developed. This methodology handles the heap space error [10] which frequently occurs in MapReduce applications, it provides the necessary and efficient recovery operations. If an error occurs in the MapReduce program, instead of terminating it in the middle, the program will be tuned by increasing the memory size and split size. Then the execution flow will roll back to the most recent checkpoint, from that point onwards the next phase will be executed. Hadoop availability is compared with the default hadoop configuration.

3.1 Error Injection:

Error injection in MapReduce program is done through a command line argument. An argument to the MapReduce program passed to specify the error type. It can be either the map-error, reduce-error or no-error. The following command is an example of injecting the error.

$ hadoop jar <jar-filename> maperror <size of input in GB> <input-path> <output-path>

Default value of maximum heap space is set to 10MB in hadoop. This is done by setting the "mapred.child.java.opts" to Xmx10M in the configuration file. If a maperror is passed to the program, then an error at the map phase will occur. Then the MapReduce program will be tuned by increasing the split size and memory size , as to be shown later. Java xmx is the maximum memory allocation pool for a Java Virtual Machine (JVM) and if this sets to a lesser value then a Heap Space Error will occur.

The implementation of checkpoint and rollback recovery process is summarized as follows.

- Error detection in the program
- Modifying configuration properties in recovery operation
- Recovery through Java Virtual Remote Calls
- Error Recovery Process
- Increasing the MapReduce Performance

3.2 Error Detection in the Program

If MapReduce program has terminated abruptly then there should be an error occurred in the program. It is proposed in this work how to detect the error occurred in MapReduce and provides the necessary solution to it. Particularly heap space error is targeted for detection in the MapReduce program.

A thread is maintained in parallel to the MapReduce application and it will monitor the implementation and process of completion of each phase. If the execution flow of each phase is normally running during the MapReduce phase then it is an identification of no error in the program. Otherwise, an error occurs in the MapReduce program.

The following diagram illustrates the normal flow of a MapReduce application in the proposed new hadoop configuration.

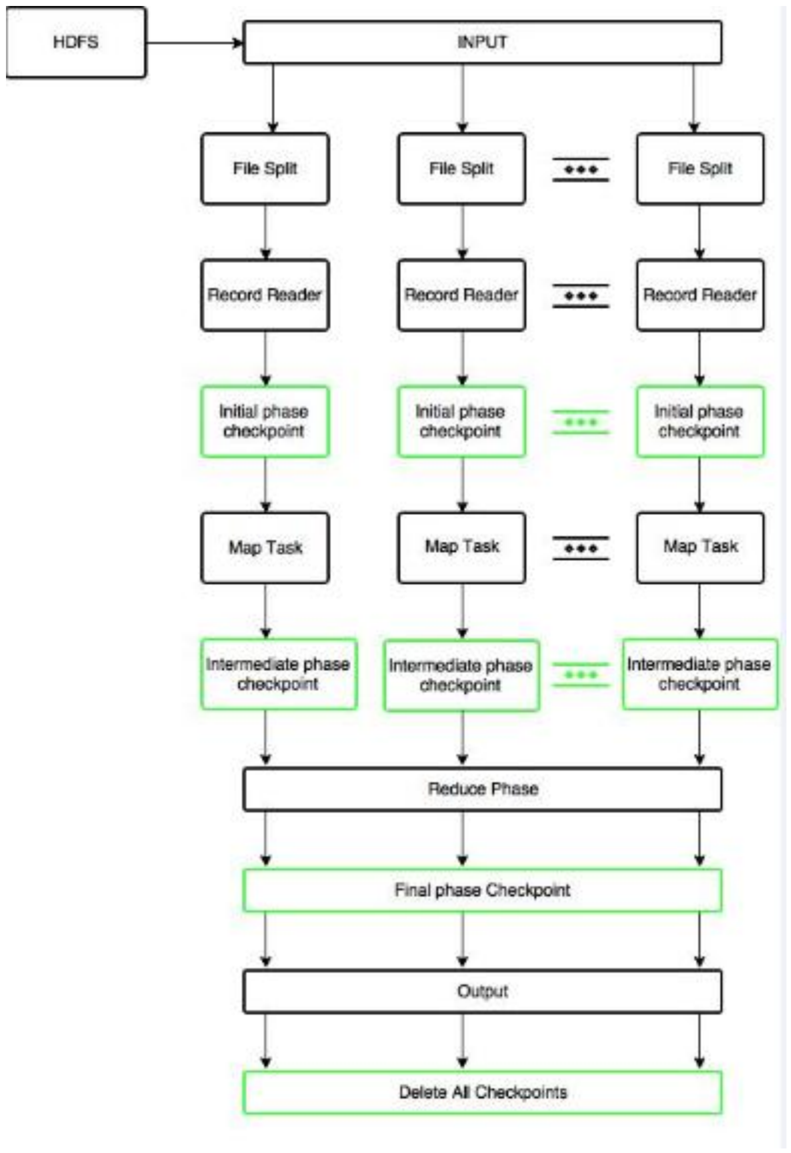


Fig. 3.1 Normal MapReduce flow in the proposed new hadoop configuration

Initially, the input is taken from HDFS and InputFormat class is used to split the input into multiple file splits. After dividing the file the initial checkpoint will be created. Once the initial phase checkpoint is created the Map task will be executed. After successful completion of these map tasks another checkpoint will be created called as intermediate phase checkpoint.

The next step in the MapReduce is Reduce phase where the output of each map task is processed to the reduce phase. After successful completion of the reduce phase, a final phase checkpoint will be created. After creating the final checkpoint in the MapReduce program all checkpoints will be removed to save the memory in the local system.

Error at Map phase:

The following flow illustrates the error location occurred during the Map task.

