CS777 Module 1

Topics:

- Introduction to Big Data Analytics. What is Big Data? What are the challenges?
- Introduction to Apache Hadoop and MapReduce. Apache Spark.
- Spark programming. (Python and pySpark)
- Spark - Resilient Distributed Dataset (RDDs).

Module Overview

This module provides an overview of the main challenges of Big Data storage and processing. We discuss concepts regarding scalability issues and describe how parallelization can help to concurrently process large amounts of data. We will learn about the main concept of the Map and Reduce paradigm by using examples in text and graph processing. Finally, we start learning about concrete implementations of the Map-Reduce paradigm and will learn about cluster computing systems like Apache Hadoop MapReduce and Apache Spark. In this module, we will learn the basic data transformation operations of distributed data structures of Spark—Resilient Distributed Datasets (RDDs). In the subsequent module, we will learn about Dataframes and datasets in Spark.

1.1. Learning Objectives

The main learning objectives of this module are:

- Understanding the main problems of Big Data analysis
- Understanding the main concepts of parallel and distributed computation
- Concepts of Cluster and cloud computing
- Understanding the functional computation paradigm with Map and Reduce
- Introduction to Apache *Hadoop MapReduce* cluster computing system and Hadoop File System (HDFS)
- Introduction to *Apache Spark* programming
- Spark Resilient Distributed Datasets (RDD)
- Spark Data Transformations and Action operations

1.2. What Is Big Data?

With the emergence of the World Wide Web, sensor networks and other on-line data-intensive systems, we have created a large number of systems that generate huge amount of data sets every single day.

The examples following are just a fraction of such systems that generate huge data sets:

- Web search engines get billions of search requests
- Billion of website visits
- Billions of e-mail messages are sent, received and stored everyday.
- Social networks on Webs and Mobile applications like posting and sharing photos, videos, comments, likes, creating relations (friendships).
- *Internet of Things,* sensors that generate large number of data
- Research fields like astronomy research, observation satellites that generate 200 TB of data every night.
- E-health systems, like hospital equipments and medical research
- Biotechnology research

1.2.1. 5 Vs Problem

In all of the above use cases, data is generated with huge ***V**olumes*, in different formats and ***V**ariations*, mostly at a very high speed and ***V**elocity* and has different ***V**alues* for business and comes with different quality or trustworthiness or ***V**eracity*. In big data business, all of these problems are summarized as 5 ***Vs*** keywords *(Volume, Variation, Velocity, Values and Veracity)*. This list of problems can be extend to further Vs and sub-problems like Visualizations.

The described **5 *Vs*** *(Volume, Velocity, Variation, value, Veracity)* problems require a variation of solution approaches. Imagine only the data volume problem that we face in today's data processing applications. Back in the 90's it was not possible to store for example 4 GB of data on a commodity personal computer. There was no hard disk developed to store such data size. Today we have access to huge volume of data sets and we have moved from MB, GB and TB to *PB (Petabyte), EB (Exabyte), ZB (Zettabyte)* and maybe larger.

1.3. Scalability Problem

We need to have systems that can process large data sets in a feasible time, and are scalable to process large

amounts of data volumes.

We differ between *Scaling-Out* and *Scaling-Up* in data processing systems. ”Scaling-Up means that when you need to improve the processing power of your system because you have large data sets or complex computations, you may be able to get a larger computer system that has more resources (like faster CPUs, more CPU cores, RAM or disk). It will be a single machine but with faster CPUs or more CPU cores and memory, like a Supercomputer that is a single machine.

We can understand that we have a physical limitation to get better and larger machines to run our data processing tasks on it. The prediction of Gordon Moore[1], co-founder of the Intel company, in 1965 about the progress of capabilities computer systems that the number of components in integrated circuits would double each year. This is known in computer science as Moore’s Law[2].

During the past years of 1978 to 2002, we observed exponential growth but in the past most recent years we see that the curve of exponential growth is changing its slope to a horizontal line.

Because of the physical limitations of developing faster CPUs (higher performance than 4.5GHz), new desktop and server computer systems are developed that include many processing cores to execute parallel computation tasks.

It is also from the business perspective much cheaper to buy many *commodity servers* than a supercom- puter. A commodity server is an average cost computer server. The cost of a supercomputer is equal to many many commodity servers.

The main computation challenge is to use *parallelism* to spread computational data processing tasks among many commodity servers rather than requiring a supercomputer to process the data. When we have additional data sets to process, we can simply add additional machines (resources like more CPU cores, RAM, hard disk) to process the data. Our system can process the whole data in the same specific time without any issues (e.g., crashes), then we would say that the system can **scale out**.

---

1. Gordon Moore Wiki

2. Gordon E. Moore (1995). ”Lithography and the future of Moore’s law” (PDF). SPIE. Retrieved May 15, 2020.

1.4. Parallel Data Processing

Parallelization is the key concept to achieve high-scalability. To process large amounts of data, we separate the

computation tasks into parts and run them on many computer machines (commodity servers) each including multiple CPU/GPU cores. We can say that we use the divide and conquer principle. We can achieve high parallelization by executing multiple threads and processes on cluster of machines.

Multithreading is the functionality of a multi-core processor to execute multiple threads concurrently in parallel within a single process. As threads are running within a single process they share resources like memory caches and buffers of a single or multiple cores of a processor.

1.4.1. Multiprocessing

Multiple processes can be used to run computations on a cluster of machines. Each process can execute multiple threads. Multiple processes can access the memory of a cluster of machines, send and receive data over network connections.

Figure 1.1 illustrates a cluster of computing machines that can store data in a parallelized setting and have their CPU and RAM (Random Access Memory). The machines can process data and can also store a large volume of data. Consider that each of these machines is a commodity computer server (for example each has 20 CPU cores, 64 GB RAM, and two hard disk drive of 4TB). A cluster of computer servers like shown in Figure 1.1 can have in total enough storage capacity and a large CPU computation power (for example, number of machines n = 4, means 4X20 CPU cores = 80 CPU cores).

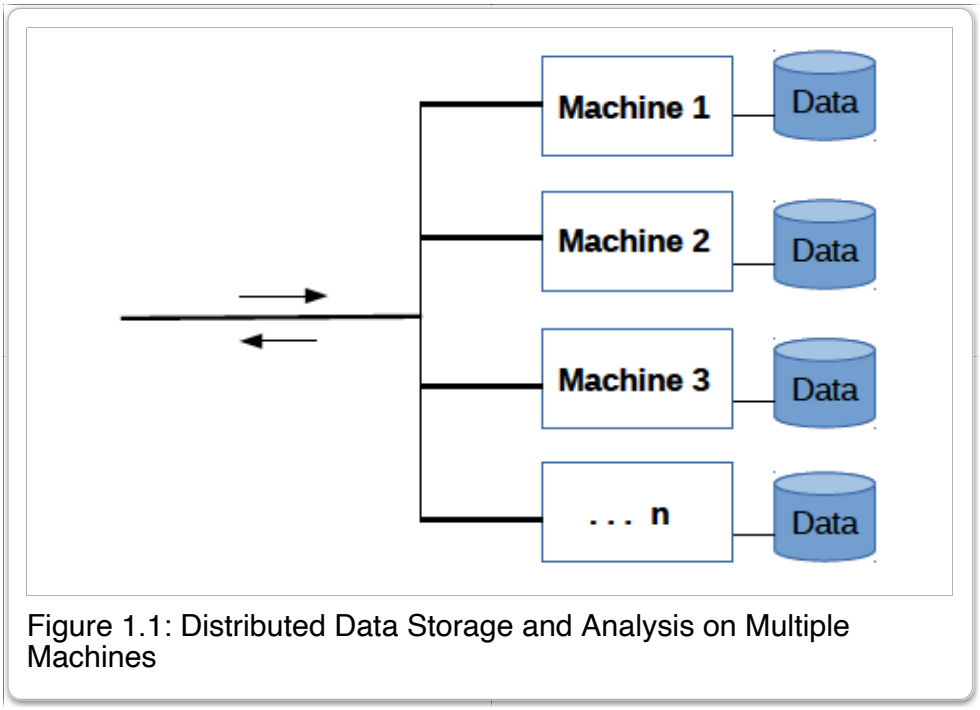

Figure 1.1: Distributed Data Storage and Analysis on Multiple Machines

Worker servers can store data and process it. Worker nodes that only store data sets are named data nodes and worker servers that store and compute them are normal computation workers. In this setup, worker machines have no knowledge about each other and do not know which other workers are participating in a cluster, worker machines only know about the master. A master server has an organizational role and keeps knowledge about all

participating worker machines. A Master machine orchestrates the communication and team-work of worker nodes so that they can run a complete data processing tasks by passing data sets and computation definitions (Defined functions by using functional programming) around the clusters.

Cluster computing is build using a centralized coordination system based on orchestration of worker nodes and worker nodes are not doing a choreography. The master node is not a single point of failure because the master node can have one or many replications. In case of any failure in a master node, one of the master replication nodes can take the organization/orchestration task over and continue the data processing task.

It is also from the business perspective much cheaper to buy many *commodity servers* than a supercomputer. A commodity server is an average cost computer server. The cost of a supercomputer is equal to many many commodity servers.

The main computation challenge is to use *parallelism* to spread computational data processing tasks among many commodity servers rather than requiring a supercomputer to process the data. When we have additional data sets to process, we can simply add additional machines (resources like more CPU cores, RAM, hard disk) to process the data. Our system can process the whole data in the same specific time without any issues (e.g., crashes), then we would say that the system can **scale out**.

