INCRDD: INCREMENTAL UPDATES FOR RDD IN APACHE SPARK

by

Prathish Dodabelle Prakash

APPROVED BY SUPERVISORY COMMITTEE:

_____

Dr. Weili Wu, Chair

_____

Dr. Latifur Khan

_____

Dr. Balakrishnan Prabhakaran

Dedicated to my family

INCRDD: INCREMENTAL UPDATES FOR RDD IN APACHE SPARK

by

PRATHISH DODABELLE PRAKASH, BE

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2017

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Dr. Weili Wu for the expert guidance and providing me with continuous support, encouragement and motivation for my thesis study and related research. Her unwavering enthusiasm kept me constantly engaged in my research throughout my graduate education. I would also like to thank Dr. Latifur Khan for teaching the Big data management and analytics course, which was invaluable for this thesis. Likewise, I express gratitude toward Dr. Balakrishnan Prabhakaran for giving me the sagacious remarks and to serve on my supervisory board.

I would like to thank my friends in the Data Communication and Data Management Laboratory for their help and support. Discussions with them and their early insights launched the greater part of this thesis. I express my sincere gratitude to Department Head, Dr. Gopal Gupta and the entire faculty for providing high-quality administrative and educational service during my studies.

Above ground, I am indebted to my family and friends for being my strong support system and providing the necessary help and encouragement.

October 2016

INCRDD: INCREMENTAL UPDATES FOR RDD IN APACHE SPARK

Publication No. _____

Prathish Dodabelle Prakash, MS
The University of Texas at Dallas, 2017

Supervising Professor: Weili Wu, Ph.D.

Data is constantly changing. Today, there can be incremental updates to the existing data. As the data is evolving with new updates, the results of big data applications gradually become out of date and stale. It is required to refresh the results for every update efficiently.

Apache Spark is used to process multiple petabytes of data on clusters having thousands of nodes. The core abstraction of Spark is RDD (Resilient Distributed Dataset), which is an immutable collection of elements. Due to the immutability of RDD, Spark works information in parallel, permits information reuse, and handles failures and stragglers productively. But Spark lacks flexibility and efficiency of incremental processing of small updates.

In this thesis, IncRDD framework is proposed for incremental processing of updates to the existing data. IncRDD sustains all the powerful features of Spark including parallel processing, data reusability, and fault tolerance. New operations for RDD are implemented to add new records, update the existing records, and delete them.

We introduce a new variant of Cuckoo hashing, Dual-CH Fast-Simple. Dual Cuckoo hashing uses two cuckoo hash tables. The first cuckoo table is used to store records, in every partition of a node. The second hash table is used to implement structural sharing, which adds persistence, utilize previous versions, and avoids expensive re-computation. We evaluate IncRDD using incremental algorithms and provide experimental results to show the significant improvement in the performance of Incremental RDD.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Problem context

Big data is constantly changing. Petabytes of data are being generated in social networks, mobile data, health care systems, software companies, and many other environments. It is important to process the data, draw conclusions and produce the results. Since the data is constantly evolving, the results of big data applications become obsolete and inconsistent. Incremental updates need to be processed to refresh the results of big data applications.

Consider a big data application, which analyzes the social networking website data, to generate a report of the number of followers of celebrities. However, the underlying data is constantly changing; new followers of celebrities are added and deleted. Since the data is constantly changing, the results of this application gradually become inconsistent and invalid. Therefore, it is required to refresh the results of big data applications regularly.

Incremental processing adds persistence and structural sharing to the data. It supports version controlled updates to the data. Past versions are retained in the system, even after they are updated. For every update of data, new versions are created instead of modifying existing versions. Hence, incremental updates return a new copy but internally share almost the same structure. Since it re-uses the existing structure and re-computes only for the new updates, incremental processing is an efficient approach for small incremental updates.

In recent years, incremental processing has been applied to many big data computing framework, MapReduce [1]. The incremental processing [2], [3] approaches have been proven to perform better than the traditional batch processing approaches [4]. Percolator [5] is deployed to create google web index by incrementally processing updates. CBP [6] and Naid [7] are incremental processing systems which require the algorithms to be re-implemented. Incoop [8] detects the changes in input, but only supports task-level incremental processing. Inc-HDFS [8], [9] enables incremental computations to HDFS.

1.2 Problem Statement

Apache spark [10] is in-memory general-purpose distributed data processing system. Spark introduces a basic abstraction called as Resilient distributed dataset (RDD) [11]. RDD is a collection of immutable elements which are operated in parallel and allow data re-use and efficient fault tolerance. RDDs are useful for coarse-grained transformations, but not used for fine-grained updates.

Since RDD is immutable, they have the following advantages:

1. Parallel operation: The data can be operated in parallel and distributed architecture because the underlying dataset is constant (immutable)
2. Data reuse: RDDs can be re-used many times due to immutable data.
3. Divide and conquer: Since the dataset is constant, it can be divided into blocks of data, processed individually, and combine the results.
4. Fault tolerance: Spark can achieve efficient fault tolerance by re-computing the results on different nodes using RDD and lineage graph.

5. Straggler mitigation: Slow running tasks are identified and re-run on other nodes because the dataset is immutable.

Today, in real life big data applications, the data is constantly changing. It is required to process the updates to data efficiently. Existing solutions to re-compute the complete task sacrifice flexibility and efficiency.

The challenge is to introduce the incremental updates in Spark RDD but retains all the above-mentioned advantages of Spark (parallel operation, data re-usability, fault tolerant and straggler mitigation). We implement the IncRDD interface to efficiently process the incremental updates. We also introduce a new variant of cuckoo hashing, the dual-cuckoo hashing, to provide fine-grain incremental processing.

1.3 Document Organization

The remainder of this thesis documents the IncRDD framework, implementation, and the results for incremental updates.

Chapter 2 presents an introduction to Hadoop MapReduce [1] and Apache spark [10] and its advantages. Chapter 3 describes the immutability of spark and depicts the need for Incremental updates. Chapter 4 describes the Existing solutions for incremental updates and motivation for this thesis. Chapter 5 introduces dual-cuckoo hashing techniques and variants. Chapter 6 documents my contribution of IncRDD interface design and implementation. The results of the incremental processing are presented in Chapter 7. Finally, Chapter 8 concludes the thesis and introduces the future work of incremental processing in Spark

CHAPTER 2

BACKGROUND

2.1 MapReduce Background

MapReduce [1] is a distributed programming model for processing large data sets. The two important phases of this programming model are map phase and reduce phase. Mappers consume key, value pairs as the input and produce the intermediate key, value pairs as the output. Reducers accept the key, list of values as the input to generate the key, value output, as shown in Figure 2.1.

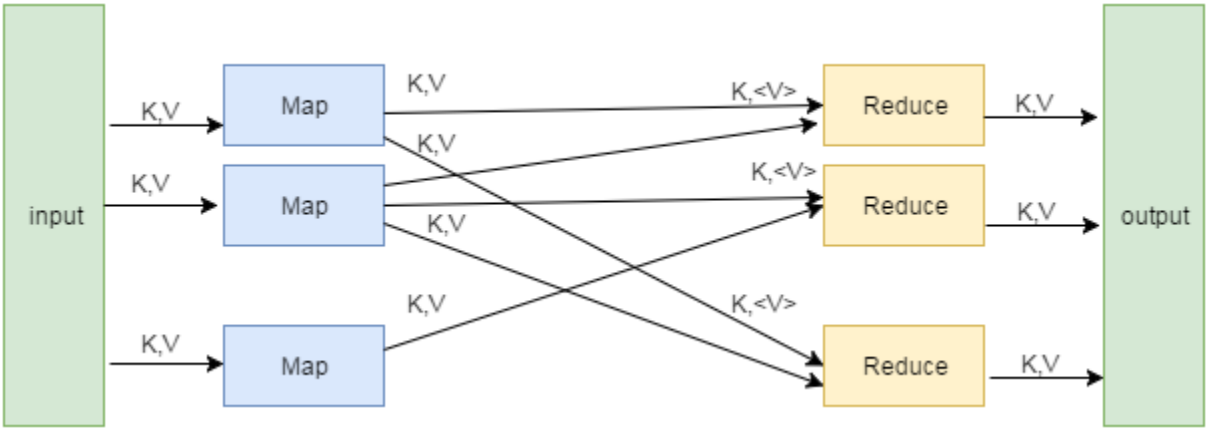

Figure 2.1. Computations in MapReduce

MapReduce works on master-slave architecture. Master node has JobTracker which assigns the tasks to the worker nodes. Worker node executes the map or reduces tasks until

completion. When all map and reduce tasks have been executed, the master node returns the output to the user program.

2.2 Spark Background

Today, Apache Spark [10] is being deployed by major players like yahoo, Twitter, Netflix and eBay. Spark is a large scale data processing engine [12] that has advanced execution engine that supports in-memory computation. Spark was started at UC Berkeley in 2009 and has been rapidly adopted by a wide range of companies [13] and has become largest big data open source community. It is used to process multiple petabytes of data on clusters having thousands of nodes.

Since Spark supports in-memory computation, it is highly popular for executing iterative algorithms. It runs programs up to 100x faster than Hadoop MapReduce. Spark provides high-level APIs in Scala, python, Java, and R.