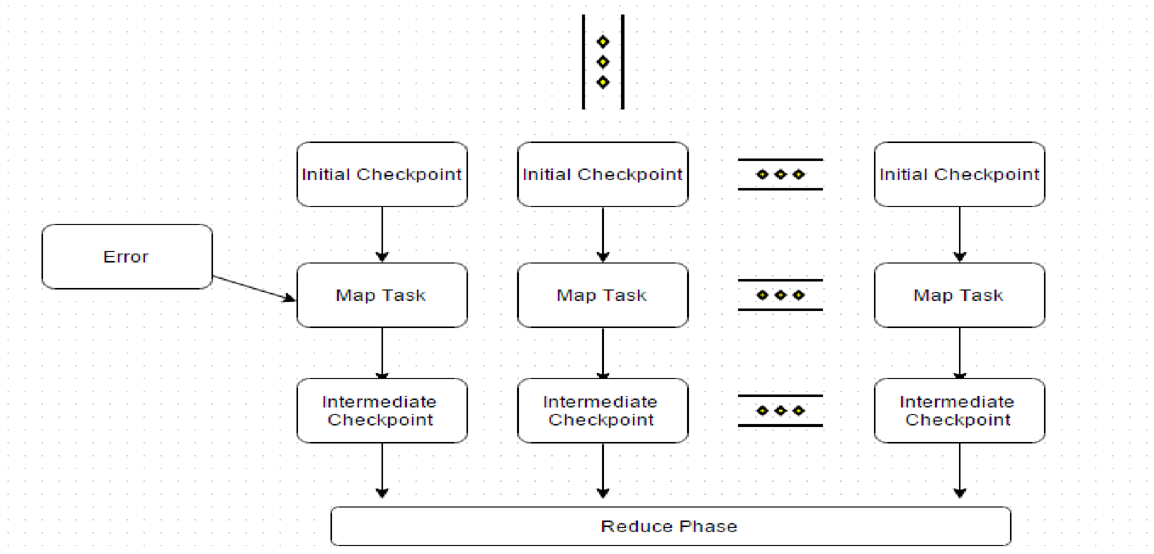


Fig. 3.2 Error at Map phase

In the conventional MapReduce program, if an error occurs during map phase then the program terminates in the middle of execution. But in the proposed methodology, if an error occurs during the map phase the program will be tuned and the control will move over to the initial phase checkpoint.

If an error occurs before the map task, i.e., at the Record Reader, then the input from the HDFS will have to be obtained and then the file split operation is performed and the initial checkpoint will be created. Then, it will proceed to the next phases.

Error at Reduce phase:

The following diagram illustrates an error occurs in Reduce phase.

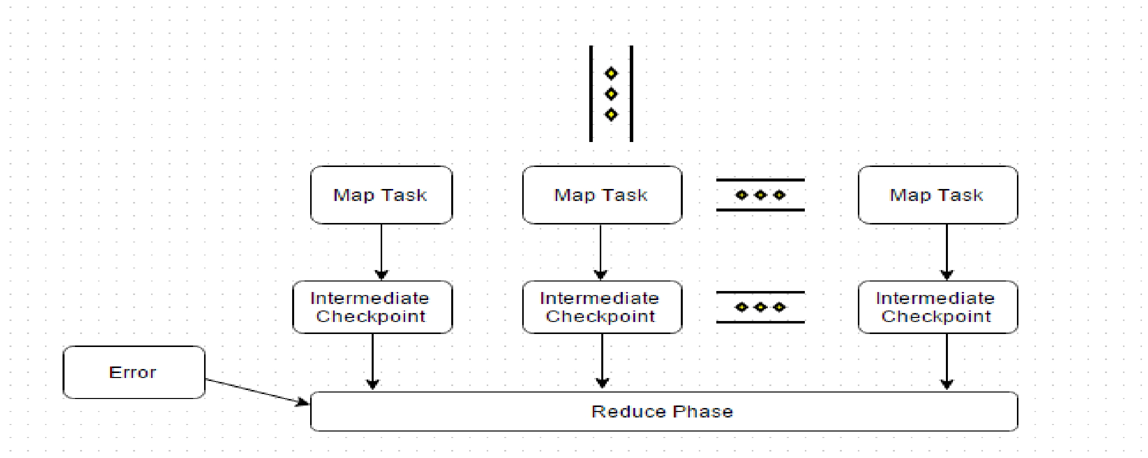


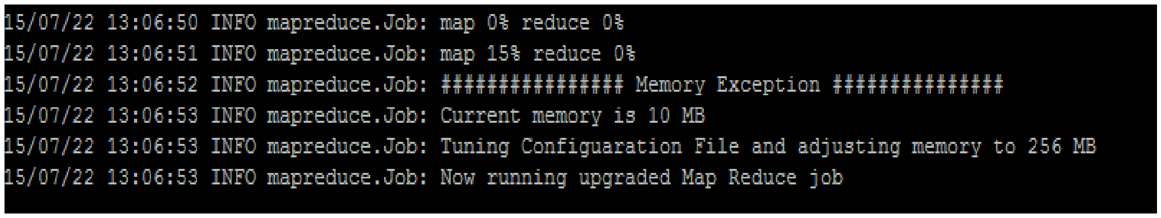
Fig. 3.3 Error at Reduce phase

In conventional MapReduce program, if an error occurs during Reduce phase the execution flow will be stopped in the middle of the execution and the program will be terminated. In this methodology, instead of terminating it in the middle it will roll back to the immediate checkpoint.

Checkpoints will be created in each phase of MapReduce program such as checkpoints at Map phase and Reduce phase. If any error occurs in any phase then program will roll back to the nearest checkpoint which is created and known to be safe in MapReduce program.

3.3 Configuration properties in recovery operation

In the proposed methodology targeting at the heap space error, the following default attributes will be considered and modified in the configuration file. They are split size, the number of splits and the java xmx value.



```
15/07/22 13:06:50 INFO mapreduce.Job: map 0% reduce 0%
15/07/22 13:06:51 INFO mapreduce.Job: map 15% reduce 0%
15/07/22 13:06:52 INFO mapreduce.Job: ############### Memory Exception ##############
15/07/22 13:06:53 INFO mapreduce.Job: Current memory is 10 MB
15/07/22 13:06:53 INFO mapreduce.Job: Tuning Configuaration File and adjusting memory to 256 MB
15/07/22 13:06:53 INFO mapreduce.Job: Now running upgraded Map Reduce job
```

Fig 3.4 Modification of configuration properties

Initially by deafult maximum heap size is set to 10 MB by using the configuration property "mapred.child.java.opts", "-Xmx10m". Then, memory size will be modified based on the xmx value. If xmx value is in between 10 and 50, then memory size will be modified to 100MB or if xmx value is in between 50 and 100 then memory size will be modified to 150MB. In this way, MapReduce will be tuned by modifying the maximum heap space size in the configuration file.

MapReduce application can also be tuned by increasing the split size. The initial split size is 128 MB and its value will be set by using the "mapred.max.split.size" property in the configuration file. The split size will be modified according to the xmx value. If maximum heap size (java xmx) value is in between 100 and 150, then split size will be modified to 150MB.

