OCTWAS - Online Check-pointer for Workflows on Apache Spark

by

APARNA BALACHANDRA BHAT

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
October 2015

Copyright © Aparna Balachandra Bhat, 2015
Abstract

With the advent of Web 2.0, Big Data workflows which are being designed in order to extract useful information from data, are run on many engines, one of which is Apache Spark. Apache Spark is a fast and efficient in-memory parallel-processing software framework. By retaining the data in memory, it speeds up the execution of a wide variety of applications. It maintains fault-tolerance by storing information required to recompute the data in case of failure in stable storage. However, for Big Data, recomputing large amounts of data will lead to a performance penalty. In addition, because Apache Spark runs on commodity hardware, the increased likelihood of failure strengthens the need for a fault-tolerance mechanism.

In order to reduce the impact of failures on the processing of these workflows, we propose a framework called OCTWAS (Online Check-pointer for Workflows on Apache Spark). OCTWAS makes use of knowledge of how the data is used in a workflow and takes into account the gain in time obtained when the data is check-pointed at a particular stage and a failure occurs. The framework is designed to be lightweight so that the performance does not degrade when there are no failures. OCTWAS decides the check-points and writes the data to the disks at these check-points while the workflow is being executed. OCTWAS has been tested by simulating failures at fixed intervals. A sample workflow has been designed for Apache Spark for
testing purposes. The performance when the workflow is run with OCTWAS, both with and without failures, is compared to the performance when the workflow is run without OCTWAS, under the same conditions. Results indicate that OCTWAS not only reduces the impact of failures on the processing time of Big Data workflows, but also does not degrade the performance when failures do not occur.
I would like to sincerely thank my professor, Prof. Patrick Martin, for his excellent guidance and support during my master’s degree. In addition to his motivation, patience and immense knowledge, I would like to thank him for his encouragement during the writing of this thesis and for the patient reviews thereafter.

I would like to thank Wendy Powley, for her many reviews which enabled me to complete the writing of this thesis successfully. I am also grateful to her for her encouragement and support during the course of my research. I would like to thank my IBM colleagues and lab mates for their many comments and valuable suggestions.

I would also like to thank Queen’s University, School of Graduate Studies at Queen’s University and the School of Computing for providing me the opportunity to enhance my knowledge and broaden my technical and personal skills by pursuing my master’s degree in computing. I am extremely grateful to IBM Canada Ltd. and NSERC for their financial support to the Database System’s laboratory.

Last but not the least, I am indebted to my family members for their continual support, for their belief in my abilities and for their encouragement to pursue this degree.
Contents

Abstract i

Acknowledgments iii

Contents iv

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Research Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Chapter 2: Background and Related Work 6

2.1 Apache Spark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.1.2 Programming Model . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.1.3 Cluster Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1.4 Apache Spark EcoSystem . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Resilient Distributed Datasets . . . . . . . . . . . . . . . . . . . . . . 13
2.2.1 RDD abstraction . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.2.2 Programming Interface . . . . . . . . . . . . . . . . . . . . . . . 15
2.2.3 Fault-tolerance in RDDs . . . . . . . . . . . . . . . . . . . . . . 16
2.2.4 Architecture of RDD . . . . . . . . . . . . . . . . . . . . . . . . 17
2.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.3.1 Parallel-Processing Frameworks . . . . . . . . . . . . . . . . . . 19
2.3.2 In-memory databases . . . . . . . . . . . . . . . . . . . . . . . . . 20

Chapter 3: OCTWAS 23

3.1 Criticality DAG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
List of Tables

3.1 CriticalityPercentage ........................................ 29
3.2 Factors involved in GainRatio ................................. 30
3.3 SparkFTAlgoParams ........................................... 33

4.1 Standard Deviation of various experiments ....................... 57
4.2 Model1 .................................................... 61
4.3 Model2 .................................................... 61
4.4 Model3 .................................................... 62

A.1 Transformations available for RDDs .......................... 76
B.1 Actions available for RDDs .................................. 78
C.1 Example Data for SparkFTAlgo ............................... 79
# List of Figures

1.1 Example of Analytic Workflows ........................................ 2

2.1 Apache Spark Cluster Mode ............................................. 11
2.2 Apache Spark Eco System .............................................. 12
2.3 Example Lineage in Apache Spark ................................. 17

3.1 Criticality DAG ......................................................... 25
3.2 Architecture of OCTWAS ............................................... 35
3.3 Lineage information of an RDD ..................................... 37
3.4 Criticality DAG - Word Count ...................................... 38
3.5 Information Extraction Unit .......................................... 41
3.6 Class Diagram of OCTWAS ........................................... 43

4.1 TestWorkflow Criticality DAG ........................................ 48
4.2 Amazon EC2 Setup ...................................................... 51
4.3 Status of the Slaves .................................................... 52
4.4 DFS capacity ............................................................ 53
4.5 Status of submitted application .................................... 54
4.6 Spark - Failed Tasks .................................................. 55
4.7 Running TestWorkflow ................................................ 57
Apache Spark is a fast and fault-tolerant in-memory software framework [43]. By retaining the data in-memory, it speeds up the execution of applications that reuse the data such as machine learning algorithms and interactive queries. It manipulates the data through a logical abstraction called the Resilient Distributed Dataset (RDD), which is the primary backbone of Apache Spark [42]. RDDs can be manipulated in a distributed manner and are fault-tolerant.

A workflow is a sequence of operations that consists of repeatable patterns of activities to process information [27]. In this thesis, we examine a specific class of workflows, called analytic workflows. An analytic workflow is the process in which data analysts filter, mine, and exploit the data to obtain useful information for solving scientific or business problems. Big Data analytic workflows are common today due to the increasing amount of data available as a result of the widespread use of Web 2.0.
A sample analytic workflow is illustrated in Figure 1.1. In this workflow, the dataset containing the details of the patients is clustered and a prediction algorithm is run in order to extract useful information from the data. The dataset Patient Details is initially preprocessed and then two separate analyses are performed on the dataset. First, the preprocessed dataset is clustered and then joined with the Demographics dataset after which it is visualized. Second, the preprocessed Patient Details dataset is split into two, the training dataset and the test dataset. The training dataset is used to create a model, which is then scored using the test dataset. The output is
then displayed. This output can then be used to make important decisions related to the patients. Hence, by using various techniques, useful information is obtained from the Patient Details dataset.

A number of different types of failures can disrupt the performance of a parallel-processing framework like Apache Spark, which runs on commodity hardware. Some of these failures include loss of power supply, network failures, and running out of resources. Also, as Apache Spark is an in-memory parallel-processing framework, the volatility of the memory further increases the likelihood of failures, thus increasing the need for a fault-tolerance mechanism.

We propose an algorithm for fault-tolerance for Big Data workflows running on top of Apache Spark. The algorithm, which is lightweight, decides the points at which the data should be written to the disk while the workflows are being executed to reduce the recovery time in the event of failure.

1.1 Motivation

The fault-tolerance in Apache Spark is achieved through the architectural design of RDDs. RDDs are read-only and designed in such a way that they can be computed from data in stable storage in the event that a part of the RDD is lost. Hence, only the information necessary to compute the RDD is stored on the disk, rather than storing the entire RDD.

The nature of the fault tolerance that currently exists in Apache Spark involves maintaining only the information required for computation from stable storage, and not the interim data, to generate the final output(s) of a workflow. In the event of a failure before completing the workflow, the existing provision of fault tolerance in
Apache Spark requires the computations to be performed again from the beginning (and yet again from the beginning if there is another failure before completion). When processing large workflows, this form of fault-tolerance may cause performance to degrade significantly due to the amount of recomputation with more failures.

To enhance the performance for recovery in the presence of failures when using Apache Spark, this thesis proposes an online approach for identifying opportunities to retain interim data at selected points as a workflow is being processed. By writing interim data to stable storage, which is known as check-pointing or persisting the data, recovery from failure will involve a reduced amount of computation using data from a recent check-point, rather than all of the computation from the beginning of the workflow. Check-pointing and any computation to make decisions about check-pointing introduce overhead, and such overhead will be considered in the approach proposed in this thesis.

1.2 Research Statement

The main goal of this research is to design a framework to determine which RDDs in an analytic workflow to persist to the disk while the workflow is being executed such that the recovery time in the case of failures is reduced and that the performance when there are no failures is not affected. The framework designed should be tested in a distributed environment to ensure that the above goals are satisfied by simulating failures.

The research contributions of our work are the following:

1. A new concept that is used to measure the importance of an RDD in an analytic workload.
2. An algorithm, which uses the above-mentioned concept and the gain in time if a particular RDD is check-pointed and a failure occurs, to decide if it is effective to persist an RDD produced during the execution of a workflow.

3. A lightweight online framework that enhances the reliability and reduces recovery times for analytic workflows running on Apache Spark.

4. A proof-of-concept implementation of the framework.

1.3 Thesis Organization

The thesis is organized as follow; Chapter 2 discusses the literature related to this research. Chapter 3 discusses the architecture and implementation details of OCTWAS (Online Check-pointer for Workflows on Apache Spark). Chapter 4 presents a set of experiments to validate the algorithm. Chapter 5 concludes the thesis, describes the limitations of the thesis, and presents ideas for future work.
Chapter 2

Background and Related Work

This chapter discusses the literature related to this thesis. Section 2.1 discusses the motivation that led to the birth of Apache Spark, its architecture, and its programming model. Section 2.2 examines the Resilient Distributed Datasets. Section 2.3 discusses the fault-tolerance mechanisms incorporated in various in-memory frameworks and databases.

2.1 Apache Spark

Apache Spark is an open-source, low-latency, in-memory software framework that was developed at the University of Berkeley. By retaining the data in between various stages of a job in memory, Apache Spark speeds up the execution of applications that repeatedly query the same dataset across parallel operations. The use cases that kindled the creation of Apache Spark are discussed in Section 2.1.1. The programming model that is used in Apache Spark is discussed in Section 2.1.2, and the working of Apache Spark on a cluster is discussed in Section 2.1.3. Apache Spark consists of multiple project components that together make Big Data analytics an easy and efficient process. The various components present in Apache Spark are discussed in
Section 2.1.4.

2.1.1 Motivation

Over the years, Apache Hadoop [37], which is an open-source implementation of MapReduce, has become popular in both industry and academia. Apache Hadoop, enables the processing and storage of data in parallel and is scalable and fault-tolerant. In addition to Hadoop, other systems such as Dryad [14] and MapReduce Merge [39] support other general forms of workflows.