To summarize components of a cluster computing system we can describe them as the following 3 points:

- **Workers (aka Slaves)** are machines that are used to store and/or run computations.
- **Master node (aka Supervisor)** is a logical cluster management role that manages the whole cluster.
- Workers move data batches (chunks of a data set) within the cluster and/or from/to outside of private cluster network.

As shown in Figure 1.2 a gateway or switch connects the distributed worker servers so that they can communicate with each other and send data messages back and forth. Over the same network an external client or server (another computer) can connect to the workers or the master server.

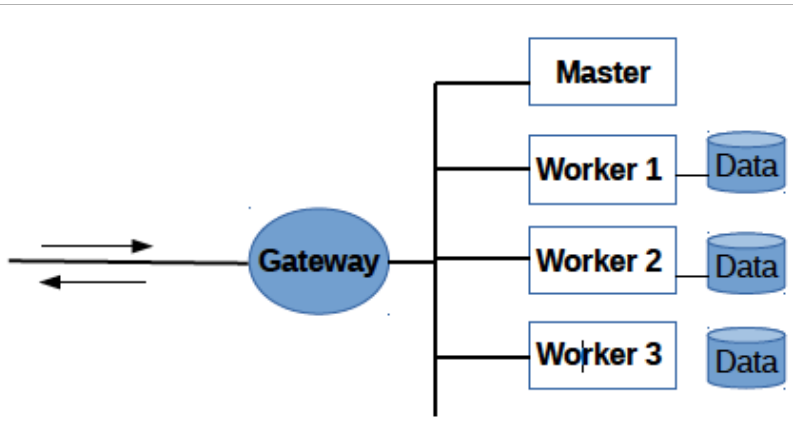

Figure 1.2: A network switch or gateway connects distributed servers and allows them to communicate with each other and

with external servers/clients over Internet network.

Agreement on **"Shared Nothing. "** Storage and analysis of huge data sets on a number of commodity machines is based on a so called "shared nothing agreement", i.e., the machines participating in a cluster do not shared storage attached to each of them, their only link exist via a network (e.g, High throughput LAN, mostly Gigabit Ethernet between servers and network switches and fiber optics switches). These machines can send data batches to each other and get back the result of computation. *"Shared nothing"* refers to not sharing of RAM, CPU or storage disks. In this case, our **Loosely Coupled Design** allow us to build flexible applications that can push data batches, are more reliable and reliable because they have reduced the interdependencies between components. Any component in this design can be replaced in failure cases and system can continue to process data. There is no single point of failure.

Figure 1.3 illustrates a firewall that protects the cluster from outside public Internet. Only specific network ports are open for network communication, mostly only TCP (Transmission Control Protocol) port 22 for Secure Shell (SSH) and any other ports are closed.

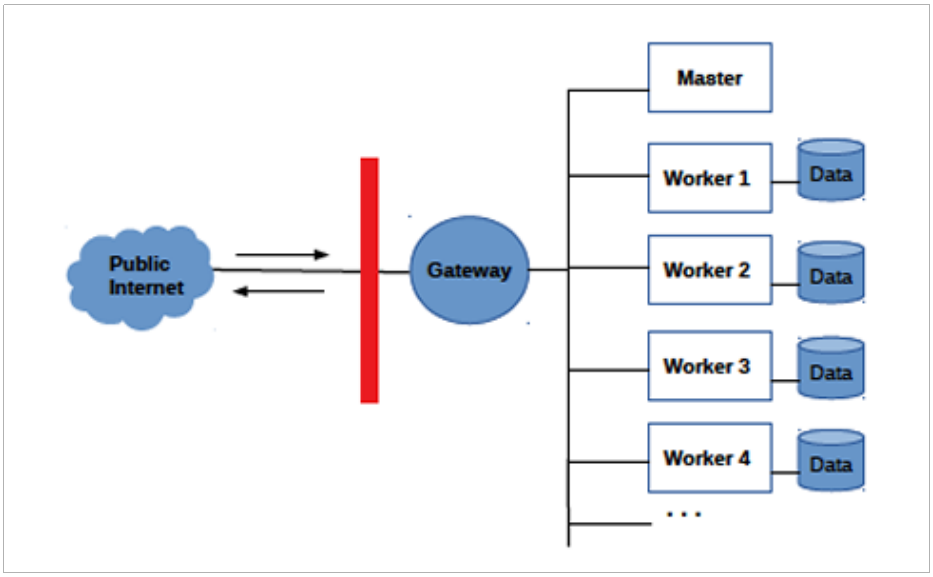

Figure 1.3: A Cluster of Machines with a Network Firewall

Figure 1.4 shows an SSH tunnel that is mostly used to access the cluster from the public Internet. The SSH tunnel can be used to do almost everything with the cluster like pushing data from client side, triggering the computations and downloading the results.

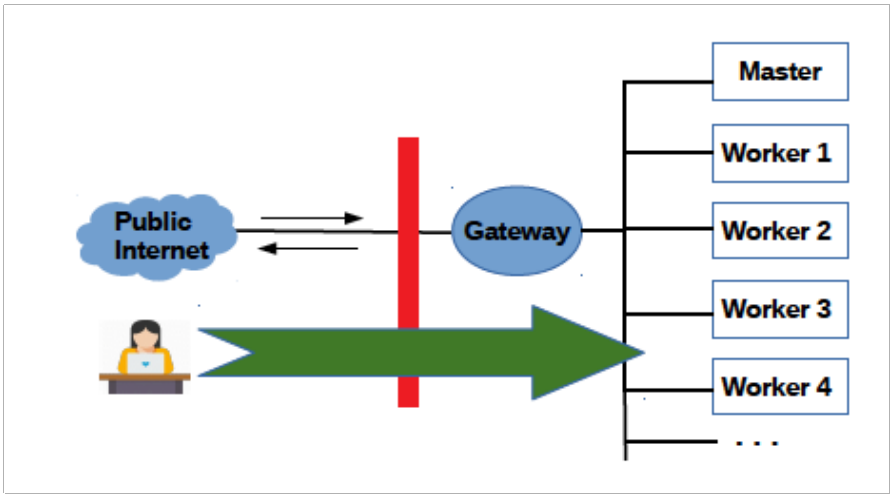

Figure 1.4: Building an SSH Tunnel Connection to access a Private Network to access Servers

In Big Data Analysis, we separate the large computation task to subtasks. Each subtask includes a function that defines which computation should be done, and a data chunk of the large data set (a data batch). Such subtasks are then distributed around the cluster and each computer does its computation give two pieces of inputs:

1. a function (what to do with the data set)
2. a data batch

After the machines finish their computation, a master node will provide the machines with next round of computation subtasks until the whole large computation is done.

1.5. Functional Programming

Functional programming is a specific type of programming languages that is designed to consider computations as set of self-contained mathematical functions. A complex computation is build by combining multiple computations with each other. Many programming languages are designed based on functional programming concepts like Languages like Haskell, Lisp[3], Erlang, Scheme, Logo, D, R, ...

1.5.1. Lamda Functions

Functional programming language is modeled based on concept of mathematical functions. Lambda functions are defined based on the definition of **Lambda Calculus**.

Sum of two numbers can be specific as following lambda: $\lambda x, y. +xy$

Also as following lambda expression: $\lambda x.\ (\lambda y.\ +xy)$

Multiplication operation can be specified as: $\lambda x.\ *xx$

In the following code[4] you can see some examples of lambda expression in Lisp programming language.

Listing 1.1: Examples of Lisp Programing Language Lambda Functions

```
% Some Examples of Lisp Lambda functions

(defun sqr(x) (* x x))

(map 'list #'sqr (list 4 5 6))
=> (16 25 36)

(reduce #'+ '(16 25 36))
=> 77

(map 'list #' length (list "This" "is" "a" "test"))
=> (4 2 1 4)
```

Lambda Expression in Python programming language is demonstrated in the below example.

Listing 1.2: Example of Python Lambda Functions

```
# Lambda Function in python
def sumOfSquares(x):
    return sum([i**2 for i in x])

# We can define the above function using a lambda
sumOfSquares = lambda x: sum([i**2 for i in x])

print(sumOfSquares([2, 4, 6]))
> 56
# 36+16+4
```

Special Characteristics of Functional Programming.

- **Data generation from raw data.** Functions can be used to generate new data sets from the input data set. In case of any interruptions (like network or disk failure) this process can be restarted to generate new data from the raw input data set or from the previous intermediate results.
- **Raw data remains in place.** Functions generate new data sets from the old or raw input data set which remains in its location unchanged. The process have only a read only access to the input data and each process reads in parallel a portion of the input data. This concept of keeping the input data inplace is important in cluster computing because we can restart the data processing in case of any failure.
- **Building data processing pipelines.** Data sets are passed through several functions and a sequence of executing operations on data builds a complex computation. Data processing pipelines can be executed in parallel depending on the design of processing sequences. The designer should avoid having any

bottlenecks to be achieve maximum processing parallelization to use the complete computation power of cluster machines.

Example 1. Sum a list of numbers.

Adding up a list of numbers can be computed in a parallelized setting on multiple threads or multiple processes. The solution is to divide the list of numbers to sub-parts or smaller lists, then add the numbers in each of these lists, and finally sum the results of pervious sum tasks.

In List programming language this can be implemented using a reduce function.

Listing 1.3: Reduce Operation in Lisp

```
% (reduce + (16 25 36 2 43 92 12))
```

---

3. List is one of the very old programming languages. It is originally specified back in 1958. Because of its nice properties in functional programing. Lisp survived a very long time—it was developed almost 60 years.