2.2.1 Spark Modules

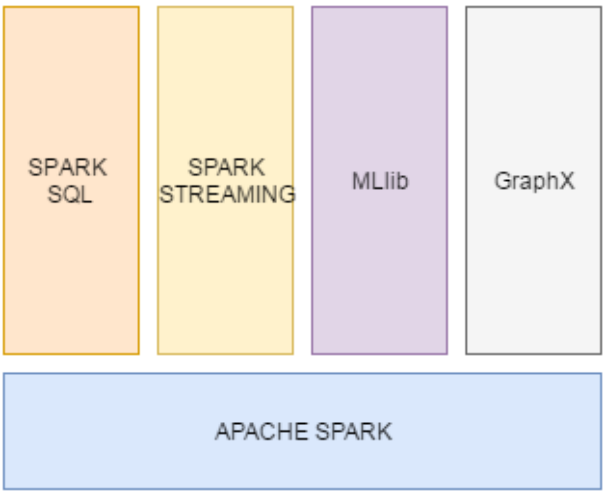

Figure 2.2. Spark modules

The Powerfulness of Apache Spark is due to the stack of libraries that it provides. The Spark core [15] is the foundation of the Apache Spark. It provides the basic functionalities of Spark through application programming using RDD abstraction.

Spark SQL [16] is a component built on top of Spark. With the benefits of relational processing, Spark SQL lets users call complex libraries in Spark. It has packages to manipulate Data frames, which is a collection of structured and semi-structured data.

Spark Streaming enables the fast scheduling in Spark and used in streaming analytics. It ingests data in small batches and performs operations on them. It has built-in to consume Kafka, Flume, Twitter, ZeroMQ and TCP/IP packets.

MLlib is a distributed, machine learning framework, which is 9x faster than Apache mahout. It includes many statistical and machine learning algorithms.

GraphX is a distributed, graph processing framework. It is similar to the in-memory version of Apache Giraph but does not support for graphs that need to be updated.

2.2.2 RDD

The basic abstraction of Apache Spark is RDD [11] - Resilient Distributed Dataset. RDD is a collection of immutable elements which are operated in parallel and allow data re-use and efficient fault tolerance. RDDs are useful for coarse-grained transformations, but not used for fine-grained updates.

There are two methods to create an RDD: parallelizing an existing data or using data in the external shared system, like HDFS.

```
scala> val data = Array("utd","cs","big_data")
data: Array[String] = Array(utd, cs, big_data)

scala> val myRDD= sc.parallelize(data)
myRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> myRDD.collect()
res0: Array[String] = Array(utd, cs, big_data)
```

Figure 2.3. Spark shell screenshot to create RDD.

In the above figure 2.3, an array of strings is created. Note that Scala is statically typed. Spark Programmer does not need to specify the data type explicitly. RDD is created by passing data to SparkContext's parallelize() method. This creates an RDD, which is the parallel collection of immutable elements.

There are two types of RDD operations: transformations and actions. RDDs are lazy in transformation. They do not execute transformations immediately but compute their results when an action is invoked. Spark maintains a lineage graph to record the transformations, which are used to build the data. Examples of transformations include map, reduce, join, and filter.

Consider a WordCount example in Spark, shown in Figure 2.4, counts the number of occurrences of every word in a given text document.

```
scala> val wordCounts = textFile.flatMap(
     | line => line.split(" ")).
     | map(word => (word, 1)).
     | .reduceByKey((a, b) => a + b)
```

Figure 2.4. WordCount program in Spark

The lineage graph is shown in Figure 2.5 below.

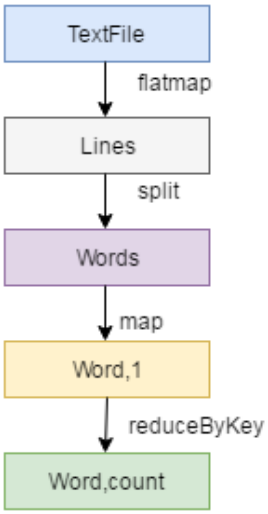

Figure 2.5. Lineage graph for WordCount program.

Spark creates this lineage graph when a set of transformations are used. They are executed when an action is invoked. This mechanism is used to achieve efficient fault tolerance.

CHAPTER 3

SPARK IMMUTABILITY AND NEED FOR UPDATES

3.1 Immutability in Spark

Scala [18] is a statically typed programming language which combines the features of object-oriented and functional programming. Scala has two types of variables: value (Val) or variable (var). 'Val' means that it is a variable that cannot be changed and this is called as the immutable variable. The below Figure 3.1 depicts the use of ‘Val’ and ‘Var’ types.

```
scala> var thesis1= "big data"
thesis1: String = big data

scala> thesis1 = "spark"
thesis1: String = spark

scala> println(thesis1)
spark

scala> val thesis2 = "big data"
thesis2: String = big data

scala> thesis2="spark"
<console>:27: error: reassignment to val
         thesis2="spark"
                 ^

scala> println(thesis2)
big data
```

Figure 3.1. Spark shell screenshot to show ‘val’ and ‘var’ types.

RDDs are declared as 'val' and cannot change its value. Since RDD is immutable, they have the following advantages:

3.1.1 Parallel Operation

The data can be operated in parallel and distributed architecture because the underlying dataset is constant / immutable.

```
scala> val data= Array("thesis","spark", "incremental", "updates")
data: Array[String] = Array(thesis, spark, incremental, updates)

scala> val myRDD= sc.parallelize(data)
myRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:29
```

Figure 3.2. RDD creation

As shown in the above figure 3.2, RDD is created by using the ‘parallelize()’ method of Spark Context. This creates a parallel collection of data because the data is immutable ('Val' type).

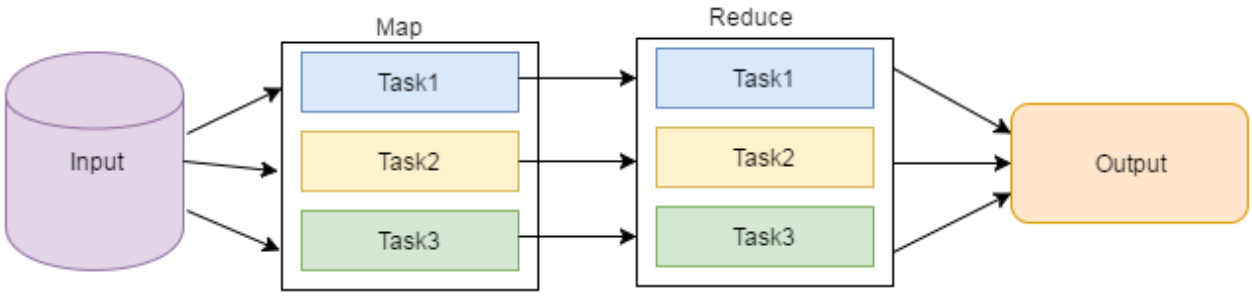

Figure 3.3. Parallel operation in Spark

As shown in Figure 3.3, the Driver program partitions the data into blocks and sends it to the worker nodes for execution. A worker node executes a block of data and returns the result to the driver program. In a distributed architecture, this is executed in parallel. Due to parallel operation, every task / node is independent and hence Spark is the faster processing system.

3.1.2 Data Reuse

Since the data is immutable, RDD can be reused multiple times. This feature is important for an efficient handling of failures in Spark.

The task can be retried in same node or replicated in a different node multiple times, to get a similar result each time.

3.1.3 Divide and Conquer

Since the dataset is constant, it can be divided into blocks of data, processed individually, and combine the results. Spark uses a master-slave architecture where the master program partitions (divide) the data, the worker node execute the tasks, and results are returned to the master program (conquer).Figure 3.3 depicts the divide and conquer technique used in Spark.

3.1.4 Fault Tolerance

Node failures are a common issue in a distributed architecture system. Spark can achieve efficient fault tolerance by re-computing the results on different nodes using RDD and lineage graph. Spark detects the node failures and re-runs the task in another node. Since RDD is immutable, the data is independent of a node and using the lineage graph, the task is retried.

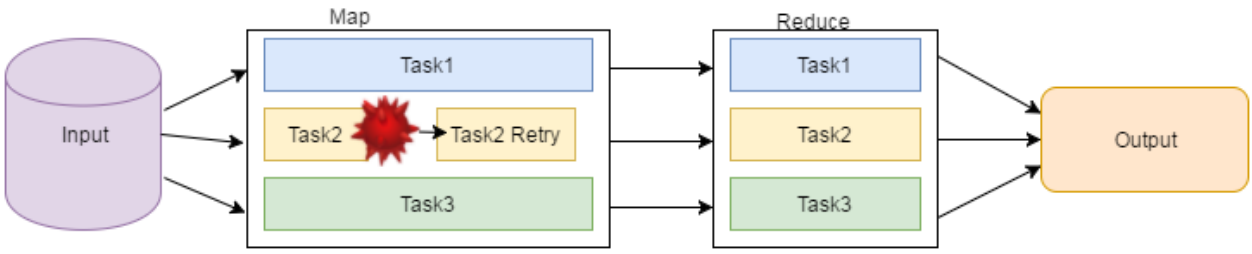

Figure 3.4. Fault tolerance in Spark

In Figure 3.4, if the Task2 fails, then it can be retried multiple times because the input data to the task is immutable. Also, whenever the task is retried, it returns the same result because the data is not updated /changed.

3.1.5 Straggler Mitigation

Slow running tasks are called as Stragglers. Tasks are executed slowly because of many reasons, including garbage collection pauses. Stragglers are efficiently identified and re-run on other nodes because the dataset is immutable. Spark RDD shows an independent behavior by showing the same result if they are rerun many times.

