# Implementations of iterative algorithms in Hadoop and Spark

by

Junyu Lai

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Facing the challenges of large amounts of data generated by various companies (such as Facebook, Amazon, and Twitter), cloud computing frameworks such as Hadoop are used to store and process the Big Data. Hadoop, an open source cloud computing framework, is popular because of its scalability and fault tolerance. However, by frequently writing and reading data from the Hadoop Distributed File System (HDFS), Hadoop is quite slow in many applications. Apache Spark, a new cloud computing framework developed at AMPLab of UC Berkeley, solves this problem by caching data in memory. Spark develops a new abstraction called resilient distributed dataset (RDD) which is both scalable and fault-tolerant. In this thesis, we describe the architecture of Hadoop and Spark and discuss their differences. Properties of RDDs and how they work in Spark are discussed in detail, which gives a guide on how to use them efficiently. The main contribution of the thesis is to implement the PageRank algorithm and Conjugate Gradient (CG) method in Hadoop and Spark, and show how Spark out-performs Hadoop by taking advantage of memory caching.

**Acknowledgements**

I am thankful to my supervisor, Professor Hans De Sterck, who guided me inspirationally throughout my master's study. Without his superb diligence, kindness and patience, this thesis would not have been possible. Additional thanks goes to my committee members, Lilia Krivodonova and Marek Stastna. I would also like to thank Professor Nathalie Lanson for teaching me numerical analysis. Moreover, I want to thank Chen Zhang, who is also my supervisor's student, for offering me helpful suggestions when I was facing difficulties in Hadoop. Finally, I would like to thank Anthony Caterini and Shawn Brunsting for giving me valuable feedback and critique when I was writing the thesis.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Iterative methods

Iterative numerical methods are used in many applications. Many algorithms in scientific computing, such as the Newton method, the Jacobi method, and the Pagerank algorithm, are iterative in nature and they often play vital roles in application areas such as computer science, physics, finance, etc. For example, iterative methods can be used in solving large linear systems which is an essential part of scientific computing and engineering. One may encounter this problem when some equations need to be discretized. Given a hyperbolic partial differential equation (PDE), it might be necessary to solve a system of numerical equations in each time step when applying an implicit time integration method. Using the finite difference method (FDM) for elliptic PDEs may also result in large sparse linear systems in which the matrix is symmetric positive definite (SPD) [1]. In such cases, the Conjugate Gradient (CG) method, an iterative algorithm to solve the linear system, might be a good choice. Machine learning also uses many iterative algorithms, such as K-means which is used in signal processing, clustering, feature learning, etc. It is also worth to mention the Pagerank algorithm, which is an iterative algorithm pioneered at Google to measure the relative importance of websites and compute a rank for each web page based on the web graph [2].

Matrix-Vector Multiplication, a basic operation in numerical linear algebra, plays key roles in some iterative methods such as the Conjugate Gradient method, the Lanczos algorithm for computing eigenvalues and eigenvectors and singular value decomposition (SVD). The Pagerank algorithm also involves Matrix-Vector Multiplication when it computes each

page's new rank obtained from other pages in each iteration. In this thesis, we are concerned with these kinds of iterative algorithms and their implementations.

## 1.2 Cloud computing

Cloud computing, which provides a homogeneous operating environment and full control over dedicated resources, has become more and more popular recently, both in the research and commercial arenas [3]. Google's web processing system is a well-known framework for cloud computing and originally had three main components: MapReduce, Google File System (GFS) and BigTable. The MapReduce programming model, in which the users can specify a map function that processes input data to generate a set of intermediate key/-value pairs and a reduce function that merges all intermediate values associated with the same intermediate key, can be implemented scalably on a large cluster of commodity machines [4]. The Google File System is, a scalable distributed file system for large distributed data-intensive applications, provides fault tolerance while running on inexpensive commodity hardware, and delivers high aggregate performance to a large number of clients [5]. BigTable is a distributed storage system built upon GFS for managing structured data and is designed to scale to a very large size. It provides a flexible, high-performance data store for Google products (such as web indexing, Google Earth, and Google Finance) [6]. GigaSpaces [7], Elastra [8], and Amazon [9] are well-known cloud computing providers. Many open-source software frameworks that provide the foundation for cloud computing environments have recently appeared, including Apache Hadoop [10] and VMware's Cloud Foundry [11].

As a framework for cloud computing, Hadoop has three main components analogous to Google's components: the Hadoop Distributed File System (HDFS), Hadoop MapReduce, and HBase. It is a software project supported by the Apache Software Foundation [12] (an American non-profit corporation) and written in Java. It can also be used with Python and C++. Many companies like Facebook, Yahoo!, LinkedIn and Twitter are using Hadoop as part of their computing infrastructure. Other related projects based on the Hadoop platform include Hadoop Yarn (a resource negotiator for managing running applications), Pig and Hive (tools similar to SQL), ZooKeeper (for configuration services) and others.

Hadoop is well-known for its scalability and fault tolerance; however, it can be very slow compared with solutions based on other platforms like the Message Passing Interface (MPI) since Hadoop frequently writes and reads data from HDFS instead of caching it in memory. To overcome this drawback, a new cluster computing framework called Apache Spark [13], which can be much faster than Hadoop while keeping the scalability and fault tolerance,

has been developed at AMPLab of UC Berkeley. Spark is implemented in Scala (an object-oriented and functional programming language which is simpler and more compact than Java), Java, and Python. Spark is built on top of HDFS and has four main high-level tools: Shark for SQL, MLlib for machine learning, GraphX for graph computation, and Spark Streaming for stream processing [14]. Replacing the MapReduce model in Hadoop, a new abstraction called resilient distributed dataset (RDD) is developed by Spark. RDDs are fault-tolerant, parallel structures that allow users to persist intermediate data in memory, control its partitioning and manipulate it using the operators provided by Spark, enabling efficient data reuse in various applications, including, in particular, iterative algorithms [15].

## 1.3 Motivation and thesis outline

Companies now face the challenges of large amounts of data that are too large to fit in memory in any single machine. For example, by July 2010, Facebook handled about 50 billion photos in total and dealt with 130 terabytes (TB) of logs every day [16]; as of 2005, Amazon had three Linux databases, with a total capacity of 7.8 TB, 18.5 TB and 24.7 TB respectively [17]; as of 2013, E-bay stored about 90 petabytes of data in three systems: Teradata enterprise data warehouse, commodity Hadoop clusters and a custom system for deep-dive analysis [18]. They may use HDFS to store large files and want to do advanced analysis which requires iterative algorithms with efficient execution. Big Data software such as Hadoop is used for large scale data processing and some iterative algorithms have been parallelized and implemented based on MapReduce as a result. For example, the Generalized Iterated Matrix-Vector multiplication (GIM-V) in Pegasus, a library implemented on top of the Hadoop platform, is optimized with block-multiplication [19]; the clustering of growing volumes of data is a challenging task, for which parallel K-means clustering based on Hadoop has been developed [20]. However, Hadoop frequently writes and reads data from HDFS, making implementations of those algorithms inefficient. Spark solves this problem by performing in-memory computations and makes it more worthwhile to implement those iterative algorithms in Spark.

Our work investigates implementation of iterative algorithms in Hadoop and Spark. The rest of the thesis is organized as follows. In Chapter 2, we will discuss background on the architecture of Hadoop and Spark and give some examples. In Chapter 3, we give a brief description of the algorithms to be implemented. Chapter 4 shows how those algorithms are implemented in Hadoop/MapReduce and presents the performance tests in Hadoop. In Chapter 5, we will investigate the details of how RDD transformations work in Spark. In Chapter 6, we describe the algorithm implementations and optimizations in Spark. The

Spark performance of our implementations is discussed in Chapter 7. Chapter 8 concludes the thesis, discussing some limitations and future work.

# Chapter 2

# Background

## 2.1 Apache Hadoop

Many parallel computing application programming interfaces like MPI divide large computing problems into smaller ones which are distributed among different central processing units (CPUs) and executed simultaneously. However, the necessity of communications between processors makes fault tolerance a challenge for parallel computing as each node is required not to fail. This deficiency is overcome in Hadoop by limiting these kinds of communications. We focus our discussion on the HDFS and MapReduce components of Hadoop. Hadoop runs on clusters composed of nodes, one of which is designated the master node, and the other nodes are called slaves. A node normally corresponds to a physical machine (a "box") and usually has multiple processors and cores.

### 2.1.1 Hadoop distributed file system

The Hadoop distributed file system (HDFS) is a highly fault-tolerant distributed file system designed for big data storage and to run on clusters of commodity machines. HDFS has three main daemon processes: a NameNode, several DataNodes, and a Secondary NameNode. Running on the master node in HDFS, the NameNode acts as the manager of the whole file system and is, thus, more important than other nodes. The metadata (including the names of files and directories, the permissions, and the locations of their blocks) for HDFS are stored in the NameNode which does not store the data blocks. It is recommended to use the most powerful node as the master node. Note that once the NameNode fails

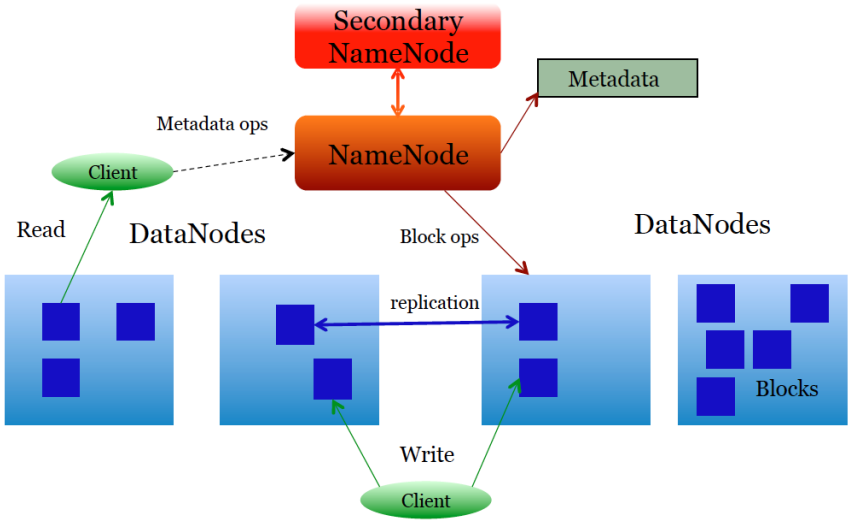



Figure 2.1: HDFS architecture

it will not be restarted automatically (it is a single point of failure). DataNodes, running on slave nodes, are responsible for storing the data blocks and executing instructions from the NameNode. Usually, each slave machine has one DataNode daemon. The Secondary NameNode acts as an assistant for the HDFS cluster: it communicates periodically with the NameNode to download the metadata in order to perform periodic checkpoints. If the NameNode fails it can be restarted on the same physical machine without restarting other DataNodes. However, the Secondary NameNode can never become a NameNode and cannot be used as a substitute when the NameNode fails, since it cannot connect to the DataNodes like the NameNode.

Figure 2.1 shows the HDFS architecture and how the NameNode and DataNodes work. The input data is broken into many blocks and these data blocks will be replicated for fault-tolerance and stored in many different DataNodes. The NameNode decides how to break the initial file into blocks and how to replicate those blocks across the DataNodes, and sends instructions to do so to the DataNodes. The client communicates with the NameNode to get information about the files it needs to access. Then the client writes and reads those data blocks through the corresponding DataNodes.

Figure 2.2 gives an example of data replication in HDFS. We assume two files are stored in HDFS: foo and bar with size of 192 MB and 128 MB, respectively. The default block size for Hadoop (dfs.block.size) is 64 MB which we will keep for this example. Each block is

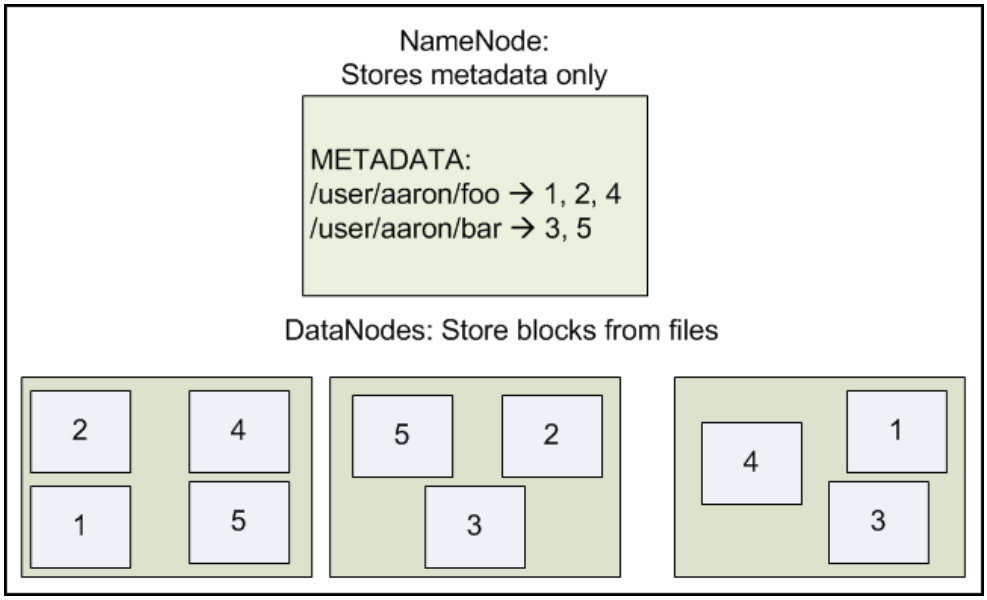



Figure 2.2: Data replication. (Figure from [21].)

replicated twice by setting the parameter dfs.replication to 2 (it is 3 by default). The file foo is split into three blocks (we name them 1, 2 and 4) and file bar is split into two blocks (we name them 3 and 5). Those blocks are replicated twice and then stored across the three DataNodes in the HDFS cluster. For example, the two copies of block 1 are stored in the first and third DataNodes. If the client wants to read the file bar, for example, it will contact the NameNode to obtain the information about this file (including the storage scheme of the blocks of file bar). Then the client knows the block copies of 3 are located in the DataNodes 2 and 3 while the block copies of 5 are located in the DataNodes 1 and 2. The client may contact the DataNodes 2 and 1 to read the blocks 3 and 5, which has a higher degree of parallelism than, for example, reading both blocks 3 and 5 from DataNode 2. If one DataNode fails, say the first one, the client can still read the files foo and bar by communicating with the rest of the DataNodes, 2 and 3, as all the blocks needed are still available in these DataNodes. The parameter dfs.block.size determines the size each block occupies in HDFS. A file of size 200 MB is broken into blocks of sizes 64 MB, 64 MB, 64 MB and 8 MB, if the default block size is set in the configuration file. In this case, the block size can be changed to 50 MB to make the size of the data blocks balanced if one wants to split the file into 4 blocks. Choosing the block size is important in Hadoop as each block will usually be assigned one Mapper process, as we will discuss later.

### 2.1.2 MapReduce in Hadoop

Each MapReduce job is composed of tasks which may be executed in parallel. We first describe two daemons in Hadoop MapReduce: JobTracker and TaskTracker. Just like the NameNode in HDFS, every MapReduce cluster has only one JobTracker which is run on the master node. It is responsible for determining the execution plan including: determining which files to process, assigning nodes to different tasks, and monitoring all running tasks [22]. Each slave node usually has one TaskTracker which is used to execute the tasks assigned by the JobTracker.

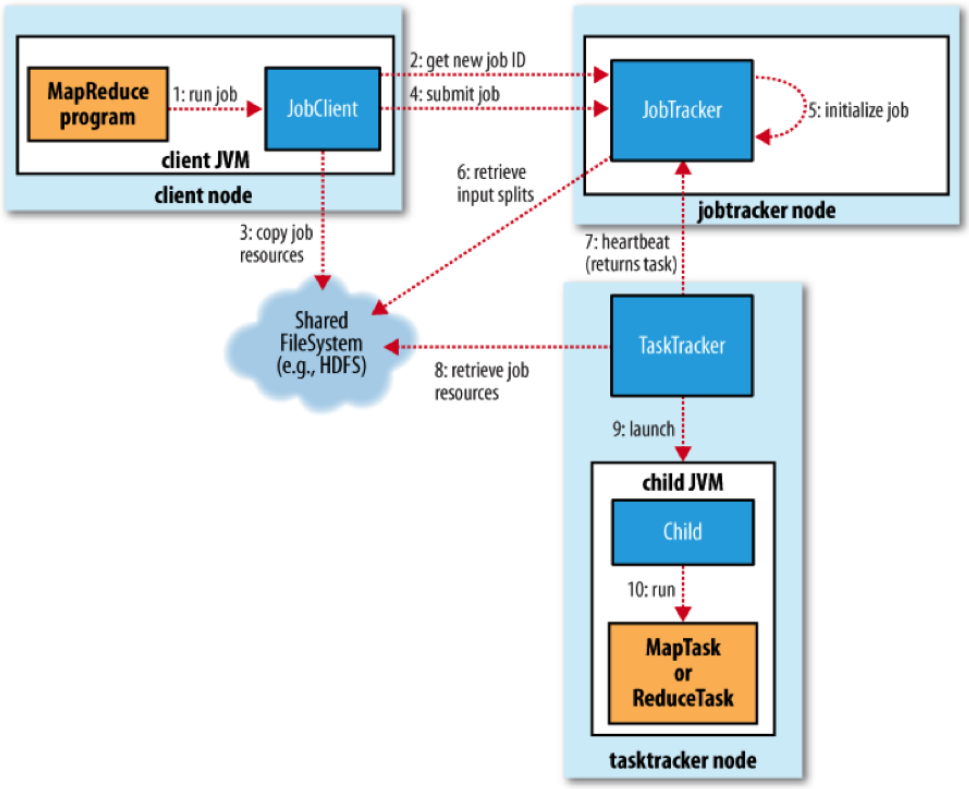



Figure 2.3: MapReduce in Hadoop. (Figure from [23].)

Figure 2.3 illustrates how to run a MapReduce job in Hadoop. Once the JobClient (the primary interface for the user-job to interact with the cluster) submits the MapReduce job, the JobTracker initializes the job and retrieves the input splits (the blocks, where one Map task will be started for each block) from HDFS. The input splits are determined by the

block size which determines the number of blocks in the input file before replication. By communicating with the JobTracker periodically, the TaskTracker in each slave node can let the JobTracker know about its availability (whether or not it is alive and whether or not it is ready to start a new task). New tasks are assigned to TaskTrackers according to their availability. If the JobTracker does not receive a heartbeat message from a TaskTracker within a certain time interval, then this TackTracker is assumed to be stale and the tasks running on this node can be restarted on other healthy nodes for fault-tolerance. To run a task, the TaskTracker retrieves job resources from the HDFS and launches the Java Virtual Machine (JVM) in which the task will be running. Each task has its own JVM. Starting the JVM takes about one second overhead. However, JVM reuse can be enabled in Hadoop when there are many short-time tasks (a JVM can run multiple tasks sequentially to reduce the total overhead).

Now we provide some details on how MapReduce works in Hadoop. Figure 2.4 illustrates the high-level MapReduce process. MapReduce has two phases: mapping and reducing. During the mapping phase, the TaskTracker on each slave node starts one Mapper (a Map task) for each input split assigned to the node. A Mapper is responsible for reading the records from its split, applying the map function (defined by the user) to each record, and outputting intermediate (key, value) pairs which are written to the local disk. The map outputs are divided into partitions corresponding to the reducers that they will ultimately be sent to before they are written to the disk [23]. The Partitioner will decide which partition an intermediate (key, value) pair should go to and all (key, value) pairs with the same key will be put together in the same partition. The default Partitioner is the HashPartitioner which partitions the keys based on their hash values. The reducing phase has two main processes: shuffle and reduce. After a Mapper has finished, each Reducer (Reduce task) starts copying its partition from the intermediate outputs of this map task, which is known as shuffling. This is the only process that requires node communication in MapReduce. After the shuffling is finished, which means all relevant map outputs have been copied, the inputs to each Reducer are merged so that the values with the same intermediate key will be merged together to form a new (key, values[]) pair, where values[] means a sequence of values. Then the reduce function is applied to the merged (key, values[]) pairs and the outputs are written to HDFS. Unlike the number of Mappers which is determined by the number of blocks in the input, the number of Reducers can be specified by the mapred.reduce.tasks parameter as desired by the user and each partition is assigned to one Reducer.

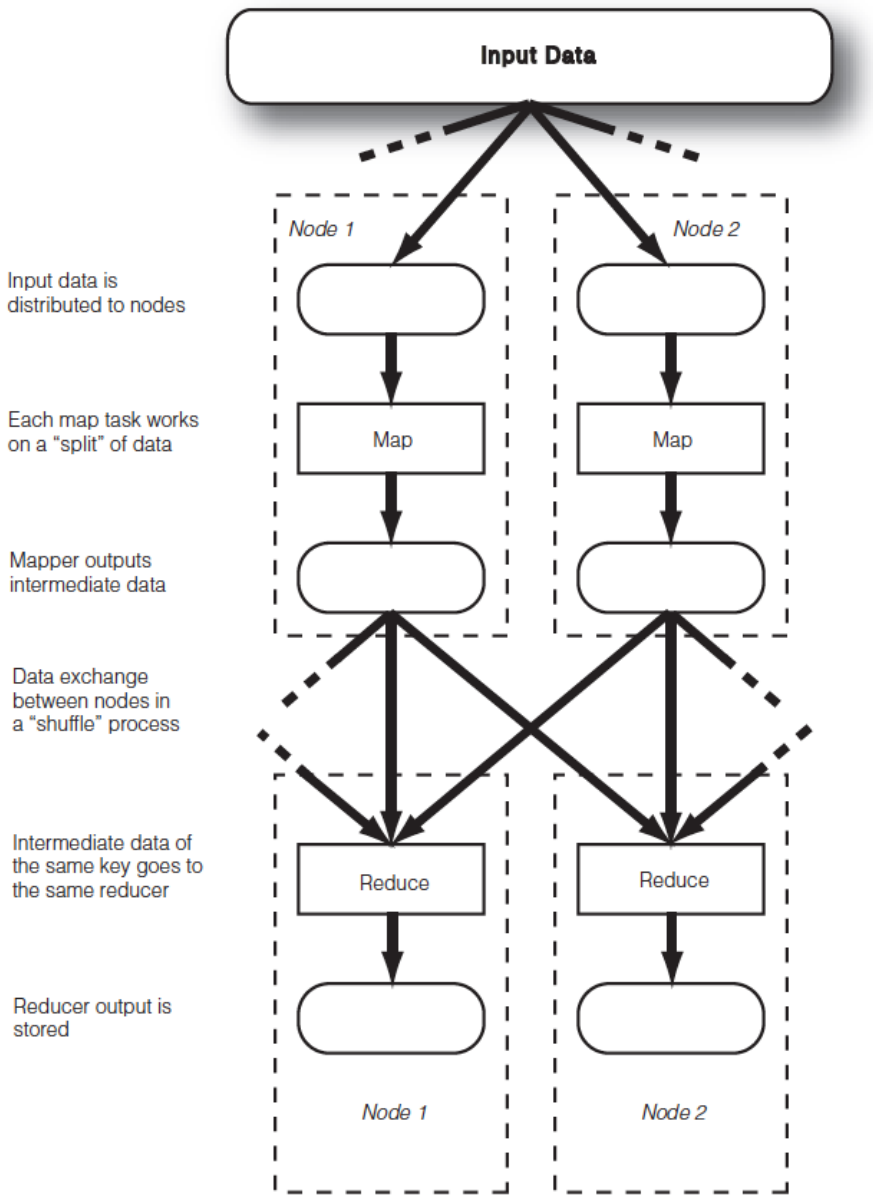



Figure 2.4: The high-level MapReduce dataflow. (Figure from [22].)