4. You can install CLISP or Clojure on Linux machines. Try it!

1.6. Map Reduce Paradigm

Map and Reduce paradigm has its roots in functional programming concepts. Developers Programmers found out that parallel data processing can be achieved by splitting the computation into two major steps. The Map phase includes converting the input data by applying a function to it and producing intermediate output data. We can parallelize the map data processing because each process has its batch of input data and outputs to distribute file storage to its output file, or it outputs to main memory RAM. The Reduce phase then takes the intermediate data (output of map phase) and reduce it to a final output data. Reduce phase processes the data sets in a parallel setting because each process has its batch of data to work on and function that defines Reduce computation.

Example 2. Word Count of a Text Document with Map Reduce Pattern.

The goal of this Map process is to count how many times each word appears in a large text file/corpus. Figure 1.5 depicts the Map and Reduce processes on an example text corpus. The text files can be read from a file, folder or a database.

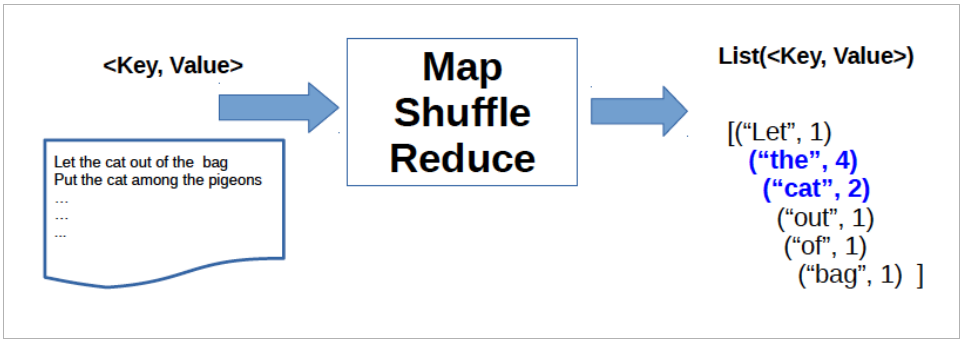

Figure 1.5: Map, Shuffle and Reduce operations in Word Count

Figure 1.6 illustrates the Map phase. Each Map process reads a single line from the file in parallel and maps it to a list of tuples of form *List*(< *word*, 1 >) which is an intermediate output of Map process. In case of any duplicated words in a sentence, the Map process converts them into separate tuples. The number one in each of these tuples is the count of appearances of this specific word in the text line. This number in the tuple is also some kind of placeholder for summing up the counts of words in the next Reduce phase.

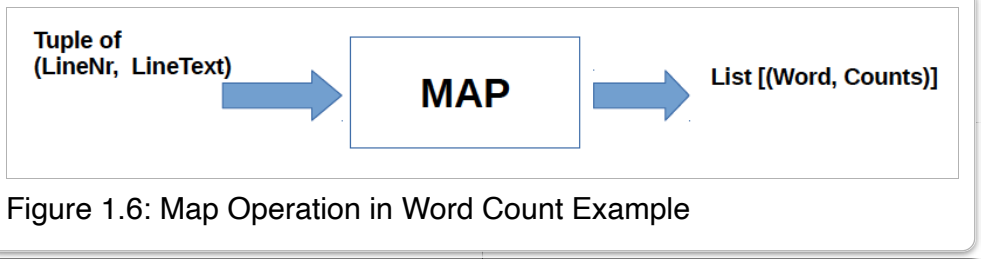

Figure 1.6: Map Operation in Word Count Example

Before running the Reduce task, the system needs to move the data to different machines on the network and organized them in a form that Tuples with the same word are going to the same machines to reduce and generate the total counts. Words in tuples are the keys for the Shuffle phase, which Figure 1.7 illustrates.

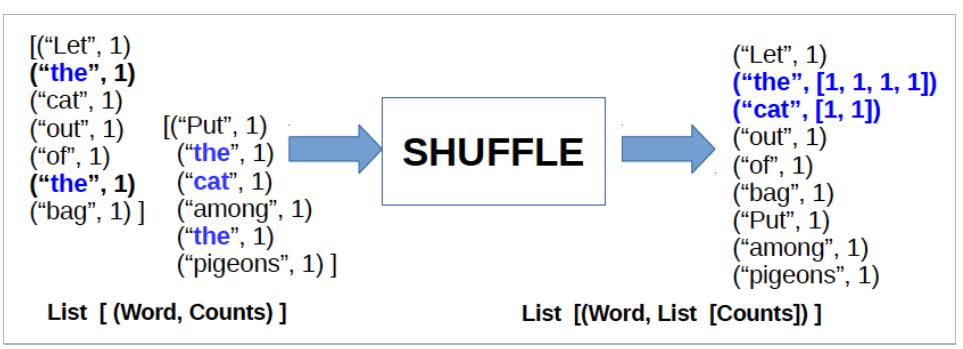

Figure 1.7: Data Shuffle in Word Count Example

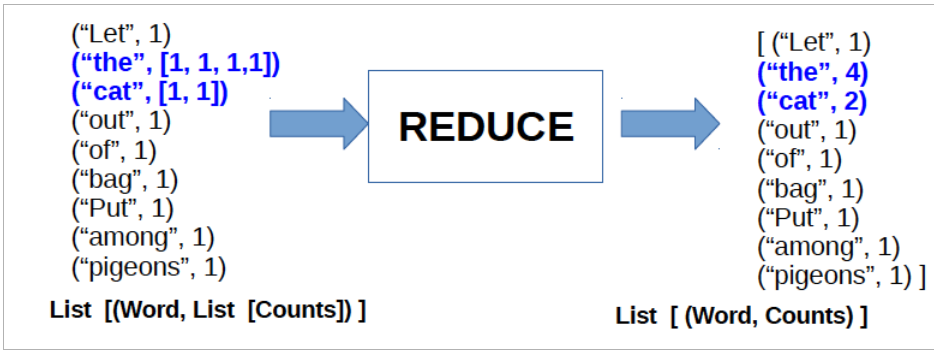

Figure 1.8: Reduce Operation in Word Count Example

Figure 1.9 illustrates the complete process of Map and Reduce on a cluster of computers. The input data source is distributed system (mostly a distributed file system) that can deliver splits of input data source for each of data processors. The system distributes the Map processes on multiple worker machines for execution. Then, the system will organize the intermediate output result and shuffle the data to the Reduce processes distributed on multiple machines. The output data holder is also a distributed system (mostly a distributed file system) and can store chucks of data as an output file of each of Reducer processes.

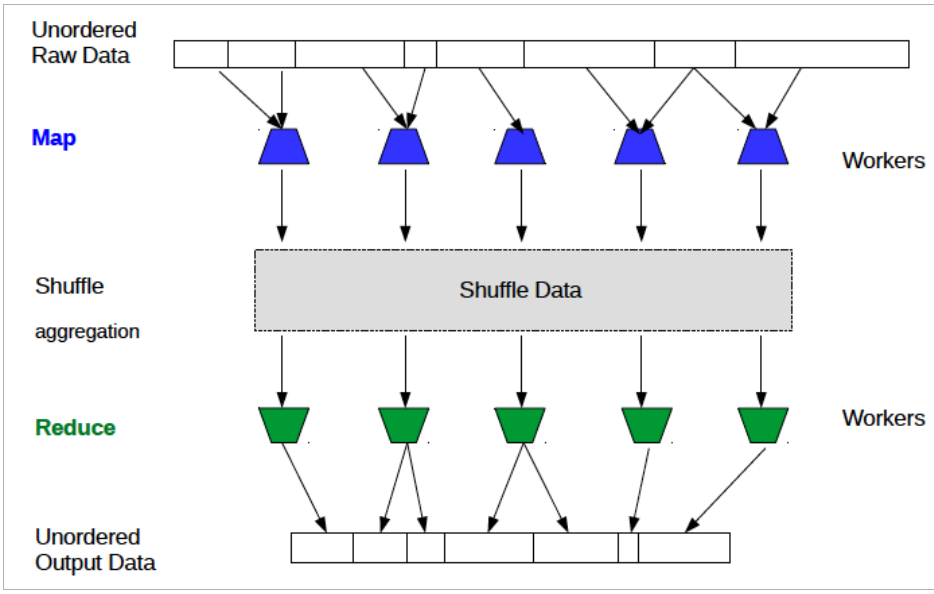

Figure 1.9: Map and Reduce Process on a Cluster Computing System

The Map Reduce is a general and popular concept in Big Data processing. In 2004, Dean et al. [1] (employed at Google) published a paper about using Map-Reduce and showed how Map-Reduce can be used in a distributed setting to process large amounts of data sets. Apache Hadoop MapReduce[5] is an open source Java implementation of Map Reduce concepts. Apache is a software foundation, an organization that manages many open source software projects.

Example 3. Graph Processing using Map Reduce Paradigm.

Consider a weighted graph like shown in Figure 1.10. Nodes have some IDs (like numbers as Node ID) and each edge between the nodes has some weights. The goal is to calculate the sum of weights for each node in a given graph, i.e., outgoing from a given node what is the weight sum of the connected edges to the node.