A number of different types of applications benefit from the MapReduce programming model, which is a two-stage process, consisting of the Map phase and the Reduce phase. In the Map phase, the data is sorted or filtered, generating a Key-Value pair. The Reduce phase collects the pairs generated by the Map phase according to the Key and performs some summary operation on the Values of the pairs [37]. However, some applications cannot be expressed effectively using these frameworks. One such application involves the repeated use of a working dataset across various parallel machines. Two of the uses cases are described below [43].

1. A function is applied to the same dataset repeatedly for optimization by certain machine learning algorithms. In the case of a MapReduce job, the data must be reloaded from the disk between each job. This repeated activity of reading and writing leads to a decrease in the performance of the system.

2. Often, the same dataset is queried repeatedly. For example, in Hadoop, ad-hoc queries are run on the same dataset through Hive [34] and PIG [21]. In this case, each query is executed as a job which reads data from the disk, thus incurring a performance penalty.
2.1. APACHE SPARK

Most of the abstractions that are present in the parallel-processing frameworks store data on disks but in the case of the above two use cases which reuse the data, retaining the intermediate data in memory can lead to an increase in performance. Systems that retain data in memory include Pregel [16], which is a graph processing system that retains the data in memory between computations and HaLoop [6], which offers an iterative MapReduce interface. However these systems can only serve specific use cases whereas a framework to support different kinds of applications is required. In order to satisfy this need, Apache Spark is designed to support different kinds of applications that reuse the dataset, while retaining the scalability and the fault-tolerance of the MapReduce platforms.

In the case of the first use case discussed above, Apache Spark retains the data in the memory in between the iterations, thus considerably speeding up the execution of these algorithms. It outperforms Hadoop by about 10 times in the case of iterative machine learning algorithms. For the second use case, the dataset to be queried is loaded into the memory and the users can then query the dataset. The elimination of reading the data from the disk provides an increase in performance. A 39 GB dataset can be queried with a sub-second latency [43].

2.1.2 Programming Model

Apache Spark provides two abstractions for parallel programming: Resilient Distributed Datasets (RDDs) and operations on these RDDs. The RDD is a read-only abstraction that is partitioned across many machines. Apache Spark does not store
the contents of the RDD on the disk, rather the information that is necessary to construct the RDD from a reliable storage is maintained. Hence, an RDD can be reconstructed even if a partition of an RDD is lost. The different methods of construction, the parallel operations that can be performed on the RDDs, and the fault-tolerance mechanisms of RDDs are discussed in Section 2.2.

Apache Spark is implemented in Scala. In order to make use of Apache Spark, the user can either write a driver program in Java, Python, or Scala [3], or use an interactive console, wherein the user can type queries one after another and obtain the results after each query. In the latest version (Apache Spark 1.4.0), support for the R language is introduced. In the case of a driver program, the driver loads the necessary context and launches the operations in parallel [43].

In order to facilitate the running of various programs, Apache Spark contains two restricted types of shared variables. Generally, when a program is run on Apache Spark and a function is called on an RDD, the local variables are copied onto each worker node. The two types of shared variables support some common usage patterns and are described below [43]:

1. *Broadcast Variables*: When a large piece of read-only data is to be used on multiple parallel operations, these variables can be used to copy such data to each of the workers only once.

2. *Accumulators*: These are variables to which different workers can add using associative functions. These variables can only be read by the driver. It is a substitute for the global variable in functional programming.
2.1. APACHE SPARK

2.1.3 Cluster Mode

Apache Spark can be deployed either in the stand-alone mode or as a part of a cluster, wherein it can be deployed on top of cluster managers like Apache Yarn [40] or Apache Mesos [12]. When it is deployed in the stand-alone mode, it makes use of its own Cluster Manager.

The Apache Spark applications run as independent processes on the cluster and are coordinated by a SparkContext object. This object is present in the main program for applications written as a program. For interactive queries, a SparkContext object is created through the interactive terminal, before the queries are issued. The SparkContext object connects with the corresponding Cluster Manager (Apache Spark’s own Cluster Manager or Cluster Managers of Mesos or YARN). All the Cluster Managers allocate resources across the cluster.

Once the SparkContext object connects with the Cluster Manager, it requests and then acquires executors on the various nodes in the cluster. These executors are the processes that perform computations and store data. Once the executors are acquired, Apache Spark sends the code to the executors and assigns tasks to them.
2.1. APACHE SPARK

Figure 2.1: Apache Spark Cluster Mode

The flow of control between the SparkContext and the Cluster Manager is depicted in Figure 2.1. Each Apache Spark application has its own set of executors that are not released until the application comes to an end. Apache Spark can run on any Cluster Manager as long as it is able to acquire the executors and issue commands to it. For this project, Apache Spark has been set up on top of Apache YARN [20].

2.1.4 Apache Spark EcoSystem

As mentioned before, Apache Spark consists of multiple components. The various components are described in this section and depicted in Figure 2.2.

The foundation of the overall project is the Apache Spark core that provides distributed task dispatching, scheduling, and other I/O facilities. It serves as the base platform on which the other components are installed.

Apache Spark provides support for processing structured and semi-structured data using SparkSQL [4]. The RDDs of SparkSQL can be manipulated using a domain specific language. SQL support is also provided either through command-line interfaces
or JDBC/ODBC [10] servers.

Spark Streaming [30] enables the Apache Spark engine to perform fault-tolerant, scalable, and reliable stream-processing [13]. The data is divided into batches and all the transformations and actions are performed on these batches one at a time. Finally the processed data can be pushed onto a file system, database, or even on a live dashboard. This enables the reuse of the code written for batch data processing, but on streaming data.

Figure 2.2: Apache Spark Eco System

Another important component of Apache Spark is MLlib [19]. It is the distributed machine learning framework that implements common machine learning and statistical algorithms. Some of the algorithms implemented include K-Means, Latent Dirchlet Allocation, Singular Value Decomposition, and Feature Extraction.

GraphX is the distributed graph processing system that runs on top of Apache Spark [38]. It enables the user to work with both graphs and collections. The graphs can either be transformed to RDDs or joined with RDDs. It competes in speed with many of the fastest graph processing systems available. It supports a number of
2.2. RESILIENT DISTRIBUTED DATASETS

different graph algorithms including PageRank and Connected Components.

2.2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs) are the primary logical abstractions through which Apache Spark manipulates the data. The RDDs are a collection of objects that are partitioned across several machines so that parallel operations can be performed on these RDDs. The data in Apache Spark is represented as RDDs and these RDDs can be manipulated using a variety of operations provided by the Apache Spark API [42].

The available in-memory storage abstractions such as distributed shared memory, key value stores, and distributed databases make use of fine-grained updates where fault-tolerance is achieved through either replicating the data across several machines or maintaining logs across the machines. All of these techniques pose an overhead for data-intensive applications, as they require the transfer of data across the cluster which monopolizes the network bandwidth. The network bandwidth can become the bottleneck in these systems. These techniques also require more storage overhead [42].

RDDs make use of coarse-grained transformations such as map and filter, wherein the same operation is applied to many data items. Hence, it is not necessary to store the entire dataset on the disk, rather only the information necessary to compute the RDD again, from the original data that exists either on the disk or in-memory, has to be stored. This enables the RDDs to be recovered if lost, without expensive replication.

Unlike systems like Pregel or HaLoop, which are in-memory parallel-processing
systems but tailored for specific use cases, Apache Spark is versatile and can work with most parallel applications. This is owing to the fact that most parallel applications apply the same operations to multiple datasets. It has been shown that RDDs can represent many parallel programming models that have been proposed as separate systems such as MapReduce [37], DryadLINQ [41], SQL [7], Pregel [16], and HaLoop [6], as well as some other programming models such as interactive mining. The ability to represent the programming models of separate frameworks by one abstraction indicates the power of RDDs [42].

Each RDD may be partitioned into one or more partitions, and these partitions are distributed across several machines. The user has the ability to control the number and the locations of these partitions.

2.2.1 RDD abstraction

As mentioned in the previous section, only the lineage information, that is the information necessary to compute an RDD from stable storage, has to be stored. Hence, Apache Spark only stores the lineage information of all RDDs and does not store the entire contents of the RDD on the disk. So even if a portion of an RDD is lost, it can be recomputed again with the help of lineage information from the data in the stable storage.

The RDDs can be created in four ways in Apache Spark:

1. From a file in a shared file system such as the Hadoop Distributed File System (HDFS).

2. From a Scala collection. This collection is partitioned and sent to multiple nodes.
3. By transforming an existing RDD. This is further explained in Section 2.2.2

4. By changing the persistence of an existing RDD. This is further described below.

RDDs are materialized lazily and are ephemeral. RDDs are computed in memory but are not stored on disk after an operation is performed on them. They are materialized on demand when they are used and once that step of materialization is completed, they are discarded. Apache Spark does not analyze whether the RDD being discarded might be reused in the future and hence, does not maintain the RDDs that might be used again in memory. Users have the option of explicitly caching an RDD, that is retaining the RDD in memory for future computations. Users also have the option of explicitly persisting the RDDs on the disk. In this case, Apache Spark writes the RDD to the underlying file system. For example, when Apache YARN is used, Apache Spark makes use of the HDFS.

2.2.2 Programming Interface

RDDs can be manipulated using two types of operations, actions and transformations. Transformations are the operations that transform an RDD from one form to another. For example, consider an RDD called ‘file’, which is created by loading the contents of a file. Suppose we have to filter out the lines which contain the word “Spark”. This is done by the line

```java
JavaRDD<String> ones = file.filter(new Function<String, String>()
{public Boolean call(String s)
{
return s.contains("Spark");

```
Here the RDD ‘file’ is being transformed into another RDD called ‘ones’ using the filter operation. Because RDDs are read-only, the RDD which contains only the word “Spark” is created. The other commonly used transformations include map (returns a new dataset where each element of the source is passed through a function) and union (returns a new dataset which is the union of the elements of two RDDs) [11]. A list of all the available transformations is given in Appendix A.

Actions enable the user to perform certain operations on an RDD. An example of an action is,

```java
int count = file.count();
```

In the above line, in order to count the number of lines in the file, the contents of the file are first loaded into an RDD called ‘file’. Then, the count operation is issued on it.

Each action is considered as a job in Apache Spark. Some of the other actions include collect (all the elements of the dataset are returned as an array to the driver program), take (returns an array with the first \( n \) elements of the RDDs), and first (returns the first element of the RDD) [11]. A list of all available actions is available in Appendix B.