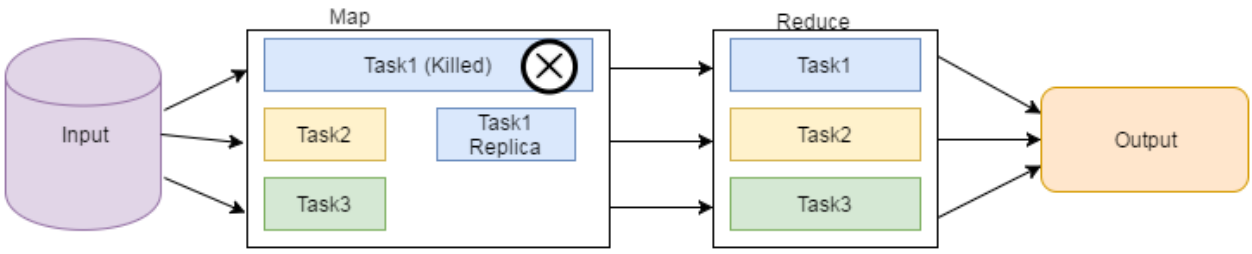

Figure 3.5. Straggler Mitigation in Spark

In Figure 3.5, the task1 is a straggler. Spark efficiently identifies slow nodes. It is killed by the master program, and then task1 is replicated, and executed by node2.

3.2 Need for Efficient Updates

Real-time data processing is changing business [19]. In recent years, there is a shift in the method of big data processing, from batch processing to real-time data processing. Data is collected from many environments, including the social media websites, mobile devices, and

smart IOT appliances. All the generated data needs to be stored, processed and analyzed to draw conclusions or to make decisions.

Constant data flow demands a real-time data processing. In real time data, there may be small updates, new additions, and few deletions to the existing data. Consider the social networking websites like Facebook, where new comments are added every second, profile pictures are updated, and few old posts are deleted.

Consider an application where Spark is used to generate an analytics of the names and number of tweet counts of user's followers in Twitter.

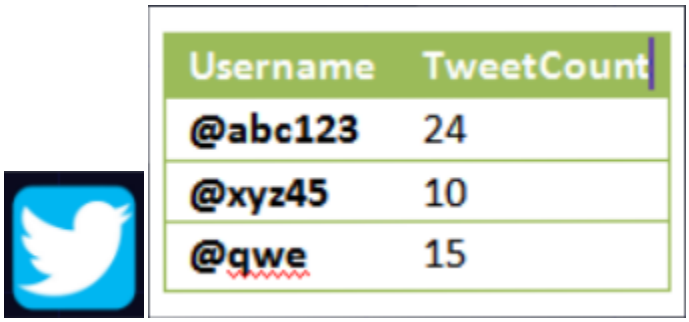

Figure 3.6. Twitter sample report

Figure 3.6 depicts the sample report. Since its real-time data, the report generated may be no longer valid because the input data would have changed, while the data is being processed. The computed results will be invalid and obsolete.

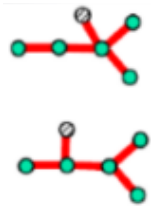

Figure 3.7. Graph updating problem

Spark can consume real-time data for stream aggregation algorithms, incremental algorithms, and graph algorithms. In these algorithms, the data may be dynamically changed. An example is shown in Figure 3.7.

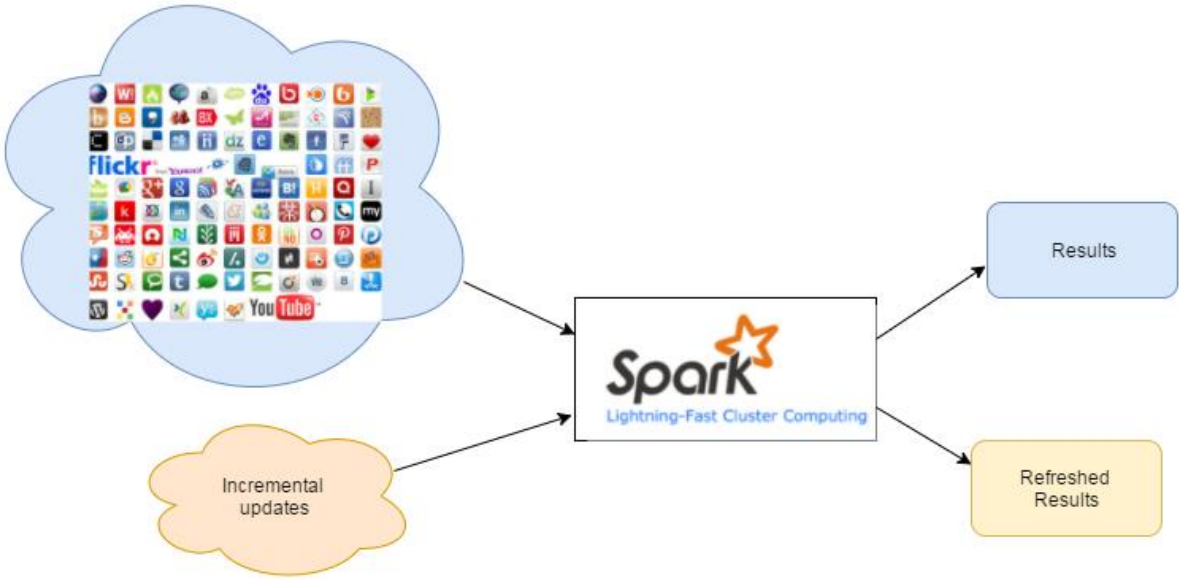

Figure 3.8. Need for Incremental processing in Spark

There is a need for processing the updates to existing data. Apache Spark needs to efficiently process the incremental data. Figure 3.8 depicts the data gathered from the different environment that are processed by Apache Spark. There are incremental updates to existing data, which are required to be processed, and to get the refreshed results.

CHAPTER 4

EXISTING SOLUTIONS

4.1 Existing Solution

In this chapter, we discuss the existing solutions for incremental updates [20], [21] in Apache Spark.

4.1.1 In-Place Mutation

The standard solution for incremental updates is to accept an in-place mutation. This approach allows updating the RDD when the real-time data changes.

Consider an example, in Scala, where in-place mutation updates the variables directly, as shown in Figure 4.1. Here, we update the total_tweet count and tweet count of a particular user.

```
scala> def update (tweet : Tweet) = {
     | total_tweet = tweet.count + 1
     | println(tweet.name)
     | tweet.count++
     | }
```

Figure 4.1. Update method to change tweet counts

Using in-place mutation, we can update the user's tweet count and total tweet count directly. This method is a faster solution to incremental update but suffers from providing fault tolerance. Initally assume that the tweet_count=0 and total_tweet =0. Consider a situation where the total tweets are updated, but the task fails.

Figure 4.2. Fault tolerance in spark

Spark retries the task in another node. Now the user's tweet count is updated correctly, but the total tweet count is updated twice. The tweet_count =1 and total_tweet =2, and it becomes inconsistent / invalid.

4.1.2 Batch Operations

Batch operations are similar to the atomic operations in a database [22], such that either all occur or nothing occurs. This can be achieved by using LOCK and UNLOCK, or COMMIT and RELEASE. This ensures that a serial execution occurs.

```
scala> def update (tweet : Tweet) = {
     | LOCK
     |  total_tweet = tweet.count + 1
     | println(tweet.name)
     | tweet.count++
     | UNLOCK
     | }
```

Figure 4.3. Atomic Batch operations

In Figure 4.3, LOCK and UNLOCK are used for updating the existing data. But, this method proves to be inefficient for large distributed dataset. Long-lived snapshots are inefficient for Spark.

4.1.3 Object Cloning

Another solution to the incremental updates is to clone an object. This approach specifies to duplicate an object that needs to be modified and then perform the operations of the replica.

```
scala> def update (tweet : Tweet) = {
     | var new_tweet = tweet.clone()
     | total_tweet = new_tweet.count +1
     | println (new_tweet.name)
     | new_tweet.count++
     | return new_tweet
```

Figure 4.4. Object cloning

Figure 4.4 shows the cloning method. In this method, a duplicate of the tweet is created, modified and returned from the function.

This solution may be feasible, but it is not an optimal solution. Duplicating all objects in a real-time application is not an efficient solution.

4.1.4 IndexedRDD

IndexedRDD [20] is an approach developed for supporting efficient fine-grained updates in Spark. It is an RDD based distributed key-value storage interface that supports immutable semantics [21]. This is an interface where application programmer can use it for real-time updates. This supports fine-grained operations including 'get', 'put' and 'delete'. It also supports accelerated RDD operations including 'filter' and 'joins'.

IndexedRDD uses a classic data structure called as an Adaptive Radix Tree (ART) [23]. This is a recent work for an efficient main memory indexing in databases. Radix trees have sorted order traversals and have better asymptotic performance than binary trees. It has efficient union and intersection operations.

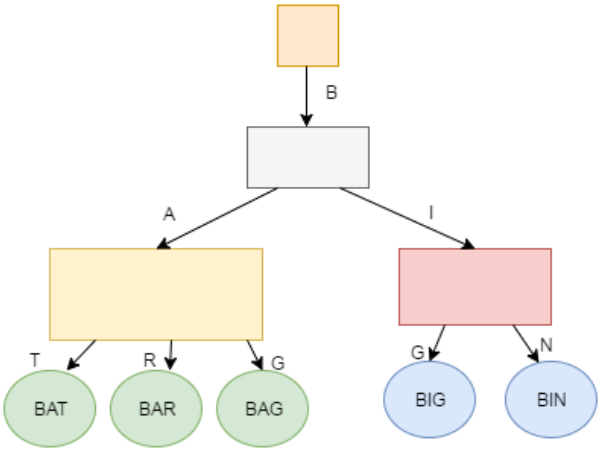

Figure 4.5. Adaptive Radix trees

Adaptive Radix trees are the radix trees where the size of the nodes is adaptive to the content of the tree. Although the height of the radix tree remains unchanged, the size of node differs based on the content. As shown in Figure 4.5, ART has two important features which provides the adaptive behavior; node compression and lazy expansion.