By modifying the above two configuration properties, the MapReduce application can be tuned and recovered to a normal operation.

3.4 Recovery through Java Virtual Remote Calls

In this process, the MapReduce application is tuned to increase its performance and then a recovery process will be performed. Initially, the error occurred in the MapReduce program is detected and then the most recent checkpoint in the program is checked.

Java Virtual Remote Calls

The intermediate results in the map phase are stored in the individual local nodes with unique triplet string of 12 bytes length. It is sufficient to store the checkpoint information which includes the task ID and unique identifier. This data is stored in the local nodes, and a custom Java remote method invocation sends the data to the Reducer node. The custom built reducer input format takes in the intermediate data from the map phase and feeds it to WrappedReducer. A Wrapped Reducer extends the user defined Reducer class and appends some built-in helper methods.

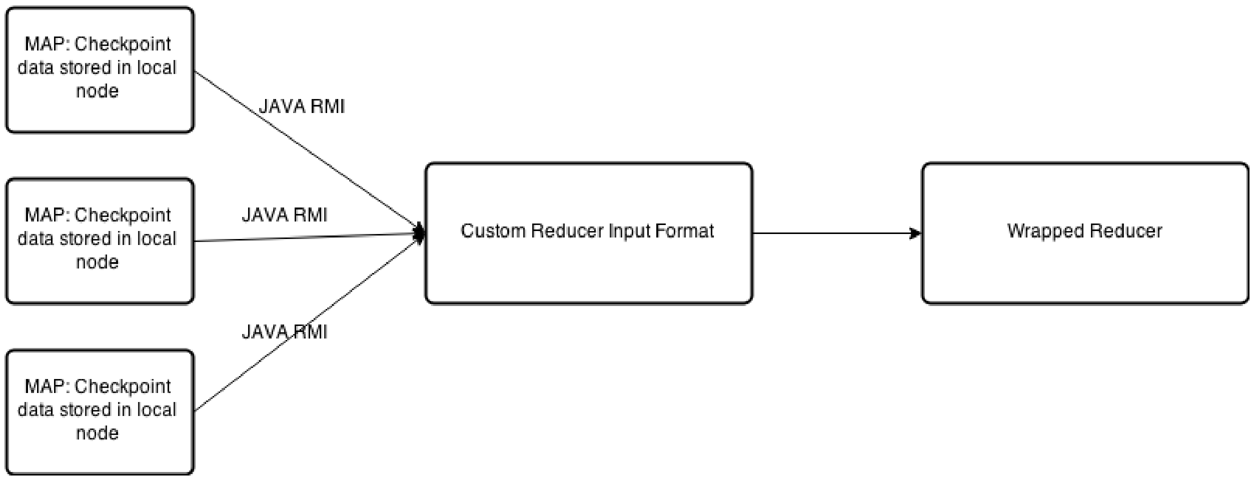

MAP: Checkpoint data stored in local node
JAVA RMI
MAP: Checkpoint data stored in local node
JAVA RMI
JAVA RMI
MAP: Checkpoint data stored in local node
Custom Reducer Input Format
Wrapped Reducer

Fig. 3.5 Facilitation through RMI

If the intermediate data is stored in the local HDFS, then this data and new configuration files will be passed to the reduce phase such that it re-executes the reduce phase. Thus, it can lead to the memory error in recovery process. As it starts from the middle of MapReduce program, that is, from Reduce phase, it will take less time to complete its execution.

3.5 Error Recovery Process

A new phase level checkpoints is proposed, particularly checkpoints at Map phase and Reduce phase are implemented.

In the proposed MapReduce recovery process intermediate results are stored and kept at several points in time to checkpoint the computing progress done by mappers and reducers. This enables MapReduce to rollback and resume tasks from last checkpoints in case of a task failure.

As conventional MapReduce does not keep the progress of tasks, it must perform failed tasks from the beginning. A Local Checkpointing is employed to deal with this. This checkpointing stores task progress computation on the local disk of tasks without sending replicas through the network so as to not increase network. It might seems like that local checkpointing may considerably slow down tasks, since it has to repeatedly write all checkpoint information to disk, including the output produced so far. In the proposed approach, however, tasks only perform local checkpointing when they store intermediate results of tasks to disk anyway.

3.5.1 Error Recovery at Map Phase:

If a memory error occurs at map phase, a tuning operation is performed and will roll back to the initial checkpoint.

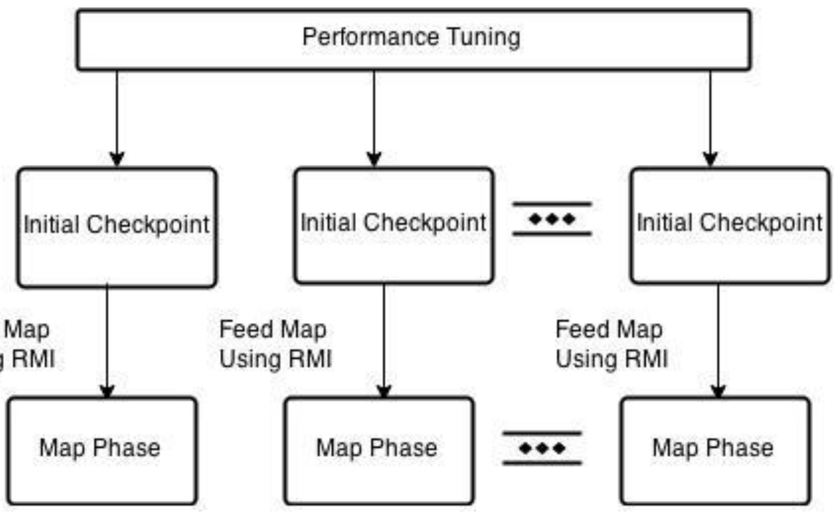


Fig. 3.6 Error recovery at Map phase

After the initial checkpoint, the intermediate data stored in the local storage will be collected and send this data to the map phase by using the RMI (Remote Method Invocation) method. Then. this RMI will collect the data from the local disks and send them to the map phase. Finally, the corresponding next phases will be executed up next as scheduled.