### 2.1.3 MapReduce example

We present the WordCount example, which counts occurrence of each word in a text file, to further understand MapReduce programming in Hadoop. We use the example from the MapReduce tutorial [24]. At first, we write the Map and Reduce classes in the WordCount class of the WordCount.java program:

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    // define the map function
    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String line = value.toString();
        // split the line to obtain the word sequence
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            // collect the map (key, value) pairs (intermediate outputs)
            output.collect(word, one);
        }
    }
}
public static class Reduce extends MapReduceBase implements
    Reducer<Text, IntWritable, Text, IntWritable> {
    // define the reduce function
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        int sum = 0;
        // count how many times the word "key" occurs by adding all the values
            (1) together
        while (values.hasNext()) {
            sum += values.next().get();
        }
        // collect the reduce outputs
        output.collect(key, new IntWritable(sum));
    }
```

```
}
```

In the Map class, we define a map function which is applied to each (key, value) pair read from the input records. In this example, the input file is the text file with several words in each line. Using the TextInputFormat (default), a Mapper reads the records from the split line by line, passing the byte offset of the line to the key and the line contents to the value to form an input (key, value) pair for each line. In the map function, the line (value) is split into words and each word is used to output a new (key, value) pair (word, one) ("one" stands for one occurrence and is used to count the number of occurrences of this word). The reduce function in the Reduce class is applied to each (key, values[]) pair from the intermediate data after shuffling. In this function, values is the iterator for the sequence of ones which share the same key (the word). Those ones in the sequence are reduced to calculate the total number of occurrences of the word which becomes the value of the final output (key, value) pair.

After defining the Map and Reduce classes, we need to configure the job and submit it in the main function:

```
public static void main(String[] args) throws Exception {
    // Create a new JobConf
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    // set the key, value type for each output (key, value) pair
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    // set the Map, Reduce, and Combine class
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    // set the Input (Output)Format
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    //specify the total number of Reducers:
    conf.setNumReduceTasks(Integer.parseInt(args[2]));
    //set the input (output) path in HDFS
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    //submit the job
    JobClient.runJob(conf);
}
```

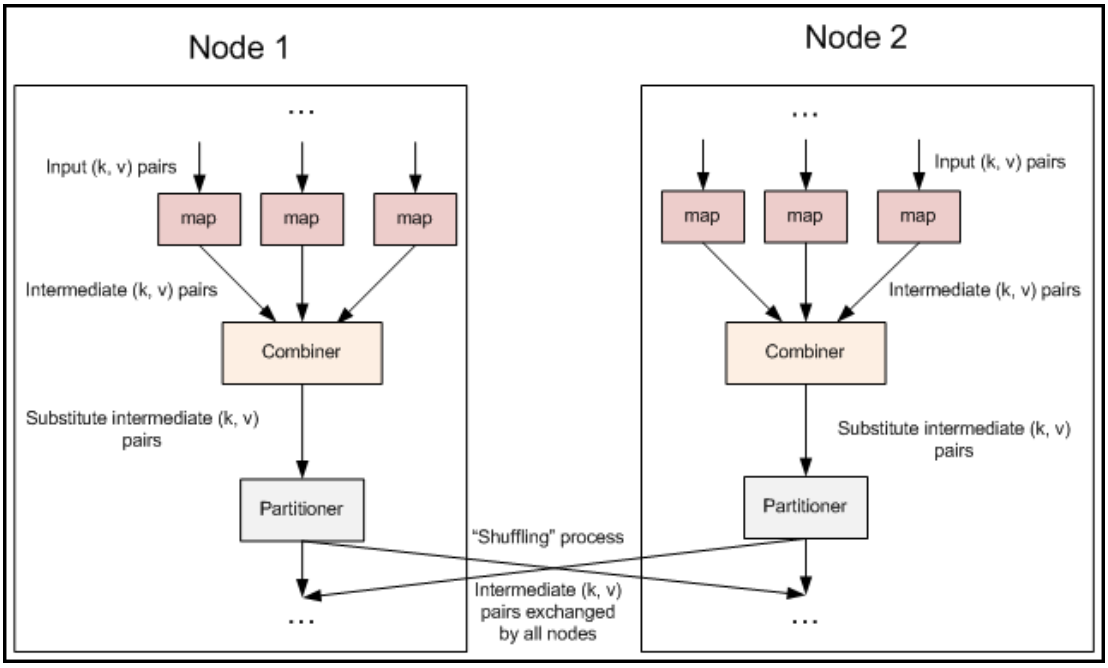



Figure 2.5: Combiner step inserted into the MapReduce data flow. (Figure from [25].)

The JobConf object conf is created to specify the job parameters. Then the JobClient submits the job according to the configuration in conf. Setting a CombinerClass like in this example, the outputs from a map process are combined for efficiency before being sent to the Reducers. For example, if a Mapper outputs several (key, value) pairs like: (word1, 1), (word2, 1), (word1, 1), (word1, 1), (word2, 1), (word3, 1), they are combined into (word1, 3), (word2, 2), (word3, 1). Only three (key, value) pairs are then sent to the Reducers instead of six without using the combiner function, which reduces the data transferred between nodes in the shuffling. Figure 2.5 illustrates how the Combiner works as a pre/mini-Reducer which runs only on the intermediate data generated by one node.

Hadoop is scalable because more nodes can be used as input file increases. Hadoop is fault-tolerant because file blocks are replicated across disks and tasks on failing nodes are restarted on healthy nodes.

## 2.2 Apache Spark

Now we introduce a second cluster computing framework, Apache Spark, which is scalable and fault-tolerant, and can be much faster than Hadoop. Instead of using the MapReduce paradigm like Hadoop, Spark introduces a new parallel and fault-tolerant data structure called Resilient Distributed Dataset (RDD) [15]. Although the Spark library can also be used from Java and Python, we choose the Scala language for our implementations since it results in more compact code.

We first give the meaning of some terms used to refer to Spark concepts [26].

- **Application** User program built on Spark. An application consists of a driver program and several executors on the cluster.
- **Driver Program** The process running the main() function of the application and creating the SparkContext (see section 2.2.1) in the main function. The application name is specified in the SparkContext.
- **Cluster manager** An external service for acquiring resources on a cluster. The Spark system supports three types of cluster managers: Standalone, Apache Mesos and Hadoop YARN. In our work, we only use the Standalone cluster manager.
- **Worker node** Similar to slave node in Hadoop cluster, runs the application in a cluster. The hostnames of all Worker nodes are specified in a file called slaves in the SPARK_HOME/conf directory.
- **Executor** A process launched for an application on a Worker node that runs tasks and keeps data in memory or disk storage. Each application has its own executors. Each Worker node has one Executor.
- **Task** A unit of work that will be sent to one executor. Each task runs on one partition (slice) of an RDD and applies a sequence of RDD transformations to that partition.
- **Job** A parallel computation consisting of multiple stages. A job is triggered by a Spark action (defined in section 2.2.2).
- **Stage** Each job is divided into smaller sets of tasks called stages. Stages depend on each other and are executed sequentially (similar to the map and reduce stages in MapReduce). A stage is divided into several tasks which may be concurrent and

are sent to executors on Worker nodes for execution. Each stage pipelines a series of transformations to its input RDD and the resulting RDD becomes the input data of the next stage.

### 2.2.1 RDD basics

An RDD is a read-only, partitioned collection of elements and can be created from data in stable storage such as HDFS or by transforming other RDDs [15]. To create an RDD from HDFS or the local file system, one first creates a SparkContext object which acts as the main entry point to Spark and represents the connection to the master node. When creating a SparkContext object, one needs to specify the network address of the master node, the application name and the configuration object (used to set various Spark parameters) like this:

```
1 val sc = new SparkContext(master: String, appName: String, conf: SparkConf)
```

Once the SparkContext object sc is created,one can create text file RDDs from text files in HDFS or the local file system using the SparkContexts textFile method. For example, if one wants to create an RDD from the files in the directory "/user/user0/data" in HDFS, one just needs to specify the hostname of the machine where HDFS is located and the directory name in the HDFS using the textFile method:

```
1 val data = sc.textFile("hdfs://[HDFS name]/user/user0/data")
```

Each line of the text file in the HDFS is then read as a String and becomes an element in the newly created RDD. For example, for a text file with three lines like this:

> Hello World Bye World
> Hello Spark GoodBye Spark
> Hello Hadoop Bye Hadoop

the resultant RDD has three String elements: "Hello World Bye World", "Hello Spark GoodBye Spark" and "Hello Hadoop Bye Hadoop". One can also create other types of RDDs using other methods in the SparkContext class. Using the SparkContext's sequence-File[K, V] method, one can create RDDs from the Hadoop SequenceFile consisting of binary (key, value) pairs. One can also create RDDs from data of other InputFormats using, for example, the hadoopFile method.

An RDD can also be parallelized by controlling the number of its slices (or partitions). The number of partitions of an RDD depends on the slicing of the input data. For example, each block of a file in HDFS results in one slice by default. However, one can create multiple slices for each block by passing the total desired number of slices as the second argument to the textFile method when creating a text file RDD. Note that this number must be larger than the total number of blocks as each block must have at least one slice. Choosing a suitable number of slices can improve performance as one processing task is assigned to each slice (this will be discussed in later sections).

### 2.2.2 RDD operations

RDDs have two types of operations (or methods): transformations (which we mentioned in the previous section) and actions. A transformation creates a new RDD by applying a function to each element of an existing RDD, and an action results in a value which is returned to the driver program (such as print final results on the screen or save them as files). RDDs of (key, value) pairs are used in most implementations of Spark and can be building blocks for algorithms based on MapReduce principles. PairRDDFunctions in the Spark library provides operations for (key, value) pair RDDs in addition to the operations for general RDDs. Some RDD operations (including operations in PairRDDFunctions) have similarities to Map and Reduce processes in Hadoop when they are applied to a (key, value) pair RDD. For example, the flatMap() transformation applies a function to each element in a (key, value) pair RDD and then outputs several new (key, value) pairs, which is similar to the Map function in Hadoop. The groupByKey() groups the values for each key in the (key, value) pair RDD into a single sequence, which is similar to the process of forming (key, values[]) pairs in Hadoop.

Now we list some basic RDD operations we use in our work. Here RDD[T] means RDD with elements of type T. A full list of RDD operations (including PairRDDFunctions) can be found in [27] and [28].

**RDD transformations**

- map($func$: T => U): RDD[T] => RDD[U]

  Return a new RDD by applying a function $func$ to all elements of this RDD. For example, given an RDD[String] with three String elements: "1", "2" and "3", we use the function $func$: s => s.toInt which transforms a String to an Int. After the map transformation the resulting RDD[Int] has three Int elements: 1, 2 and 3.

- filter($func$: T => Boolean): RDD[T] => RDD[T]

  Return a new RDD[T] containing only the elements that satisfy the condition of $func$.

- flatMap($func$: T => Seq[U]): RDD[T] => RDD[U]

  Similar to map, but $func$ maps each element to multiple elements. Here Seq[U] means a sequence of elements of type U. After applying $func$ to all elements of this RDD, the results are flattened. For example, given an RDD[String] ("word1 word2", "word3 word4", "word5 word6") and a function $func$ that splits a String into words, then the resulting RDD[String] has six String elements: "word1", "word2", "word3", "word4", "word5", "word6".

- union(*other*: RDD[T]): RDD[T] => RDD[T]

  Return the union of this RDD and another one of the same type (*other* of the type RDD[T]).

- mapValues($func$: V => U): RDD[(K, V)] => RDD[(K, U)]

  Pass each value in the (key, value) pair RDD through $func$ without changing the keys. This transformation also retains the original RDD's partitioning.

- groupByKey(): RDD[(K, V)] => RDD[(K, Seq[V])]

  Group the values for each key in the RDD into a single sequence. This transformation involves shuffling.

- reduceByKey($func$: (V, V) => V): RDD[(K, V)] => RDD[(K, V)]

  Merge the values for each key using an associative reduce function $func$ which satisfies $func(x, y) = func(y, x)$. This transformation also involves shuffling.

- cogroup(*other*: RDD[(K, W)]): RDD[(K, V)] => RDD[(K,(Seq[V], Seq[W]))]

  For each key in this RDD and in *other*, return a resulting RDD that contains a tuple with the list of values for that key in both RDDs.

- join(*other*: RDD[(K, W)]): RDD[(K, V)] => RDD[(K, (V, W))]

  Return an RDD containing all pairs of elements with maching keys in this RDD and *other*. This transformation may involve shuffling between nodes.

**RDD actions**

- reduce($func$: (T, T) => T): RDD[T] => T

  Reduces all elements of this RDD using the specified commutative and associative binary operator to one element and return it to driver program.

- collect(): RDD[T] => Array[T]

  Return an array that contains all of the elements in this RDD.

- count(): RDD[T] => Long

  Return the number of elements in this RDD.

- saveAsTextFile(path: String)

  Write the elements of this RDD as a text file in a given path in the local file system, HDFS or any other Hadoop-supported file system.

- saveAsSequenceFile(path: String)

  Write the elements of this RDD as a Hadoop SequenceFile in a given path in the local file system, HDFS or any other Hadoop-supported file system.

Unlike the RDD actions, the transformations for RDDs are lazy, which means the computation for a new RDD is not executed immediately after the command is given in the code, but only when it is needed for producing the result of an action that comes later. For example, when executing the Spark program:

```
object Filter {
  def main(args: Array[String]) {
    val file = sc.textFile("hdfs://.../README.md") //transformation
    println("Filtering Spark:")
    val spark = file.filter(line => line.contains("Spark")) //transformation
    println("Filtered")
    val sparkcount = spark.count() //action
    println("Count all the Spark:" + sparkcount)
  }
}
```

The result shown on the screen is:

[info] Running WordCount
14/05/06 07:19:38 INFO .....
Filtering Spark:
Filtered
.....
14/05/06 07:19:39 INFO spark.SparkContext: Starting job: count at Filter.scala:7
.....
14/05/06 07:19:38 INFO scheduler.DAGScheduler: Submitting Stage 0 (FilteredRDD[2] at filter at Filter.scala:5), which has no missing parents
.....
14/05/06 07:20:01 INFO spark.SparkContext: Job finished: .....
Count all the Spark:150

As we can see from the screen, the filter (line 5) does not actually happen after "Filtered" (line 6) has been printed to the console. The filtering process was triggered by an RDD action (line 7) at 07:19:39 and finished at 07:20:01. Since RDD transformations are lazy, it is recommended to use immutable variables (val) instead of mutable variables (var) in Scala if RDDs depend on those variables. If we use mutable variables, unexpected results may be obtained if those variables change before the actions on RDDs are executed. To further understand the laziness of transformations of RDDs, we give some examples. First, consider the code fragment:

```
1 val data = Array(1, 2, 3, 4, 5)
2 //creating a mutable variable b
3 var b = 2
4 //creating a new RDD aa which depends on mutable variable b
5 val aa = sc.parallelize(data).map(i => i * b) //transformation
6 b = 3
7 val aarray = aa.collect() //action
```

The SparkContext's parallelize method is used to create parallelized collections from an existing collection (linke Array in this example) in the driver program. Here, an RDD aa with Int elements 1, 2, 3, 4, and 5 is created. Although it is created before the statement "b=3", aarray is (3, 6, 9, 12, 15) instead of (2, 4, 6, 8, 10) at the end of the code fragment because the map transformation in the 5th line is not executed until the collect action is applied to RDD aa in line 7. Below is another example.

```
1 val data = Array(1, 2, 3, 4, 5)
```

```
2  //creating a mutable variable b which refers to an RDD
3  var b = sc.parallelize(data)
4  //creating a new RDD aa which depends on mutable variable b
5  val aa = b.map(i => i * 2)
6  // variable b can be modified, but the underlying RDD is immutable
7  b  = b.map(i => i * 5)
8  // applying action to b to execute the modification
9  val barray = b.collect()
10 val aarray = aa.collect()
```

In this example, the mutable variable b refers to an RDD instead of an other type of object in Scala. However, in this case, the change to b in line 7 does not influence the final value of aarray. The reason is that an RDD has the information (also called lineage) about how it was derived from other RDDs and this information is used to execute the final action (note that RDDs are immutable). So aarray is (2, 4, 6, 8, 10) instead of (10, 20, 30, 40, 50). These issues are not clearly expressed in the code syntax, so the programmer has to be aware of these differences that are implicit in the code syntax.

### 2.2.3 Lineage, persistence and representation

An RDD may be created through several transformations. For example, if one wants to count the number of occurences of each word in a text file, one may type the following Scala code:

```
1 val file = spark.textFile("hdfs://...") // creates and RDD with text line
     elements
2 val lines = file.flatMap(line => line.split(" ")) // transforms the RDD to words
3 val words = lines.map(word => (word, 1)) // transforms the RDD to (word, 1)
     pairs
4 val counts = words.reduceByKey(_ + _) // transforms the RDD to (word, #) pairs
5 counts.saveAsTextFile("hdfs://...") // action that saves the RDD to a text file
```

This creates four RDDs in line 1, 2, 3 and 4. Figure 2.6 shows the lineage graph for those RDDs. The arrows represent the transformations between RDDs. Spark achieves fault tolerance by tracking the lineage for the RDDs. For example, if a slice (partition) of RDD words is lost, Spark tracks the lineage graph and recomputes it by applying transformations flatMap and map to the grandparent of that slice (slice of the RDD file) and then pipelines its own transformation (reduceByKey) to it. Some transformations may take a long time

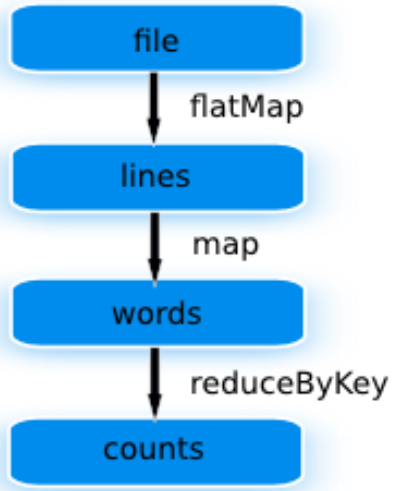



Figure 2.6: The lineage graph for RDDs in the WordCount example

to finish and therefore it may be usefule to cache and reuse RDDs created from them. For example, after the saveAsTextFile action, one may want to get an RDD only with lines containing "Spark" by applying a filter transformation to the RDD lines:

```
lines.filter(_.contains("Spark"))
```

In those cases, it is recommended to persist (or cache) the original RDD lines in memory:

```
val lines = file.flatMap(line => line.split(" ")).cache()
```

because otherwise it may have to be recreated when it is used in a second action.

Should slices of any other RDDs that depend on lines be lost, the transformation is started from the corresponding slices of RDD lines since they are cached in memory. After the RDD lines is cached, it can be transformed to other RDDs directly without re-executing the transformation before it (flatMap) in the lineage. An RDD can be persisted at different levels; the details can be found in [29].

Each RDD is characterized by five pieces of information [15]:

- a list of partitions (slices);
- a list of dependencies on other RDDs;

- a function for computing each partition;
- a partitioner (optional, for RDDs of key-value pairs);
- a list of preferred locations of each partition (optional).

The partitioner of a (key, value) pair RDD decides how the elements are partitioned by key. Note that all elements with the same key are located in the same partition if an RDD of (key, value) pairs has a partitioner. An RDD may not have a partitioner, for example, it is created by reading text file from HDFS. In this case, elements with the same key may be located in different partitions and even different Worker nodes.

### 2.2.4 Running Spark on a cluster

A basic Spark cluster is very similar to a Hadoop cluster. It consists of a cluster manager (the master) and several Worker nodes (similar to the slave nodes in Hadoop cluster). The cluster manager, which acts as the resource manager, is connected to the SparkContext object in the main program (the driver) which is responsible for coordination in the cluster. One executor is launched on each Worker node to run the tasks and persist the data (the slices) on the node. An executor can run multiple tasks simultaneously and the number of tasks is determined by the number of cores in the node (each core can only run one task at a time). Note that one task is executed for each slice (or partition) on each Worker node, but these tasks may not all run concurrently if there are more slices than cores on a node. For example, given an input RDD with 24 partitions for a stage, if there are two nodes in a cluster with 4 and 8 cores respectively, then the first 12 tasks can be executed at the same time in the cluster. Spark can apply the operations to 12 slices (partitions) at the same time. Once a task is finished, the core that runs this task can start executing another task in the same stage.

### 2.2.5 Spark example

In this section, we give the WordCount example to illustrate how Spark schedules jobs and tasks within an application. In this application, we use three machines: gridbase1.math.uwaterloo.ca, gridbase2.math.uwaterloo.ca and gridbase3.math.uwaterloo.ca. Below is the information for each machine:

Each machine is used as a slave node while gridbase 1 acts as master node as well. The Spark code in Scala is as follows:

| gridbase | No. of cores | Memory |
|---|---|---|
| 1 | 4 | 6.8 GB |
| 2 | 8 | 6.8 GB |
| 3 | 16 | 14.7 GB |

Table 2.1: The CPU information for gridbase machines