This approach introduces persistence to radix trees, named as Persistent Adaptive Radix Trees (PART). This support versioned updates. They create new versions instead of modifying the existing version. It exhibits structural sharing, where multiple versions share the same internal structure. Persistence can be easily implemented in a Radix tree due to its structure.

This solution provides APIs for incremental updates and shows experimental evaluation for efficient fine-grained updates. However, this method fails to perform better than hash tables with real-time applications in streaming aggregation. This is further described in the next section of this chapter.

4.2 Motivation

The existing solution, IndexedRDD is evaluated to show the performance, compared to the other mutable data structures.

A real-time experiment is performed, on IndexedRDD solution. Counting occurrences of 26 character string ids is performed. It is loaded with one billion keys. A stream of 100 million keys is used to check the efficiency of insertions to RDDs. The experimental results are shown in Figure 4.6

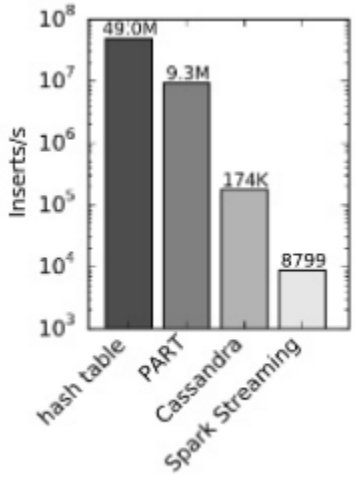

Figure 4.6. Real-time application with IndexedRDD

From the figure, it is evident that the solution, PART is 50 times faster than Cassandra for insertions. However, a mutable hash table shows an 18% improvement in performance of insertions in Spark. This result leads to a question that whether hashing can be used to provide incremental updates in Spark.

The Research also provides evidence that the Hash tables, perform better than Adaptive Radix trees [23]. A complete experiment involving a comparison of Adaptive Radix trees, with the variants of hashing techniques, is shown in below figures 4.7, 4.8.

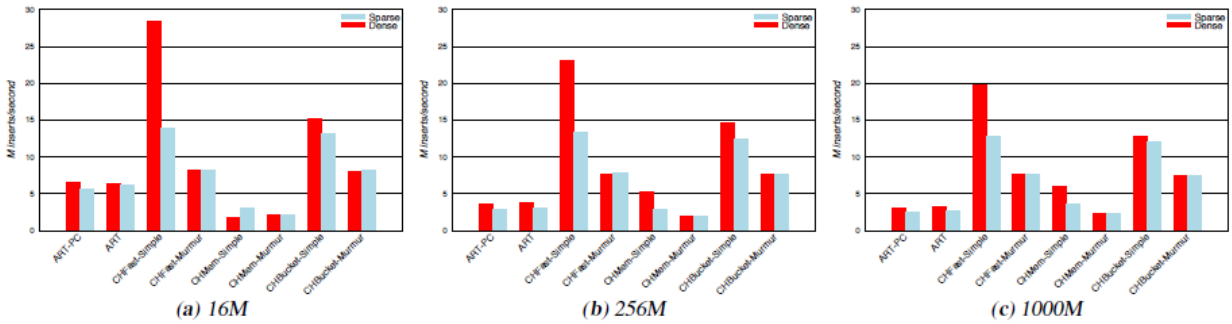

Figure 4.7. Insertion throughput comparison between ART and cuckoo hashing

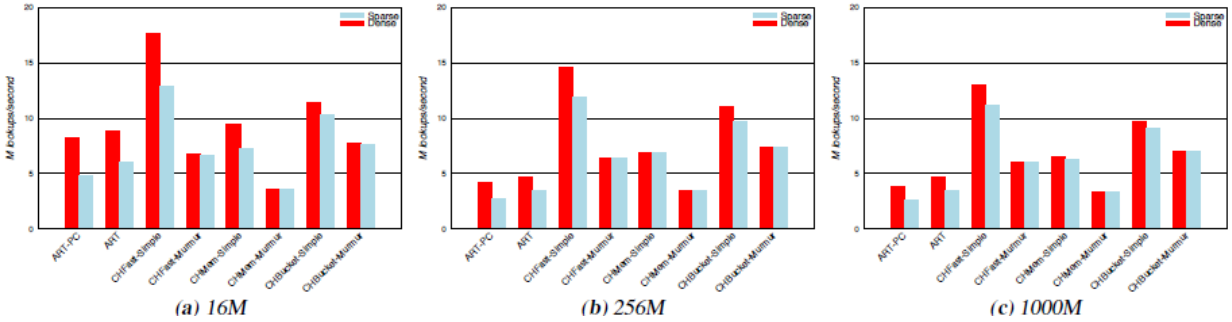

Figure 4.8. Lookup comparison between ART and cuckoo hashing

It is evident that the variants of cuckoo hashing are known to have better efficiency than the Adaptive Radix Trees. Cuckoo hashing is at least 4.8x faster for insertions than ART, at least 2.8x faster for lookups than ART.

This provides a motivation for using hashing schemes in Spark, to provide incremental updates to RDDs.

CHAPTER 5

DUAL CUCKOO HASHING

5.1 Cuckoo Hashing

Cuckoo hashing [23], [24] is a hashing strategy to resolve collisions in hash tables. It produces worst-case constant time [O(1)] efficiency for lookup and deletions. It also gives an amortized constant-time for insertion.

The name derives from the species of cuckoo and its behavior. Cuckoo bird kicks the eggs / young out of its nest when it recognizes that this doesn't belong to it. Similarly, when the key hashes to an index of the hash table, it kicks out the element that was stored previously to a new index position, as shown in Figure 5.1.

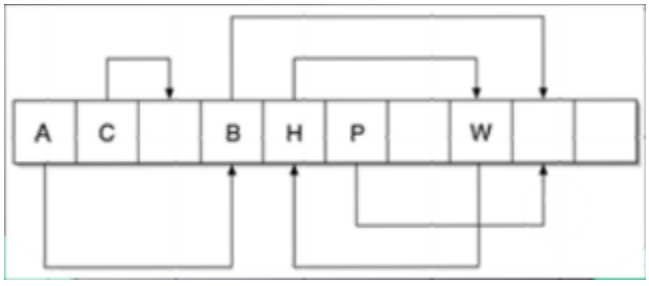

Figure 5.1. Sample Cuckoo hash table

5.2 Operations in Cuckoo Hashing

This section describes the algorithms of all the operations on cuckoo hash table. Cuckoo hashing usually has two hash tables T1 and T2 with r buckets. There are two hash functions h1 and h2 that are applied to the hash table.

Figure 5.2. Cuckoo hashing

5.2.1 Lookup

Cuckoo hashing performs a lookup in a worst-case constant time. A key 'x' can be stored in either T1[h1(x)] or T2[h2(x)].

```
function lookup(x)
    return T1[h1(x)] = x  ∨  T2[h2(x)] = x
end
```

The lookup algorithm checks for the existence of the key in both of the locations and returns the result.

5.2.2 Deletion

Deletion of an element is a simple operation, which removes the key from the table. This is also performed in a worst-case constant time.

5.2.3 Insertion

The insert algorithm exhibits the behavior of cuckoo hashing. Every element is either stored in T1[h1(x)] or T2[h2(x)], but not both. When inserting the element x, first the location T1[h1(x)] is checked for emptiness. If it is empty, then insert 'x' in that location, otherwise, x kicks out the element y already present in that location, and x is stored there. To insert the element ‘y’ in table T2, the location T2[h2(x)] is checked if it is empty, if so, then insert 'y' in that location, otherwise, ‘y’ kicks out the element, and this process is repeated.

```
procedure insert(x)
    if lookup(x) then return
    loop MaxLoop times
        x ↔ T1[h1(x)]
        if x = ⊥ then return
        x ↔ T2[h2(x)]
        if x = ⊥ then return
    end loop
    rehash(); insert(x);
end
```

Consider, two hash functions h1(x) = x mod 5, and h2(x) = (x/5) mod 5.Let the elements to be inserted into two tables be {20, 50, 71, 1}. Table 5.1 shows the iterations of insertions of keys to the cuckoo hash tables.

Table 5.1. Hashing of keys in array

x	h1(x) = x mod 5	h2(x) = (x/5) mod 5
20	20 mod 5= 0	20/5 mod 5= 4
50	50 mod 5 =0	50/5 mod 5= 0
71	71 mod 5 = 1	71/5 mod 5= 4
1	1 mod 5 =1	1/5 mod 5 = 0

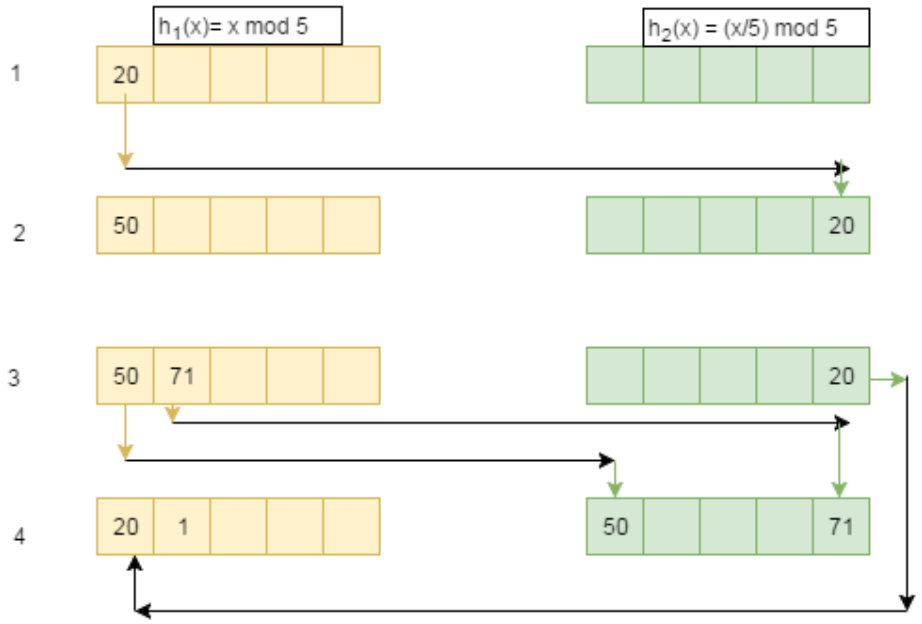

Figure 5.3. Cuckoo hash insertions

The Figure 5.3 shows the iterations of every key inserted to hash tables using cuckoo hashing mechanism. It is evident that in the fourth iteration when key 1 is inserted, it kicks out an element 71, and then 71 kicks out an element 20, and finally 20 kicks out element 50. However, it may happen that this process repeats and enters a loop. This is solved by rehashing mechanism. Rehashing is a design strategy. It does not require allocating new tables, but instead choose new hash functions to reinsert the elements within the tables.