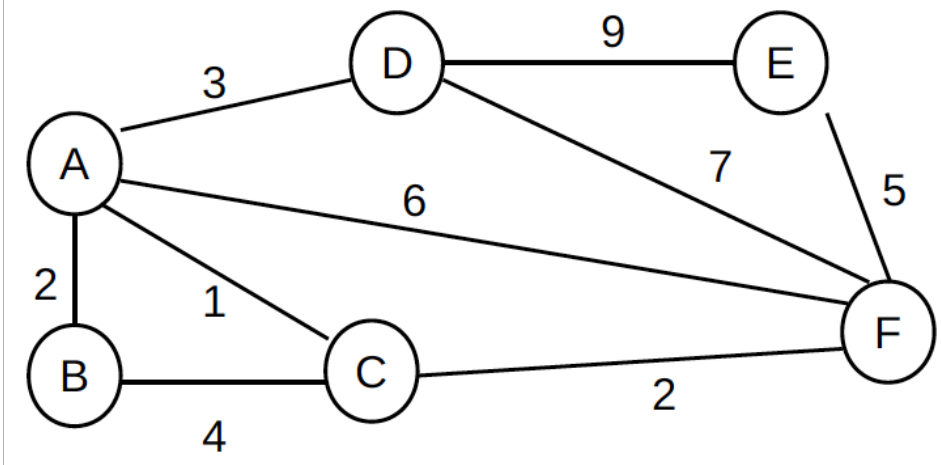

Figure 1.10: Graph Processing using Map Reduce Paradigm

In such cases, the first step is to understand how you would represent this data in an input data format. For example, you can consider the following as your data input format of the graph.

Input data as (Graph Data serialization in text form):

[((A, D), 3) ((A, B), 2) ((A, C), 1) ((A, F), 6) . . .]

The goal is to calculate an output in Tuple form, (NodeID, SumOfWeights).

Output should be like:

[(A, 12) (B, 6) (D, 19) . . .]

The Map Reduces steps are shown in Listing 1.4.

Listing 1.4: Graph Data Processing with Map Reduce

```
Input Data:
[((A, D), 3) ((A, B), 2) ((A, C), 1) ((A, F), 6)...]

Map by node name:
(A, 3) (D, 3) (A, 2) (B, 2) (A, 1) (C, 1) (A, 6) (F, 6)

Shuffle - Aggregate by Key (Key is the node ID)
[(A, [3, 2, 1, 6, ...])
 (B, [2, ...])
 (C, [1, ...]) (D, [3, ...])]

Reduce - Sum by key
```

```
[(A, 12) (B, 6) (D, 19) ...]
```

Example 4. Linear Algebra Computation using Map Reduce Paradigm.

In this example we describe how to use Map Reduce paradigm to compute Matrix Transpose of given Matterix like shown in Figure 1.11.

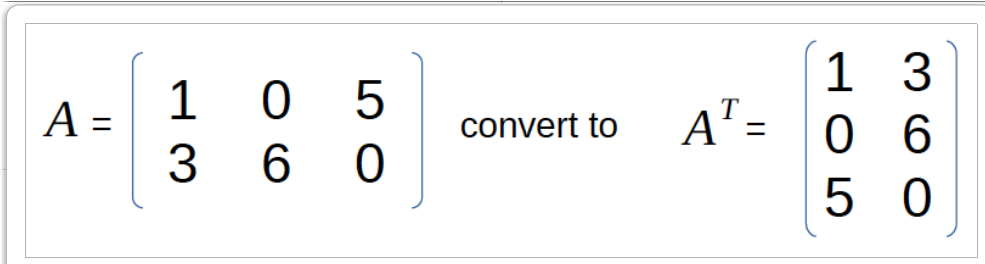

Figure 1.11: Matrix Transpose Operation using Map Reduce

The data serialization in text data format can be the following representation. We have ID keys for the rows and columns of the given Matrix and we store only none zero values - a Sparse matrix storage.

Input Data:

(0 , [(0, 1), (2, 5)])
(1 , [(0, 3), (1, 6)])

The desired output of this data is given in the following.

Output:

(0, [(0,1), (1, 3)])
(1, [(1, 6)])
(2, [(0, 5)])

The goal is to use Map and Reduce operations to go from the above Input data and transpose the matrix into the output data. Listing 1.5 describes how these operations can be implemented considering ID keys for rows and columns of the matrix. We also provide details of the shuffle phase but normally this happens in implemented systems under the hood and there is no need for additional coding for the shuffle phase (it will be done internally by the Big Data Processing systems).

Listing 1.5: Matrix Transpose Computation using Map Reduce

```
Input Data:
(row-0, [(col-0, 1), (col-2, 5)])
(row-1, [(col-0, 3), (col-1, 6)])

Map to:
(col-0, (row-0, 1)) (col-2, (row-0, 5))
(col-0, (row-1, 3)) (col-1, (row-1, 6)
```

```
Shuffle to:
(col-0, [(row-0, 1), (row-1, 3)])
(col-1, [(row-1, 6)]
(col-2, [(row-0, 5)]

Reduce to:
(col-0, [(row-0, 1), (row-1, 3)])
(col-1, [(row-1, 6)]
(col-2, [(row-0, 5)]
```

---

5. Apache Hadoop

1.7. Apache Hadoop MapReduce

Apache Hadoop includes multiple components that work together to process Big Data sets. It includes a distributed file system named Hadoop Distributed File System (HDFS) to store large data files on multiple machines. A cluster management system named Hadoop Yarn to mange large cluster of computers and Hadoop MapReduce.

Apache Hadoop project includes following sub-projects:

- **Hadoop Common:** Common Hadoop utilities that are used in all other Hadoop components.
- **Hadoop MapReduce:** An implementation of Map Reduce Paradigm in Java.
- **Hadoop Distributed File System (HDFS):** A distributed file system providing high-throughput access to data.
- **Hadoop YARN:** Cluster resource management and job scheduling

1.7.1. Hadoop Distributed File System (HDFS)

Data files are too large to be stored on a set of hard disks managed by a single machine. Even if we can get a server with multiple hard disks and each of them in largest feasible size. For example as of April 2020, a commodity server might have 10 hard disks each 20TB large which would make in total 200 TB of storage space.[6] This would be the maximum capacity with no data replication, i.e., in case of disk failure data files would be lost.

Hadoop Distributed File System (HDFS)[7] is a distributed system that can store huge files on a set of commodity servers with regular hard disks. HDFS can easily scale out to many servers to store huge Files. HDFS is designed to be deployed on any ”low-cost” hardware HDFS is like existing distributed file systems with the difference that is highly fault-tolerance and provides high throughput access to files.

Figure 1.12 illustrates the main design principles of HDFS[8] write and read processes. Multiple servers named data nodes store chunks of a file multiple times so that each chunk is replicated to multiple servers. This replication has two main goals, 1. fault tolerance of file storage, i.e., if one disk fails data chucks are stored on multiple files, 2. data files can be read with huge network throughput because data chucks are read from multiple servers. In this way, each Map-Reduce process can have its data batch to execute concurrently without any bottlenecks (not waiting for file reads from disk).

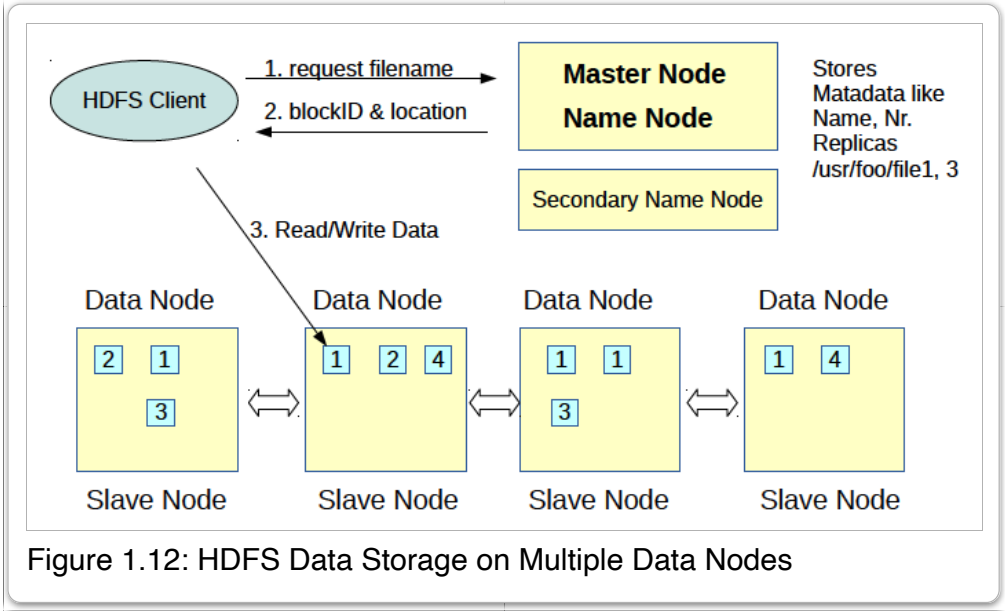

Figure 1.12: HDFS Data Storage on Multiple Data Nodes

Working with HDFS is very similar to any other File System (FS). HDFS includes a set of shell-like commands[9] following the same naming conventions of Unix/Linux shell commands. Here we describe briefly some of the most frequent used commands and refer the reader to the HDFS main documentation for further readings.

After installation of Hadoop client software users can use the HDFS commands. The command ”hadoop” will be available to use on the command line and by using keyword ”fs”, it indicates that we want to use Hadoop File System. Listing 1.6 is the command to create a file directory on HDFS. Address space on HDFS can be addressed by using the address space of ”hdfs://” as a global address, and if you are running the command on the main master node, then you can address the main root folder just by using a single slash.

Listing 1.6: HDFS mkdir command

```
% Make Directory
# hadoop fs -mkdir [-p] <paths>

% Examples:
# hadoop fs -mkdir /user/hadoop/dir1/user/hadoop/dir2
# hadoop fs -mkdir hdfs://nn1.example.com/user/hadoop/dir
```

The command ”put” loads files from the local machine or any other server to the HDFS file system. Listing 1.7 is describing the syntax of the ’put’ command. This command will internally split the file into chunks and replace them

on the data nodes regarding the configuration of your cluster.