```
1  import org.apache.spark.SparkContext
2  import org.apache.spark.SparkContext._
3
4  object WordCount {
5    def main(args: Array[String]) {
6      System.setProperty("spark.executor.memory", "1024m")
7      System.setProperty("spark.cores.max", "12")
8      // gridbase1.math is the hostname of the Master node, "WordCount" is the
9      // application name
10     val sc = new SparkContext("spark://gridbase1.math:7077", "WordCount",
           "/home/j33lai/spark-0.8.1-incubating", // Spark home directory
11       List("target/scala-2.9.3/word-count_2.9.3-1.0.jar")) // list of JAR files
12     val file = sc.textFile("hdfs://gridbase1.math:56638/user/j33lai/data")
13     val counts = file.flatMap(line => line.split(" "))
14                      .map(word => (word, 1))
15                      .reduceByKey(_ + _)
16     counts.saveAsTextFile("hdfs://gridbase1.math:56638/user/j33lai/output")
17   }
18 }
```

The memory for each executor is 1024 MB (line 6), which means each machine has to provide this amount of memory to its executor process. This parameter must be lower than the memory available in each machine, otherwise one would get an "out of memory" error. The parameter spark.cores.max determines the total number of cores used in the cluster (line 7). Spark tries to use the cores available on each machine equally so in this case 4 cores will be used on each node. The default port for the master is 7077 so "spark://gridbase1.math:7077" has been put in the first argument of SparkContext (line 10).

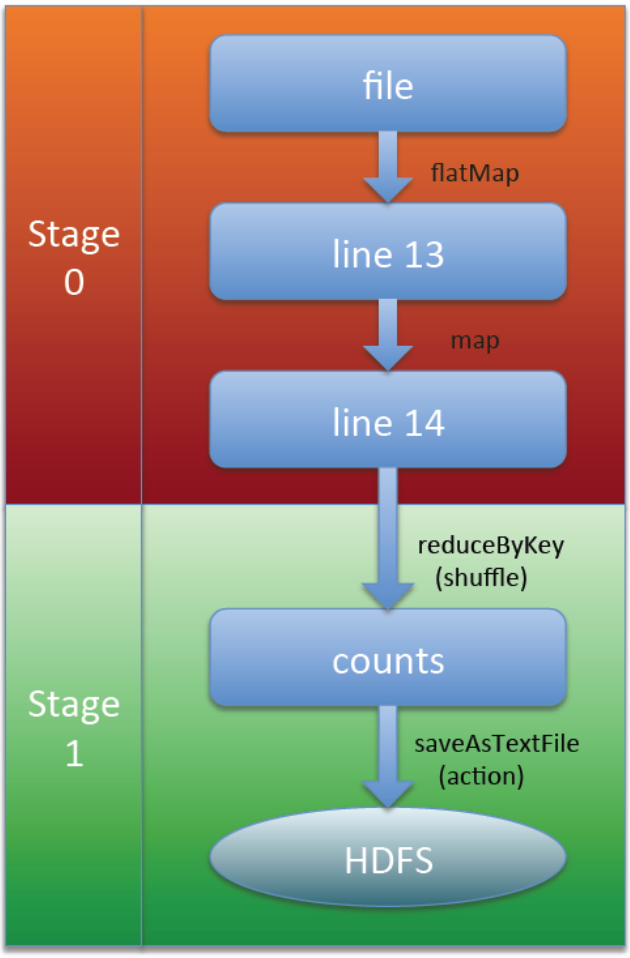



Figure 2.7: WordCount in Spark

Figure 2.7 illustrates how the WordCount application runs in Spark. A blue box in the figure represents an RDD. In this application, we only have one action on an RDD: saveAsTextFile (line 16). Therefore only one job will run. The job is divided into two stages due to the shuffling process that occurs in reduceByKey and these two stages are executed sequentially. Stage 0 contains flatMap and map transformations and ends at the RDD in line 14. The transformation reduceByKey involves shuffling and therefore it starts a new stage. The two stages are similar to the MapReduce stages in Hadoop. Stage 0 is similar to the Map phase in Hadoop; Stage 1 is similar to the Reduce phase in Hadoop: it shuffles and writes results into HDFS. However the "reduce" stage (stage 1) in Spark will not be

launched until the "map" stage (stage 0) is finished. The information (time, RDD storage, etc.) of stages and tasks in an application can be observed visually from the master's web User Interface (`http://localhost:8080`). Figure 2.8 gives an example of monitoring stages in an application.

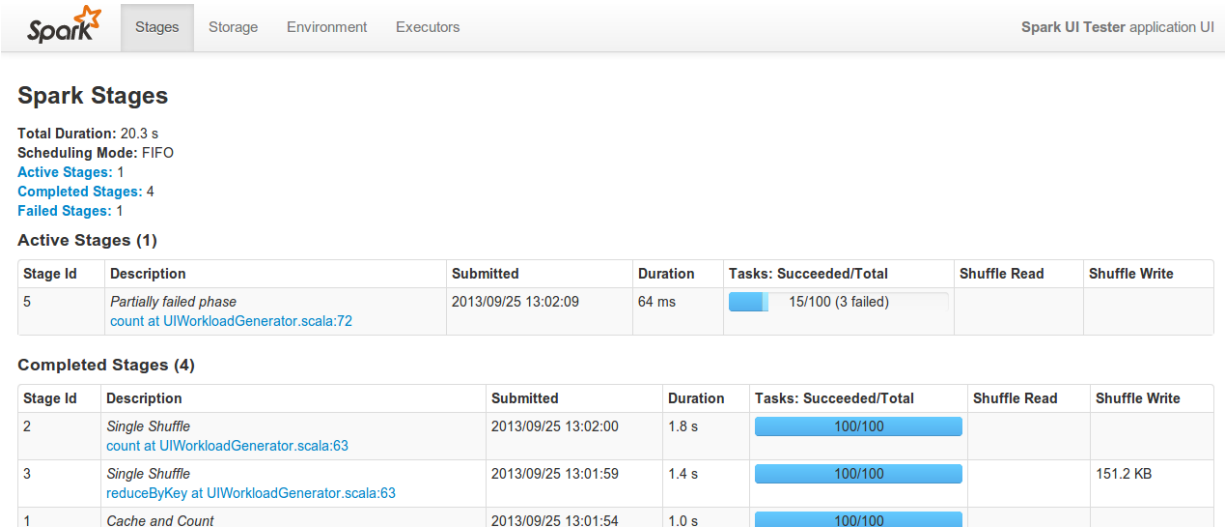



Figure 2.8: Monitoring data in a web UI. (Figure from [30].)

# Chapter 3

# Iterative algorithm description

## 3.1 Pagerank

Consider $N$ web pages: $P_0$, $P_1$,...,$P_{N-1}$ and consider the graph formed by the directed links between the web pages. Assume pages have no self-links. For simplicity, we first assume that every page has outlinks. We also assume that every page has at least one inlink (link from another page). We use $R_k^{(j)}$ to represent the rank of page $P_k$ after the $j$th iteration. For each page $P_i$, we assume it has $N_i$ outlinks and $N_i > 0$ by assumption. Then the simplest version of Pagerank computes the rank of $P_i$ at the $j+1$th iteration as:

$$R_i^{(j+1)} = \sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)} \tag{3.1}$$

Note that reducible graphs have rank sinks (subgraphs that attract all the rank in the graph), but irreducible graphs don't. We say the graph is irreducible if, for every page, there exists a path to all other pages. In case we have a reducible graph, we make the graph irreducible by including a damping factor $\alpha$ and computing

$$R_i^{(j+1)} = (1-\alpha) + \alpha \sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)} \tag{3.2}$$

This effectively reduces the weights of the original graph edges by the factor $\alpha$ , and adds new links from every node to every other node with weight $1-\alpha$. In the context of web surfing, the damping factor $\alpha$ represents the probability that a user will follow a link from the web page they are currently visiting. $1-\alpha$ is the probability that they will jump

to another page by typing in a web address. The damping factor $\alpha$ is normally taken $\alpha = 0.85$ [31]. Once we choose to normalize the ranks as in Page and Brin's paper [2], letting the sum of all ranks be one (such that ranks become probabilities), we have:

$$R_i^{(j+1)} = \frac{1-\alpha}{N} + \alpha \sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)} \tag{3.3}$$

Here $N$ is the total number of pages in the graph. In this section, we assume the ranks are not normalized. In a real network, there may be many dangling nodes (pages) which have no outlinks and the total rank (or probability) decreases if we calculate each rank by equation 3.2 or 3.3. The ranks still converge but the sum of all ranks may be much less than 1 or $N$ (if initial ranks are not normalized). One solution is to distribute the ranks of those dangling nodes to each page equally and the total rank will remain unchanged during the iterations. Then equation 3.2 becomes:

$$R_i^{(j+1)} = \alpha \left( \sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)} + \sum_{\text{dangling pages k}} \frac{1}{N} R_k^{(j)} \right) + 1 - \alpha \tag{3.4}$$

We use $\mathbf{R}^{(j)}$ to represent the rank vector after the $j$th iteration. So the Pagerank algorithm can be also described as the following matrix equation:

$$\mathbf{R}^{(j+1)} = \alpha \left( \mathbf{M} + \frac{1}{N} \mathbf{1}\mathbf{d}' \right) \mathbf{R}^{(\mathbf{j})} + (1-\alpha)\mathbf{1}, \tag{3.5}$$

where $M$ is defined as

$$M_{ij} = \begin{cases} \frac{1}{N_j} & \text{if Page j links to Page i;} \\ 0 & \text{otherwise,} \end{cases} \tag{3.6}$$

and $\mathbf{d}$ is a vector defined as

$$\mathbf{d}_i = \begin{cases} 1 & \text{if Page i has no outlinks (dangling node);} \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

As we can see, the Pagerank algorithm involves matrix-vector multiplications and it can be implemented in Hadoop MapReduce and Spark. The detailed implementations will be discussed in Chapter 4.

## 3.2 Conjugate Gradient Algorithm

The conjugate gradient method is an iterative algorithm for solving linear systems where the matrix is symmetric and positive-definite (SPD). Consider the linear equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is SPD, then the algorithm is given as follows:

**Algorithm 1** Conjugate Gradient algorithm

1: $\mathbf{r_0} \leftarrow \mathbf{b} - \mathbf{Ax_0}$
2: $\mathbf{p_0} \leftarrow \mathbf{r_0}$
3: $k \leftarrow 0$
4: repeat
5: $\quad \alpha_k \leftarrow \frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T A p_k}}$
6: $\quad \mathbf{x_{k+1}} \leftarrow \mathbf{x_k} + \alpha_k \mathbf{p_k}$
7: $\quad \mathbf{r_{k+1}} \leftarrow \mathbf{r_k} - \alpha_k \mathbf{A p_k}$
8: $\quad$ if $\mathbf{r_{k+1}}$ is sufficiently small then exit loop
9: $\quad \beta_k \leftarrow \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
10: $\quad \mathbf{p_{k+1}} \leftarrow \mathbf{r_{k+1}} + \beta_k \mathbf{p_k}$
11: $\quad k \leftarrow k + 1$
12: end repeat
13: The result is $\mathbf{x_{k+1}}$

In this algorithm, there is one matrix-vector multiplication, three vector additions and two vector scalar products in each iteration. After the first iteration, $\mathbf{r_k^T r_k}$ in line 5 can be obtained from the computation in the previous iteration (line 9). For the matrix-vector multiplication and the vector scalar product (which can also be considered a type of matrix-vector multiplication), two MapReduce stages are required in general. However, the vector scalar product $\mathbf{r_k^T r_k}$ only needs one MapReduce stage because it involves the same vector twice. The vector addition needs two stages as well. Therefore eleven MapReduce stages may be required in each iteration. The implementation in Hadoop MapReduce will be discussed in Chapter 4.

# Chapter 4

# Implementations in Hadoop

## 4.1 Matrix and Vector Operations

In this section, we discuss how Matrix-Vector Multiplication, vector scalar product, and vector addition can be implemented in Hadoop MapReduce.

### 4.1.1 Matrix-Vector Multiplication

We first present the Generalized Iterative Matrix-Vector Multiplication (GIM-V) algorithm for Graph Mining Library Pegasus [19]. Given a sparse matrix $\mathbf{A}$ and a vector $\mathbf{x}$, we want to compute $\mathbf{Ax}$. We use $A_{i,j}$ to represent the $i,j$th entry of $\mathbf{A}$ and $x_i$ to represent the $i$th element of $\mathbf{x}$. Figure 4.1 illustrates how the GIM-V method works.

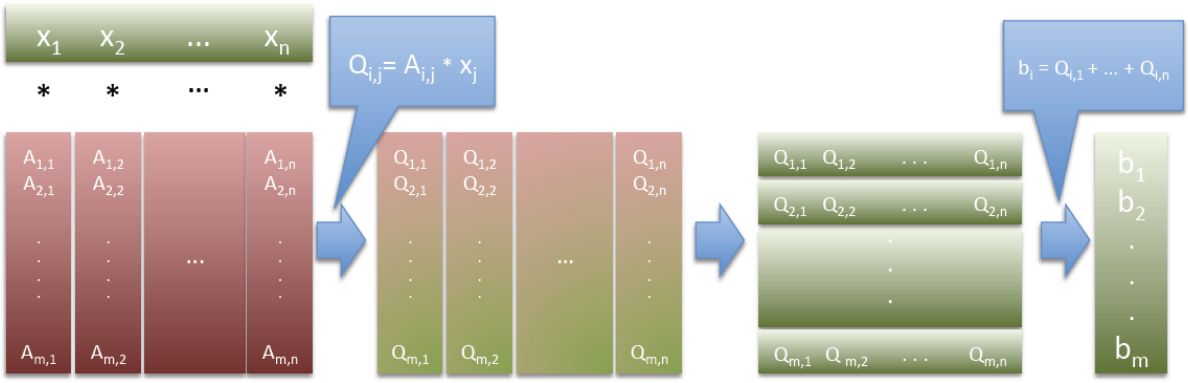



Figure 4.1: $\mathbf{Ax} = \mathbf{b}$ for GIM-V

In our implementation in Hadoop, only nonzeros in the matrix and vector are stored in the input file. The matrix and vector are stored as a text file in HDFS with the following format:

Matrix $\mathbf{A}$

$i_1\ j_1\ A_{i_1,j_1}$
$i_1\ j_2\ A_{i_1,j_2}$
.....
$i_2\ j_1\ A_{i_2,j_1}$
.....
$i_k\ j_l\ A_{i_k,j_l}$

Vector $\mathbf{x}$

$j_1\ x_{j_1}$
.....
$j_s\ x_{j_s}$

For the file that stores a matrix, each line has the row and column information of an element of the matrix. For a vector, each line in the file that stores it contains the row

information of a vector element. When we run Hadoop for GIM-V, matrix and vector files are read together and the Map function can determine whether a line represents a matrix or vector element through its structure. GIM-V is done in two MapReduce stages. In stage 1, the Stage-1 Map function (Figure 4.2) reads each line from the input files (matrix and vector file). If a line represents a vector element, say $x_i$, the Map function outputs $(i,\ x_i)$ with the row index as the key. For a matrix element $A_{i,j}$, it outputs $(j,\ i \quad A_{i,j})$ with the column index as the key, and the row index along with the matrix element as the value. Then matrix elements in the same column are merged together with the corresponding vector element in a shuffle operation. For each matrix element $A_{i,j}$ the Stage 1 Reduce function (Figure 4.3) multiplies it with the vector element $x_j$, and outputs $(i,\ A_{i,j} * x_j)$. The pseudocode for Stage 1 is given as follows:

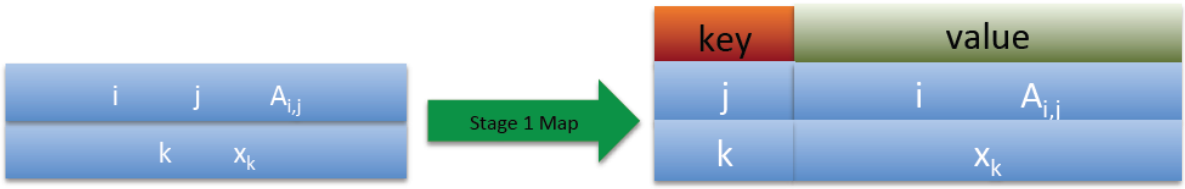



Figure 4.2: GIM-V Stage 1 Map

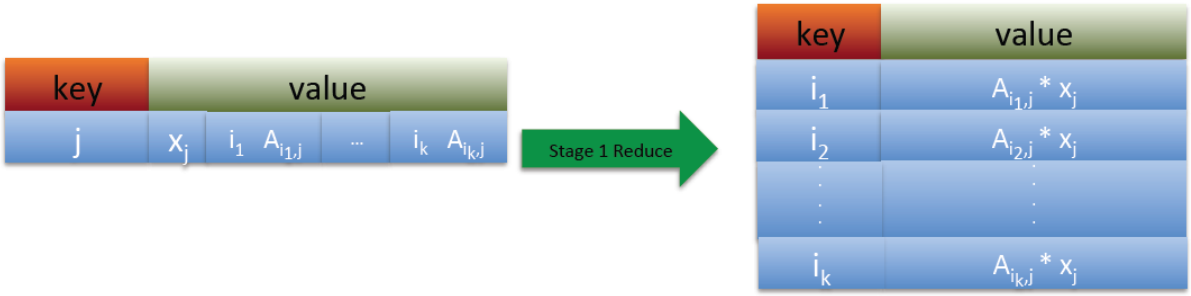



Figure 4.3: GIM-V Stage 1 Reduce

**Algorithm 2** GIM-V Stage 1

**Function Stage-1-Map(LongWritable Key, Text Value)**
*Value is either a matrix element [row(i) column(j) value($A_{i,j}$)]*
*or a vector element [row(i) value($x_i$)]*

```
1:
2: if Value is a matrix element then
3:     output (column, row value)
4: else if Value is a vector element then
5:     output(row, value)
6: end if
```

**Function Stage-1-Reduce(LongWritable Key, Text Values[])**
*For Key = j, Values contains a sequence of matrix elements [row(i) value($A_{i,j}$)]*
*and vector element [$x_j$]*