### 2.2.3 Fault-tolerance in RDDs

As mentioned before, an RDD can be computed again from the data in stable storage by making use of the lineage information.
Figure 2.3 illustrates the concept of lineage for an application where the line containing the word “ERROR” is counted. In the figure, the application lineage information of four RDDs is depicted. The first RDD called ‘File’ is constructed by reading in a text file from HDFS. Hence, the path to the text file is stored. The second RDD called ‘Errs’ is constructed by applying the filter function to the first RDD. Similarly, the third RDD ‘CachedErrs’ is a cachedRDD where the cache function is called on the RDD and the fourth RDD ‘Ones’ is obtained by using a Map function on ‘CachedErrs.’ If a failure occurs, the lineage information portrayed in Figure 2.3 is used to reconstruct the RDD in parallel [43].

2.2.4 Architecture of RDD

The design of RDDs require that it should be able to represent a wide variety of operations while maintaining the same interface to the user. Each RDD is represented
through a common interface that has five pieces of information [42]:

1. A set of partitions. These partitions are atomic pieces of the dataset. The partitions of each RDD can be obtained by `partitions()`.

2. A set of dependencies on parent RDDs. The function `dependencies()` returns a list of dependencies.

3. Data Placement Details of the RDD. These details can be used to make decisions for optimization such as accessing the data based on node locality. The function `preferredLocations(p)` lists nodes where partition \( p \) can be accessed more quickly due to data locality. These details are used by Apache Spark for optimizations and can also be used by the user for optimizations.

4. Partitioning Scheme. The details of the partitioning scheme are stored here. For example, the RDD may be range-partitioned or hash-partitioned and this information can be used for optimizations. The function `partitioner()` returns this metadata.

5. Function to compute the RDD from its parents. The function `iterator(p, parentIters)` can be used to compute the elements of partition \( p \) when the parent partitions are given.

For the RDD that stores an HDFS file, `partitions` returns one partition for each block of the file, `iterator` reads the block, and `preferredLocations` returns the list of nodes which contain the block.

For the RDD generated by a map function, the partitions and the preferred locations are the same as that of its parent. In the `iterator` method, the function is applied to each of its parents' records.
2.3 Related Work

In this section, the fault-tolerance mechanisms in some in-memory parallel-processing systems and in-memory databases are discussed.

In Section 2.3.1, we discuss the fault-tolerance mechanisms in some parallel-processing systems. In Section 2.3.2, we present the fault-tolerance mechanisms in in-memory databases.

2.3.1 Parallel-Processing Frameworks

Apache Hadoop, which we discussed in Section 2.1, stores the data on disk between the Map and the Reduce tasks and also between different MapReduce jobs. Apache Hadoop is designed considering that it runs on commodity servers and the chances of the servers failing is high. Hence, different types of fault-tolerance mechanisms are implemented at the HDFS level. These mechanisms include Data Duplication, Checkpoint, and Recovery. The data is duplicated across the various nodes of the cluster. The number of replications can be customized by the user. Another mechanism is the periodic check-pointing of the NameNode, which is the component that holds the details of the locations of various blocks [29].

Twister is a distributed in-memory framework that is optimized for iterative MapReduce computations [9]. The intermediate data between the iterations is retained in distributed memory and all the data transfers and communications between this distributed data is performed by using a publish/subscribe messaging infrastructure. The fault-tolerance in Twister is provided by storing the application state between computations. Hence, in the case of a failure, the entire computation is rolled back only a few iterations [9].
M3R (Main Memory Map Reduce) is an implementation of the Hadoop MapReduce API that retains the data in memory [28]. It retains the Key-Value sequences in a family of JVMs (Java Virtual Machines). These JVMs share heap spaces between the jobs. As a result, only the workflows that fit into the memory of the cluster can be run. M3R does not support any type of resilience. The M3R engine does not recover from node failure. However, the implementation of a mechanism for resilience is in the future road map of the M3R engine.

Pregel, which is a large-scale distributed graph processing system, stores the data in memory for its processing [16]. It makes use of the Bulk Synchronous Parallel model for processing the graphs. In this model, the computations consist of iterations, called supersteps. During each superstep, concurrent computation takes place, followed by communication between the different vertices, and finally barrier synchronization, wherein the nodes vote to be either active or inactive. The fault-tolerance in Pregel is achieved through check-pointing. The master instructs the workers to save the state of their vertex values, edge values, and incoming messages at the beginning of each superstep.

2.3.2 In-memory databases

Redis is an open-source in-memory database system that is a Key-Value cache and store [24]. It is a data-structure server as the keys can be not only strings, but also abstract data types such as hashes, sets, or lists. The entire dataset is retained in the memory in Redis, hence persistence is an important factor. Persistence in Redis is provided by two techniques. The first technique is to take snapshots at fixed intervals, wherein the data is asynchronously transferred from the memory to the disk after a
fixed interval. The second technique is AOF, an Append Only File, in which the operations that are being performed on the data are recorded. By default, Redis synchronizes the data between the disk and the memory at least every 2 seconds. This value can be customized.

SAP HANA is a column-oriented in-memory relational database designed by SAP [25]. Both complex query processing and high transaction rates are handled by the architecture. It has a layer called the persistence layer to handle the persistence details [17]. The persistence layer is responsible for the durability and atomicity of the transactions. It also ensures that the transactions are either completely rolled back or completely executed. The persistence layer makes use of a combination of write-ahead logs, shadow paging, and save points [17]. The default internal value for asynchronous check-points is 5 minutes in SAP HANA.

Altibase is a database that combines both an in-memory system and a disk-based database system to provide high performance with durability and reliability [1]. Altibase provides fault-tolerance by making use of standby systems. Different types of replication models are provided. One of them is the primary-standby mode, where each primary server has a standby server. The database requests are handled by the primary server and these changes are propagated to the standby server. Another model is the active-active model, in which a primary server also serves as a standby server. Hence, there are a number of servers with copies of the same data, which enables load balancing [15].

Datablitz is a general-purpose database that was developed at the Lucent Bell labs [5]. Like traditional systems, it contains a transactional log that reflects the database updates, and when a transaction is committed, the logs are written to the
disk. Periodic check-pointing is also performed where the dirty pages are flushed onto the disk. The writing of the images is alternated between two disks (*ping-pong check-pointing*) and are written in the background (*fuzzy check-pointing*). Both of these check-pointing techniques reduce the performance impact of this writing to disk on the applications. Other features include automatic failure detection and recovery from failures. Protection of database pages from corruption by bad writes is also incorporated [36].

TimesTen is an in-memory relational database from Oracle that runs in the application tier storing all the data in the main memory [23]. Several levels of durability are provided. The transaction log can be customized to be written after every transaction, when the allocated memory is filled by the transactional log, or never. Shadow Paging is also supported wherein the transactional log is transferred to the backup. The backup then updates its local in-memory database with the transactional log. The backup, however, lags from the primary, and recent transactions might be lost if the primary crashes [36].

MemSQL is an in-memory distributed system that is an ACID-compliant system [18]. It combines the horizontal scalability of a distributed system with the familiarity of SQL systems. Durability is implemented in MemSQL by making use of write-ahead logging and snapshots. As soon as the transaction is acknowledged in memory, the database writes the transaction to the disk, as per the default settings. The maximum size of the in-memory database buffer can be configured, and when the buffer is full, MemSQL writes the transactional log to the disk. When MemSQL starts, the data is loaded into the memory asynchronously using the logs and the snapshots [18].
Chapter 3

OCTWAS

In this chapter, the architecture of OCTWAS is discussed in detail. The concepts involved in OCTWAS are explained in Sections 3.1 and 3.2. The architecture and implementation of OCTWAS are explored in Section 3.3.

As explained in Chapter 1, OCTWAS analyses the workflow as it executes, and it check-points RDDs in a manner that reduces the degradation of performance due to failures.

As described in Section 2.3.2, some of the existing in-memory systems make use of a mechanism where the data is written to the disk at predefined fixed intervals. In this case, everything that is computed between these intervals must be written to the disk. However, in the case of Apache Spark, the fault-tolerance mechanism of RDDs (wherein the RDDs can be recomputed from the data in a stable storage using the lineage information) permits the check-pointing of a subset of RDDs. Hence, the need for writing the entire data onto the disks between check-points is eliminated, leading to better performance than the fixed-interval scheme.

Unlike the mechanism of writing the data to the disks at predefined intervals, we have the knowledge of how the data will be used in the future from a particular point.
The workflows are designed in advance and provided to the Apache Spark Engine, therefore we know which RDDs will be reused in the future. We can further reduce the writing of unnecessary data to disk because the RDDs that will not be reused need not be saved.

The time of a check-point must be chosen in such a way that persisting the RDD offers some advantages in case of failure of one of the slaves. On the other hand, the check-pointing mechanism should be lightweight with low overhead.

Unlike Altibase and TimesTen, having two or more systems for back-up is not efficient in our case; the disks of all the machines involved can be used for storing the check-points because the computation occurs entirely in memory. Furthermore, RDDs can be computed from the data in stable storage, hence it is not necessary to store the entire replica of the memory of machines involved in computations.

In OCTWAS, we select the points to check-point the RDD to the HDFS. As mentioned before, Apache Spark runs on top of Apache YARN, and hence the underlying file system is the HDFS. When an RDD is written to the HDFS, it is automatically read from the disk when necessary. In the case of failure, if an RDD in the lineage of the failed RDD is present on the disk, that RDD is read and then the failed RDD is computed from it. On the other hand, if the failed RDD is present in completed form on the disk, it is directly loaded into the memory. Hence, persisting some RDDs should ensure that the recovery time is reduced in the event of failure. In Apache Spark, an RDD can be persisted by making use of the API provided.

OCTWAS enables check-pointing by making use of two concepts: the Criticality DAG (Directed Acyclic Graph) and the algorithm, named SparkFTAlgo (Spark Fault-tolerance Algorithm). The Criticality DAG indicates the importance of that
3.1 CRITICALITY DAG

particular RDD in the workflow whereas SparkFTAlgo makes the actual decision of whether to persist an RDD or not.

3.1 Criticality DAG

The Criticality DAG takes into account the future usage of an RDD. It is a portrayal of how important the RDD is in the workflow structure. The Criticality DAG is built from the provided workflow.