Listing 1.7: HDFS mkdir command

```
% Put files to HDFS
# hadoop fs -put <localsrc> ... <dst>

% Examples:
# hadoop fs -put localfile /user/hadoop/hadoopfile
# hadoop fs -put localfile hdfs://nn.example.com/hadoop/hadoopfile
```

If you want to remove an entire folder with all the files and sub-folders in it, you can use the command ”rmdir” as described in Listing 1.8.

Listing 1.8: HDFS mkdir command

```
% Remove Directory
# hadoop fs -rmdir  [--ignore-fail-on-non-empty] URI [URI ...]

% Examples:
# hadoop fs -rmdir /user/hadoop/emptydir

% Remove directories recursively
# hadoop fs -rmr  [-skipTrash] URI [URI ...]
```

Listing 1.9 shows the move ”mv” command to move a file or folder from one location to another location on HDFS.

Listing 1.9: HDFS mkdir command

```
% Move files
# hadoop fs -mv URI [URI ...] <dest>

% Examples:
# hadoop fs -mv /user/hadoop/file1/user/hadoop/file2

% Checking Disk usage
# hadoop fs -du
# hadoop fs -df
```

Further HDFS commands[10] are: appendToFile, cat, checksum, chgrp, chmod, chown, copyFromLocal, copyToLocal, count, cp, createSnapshot, deleteSnapshot, df, du, dus, expunge, find, get, getfacl, getfattr, getmerge, help, ls, lsr, mkdir, moveFromLocal, moveToLocal, mv, put, renameSnapshot, rm, rmdir, rmr, setfacl, setfattr, setrep, stat, tail, test, text, touchz, truncate, usage

Note: HDFS is widely used to store very large data files on servers, especially in private cloud systems. HDFS is one of the important Big Data system that my exist in the next 20 years because of its wide usage and its ability to fail tolerance storage of large data set files.

1.7.2. Hadoop MapReduce

Hadoop MapReduce is an implementation of Map-Reduce paradigm in Java programing language. To implement a computation job you need to implement three main components:

1. Implement a Mapper class that extends the class Mapper[11] and specifies how to do the mapping and what is the output of Map.
2. Implement a Reduce class that extends the class Reducer[12] and specifies how to reduce and what is the final output of the reduce process.
3. A driver that defines which the Mapper and which Reducer should work together to execute a MapReuce job, where is the input data source and where should be the output data stored.

The listings 1.10, 1.11, and 1.12 are a complete Word Count implementation example in Hadoop MapReduce. You can find on the Hadoop MapReduce Tutorial on the main documentation website of Hadoop[13] It is also useful for you to take a look at the Hadoop MapReduce Java API[14].

Example 5. Word Count Example[15] Implementation in Hadoop[16] Mapper

The first task is to write Mapper and Reducer Classes. The listing 1.10 is the implementation of Word Count Mapper. The class Mapper would have the following generic type parameter:

ClassMapper < KEYIN, VALUEIN, KEYOUT, VALUEOUT >

KEYIN is the input key value. VALUEIN is the actual input value content. KEYOUT is the output key and VALUEOUT is the actual output value. Implementation of a concrete Mapper Class needs to specify the types of (Key, Value), i.e., are they Integer, Text, ... or any Objects.

Listing 1.10: Hadoop Mapper Implementation - Hadoop Word Count Example

```
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class MyWordCountReducer extends Mapper<Object, Text, Text, IntWritable>{
  // Create a counter and initialize with 1
  private final IntWritable counter = new IntWritable(1);

  // Create a hadoop text object to store words
  private Text word = new Text();

  // Implement a map method.
  public void map(Object key, Text value, Context context) throws IOException,
      Interrupted Exception {
    // Create tokens from text and have an iterator
    StringTokenizer itr = new StringTokenizer(value.toString());

    // Iterate over the list of tokens and write word and counts into the
      Context.
```

```
    // Context would be then the output of the Map Task.
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, counter);
    }
  }
}
```

Hadoop Reducer Implementation

The Reducer defines how to reduce the output results from the Mapper task. Listing 1.11 is an implementation of Reducer for the Word Count Example.

Listing 1.11: Matrix Transpose Computation using Map Reduce

```
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class MyWordCountReducer extends Reducer <Text, IntWritable, Text,
    IntWritable> {

public void reduce(Text text, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
          // set the initial sum to zero
          int sum = 0;

          // Then iterate over the counts and sum them up.
          // Values are from type IntWritable.
          for (IntWritable value: values) {
              sum += value.get();
          }
          // Then write the output of this reducer task.
          context.write(text, new IntWritable(sum));
        }
}
```

Driver Implementation

Driver defines which Mapper and Reducer should work together and what are their input and output files/folders.

Listing 1.12: Driver Implementation in Hadoop MapReduice - Word Count Example

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextOutputFormat;
```

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    // get an instance of the Configuration class
    Configuration conf = new Configuration class
    // create a new JOB and set its config object
    Job job = new Job(conf, "WordCountExample");

    // define what are the types of the output Key and Value
    // In this case we have Text and Integer
    job.setOutputKeyClass(Text.class;
    job.setOutputValueClass(IntWritable.class);

    // Set What is the Mapper implementation for this Job
    // Set what is the Reducer implementation for this Job
    job.setMapperClass(MyWordCountMapper.class);
    job.setReducerClass(MyWordCounReducer.class);

    // Define what is the input data type and what is the output data type
    // We read from a text file and write the results to multiple output text
        files
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    // Define which file or whole folder to proces and which folder to use to
        store the outputs.
    // args[0] is the first argument when we run the application and it defines
        here the text input files.
    // args[1] is the second argument when we run the application and it defines
        the output folder.
    FileInputFormat.addInputPath(job, new Path(args[0]))
    FileOutputFormat.setOutputPath(job, new Path(args[1]))

    // run it and wait until completion
    job.waitForCompletion(true);
  }  // Execute the job
}
```

How to run this example?

To execute the above example, you can run the command described in Listing 1.13. You would need to compile the program and create a Java Jar file that includes all the class binaries plus all the library dependencies.

Listing 1.13: Execute the Hadoop MapReduice - Word Count Example

```
# bin/hadoop-*-examples.jar wordount [-m <#maps>][-r <#reducers>]
  <in-ir> <out-dir>
```

Note: You can combine multiple Mapper and Reducer to generate a data processing pipeline. It must

not be always a Mapper followed by a Reducer, but mostly we write it in this form because we can define computations based on a sequence of multiple Mappers and Reducers.

1.7.3. Disadvantages of Hadoop MapReduce

- Intermediate task (like map task) has to finish completely until the subsequent reducer tasks can start. This would hit the overall performance of Hadoop MapReducer data processing tasks because each job has to wait until previous Job terminates.
- Reuse of intermediate results are possible. It is possible to reuse the intermediate results multiple times, but the intermediate data has to be stored on some large-scale storage system. It is not possible to cache the intermediate results into the main memory of cluster computers.
- API is only in Java programing language.
- Writing code in Hadoop MapReduce is very complex and needs lots of code lines for simple computation like Word Count for example.

---

6. In RAID Level 0 - RAID (Redundant Array of Independent Disks) Configurations

7. Hadoop Distributed File System (HDFS), retrieved May 2020

8. HDFS Design and HDFS User Guide

9. File System (FS) Shell-Like Commands Overview Note: you do not need to memorize all of the commands. You can always look for the commands in online documentations, e.g., using a web search engines.

10. Read more about HDFS Commands

11. Class Mapper

12. Class Reducer

13. Map Reduce Tutorial

14. Apache Hadoop Main

15. Note: Do not worry if you do not understand this code completely. We provided this example here to illustrate how the Hadoop Code looks like. Later in this chapter, we will learn about Apache Spark which has much smaller code in python for the same task of Word Count.

16. The complete implementation code of Word Count in Hadoop MapReduce

1.8. Apache Spark

As we described in the previous section, Hadoop MapReduce implementation has some important issues like implementing code in Hadoop is not so easy for applied data scientists, or that intermediate data have to be stored and can not be kept in main memory. Developers have realized these problems over the past 10-15 years and have developed a bunch of different systems to solve some of the Hadoop issues, like Apache Hive[17] with major goal to make implementation in Hadoop MapReduce easier and provide a SQL-like programing interface to work with data on top of Hadoop.

Apache Spark[18] is another Big Data system that is developed based on Hadoop eco-system, and provides a low latency distributed parallel computing platform for Big Data computation. Spark has many advantages compared to perviously developed big data analytic system like *Hadoop MapReduce*.[19]

Some of the **Spark advantages** are the following improvements:

- Reuse of intermediate results is possible and it is possible to cache intermediate results into main memory of cluster computers. This would improve the overall data processing performance. In some specific computations it is in average faster than Hadoop (10 times faster or sometimes 50 times). Shi et al. [2] have done an experimental evaluations of Hadoop and Spark big data systems for different computation patterns.
- Rich API for Scala, Python and Java (also R in RSpark).
- Java and Scala API of Spark performs faster because of static data types in these programing languages.
- It is possible to use Spark in combination with Hadoop HDFS to store large files and process them in parallel.

1.8.1. Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs) are the primary abstraction in Spark. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

RDDs have the following important properties:

- RDDs enable efficient reuse of intermediate data.
- RDDs are fault tolerance because of the Spark distributed architecture.
- RDDs are parallel data structures and data chucks are stored on multiple machines.