Pseudo code for error recovery at Map phase:

```
initCheckpoint(jb); //known as initial checkpoint
    try{
        userDefinedMap();
        // in the above method, we will retrieve the stored key value pairs from LOG
        // then we will run the all map methods in parallel
        // after successful completion above method the reduce phase will be performed
    } catch(HadoopMemoryException e){
    }
    initCheckpoint(jb); //known as intermediate checkpoint
```

3.5.2 Creation of Checkpoint

Initially, a checkpoint will be created called as initial checkpoint and then a Map method will be called. If any error occurs in the Map phase then the memory size, split size will be increased. Then, the most recent checkpoint information will be obtained and the Map-Recovery method will be called. This method will have the information about the Map phase and tuning data with the most recent checkpoint so that we do a recovery from the error at Map phase. After successful completion of map phase intermediate checkpoint is created.

Pseudo code for creating the checkpoint:

```
begin initCheckPoint(JobHelper jb)
        CheckPoint chk = null;
        JobContext jb = getJobContext();
        if jb.isLocalCheckpoint() then
                chk = new LocalCheckPoint(jb);
        if jb.isRemoteCheckpoint() then
                chk = new RemoteCheckPoint(jb);
end
```

The above pseudo code is used to create a checkpoint in the program, and it can be either a local checkpoint or remote checkpoint.

Pseudo code for creating local checkpoint:

```
begin LocalCheckpoint(JobHelper jb)
                String jobID = JobHelper.getJobID(context);
                String taskID = JobHelper.getTaskID(context);
                String id = jobID +":" +taskID;
                byte[] md5 = MD5Hash.digest(id.getBytes()).toString().getBytes();
                LocalFileSystem fs = FileSystem.getLocal(context.getConfiguration());
```

```
                    Path inputPath = fs.makeQualified(new
Path("/opt/"+jobID+"/"+taskID+".tmp"));
                    fs.mkdirs(inputPath);
end
```

In the above pseudo code, a local checkpoint is created in the local file system. The jobID, taskID are retrieved and these two ids are combined. The local checkpoint will focus on task blocks in the local file system where input is stored. With the id generated, an md5 will be created and by using all the above the local checkpoint will be created. The checkpoint path will be stored in local file system.

Pseudo code for creating remote checkpoint:

```
begin RemoteCheckpoint(JobHelper jb)
          String taskID = JobHelper.getTaskID(context);
                    String id = jobID +":" +taskID;
                    byte[] md5 = MD5Hash.digest(id.getBytes()).toString().getBytes();
                    FileSystem fs = FileSystem.get(context.getConfiguration());
                    Path inputPath = fs.makeQualified(new
Path("/user/tmp/"+jobID+"/"+taskID+".tmp"));
                    fs.mkdirs(inputPath);
end
```

Creation of remote checkpoint is the same as creation of local checkpoint, except that it focuses on hadoop filesystem rather than local file system. Remote checkpointing is a backup checkpoint, because if a data node fails then the checkpoints which was created in the local system may get lost. To address this, a remote checkpoint will have the equivalent checkpoint data, so that if any data node fails it can be recovered from this remote checkpoint.

A mapper executes this algorithm when it splits intermediate results to local disk. Local Checkpointing first retrieves progress information from the buffer containing input data before allowing for any further computation on the input buffer. After that, the mapper writes the splits to local disk by parallel threads. If the split is correctly written, it proceeds to stored to the local checkpoint storage on disk.

A simple string of 12 bytes length is sufficient to store the checkpoint information: taskID, a unique task identifier that remains invariant over several attempts of the same task; splitID, the local path to the split data; offset, specifying the last byte of input data processed by splitting time. If an earlier checkpoint existed, it would be simply overwritten. Notice that split data is implicitly chained backwards. Thus, any checkpoint with a reference to the most recent split is sufficient to locate all earlier split files as well.

3.5.3 Error Recovery at Reduce phase:

If an error occurs at the reduce phase then the MapReduce program will be tuned and then roll back to the intermediate checkpoint.

Pseudo code for error recovery at Reduce phase:

```
initCheckpoint(jb); // known as intermediate checkpoint
try{
            userDefinedReduce();
            // in the above method, it will combine map output key value pairs
            // we tune the MapReduce application
            // then it will perform recovery operation
      } catch(HadoopMemoryException e){

      }
      initCheckpoint(jb); //known as final checkpoint
      cleanupCheckpoints(true);
```

After successful completion of map phase one more checkpoint called as intermediate checkpoint will be created then the reduce method will be called. If any error occurs in the reduce phase then the memory size is increased and then recovery method will be called. This method will pass the information about the Reduce phase tuning data with the most recent checkpoint so that a recovery from the error at Reduce phase can be performed correctly.

Upon successful completion of Reduce phase another checkpoint referred to as the final checkpoint will be created. Once the final checkpoint is created and the program doesn't cause any error then all the checkpoints will be removed by calling the below method.

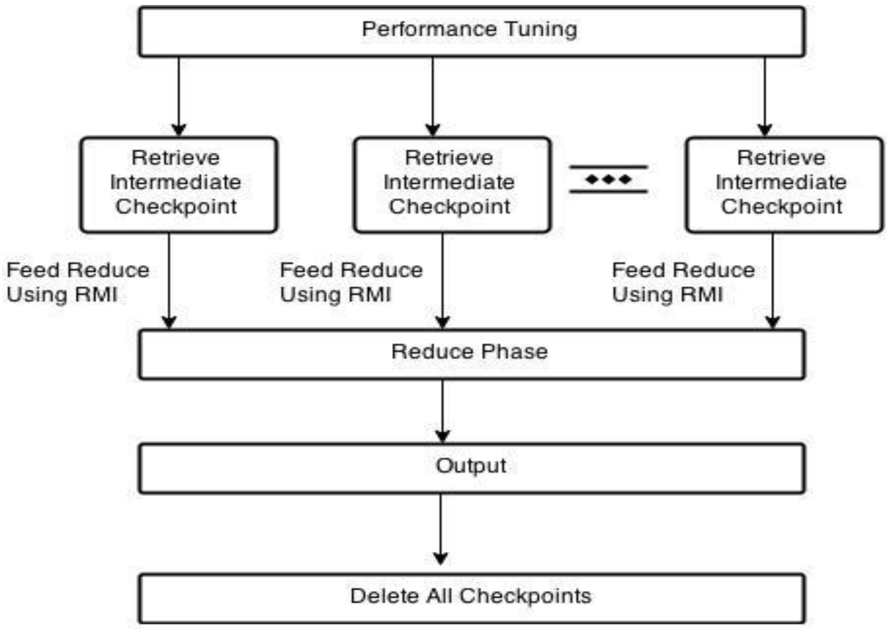


Fig. 3.7 Error recovery at Reduce phase

Pseudo code for deleting all checkpoints:

```
begin cleanCheckpoints()
        LocalFileSystem fs = FileSystem.getLocal(context.getConfiguration());
                fs.delete(inputPath);
end
```

The above pseudo code deletes the all checkpoints created in the program.