```
7:
8: v ← 0
9: matrixColumnElements ← []
10: for each element e in Values
11:     if e is a vector element then
12:         v ← x_j
13:     else
14:         matrixColumnElements[end] ← A_{i,j}
15:     end if
16: end for
17: if v is not equal to 0 then
18:     for each matrix element m in matrixColumnElements
19:         q ← A_{i,j} * v
20:         output(row(i), q)
21:     end for
22: end if
```

The result of Stage 1 is stored in HDFS and one can choose the suitable OutputFormat of the output data generated from Reduce phase in Stage 1 to make sure the it can be read as (key, value) pairs from HDFS in the next MapReduce stage (Stage 2). In Stage 2, the Stage-2-Map function (Figure 4.4) performs identity mapping (assigning the row as the key) and the Stage-2 Reduce function (Figure 4.5) adds values on the same row together

and then outputs the final result.


Figure 4.4: GIM-V Stage 2 Map

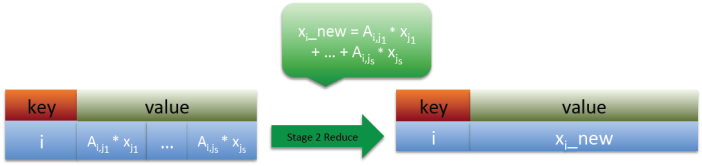



Figure 4.5: GIM-V Stage 2 Reduce

**Algorithm 3** GIM-V Stage-2

**Function Stage-2-Map(LongWritable Key, Text Value)**
*For Key = i, Value is a component* $[A_{i,j} * x_j]$
1: $output(row(i), A_{i,j} * x_j)$

**Function Stage-2-Reduce(LongWritable Key, DoubleWritable Values[])**
*For Key = i, Values contains a sequence of components* $A_{i,j} * x_j$
2:
3: $vNew \leftarrow 0$
4: **for each** *element* **e in** $Values$
5: $vNew \leftarrow vNew + e(value)$
6: **end for**
7: $output(Key, vNew)$

### 4.1.2 Vector scalar product

The product of two different vectors can be considered as a type of Matrix-Vector Multiplication and can be computed using the same algorithm as in the section 4.1.1. The scalar product of a vector with itself (the square of the vector) can be finished in one MapReduce

stage. In this case, for each input line "$i \quad x_i$", the Map function (Figure 4.6) outputs $(-1, x_i^2)$ with "$-1$" as the key. Then all (key, value) pairs from the Map phase share the same key and can be summed up to obtain the square of the vector during the Reduce phase (Figure 4.7). Note that to make the implementation effective one must use combiner in the MapReduce. Otherwise the single Reducer has to do all the additions as only one key (-1) exists in the Reduce phase.


Figure 4.6: Square of vector Map

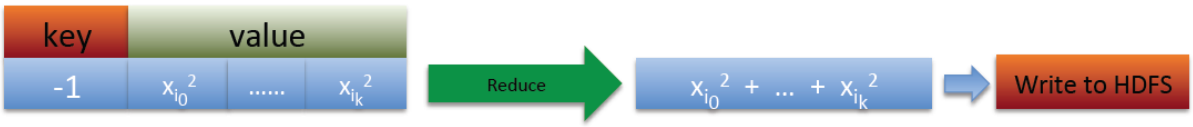



Figure 4.7: Square of vector Reduce

The pseudocode for square of a vector is given as follows:

**Algorithm 4** Square of a vector

**Function Map(LongWritable Key, Text Value)**
*Value is a line that represents a vector element: [row(i)* $x_i$*]*
1: *output*$(-1, x_i^2)$

**Function Reduce(LongWritable Key, DoubleWritable Values[])**
*All Keys are -1; Values contains a sequence of the squares of all non-zero vector elements* $x_j^2$

2:
3: $vNew \leftarrow 0$
4: **for each** *element* **e in** $Values$
5: $vNew \leftarrow vNew + e(value)$
6: **end for**
7: Write $vNew$ to HDFS

### 4.1.3 Vector addition

We now discuss how to compute $\mathbf{a} + \gamma\mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are sparse vectors, and $\gamma$ is a scalar. Two MapReduce stages are required in the implementation. It is difficult to do it in 1 stage because $\mathbf{a}$ and $\mathbf{b}$ are read from separate files in the same directory in HDFS, and cannot be distinguished when you add them. The first stage only has a Map phase (Figure 4.8), which computes $\gamma\mathbf{b}$. The second stage (Figure 4.9 and Figure 4.10) adds the two vectors $\mathbf{a}$ and $\gamma\mathbf{b}$ together. In Figure 4.10, there are three types of value since some elements of sparse vectors $\mathbf{a}$ and $\mathbf{b}$ are zero and not stored in the input files.


Figure 4.8: Vector-Scalar Multiplication Map

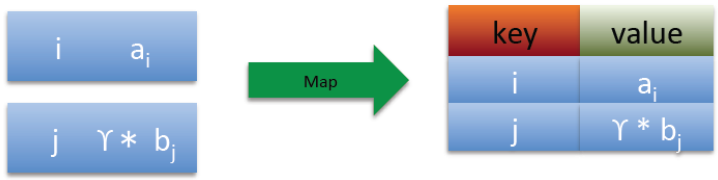



Figure 4.9: Vector addition Map

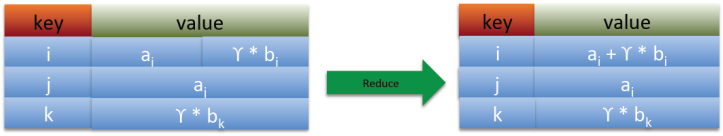



Figure 4.10: Vector addition Reduce

See Algorithm 5 and Algorithm 6 for Vector-Scalar Multiplication and Vector addition, respectively.

**Algorithm 5** Vector-Scalar Multiplication

**Function Map(LongWritable Key, Text Value)**
*Value is a line that represents a vector element: [row(i) $x_i$]*
1:
2: $output(row(i), \gamma * x_i)$

**Algorithm 6** Vector addition

**Function Map(LongWritable Key, Text Value)**
*Value is a line that represents a vector element: [row(i) $a_i$] or [row(j) $b_j$]*
1:
2: $output(row(i), a_i)$ or $(row(j), b_j)$

**Function Reduce(LongWritable Key, DoubleWritable Values[])**
*For Key = i, Values contains the ith vector elements of* $\boldsymbol{a}$ *and* $\boldsymbol{b}$, *[$a_i$ $b_i$]*
3: $output(row(i), a_i + b_i)$

## 4.2 Pagerank in Hadoop

In this section, we discuss the implementation of the Pagerank algorithm in Hadoop MapReduce. The variables used in this section are shown in the following table:

| Variables | Description |
|---|---|
| $P_i$ | Page i |
| $R_i$ | Rank of Page i |
| $N$ | Number of total pages |
| $N_i$ | Number of outlinks of Page i |
| $\alpha$ | Discount (Damping factor) |

Table 4.1: Description of variables in algorithm 7, 8, 9, 10, 11 and 12

.

### 4.2.1 Pagerank without dangling nodes

At first, we consider a network without dangling nodes. Let the links of the web graph (a directed graph) be stored in a text file where every line represents a link between two pages. The input file is supposed to be in the following format:

$url_0$ $urln_0$
$url_1$ $urln_1$
$url_2$ $urln_2$
$url_3$ $urln_3$
.....
$url_k$ $urln_k$

As we can see, each link in the graph occupies one line in the input file. For example, the first line means page $url_0$ has an outgoing link to page $urln0$. Before we start computing the ranks iteratively, we need to collect a list of the outlinks of each page in a pre-Stage phase such that we can later calculate the rank contributions from each page (the $R_k^{(j)}/N_k$ in equation 3.3) more easily and then reduce the contributions to get the new rank for each page according to equation 3.3 iteratively. In the pre-Stage, each line in the input file is assigned to the pre-Stage Map function (Figure 4.11). The Map function transforms each

line into a (key, value) pair. After shuffling, every shuffled (key, value) pair is assigned to the pre-Stage Reduce function (Figure 4.12) and an initial rank is added to the value field.


Figure 4.11: Pagerank pre-Stage Map


Figure 4.12: Pagerank pre-Stage Reduce

Now we give the MapReduce algorithm for the pre-Stage of the Pagerank computing:

---

**Algorithm 7** Pagerank pre-Stage without dangling nodes

---

**Function Map(LongWritable $P_i$, Text Value)**
*Value contains the url of a page and one of its outlinks:*
*[$P_i$ $P_{ik}$]*
1: *output*($P_i, P_{ik}$)

**Function Reduce(Text Key, Text Values[])**
*For Key = $P_i$, Values contains a list of the outlinks of $P_i$:*
*[$P_{i0}$ $P_{i1}$ $P_{i2}$ ...]*
2:
3: *Outlinks* $\leftarrow$ *Ranki*(Initial Rank)
4: **for each** *element* **Value in** *Values*
5: *Outlinks* += **Value** // add **Value** to Outlinks String
6: **end for**
7: *output($P_i$, Outlinks)*

---

The desired input file of the iterative stages has the following format:

$url_0$ $rank_0$ $url_0n_0$ $url_0n_1$ ... $url_0n_{s_0}$
$url_1$ $rank_1$ $url_1n_0$ $url_1n_1$ ... $url_1n_{s_1}$
.....
$url_k$ $rank_k$ $url_kn_0$ $url_kn_1$ ... $url_kn_{s_k}$

This is the format produced by the pre-Stage.

Each line contains the following arguments: a page's url, its rank, and a list of its outlinks. For example, the $url_0$ in the first line has rank of $rank_0$ (a String which can be transformed to Double or Float during the calculation) and outlinks: $url_0n_1$ ... $url_0ns_0$. As we mentioned before, one can choose the suitable OutputFormat for the output file from the pre-Stage to make sure it can be read as (key, value) pairs from HDFS as input file of the next MapReduce stage. When we read this file in the Map process of an iteration stage, the name of the page will be read as key and the rest of the line, the rank and outlinks together, will be read as a value of type String. Given (key, value) pair ($url_i$, $rank_i$ $url_in_0$ $url_in_1$ ... $url_in_{s_i}$), the Map function (Figure 4.13) divides ranki by $s_i$ (number of outlinks) and outputs ($url_in_j$, $rank_i/s_i$) for each $j$. The function also outputs ($url_i$, $m$ $url_in_0$ $url_in_1$ ... $url_in_{s_i}$) with the list of $url_i$'s outlinks as the value and $url_i$ as the key ($m$ is used as a flag to indicate that this (key, value) pair represents $url_i$'s outlinks). For each (key, value) pair after shuffling, say ($url_i$, value), the Reduce function (Figure 4.14) receives the outlinks of $urli$ and its rank contributions from other pages. It adds those rank contributions together and adjusts the sum using damping factor $\alpha$ to obtain a new rank for page $url_i$. Note that the algorithm "Pagerank without dangling nodes" can also be applied to a network with dangling nodes, but then the sum of the ranks decays, since dangling nodes are not properly taken into account.

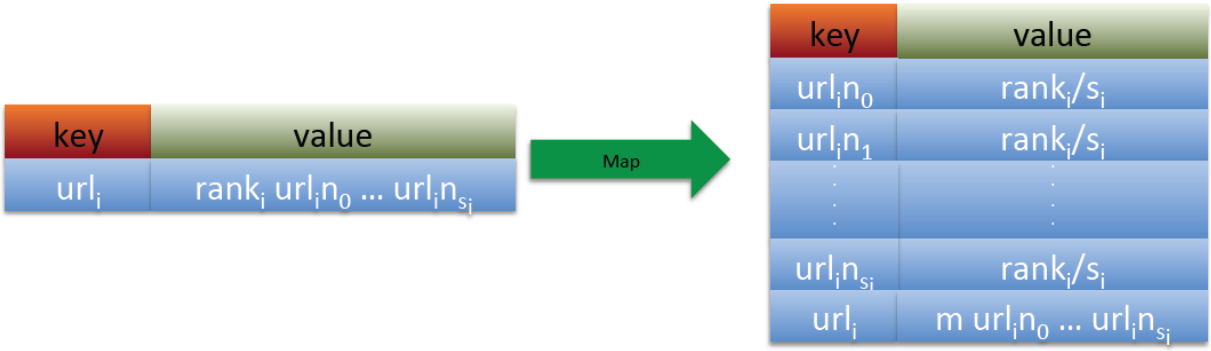



Figure 4.13: Pagerank Map

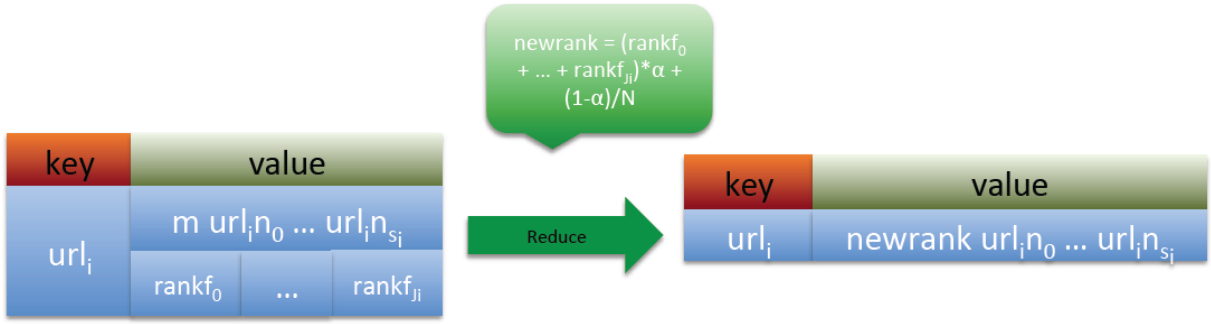



Figure 4.14: Pagerank Reduce

The MapReduce algorithm for each iteration of the Pagerank algorithm is given as follows:

**Algorithm 8** Pagerank without dangling nodes

**Function Map(Text $P_i$, Text Value)**
*Value contains the rank of page $P_i$ and its outlinks:*
*[$R_i$ $P_{i0}$ $P_{i1}$ $P_{i2}$ ...]*
1: $N_i$ = *Number of outlinks*
2: **for each** *outlink* $P_k$ **in** $Value$
3: *output*$(P_k, R_i/N_i)$
4: **end for**
5: *output($P_i$, m $P_{i0}$ $P_{i1}$ $P_{i2}$.....)(m means the value is a list of outlinks)*

**Function Reduce(Text Key, Text Values[])**
*For Key = $P_k$, Values contains list of outlinks of $P_k$ and Ranks of $P_k$ from other pages →[(m $P_{i0}$ $P_{i1}$ $P_{i2}$ ...) $R_0/N_0$ $R_1/N_1$ $R_2/N_2$ ...]*
6:
7: $R_k \leftarrow 0$
8: **for each** *element* **Value in** $Values$
9: **if** Value is the list of outlinks **then**
10: *Outlinks* ← *Value delete m* // according to equation 3.3
11: **else**
12: $R_k += R_i/N_i * \alpha$
13: **end if**
14: **end for**
15: $R_k$ += (1-$\alpha$)/$N$ // according to equation 3.3
16: *output($P_k$,$R_k$ Outlinks)*

### 4.2.2 Pagerank with dangling nodes

In this section, we consider the network with dangling nodes. With normalization, equation 3.4 becomes:

$$R_i^{(j+1)} = \alpha \left( \sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)} + \sum_{\text{dangling pages k}} \frac{1}{N} R_k^{(j)} \right) + \frac{1-\alpha}{N} \tag{4.1}$$

Here we use $\bar{r}_i$ to represent total rank contributions from all pages with outlink to Page $i$, namely, $\sum_{\text{inlinks k of i}} \frac{1}{N_k} R_k^{(j)}$ in equation 4.1. And we use $\hat{r}$ to represent total rank contributions of all dangling pages, namely, $\sum_{\text{dangling pages k}} R_k^{(j)}$. Assume the input file has

the same format as in the previous section except that some pages may have no outlinks, which means they do not appear in the first column of the input text file. Before we start a Pagerank iteration, we need to collect each page's outlinks in a pre-Stage as well. Moreover, we should collect the dangling pages from the second column of the input file. For each line "$url_i \quad urln_i$", the Map function (Figure 4.15) in the pre-Stage now not only outputs the (key, value) pair ($url_i$,$urln_i$) but also outputs ($urln_i$, "") with the empty String as the key to collect dangling pages. The Reduce function (Figure 4.16) in the pre-Stage collects every page's outlinks and outputs a (key, value) pair as we do in the previous section. For each dangling page, the value in the (key, value) pair only contains the initial rank.

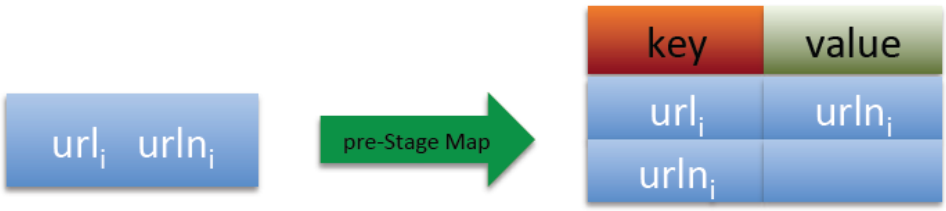



Figure 4.15: Pagerank with dangling nodes pre-Stage Map

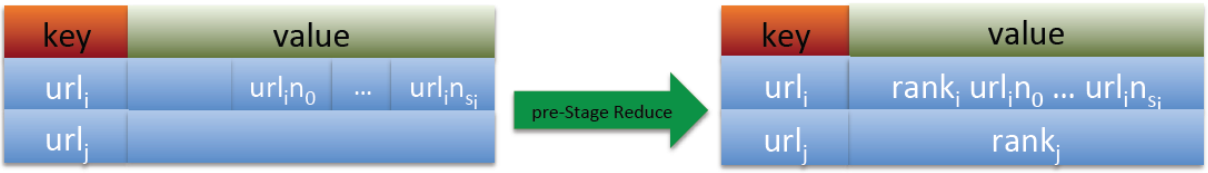



Figure 4.16: Pagerank with dangling nodes pre-Stage Reduce

The pseudocode of the pre-Stage is given as follows:

**Algorithm 9** Pagerank pre-Stage with dangling nodes

**Function Map(LongWritable $P_i$, Text Value)**
*Value contains the url of a page and one of its outlinks:*
*[$P_i$ $P_{ik}$]*
1: *output*($P_i, P_{ik}$)
2: *output*($P_{ik}$, “”)

**Function Reduce(Text Key, Text Values[])**
*For Key = $P_i$, Values contains list of outlinks of $P_i$:*
*[$P_{i0}$ $P_{i1}$ $P_{i2}$ ...]*
3:
4: *Outlinks* ← *Ranki*(Initial Rank)
5: **for each** *element* **Value in** *Values*
6: *Outlinks* += **Value** // add **Value** to Outlinks String
7: **end for**
8: *output($P_i$, Outlinks)*

The input file of the iterative stages has the same format as in the previous section. Each Pagerank iteration now requires two MapReduce stages. Some lines may only have one page and its rank, which means those lines represent dangling pages. The MapReduce code for pages with outlinks is almost the same as in the previous section. For dangling pages, the Stage 1 Map function (Figure 4.17) outputs a (key, value) pair with “-1” as the key and the rank as the value to collect all rank contributions from dangling nodes together in the Reduce phase. It also outputs a (key, value) pair that represents the page's outlink information. The Stage 1 Reduce function (Figure 4.18) adds all ranks with key of “-1” together, multiplies the sum by $\alpha/N$, and then writes it to HDFS or the local file system as the result will be used in the next stage.

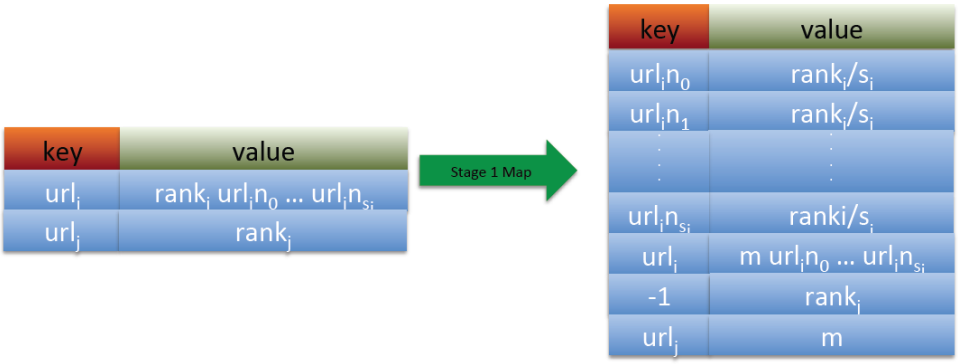



Figure 4.17: Pagerank with dangling nodes Stage 1 Map

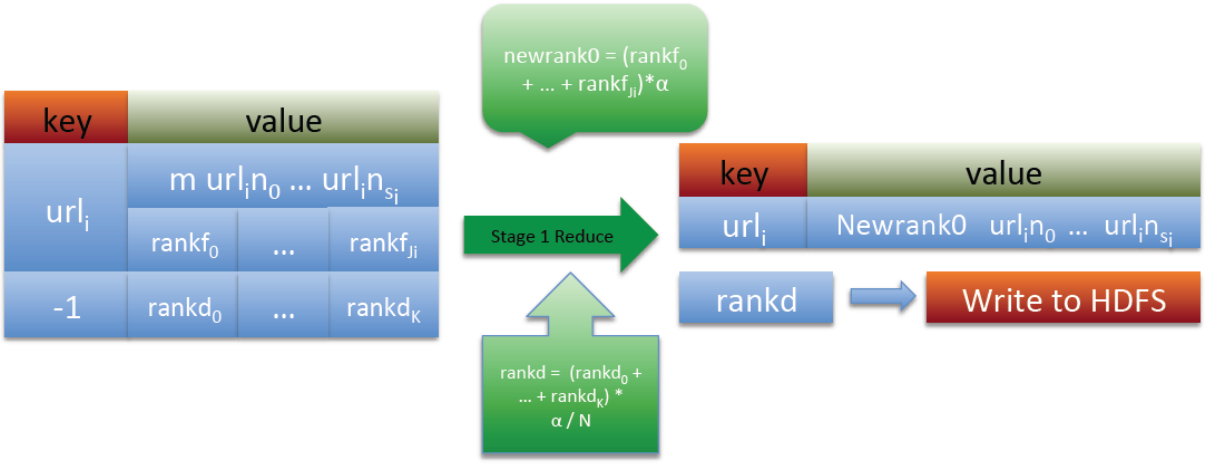



Figure 4.18: Pagerank with dangling nodes Stage 1 Reduce

Below is the algorithm for Stage 1 in each iteration of Pagerank with dangling nodes:

**Algorithm 10** Pagerank Stage 1 with dangling nodes

**Function Stage-1-Map(Text $P_i$, Text Value)**
*Value contains the rank of page $P_i$ and its outlinks:*
*[$R_i$ $P_{i_0}$ $P_{i_1}$ $P_{i_2}$ ...]*
**if** *page $P_i$ has outlinks* **then**
  $N_i$ = *Number of outlinks*
  **for each** *outlink $P_k$* **in** $Value$
    $output(P_k, R_i/N_i)$
  **end for**
  *output($P_i$, m $P_{i0}$ $P_{i1}$ $P_{i2}$.....)(m indicates that the value is the list of outlinks)*
**else if** page $P_i$ doesn't have outlinks **then**
  $output(-1, R_i)$
  $output(P_i, m)$
**end if**

**Function Stage-1-Reduce(Text Key, Text Values[])**
*For Key = -1, Values contains Rank contributions of pages without outlinks*
*→ [$R_{n_0}$ $R_{n_1}$ $R_{n_2}$ ...]*
*For Key = $P_k$, Values contains list of outlinks of $P_k$ and rank contributions to $P_k$ from other pages →[[m $P_{i0}$ $P_{i1}$ $P_{i2}$ ...] $R_0/N_0$ $R_1/N_1$ $R_2/N_2$ ...]*

**if** *Key = -1* **then**
  $\hat{r} \leftarrow 0$
  **for each** *element $R_{n_i}$* **in** $Values$
    $\hat{r} += R_{n_i}$
  **end for**
  $\hat{r} = \alpha * \hat{r}/N$ // *N is the number of total pages*
  *Write $\hat{r}$ into a HDFS file*
**else**
  $\bar{r}_k \leftarrow 0$
  **for each** *element* **Value in** $Values$
    **if** Value is the list of outlinks **then**
      *Outlinks ← Value delete m*
    **else**
      $\bar{r}_k += R_i/N_i$
    **end if**
  **end for**
  $\bar{r}_k = \alpha * \bar{r}_k$
  *output($P_k$, $\bar{r}_k$ Outlinks)*
**end if**

After Stage 1, we just need one Map phase to finish the iteration. The Stage 2 Map function (Figure 4.19) computes the new rank for each page by reading $\hat{r}$ from HDFS and adjusting the rank with damping factor $\alpha$.

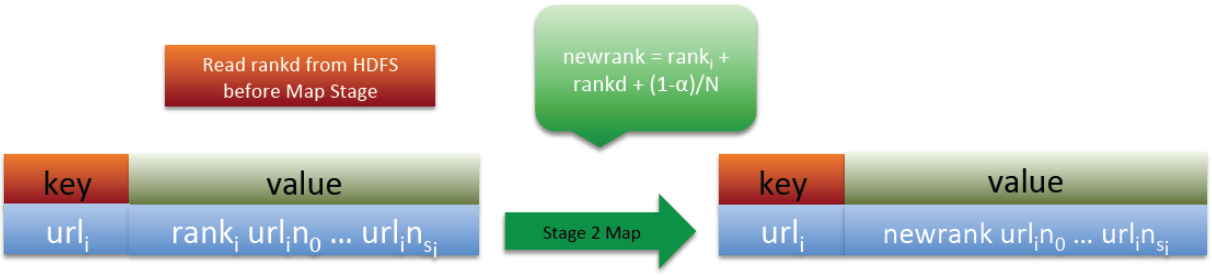



Figure 4.19: Pagerank with dangling nodes Stage 2 Map

The pseudocode of the Stage 2 Map is given as follows:

**Algorithm 11** Pagerank Stage 2 with dangling nodes

1: *Before we launch the Map process, we read $\hat{r}$ from the HDFS file created in the*
2: *the Reduce process in Stage 1*

**Function Stage-2-Map(Text Key,Text Value)**

*For Key = $P_k$, Value contains total rank contribution from all pages with outlink to $P_k$ and list of outlinks of $P_k \rightarrow [\bar{r}_k\ P_{i0}\ P_{i1}\ P_{i2}\ ...]$*

3: $R_k \leftarrow 0$
4: $R_k = \bar{r}_k + \hat{r} + (1\text{-}\alpha)/N$ // according to the equation 4.1
5: *output($P_k$, $R_k$ $P_{i0}$ $P_{i1}$ $P_{i2}$ ...)*

Here we will give a example to show how our algorithm works. Figure 4.20 is the directed graph of a small network.

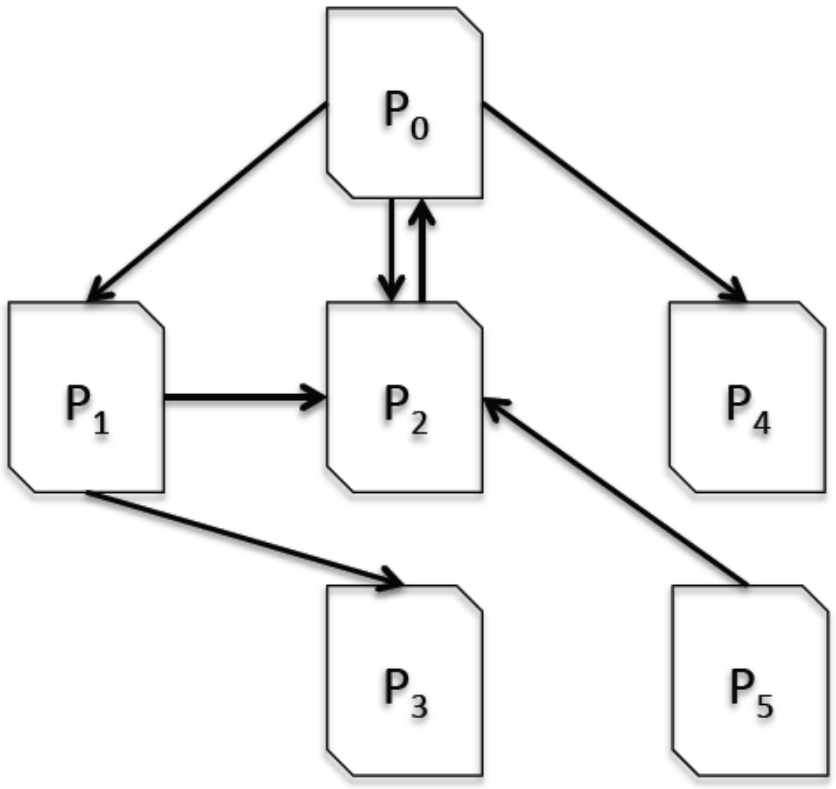



Figure 4.20: Directed graph of a small network

The input file (after pre-Stage) for the above network is as follows:

$P_0$ $R_0$ $P_1$ $P_2$ $P_4$
$P_1$ $R_1$ $P_2$ $P_3$
$P_2$ $R_2$ $P_0$
$P_3$ $R_3$
$P_4$ $R_4$
$P_5$ $R_5$ $P_2$

$P_3$ and $P_4$ are dangling nodes.

After the Map process in stage 1, the shuffled data will become:

{-1, [$R_3$ $R_4$]}
{$P_0$,[“m” $P_1$ $P_2$ $P_4$] $R_2/N_2$ }  // $P_0$ has an inlink from $P_2$
{$P_1$,[“m” $P_2$ $P_3$] $R_0/N_0$ }
{$P_2$,[“m” $P_0$] $R_0/N_0$ $R_1/N_1$}
{$P_3$,“m” $R_1/N_1$}
{$P_4$,“m” $R_0/N_0$}
{$P_5$,[“m” $P_2$] }

After the Reduce process in stage 1, we will write $\alpha*(R_3+R_4)/N$ into a HDFS file and have:

| | | | | |
|---|---|---|---|---|
| $P_0$ | $\alpha * R_2/N_2$ | $P_1$ | $P_2$ | $P_4$ |
| $P_1$ | $\alpha * R_0/N_0$ | $P_2$ | $P_3$ | |
| $P_2$ | $\alpha * (R_0/N_0 + R_1/N_1)$ | $P_0$ | | |
| $P_3$ | $\alpha * R_1/N_1$ | | | |
| $P_4$ | $\alpha * R_0/N_0$ | | | |
| $P_5$ | 0.0 $P_2$ // $P_5$ does not have an inlink | | | |

In the stage 2 Map we output the result after one iteration:

| | | | | |
|---|---|---|---|---|
| $P_0$ | $\alpha * R_2/N_2 + \alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | $P_1$ | $P_2$ | $P_4$ |
| $P_1$ | $\alpha * R_0/N_0 + \alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | $P_2$ | $P_3$ | |
| $P_2$ | $\alpha * (R_0/N_0 + R_1/N_1) + \alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | $P_0$ | | |
| $P_3$ | $\alpha * R_1/N_1 + \alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | | | |
| $P_4$ | $\alpha * R_0/N_0 + \alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | | | |
| $P_5$ | $\alpha * (R_3 + R_4)/N + (1 - \alpha)/N$ | $P_2$ | | |

As we can see, it seems we need to use two MapReduce stages in each iteration for Pagerank with dangling nodes. But in fact, the Stage 2 Map process can be done in the next iteration's Stage 1 Map process. Then each iteration can be done with only one MapReduce

stage, which reduces the overhead for starting a new stage and saves time for reading and writing data in HDFS. The Stage 1 Map in algorithm 10 can be modified as follows:

**Algorithm 12** Modified Pagerank Stage 1 with dangling nodes

1: *Before we launch the Map process, we read $\hat{r}$ from the HDFS file created in the*
2: *the Reduce process in Stage 1*

**Function Stage-1-Map(Text $P_i$,Text Value)**

*Value contains the rank (not fully updated) of page $P_i$ and its outlinks:*
*$[R_i\ P_{i_0}\ P_{i_1}\ P_{i_2}\ ...]$*

3:
4: **if** *page $P_i$ has outlinks* **then**
5: **for each** *outlink $P_k$* **in** $Value$
6: $N_i$ = *Number of outlinks*
7: $output(P_k, (R_i + \hat{r} + (1-\alpha)/N)/N_i)$
8: **end for**
9: *output($P_i$, "m" $P_{i0}$ $P_{i1}$ $P_{i2}$.....)("m" means the value is the list of outlinks)*
10: **else if** page $P_i$ doesn't have outlinks **then**
11: $output(-1, R_i + \hat{r} + (1-\alpha)/N)$
12: $output(P_i, \text{"}m\text{"})$
13: **end if**

In the last iteration, one should use one more MapReduce Stage, namely, Algorithm 11 to compute the final ranks as there is no more iteration and therefore ranks can not be computed using Algorithm 12. Assume we want to compute ranks using $n$ iterations, then one pre-Stage MapReduce (Algorithm 9), one MapReduce stage for Algorithm 10 (for the first iteration), $n-1$ MapReduce stages for Algorithm 12 (for the last $n-1$ iterations), and one final MapReduce stage for Algorithm 11 to compute the final ranks. In total, we should have $n+1$ MapReduce stages if we compute ranks using $n$ iterations.

## 4.3 CG in Hadoop

The following operations are used in CG:
(1) Matrix-vector multiplication: given a matrix $\mathbf{A}$ and a vector $\mathbf{x}$, calculate $\mathbf{Ax}$ (line 1, 5, and 7 in Algorithm 1);
(2) Square of a vector: given a vector $\mathbf{v}$, calculate $\mathbf{v^T v}$ (line 5 and 9 in Algorithm 1);

(3) Scalar product of vectors: given two vectors $\mathbf{v_0}$ and $\mathbf{v_1}$, calculate $\mathbf{v_0^T v_1}$ (line 5 in Algorithm 1);
(4) Vector addition: given two vectors $\mathbf{v_0}$ and $\mathbf{v_1}$, and a scalar $\alpha$, calculate $\mathbf{v_0} + \alpha\mathbf{v_1}$ (line 1, 6, 7 and 10 in Algorithm 1).

The implementations of these operations have been discussed in Section 4.1. We use four functions to launch Hadoop MapReduce for each operation: MV(Path of $\mathbf{A}$, Path of $\mathbf{x}$), VSquare(Path of $\mathbf{v}$), VVM(Path of $\mathbf{v_0}$, Path of $\mathbf{v_1}$), VA(Path of $\mathbf{v_0}$, Path of $\mathbf{v_1}$, $\alpha$), respectively. For operations (1),(3) and (4), we need to launch two MapReduce stages while only one stage is needed for operation (2).
The file format of the matrix and vector is the same as that in Section 4.1. The matrix, vectors and parameters used or created in the $k$th iteration are stored in HDFS with the following directory structure:

CG/A : the given matrix $\mathbf{A}$
CG/b : the given vector $\mathbf{b}$
CG/x/k : the kth solution $\mathbf{x_k}$
CG/x/k+1 : the k+1th solution $\mathbf{x_{k+1}}$
CG/r/k : the kth residual $\mathbf{r_k}$
CG/r/k+1 : the k+1th residual $\mathbf{r_{k+1}}$
CG/p/k : $\mathbf{p_k}$
CG/p/k+1 : $\mathbf{p_{k+1}}$
CG/Apk/k : $\mathbf{Ap_k}$

The pseudo code for our CG implementation is given as follows:

**Algorithm 13** CG-Hadoop algorithm

1: calculate $\mathbf{Ax_0}$ by MV(CG/A,CG/x/0)
2: calculate $\mathbf{r_0}$ by VA(CG/b,Path of $\mathbf{Ax_0}$,-1)
3: $\mathbf{p_0} \leftarrow \mathbf{r_0}$
4: $k \leftarrow 0$
5: calculate $\mathbf{r_0^T r_0}$ by VSquare(CG/r/0)
6: repeat
7: (if k = 0, we use CG/r/0 to substitute CG/p/0)
8: calculate $\mathbf{Ap_k}$ by MV(CG/A,CG/p/k)
9: calculate $\mathbf{p_k^T Ap_k}$ by VV(CG/p/k,CG/Apk/k)
10: $\alpha_k \leftarrow \frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T Ap_k}}$
11: calculate $\mathbf{x_{k+1}}$ by VA(CG/x/k,CG/p/k,$\alpha_k$)
12: calculate $\mathbf{r_{k+1}}$ by VA(CG/r/k,CG/Apk/k,$-\alpha_k$)
13: calculate $\mathbf{r_{k+1}^T r_{k+1}}$ by VSquare(CG/r/k+1)
14: if $\mathbf{r_{k+1}^T r_{k+1}}$ is sufficiently small then exit loop
15: $\beta_k \leftarrow \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
16: calculate $\mathbf{p_{k+1}}$ by VA(CG/r/k+1,CG/p/k,$\beta_k$)
17: $k \leftarrow k + 1$
18: end repeat
19: The result is stored in CG/x/k+1