A sample Criticality DAG is illustrated in Figure 3.1. Each vertex represents an RDD. The arrows indicate a dependency between the RDDs. The alphabet on the vertex stands for an id which is used to identify the vertex throughout this section and the number stands for the CriticalityNumber of the vertex. An edge from Vertex
3.1. CRITICALITY DAG

$V_i$ to Vertex $V_j$ means that the RDD represented by Vertex $V_j$ is generated from the RDD represented by Vertex $V_i$. If an RDD is generated from two or more RDDs, then the Vertex representing the generated RDD will have more than one Vertex pointing to it. Also, if many RDDs are generated from the same source RDD, then the vertex representing the source RDD will point to more than one vertex. For example, in Figure 3.1, Vertex C is generated from Vertex A. Vertex E is generated from Vertices B, C and D. Vertex G is generated from Vertices E and F. This might be done by operations such as union or join.

Vertex H is the final outcome of the workflow. In our mechanism, this is considered as the ending vertex and the entire structure is considered a DAG. The DAG may have more than one ending vertex, if there is more than one final job in the workflow.

The CriticalityNumber is one of the factors that is used to indicate how important the vertex is in the workflow. From Figure 3.1, it is clear that Vertex H has a CriticalityNumber of 0 because no vertices depend on it. Similarly, Vertex G has only one vertex that depends on it, Vertex H, thus giving it a CriticalityNumber of 1. Vertices B, C and D have 3 vertices depending on them; Vertices E, G and H. Vertex A has 4 vertices depending on it; Vertices C, E, G and H. Hence it can be seen that this vertex is more critical than the other vertices. The CriticalityNumber of a vertex can be calculated by the algorithm provided in Algorithm 1.
Algorithm 1 Algorithm for Calculating Criticality Number

function CalculateCriticalityNumber(root)
    criticality ← 0
    if root.children.size = 0 then
        root.criticality ← 0
        return 0
    end if
    for child : root.children do
        criticality ← criticality + CalculateCriticalityNumber(child)
    end for
    criticality ← criticality + root.children.size
    root.criticality ← criticality
    return criticality
end function

The relative criticality of a vertex is indicated by the term CriticalityPercentage (CP). The CriticalityPercentage is the CriticalityNumber divided by the total number of vertices in the DAG, expressed as a percentage. The CriticalityNumber alone cannot be used to indicate how important the vertex is in the workflow. For example, in the DAG illustrated in Figure 3.1, the CriticalityNumber of Vertex A is 4. In this DAG, a CriticalityNumber of 4 is more critical than the other vertices. However, for a DAG with 100 vertices, a vertex with a CriticalityNumber of 4 is not as important. Hence, the CriticalityPercentage is the factor that takes this relevancy into account and its calculation is given by Algorithm 2. In Algorithm 2, the variable Number stands for the total number of vertices in the DAG.
Algorithm 2 Algorithm for Calculating CriticalityPercentage

\begin{algorithm}
\begin{algorithmic}
\Function{CalculateCriticalityPercentage}{root}
\State{CriticalityPercentage ← root\textunderscore criticality ÷ Number} \Comment{Number is the total number of Vertices in the DAG}
\State{CriticalityPercentage ← CriticalityPercentage × 100}
\If{root\textunderscore children\textunderscore size = 0}
\State{\textbf{return}}
\EndIf
\For{child : root\textunderscore children}
\State{CalculateCriticalityPercentage(child)}
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

The CP of the different vertices in Figure 3.1 are tabulated in Table 3.1. Vertex H has a CP of 0%. Vertex G has a CP of 12.5%. This means 12.5% of the vertices in the DAG depend on Vertex G. Similarly, Vertex B has a CP of 37.5%, meaning that 37.5% of the vertices in the DAG depend on Vertex B. Finally, Vertex A has a CP of 50%. Therefore, for the DAG depicted in Figure 3.1, Vertex A is more critical than all other vertices and Vertices B, C and D have a medium criticality compared to other vertices. Thus, the CP provides a factor that takes into account the importance of the vertex in that particular DAG.

3.2 SparkFTAlgo

In this section, we discuss the algorithm used by OCTWAS, named the SparkFTAlgo (Spark Fault-tolerance Algorithm). This algorithm decides if the persistence of a
particular RDD reduces the impact of failure of machines on the performance of workflows. The algorithm is executed as soon as each RDD is computed and then if an RDD is to be persisted, it is saved to the disk. The decision taken is based on the concept of the Criticality DAG and the time gained if that RDD is check-pointed and a subsequent failure occurs.

The time potentially gained during recovery by check-pointing a particular RDD is represented by the parameter called GainRatio. GainRatio is calculated by making use of the formula,

\[
GainRatio = \frac{(Time_{rt(cp)} + Time_{rc}) - (Time_{cp} + Time_{rt})}{Time_{cp}}
\]  

(3.1)

The parameters of Equation 3.1 are explained in Table 3.2. The GainRatio indicates the gain in time for recovery when the RDD is check-pointed compared to the amount of time required to compute the RDD, if it is restored from a previous check-point. In other words, it is the difference between the time taken to recompute the RDD from a previous check-point (Time taken to restore the old check-point and

<table>
<thead>
<tr>
<th>Vertex id</th>
<th>CriticalityPercentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>12.5</td>
</tr>
<tr>
<td>F</td>
<td>25</td>
</tr>
<tr>
<td>E</td>
<td>25</td>
</tr>
<tr>
<td>D</td>
<td>37.5</td>
</tr>
<tr>
<td>C</td>
<td>37.5</td>
</tr>
<tr>
<td>B</td>
<td>37.5</td>
</tr>
<tr>
<td>A</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 3.1: CriticalityPercentage
time taken to recompute from the restored check-point) and the time taken for recovery if that particular RDD is check-pointed (Time taken to check-point the RDD now and the time taken to restore the RDD when there is a failure) divided by the time to check-point the RDD.

The time taken to restore the RDD if it is check-pointed is represented by $T_{cp}$ and $T_{rt}$. The factor $T_{cp}$ is the amount of time taken to check-point the RDD whereas $T_{rt}$ is the time taken to restore the current RDD, if it is check-pointed. The time taken to recompute the RDD from a previously check-pointed RDD, in the case of failure, is represented by the factors $T_{rt(cp)}$, which is the amount of time taken for restoring the RDD from the previously check-pointed RDD from stable storage, and $T_{rc}$ which is the amount of time taken to recompute the desired RDD from the restored, previously check-pointed RDD using the lineage information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{rt(cp)}$</td>
<td>Time to restore from the previously check-pointed RDD from stable storage</td>
</tr>
<tr>
<td>$T_{rc}$</td>
<td>Time to recompute the desired RDD from the restored previously check-pointed RDD</td>
</tr>
<tr>
<td>$T_{cp}$</td>
<td>Time to check-point the current RDD</td>
</tr>
<tr>
<td>$T_{rt}$</td>
<td>Time to restore the current RDD if check-pointed</td>
</tr>
</tbody>
</table>

Table 3.2: Factors involved in GainRatio

The GainRatio is calculated immediately after an RDD is computed by Apache Spark. GainRatio represented as percentage is called GainPercent and is the factor used in OCTWAS. It can be calculated using Equation 3.2.
3.2. SPARKFTALGO

\[ GainPercent = GainRatio \times 100 \] \hspace{1cm} (3.2)

For example, for an RDD A, if the \( T_{rt(cp)} \) is 3.5 minutes, the \( T_{rc} \) is 4 minutes, the \( T_{cp} \) is 2.3 minutes and the \( T_{rt} \) is 1.4 minutes, then the GainRatio is 1.65 and is illustrated in the Equation 3.3,

\[ GainRatio = \frac{(3.5 + 4 - 2.3 - 1.4)}{2.3} = 1.65 \] \hspace{1cm} (3.3)

It can be observed that the GainPercent of RDD A is 165%. This implies that the gain in time when we check-point the RDD is about 165%.

For another RDD, RDD B, if the \( T_{rt(cp)} \) is 1.23 minutes, the \( T_{rc} \) is 2.34 minutes, the \( T_{cp} \) is 4.56 minutes and the \( T_{rt} \) is 1.3 minutes, then the GainRatio is,

\[ GainRatio = \frac{(1.23 + 2.34 - 4.56 - 1.3)}{4.56} = -0.50 \] \hspace{1cm} (3.4)

It is observed that the GainPercent of RDD B is -50%, implying that there is a loss of about 50% when the RDD B is check-pointed.

SparkFTAlgo uses the Criticality DAG and the GainPercent to decide which RDD to persist, to reduce the degradation when one or more slaves fail. Algorithm 3 portrays the entire SparkFTAlgo.

In Algorithm 3, GP stands for GainPercent and CP stands for CriticalityPercentage. HLOSS, HGAIN, \( GV_{11} \), \( GV_{12} \), \( GV_{21} \), \( GV_{22} \), \( GV_{31} \), \( GV_{32} \), \( CV_1 \), \( CV_2 \) and \( CV_3 \) are
the threshold variables whose values can be set by the user. In Apache Spark, jobs are actions performed on RDDs. Actions have been described in Section 2.2.2. Jobs may consist of stages. Stages are important RDDs that are part of jobs. A job may either consist of stages or may contain no stages at all. At present the algorithm is run after job RDDs and stage RDDs, due to some restrictions and the decision of either persisting or not persisting is made for each stage or job. The RDDs that are not stages or jobs are used to calculate criticality. For example, in the Figure 3.1, if the vertices B and F are stages or job RDDs, then SparkFTAlgo is invoked after these RDDs are computed. The rest of the RDDs are used only for calculating the CP. When Vertex B is computed, SparkFTAlgo is applied to it, after calculating its GP. It’s CP is 12.5%. When Vertex F has been computed, SparkFTAlgo is applied to it after calculating its GP. Its CP is 37.5%. This is further discussed in Section 3.3.2.
Table 3.3: SparkFTAlgoParams

<table>
<thead>
<tr>
<th>Threshold variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLOSS</td>
<td>-100</td>
</tr>
<tr>
<td>HGAIN</td>
<td>100</td>
</tr>
<tr>
<td>GV11</td>
<td>-25</td>
</tr>
<tr>
<td>GV12</td>
<td>25</td>
</tr>
<tr>
<td>CV1</td>
<td>75</td>
</tr>
<tr>
<td>GV21</td>
<td>25</td>
</tr>
<tr>
<td>GV22</td>
<td>75</td>
</tr>
<tr>
<td>CV2</td>
<td>50</td>
</tr>
<tr>
<td>GV31</td>
<td>75</td>
</tr>
<tr>
<td>GV32</td>
<td>100</td>
</tr>
<tr>
<td>CV3</td>
<td>25</td>
</tr>
</tbody>
</table>

The values of the threshold variables used in Algorithm 3 have been tabulated in Table 3.3. This set has been named SparkFTAlgoParams. SparkFTAlgoParams tries to balance the CP and the GP. From the algorithm and SparkFTAlgoParams, it is clear that if there is a loss of more than 100%, the RDD is not persisted. The re-computation from the previous check-point is considered to be better in this case. Similarly if the gain in check-pointing is more than 100%, persisting the RDD is considered to be profitable. If the GainPercent lies between -25% and 25%, but the CriticalityPercentage(CP) is high (> 75%), persisting the RDD is considered to be worthwhile. Similarly, if the GainPercent lies between 25% and 75%, only RDDs with a CP of 50% or more are persisted. However, if the gain is high and lies between 75%
3.3. ARCHITECTURE OF OCTWAS

and 100%, the RDD is persisted if it has a CP of at least 25%. The SparkFTAlgo-
Params were experimentally chosen as described in Chapter 4. Appendix C provides
a walk-through of the calculations performed by SparkFTAlgo.