- RDDs allow user-specified data partitioning.
- RDDs allow users to cache intermediate results in memory so that they can be reused if needed.

Spark can create RDDs from any files.[20] For now, assume that you have access to a folder with files inside it. Later we see that these files can be from different sources and types.

On Spark RDDs we can run two types of data processing operations 1. **Transformation operation** and 2. **Action operation**. A **Transformation** operation transforms an RDD into another RDD by applying a function into rows of it. A data processing pipeline is built by combining a lot of different transformation operations with each other to process the data set. An **Action operation** converts an RDD back into not-parallelized value and pulls the data from the distributed cloud into a single machine. RDD is a collection of data elements that are partitioned across the participating cluster nodes. It is important to understand that data is distributed across the cluster in data partitions and stored on multiple machines.

You can find the complete documentation of Spark RDD operations on Spark API documentation website.[21]

As shown in Figure 1.13, we can import the large input data sets from an input source and create an RDD from it. We can then transform many times RDDs to each other by using transformation operation and data will still be processed in a large cluster so that we can work with a very large data set. An action operation pulls the data set from the cloud and moves it back to a single machine. We run the action operations when we know that the final result will not be very large and can be hold in a single machine main memory. Note that the action operations cover the object types from Spark distributed objects to the main programing language object type. As we are using PySpark, action operations will move our code to go back to regular python code, for example we will get a list of objects or single integer number.

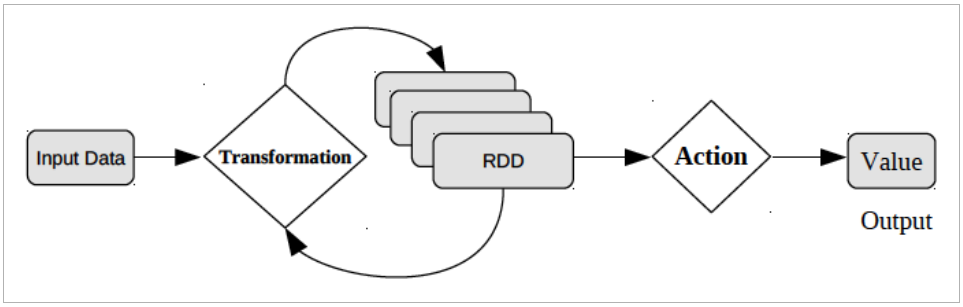

Figure 1.13: Spark RDD Transformation and Action Operations

Listing 1.14 is the instantiations of SparkContext object. All of the operations of spark RDD are functions implemented in this class.

Listing 1.14: Spark Context

```
# import the required Spark library
<<< from pyspark import SparkContext
```

```
# instantiate an instance of spark Context.
<<< sc = SparkContext()

# You can then use sc.
```

1.8.2. RDD Transformation Operations

Spark provides a set of different RDD transformation operation. In the following we describe the most important ones and refer you for the complete operation lists to Spark main documentation.[22]

You can read more about the RDD transformation operations on the main documentation website[23] of Apache Spark. In the following we describe the most relevant operations.

1.8.3. Map Operation

Map converts the content of a single row of an RDD to a new single row of a new RDD by apply a function. We can define function by using lambda functions or a normal named function. Note that Map() operation will convert one value to another one value (one row of an RDD to a new row a one-to-one relationship).

Listing 1.15 provides an example of the map operation. We map each value of the RDD is mapped from a single character to a tuple of that character and an integer one.

Listing 1.15: Map Transformation Operation

```
# map(func)
# return a new distributed dataset formed by passing each element of the source
    through a function func
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> newRDD = rdd.map(lambda x: (x, 1))
>>> newRDD.collect()
[('a', 1), ('b', 1), ('c', 1)]
```

1.8.4. FlatMap Operation

FlatMap maps each single row of an RDD to many values so we can have a one- to-many relationship between the original RDD and the mapped one after applying a function. FlatMap requires having a function that specifies how a single value (row) of an RDD should be flat-mapped to a list of values and generate many rows of the new RDD.

Listing 1.16 is an example of flatmap operation. We have an RDDs of integer and we use Python function *"range(1, x)"* to flat-map each integer to many new integers. In this example we use collect() action operation to print out the results. Here the result is not extensive list so we can use collect() action to pull the results from the cloud to the

client. We will learn about the action operations later in this module.

Listing 1.16: FlatMap Transformation Operation

```
# flatMap(func)
# Similar to map, but each input item can be mapped to 0 or more output items
    (so func should return a Seq rather than a single item)

>>> rdd = sc.parallelize([2, 3, 4])
>>> newRDD = rdd.flatmap(lambda x: (1, x))
>>> newRDD.collect()
[1, 1, 2, 1, 2, 3]
```

1.8.5. Map Values Operation

We can use mapValues() operation to map values of an RDD and pass each value in the key-value pair of an RDD through a map function without changing the keys. Listing 1.17 is an example of mapValues() operation. We can also use a simple map() operation to only mapValues like shown.

Listing 1.17: mapValues Operation

```
>>> rdd = sc.parallelize(["a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])
>>> rdd.mapValues(lambda x: len(x)).collect()
>>> newRDD.collect()
[('a', 3), ('b', 1)]]

# The above mapValues operation can also be achieved
# by using a simple map function like the following
# Index 0 is the key and index 1 is the value
>>> rdd.map(lambda x: (x[0], len(x[1]))).collect()
[('a', 3), ('b', 1)]]
```

1.8.6. Filter Operation

Filter is an operation to do selection operation similar to relational databases. Listing 1.18 is an example of filter operation, first we generate an RDD that includes numbers 1 to 5 in its rows and we use filter operation to filter out even numbers from the RDD.

Listing 1.18: Filter Transformation Operation

```
# filter(func)
# Return a new dataset formed by selecting those
# elements of the source on which func returns true

>>> rdd = sc.parallelize([1, 2, 3, 4, 5])
>>> newRDD.filter(lambda x: x%2 = 0).collect()
[2, 4]
```

1.8.7. Reduce Operation

Reduce operation can reduce the elements of an RDD using the specified commutative and associative binary operator. We should define a function to process the reduce operation commutative (given two inputs resulting in one output) and be associative (to reduce the results of the past reduce processes).

Listing 1.19: Reduce Operation

```
# reduce(func)
# Reduces the elements of the RDD using the
# specified commutative and associative binary operator.

>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(lambda x, y: x+y)
15
```

1.8.8. Reduce By Key Operation

ReduceByKey operation can reduce the values based on a key and a reduce function that specifies how Spark should run the reduction operation. Listing 1.20 is an example of reduceByKey operation. We reduce integer values by using the key of tuples (the first element in a tuple is the key).

Listing 1.20: reduceByKey Operation

```
# reduceByKey(func)
# Merge the values for each key using an associative and commutative reduce
    function.
>>> rdd = sc.parallelize([("to", 1), ("be", 1), ("or", 1), ("not", 1), ("to", 1),
    ("be", 1)])
>>> rdd.reduceByKey(lambda x, y: x+y).collect()
[('to', 2), ('be', 2), ('or', 1), ('not', 1)]
```

1.8.9. Complete Word Count Example in PySpark

Listing 1.21 is a complete PySpark Word Count code[24]. We read the input data from a data source that is given to the script. We define the second argument of this script to be the folder/distributed File system to store the output files. Similar to Hadoop Word Count Example in Listing 1.10 and 1.11, we read the source text data file in parallel into multiple sources, then the Spark system splits each line of the text file into multiple Strings (or tokens) by using a flatMap() transformation, and by using another map operation we convert each String into a Tuple of (Word, 1), finally we used a reduceByKey transformation to reduce and get the counts of each word. You can compare the Spark code to the Hadoop MapReduce code and see how much compact and easier to understand is.

In this word count example code the following code:

```
lines = sc.textFile(YOUR-FILE-PATH)
```

provides the lines of the text file as an RDD. A text file is a well format for parallel processing because we can easily split the file into parts and process can start processing parts of the file. The RDD operation has also the following argument:

```
textFile(name, minPartitions=None, use unicode=True)
```

The number of partitions defines how the source file is split into parts and how many processes (aka executors) can start reading the file and process it.

The following formats are file formats that enable parallel data reading and processing:

- **Plain Text file**
- **Compressed files with Bzip2:** The compressing with BZIP2[25] of text files can be read in parallel. This is open source compression format mostly used in UNIX-based operation systems.
- **Apache Parquet:** Apache Parquet is a columnar storage format for Hadoop based systems called Hadoop ecosystem.

Listing 1.21: Complete Word Count Example in PySpark

```
# This is a complete Word Count Example in PySpark
from __future__ import print_function
import sys
from operator import add
from pyspark import SparkContext

sc = SparkContext(appName="PythonWordCount")

# Read the text file from source. Assume that sys.arg[1] is the file PATH
lines = sc.textFile(sys.argv[1]

# split the string line by flatMap it to multiple words and map into a tuple
# then use reduceByKey to sum the numbers (Keys in reduceByKey are the words)

counts = lines.flatMap(lambda x: x.split(' ')) \
              .map(lambda x: (x, 1)) \
              .reduceByKey(add)

# Use collect() to pull data from cloud to a single machine python
counts.collect()
```

1.8.10. Union of two RDDs

Union operation merges two RDDs into one RDD. Listing 1.22 is an example of Union which merges an RDD with itself and the result will be new RDD doubled in size.