Note that the MapReduce processes in line 11 and 12 can be done in parallel since they are independent from each other.

## 4.4 Performance tests in Hadoop

### 4.4.1 Single Node test

In this section, we first do a test for WordCount on a linux machine with one processor (Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz) and four cores available. We use the data from http://www.memetracker.org/data.html as the input file (4116073668B, namely 4.2 GB). In this test, the block size is changed to 257,254,912 to make sure we have 16 Mappers (Map tasks) in total. At first, we do tests by changing the parameter mapred.tasktracker.map.tasks.maximum from 2 to 16 and then compare the performance.

This parameter controls the maximum number of Mappers running simultaneously on each node (concurrent Mappers). Here the total number of Reducers is one by default. The test results are shown in Table 4.2:

| No. of Mappers | No. of concurrent Mappers | Time(s) |
|---|---|---|
| 16 | 2 | 971.058 |
| 16 | 4 | 675.701 |
| 16 | 8 | 914.731 |
| 16 | 12 | 1,075.751 |
| 16 | 16 | 3,701.899 |

Table 4.2: The WordCount test on a single node with 16 Mappers and 1 Reducer

As we can see, increasing the number of concurrent Mappers makes the performance worse. So we should choose this parameter according to the cores available on the node. Each core should only run one Mapper at the same time. Now we turn to how to choose the total number of Reducers. We do the test by changing two parameters: mapred.tasktracker.reduce.tasks.maximum (maximum number of Reducers running simultaneously on each node) and mapred.reduce.tasks (total number of Reducers for the job). In this test, the number of concurrent Mapper is chosen to be 4 (the optimal one). Table 4.3 reports the test results:

| No. of concurrent Reducers \ No. of Reducers | 4 | 8 | 12 | 16 |
|---|---|---|---|---|
| 2 | 614.304 | 611.625 | 647.292 | 645.57 |
| 4 | 679.756 | 666.36 | 680.7 | 638.753 |
| 8 | N/A | 841.445 | 838.048 | 815.7 |
| 12 | N/A | N/A | 1523.884 | 1203.268 |
| 16 | N/A | N/A | N/A | 2732.139 |

Table 4.3: The WordCount test on a single node with 16 Mappers and different number of Reducers

The total number of Reducers does not have much influence on the performance of WordCount on a single node as there in no communication between different nodes. However, the number of co-current Reducers should not be larger than the number of cores available on the node otherwise the performance can be greatly harmed.

### 4.4.2 Tests on gridbase cluster

At first, we discuss how Mappers and Reducers are assigned to each node, which is determined by four parameters: dfs.block.size, mapred.reduce.tasks, and mapred.tasktracker.map (reduce).tasks.maximum. The first two parameters determine the total number of Mappers and Reducers for a job, respectively, and the last two define the maximum number of concurrent Mappers (Reducers) on each node. We first illustrate how the Mappers are scheduled. The scheduling for Reducers is similar. Assume we have $n$ slave nodes in the cluster, $M$ Mappers in total, and at most $N$ Mappers are allowed to run simultaneously on a node. If $n * N \geq M$, which means each node runs less than $N$ Mappers on average, all Mappers are distributed across the slave nodes equally, in the sense that each node runs $[M/n]$ or $[M/n] + 1$ Mappers. For the case $n * N < M$, $N$ Mappers are assigned to each node once the MapReduce job is launched. Once a node finishes some Map tasks, new Mappers (from the remaining unlaunched $M - n * N$ Mappers) are assigned to it to fill up the Map slots in the node. This process continues until all Mappers are launched or finished. The map phase ends when all Mappers are finished. Figure 4.21 illustrates the scheduling process for $M = 20$, $N = 4$ and $n = 3$. A gray box represents a slave node and a blue box represents a Mapper.

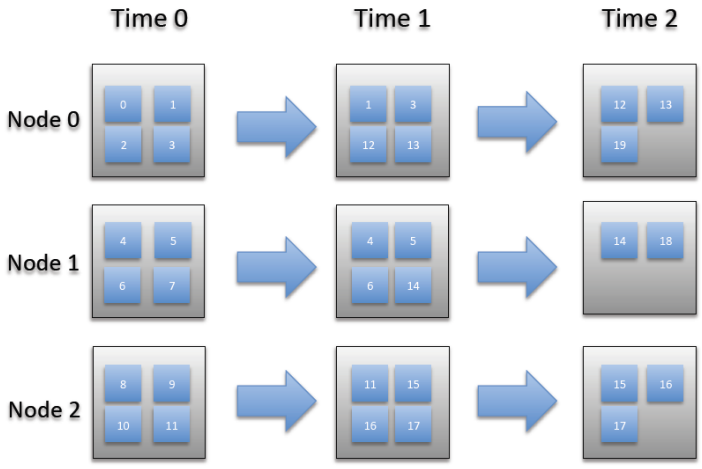



Figure 4.21: Scheduling Mappers in a Hadoop cluster

Our test in this section is done on the gridbase cluster with 28 cores in total. The cluster has three machines: gridbase1, gridbase2 and gridbase3. All machines are used as slave nodes and gridbase1 also acts as master node. Table 4.4 reports the properties of each

machine:

| gridbase | No. of cores | No. of processors | Name of processor |
|---|---|---|---|
| 1 | 4 | 2 | Dual-Core AMD Opteron(tm) 2218@2.6Hz |
| 2 | 8 | 2 | Intel(R) Xeon(R) CPU X5460 @ 3.16GHz |
| 3 | 16 | 4 | Intel(R) Xeon(R) CPU X5460 @ 3.16GHz |

Table 4.4: The CPU information of the gridbase machines

Table 4.5 reports the results of WordCount tests in Hadoop. As we can see, increasing the No. of concurrent Mappers per node makes the performance worse in most cases. The reason is because the gridbase cluster is heterogeneous: each slave node has a different number of cores. The node gridbase 3 may perform well for 16 concurrent Mappers while gridbase 1 gets stuck with 16 Mappers running simultaneously. Although gridbase 2 and gridbase 3 are capable of running more Mappers at the same time, we suggest that the number of concurrent Mappers should be 4 as running more Mappers on gridbase 1 may strongly harm the performance. For example, given 48 Mappers in total and 16 concurrent Mappers per node, each node gets 16 Mappers but gridbase 3 can finish its 16 Mappers quickly while gridbase 1 needs more time to complete, which is not parallel and is wasteful. Table 4.2 shows that running more Mappers (than the number of cores available) in a node could strongly harm the performance. Using 4 concurrent Mappers in each node may be a little bit slower sometimes but it is much more stable. The number of concurrent Reducers should also be set as 4 for the same reason.

| No. of Mappers / No. of concurrent Mappers | 123 | 62 | 49 | 31 |
|---|---|---|---|---|
| 4 | 218.901 | 204.138 | 185.957 | 188.048 |
| 8 | 234.753 | 199.944 | 171.371 | 183.614 |
| 16 | 251.919 | 190.586 | 212.019 | 206.95 |

Table 4.5: The WordCount test on the gridbase cluster with 12 Reducers in total and 4 concurrent Reducers on each node. (Time in seconds.)

The test results for Pagerank and CG are presented and discussed in Section 7.4.

# Chapter 5

# RDD transformations in Spark

We now present some basic and essential RDD transformations in detail and illustrate how they work in Spark since many RDD transformations that may involve shuffling are based on them. We also investigate how co-partitioning and co-location of RDDs influence their performance in Spark.

## 5.1 combineByKey

CombineByKey combines the elements for each key using a a custom set of aggregation functions. It turns an RDD[(K, V)] into a result of type RDD[(K, C)]. It is a transformation for RDDs of (key, value) pairs and many other (key, value) pair RDD transformations like reduceByKey are based on combineByKey. This RDD transformation combines the values of each key into a new type of element. Given an RDD of type RDD[(K, V)], it transforms it into an RDD of type RDD[(K, C)] where C is called a combined type. CombineByKey is mainly determined by five arguments: createCombiner, mergeValue, mergeCombiners, partitioner, and mapSideCombine. CreateCombiner ((V) => C) is a function used for creating a combiner (C) if no combiner has been created for a key. Here a combiner refers to the result of a combining operation. MergeValue ((C, V) => C) is used to merge a value (V) into a combiner (C). MapSideCombine (Boolean = true by default) decides whether or not to perform combining during the map-side (perform combining locally), which means a key may initially have multiple combiners. If one chooses to use map-side combining, then mergeCombiners must be defined. MergeCombiners ((C, C) => C) merges two combiners into a single one. The partitioner defines the partitioner of the resulting RDD.

Shuffling between different nodes may occur when combineByKey is applied to an RDD. This depends on the partitioners of this RDD and the output RDD (the child RDD). If an RDD has the same partitioner as its output RDD, then there is no shuffling process between nodes during this RDD transformation. However, shuffling is inevitable in a Spark cluster with several Worker nodes if the RDDs have different partitioners.

A Hash table is created for each partition of the input RDD to merge values into a combiner for each key. After the Hash table is created, all elements (key-value Tuples) of the corresponding partition are "put" into the Hash table one by one. Here "put" means: if the combiner for the key in this (key ,value) pair has not been created yet in the Hash table, then create one by applying the createCombiner function to this (key, value) pair element; if the combiner for this key has already been created, then combine the value of this element into the combiner for this key using function mergeValue. Once all elements in the partition are put in the Hash table, the computation for the partition of the output RDD is finished, which means all elements of the output RDD have been generated.

Now we give an example to show how combineByKey works and how elements in a partition are put into the Hash table. Assume we want to combine values of type String into a sequence. Let V = String and C = Seq[String] (Seq[T] represents a sequence of elements of type T). The createCombiner, mergeValue, mergeCombiners are defined as: createCombiner transforms a String value into a sequence containing only this value; mergeValue adds a value to the end of a given sequence and then returns the sequence; mergeCombiners adds all elements of a sequence to another sequence and returns the new sequence containing all elements of the two sequences. Suppose a partition has three elements: (1, "hadoop"), (2, "spark") and (1, "java"). Figure 5.1 illustrates how elements of this partition are put in the Hash table using createCombiner and mergeValue. The process of putting combiners in the Hash table after shuffling with map-side combining is similar.

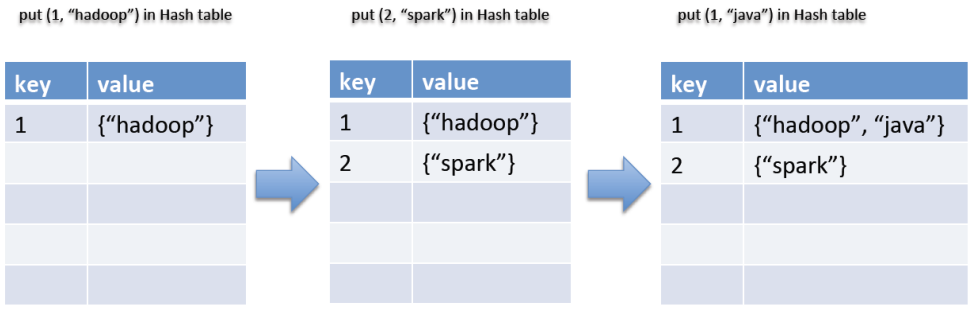



Figure 5.1: Putting elements of a partition in the Hash table during a combineByKey combining operation

Figure 5.2, 5.3 and 5.4 show how combineByKey works. The blue, red and green boxes represent RDD partitions. The grey box stands for a Worker node. Figure 5.2 illustrates the case when the input RDD and output RDD have the same partitioner. The input RDD is stored in three Worker nodes with partition 0 and 1 in node 0, partition 2 and 3 in node 1 and partition 4 and 5 in node 2.

Figure 5.3 shows how combineByKey works when the output RDD's partitioner is supposed to be changed after this transformation. Here, the argument mapSideCombine is set to be true, which means a combining process called map-side combining is applied to each partition before shuffling. The scheme of this combining is the same as in Figure 5.2. Each element of the resulting RDD after this combining has the type of (K, C) with the value (of type C) as the combiner produced by the map-side combining. However, each key may have multiple combiners belonging to different partitions which may be located on different nodes. Since the output RDD has a new partitioner, new partitions are created and each element needs to be shuffled to the desired partition. After the shuffling, another combining process is necessary as a key may have multiple combiners. Those combiners must be merged using the function mergeCombiners. A Hash table is created for each partition of the shuffled RDD to merge combiners for all keys in the partition. This process is similar to map-side combining.

One can also choose not to use map-side combining by setting the argument mapSideCombine to be false. Figure 5.4 illustrates how combineByKey works in this case. At first, the input RDD needs to be shuffled to make sure all keys go to their desired partitions. After shuffling, the combining process is the same as shown in Figure 5.2.

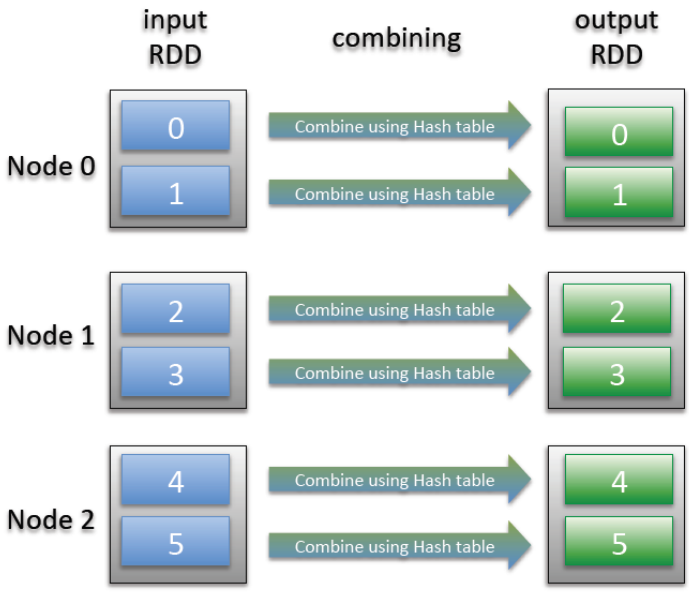



Figure 5.2: combineByKey without shuffling, for the case where the input RDD has the same partitioner as the output RDD.

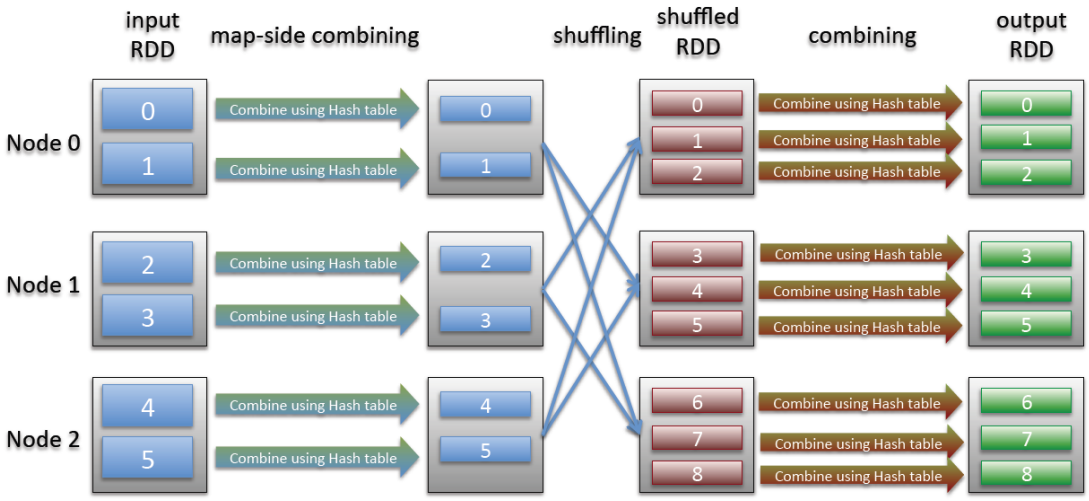



Figure 5.3: combineByKey using map-side combine before shuffling, for the case where the input RDD has no partitioner or a partitioner different from the output RDD.

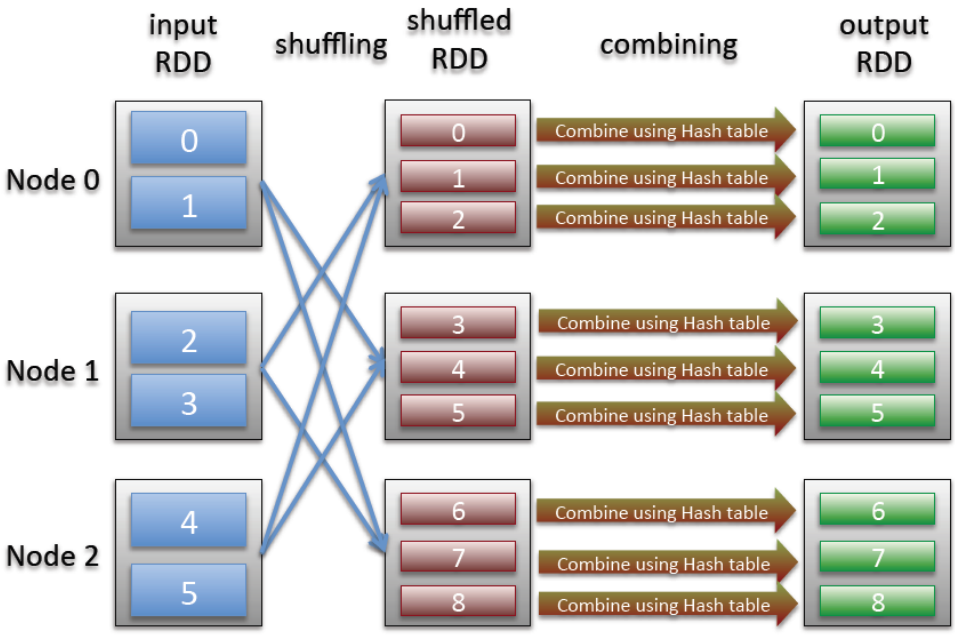



Figure 5.4: combineByKey without map-side combine before shuffling, for the case where the input RDD has no partitioner or a partitioner different from the output RDD.

Note that Figure 5.3 is similar to Hadoop MapReduce with a combiner function (to do partial reduces), and Figure 5.4 is similar to Hadoop MapRedue without a combiner function.

Finally, we discuss how shuffling works in Spark. Shuffling occurs (for example, in combineByKey) because the output RDD has a different partitioner. Computing each partition of the output RDD requires fetching some elements from potentially all partitions of the input RDD, which means each partition of the output RDD may depend on all partitions of the input RDD (this is called wide dependency in Spark terminology). For a partition of the output RDD, we call those elements needed from a partition of the input RDD a block. To compute a partition of the output RDD we need to fetch all the blocks. If a block and the partition are located on the same node, we call this block a local block; otherwise we call it a remote block. As we said before, Spark stages are divided by shuffling. An RDD transformation involving shuffling is the first transformation of its stage. Those blocks to be fetched are created in its previous stage. If the output RDD has N partitions, then each Worker node creates N blocks while computing the input RDD. The data (elements of the input RDD) can be written to those files when computing a partition of the input RDD

is finished. Figure 5.5 gives an example of shuffling. The input RDD is computed on two nodes and has four partitions. Each Worker node creates three files (blocks) according to the partitioner of the output RDD. After each partition of the input RDD is computed, its elements are written to the corresponding blocks. Writing data into blocks is also included in Stage 0 and must be finished before Stage 1 starts. In Stage 1, each partition of the output RDD is computed by fetching the corresponding blocks and then applying RDD transformations to the elements.

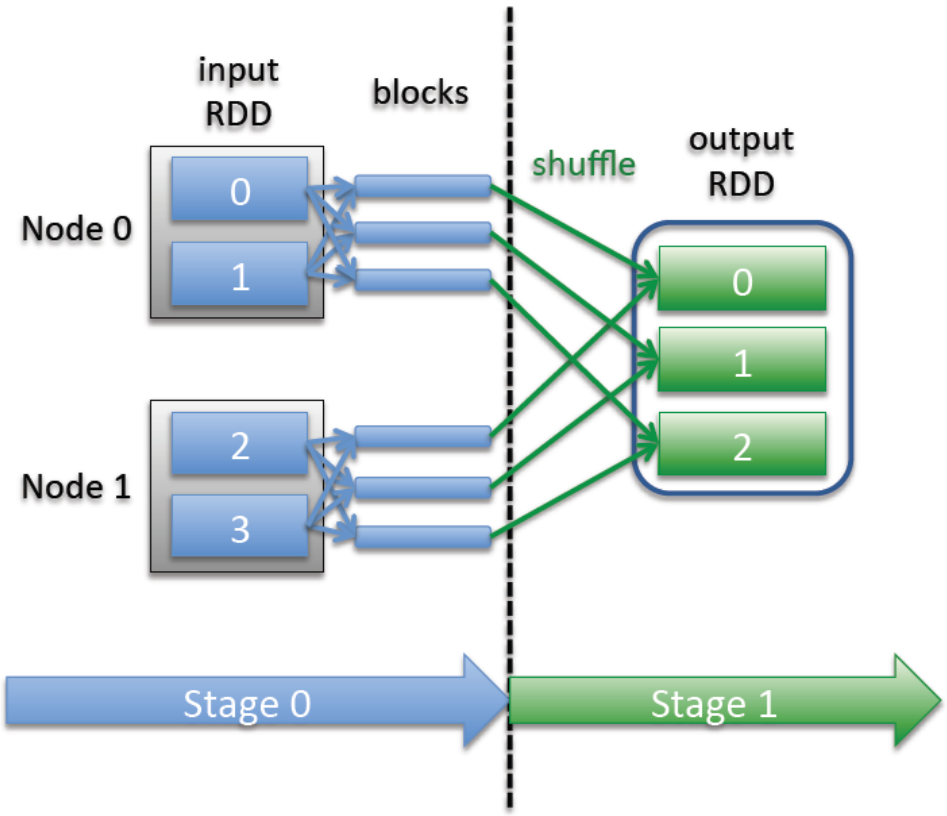



Figure 5.5: Shuffling in Spark

## 5.2 cogroup

The cogroup transformation groups an RDD of type RDD[(K,V)] with other (key, value) pair RDDs of type RDD[(K,W)] (W can be of different type than V). The cogroup can

group at most three (key, value) pair RDDs sharing the same key type in Spark. Each element of the resulting RDD has the type (K, (Seq[V], Seq[W])) (group two RDDs) or (K, (Seq[V], Seq[W1], Seq[W2])) (group three RDDs). For each key in the RDDs to be grouped together, the value of the resulting RDD is a Tuple2 (2-tuple) or Tuple3 (3-tuple) containing the list of values for this key in the input RDDs. Each element of the tuple is used to represent a sequence of values for this key from an input RDD. If no value exists for a key in an input RDD, then the corresponding element of the tuple is an empty sequence. In this section, we discuss how cogroup groups two RDDs; grouping three RDDs would be similar.

Assume the input RDD has the type RDD[(K, V)] and cogroup is applied to it to group it with another RDD of type RDD[(K, W)] (we name it "other RDD" in this section). The cogroup transformation may also involve shuffling between different Worker nodes and this is determined by several factors such as the partitioners of the RDDs involved in this transformation. Table 5.1 lists different cases which determine the shuffling in cogroup.

| Case | input RDD partitioner: same as output RDD | other RDD partitioner: same as output RDD |
|---|---|---|
| 0 | Yes | Yes |
| 1 | Yes | No |
| 2 | No | Yes |
| 3 | No | No |

Table 5.1: partitioner information of input, other and output RDDs in cogroup

We now illustrate how cogroup works for case 0, 1 and 3. Case 1 and case 2 are similar. The cogroup for each partition contains two steps:

- **Step 1** Fetch RDD partitions and blocks (for case 1, 2, and 3 only);
- **Step 2** Then put all the elements of fetched RDD partitions and blocks in a Hash table for this partition.

Here an RDD block in step 1 represents a portion of an RDD partition. An RDD partition has to be divided into several blocks if the partitioning of this RDD is different from the output RDD. For example, in case 1, the partitions of the other RDD have to be divided into blocks.

Case 0 does not involve shuffling between partitions as all RDDs have the same partitioner. Each partition of the output RDD is computed by several steps. Figure 5.6 gives an example

of cogroup for case 0 where a solid rectangle represents an RDD partition. We assume that partition i (i = 0, 1, 2) of the input RDD is located on the same node as partition i of the output RDD. At first, the corresponding partitions of input RDD and other RDD (the partitions that a partition of output RDD depends on) are fetched to the node where the partition of output RDD is located. For example, partition 0 from the input RDD and partition 0 from the other RDD are fetched to the same node. After fetching, a Hash table is created for each partition of output RDD. The Hash table has keys of type K and values of type Seq(Seq[V], Seq[W]). All elements of the corresponding partitions from the input RDD and the output RDD are put into the Hash table one by one. For example, all elements from partitions 0 of the input RDD and the output RDD are put into the top Hash table. When an element is put into the Hash table, the value of the corresponding (key, value) pair in the Hash table is updated (the value of the element is added to Seq[V] or Seq[W]) if the (key, value) pair already exists; otherwise, a (key, value) pair is created in the Hash table with value containing the value of this element. Putting elements in the Hash table is similar to the case of combineByKey as we discussed in section 5.1. After all elements are put in the Hash table, the partition of output RDD is created using the final Hash table.

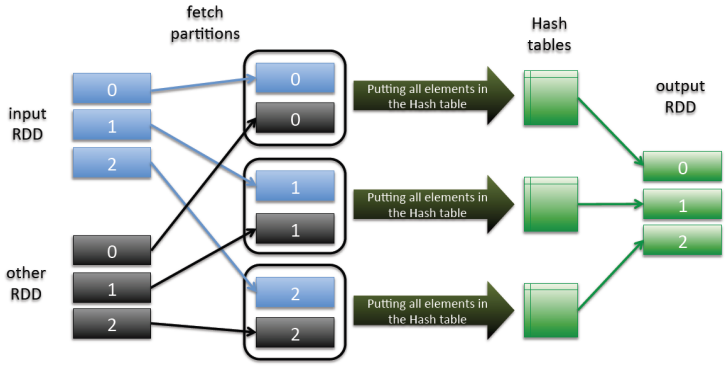



Figure 5.6: cogroup for case 0

Figure 5.7 shows cogroup for case 1. Small black rectangles represent blocks created from the previous stage. The tiny blocks with numbers on them represent local blocks (which do not need to be fetched from other Worker nodes remotely). Other blue, black or green rectangles represent RDD partitions. Here we assume that partition i (i = 0, 1, 2) of the input RDD is located on the same node as partition i of the other RDD. Partition 3 of the other RDD is located in a different node than partitions 0, 1 and 2. The input RDD has the same partitioner as the output RDD while the other RDD has a different partitioner.

We illustrate this process by explaining how partition 0 of output RDD is computed. After determining that input RDD has the same partitioner as output RDD, partition 0 of the input RDD is fetched locally. When turning to the other RDD, a different partitioner is found and then only a portion of the partition 0 (local block 0) is located on the same node where partition 0 of the input RDD is located. The local block fetching takes a very short time. Then the Spark starts the process of fetching remote blocks from other partitions (1, 2, 3). Step 2 starts immediately after the local block fetching and does not wait for the completion of remote fetching. In summary, step 1 deals with local fetching (fetch local blocks 0 or entire partition 0 of input and other RDD) while step 2 involves remote fetching and putting elements in the Hash table. For case 3, the process for input RDD of case 1 is replaced by the step for other RDD.

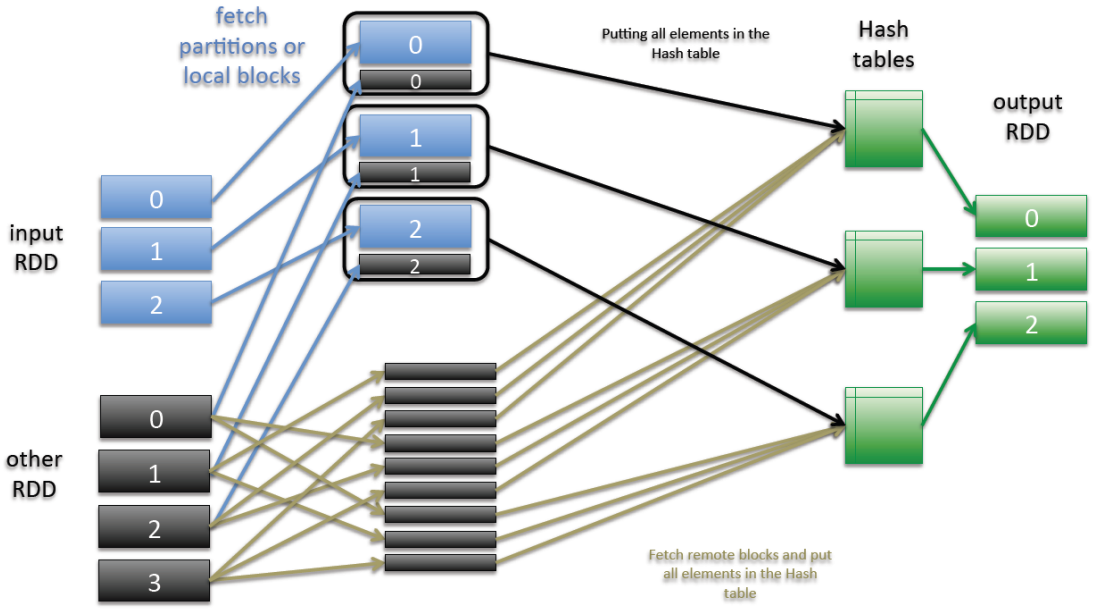



Figure 5.7: cogroup for case 1

## 5.3 co-partition and co-location

We now discuss the co-partition and co-location of two RDDs as they may influence the performance of cogroup. The term co-partition means two RDDs have the same partitioner. In this case, the cogroup of them does not involve shuffling between different partitions as

we discussed in section 5.2 (we assume the partitioner of output RDD is the same as input RDD). However, co-partitioning two RDDs does not mean there is no remote fetching of entire partitions if they are not co-located. We call two RDDs co-located if they are co-partitioned and their corresponding partitions are located in the same Worker node. Figure 5.8 shows the difference between co-partition and co-location.

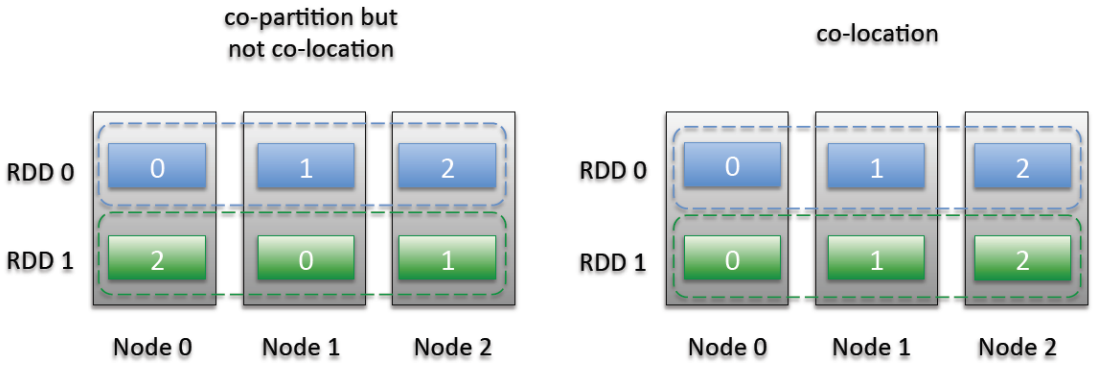
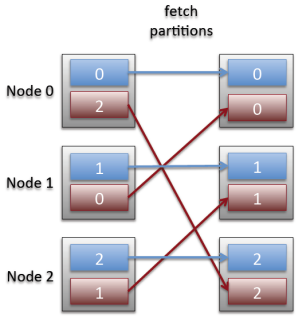
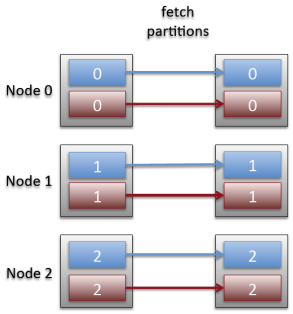



Figure 5.8: co-partition and co-location

When two co-partitioned RDDs are cogrouped into an output RDD with the same partitioner, the two source RDDs are co-located if the process of computing them and the cogroup transformation are in the same job triggered by an action. In this case, Spark makes their co-location possible by launching their computing stage at the same time, which means two stages may be running simultaneously before cogroup is launched. However, if the two RDDs are computed from different jobs and cached in memory after the computing, their co-location can not be guaranteed as the cogroup cannot choose their partitions' locations at this moment. Cogroup can not control an RDD's storage if this RDD is computed from another job. If the two RDDs are not co-located, the storage of the output RDD is aligned with the input RDD. Some partitions of the other RDD have to be fetched remotely in Step 1 of cogroup. Figure 5.9 shows how cogroup works in Step 1 for two co-partitioned RDDs which are not co-located and for two co-located RDDs, respectively.




(a) co-partition but not co-location




(b) co-location

Figure 5.9: cogroup step 1 for co-partitioned RDD

# Chapter 6

# Algorithm implementations in Spark

## 6.1 Pagerank in Spark

The Pagerank algorithm (without considering dangling nodes) has already been implemented in Spark [15]. In this section, we give a brief description of the Pagerank algorithm (without considering dangling nodes) implementation in Spark. First, we present the code for Pagerank from Spark package org.apache.spark.examples (see https://github.com/mateiz/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkPageRank.scala):