3.3 Architecture of OCTWAS

The architecture of OCTWAS and its implementation details are discussed in this
section. The architecture is depicted in Figure 3.2. Apache Spark 1.3.1 is installed
on top of an Apache YARN 2.5 cluster. The Workflow Program is the workflow
written by the user. The Workflow Program must create an instance of OCTWAS,
which implements both the Criticality DAG concept as well the SparkFTAlgo. When
the Workflow Program is submitted to the Apache Spark Engine by the user, Apache
Spark generates the logs. These logs are parsed by OCTWAS to obtain the parameters
needed for computing the factors that are necessary in the SparkFTAlgo. This is
discussed further in Section 3.3.2.
In case SparkFTAlgo decides to persist an RDD, it calls the built-in API on that RDD to persist the RDD to the disk. The RDD is written to the HDFS.

The Criticality Component constructs the Criticality DAG and calculates the CP. The Information Extraction Unit parses the logs in order to extract the parameters needed for SparkFTAlgo, such as the amount of memory occupied by the RDD and the time to compute the RDD. The Algorithm Unit executes the algorithm and the Central Driver Unit stores all the data-structures of OCTWAS. It also coordinates the control flow between the various components.

HDFS replicates the blocks across various machines for fault-tolerance. The replication factor can be customized by the user. If the number of replications of a block goes below the specified level, YARN automatically initiates the creation of replicas to ensure that the replication factor of the block is maintained.
This flow is further described in the subsections that follow. The following subsections consist of a brief description of the different components that are a part of OCTWAS.

### 3.3.1 Criticality Component

The Criticality Component implements the Criticality DAG concept. The DAG is constructed using Apache Spark’s API. The toDebugString() method of the RDD class provides the lineage information of the RDD. An example of the lineage information of an RDD is provided in Figure 3.3.
3.3. ARCHITECTURE OF OCTWAS

In this information, the RDD in the first line is constructed from the RDD in the second line, meaning that the vertex representing the RDD present in the second line points to the vertex representing the RDD present in the first line.

1) MapPartitionsRDD[25] at reduceByKey at WordCountWorkflow.java:247 []

In the above line the RDD being executed is formed by a reduceByKey operation and the type of the RDD is a MapPartitions RDD. The RDD number is 25 and this operation occurs at line number 247 of the file, WordCountWorkflow.java.

Figure 3.3: Lineage information of an RDD
The Criticality DAG constructed for the lineage information in Figure 3.3 by OCTWAS is portrayed in Figure 3.4. In the DAG, the name of the RDD as well as its corresponding CriticalityNumber is displayed on each vertex. The Criticality
3.3. ARCHITECTURE OF OCTWAS

DAG is constructed using java.

### 3.3.2 Information Extraction Unit

The Information Extraction Unit (IEU) parses the logs and fetches the necessary information for calculating the parameters $T_{cp}$, $T_{rc}$, $T_{rt}$, and $T_{rt(cp)}$ for SparkFTAlgo.

In order to calculate $T_{cp}$ (Time to check-point), the amount of memory occupied by the RDD is obtained and converted to MegaBytes (MB). This is multiplied by the time taken to write one MB to disk in HDFS which is obtained experimentally by running the TestDFSIO benchmark [31]. The TestDFSIO benchmark gives the average time taken to read and write one MB to the HDFS. TestDFSIO was run three times on the cluster and the average time taken to read and write one MB to the disk was obtained.

$T_{cp}$ is calculated by Equation 3.5. In order to obtain the amount of memory occupied by the RDD, the RDD has to explicitly be cached. When the RDD is explicitly cached, the details containing the amount of memory occupied by the RDD is written to the logs, which is extracted by the IEU. The effect of this caching is discussed in the Chapter 4.

$$T_{cp} = \text{Amount of memory occupied by the RDD (in MB)} \times \text{Time taken to write one MB to the HDFS (in secs)} \quad (3.5)$$

The amount of time required to calculate the RDD is obtained from the logs. $T_{rc}$ (Time to recompute from the previous check-point) is the sum of the time to recompute all the RDDs from the last check-pointed RDD. If the current RDD has been
check-pointed, then its $T_{rc}$ is set to zero.

$$
T_{rc} = \text{Time Required to calculate the RDD (in secs)} + \\
T_{rc} \text{ of all RDDs from the last check-point (in secs)} \quad (3.6)
$$

$T_{rt}$ (Time to restore the current RDD if check-pointed) is calculated by multiplying the amount of memory occupied by the RDD by the amount of time taken to read one MB from the HDFS (obtained by running the TestDFSIO benchmark). The amount of memory that is occupied by the RDD, that was obtained for the $T_{cp}$, is used for this calculation.

$$
T_{rt} = \text{Amount of memory occupied by the RDD (in MB)} \times \\
\text{Time taken to read one MB from the HDFS (in secs)} \quad (3.7)
$$

$T_{rt(cp)}$ (Time to restore from the previous check-point) is the $T_{rt}$ of the last check-pointed RDD in its lineage. If no RDD has been check-pointed, then the $T_{rt(cp)}$ is initialized to zero.

**Implementation Details**

The IEU is written in Java. It makes use of Apache Tailer [32], which is a library that reads a file as it is being generated and then enables writing functions to process the lines that are being read. Multithreaded Programs have been written to the process the lines that are important. These lines are processed in real time as the Apache Spark logs are being generated.
3.3. ARCHITECTURE OF OCTWAS

The architecture of the IEU is depicted in Figure 3.5. The Memory Extraction thread is the thread that extracts the amount of memory occupied by the RDD whereas the Time Extraction thread extracts the time it takes to compute the RDD.

3.3.3 Algorithm Unit

The Algorithm unit decides whether to persist the RDD or not, after the RDD is computed by Apache Spark. The SparkFTAlgo is implemented in this unit. This unit interacts with the Criticality Component and the Central Driver Unit to obtain the parameters and uses them in the SparkFTAlgo to decide whether or not to persist the RDD. The decision is sent to the Central Driver Unit.
3.3.4 Central Driver Unit

The Central Driver Unit is the main driver of OCTWAS. Both the Criticality Component and the Information Extraction Unit store the results of their computations in the Central Driver Unit. It sends the signal to the Workflow Program to persist the RDD and coordinates the activities between all the components.

3.3.5 Overall Implementation Details

The overall class diagram of OCTWAS is illustrated in the Figure 3.6. The PersistRDDs is an interface that contains two methods: cache and persist. The Workflow Program that is implemented by the user inherits the PersistRDDs interface and implements the cache and persist methods. OCTWAS also has an instance of PersistRDDs through which it manipulates the Workflow Program, as the Workflow Program is an instance of PersistRDDs.

The FTDriver class is the implementation of the Central Driver Unit. FTDriver maintains a list called the RDDList, which is a hashmap of Integer and rddData. All the RDD details in the FTDriver are maintained via rddData. The rddData is the class which stores all the information related to the RDD in FTDriver. For each RDD, an rddData object is created and the information related to the RDD, that is necessary for the FTDriver, is stored in this object. Finally the object is pushed in a hashmap that maps the RDD number to the rddData. Any function that further needs to work with these details accesses the details through this hashmap in the FTDriver.

The CDAG variable in the FTDriver is where the Criticality DAG is stored. The
Algorithm class that makes use of the Criticality DAG concept uses this data structure.

Figure 3.6: Class Diagram of OCTWAS

The Information Extraction Unit is represented by the classes, TailerCall, MyTailerListener, processAddedRDD and processFinishedRDD. The FTDriver class creates the Tailer class, that then creates the listener. This listener listens to the log file and creates the objects of the classes, ProcessAddedRDD and ProcessFinishRDD which inherit from the thread class. The processAddedRDD obtains the amount of memory occupied by the RDD whereas the ProcessFinishedRDD obtains the time taken to finish the RDD. The FTDriver instance is passed onto these classes to manipulate the data.
The Criticality Component is represented by the CDAG, Vertex and CriticalityDAG classes. The CDAG and the Vertex classes are used to represent the Criticality DAG. The construction of the DAG, the calculation of the CriticalityNumber and the CriticalityPercentage is done by the CriticalityDAG class.

Finally, the Algorithm unit is represented by the class SparkFTAlgo. It makes use of the data structures present in the FTDriver class for executing the SparkFTAlgo.
Chapter 4

Evaluation

This chapter describes the evaluation of OCTWAS. The evaluation methodology used is described in Section 4.1. Section 4.2 describes the sample workflow that is used for evaluating OCTWAS. Section 4.3 discusses the experimental setup of Apache Spark, Apache YARN, and OCTWAS. Section 4.4 provides the results of the experiments along with a discussion of these obtained results.

4.1 Methodology

In order to test the performance of OCTWAS, we use methodology inspired by Toshniwal et al. [35] who designed a system to test the fault-tolerance of Twitter Storm, which is a distributed stream-processing system. In this methodology, some of the machines on which Storm was set up were shut down at regular intervals to simulate failure.

In order to evaluate the effectiveness of OCTWAS, we simulated the failure of some of the machines in the cluster at regular intervals. This was done by shutting down machines at regular intervals and recording the time taken to run the Workflow Program. The results obtained are compared to the same failure scenarios where the
Workflow Program was run without OCTWAS. Through this comparison, we show the difference in execution time with the usage of OCTWAS when the Workflow Program is run. Thus, this enables us to evaluate the success of OCTWAS.

In order to test the system, a sample workflow was designed. The workflow is named TestWorkflow for this discussion. TestWorkflow is the implementation of a part of the sample analytic workflow illustrated in Figure 1.1. It is described in detail in Section 4.2.