Listing 1.22: Union Operation

```
# union(rdd)
# Build the union of a list of RDDs.
```

```
>>> rdd = sc.parallelize([1, 1, 2, 3])
>>> rdd.union(rdd).collect()
[1, 1, 2, 3, 1, 1, 2, 3]
```

1.8.11. Distinct values of an RDD

If rows of an RDD have duplicates, you can use distinct() transformation to remove the duplicates. Listing 1.23 is an example of distinct operation.

Listing 1.23: Distinct Operation

```
# distinct()
# return a new RDD containing the distinct elements in this RDD.

>>> sc.parallelize([1, 1, 2, 3]).distinct().collect()
[2, 1, 3]
```

1.8.12. Join of two RDDs

Join operation joins two RDDs using their keys. Listing 1.24 is a simple join operation and creates an intersection of the two RDDs. Listing 1.25 is a left outer join considering the operation direction to define left and right side of the two RDDs data sets.

Listing 1.24: Join Operation

```
# join(rdds)
# When called on datasets of type (K, V) and (K, W),
# retuns a dataset of (K, (V, W)) pairs
#with all pairs of elements for each key
>>> rdd1 = sc.parallelize([(1, 'a'), (1, 'b'), (5, 'c'), (2, 'd'), (3, 'e')])
>>> rdd2 = sc.parallelize([(1, 'AA'), (5, 'BB'), (5, 'CC'), (6, 'DD')])

>>> rdd1.join(rdd2).collect()

[(1, ('a', 'AA')), (1, ('b', 'AA')), (5, ('c', 'BB')), (5, ('c', 'CC'))]
```

Listing 1.25: leftOuterJoin Operation

```
# leftOuterJoin(rdds)
# Do a Left Join on the two RDDs
>>> rdd1 = sc.parallelize([(1, 'a'), (1, 'b'), (5, 'c'), (2, 'd'), (3, 'e')])
>>> rdd2 = sc.parallelize([(1, 'AA'), (5, 'BB'), (5, 'CC'), (6, 'DD')])

>>> rdd1.leftOuterJoin(rdd2).collect()
[(1, ('a', 'AA')),
 (1, ('b', 'AA')),
 (5, ('c', 'BB')),
 (5, ('c', 'CC')),
```

```
 (2, ('d', None)),
 (3, ('e', None))]
```

Listing 1.26 is a right outer join operation considering the operation direction to define the righter and left side. The same right operation can be implemented when swapping two RDDs, i.e., *rdd*1.*leftOuterJoin*(*rdd*2) is the same as *rdd*2.*rightOuterJoin*(*rdd*1) and *rdd*1.*rightOuterJoin*(*rdd*2) is the same as *rdd*2.*leftOuterJoin*(*rdd*1) so that it will generate the same result RDD.

Listing 1.26: Join Operation

```
# rightOuterJoin(rdds)
# Do a Right Join on the two RDDs
>>> rdd1 = sc.parallelize([(1, 'a'), (1, 'b'), (5, 'c'), (2, 'd'), (3, 'e'}])
>>> rdd2 = sc.parallelize([(1, 'AA'), (5, 'BB'), (5, 'CC'), (6, 'DD')])

>>> rdd1.rightOuterJoin(rdd2).collect()

[(1, ('a', 'AA')),
 (1, ('b', 'AA')),
 (5, ('c', 'BB')),
 (5, ('c', 'CC')),
 (6, (None, 'DD'))]
```

Listing 1.27 is a full outer join on the two RDDs. Any other join operation can be implemented by combining the described join operations.

Listing 1.27: fullOuterJoin Operation

```
# fullOuterJoin(rdds)
# Do a fullOuterJoin on the two RDDs
>>> rdd1 = sc.parallelize([(1, 'a'), (1, 'b'), (5, 'c'), (2, 'd'), (3, 'e')])
>>> rdd2 = sc.parallelize([(1, 'AA'), (5, 'BB'), (5, 'CC'), (6, 'DD')])

>>> rdd1.fullOuterJoin(rdd2).collect()

[(1, ('a', 'AA')),
 (1, ('b', 'AA')),
 (5, ('c', 'BB')),
 (5, ('c', 'CC')),
 (6, ('d', None)),
 (6, (None, 'DD')),
 (6, ('e', None))]
```

1.8.13. Cartesian Product of two RDDs

As we learned RDDs are distributed data structures that can be very huge in data volume distributed over a set of cluster computers. Considering this, it is not possible to pull the RDD from the cloud and write a for-loop to iterate over it. We would then use Map operations to iterate over the data. Now, consider that you need to iterate over all

combinations of each element/row of the RDD with each other. In normal case of application programming, if you need to iterate over the combinations of a collection, you would write two for-loops and then iterate over them. In large scale data processing this is not possible, instead of two for-loops, we use Cartesian Product to generate all combinations first and then process it with a subsequent Map operation. List 1.28 illustrates an example of a Cartesian product of an RDD with itself.

> **Note:** The cartesian product will generate an enormous data set that may not fit into your overall available memory. The processing of such a huge result can take a longer time to process. Also, note that cartesian will generate all combinations including combining of elements with itself and filter them out before further processing.

Listing 1.28: Cartesian Operation

```
# cartesian(rdds)
# return the Cartesian product of this RDD
# and another one, that is, the RDD of
# all pairs of elements (a, b) where
# a is in self and b is in other.

>>> rdd1 = sc.parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]

# This operation will generate huge data
# if n is number of elements in rdd, then
# new rdd after cartesian it will have n^2 (power 2)
# number of elements
```

1.8.14. GroupByKey Operation

When groupbyKey is called on an RDD of (K, V) pairs, it returns an RDD dataset of (K, Seq[V]) pairs. GroupByKey operation forces Spark to shuffle the data with same key to be pushed to the same node and can cause bottlenecks if datasets includes very large amounts of objects with the same key.

Listing 1.29 provides an example of groupByKey operation on an RDD. The result of groupByKey is an Spark internal iteratable object that includes data collection. As shown in Listing 1.29, show the content of groupByKey operation we convert it to simple python list objects, collect and sort it for pretty printing.

Listing 1.29: groupByKey Operation

```
# Group the values for each key in the RDD into a single sequence.
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> rdd.groupByKey().collect()
[('b', <pyspark.resultiterable.ResultIterable at 0x7f5f8d5c76a0>,
 ('a', <pyspark.resultiterable.ResultIterable at 0x7f5f8d5c78d0>)]
```

```
>>> sorted(rdd.groupByDey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

In our Spark implementation, we should not use "`groupByKey`" when the result of grouping can be large and cause a bottleneck in our cluster. Instead, it is preferred to use "`reduceByKey`" over "`groupByKey`" because "`reduceByKey`" operation does not force shuffle and it can reduce the dataset in a distributed setting.[26]

1.8.15. SortByKey Operation

SortByKey operation can be used on an RDD of (KEY, VALUE) pairs to sort the RDD based on the keys. The operation returns an RDD of the same type of (KEY, VALUE) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

Listing 1.30: SortByKey Operation

```
>>> rdd = sc.parallelize([("c", 1), ("a", 1), (("b", 1)])
>>> rdd.collect()
[('c', 1), ('a', 1), (('b', 1)]

>>> rdd.sortByKey().collect()
[('a', 1), ('b', 1), (('c', 1)]
```

> **Note:** If you need to get the top of an RDD use the top operation instead of sortByKey and take operation combined. The top operation would be a more efficient operation to get the top of a list.

1.8.16. Aggregate by key Operation

Sometimes we need to pass over the dataset and run aggregation operation by using a key similar to the reduceByKey operation. AggregateByKey operation can run aggregation of data items into a final result collection like a list. For example, consider that you have dataset about customers of a shop and their orders, and your goal is to collect a list of customers and a list of products that each of the customers bought from the shop. You would need to pass over the data and collect the products that each customer ordered.

An aggregateByKey operation requires the following 3 argument specifications to run the aggregation:

1. **Zero Value.** The “zero” value to initialzie the aggregation. This should for example define how to initialize a collection like a list or set.
2. **Sequence Function.** Lambda that takes x1, x2 and aggs them, where x1 already aggregated, x2 is not. For example, this argument defines how to add one element to a list.
3. **Combiner Function.** Lambda that takes x1, x2 and aggs them, where both already aggregated. For

example, this argument should define how to merge two lists into one list.

Listing 1.31 provides an example of an aggregation.

Listing 1.31: aggregateByKey Operation

```
>>> rdd = sc.parallelize([("c1", "p1"), ("c2", "p1"), ("c1", "p1"), ("c2", "p2"),
    ("c2", "p3")])
>>> rdd.collect()
[('c1', 'p1'), ('c2', 'p1'), ('c1', 'p1'), ('c2', 'p2'), ('c2', 'p3')]

# Aggregate the above RDD
>>> def mySequenceFunction(x, y);
...    x.add(y)
...    return x

>>> def myCombinerFunction(x, y);
...    x.update(y)
...    return x

>>> rdd.aggregateByKey(set([]), my_add, myCombinerFunction).collect()
[('c1', {'p1'}), ('c2', {'p1', 'p2', 'p3'})]
```

Note: combineByKey is another similar operation that is more general form of aggregateByKey which needs again 3 function definitions:

1. **createCombiner**, which turns a Values into a ”Combined Type” (e.g., creates a one-element list)
2. **mergeValue**, to merge a Values into a Collection (e.g., adds it to the end of a list)
3. **mergeCombiners**, to combine two Collection’s into a single one (e.g., merges the lists)

1.8.17. Tree Aggregate Operation

Is an aggregation operation similar to aggregateByKey operation but organizes the aggregation process in a specified tree structure. Users can specify the level of tree aggregation. Similar to aggregateByKey operations, users need to specify 3 functions.

treeAggregate(zeroValue, seqOp, combOp, depth=2)

Note: treeAggregate is an efficient and scalable operation.

1.8.18. Zip Operation

Zip operation zips an RDD with another one so that you will get a new RDD with a first element in from the first RDD and second element from the second. Zip operation considers that the two RDDs have the same number of partitions and the same number of elements in each partition so it can run the zip process. Listing 1.32 is an example of this operation.

Listing 1.32: zip Operation

```
>>> x = sc.parallelize(range(0, 5))
>>> y = sc.parallelize(range(1000, 1005))
>>> x.zip(y).collect()
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

1.8.19. Zip with Index Operation

Zips an RDD with its element indices. This is a very useful operation when we need to give elements in an RDD specific index numbers so that we can process some of them based on the index numbers in a subsequent processing step. The ordering is first based on the partition index and then the ordering of items within each partition. So the first item in the first partition gets index 0, and the last item in the last partition receives the largest index. This method needs to trigger a spark job when this RDD contains more than one partition.