5.3 Variants of Cuckoo Hashing

Cuckoo hashing is divided into three types based on the number of tables used to store the elements:

1. CHFast: This variant stores elements in two hash tables, with two hash functions. This is used to gain performance.

2. CHMem: This variant stores elements in four hash tables, with multiple hash functions. This is used to gain efficiency in memory.

3. CHBucket: This variant allows many elements to be stored in one location of the hash table [23]. Each location of the hash table has a bucket of size equal to cache-line size.

Cuckoo hashing is sensitive to hash functions used [25]. It performs well if good hash functions are used. The two important hash functions used in cuckoo hashing are:

1. MurmurHash64A

2. Multiplicative hashing: Is a well-known hashing scheme, given by:

$$h_z(x) = (x \cdot z \mod 2^w) \text{ div } 2^{w-d}$$

Where, x is the integer to be hashed, w is a number of bits in x {0, . . . 2^w-1}, z is odd w-bit integer in {0, ... 2^w-1}, and the hash table is of size 2^d. This can be efficiently implemented by using multiplication with a random integer and modulo division. It has been proven [26] that the collision probability is:

$$Pr[h_z(x) = h_z(y)] \leq \frac{2}{2^d} = \frac{1}{2^{d-1}}$$

This probability is twice of the ideal probability and shows that the multiplicative hashing is a simple, and a robust hash function.

Using a different number of tables and hash functions, multiple variants of cuckoo hashing can be implemented.

5.4 Implementation of a new Cuckoo Hashing - DUAL CHFast-Simple

We implement a new variant of Cuckoo hashing- Dual CHFast-Simple. CHFast-Simple is a cuckoo hashing technique where two tables are used with two multiplicative hash functions.

We create hash functions using the hash code given by,

Hash Code = ((UPPER * A) + (LOWER *B)) / (2^ (32 - k))

Where A, B are random integers, UPPER is the higher 16-bit value of a 32-bit integer, LOWER is the lower 16-bit value of a 32-bit integer, and 2^k is the number of buckets we are hashing into.

We implement Dual CHFast-Simple for introducing the structural sharing between the data. This is implemented by using two CHFast-Simple cuckoo hash tables.

Figure 5.4. First CHFast-Simple

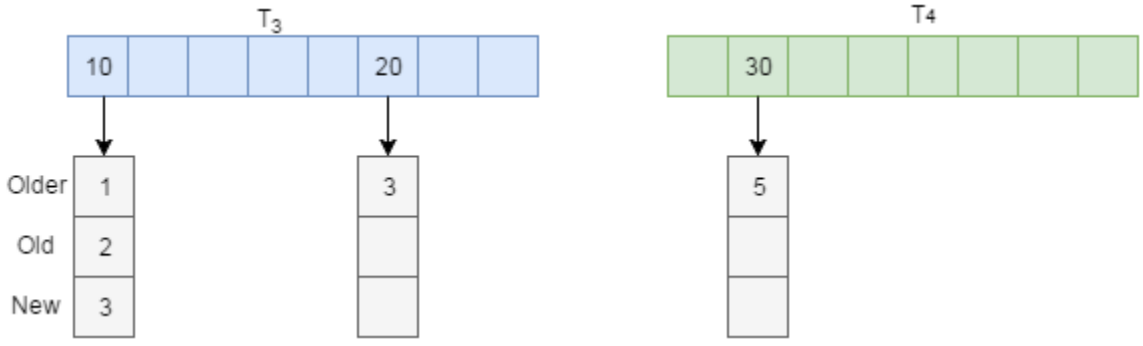

Figure 5.5. Second CHFast-Simple

As shown in Figure 5.4, 5.5, the first cuckoo hash table is used to store the elements in a key, value pair, within a node. The second cuckoo hash table is used maintain the versions of data. If there are new updates to data, the second cuckoo hash table maintains the structure, for storing the past versions even after they are updated. This introduces structural sharing of data, where it updates and returns a new copy but internally share almost the same structure. Dual CHFast-Simple introduces efficient method of Data persistence.

CHAPTER 6

INCRDD: INCREMENTAL UPDATES FOR RDD IN APACHE SPARK

6.1 Contribution

We present the solution to efficient incremental processing in Apache Spark in the interface, IncRDD. IncRDD is a new custom RDD, created to accept and process the updates of the real-time data.

RDD is a collection of elements which are immutable or constant. IncRDD is a collection of Dual CHFast-Simple, partitioned, and mutable elements. It can be pictorially represented as below:

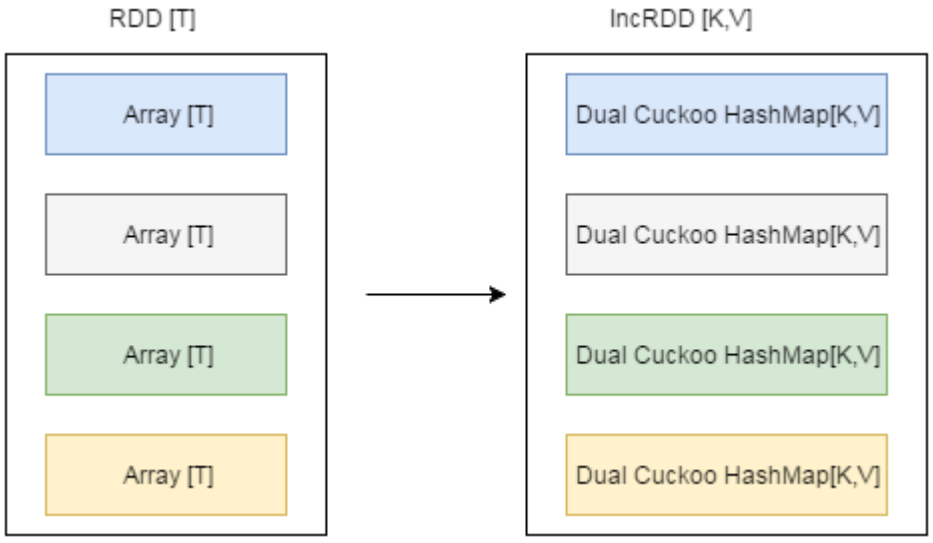

Figure 6.1. IncRDD mapping to RDD

We also provide a few APIs to the application programmer to make updates to RDD elements. This includes add, update, and delete.

The programmer should create an RDD using this package, IncRDD and then use the APIs.

The internal structure of the framework is shown below in Figure 6.2:

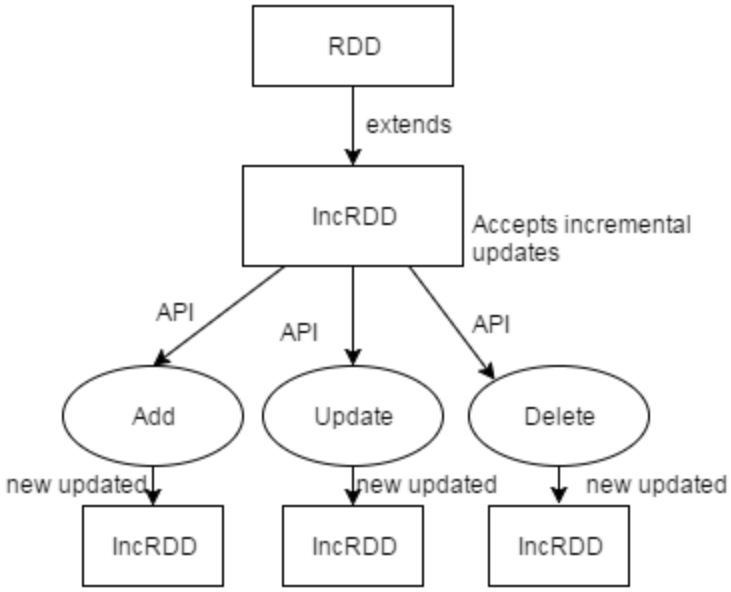

Figure 6.2. IncRDD interface structure

The internal structure of the framework is shown below:

```
class IncRDD[K, V] extends RDD[K, V] {
    // Incremental operations to add, modify and delete data in RDD
    def add (K, V)
    def update (K, V)
    def delete (K, V)
     }
```

6.1.1 Add

This API is used to add new values to existing RDD. As mentioned in section 3.1, RDD does not support any operations / methods that change the value of RDD by default. Add() is used to add a key, value pair to existing RDD.

Parameters - <Key, Value> pair that needs to be added in RDD

Returns - A new version of IncRDD including the additions.

6.1.2 Update

This API is used to add incremental updates to the existing RDD. As mentioned in section 3.1, RDD does not support any operations / methods that change the value of RDD by default. Update() is used to change the value of an existing key in an RDD. As the real-time data changes, this API supports these changes to the existing RDD. An application programmer should use this API for incremental updates in IncRDD.