TestWorkflow is run with different sets of values for the threshold variables in SparkFTAlgo and the results obtained are presented in Section 4.4.

4.2 TestWorkflow

TestWorkflow is based on the data provided by the South East Local Health Integration Network (SELHIN). The data contains details of patients, their diagnosis, admittance codes, etc. The dataset is 703.8 MB in size and contains 272,862 records and 26 columns. The dataset represents the class of applications that contain the real world data where data is not simulated, but collected from patients. The dataset has inconsistencies in that some of the columns do not contain values and some columns contain NULL values. Hence the data needed a preprocessing and cleaning stage, which is typical of many classes of datasets available. TestWorkflow clusters the details of the patient based on the diagnosis and the details of each cluster are written to a separate text file. TestWorkflow also tries to predict whether the patient was admitted to hospital.

As mentioned before, TestWorkflow is an implementation of the workflow described in Chapter 1. Only one of the analyses of the workflow, illustrated in Figure
4.2. TESTWORKFLOW

1.1, has been implemented. In TestWorkflow only the analysis in which the Patient Details dataset has been preprocessed, followed by clustering and joining with the Demographics dataset has been implemented.

As we are interested only in evaluating the performance of OCTWAS and hence the workflow completion, the details of the workflow results are unimportant here. The goal of the workflow for testing purposes is to have a long lineage, so that the impact of OCTWAS is more visible. A long lineage implies an increase in the probability of failures, providing more opportunities for OCTWAS to demonstrate its benefits. The Criticality DAG, which consists of 35 vertices, of TestWorkflow is depicted in Figure 4.1 in which each vertex contains the name of the vertex and its corresponding CriticalityNumber. In TestWorkflow, FormattedCluster0, FormattedCluster1 and FormattedCluster2 are the final jobs. It can be seen that all the jobs have common RDDs among them.

In TestWorkflow, the data is initially read from the HDFS. After this, the preprocessing on the data is performed. Records containing a NULL value in any field are removed. This is followed by removing records which do not have enough fields and then removing records with values that do not match their column’s data type. As the clustering is performed by using K-means which works with only numerical data, the string data type is converted to numerical values. The data is then clustered using the K-means algorithm followed by running the random forest algorithm for prediction.
Figure 4.1: TestWorkflow Criticality DAG
4.3 Experimental Environment

For the evaluation, Apache YARN 2.5 [40] was set up on top of Amazon EC2 (Elastic Compute Cloud) [8]. Apache Spark 1.3.1 was set up on top of YARN. The virtual machine instances chosen were of type, m3.medium and had the following configurations: one virtual CPU, 3.75 GB RAM and 4 GB SSD (Solid State Drive) Storage. The processor is a high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processor or Intel Xeon E5-2670 (Sandy Bridge) Processor running at 2.6 GHz. The m3.medium instances balance compute, memory, and network resources [2].

The cluster consists of one master and 10 slave instances. The number of slaves needed to run the workflow was experimentally determined. Initially, TestWorkflow was run with 4 slaves, for which YARN threw an exception stating that the minimum number of HDFS blocks was not satisfied. The same exception was thrown when the number of slaves was increased up to 7. When the number of slaves was increased to 8, YARN threw an exception stating that there was not enough resources. Hence, the number of slaves chosen was 10.

TestWorkflow was submitted on the master which also contains the code to shut down the instances. The shutting-down was done by a Java program that made use of the Timer class to schedule the shut-down function. Password-less communication was set up between the master and the slaves for this purpose. The shutting-down was done by issuing the shut-down command via SSH [22].

Each time four slaves were shut down. The number four was chosen, because it was the maximum number of slaves that could be shut down before HDFS threw an exception stating that it ran out of disk space. The slaves 7, 8, 9 and 10 were shut down at intervals of three minutes each. The results obtained when one, two and
three slaves were shut down were very similar to the ones obtained for shutting down four slaves and hence only the results obtained when four slaves were shut down are discussed in this chapter. The number of replicas for each block on the HDFS was configured to be two.

Each set of experiments was run three times to account for the unpredictable EC2 performance and the average was taken [26]. Some of the screen shots related to the experimental set-up are available in Figures 4.2, 4.3, 4.4, 4.5 and 4.6.

Figure 4.2 is a screen shot of Apache Spark on the Apache YARN cluster, set up on Amazon EC2. There is one master instance and ten slave instances.

Figure 4.3 shows that all ten nodes in the YARN cluster are in-service.

Figure 4.4 shows the configuration details of the HDFS in our cluster. This was accessed through the NameNode of the master.

Figure 4.5 shows the status of all the submitted applications on the YARN cluster. The implementation of TestWorkflow was called NACRSAnalysis. It can be seen that NACRSAnalysis is a Spark application. It can also be seen from the cluster metrics, active nodes and lost nodes, that four nodes were shut down when the screen shot was taken and that there were 6 active nodes.

Figure 4.6 shows that for Job id 43, two tasks have failed when some of the slaves were shut-down. This can be observed from the status of the Tasks where it displays that 2 out of 13 tasks have been completed with 2 failed tasks.

### 4.4 Validation

The following experiments were run to evaluate the performance of OCTWAS. TestWorkflow was run without OCTWAS, with and without failures. This was done to
4.4. VALIDATION

Figure 4.2: Amazon EC2 Setup
Figure 4.3: Status of the Slaves
### Summary

Security is off.
Safe mode is off.
74 files and directories, 110 blocks = 312 total filesystem objects.

Heap Memory used 74.61 MB of 315 MB Heap Memory. Max Heap Memory is 889 MB.
Non Heap Memory used 36.64 MB of 37.19 MB Committed Non Heap Memory. Max Non Heap Memory is 234 MB.

<table>
<thead>
<tr>
<th>Configured Capacity:</th>
<th>77.49 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS Used:</td>
<td>30.25 GB</td>
</tr>
<tr>
<td>Non DFS Used:</td>
<td>31.03 GB</td>
</tr>
<tr>
<td>DFS Remaining:</td>
<td>14.12 GB</td>
</tr>
<tr>
<td>DFS Used %:</td>
<td>33.04%</td>
</tr>
<tr>
<td>DFS Remaining %:</td>
<td>20.92%</td>
</tr>
<tr>
<td>Block Pool Used:</td>
<td>30.25 GB</td>
</tr>
<tr>
<td>Block Pool Used %:</td>
<td>33.04%</td>
</tr>
</tbody>
</table>

DataNodes usage % (M/Median/M/Median/StdDev): 25.14% / 40.77% / 51.39% / 8.72%

Live Nodes: 12 (Decommissioned: 0)
Dead Nodes: 0 (Decommissioned: 0)
Decommissioning Nodes: 0
Number of Under-Replicated Blocks: 0
Number of Blocks Pending Deletion: 0

Figure 4.4: DFS capacity
Figure 4.5: Status of submitted application
### Figure 4.6: Spark - Failed Tasks

<table>
<thead>
<tr>
<th>Job Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Stages: Succeeded/Total</th>
<th>Tasks (for all stages): Succeeded/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td></td>
<td>2015/06/30 18:09:21</td>
<td>14 min</td>
<td>0/0</td>
<td>0/2 (2 failed)</td>
</tr>
</tbody>
</table>

**Active Jobs (1)**

**Completed Jobs (43)**

*Note: The image shows a screenshot of the EC2 Management Console with details of completed jobs, including job IDs, descriptions, submitted times, durations, stages, and tasks.*
obtain the recovery time when OCTWAS was not used. Finally, the values of the threshold variables of SparkFTAlgo were modified to study the effect on the performance of OCTWAS.

This section discusses the results obtained from the above experiments. Section 4.4.1 discusses the results of the experiments where TestWorkflow was run with and without OCTWAS, with no failures followed by a comparison where there are failures. Section 4.4.2 discusses the results of the experiments where the values of the threshold variables of SparkFTAlgo have been changed.

4.4.1 Base experiments

In this experiment, we compare the execution time of TestWorkflow without OCTWAS, without failures and with simulated failures, to the execution time with OCTWAS in the same scenarios. OCTWAS makes use of SparkFTAlgoParams in the experiments described in this section.

Initially, TestWorkflow was run without any simulated failures on the above specified Spark-YARN cluster. TestWorkflow did not use OCTWAS when this experiment was performed. Then TestWorkflow was run again, this time shutting down four instances at time intervals of 3 minutes each. The above experiments were repeated, this time with OCTWAS. All the experiments were repeated three times and the average results of these experiments are portrayed in Figure 4.7.
When the workflow was run, without the help of OCTWAS and without any failures, the average time was 33 minutes and 10 seconds. On the other hand, when there were failures, the average time was 44 minutes and 36 seconds. The difference in time, when Spark was run without any failures and when Spark was run with failures is 11 minutes and 26 seconds. This can be considered as the recovery time.

When there are no failures, TestWorkflow required an average time of 19 minutes and 15 seconds, whereas with failures, TestWorkflow required an average time of 24 minutes and 25 seconds. Thus with OCTWAS using SparkFTAlgoParams, the recovery time is about 5 minutes and 10 seconds. Compared to the time taken for
recovery without OCTWAS, which took about 11 minutes and 26 seconds, there is a gain of about 50%. The standard deviation of the results have been tabulated in Table 4.1.

The above results are again charted in the Figure 4.8. This time the runs without OCTWAS and the runs with OCTWAS have been placed together.

![Figure 4.8: Running TestWorkflow - Different Placement](image)

The gain in performance when TestWorkflow is run without failure may be due to the fact that, in order to obtain the amount of memory occupied by the RDDs, some of the RDDs are cached. It can be seen that caching is harmless in the case of the sample workflow used as it does not degrade the performance of the workflows, instead it leads to performance gains.

To test this theory, we ran TestWorkflow with OCTWAS using SparkFTAlgoParams and obtained the list of persisted RDDs. TestWorkflow was run again this time persisting the RDDs, obtained previously, manually in the program. Failures were simulated. This way, we were able to repeat the above experiment but eliminate the caching of RDDs. The results obtained are charted in Figure 4.9.
From Figure 4.9, we note that running TestWorkflow with the persistence of RDDs suggested by OCTWAS is compared to the case where TestWorkflow is run both with and without OCTWAS, but with failures. Compared to running the workflow without OCTWAS and with failures which required an average time of 44 minutes and 36 seconds, persisting the RDDs suggested by OCTWAS took an average time of 28 minutes and 39 seconds (with a standard deviation of 0.43). This means that using OCTWAS even without the benefits of caching provides a performance boost of 35.76% when there is a failure. This can be attributed to the benefits of the algorithm. When the workflow is run with OCTWAS using SparkFTAlgoParams with the caching, it gives a performance boost of 45.25%. Even when there is no failure, OCTWAS actually increases the performance by 42.14%, thus proving that OCTWAS using SparkFTAlgoParams achieves at least the same performance as Spark, if not better, in the case of TestWorkflow.