CHAPTER IV

RESULTS

4.1 Software requirements:

Operating system: Linux

Total Memory: 16 GB

Number of cores: 16

Number of nodes: 2

The proposed architecture can run for any MapReduce job. The word count program is used as a benchmark to get the following findings.

4.2 Evaluation:

The total time is the sum of running time and recovery time. Running time is the sum of map phase time and reduce phase time. If an error occurs in the application, then the recovery time is the time taken to dispatch the split, map or reduce phase from its last checkpoint.

4.2.1 Calculation of Availability:

Definition: It is the proportion of time a system is functioning [11], which is commonly referred to as “uptime” (vs. downtime, when the system is not functioning). The Hadoop Availability is defined by the following.

$$\text{Availability} = \frac{\text{Mean time to error}}{\text{Mean time to error} + \text{Recovery time}}$$

If t1 is the time taken to fail the maptask1 and t2 is the time taken to fail maptask2 then Mean time to error occur = (t1+t2)/2.

4.2.2 Total time with no error

If no error has occurred in the MapReduce program then the execution of general Hadoop architecture is faster than the new framework. The following table presents the Map-MTE (Mean time to Error occur in map phase in node 1, 2), Reduce-MTE (Mean time to Error occur in reduce phase in node 1, 2), checkpoint time in the MapReduce program with no error in it.

Input(GB)		Map-MTE (ms)	Reduce-MTE (ms)	Checkpoint time(ms)
1	Normal Hadoop	756926	265636	0
	New Hadoop	756926	265636	3987
2	Normal Hadoop	1262358	452024	0
	New Hadoop	1262358	452024	5643
3	Normal Hadoop	1728417	667499	0
	New Hadoop	1728417	667499	7543
4	Normal Hadoop	2473399	862221	0
	New Hadoop	2473399	862221	9765
5	Normal Hadoop	2968194	1039286	0
	New Hadoop	2968194	1039286	13857

Table 4.1 Total time with no error

If the program runs with no error, then conventional hadoop architecture will execute faster than the new proposed configuration as the checkpoint creation will take some time in the MapReduce program. Thus, the total time for new configuration will take more time compare to the conventional hadoop.

The availability in conventional hadoop with any input size is 100%, because there won't be any error occurred in it. Availability with no error is defined as follows.

$$\text{Availability} = \frac{\text{MapMTE+ReduceMTE}}{(\text{MapMTE+ReduceMTE})}*100$$

For 1GB input size, availability in conventional hadoop

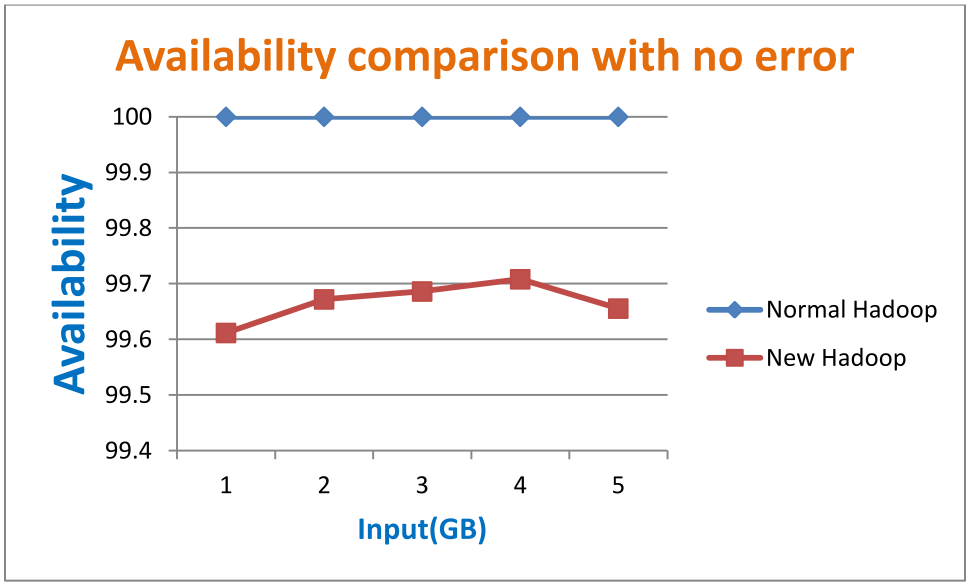
= (1022562 / 1022562)*100 = 100%

For 1GB input size, availability in new hadoop

= (1022562)/(1022562+3987)*100 = 99.61%

In the new hadoop, checkpoint time is added to the recovery process thus the availability has decreased by 0.38% in the new methodology. Similarly, the availability for the remaining input sizes for new methodology can be calculated.

The following graph illustrates the availability between conventional hadoop and new Hadoop with no error in the program. The x-axis is the input size in GB and y-axis is the availability.

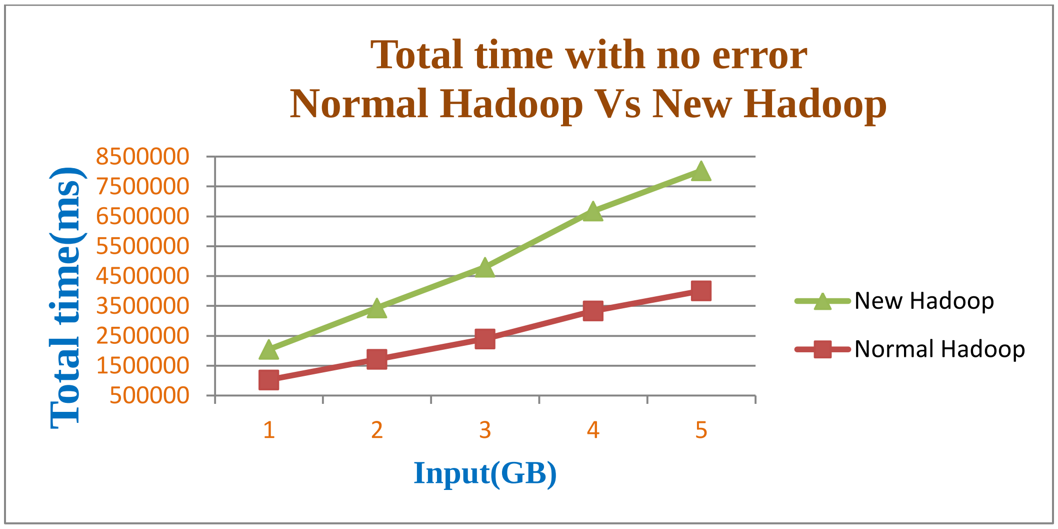


Graph 4.1 Availability comparison with no error

Based on Table 4.1 the following graph is plotted. It compares the total time between normal Hadoop and new Hadoop architecture with no error in the program.