Parameters - Key, and the value that needs to be updated.

Returns - A new version of IncRDD including the incremental updates.

6.1.3 Delete

This API is used to remove the existing values from an RDD. As mentioned in section 3.1, RDD does not support any operations / methods that change the value of RDD by default. Delete() is used to remove a key, value pair to existing RDD. Internally, it is implemented using the Dual CHFast-Simple cuckoo hashing.

Parameters - <Key, Value> pair that needs to be removed in RDD

Returns - A new version of IncRDD including the deletions.

6.2 Design and Implementation

Figure 6.2, shows the design diagram of IncRDD. Firstly, we implement the Dual CHFast-Simple cuckoo hashing technique, which we discussed in the previous chapter. An abstract class is implemented for a partition of the IncRDD. We partition the elements of IncRDD using the implementation of IncrementalPartition. Finally, the interface, IncRDD is implemented by extending the base RDD and using IncrementalPartition. We also provide a few APIs, including add, update, and delete, to perform fine-grained updates to RDDs.

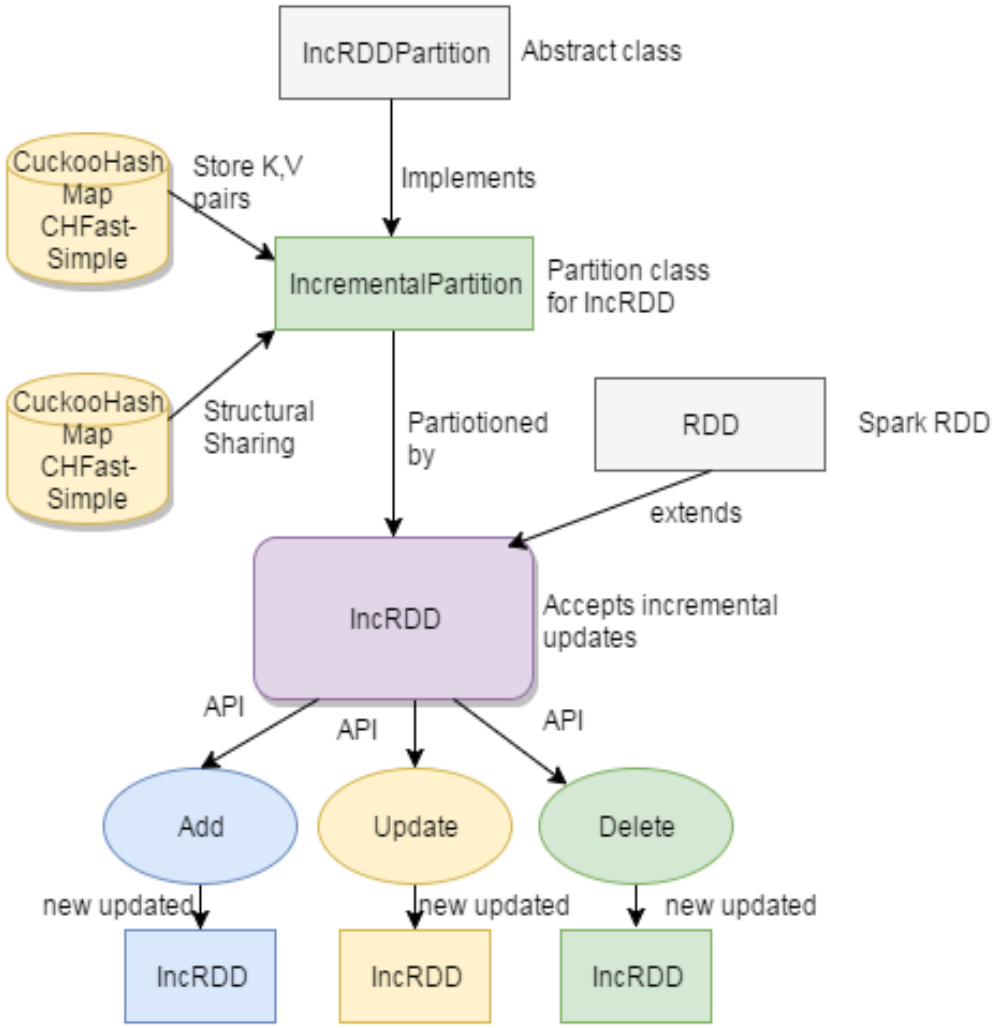

Figure 6.3. Design diagram of IncRDD

It was mainly implemented using Scala programming language. Scala has features of object-oriented programming and functional programming. Cuckoo hashing and its variant were implemented in java [28] and linked to the scala code. The challenges during design include the creation of a partition of IncRDD with cuckoo hashing techniques. The creation of a mutable RDD to accept changes and return a new RDD after an update was challenging task in the implementation.

CHAPTER 7

EXPERIMENTAL RESULTS

7.1 System Setup

The following setup was used for performing the experiments and evaluating the results.

Table 7.1. System setup for performing experiments

System	64 bit Linux
Spark	2.0
Hadoop	2.7.3
Scala	2.11.8
RAM	8 GB
Total space for experiment	150 GB
Number of worker nodes	3
Data set	Up to 5.4GB
Master node	Local and yarn cluster
Memory per executor	4GB/ 6GB
Number of cores per executor	7
Number of executors to launch	6

7.2 Comparison with Existing Solutions

In this section, we present a set of experiments to evaluate the performance of incremental updates to RDDs. First, a test application in Spark is developed to use the IncRDD framework and the existing solution for incremental updates. Figure 7.1 shows the performance evaluation for insertions of new data in RDD. The number of insertions per second is monitored for a different number of key, value pair inserts. The performance of insertion of IncRDD is compared to the existing solution IndexedRDD, which is implemented using Radix trees.

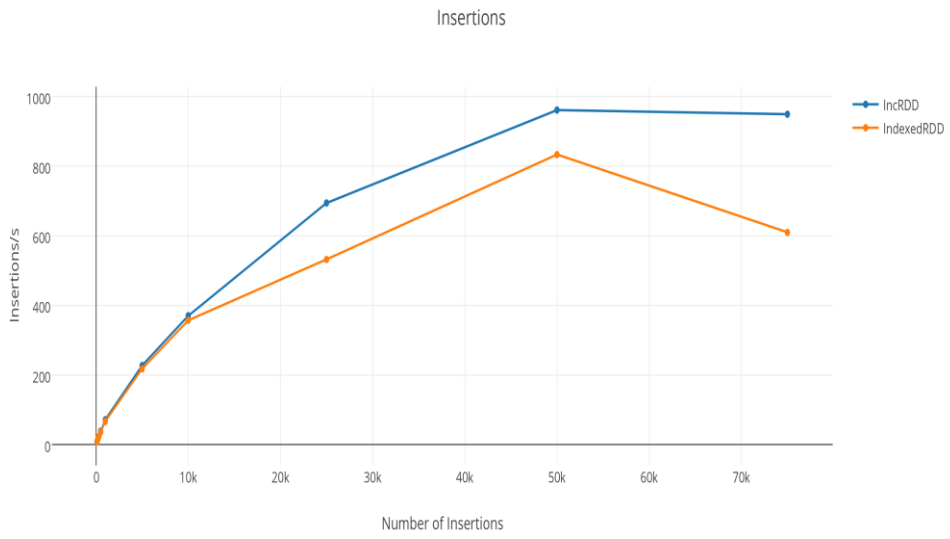

Figure 7.1. Performance of IncRDD for insertion

From the Figure 7.1, it is evident that our solution 'IncRDD' shows better performance than the IndexedRDD because IncRDD uses a new variant of cuckoo hashing that performs faster insertions than adaptive radix trees. Thus, it is natural that IncRDD performs faster than other existing solutions.

We also present the evaluation of the performance of insertion by saving the results to HDFS text file. This experiment only shows the comparison of inserts for small incremental updates. Figure 7.2 show that IncRDD performs 20% faster than IndexedRDD solution. This is very close to the expected performance gain mentioned in section 4.2 -Motivation of this thesis: that a mutable hash table shows an 18% improvement in performance of insertions in Spark.

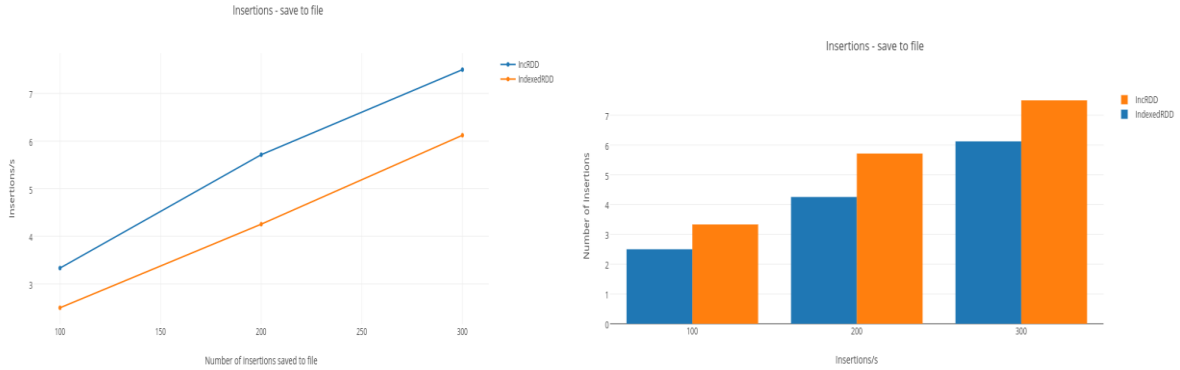

Figure 7.2. Insertions to RDD and save to HDFS text file

Next, we conduct an experiment to show the performance of deletion of data in RDDS. We write a Spark application to delete the key, value pairs of existing RDD. Figure 7.3 shows the performance evaluation for deletions of existing data in RDD. The number of deletions per second is monitored for a different number of key, value pair inserts. The performance of deletions of IncRDD is compared to the existing solution IndexedRDD, which is implemented with Radix trees. From the Figure 7.3, it is evident that our solution 'IncRDD' shows better performance than the IndexedRDD because IncRDD uses a new variant of cuckoo hashing that performs faster deletions than adaptive radix trees. Thus, it is natural that IncRDD performs faster than other existing solutions.