It can be seen that caching actually boosts the performance when TestWorkflow is executed. However, this cannot be guaranteed in all kinds of workflows and further
experimentation is required to study the benefits of caching. In the case of Test-Workflow, all the jobs have common RDDs between them. If caching is not done, the same RDDs will be recomputed whenever a job is executed, as the RDDs are lazily materialized and discarded after their usage at that point. When some RDDs are cached, Apache Spark can make use of these cached RDDs instead.

The average time taken by TestWorkflow to run will change when the experimental setup is changed. This includes a change in the number of slaves, the steps involved in TestWorkflow, the amount of data used, the amount of memory available to Spark and the amount of machines shut-down. However, the performance when OCTWAS is used and when there is a failure should be better than the performance when OCTWAS is not used and there is a failure.

4.4.2 SparkFTAlgo Threshold Variables

SparkFTAlgo makes use of two main parameters; GainPercent (which is the gain in time obtained if the current RDD is check-pointed) and the CriticalityPercent (which indicates how important the RDD is in the workflow). In order to obtain the results when different values are used for the threshold variables in SparkFTAlgo, the values of the threshold variables in SparkFTAlgo have been changed. Each set of values have been called Models.

Tables 4.2, 4.3 and 4.4 show the experimental values used in this study. For the set of values in Table 4.2, which has been called Model1, the CP for persisting has been decreased. For the set of values in Table 4.3, which has been called Model2, the CP for persisting has been increased compared to SparkFTAlgoParams. Finally for the set of values in Table 4.4, which has been called Model3, the negative GP has
4.4. VALIDATION

<table>
<thead>
<tr>
<th>Threshold variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HLOSS$</td>
<td>-100</td>
</tr>
<tr>
<td>$HGAIN$</td>
<td>100</td>
</tr>
<tr>
<td>$GV_{11}$</td>
<td>-25</td>
</tr>
<tr>
<td>$GV_{12}$</td>
<td>25</td>
</tr>
<tr>
<td>$CV_1$</td>
<td>50</td>
</tr>
<tr>
<td>$GV_{21}$</td>
<td>25</td>
</tr>
<tr>
<td>$GV_{22}$</td>
<td>75</td>
</tr>
<tr>
<td>$CV_2$</td>
<td>25</td>
</tr>
<tr>
<td>$GV_{31}$</td>
<td>75</td>
</tr>
<tr>
<td>$GV_{32}$</td>
<td>100</td>
</tr>
<tr>
<td>$CV_3$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: Model1

<table>
<thead>
<tr>
<th>Threshold variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HLOSS$</td>
<td>-100</td>
</tr>
<tr>
<td>$HGAIN$</td>
<td>100</td>
</tr>
<tr>
<td>$GV_{11}$</td>
<td>-25</td>
</tr>
<tr>
<td>$GV_{12}$</td>
<td>25</td>
</tr>
<tr>
<td>$CV_1$</td>
<td>85</td>
</tr>
<tr>
<td>$GV_{21}$</td>
<td>25</td>
</tr>
<tr>
<td>$GV_{22}$</td>
<td>75</td>
</tr>
<tr>
<td>$CV_2$</td>
<td>75</td>
</tr>
<tr>
<td>$GV_{31}$</td>
<td>75</td>
</tr>
<tr>
<td>$GV_{32}$</td>
<td>100</td>
</tr>
<tr>
<td>$CV_3$</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 4.3: Model2

been ignored. The GP between 25% and 50% should have a CP of at least 75%.

The rest of the values are held constant in all the Models.

The results obtained from Model1, Model2 and Model3 are shown in Figure 4.10. Only the average time from three runs has been depicted.
Figure 4.10: Comparison of Different Models

From Figure 4.10, we see that with a decrease in CP, the time taken to run TestWorkflow without failure increases. In this case the number of RDDs being written to the disk increases, thus increasing the time taken to write these RDDs to the disk. On the other hand, there is not much difference in time when there are failures. This might be because all the important RDDs have been persisted when using SparkFTAlgoParams.

The time taken to run Model2, where the CP has been increased when compared
4.4. VALIDATION

to SparkFTAlgoParams, without failures decreases, but the time taken to recover from failures increases. In this case, we may miss persisting critical RDDs. This algorithm can be used when the failure rate of machines is very low.

Finally for Model3, where negative GP has been ignored, it can be seen that the time taken to run without failures is a little higher. However, the time taken to recover from failures is considerably higher. This shows that important RDDs may have lower GainPercent and it is worthwhile persisting these RDDs.
Chapter 5

Conclusion and Future Work

With the decreasing cost of memory, in-memory systems are becoming more popular and widespread. Apache Spark is an in-memory system that manipulates the data through RDDs. Although Spark has its own built-in fault-tolerance allowing a lost portion of data to be computed again, with Big Data workflows, recomputing a single dataset may cause performance penalties. In order to overcome this, we develop an algorithm and framework to persist data in order to decrease recovery time, if necessary. In addition, the check-points should be chosen and the data should be written to the disk while the workflow is being executed.

5.1 Conclusion

Hence, the contributions of this thesis are:

1. A new concept called the Criticality DAG which indicates the importance of an RDD in the workflow.

2. An algorithm, called the SparkFTAlgo, that uses the Criticality DAG and the
5.1. CONCLUSION

GainPercent, which calculates the advantage in time gained if an RDD is persisted and a failure occurs, to decide which RDDs to persist to reduce the recovery time of a workflow, when a failure occurs.

3. A light-weight online framework, called OCTWAS, that enhances the fault-tolerance when analytic workflows are run on Apache Spark.

4. Implementation of OCTWAS and the study of its performance by simulating failures at fixed intervals.

OCTWAS persists selected RDDs while the workflow is being processed. In case, an RDD is to be persisted, OCTWAS writes the RDD to the HDFS, from which it can read the RDD in case a failure occurs.

OCTWAS is designed to be lightweight and its inclusion in a workflow is a simple task. It is designed as a component separate from Spark and has not been integrated with the source code of Apache Spark.

OCTWAS is evaluated with the help of a sample workflow which uses health care data. Apache Spark has been set up on top of Apache YARN. The Apache YARN cluster has been set up on top of Amazon EC2 in a distributed manner.

Failures of machines are simulated and the results evaluated. The performance when OCTWAS is used is compared to the performance when OCTWAS is not used when failures occur and do not occur, under the same conditions. We observe performance improvements using OCTWAS when there are failures and also when there are no failures, in the case of the sample workflow used. In order to obtain some parameters necessary for SparkFTAlgo, some of the RDDs were cached. For the workflow used, caching improves the performance even when failures do not occur.
5.1. CONCLUSION

This improvement in performance due to caching cannot be guaranteed for all workflows. In the case of workflows where Spark might be recomputing the data due to its lazy materialization, caching enables Spark to make use of the already computed RDDs present in the memory. However, when Spark runs out of space, it will start replacing the RDDs that have been cached in order to make space for new RDDs. In this case, a performance improvement might not be there. Rather Spark will have to recompute the RDD just as in the case when the RDD is not cached.

Experiments are conducted to study the performance of OCTWAS without caching and OCTWAS reduces the recovery time when failures occur in this case, leading to the conclusion that the concepts of CriticalityDAG and SparkFTALgo, which makes use of SparkFTAlgoParams, enable the selection of RDDs that reduce the recovery time, without the help of caching. As a part of the evaluation, different sets of values are used for the threshold variables in SparkFTAlgo and the results obtained show that among the various sets of values the ones given in Table 3.3, provide the best results. When the CriticalityPercentage values were reduced, it was seen that the time taken without failures increases compared to SparkFTAlgoParams but the time when there are failures is almost the same. When the CriticalityPercentage values were increased, even though there was a decrease in time when there are no failures, the time taken with failures increases indicating that some RDDs, which contributed to the reduction in recovery time, are missed. Also, when the RDDs with negative GainPercent are ignored, the recovery time increases. Hence, among the various set of values for the threshold variables, SparkFTAlgoParams provides good results. However, if the rate of failure of machines is lower or if a lower level of fault-tolerance is needed, Model 2, where the time taken without failures is lower, can be used. The
relative performance of OCTWAS does not change when the number of slaves, that were shut down to simulate failure, changes.

Although OCTWAS can be applied to any type of workflow, it is especially useful for Big Data workflows where repeated computations cause performance penalties due to the size of the data being recomputed and in workflows where Apache Spark might be repeatedly recomputing the data as a result of its lazy materialization. Hence, OCTWAS reduces the time taken for recovery when there are failures and does not degrade the performance when there are no failures, in the case of the sample workflow used.

5.2 Limitations and Future Work

The current implementation of OCTWAS works only with Spark Version 1.3, as the DebugString that has been used for the construction of the Criticality DAG changes from version to version. Also, the current implementation of OCTWAS supports only some transformations of Spark such as union, join, map, mapToPair, and filter. The parsing and construction of the Criticality DAG is implemented only for these operations. OCTWAS can be extended to support the rest of the operations.

OCTWAS currently only caches the jobs and stages. An algorithm to analyse which RDDs to cache will lead to better performance gains. OCTWAS only analyses the RDDs that are present in the program written by the user and does not analyse the RDDs that are present in the libraries. This can again be overcome by integrating OCTWAS with the source code of Spark.

The performance effect of OCTWAS has to be studied in detail with different types of workflows and using different experimental setups. Whether caching might
have a negative effect on the performance of the workflow has to be investigated. Also the parameters under which caching provides a positive effect should be studied in depth. The effect of the available memory, available disk, and the number of replicas being written to the HDFS, on the performance of OCTWAS have to be explored.

In the evaluation, when the effect of the values of the threshold variables are studied, the values are increased or decreased by 25%. The effect of a smaller increase or decrease on the performance of SparkFTAlgo has to be studied in depth.