The total time is calculated as the sum of Map-MTE, Reduce-MTE and checkpoint time.

The x-axis is the input in GB and y-axis is the total time in milliseconds.



Graph 4.2 Total time with no error: Normal Hadoop Vs New Hadoop

From the above graph the conventional hadoop executes faster than the new hadoop , as the creation of checkpoint will take few milliseconds in the proposed method.

4.2.3 Total time with error at map phase:

If a memory error occurs in the map phase then all nodes will be shut down and it has to start from the initial checkpoint thus it is not needed to do split operation. As checkpoint has already have split data and downtime will be decreased by not performing split operation.

The following table presents the Map-MTE (Mean time to Error occur in map phase in node 1, map phase time, reduce phase time, and recovery time on running the MapReduce program with error at map phase.

Input(GB)		Map MTE(ms)	map phase time(ms)	reduce phase time(ms)	recovery time(ms)
1	Normal Hadoop	267561	756926	265636	196800
	New Hadoop	267561	698757	265636	5676
2	Normal Hadoop	564323	1262358	452024	391330
	New Hadoop	564323	1098748	452024	6574
3	Normal Hadoop	876564	1728417	667499	290728
	New Hadoop	876564	1437689	667499	8756
4	Normal Hadoop	953259	2473399	862221	667817
	New Hadoop	953259	2097864	862221	10568
5	Normal Hadoop	1564395	2968194	1039286	831094
	New Hadoop	1564395	2563270	1039286	15642

Table 4.2 Total time with error at map phase

$\text{Availability} = \frac{\text{MapMTE}}{(\text{MapMTE}) + (\text{Recovery time})}$ is used in this evaluation.

With an error at map phase the conventional hadoop architecture will terminate the program in the middle and re-execute the flow from the scratch. In the proposed methodology, the execution flow will be rolled back to the initial checkpoint and perform the further operations from there. Thus total time for new framework will take less time compared to the conventional hadoop.

For the input size of 1 GB, the availability in conventional Hadoop is,

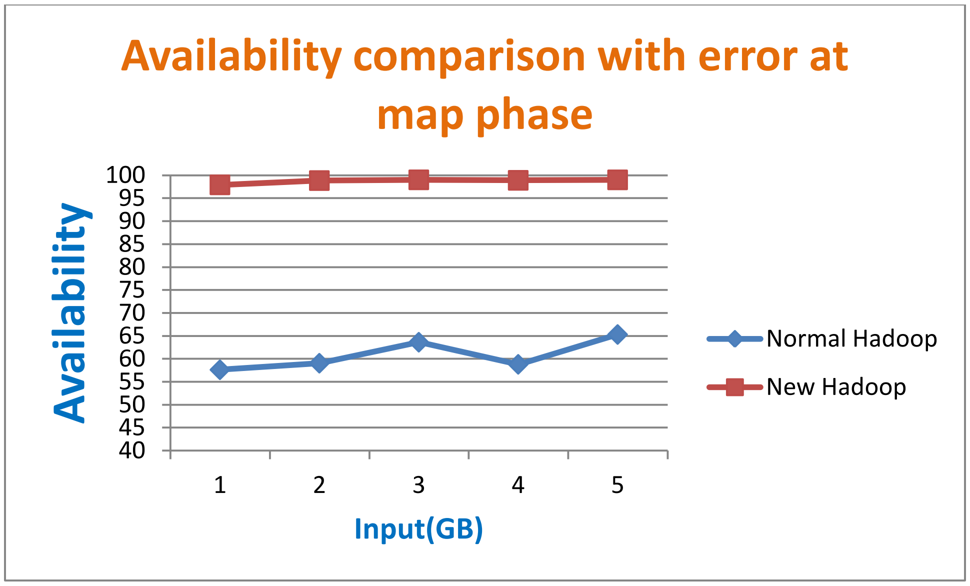
availability = [(267561) / (267561+196800)]*100 =57.61%

and availability in the new Hadoop architecture =

[(267561) / (267561+5676)]*100 = 97.92%

The availability is increased by 40.30%. Similarly, the availability for the remaining input sizes are evaluated.

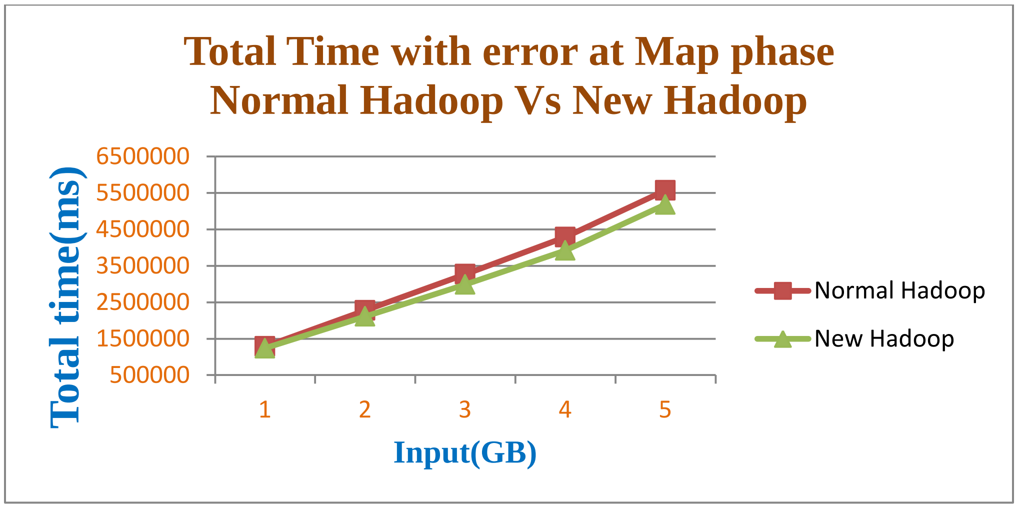
The following graph illustrates the availability between conventional hadoop and new Hadoop with error at map phase. The x-axis is the input size in GB and y-axis is the availability.