```
1  package org.apache.spark.examples
2
3  import org.apache.spark.SparkContext._
4  import org.apache.spark.{SparkConf, SparkContext}
5
6  /**
7   * Computes the PageRank of URLs from an input file. Input file should
8   * be in format of:
9   * URL         neighbor URL
10  * URL         neighbor URL
11  * URL         neighbor URL
12  * ...
13  * where URL and their neighbors are separated by space(s).
14  */
15 object SparkPageRank {
16   def main(args: Array[String]) {
```

```
17      // Configuration for a Spark application:
18      val sparkConf = new SparkConf().setAppName("PageRank")
19      // Specify the number of iterations:
20      var iters = args(1).toInt
21      // Create the SparkContext object:
22      val ctx = new SparkContext(sparkConf)
23      // Read input data line by line from text file:
24      val lines = ctx.textFile(args(0), 1)
25      // Transform each line to (key, value) pair, group all outlinks for
26      // each page:
27      val links = lines.map{ s =>
28        val parts = s.split("\\s+")
29        (parts(0), parts(1))
30      }.distinct().groupByKey().cache()
31      // Create an RDD representing the rank for each page:
32      var ranks = links.mapValues(v => 1.0)
33
34      for (i <- 1 to iters) {
35      // Create a (key, pair) RDD with the contributions sent by each page:
36        val contribs = links.join(ranks).values.flatMap{ case (urls, rank) =>
37          val size = urls.size
38          urls.map(url => (url, rank / size))
39        }
40        //Sum contributions for each page and update the ranks:
41        ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
42      }
43      // Collect ranks and print the results out:
44      val output = ranks.collect() // RDD action collect
45      output.foreach(tup => println(tup._1 + " has rank: " + tup._2 + "."))
46
47      ctx.stop()
48    }
49  }
```

In line 30, RDD links obtains a partitioner (HashPartitioner) as it is created from the transformation groupBykey. The number of partitions is determined by the block size of the text file read from HDFS or the local file system. RDD ranks (line 32) has the same partitioner as RDD links since it is created by applying the mapValues transformation to RDD links. Therefore RDDs links and ranks are co-partitioned and the join in line 36 does not involve shuffling between partitions as a result. In line 42, the new RDD ranks keeps

the partitioner of its parent RDD contribs, which enables the co-partition of RDDs in the next iteration. Moreover, the co-partition in this Pagerank implementation also guarantees the co-location of RDDs because this application only has one job (line 44).

## 6.2 CG in Spark

We now discuss how the Conjugate Gradient method is implemented in Spark. Here we assume the matrix **A** is a sparse matrix while the vector **b** is a dense vector (zero elements are also stored). At first, we present a naive implementation of CG, which is similar to its implementation in Hadoop. Then we discuss the drawbacks of this implementation and propose a new method called "Integrated CG" which implements CG in a different and efficient way. Finally, we present a way to optimize the integrated CG method by grouping elements into blocks.

### 6.2.1 Matrix and Vector operations in Spark

Before we detail the CG implementations, we present implementations of some matrix and vector operations in Spark as they are used in CG. The matrix and vector data is stored in HDFS as text files. Each line in the file represents an element. The format of the file has been described in section 4.1. We use a SparkContext object sc (see section 2.2.1) to read a file from HDFS and create an RDD representing a matrix or vector. The following Scala code shows how matrix and vector RDDs are created.

```
1  val sc = new SparkContext(master: String, appName: String, conf: SparkConf)
2  // read matrix data from HDFS file matrix.txt
3  val matrix = sc.textFile("hdfs:...../matrix.txt")
4  // create an RDD representing a matrix where a key is the column id and a value
       contains the row id and the matrix element.
5  val RDD_matrix = matrix.map{s =>
6    val parts = s.split("\\s+")
7    (parts(1).toInt, (parts(0).toInt, parts(2).toDouble))
8  }
9  // read vector data from HDFS file vector.txt
10 val vector = sc.textFile("hdfs:...../vector.txt")
11 // create an RDD representing a vector where a key is the row id and a value is
       the vector element.
12 val RDD_vector = vector.map{s =>
```

```
  val parts = s.split("\\s+")
  (parts(0).toInt, parts(1).toDouble)
}
```

Assume we want to implement vector operation $\theta\mathbf{a} + \gamma\mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are vectors, and $\theta$ and $\gamma$ are scalars. This can be done by using two RDD transformations: join and mapValues.

```
// compute theta * a + gamma * b,
// RDDs RDD_a and RDD_b represent vector a and b, aplusb is the RDD
   representing result vector
val aplusb = RDD_a.join(RDD_b).mapValues{ s =>
  theta * s._1 + gamma * s._2
}
```

After the join transformation, the value of each element of RDD aplusb is a Tuple2 object with the vector element of $\mathbf{a}$ as its first element and the vector element of $\mathbf{b}$ as its second element.

Computing a vector scalar product requires two RDD transformations and one RDD action: join, map, and reduce. Here we use map transformation instead of mapValues since we no longer need to keep the keys.

```
// compute a * b,
// RDDs RDD_a and RDD_b represent vector a and b, atimesb is scalar product of
   the two vectors
val atimesb = RDD_a.join(RDD_b).map{ case(k, v) => v._1 * v._2}.reduce(_ + _)
```

For computing the vector square, only a map transformation and a reduce action are needed.

```
// compute a * a,
// RDDs RDD_a represents the vector a, asquare is square of the vector
val asquare = RDD_a.map{ case(k, v) => v * v}.reduce(_ + _)
```

Now we detail how the GIM-V algorithm (discussed in 4.1.1) is implemented in Spark to do Matrix-Vector Multiplication. At first, all elements in each column must be grouped together using the RDD transformation groupByKey. To multiply each element of a column with the corresponding vector element, we first need to apply the join transformation to put each column and the corresponding vector element together. Then the flatMap transformation can perform the multiplication (compute $Q_{i,j}$, see section 4.1.1), changing

key to the row id. The reduceByKey finishes the whole process by adding all $Q_{i,*}$ for each $i$ together. The following Scala code illustrates how $\mathbf{Ax}$ is computed in Spark:

```
// compute Ax:
// RDD RDD_x represents the vector x, RDD_matrix stores the matrix elements
   with column id as the key as we presented before, RDD_Ax represents the
   computed vector Ax.
val RDD_Ax = RDD_matrix.groupByKey() // group elements by column
                       .join(RDD_x) // join with vector elements
                       .flatMap{case(k, v) => // multiply with vector elements
                          v._1.map(mv => (mv._1, mv._2 * v._2))}
                       .reduceByKey(_ + _) // add Q values by row
```

### 6.2.2 Naive implementation of CG

In this section, we detail the naive implementation of CG. This implementation is based on the implementations of matrix and vector operations described in section 6.2.1. Table 6.1 reports the RDDs created in our CG implementation.

| RDD name | type | description |
|---|---|---|
| matrix | val | represents matrix $\mathbf{A}$, stores matrix elements using (key, value) pairs |
| A1 | val | represents matrix $\mathbf{A}$, stores matrix elements by column |
| r0 | val | represents vector $\mathbf{r_0}$ |
| xk | var | represents vector $\mathbf{x_k}$ for all iterations |
| rk | var | represents vector $\mathbf{r_k}$ for all iterations |
| pk | var | represents vector $\mathbf{p_k}$ for all iterations |
| Apk | val | represents vector $\mathbf{Ap_k}$ for all iterations |

Table 6.1: RDDs in CG implementation

The RDDs representing vectors (r0, xk, rk, pk, Apk) have the same format as RDD_vector in section 6.2.1 (RDD[(Int, Double)]). RDD matrix has the same format as RDD_matrix in the previous section (RDD[(Int, (Int, Double))], where the first Int is the column and the second Int is the row). RDD A1 has the format of RDD[(Int, Seq[(Int, Double)])] as it is the output RDD from applying groupByKey to RDD matrix. Most of the RDDs listed in the table above should be cached as they are used more than once. RDD A1 must be cached as it is used at the beginning to compute $\mathbf{Ax_0}$ and in all iterations to compute

$\mathbf{Ap_k}$. The RDDs rk, pk, and Apk are used twice in each iteration and they should also be cached after they are updated in the iteration if possible. The algorithm implementation is given as follows:

**Algorithm 14** Naive CG in Spark

1: Read data for $\mathbf{A}$ from HDFS and create a (key, value) pair RDD (matrix) for $\mathbf{A}$.
2: Group all elements in each column by applying groupByKey to matrix (create RDD A1), then cache A1 in memory.
3: Read vector data for $\mathbf{b}$ and $\mathbf{x_0}$ and create (key, value) pair RDDs (b for $\mathbf{b}$ and x0 for $\mathbf{x0}$).
4: Create RDD Ax0 for $\mathbf{Ax_0}$ using A1, x0.
5: Create RDD r0 for $\mathbf{r_0} = \mathbf{b} - \mathbf{Ax_0}$ using Ax0 and b.
6: xk $\leftarrow$ x0
7: rk $\leftarrow$ r0
8: pk $\leftarrow$ r0
9: Compute $\mathbf{r_0^T r_0}$ using r0 (the RDD action reduce is used)
10: $k \leftarrow 0$
11: repeat
12: Create RDD Apk using A1 and pk and cache Apk in memory.
13: Compute $\mathbf{p_k^T A p_k}$ using Apk and pk (RDD action reduce is used)
14: Compute $\alpha_k = \frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T A p_k}}$ ($\mathbf{r_k^T r_k}$ is obtained from the previous iteration).
15: /* Compute $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k}$ */
16: Update xk using pk and $\alpha_k$, the updated RDD xk represents $\mathbf{x_{k+1}}$.
17: /* Compute $\mathbf{r_{k+1}} = \mathbf{r_k} - \alpha_k \mathbf{Ap_k}$ */
18: Update rk using Apk and $\alpha_k$, the updated RDD rk represents $\mathbf{r_{k+1}}$.
19: Cache RDD rk in memory.
20: Compute $\mathbf{r_{k+1}^T r_{k+1}}$ using rk (the RDD action reduce is used)
21: If $\mathbf{r_{k+1}^T r_{k+1}}$ is sufficiently small then exit loop
22: Compute $\beta_k = \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
23: /* Compute $\mathbf{p_{k+1}} = \mathbf{r_{k+1}} + \beta_k \mathbf{p_k}$ */
24: Update pk using rk and $\beta_k$, the updated RDD pk represents $\mathbf{p_{k+1}}$.
25: Cache RDD pk in memory.
26: $k \leftarrow k + 1$
27: end repeat
28: Save RDD xk as text file in HDFS (RDD action saveAsTextFile is used)

In Algorithm 14, each iteration adds newly created RDDs pk, Apk, and rk in memory. After many iterations the total size of RDDs in memory can grow to be extremely large. Once the amount of memory is running out, Spark removes some old RDD partitions cached in memory if a newly created RDD needs to be cached. The oldest RDD in memory is removed first and Spark stops removing RDDs once enough space is released to cache the newly created RDD. Here "oldest" does not mean this RDD is created before all the other RDDs; it means this RDD has been left unused for the longest time. For example, during the 100th iteration, RDD rk created in the first iteration is the oldest RDD in memory as it has not been used since the second iteration. However, RDD A1 is used in every iteration and should not be regarded as the oldest RDD even though it was created at the beginning. Therefore RDD rk created in the first iteration should be removed automatically once memory is running out. Figure 6.1 illustrates how RDDs are removed from memory.

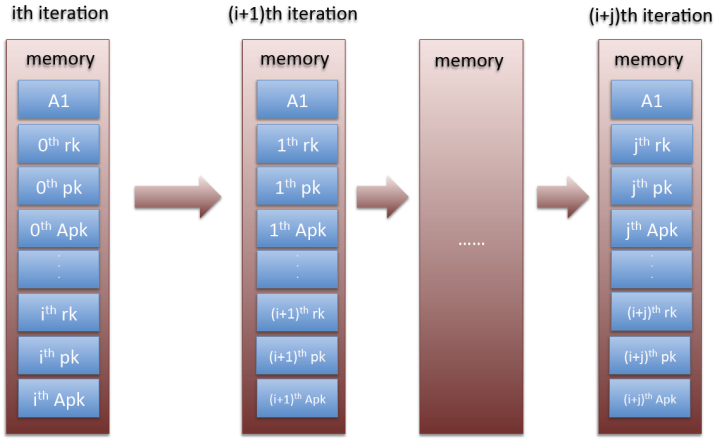



Figure 6.1: Removing RDDs from memory in CG when memory is running out since the $i$th iteration

In each iteration there are RDD actions that trigger the computation of pk, Apk, and rk. Therefore it does not matter if all their ancestor RDDs are still in memory as long as the memory is large enough to cache RDD A1 and RDDs created in any two consecutive iterations. However, note that there are no RDD actions to trigger the computation of RDD xk created in each iteration except the action in line 28, which means computation of RDDs xk from all iterations starts after the loop ends. As we know, RDD xk in each iteration depends on RDD pk in that iteration. If there is enough memory that no RDDs have been removed, then everything is fine as each pk can be obtained from memory and used to compute each xk. However, many iterations may be required in CG, therefore,

removing RDDs from memory becomes inevitable. Some old RDDs (like RDDs rk, pk, and Apk created in the first few iterations) may have been removed from memory after the loop is finished as they have not been used for a long time. In this case, removed RDDs pk must be recomputed; this causes additional problems as each RDD pk depends on the corresponding RDDs rk and Apk which may also have been removed from memory. One way to avoid this problem is to cache RDD xk from each iteration and force it to be computed immediately after its creation. One may apply a simple action which takes very little time to RDD xk. For example, using the RDD action count may only take less than a second which is a small amount of time for a big CG problem.

There are several sources of potential inefficiency in this naive implementation. For each iteration in Algorithm 14, five join transformations are applied to RDDs. The join method is actually based on cogroup by flattening values of output RDD from cogroup:

```
1 // for an RDD[(K,V)], the join applies to it as follows
2 def join(other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))] = {
3   this.cogroup(other, partitioner).flatMapValues { case (vs, ws) =>
4     for (v <- vs; w <- ws) yield (v, w)
5   }
6 }
```

When we join two vectors, it takes very little time for the flatMapValues transformation to run, since vs (Seq[V]) and ws (Seq[W]) only have at most one element respectively. As the size of vectors increases, the cogroup transformation could be quite time consuming as it may involve shuffling between different Worker nodes. The fact that cogroup puts data in the Hash tables can also harm the performance once the memory is full. Moreover, co-location for some join transformations in our implementation cannot be guaranteed. The five join transformations in Algorithm 14 are: A1 and pk (line 12), Apk and pk (line 13), xk and pk (line 16), rk and Apk (line 18), and pk and rk (line 24). RDDs pk and Apk are computed in line 13 while RDDs A1, rk and xk are computed in line 9 (where the computation of A1 is triggered by an RDD action for computing $\mathbf{r_0^T r_0}$), 20 and 28 (or some place within the loop from a naive action like count as we mentioned above), respectively. RDD pk and its child RDD Apk are computed in the same job, therefore they are co-located with RDD A1 since RDDs A1 and pk are co-located. The co-location for the join applied in line 12 and 13 is guaranteed as long as A1 is co-partitioned with pk. However, for the join in line 18, input RDD, other RDD and output RDD are computed in different jobs: rk is computed in line 20 from the previous iteration, Apk is computed in line 13 in this iteration, and the updated rk is computed in line 20 in this iteration. This may cause RDD rk and RDD Apk to not be co-located as we explained in section 5.3.

Moreover, the RDD xk and pk in line 16 may not be co-located if one chooses to trigger the computation of xk in each iteration by, for example, applying an RDD action first() to xk at the end of each iteration. Fetching partitions remotely also consumes quite a bit of time, therefore join without co-location can strongly harm the performance. To avoid or at least reduce those join transformations and the potential problems they may cause (absence of co-location), we develop a different method for CG in Spark by changing the storage of matrices and vectors (see section 6.2.3).

### 6.2.3 Integrated CG method

We now detail a new method called Integrated CG which is much more efficient than the naive implementation method. In this method, four join transformations can be avoided by integrating $\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{p_k}$ and $\mathbf{A}$ in the same RDD. Before we start describing our algorithm, we list some variables used in the implementation (Table 6.2). Each element of sequence $A_i$ is a Tuple2 (2-tuple) representing a matrix element in the $i$th column of $\mathbf{A}$ with as first element the row number and as second element the matrix element.

| variable | type | description |
|---|---|---|
| $x_i$ | Double | the $i$th element of $\mathbf{x_k}$ |
| $r_i$ | Double | the $i$th element of $\mathbf{r_k}$ |
| $p_i$ | Double | the $i$th element of $\mathbf{p_k}$ |
| $Apk_i$ | Double | the $i$th element of $\mathbf{Ap_k}$ |
| $A_i$ | Seq[(Int, Double)] | a sequence of all elements in the $i$th column of $\mathbf{A}$ |

Table 6.2: Variables in each iteration of the Integrated CG implementation

Three types of RDDs are created in each iteration: X, X1, and Apk. RDD Apk has the same format as in section 6.2.2 and represents vector $\mathbf{Ap_k}$. RDD X has the type RDD[(Int, (Double, Double, Double, Seq[(Int, Double)]))] and each element of X has structure ($i$, ($x_i$, $r_i$, $p_i$, $A_i$)) with the key indicating the row number of vectors and the column number of the matrix at the same time. Therefore all data of $\mathbf{x_k}$, $\mathbf{r_k}$, $\mathbf{p_k}$ and $\mathbf{A}$ is stored in X and parallelized by putting the corresponding element of the vectors and column of the matrix in the same Tuple. Each element of RDD X1 has the type RDD[(Int, (Double, Double, Double, Double, Seq[(Int, Double)])) and the extra element in the Tuple is used to store $Apk_i$: one element of X1 stores ($i$, ($x_i$, $r_i$, $p_i$, $Apk_i$, $A_i$)). With such kind of storage, vector additions in Algorithm 1 (line 6, 7 and 10) can be done by applying mapValue transformations to X or X1 instead of using join like in Algorithm 14. Moreover, join

can also be avoided for the Matrix-Vector Multiplication $\mathbf{Ap_k}$ as each element of the $i$th column of $\mathbf{A}$ (stored in $A_i$) can be multiplied by $r_i$ using mapValue as well. Algorithm 15 provides pseudo code of our implementation.

**Algorithm 15** Integrated CG in Spark

1: Create RDD X by reading data for $\mathbf{A}$, $\mathbf{x_0}$, and $\mathbf{b}$ from HDFS. RDD X stores data for $\mathbf{x_0}$, $\mathbf{r_0}$, $\mathbf{p_0}$ and $\mathbf{A}$.
2: Cache RDD X in memory.
3: Compute $\mathbf{r_0^T r_0}$ using X. (here RDD action reduce is used)
4: $k \leftarrow 0$
5: repeat
6: Create RDD Apk using X.
7: Create RDD X1 by cogrouping X and Apk, and cache X1 in memory.
8: Compute $\mathbf{p_k^T Ap_k}$ using X1. (here RDD action reduce is used)
9: Compute $\alpha_k = \frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T Ap_k}}$ ($\mathbf{r_k^T r_k}$ is obtained from the previous iteration).
10: /* Compute $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k}$ and $\mathbf{r_{k+1}} = \mathbf{r_k} - \alpha_k \mathbf{Ap_k}$ */
11: Update X using X1 and $\alpha_k$, the updated X stores data for $\mathbf{x_{k+1}}$, $\mathbf{r_{k+1}}$, $\mathbf{p_k}$ and $\mathbf{A}$.
12: Cache RDD X in memory.
13: Compute $\mathbf{r_{k+1}^T r_{k+1}}$ using X. (here RDD action reduce is used)
14: If $\mathbf{r_{k+1}^T r_{k+1}}$ is sufficiently small then exit loop
15: Compute $\beta_k = \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
16: /* Compute $\mathbf{p_{k+1}} = \mathbf{r_{k+1}} + \beta_k \mathbf{p_k}$ */
17: Update X using $\beta_k$, the updated X stores data for $\mathbf{x_{k+1}}$, $\mathbf{r_{k+1}}$, $\mathbf{p_{k+1}}$ and $\mathbf{A}$.
18: Cache RDD X in memory.
19: $k \leftarrow k + 1$
20: end repeat
21: Create an RDD x (which only stores data for $\mathbf{x_k}$) using RDD X and save it as text file in HDFS. (RDD action saveAsTextFile is used)

Now we detail how each step in Algorithm 15 is done. The initial RDD X is created by using several join transformations as it stores both matrix and vector data. At first, some RDDs in Algorithm 14 (A1, x0, b, Ax0) should be created before creating X. Algorithm 15 describes how the initial X (line 1 in Algorithm 15) is created.

**Algorithm 16** Create RDD X

1: Read data for **A** from HDFS and create a (key, value) pair RDD (matrix) for **A**.
2: Group all elements in each column by applying groupByKey to matrix (create RDD A1).
3: Read vector data for **b** and $\mathbf{x_0}$ and create (key, value) pair RDDs (b for **b** and x0 for $\mathbf{x0}$).
4: Create RDD X0 using A1, x0 and b; each element of X0 stores corresponding elements of A1, x0 and b: $(i, (x_i, b_i, A_i))$. Cache X0 in memory.
5: Create RDD Ax0 for $\mathbf{Ax_0}$ using X0.
6: Create RDD X using X0 and Ax0.

When we create RDD X0, the cogroup transformation is used to cogroup three RDDs: A1, x0, b. Then mapValues is used to transform the RDD element type to (Int, (Double, Double, Seq[(Int, Double)])). The value is a Tuple3 representing $x_i$, $b_i$ and $A_i$ respectively, where $b_i$ is the $i$th element of **b**. X0 can be used to create RDD Ax0 using the same scheme as in section 6.2.1. RDD X is created by grouping X0 with Ax0 and then computing $r_i$ ($p_i = r_i$ for k = 0). Figure 6.2 illustrates the process of creating RDD X and the structure of each RDD. Here $Ax_i$ represents the $i$th element of vector $\mathbf{Ax_0}$ and $r_i = p_i = b_i - Ax_i$.

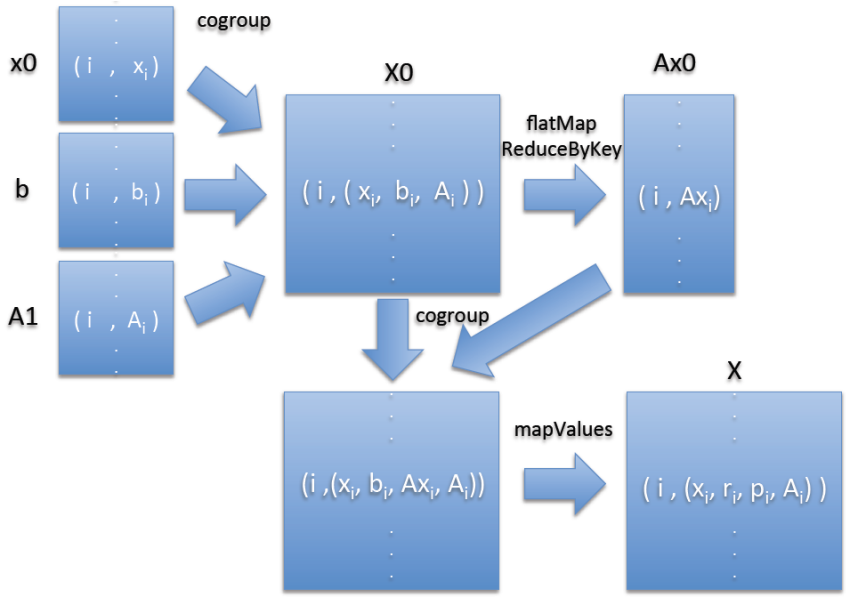



Figure 6.2: Create initial RDD X

During each iteration, only one cogroup is involved (line 7 in Algorithm 15). The RDDs to be cogrouped (X and Apk) are co-located if they are co-partitioned as they are computed in the same job (line 8). As for the vector additions, only the mapValues transformation is used in our implementation. Line 10 updates RDD X by updating $x_i$ $(= x_i + \alpha_k p_i)$ and $r_i$ $(= r_i - \alpha_k Apk_i)$ using the mapValues transformation, and so does line 15 $(p_i \leftarrow r_i + \beta_k p_i)$. Figure 6.3 shows how our implementation works in each iteration.

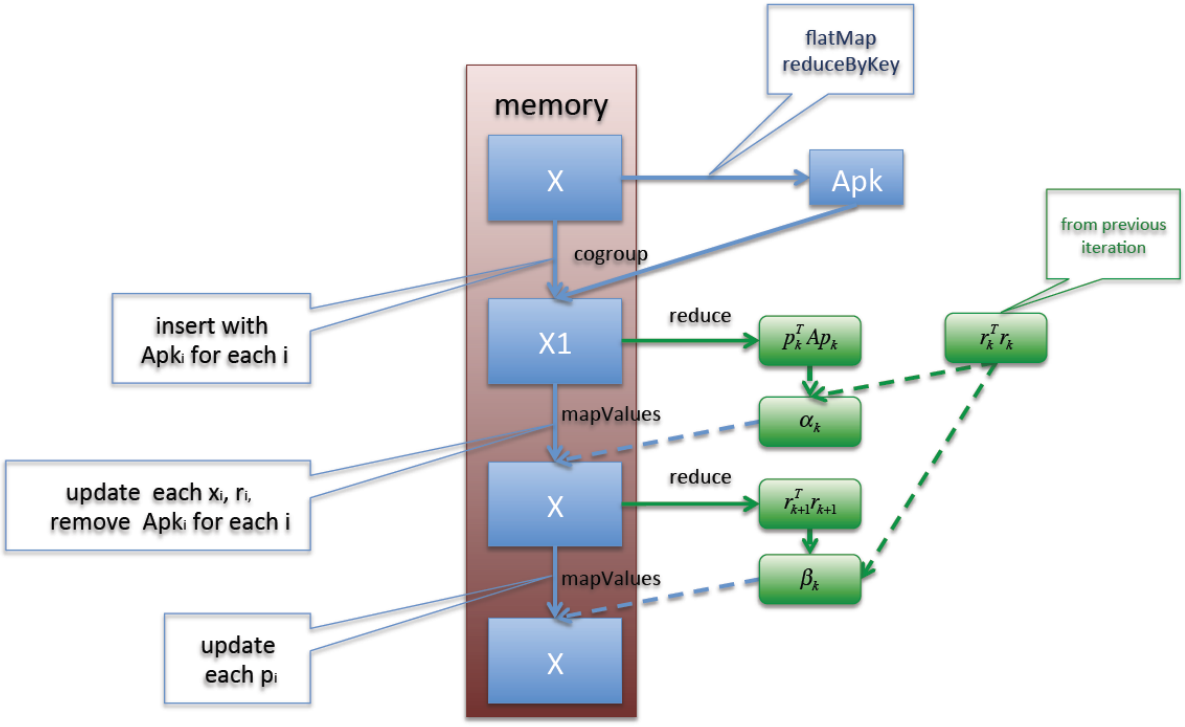



Figure 6.3: $k$th iteration in Integrated CG in Spark

Note that the size of RDD X or X1 could be very large compared to RDD A1. In each iteration, three RDDs X and one RDD X1 are added in memory. Each RDD X or X1 cached in memory only depends on its parent RDD and is used only by its child RDD of X or X1. Therefore, the oldest RDD (in this case it is the first RDD created and cached in memory) should be removed from memory first once there is insufficient memory to cache more RDD partitions. It is very important to make sure that the memory used to cache RDDs is large enough to cache any two consecutive RDDs X or X1 (all RDDs X have almost the same size, and RDDs X1 are a little larger than an RDD X). Otherwise, for example,

if the memory is running out during the computing of the second RDD (X1) in Figure 6.3, then some partitions of the first RDD (X) must be removed from memory. However, the partitions removed may be needed to compute some partitions of RDD X1 which are not computed yet (Figure 6.4). If this happens, those removed partitions should be computed again and the re-computing of X makes our implementation quite inefficient and even inapplicable. Moreover, since all RDDs cached in memory have sequential dependency, re-computing an RDD requires re-computing all of its ancestor RDDs since they are older and would have been removed before the desired RDD. So it is essential to have sufficient memory to cache any two consecutive RDDs X or X1.

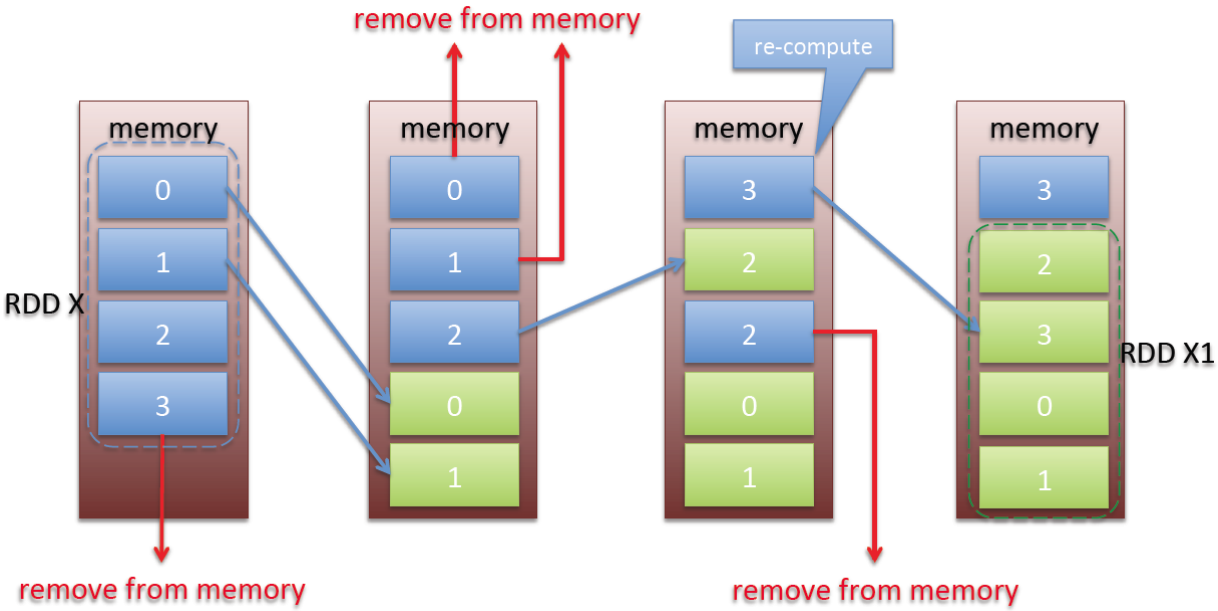



Figure 6.4: Remove RDD partitions from memory. Here a blue or green rectangle represents an RDD partition. RDD X1 narrowly depends on RDD X.

### 6.2.4 Half-integrated CG method

To solve the memory shortage that may occur in the Integrated CG implementation, we can remove the data for matrix **A** in RDD X and X1 at the expense of having one more cogroup or join transformation in each iteration. We call this method Half-integrated CG. The RDD A1 from the naive CG implementation is created here and cached in memory

for use in each iteration. The types and formats of RDD X and X1 in the Half-integrated CG are shown in Table 6.3.

| RDD | element type | format |
|---|---|---|
| X | (Int, (Double, Double, Double)) | $(i, (x_i, r_i, p_i))$ |
| X1 | (Int, (Double, Double, Double, Double)) | $(i, (x_i, r_i, p_i, Apk_i))$ |

Table 6.3: Types of RDDs in Half-integrated CG

The algorithm description is given in Algorithm 17. The main differences between Algorithm 15 and 17 are:

- Algorithm 17 has one more cogroup (line 8);
- RDD A1 is cached in memory in Algorithm 17.

Figure 6.5 illustrates how Half-integrated CG works in the $k$th iteration. As we can see, RDDs X and X1 in Figure 6.5 become smaller compared to Figure 6.3. RDD A1 can always remain in memory as it is used in each iteration and will not be chosen to be removed first if memory runs out.

**Algorithm 17** Half-integrated CG in Spark

1: Read data for $\mathbf{A}$ from HDFS and create a (key, value) pair RDD (matrix) for $\mathbf{A}$.
2: Group all elements in each column by applying groupByKey to matrix (create RDD A1), then cache A1 in memory.
3: Create RDD X by reading data for $\mathbf{x_0}$, and $\mathbf{b}$ from HDFS and using RDD A1. RDD X stores data for $\mathbf{x_0}$, $\mathbf{r_0}$, $\mathbf{p_0}$.
4: Cache RDD X in memory.
5: Compute $\mathbf{r_0^T r_0}$ using X. (here RDD action reduce is used)
6: $k \leftarrow 0$
7: repeat
8: Create RDD Apk by cogrouping X and A1.
9: Create RDD X1 by cogrouping X and Apk and cache X1 in memory.
10: Compute $\mathbf{p_k^T A p_k}$ using X1. (here RDD action reduce is used)
11: Compute $\alpha_k = \frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T A p_k}}$ ($\mathbf{r_k^T r_k}$ is obtained from the previous iteration).
12: /* Compute $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k}$ and $\mathbf{r_{k+1}} = \mathbf{r_k} - \alpha_k \mathbf{A p_k}$ */
13: Update X using X1 and $\alpha_k$, the updated X stores data for $\mathbf{x_{k+1}}$, $\mathbf{r_{k+1}}$, $\mathbf{p_k}$ and $\mathbf{A}$.
14: Cache RDD X in memory.
15: Compute $\mathbf{r_{k+1}^T r_{k+1}}$ using X. (here RDD action reduce is used)
16: If $\mathbf{r_{k+1}^T r_{k+1}}$ is sufficiently small then exit loop
17: Compute $\beta_k = \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
18: /* Compute $\mathbf{p_{k+1}} = \mathbf{r_{k+1}} + \beta_k \mathbf{p_k}$ */
19: Update X using $\beta_k$, the updated X stores data for $\mathbf{x_{k+1}}$, $\mathbf{r_{k+1}}$, $\mathbf{p_{k+1}}$ and $\mathbf{A}$.
20: Cache RDD X in memory.
21: $k \leftarrow k + 1$
22: end repeat
23: Create an RDD x (which only stores data for $\mathbf{x_k}$) using RDD X and save it as text file in HDFS. (RDD action saveAsTextFile is used)

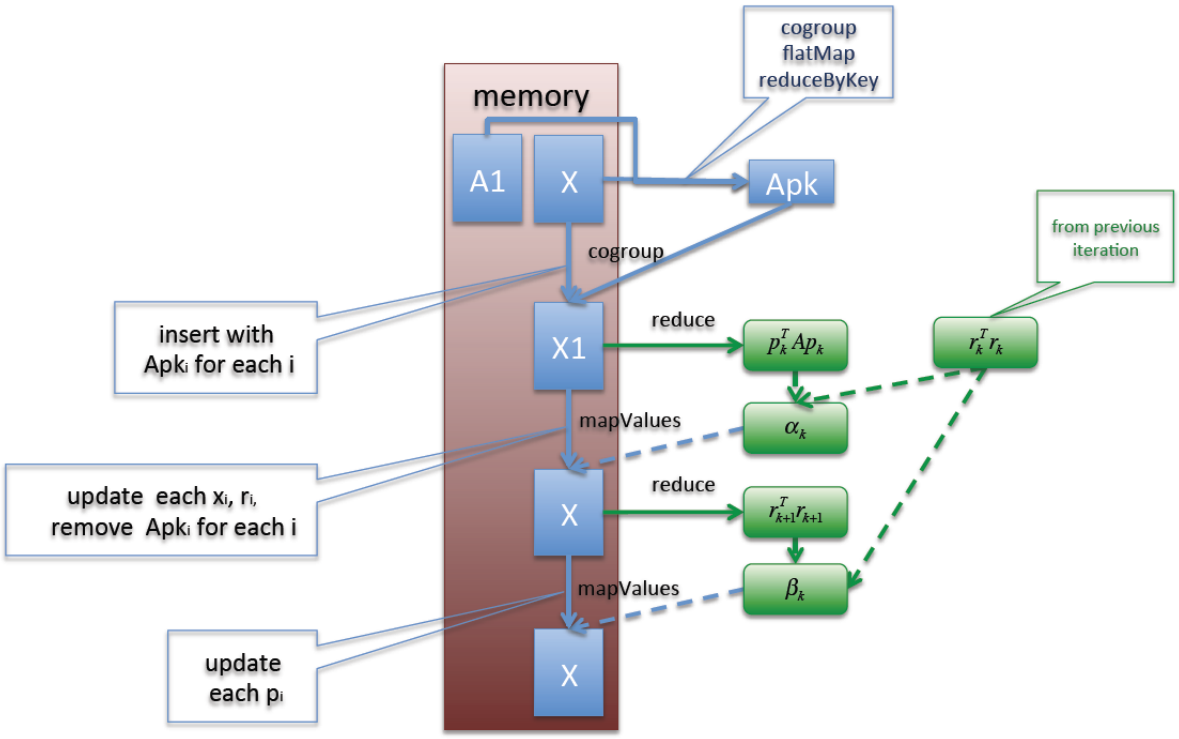



Figure 6.5: $k$th iteration in Half-integrated CG in Spark

### 6.2.5 Optimization using blocks

Now we discuss how to optimize the Integrated CG implementation in Spark by grouping elements into blocks. We assume that $\mathbf{A}$ is an $M \times M$ matrix and $\mathbf{x_0}$ and $\mathbf{b}$ are $M \times 1$ vectors. Therefore, in our Integrated CG, RDDs X, X1 and Apk are supposed to have $M$ elements. Our idea is to group elements of each RDD into blocks according to their keys. Suppose the block size is $B$, then new types of RDDs X, X1, and Apk are described in Table 6.4.

| RDD | element type |
|---|---|
| X | (Int, Array[(Double, Double, Double, Seq[(Int, Double)])](B)) |
| X1 | (Int, Array[(Double, Double, Double, Double, Seq[(Int, Double)])](B)) |
| Apk | (Int, Array[Double](B)) |

Table 6.4: Types of RDDs in Integrated CG with blocks

The format of each element of RDD X, X1 or Apk is ($BlockId$, $Block$). For the $i$th element of RDD X, X1 or Apk in section 6.2.3, it is stored in a block with $BlockId = [\frac{i}{B}]$ (here [] means function, $[x]$ is the largest integer not greater than $x$) at position ($i$ mod $B$) in the $Block$. For the element of X, X1 or Apk with $BlockId = j$, its value ($Block$) stores ($x_i$, $r_i$, $p_i$, $A_i$), ($x_i$, $r_i$, $p_i$, $Apk_i$, $A_i$) or $Apk_i$ for $j \times B \leq i < (j+1) \times B$. Using blocks can improve the performance by reducing the time for shuffling in each iteration.

# Chapter 7

# Spark performance experiments

Now we present performance test results for the algorithms we implement in Spark. Our tests are done on the gridbase cluster and the properties for each Worker node are given in Table 7.1. The node gridbase 1 also acts as Master node in our tests. The executor on each Worker, if not specified, is allocated with 5.0 GB memory. Therefore, 60 percent of total memory (15.0 GB), namely 9.0 GB, can be used to cache RDDs by default.