The possibility of including different types of concepts such as the probability of the failure of machines, the rate of the failure of machines, and the amount of available memory in SparkFTAlgo, has to be studied.
Bibliography


Appendix A

Transformations

In this appendix, all the transformations that are available in Spark have been listed in Table A.1.
<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>map ((\text{func}))</td>
<td>Each element of the dataset is passed through the function (\text{func}).</td>
</tr>
<tr>
<td>filter((\text{func}))</td>
<td>Returns a new RDD that contains only the elements for which true is returned when processed by the function (\text{func}).</td>
</tr>
<tr>
<td>flatMap((\text{func}))</td>
<td>Very similar to map but each element is mapped to 0 or more elements.</td>
</tr>
<tr>
<td>mapPartitions((\text{func}))</td>
<td>Similar to map. Difference is the fact that (\text{func}) is run on each partition</td>
</tr>
<tr>
<td>sample((\text{withReplacement, fraction, seed}))</td>
<td>Sample a fraction of the data, with or without replacement, using a random number generator seed.</td>
</tr>
<tr>
<td>union((\text{otherDataset}))</td>
<td>Returns a new dataset that is the union of the given dataset along with \text{otherDataset}.</td>
</tr>
<tr>
<td>intersection((\text{otherDataset}))</td>
<td>Returns a new dataset that contains the common elements in the current dataset and \text{otherDataset}.</td>
</tr>
<tr>
<td>distinct([\text{numTasks}])</td>
<td>Returns a new dataset that contains only the distinct elements of the dataset.</td>
</tr>
<tr>
<td>groupByKey([\text{numTasks}])</td>
<td>Returns a new dataset that contains ((\text{K, Iterable}&lt;\text{V}&gt;)) pairs where elements of key \text{K} are grouped together.</td>
</tr>
<tr>
<td>reduceByKey((\text{func, [numTasks]}))</td>
<td>Returns a dataset of ((\text{K,V})) pairs where the values of each key are aggregated using the given reduce function (\text{func}).</td>
</tr>
<tr>
<td>sortByKey([\text{ascending}, [\text{numTasks}]])</td>
<td>Returns a dataset of ((\text{K,V})) pairs sorted by keys in either ascending or descending order</td>
</tr>
<tr>
<td>join((\text{otherDataset, [numTasks]}))</td>
<td>Returns a dataset of ((\text{K, (V, W)})) pairs when called on dataset with pairs ((\text{K,V})) and ((\text{K,W})) pairs.</td>
</tr>
<tr>
<td>cogroup((\text{otherDataset, [numTasks]}))</td>
<td>Returns a dataset of ((\text{K, Iterable}&lt;\text{V}&gt;, \text{Iterable}&lt;\text{W}&gt;)) tuples.</td>
</tr>
<tr>
<td>cartesian((\text{otherDataset}))</td>
<td>Returns a dataset of ((\text{T, U})) pairs, when called on datasets of Types \text{T} and \text{U}.</td>
</tr>
<tr>
<td>coalesce((\text{numPartitions}))</td>
<td>Decrease the number of partitions to \text{numPartitions}.</td>
</tr>
<tr>
<td>repartition((\text{numPartitions}))</td>
<td>Returns an RDD where the partitions are either increased or decreased and the elements of the dataset are balanced across the partitions.</td>
</tr>
</tbody>
</table>
Appendix B

Actions

In this appendix, all the actions that are available in Spark have been listed in Table B.1.
Table B.1: Actions available for RDDs

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce (func)</code></td>
<td>The elements of a dataset are aggregated using the function <code>func</code>. <code>func</code> will be run in parallel and hence should be cumulative as well as associative.</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>All the elements of the dataset are returned as an array to the driver program</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>The number of elements in the dataset is returned</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>The first element of the dataset is returned</td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>The first <code>n</code> elements of the dataset is returned</td>
</tr>
<tr>
<td><code>takeSample(withReplacement, num, [seed])</code></td>
<td><code>num</code> elements of the dataset are returned, with or without replacement, optionally specifying a random number generator seed</td>
</tr>
<tr>
<td><code>takeOrdered(n, [ordering])</code></td>
<td>The first <code>n</code> elements are returned either using a natural order or custom order</td>
</tr>
<tr>
<td><code>saveAsTextFile(path)</code></td>
<td>The contents of the dataset is written to the local file system, HDFS or any hadoop supported file system as a Hadoop SequenceFile</td>
</tr>
<tr>
<td><code>saveAsObjectFile(path)</code> (Java and Scala)</td>
<td>The elements of the dataset are written in a simple Java serialization format which can be loaded using a SparkContext.objectFile()</td>
</tr>
<tr>
<td><code>countByKey()</code></td>
<td>The elements of the dataset are counted by the key.</td>
</tr>
<tr>
<td><code>foreach(func)</code></td>
<td>The function <code>func()</code> is run for each element of the dataset.</td>
</tr>
</tbody>
</table>
Appendix C

Walk-through of SparkFTAlgo

This appendix provides a brief walk-through of the calculations involved in SparkFTAlgo at a Vertex. Consider the Criticality DAG C.1. The value of the parameters in SparkFTAlgo for each RDD along with their CP and GP are tabulated in Table C.1.

The values tabulated are an example and the second, third, fifth, and sixth columns were randomly filled for the purpose of this discussion and to provide a walk-through of SparkFTAlgo. They were not experimentally determined. The values of the first four columns are in seconds.

The discussion will be carried out considering that SparkFTAlgo is being run at Vertex E. The discussion has been split into two cases. The RDD representing Vertex

<table>
<thead>
<tr>
<th>Vertex id</th>
<th>$T_{rt(cp)}$</th>
<th>Time to compute</th>
<th>$T_{rc}$</th>
<th>$T_{cp}$</th>
<th>$T_{rt}$</th>
<th>CP</th>
<th>GP</th>
<th>verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>18</td>
<td>15</td>
<td>62.5</td>
<td>-127</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>37.5</td>
<td>-80</td>
<td>No</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>40</td>
<td>50</td>
<td>21</td>
<td>10</td>
<td>37.5</td>
<td>90.4</td>
<td>YES</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>5</td>
<td>4</td>
<td>37.5</td>
<td>60</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>63</td>
<td>68</td>
<td>29</td>
<td>15</td>
<td>25</td>
<td>117</td>
<td>??</td>
</tr>
</tbody>
</table>

Table C.1: Example Data for SparkFTAlgo
C has been check-pointed whereas the RDDs representing Vertices A, B and D have not been check-pointed.

At E, the $T_{rt(cp)}$ is 10 as the previous check-point is at Vertex C and it takes 10 seconds to restore it. The $T_{rc}$ is 63 seconds + 3 seconds (Time to recompute RDD represented by Vertex B) + 2 seconds (Time to recompute represented by Vertex D). The time to re-compute C is not considered. This is because C will be read from the disk and hence no recomputation will be needed. Hence the $T_{rc}$ is 68 seconds. The rest of the parameters are as shown in the table. The GP for E is 117%. Hence the Vertex E will be check-pointed. If after persisting E, no failure occurs at this point, E will serve as the previous check-points in computations thereafter. If however a failure occurs after E is check-pointed, then 15 seconds will be spent in restoring E.
from the disk.

If on the other hand the RDD represented by Vertex E is not check-pointed, then in case a failure occurs, the time spent to restore the RDD E will be 10 (Time to restore RDD represented by Vertex C as it is check-pointed) + 3 (Time to recompute RDD represented by Vertex B) + 2 (Time to recompute RDD represented by Vertex D) + 63 (Time to recompute represented by Vertex E from RDDs represented by Vertices B, C and D). This is 78 seconds. Hence if E is not check-pointed, it will take 78 seconds to restore it from the previous check-point. On the other hand, if E is check-pointed it will take 15 seconds.
Appendix D

Apache YARN

Apache Hadoop is one of the most popular frameworks for processing Big Data using the MapReduce programming model. The MapReduce model consists of two phases, the Map phase and the Reduce phase. In the Map phase, the data is processed using operations like filtering and sorting. Key-value pairs are generated. The reduce operation then performs summary operations on the values that contain the same key.

Apache Hadoop has two major flaws. One is the tight coupling of the MapReduce with the resource management infrastructure. The other problem is pertaining to the architecture of Hadoop wherein the entire control flow is handled through a certain component leading to scalability issues.

In order to overcome these flaws, the second version of Hadoop called YARN was released. In addition to MapReduce, the YARN scheduling framework could support different types of programming paradigms. Some of the programming frameworks include Apache Tez [33], Apache Spark [43], Dryad [41], and Storm [35].

The architecture of YARN was designed in order to satisfy the following goals;
D.1. ARCHITECTURE

Scalability, Multi-tenancy, Serviceability, Locality Awareness, High Cluster Utilization, Reliability, Secure and auditable operation, Support for programming model diversity, Flexible Resource Model, and Backward compatibility. The architecture of YARN is discussed in detail in Section D.1. The details of HDFS, which is the file system that has been used in YARN, is discussed in Section D.2.

D.1 Architecture

The architecture of YARN is shown in the Figure D.1.

Figure D.1: Apache YARN

The Resource Manager is the unit that allocates the resources to various applications that require the resources in the cluster. The resources are allocated in the form of containers, which are the logical bundles bound to nodes. The application runs within these containers.

The Node Managers are daemons that run on Nodes and track the status of the containers. The Node Managers are responsible for monitoring resource availability, reporting faults and monitoring life-cycle management. The Node Managers and the
Resource Manager interact with each other by making use of the heartbeat signals.

The Application Master is responsible for managing the life-cycle aspects of the jobs, dynamically increasing and decreasing resource consumption, managing the execution’s flow, handling faults and performing other local optimizations.

D.2 HDFS

The HDFS, which stands for the Hadoop Distributed File System, is a distributed, fault-tolerant, scalable and portable file-system written in Java. HDFS consists of two main components, the NameNode and the DataNodes.

In HDFS, the meta-data and the application data is stored separately. The NameNode which is present in the master, stores the meta-data whereas the DataNodes store the application data. The durability of HDFS is maintained through replicating the data across various machines. This increases the opportunity to locate the computation near data.

In the NameNode, the details regarding the mapping of various blocks to the nodes, is stored. Each cluster, which consists of thousand nodes, has only one NameNode.

The replica of each block on the DataNode is represented by two files in the native local system. One file contains the actual data while the other file consists of the block’s meta-data such as the checksum for the data and generation stamps. The DataNode connects to the NameNode and performs a handshake. Each DataNode is identified by a unique storage ID. Thus even if a change of IP-address occurs, the DataNode is still recognizable.
The DataNodes and the NameNode communicate via heartbeats. These heartbeats carry information such as the total storage capacity, the fraction of storage in use and the number of data transfers currently in progress. The NameNode in turn uses these heartbeats to send instructions to the DataNodes. These instructions include commands to replicate blocks to other nodes, re-register nodes and shut-down nodes.