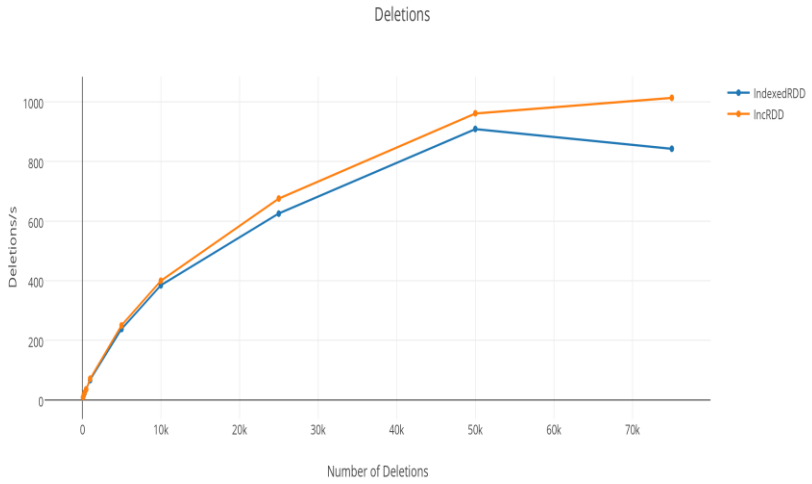

Figure 7.3. Performance of IncRDD for deletions.

We also present the evaluation of the performance of deletions by saving the results to HDFS text file. This experiment only shows the comparison of deletions of few data. Figure 7.4 show that IncRDD performs 15% faster deletions than IndexedRDD solution.

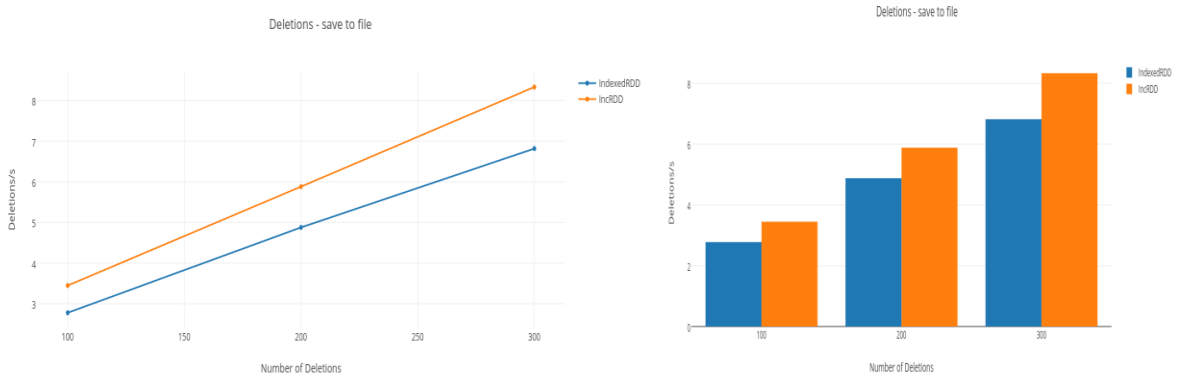

Figure 7.4. Deletions to RDD and save to HDFS text file

Finally, we conduct an experiment to show the performance of incremental updates of data in RDDS. We write a Spark application to perform incremental updates to the few of the existing key, value pairs of RDD. Figure 7.5 shows the performance evaluation for incremental

updates of existing data in RDD. The number of updates per second is monitored for a different number of key, value pair inserts. The performance of incremental updates to IncRDD is compared to the existing solution IndexedRDD, which is implemented using Radix trees. From the Figure 7.5, it is evident that our solution 'IncRDD' shows better performance than the IndexedRDD.

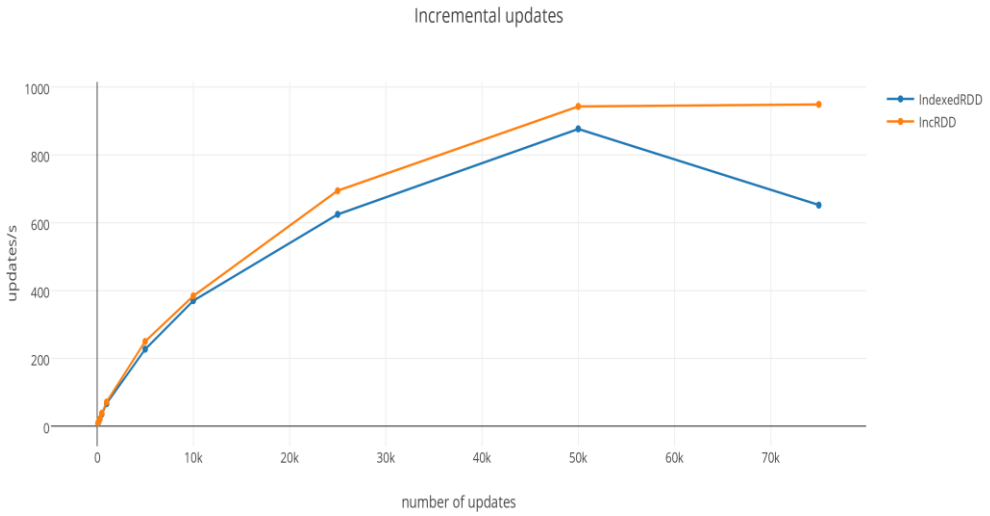

Figure 7.5. Performance of IncRDD for incremental updates

.

We also present the evaluation of the performance of updations by saving the results to HDFS text file. This experiment only shows the comparison of incremental updates to the existing IncRDD data.

The Figure 7.6 show that IncRDD performs 50% faster incremental updates than IndexedRDD solution. This shows that our solution IncRDD, which uses mutable Dual CHFast-Simple cuckoo hashing outperforms other existing solutions for providing incremental updates in Apache Spark.

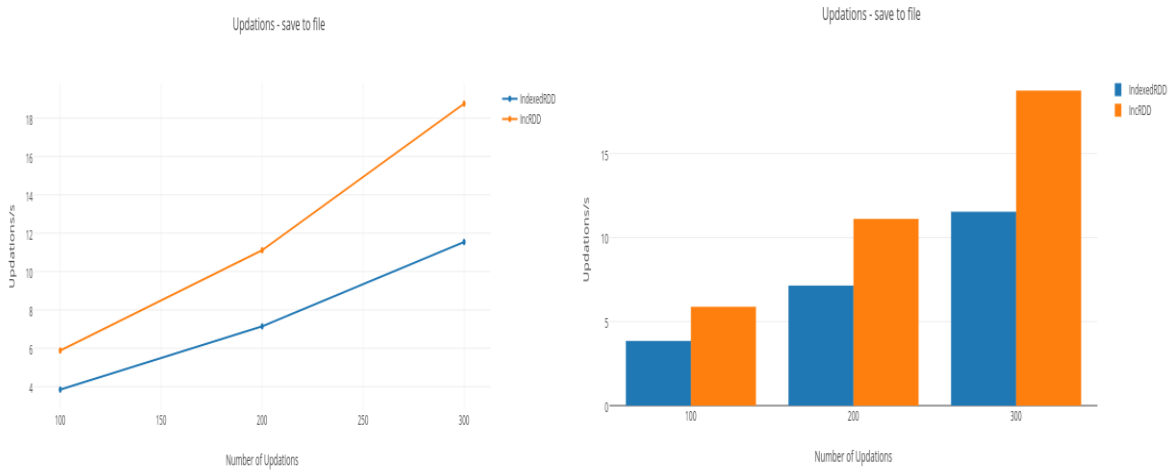

Figure 7.6. Incremental updates to RDD

7.3 Real-time Sampling with updates

Spark is used to sample large datasets and perform analysis. However, the sampled data can incur changes with real-time data. Samples include a dataset containing weather changes, flight schedules, sensor data, mobile generated data and other reports.

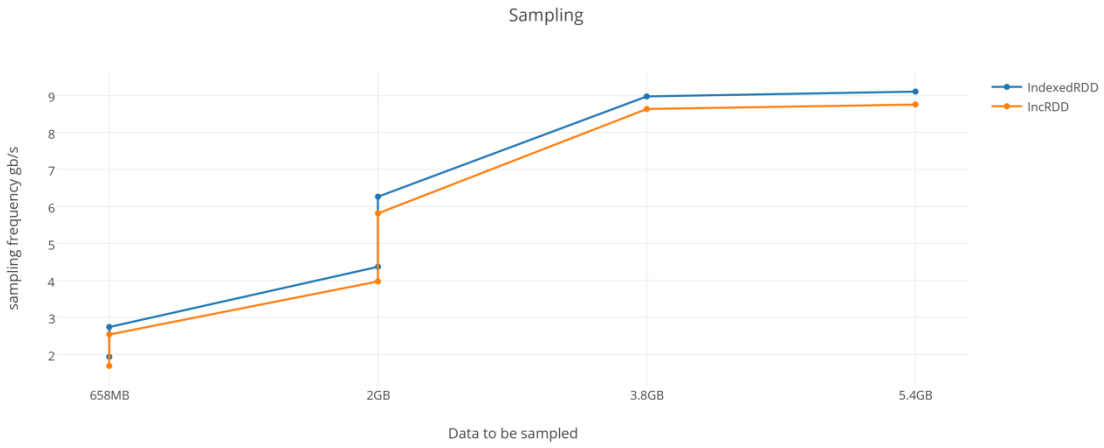

Figure 7.7. Real-time sampling with updates

In this experiment, we use Spark to perform sampling and store the sample with incremental updates to RDD and perform few operations on sampled data. Operations include additions, modifications and deletions in the sampled data.