| gridbase | No. of cores | memory | No. of processors | Name of processor |
|---|---|---|---|---|
| 1 | 4 | 6.8 GB | 2 | Dual-Core AMD Opteron(tm) 2218@2.6Hz |
| 2 | 8 | 6.8 GB | 2 | Intel(R) Xeon(R) CPU X5460@3.16GHz |
| 3 | 16 | 14.7 GB | 4 | Intel(R) Xeon(R) CPU X5460@3.16GHz |

Table 7.1: The properties of Worker nodes on the gridbase cluster

## 7.1 Pagerank tests in Spark

Now we present our test results for Pagerank in Spark (see section 6.1). In our test, we use 12 cores in total so 4 cores are used on each Worker node. Table 7.2 reports the details of adjacency matrices for the input graphs from http://snap.stanford.edu/. Some pages in each graph may not have inlinks and they are removed according to the algorithm in sec 6.1. In our tests, we used 10 iterations to compute the ranks for pages. The Pagerank algorithm in Spark is scalable on a cluster with fixed size when the input graph is not large. However, the performance becomes much worse once there is not enough memory to fit the entire graph (see Figure 7.1). We found that the time for later iterations

increased dramatically because of the Garbage Collection (GC) time, which is discussed in Chapter 7.3.

| Graph | Nodes | Edges | size |
|---|---|---|---|
| cit-Patents | 3,774,768 | 16,518,948 | 267 MB |
| soc-pokec-relationships | 1,632,803 | 30,622,564 | 404 MB |
| soc-LiveJournal1 | 4,847,571 | 68,993,773 | 1030 MB |
| com-orkut.ungraph | 3,072,441 | 117,185,083 | 1687 MB |

Table 7.2: Graphs used in our Pagerank tests

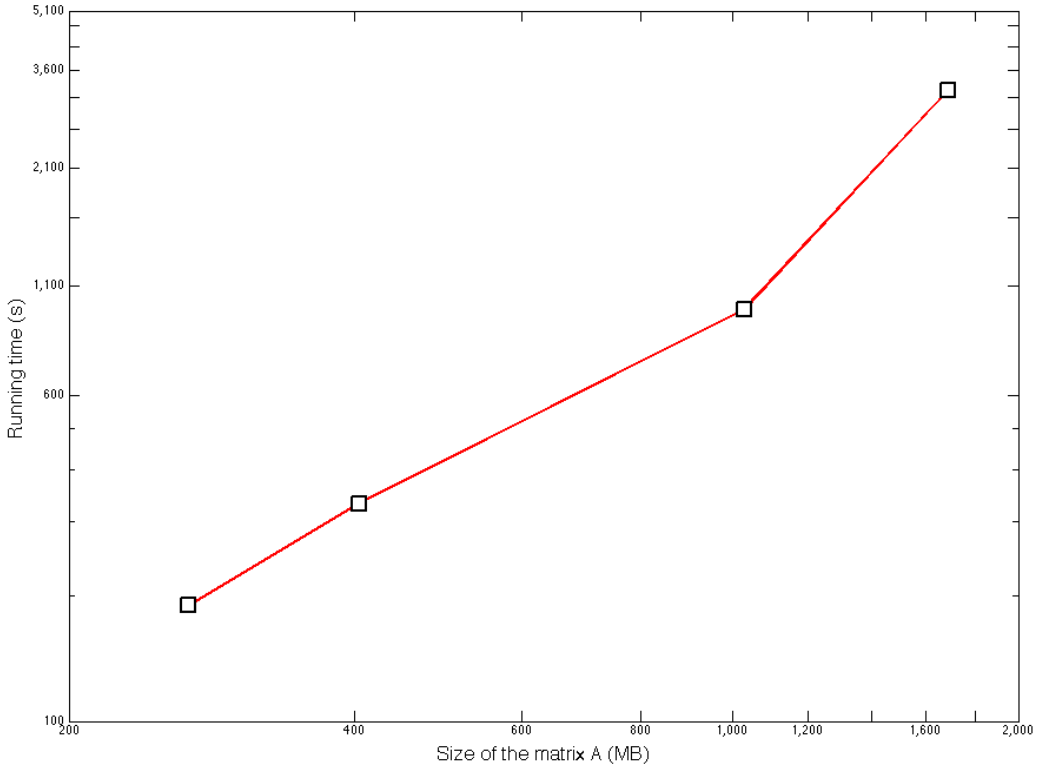



Figure 7.1: Pagerank tests in Spark

## 7.2 Tests for CG in Spark

Three matrices and six vectors are used in our tests for our CG implementation. Thus we can do tests for three problems $\mathbf{A_i x} = \mathbf{b_i}$ of different sizes. The details about the matrices ($\mathbf{A_0}$, $\mathbf{A_1}$, $\mathbf{A_2}$) and vectors ($\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{x_{00}}$, $\mathbf{x_{01}}$, $\mathbf{x_{02}}$) are shown in Table 7.3 and Table 7.4, respectively. All matrices and vectors are generated from MATLAB and stored as text files (only nonzero elements are stored). The matrices $\mathbf{A_i}$ are block diagonal where each block is 10-by-10 symmetric positive definite (SPD) matrix. All vectors are dense vectors generated randomly. Therefore, $\mathbf{A_i}$ are SPD and CG can be used to solve problems $\mathbf{A_i x} = \mathbf{b_i}$. Moreover, we run 10 iterations for each CG problem in our tests regardless of matrix size.

| Matrix | Number of rows | No. of nonzero elements | size |
|---|---|---|---|
| $\mathbf{A_0}$ | 10,000,000 | 28,000,000 | 921 MB |
| $\mathbf{A_1}$ | 5,000,000 | 14,000,000 | 458 MB |
| $\mathbf{A_2}$ | 2,500,000 | 7,000,000 | 226 MB |

Table 7.3: Matrices used in our CG tests.

| Vector | Number of components | No. of elements | size |
|---|---|---|---|
| $\mathbf{b_0}$, $\mathbf{x_{00}}$ | 10,000,000 | 10,000,000 | 248 MB |
| $\mathbf{b_1}$, $\mathbf{x_{01}}$ | 5,000,000 | 5,000,000 | 124 MB |
| $\mathbf{b_2}$, $\mathbf{x_{02}}$ | 2,500,000 | 2,500,000 | 61 MB |

Table 7.4: Vectors used in our CG tests.

At first, we investigate the scalability of the algorithms we proposed in section 6.2: the Naive CG, Half-integrated CG, Integrated CG and Integrated CG with blocks. In our tests, 12 cores are used in total therefore each Worker node is allocated four cores. The test results are shown in Figure 7.2. The Naive CG method is not efficient when compared with other methods as it has too many cogroup transformations in each iteration. The Half-integrated CG is faster than the Naive CG because it puts the data for the vectors together. The Integrated CG improves the performance of the Half-integrated CG by inserting matrix data into the integrated vector data. Grouping elements into blocks also improves performance a bit by reducing shuffling between partitions. In our tests for Integrated CG with blocks, we set the block size to be 50 for the problems $\mathbf{A_0 x} = \mathbf{b_0}$ and $\mathbf{A_1 x} = \mathbf{b_1}$, and 25 for the problem $\mathbf{A_2 x} = \mathbf{b_2}$. Using blocks in the Integrated CG method

can reduce the shuffle write and read in each iteration (see Table 7.5 and Table 7.6). The shuffle read/write in the tables means the total shuffle read/write for all stages in one iteration. Here the shuffle read/write for a stage can be observed in the Spark web UI (see Figure 2.8). The shuffle read for a stage shows the amount of data read from previous stage remotely. The shuffle write for a stage means the amount of data written to different Worker nodes in this stage. Moreover, grouping elements into blocks also reduces the size of RDD X or X1 a bit. For problem $\mathbf{A_0x} = \mathbf{b_0}$, we found that RDD X or X1 can be about 3.8 GB for the Integrated CG while the size of X or X1 can be reduced to at most 3.3 GB using blocking, which may be very helpful if there is not enough memory.

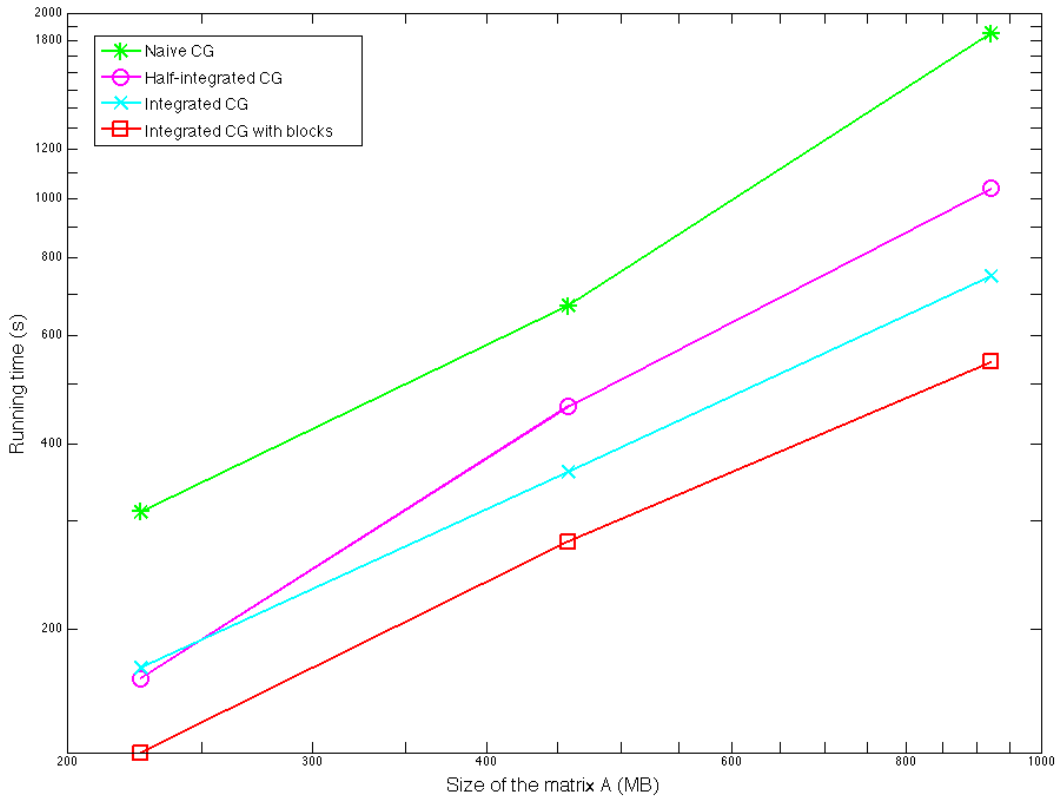



Figure 7.2: The scalability of our algorithms for CG in Spark

| CG problem | shuffle read | shuffle write | No. of partitions for each RDD |
|---|---|---|---|
| $\mathbf{A_0x = b_0}$ | 144.2 MB | 458.5 MB | 96 |
| $\mathbf{A_1x = b_1}$ | 80.0 MB | 230.4 MB | 48 |
| $\mathbf{A_2x = b_2}$ | 52.2 MB | 115.7 MB | 24 |

Table 7.5: Shuffle read/write for the first iteration in Integrated CG

| CG problem | shuffle read | shuffle write | No. of partitions for each RDD |
|---|---|---|---|
| $\mathbf{A_0x = b_0}$ | 0.0 MB | 224.9 MB | 96 |
| $\mathbf{A_1x = b_1}$ | 0.0 MB | 112.5 MB | 48 |
| $\mathbf{A_2x = b_2}$ | 0.0 MB | 58.3 MB | 24 |

Table 7.6: Shuffle read/write for the first iteration in Integrated CG using blocks

## 7.3 Spark GC time

In this section, we discuss how to reduce the Spark GC time using serialized RDD storage. Jave virtual machine (JVM) garbage collection (GC) can harm the performance in Spark when one has large RDDs in the Spark application [34]. The running time of GC is proportional to the number of objects in the program. As suggested in the Spark tuning guide [34], the first thing to try is to use serialized caching as there will be only one object per RDD partition. Moreover, the size of a serialized RDD in memory becomes much smaller, which means more memory can be used for task execution instead of memory caching to make the application more efficient. The fraction of executor memory to use for Spark's memory cache is controlled by the parameter spark.storage.memoryFraction (0.6 by default). One can choose to lower this value if cached RDDs are small.

There are two types of data serialization in Spark: Java serialization (default serializer for Spark) and Kryo serialization (faster than Java serialization). Storing RDDs in serialized form can increase the access time but reduce the memory used to cache RDDs.

Now we present the test results for Pagerank using serialized caching. Table 7.7 reports the sizes of RDDs cached in memory for each graph. The memory abailable to cache RDDs is 9.0 GB (60 percent of 15.0 GB, the total executor memory) in total by default, which is wasteful for cases using RDD serialized caching, so we can allocate more memory for task execution by lowering the parameter spark.storage.memoryFraction. In our tests, we set it to be 0.3 for cases with Java and Kryo serialized caching. Figure 7.3 shows the

scalability of Pagerank with serialized caching in Spark. Using serialized RDD storage, the time for each iteration becomes stable since Spark has enough available memory to properly perform garbage collection. Storing RDDs in serialized form also can be very helpful to our Integrated CG since it requires large RDDs to be cached in memory.

| Graph | cache without serialization | Java serialization | Kryo serialization |
|---|---|---|---|
| cit-Patents | 1,313.1 MB | 257.3 MB | 243.1 MB |
| soc-pokec-relationships | 1,936.7 MB | 333.1 MB | 300.0 MB |
| soc-LiveJournal1 | 4.2 GB | 833.9 MB | 755.9 MB |
| com-orkut.ungraph | 7.2 GB | 1247.0 MB | 1107.2 MB |

Table 7.7: The sizes of RDDs cached in memory for Pagerank in Spark

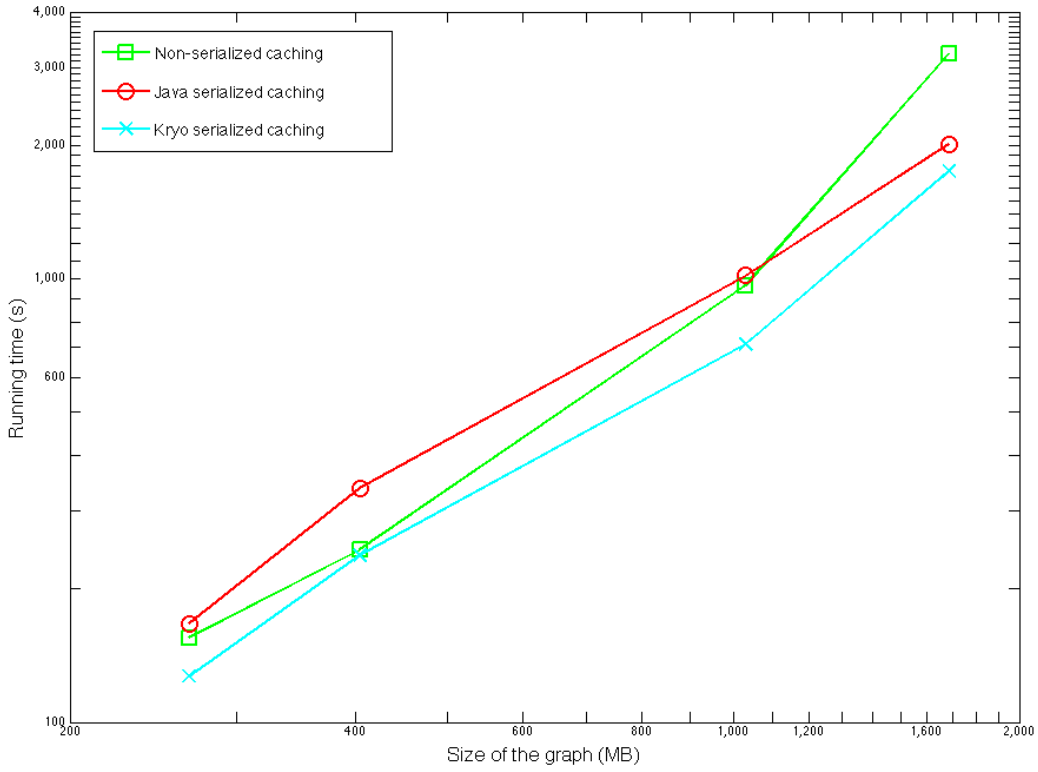



Figure 7.3: The scalability of Pagerank using serialized caching in Spark

## 7.4 Comparison with tests in Hadoop

Now we run Pagerank and CG tests in Hadoop and compare the results with Spark. Here we use the Pagerank algorithm without considering dangling nodes in the Hadoop implementation. The properties of our test data has been detailed in Table 7.2, Table 7.3 and Table 7.4. Figure 7.4 compares performance of the Pagerank implementation in Hadoop and Spark. The optimized Pagerank implementation in Spark (using Kryo serialized caching) can be at least 1.5 times faster than Hadoop. However, Hadoop out-performs Spark when there is not enough memory to fit large graphs.

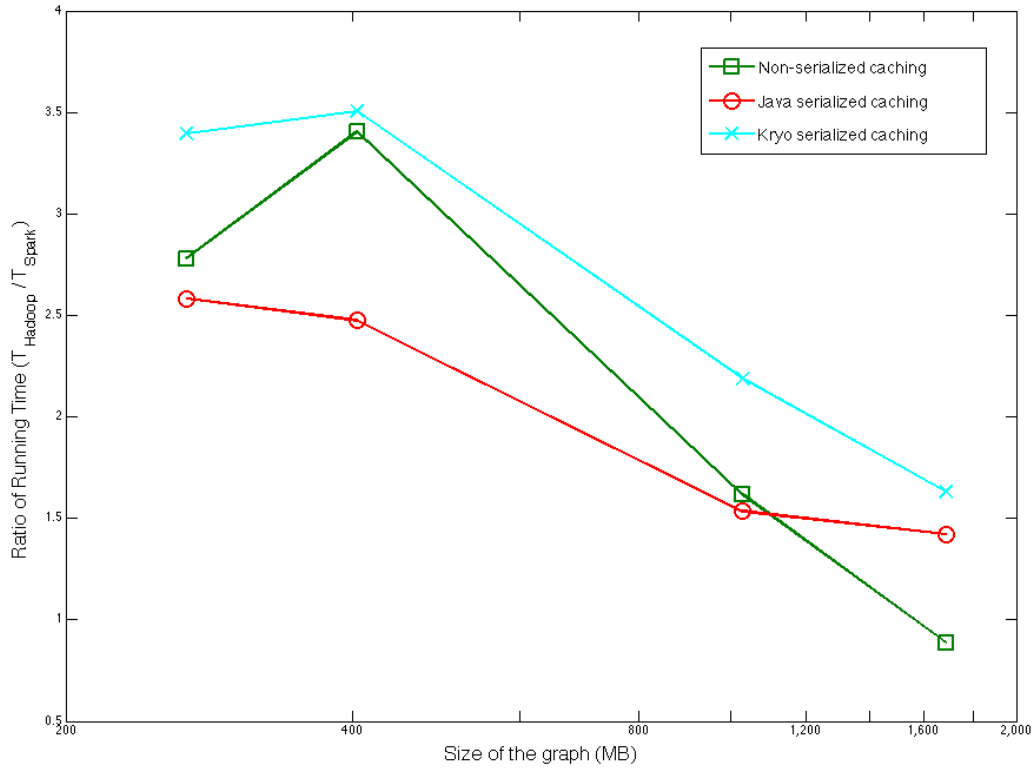



Figure 7.4: Ratio of Running time of Pagerank in Hadoop to the Running time of Pagerank in Spark

Figure 7.5 shows the ratio of running time of CG in Hadoop to the running time of CG in

Spark. When the matrix **A** is small, the overhead of Hadoop costs a lot of time therefore CG in Spark is extremely fast compared to Hadoop.

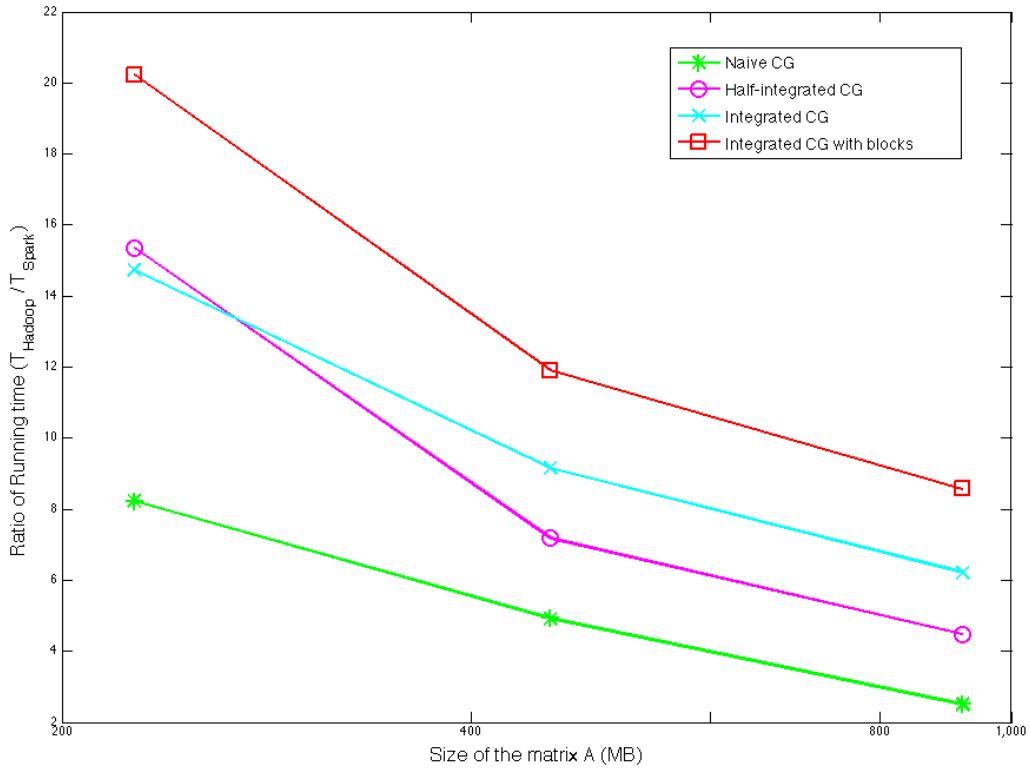



Figure 7.5: Ratio of Running time of CG in Hadoop to the Running time of CG in Spark

# Chapter 8

# Conclusion

In this thesis, we implemented the Pagerank algorithm and the CG method in Hadoop and Spark. For Pagerank with dangling pages, we developed an algorithm to compute ranks using only one MapReduce stage in each iteration. As RDDs in Spark have many special properties (like the lazy property) which are quite different from the MapReduce model in Hadoop, we presented some examples to illustrate how to use RDDs efficiently. We also illustrated how the two main RDD transformations (combineByKey and cogroup) work in Spark. The co-partition and co-location of RDDs to be co-grouped was investigated and we found it is essential to make sure that co-partitioned RDDs are co-located otherwise it takes a lot of time to fetch RDD partitions remotely. Before implementing CG in Spark, we presented how to perform matrix and vector operations using RDDs. At first we gave a naive implementation of CG and discussed its drawbacks (too much shuffling and some RDDs are not co-located). We developed a new algorithm called Integrated CG which has only one RDD transformation (cogroup) in each iteration. Our experiments showed that the Integrated CG is much faster and is scalable given enough memory to cache RDDs. When we implement the Pagerank algorithm in Spark we found that using serialized caching can substantially reduce the sizes of RDDs and therefore reduce the Spark GC time. This is also helpful to our Integrated CG method since having enough memory is vital to the success of our implementation.

The Pagerank implementation that is part of Spark only deals with the problem without dangling nodes. It would be interesting to implement Pagerank algorithm considering dangling nodes in Spark as it is more complicated than the existing Pagerank implementation. Moreover, we plan to implement other iterative algorithms like the Lanczos method which is used to find eigenvalues of a matrix. Since many iterative methods are similar to CG,

which means they use many matrix and vector operations in each iteration, our idea of integrating data can also be used in their implementations, similar to what we did in CG.

# References


[1] J. M. Ortega. Introduction to parallel and vector solutions of linear systems. Plenum Press, New York, 1988.

[2] Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry. The PageRank Citation Ranking: Bringing Order to the Web. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab, 1999.

[3] Chen Zhang, Hans De Sterck, Ashraf Aboulnaga, Haig Djambazian, and Rob Sladek. Case Study of Scientic Data Processing on a Cloud Using Hadoop. High Performance Computing Systems and Applications Lecture Notes in Computer Science Volume 5976, 2010, pp 400-415,Springer-Verlag Berlin Heidelberg 2010.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.

[7] GigaSpaces. http://www.gigaspaces.com/. [Online; accessed May 25, 2014].

[8] ELASTRA. http://www.elastra.com/. [Online; accessed May 25, 2014].

[9] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/. [Online; accessed May 25, 2014].

[10] Apache Hadoop. http://hadoop.apache.org/. [Online; accessed May 25, 2014].

[11] Cloud Foundry. http://cloudfoundry.com/. [Online; accessed May 25, 2014].

[12] Apache Software Foundation. http://www.apache.org/. [Online; accessed May 25, 2014].

[13] Apache Spark. https://spark.incubator.apache.org/. [Online; accessed May 25, 2014].

[14] AMPLab Software. https://amplab.cs.berkeley.edu/software/. [Online; accessed May 25, 2014].

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation Pages 2-2, USENIX Association Berkeley, CA, USA 2012.

[16] Scaling Facebook to 500 Million Users and Beyond. https://www.facebook.com/notes/facebook-engineering/scaling-facebook-to-500-million-users-and-beyond/409881258919. [Online; accessed May 25, 2014].

[17] Amazon Technology. http://money.howstuffworks.com/amazon1.htm. [Online; accessed May 25, 2014].

[18] Inside eBays 90PB data warehouse. http://www.itnews.com.au/News/342615,inside-ebay8217s-90pb-data-warehouse.aspx. [Online; accessed May 25, 2014].

[19] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In IEEE International Conference on Data Mining (ICDM 2009).

[20] Weizhong Zhao, Huifang Ma, Qing He. Parallel K-Means Clustering Based on MapReduce. CloudCom 2009, LNCS 5931, pp. 674679, 2009, Springer-Verlag Berlin Heidelberg 2009.

[21] Module 2: The Hadoop Distributed File System. http://developer.yahoo.com/hadoop/tutorial/module2.html. [Online; accessed March 25, 2014].

[22] Chuck Lam. Hadoop in Action. Manning Publications, 2010.

[23] Tom White. Hadoop: The Definitive Guide, 3rd Edition. O'Reilly Media, 2012.

[24] MapReduce Tutorial. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. [Online; accessed May 25, 2014].

[25] Module 4: MapReduce. http://developer.yahoo.com/hadoop/tutorial/module4.html. [Online; accessed May 25, 2014].

[26] Cluster Mode Overview. http://spark.incubator.apache.org/docs/latest/cluster-overview.html. Glossary. [Online; accessed May 25, 2014].

[27] Spark Core for Java/Scala. http://spark.apache.org/docs/latest/api/core/index.html#org.apache.spark.rdd.RDD. RDD. [Online; accessed May 25, 2014].

[28] Spark Core for Java/Scala. http://spark.apache.org/docs/latest/api/core/index.html#org.apache.spark.rdd.PairRDDFunctions. PairRDDFunctions. [Online; accessed May 25, 2014].

[29] Spark Programming Guide. http://spark.apache.org/docs/latest/scala-programming-guide.html#rdd-persistence. RDD Persistence. [Online; accessed May 25, 2014].

[30] Spark Release 0.8.0. http://spark.apache.org/releases/spark-release-0-8-0.html. Monitoring UI and Metrics. [Online; accessed May 25, 2014].

[31] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems 30: 107117.

[32] Algorithms Rank Relevant Results Higher. http://www.google.com/competition/howgooglesearchworks.html. Facts about Google and Competition. [Online; accessed May 25, 2014].

[33] HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Online; accessed May 25, 2014].

[34] Tuning Spark. http://spark.apache.org/docs/latest/tuning.html. [Online; accessed May 25, 2014].