Graph 4.3 Availability comparison with error at map phase

From the above graph, for any input size, conventional hadoop availability is lesser compared to the new hadoop availability. This is achieved by decreasing the total running time.

From Table 4.2 the following graph is plotted. It describes the total time between conventional Hadoop and the new Hadoop architecture with error at map phase. The x-axis is the input in GB and y-axis is the total time in milliseconds.



Graph 4.4 Total time with error at map phase: Normal Hadoop Vs New Hadoop

Above graph illustrates the new framework executes faster than normal Hadoop. For any input size the total time in new hadoop architecture is lesser compared to the conventional hadoop.

4.2.4 Total time with error at reduce phase:

If memory error occurs in the reduce phase, then it has to restart its execution from the intermediate checkpoint. Thus, there is no need to do split operation and map phase, because checkpoint has already has split data and map phase output. The downtime will be decreased by not performing split operation and map phase, and downtime= recovery time + reduce phase time.

Therefore, from the above equation,

$$\text{Availability} = \frac{\text{ReduceMTE}}{(\text{ReduceMTE}) + (\text{Recovery time})}$$

The following table presents the Reduce-MTE (Mean time to Error occur in reduce phase in node 1,2), map phase time, reduce phase time, and recovery time on running the MapReduce program with error at reduce phase.

Input(GB)		Reduce MTE (ms)	map phase time(ms)	reduce phase time(ms)	recovery time(ms)
1	Normal Hadoop	678543	756926	265636	783489
	New Hadoop	678543	0	265636	2453
2	Normal Hadoop	1564385	1262358	452024	1266878
	New Hadoop	1564385	0	452024	2987
3	Normal Hadoop	1876897	1728417	667499	1735091
	New Hadoop	1876897	0	667499	39815
4	Normal Hadoop	2564379	2473399	862221	2482021
	New Hadoop	2564379	0	862221	5743
5	Normal Hadoop	3075279	2968194	1039286	2978586
	New Hadoop	3075279	0	1039286	7429

Table 4.3 Total time with error at reduce phase

With 1GB input data, availability in conventional Hadoop%= [678543/(678543+783489)]*100

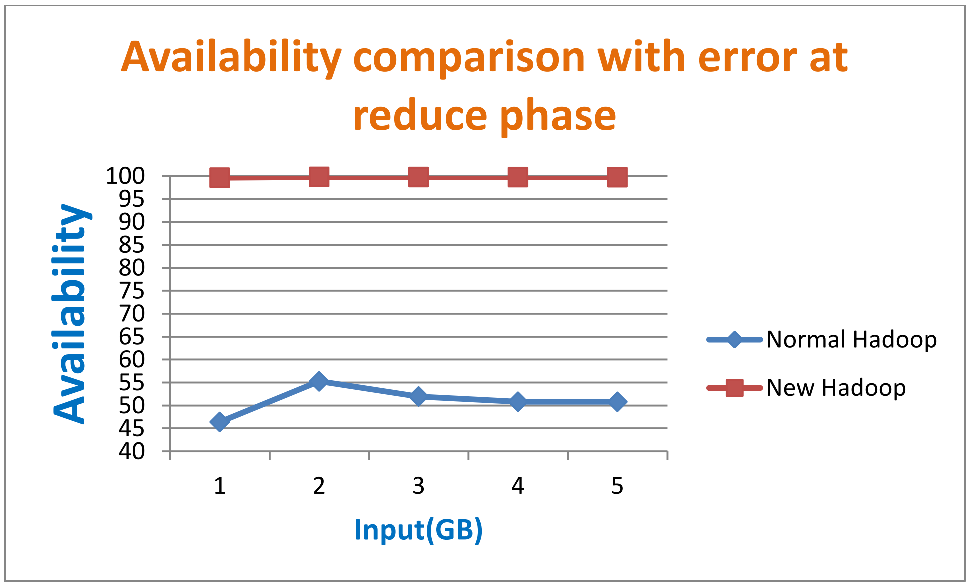
=46.41%

and availability in the new Hadoop architecture = [678543/(678543+2453)]*100

=99.63%

With 1 GB input data, the availability in conventional Hadoop and the new Hadoop are 46.41 % and 99.63 %, respectively. The availability achieved in proposed methodology has 53.22% more compared to the conventional hadoop architecture.

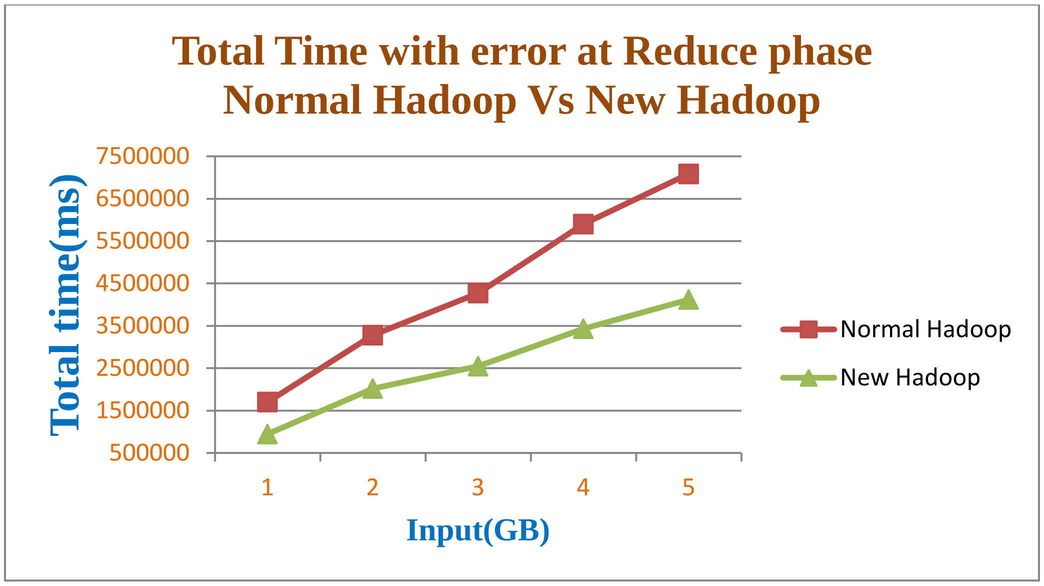
The following graph compares the availability between conventional hadoop and new Hadoop with error at reduce phase. The x-axis is the input size in GB and y-axis is the availability.