The dataset used in experiment is from Airline dataset [35] which consist of flight schedules of US domestic flights from 2008. Incremental dataset was used to perform evaluation of IncRDD to sample the dataset and execute the operations. A fraction of dataset was sampled and incrementally updated in IncRDD and finally stored into HDFS file system. As shown in Figure 7.7, the performance of IncRDD was captured against the existing solution. It depicts clearly that in an average case, IncRDD performs 15% faster sampling and execution of incremental operations than other existing solutions.

CHAPTER 8

CONCLUSION

8.1 Contributions

We proposed the solution, 'IncRDD', to support efficient incremental updates in Apache Spark's RDD. This framework presented new methods to introduce mutability in the Spark RDDs. The original contribution of the thesis is to implement incremental updates in Spark but retaining all advantages of Spark's immutability. However, implementing a new cuckoo hashing variant, Dual CHFast-Simple, with carefully chosen hashing scheme, has proved to have better efficiency in supporting immutability in RDDs.

The existing solutions were more complicated to implement and has lower efficiency due to worse average case performance. This study benchmarked the solution with previous solutions, in order to compare the efficiency of each operation of Incremental RDD. Our experiments clearly indicate that, by carefully choosing hashing scheme and hashing function, the updates perform significantly better than the existing adaptive radix trees.

At the end, we presented experiments on real-time data to test performance of IncRDD under large datasets. We demonstrated that IncRDD is a superior alternative to the traditional solutions for updates in RDD. The Experiments supported the claim that the efficiency of incremental updates with IncRDD, increases with large datasets on Spark system.

While each operation challenges for optimal performance, such as implementing algorithms to best utilize the Spark platform and retaining Spark's advantages, this study shows an efficient method for incremental updates to Spark platform. In addition to incremental updates, IncRDD also provides fine-grained operations for insertions and deletions of data in RDD.

8.2 Future Work

In the future, we intend to extend IncRDD to support mutability / incremental updates in accelerated RDD operations including reduce, filter and persist. Also, IncRDD should support special Joins including full outer joins, inner joins, and re-indexing.

Spark supports two kinds of shared variables [29]: broadcast variables, which are used to cache the values in memory on all the worker nodes, and accumulators, which are only added to support counters. There could be instances where the data shared by these shared variables need support few updates. IncRDD can be extended to implement incremental updates to data stored in shared variables in Spark. New algorithms can be developed to work with scalable and heterogeneous data that is being shared among all the worker nodes in Spark platform.

IncRDD can be applied to Incremental checkpointing in Spark to provide efficient fault tolerance for RDDs. We also intend to work on synchronization of concurrent, incremental updates. We plan to create new APIs to IncRDD to support incremental processing in all the above mentioned functionalities of Spark.

REFERENCES

[1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proc of the 6th Conference on Symposium on Operating Systems Design & Implementation (OCDI)*, 2004, pp. 10-10.

[2] Y. Zhang, S. Chen, Q. Wang and G. Yu, "i2MapReduce: Incremental mapreduce for mining evolving big data," *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, Helsinki, 2016, pp. 1482-1483.

[3] C. Yan, X. Yang, Z. Yu, M. Li and X. Li, "IncMR: Incremental Data Processing Based on MapReduce," *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, Honolulu, HI, 2012, pp. 534-541.

[4] Q. He, C. Du, Q. Wang, F. Zhuang, and Z. Shi, “A parallel incremental extreme svm classifier,” Neurocomputing, 2011, vol. 74, no. 16, pp. 2532– 2540.

[5] D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” In Proc. of OSDI ’10, 2010, pp. 1–15.

[6] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, “Stateful bulk processing for incremental analytics,” In Proc. of SOCC ’10, 2010.

[7] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” In Proc. of SOSP, 2013, pp. 439–455.

[8] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In Proc. of SOCC ’11, 2011.

[9] S. Sakr, M. Gaber, “Large Scale and Big Data: Processing and Management”, Auerbach Publications, 2014.

[10] M. Zaharia, M. Chowdhury, M.Franklin, S. Shenker, I. Stoica, “Spark: Cluster Computing with Working Sets”, In Proc. of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, 2010, pp. 10-10.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, I. Stoica , "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation* (NSDI'12). USENIX Association, Berkeley, CA, USA, 2012, pp. 2-2.

[12] "Apache Spark – lightning-fast cluster computing". [Online]. Available: http://spark.apache.org/

[13] "Spark summit". [Online]. Available: https://spark-summit.org/

[14] "Spark programming guide". [Online]. Available: http://spark.apache.org/ docs/latest/programming-guide.html

[15] "Apache Spark". [Online]. Available: https://en.wikipedia.org/wiki/Apache_Spark

[16] C. Armbrust , R.Xin , C. Lian , Y. Huai , D. Liu , J. Bradley , X.Meng , T. Kaftan , M.Franklin, A. Ghodsi , M. Zaharia, "Spark SQL: Relational Data Processing in Spark," In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data* (SIGMOD '15). ACM, New York, NY, USA, 2015, 1383-1394.

[17] A. Dave, "Indexedrdd". [Online]. Available: https://github.com/amplab/spark-indexedrdd

[18] M. Odersky, P. Altherr, V. Crement ,I. Dragos, "An Overview of Scala Programming Language, " Second Edition, Switzerland, 2006

[19] V.Fedorav, "How Near Real-Time Data Processing is Changing Business", 2015. [Online]. Available : http://www.dbta.com/Editorial/Trends-and-Applications/How-Near-Real -Time-Data-Processing-is-Changing-Business-102954.aspx

[20] A. Dave," IndexedRDD: Efficient Fine-Grained Updates for RDDs" presented at Spark Summit, 2015, [Online]. Available: https://spark-summit.org/2015/events/indexedrdd-efficient-fine-grained-updates-for-rdds/

[21] A. Dave," IndexedRDD: Efficient Fine-Grained Updates for RDDs" presented at Spark Summit, 2015, [Online]. Available: http://www.slideshare.net/SparkSummit/ankur-dave

[22] Birreil Michael B. Edward P. Wobber, "A Simple and Efficient Implementation for Small Databases," In *Proc. of the eleventh ACM Symposium on Operating systems principles*(SOSP '87). ACM, New York, NY, USA, 1987, pp.149-154

[23] V. Leis, A. Kemper, t. Neumann, "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases," In Proc. Of 2013 IEE International Conference on Data Engineering (ICDE), pp. 38-49, 2013

[24] R. Pagh and F. Rodler, “Cuckoo Hashing,” In Journal of Algorithms, pages 122–144. Elseiver, 2004.

[25] M. Dietzfelbinger and U. Schellbach, “On risks of using cuckoo hashing with simple universal hash classes,” in SODA, pp. 795–804, 2009.

[26] M. Thorup, “String Hashing for Linear Probing,” In Proc. of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 655-664, 2009.

[27] K. Shwarz, “CuckooHashmap.java”. [Online]. Available: http://www.keithschwarz.com/interesting/code/?dir=cuckoo-hashmap

[28] ”Spark Programming guide”. [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html

[29] R. Pagh,” On the Cell Probe Complexity of Membership and Perfect Hashing,” In Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC 01), ACM Press, pp 425–432, 2001.

[30] Y. Arbitman, M. Naor, G. Segev, G, “De-amortized Cuckoo hashing: Provable worstcase performance and experimental results,” In: ICALP, pp. 107–118, 2009.

[31] “Statistical Computing Statistical Graphics”, 2009. [Online]. Available: http://stat-computing.org/dataexpo/2009/

[32] S. Wadkar, “Sampling large Datasets using Spark”, 2015. [Online]. Available: http://www.bigsynapse.com/sampling-large-datasets-using-spark

[33] W. Wu *et al*., "Minimizing makespan and total completion time in MapReduce-like systems," *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, Toronto, ON, 2014, pp. 2166-2174.

[34] Weili Wu, Hong Gao and Jianzhong Li, "New Algorithm for Computing Cube on Very Large Compressed Data Sets," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 12, pp. 1667-1680, Dec. 2006.

[35] Z. Lu, Y. Shi, W. Wu and B. Fu, "Data Retrieval Scheduling for Multi-Item Requests in Multi-Channel Wireless Broadcast Environments," in *IEEE Transactions on Mobile Computing*, vol. 13, no. 4, pp. 752-765, April 2014.

[36] Z. Lu, W. Wu and B. Fu, "Optimal Data Retrieval Scheduling in the Multichannel Wireless Broadcast Environments," in *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2427-2439, Dec. 2013.

VITA

Prathish Dodabelle Prakash was born in Bangalore, India. He received his Bachelor's degree in Computer Science from Visveswaraya Technological University, India, in 2012. He has 3 years of professional experience as Software Engineer and Build Manager in Evolving Systems. In August 2015, he entered The Erik Jonsson School of Engineering and Computer Science at The University of Texas at Dallas for his Master’s program. In his graduate school, he has completed many projects in virtual reality, machine learning and big data management and analytics. His research interests include Big data analysis, Virtual reality, Apache Spark, Hadoop MapReduce, Data mining and information management.