Graph 4.5 Availability comparison with error at reduce phase

The above graph illustrates that the availability in the new Hadoop architecture is much higher compared to the conventional hadoop. Actually, map phase will take a lot of time compared to the reduce phase. If an error occurs at reduce phase in conventional hadoop, then it will re-execute the map phase. Thus, the map phase time can be saved in new methodology.

From Table 4.3, the following graph is plotted. It describes the total time between conventional Hadoop and the new Hadoop architecture with error at reduce phase. The x-axis is the input in GB and y-axis is the total time in milliseconds.



Graph 4.6Total time with error at reduce phase: Normal Hadoop Vs New Hadoop

The above graph illustrates the new hadoop executes faster than conventional hadoop. For any input size, the total time in the new hadoop is less compare to conventional hadoop.

4.3 Comparison of existing approaches with new methodology

Property	Existing Approach	New Methodology
Program restart location if error occurred	In Hadoop architecture, if an error occurs in the program it will restart all the way from the scratch i.e., from the map phase regardless of where the error has occurred.	In this methodology, if an error occurs it goes to the nearest checkpoint set in the program and from there the program will restart.
How	In general Hadoop, checkpointing is	In this methodology checkpoint is

checkpoint created	done by using fsimage and edit log [8].	created based on: taskID, a unique task identifier that remains invariant over several attempts of the same task; spillID, the local path to the split data; offset, specifying the last byte of input data processed by split time.
Performance	Checkpointing techniques require the system to replicate intermediate results on stable storage. This can significantly decrease performance as MapReduce jobs often produce large amounts of intermediate results.	The intermediate results are stored in the local system. When a new checkpoint is created it has the previous checkpoint data, then we will delete the old checkpoint. This way unnecessary wasting of systems memory can be avoided.
When checkpoint created	Resilient map task checkpointing tactics(ReCT) is an existing approach to create checkpoints. Users can tell ReCT to create checkpoints at periodic intervals, or every time after certain amount of input data is processed.	Checkpoints are created based on the phase completion. That means less checkpoints in the program can be created.
Downtime	If periodic interval time is very low then ReCT creates lot of checkpoints which yields high downtime.	Downtime reduced by not creating unnecessary checkpoints.
Hadoop availability	If error occurs in MapReduce, then Hadoop availability is less in existing Hadoop architecture.	The Hadoop availability increased by 50 percent by the new methodology.

Checkpoint criteria	Recovery Algorithms for Fast-Tracking (RAFT) MapReduce is another approach to create checkpoint in Hadoop. It creates checkpoints on the task progress computation.	The implementation is different as it is based on the phase level implementation.
Which errors are handled	Rafting method focus on task failures and Master node failures.	The errors which occurred in the MapReduce program.
Performance with no error	Creating too many spills will downgrade performance under less failures, while it won't benefit much from ReCT.	Not many checkpoints will be created compared to ReCT. So if program doesn't have any error it will take more time compared to general hadoop execution and will take less time compared to ReCT.
Running Time	Total running time is high as it has to re-execute from the scratch.	Total running time will be less as it reduces the amount of re-execution.

Table 4.4 Difference between existing approaches and new framework

CHAPTER V

CONCLUSION

A new phase based checkpoint and rollback method has been proposed and implemented with a specific target error, that is the heap space error. Detection and recovery schemes have been developed to address and resolve the heap space errors. For error recovery purpose, the memory size required by the program is raised in the configuration file, then a checkpoint is set, and then the next phases will be executed. As a result of the proposed recovery scheme, the already completed phases need not be re-executed.

From the experimental results, the proposed hadoop availability is increased by 53.22% compared to the conventional and default hadoop configuration, thereby decreasing the running time of the application and ultimately decreasing the downtime as well. Notice that if there is no error in the program then the execution time of the proposed methodology is longer compared to the conventional hadoop architecture, thus availability degrades from 0-10%. This is because the additional operations such as creating checkpoints will lead to the additional time.

As a future work, an orchestration between replications and checkpoint-and-rollback methods could be sought to further optimize the availability.

REFERENCES

Bibliography

[1] D. Jeffrey and G. Sanjay, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary issue: 1958 - 2008 ,* vol. 51, no. 1, pp. 107-113, 01 January 2008.

[2] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-performance computing systems," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, Philadelphia, PA, 2006.

[3] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad and J. Dittrich, "RAFTing MapReduce: Fast recovery on the RAFT," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, Hannover, 2011.

[4] C. Yang, C. Yen, C. Tan and S. Madden, "Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* , Long Beach, CA, 2010.

[5] E. N. Elnozahy, L. Alvisi, Y.-M. Wang and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR),* vol. 34, no. 3, pp. 375-408, 01 September 2002.

[6] C. Memory-Resident, "Checkpointing memory-resident databases," in *Data Engineering, 1989. Proceedings. Fifth International Conference o*, Los Angeles, CA, 1989.

[7] H. Wang, H. Chen and Fei Hu, "ReCT: Improving MapReduce performance under failures with resilient checkpointing tactics," in *Big Data (Big Data), 2014 IEEE International*

Conference on, Washington, DC, 2014.

[8] W. Andrew, "Cloudera," 05 March 2014. [Online]. Available: http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop/. [Accessed 13 February 2015].

[9] ˇ. Vladimir and S. Satish Narayana, "Automated Statistical Approach for Memory Leak Detection: Case Studies," in *On the Move to Meaningful Internet Systems: OTM 2011*, Hersonissos, Crete: Springer Berlin Heidelberg, 2011, pp. 635-642.

[10] E. Aaron, "MAPR," MapR Technologies, 2 1 214. [Online]. Available: https://www.mapr.com/blog/how-to-avoid-java-heap-space-errors-understanding-and-managing-task-attempt-memory#.Vb1duDZRHIU. [Accessed 20 2 2015].

[11] C. Eli, "Apache Hadoop Availability," Cloudera, 10 February 2011. [Online]. Available: http://blog.cloudera.com/blog/2011/02/hadoop-availability/. [Accessed 02 June 2015].

VITA

Nikhil Pendem

Candidate for the Degree of

Master of Science

Thesis: A NEW CHECKPOINT AND ROLLBACK FOR HIGH AVAILABILITY OF MAPREDUCE COMPUTING

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in your Computer Science at Oklahoma State University, Stillwater, Oklahoma in July, 2015.