Listing 1.33: zipWithIndex Operation

```
>>>  sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()
[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

1.8.20. Partitions and Partitioning Operation

As described, RDD is a collection of data elements partitioned across the cluster nodes so that Spark uses multiple processes in parallel to process the massive data set.

1.8.21. Glom Operation

Glom operation returns an RDD created by coalescing all elements within each partition into a list. Glom is a highly useful operation when you want to access batches of an RDD. Listing 1.34 is a simple example of how we can access data batches of an RDD.

Listing 1.34: Glom Operation

```
# Glom operation returns an RDD created by coalescing all elements within each
    partition into a list.
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> sorted(rdd.glom().collect())
[[1, 2], [3, 4]]
```

```
>>> sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
[[1], [2, 3], [4, 5]]
```

1.8.22. Repartition Operation

It returns an RDD that has precisely the number of data partitions requested. Repartition can increase or decrease the level of parallelism in this RDD. Internally, this shuffles the data to redistribute it. If you are decreasing the number of partitions in this RDD, consider using coalesce operation, which can avoid performing a data shuffle operation.

Listing 1.35: Repartition Operation

```
>>> rdd = sc.parallelize([1,2,3,4,5,6,7,8], 4)
>>> rdd.glom().collect()
[[1, 2], [3, 4], [5, 6], [7, 8]]
>>> len(rdd.repartition(2).glom().collect())
2
>>> len(rdd.repartition(10).glom().collect())
10
```

1.8.23. Coalesce Operation

Coalesce is doing repartitioning, but you can avoid data shuffling around the cluster. It returns a new RDD with the reduced number of data partitions.

Listing 1.36: Coalesce Operation

```
>>> sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
[[1], [2, 3], [4, 5]]
# coalesce(numPartitions, shuffle=False)
>>> sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()
[[1, 2, 3, 4, 5]]
>>> sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(2).glom().collect()
[[1], [2, 3, 4, 5]]
```

1.8.24. Cache Operation

When we want to reuse an intermediate RDD in a subsequent processing step multiple time, Cache operation are useful to keep the intermediate RDD in the main memory of the cluster machines. Listing 1.37 is a simple example of this operation. We normally use Cache operation after we clean up the raw data set and are ready for any subsequent process, e.g., learning a data model or testing a data model. If we do not cache the intermediate RDD data, the Spark process will restart to generate the intermediate data again and again from the original raw data set, on each code line that we want to use the intermediate RDD. In the subsequent Chapters, we will see many

examples of reusing intermediate RDDs when we work on large-scale machine learning data pipeline.

Cache operation has many usage forms, you can cache data in main memory or on disk of your cluster machines. It is important to know that you cluster need to have enough memory space to cache the data. We should cache RDDs when we really need them again in subsequent processing steps. An overuse of Cache operation can use an enormous size of cluster memory which may cause out of memory errors, or make the process very inefficient and slow.

Listing 1.37: Cache Operation

```
# Persist this RDD with the default storage level (MEMORY\_ONLY).
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.cache()
```

1.8.25. RDD Action Operations

As a last operation on RDDs, we run an Action operation to get the last answers of the whole data processing pipeline. The result of the action operation on an RDD is an object in the programing language (Python, Scala or Java). Listing 1.38 is an example of the `collect()` Action operation. `Collect()` operation returns a list in python that contains all the elements in the RDD.

> **Note:** You should use action operations only when you know that the results will be small enough to fit into the main memory of a single machine that you are using (your client machine). You can check the size of your RDD before running an action operation. A safe action operation that you can run on a large RDD is the operation `count()` because the result is just a single number. The listing 1.39 shows an example of using count operation.

Listing 1.38: collect() - Action Operation

```
# collect()
# Return a list that contains all of the elements in this RDD.
>>> rdd = sc.parallelize([2, 3, 4])
>>> rdd.collect()
[2, 3, 4]
```

1.8.26. Count Operation

Count operation returns the number of elements in an RDD. As the result is just a single number, count operation can be run on very large RDDs relatively fast without any problems. Listing 1.39 provides a simple example of count operation on an RDD with 3 values.

Listing 1.39: Count Action Operation

```
# count()
# Return the number of elements in this RDD.
>>> sc.parallelize([2, 3, 4]).count()
3
```

Spark provides two additional count operations. The first operation is countByKey to count the number of different keys. Listing 1.40 is an example of counting by keys, as we can see two instances of key ”a” is counted and one instance of key ”b” exist in the RDD.

Listing 1.40: countByKey Action Operation

```
# countByKey()
# Count the number of elements for each key, and
# return the result to the master as a dictionary.
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.countByKey().items())
[('a', 2), ('b', 1)]
```

The second count operation is a `"countByValue()"` operation which counts each unique value in an RDD and returns a dictionary type object. Listing 1.41 provides an example of count by value.

Listing 1.41: CountByValue Action Operation

```
# countByValue()
# Return the count of each unique value
# in this RDD as a dictionary of (value, count) pairs.
>>> sc.parallelize([1, 2, 1, 2, 2], 2).countByValue()
defaultdict(int, {1: 2, 2: 3})
# To print the results better we can do
>>> sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())
[(1, 2), (2, 3)]
```

1.8.27. First operation

First operation (Listing 1.42) returns the first value of an RDD. Note that the data elements are not sorted. The *`first()`* just returns the first element in the RDD without sorting them. The result of this operation is just one value out the given RDD so that the result would not be very large and the operation is mostly a safe operation to run depending on the size of each RDD value.

Listing 1.42: first() - Action Operation

```
# First()
# Return the first element in this RDD.
>>> sc.parallelize([4, 2, 3]).first()
4
# Or if you tuple data, e.g., it would be like following
```

```
>>> sc.parallelize([(4, 2), (1, 2), (3, 2)]).first()
(4, 2)
```

1.8.28. Take operation

You can take a specific number of values from an RDD by using (take()) operation. Listing 1.43 is an example of this operation, given an RDD and a number *n*, `"take()"` operation returns n values of the RDD. This is a very useful operation that you can use during your implementation to look at the content of an RDD and see if the content is what you would expect, e.g., if the map/reduce operation could work in a way that you wanted or know about the exact format of data tuples inside an RDD.

Listing 1.43: Take Action Operation

```
# take(num)
>>> sc.parallelize([2, 3, 4, 5, 6]).take(2)
[2, 3]
```

1.8.29. Top operation

To get the top value of an RDD based on some sorting key you can use `"top()"` action operation. Given a number k, `"top(k)"` operation returns the top k values of the RDD based on the key. Top operation takes two arguments, the number of values to return, and second (optional) a lambda to be used to get key for comparison. Spark uses the first position of a tuple as the key by default. If you would need to get the top of values based on any other value, you can define it by using a function that returns the key, like specified in Listing 1.44. You can also drop the `"key="` because it is clear.

Listing 1.44: Top() Action Operation

```
>>> counts = lines.map (lambda word: (word, 1))
>>> aggCounts = counts.reduceByKey (lambda (a, b): a + b)
>>> aggCounts.top(10, key=lamda p: p[1])

# The same top operation by dropping "key="
>>> >>> aggCounts.top(10, lamda p: p[1])
```

> **Note:** top collects the RDD object and moves it from cloud to local So result is not an RDD. So be careful that result is small.

Listing 1.45: Top Action Operation

```
# top()
```

```
# Get the top N elements from a RDD.
# It returns the list sorted in descending order.
>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)
[12]
>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
[6, 5]
```

1.8.30. Save Data on Distributed File Systems

Spark provides action operations to store large scale result files. As we described before the Spark system is a multi-threaded and multi-process system. Each Spark process would need its output.

1.8.31. Save as Text File

By using saveAsTextFile operation, we can store an RDD into text files. This operation requires a path to store text files. Path can be the local file system path like your local computer file system, starting with `"file:///"` or it can he a distributed file system that can be addressed in different ways, like storage of files in HDFS files would need a path file that starts with `"hdfs://ADDRESS/"`, or storage in Amazon AWS S3 storage would be a path that starts with `"s3://ADDRESS/"`, or google cloud storage system with `"gs://ADDRESS/"`.

The path for folders should exist on the file system. In some special cases like AWS S3, the main folder named bucket should exist, but the exact sub-folder should not exist and the operation will create it. Spark stores the final results in multiple files depending on the number of partitions of your RDD. If your final RDD is relatively small and you want to store it in a single file, then you can re-partition it into a single partition (e.g., using coalesce operation) and then use `saveAsTextFile()` operation.

1.8.32. Save as Binary Sequence File

Spark provides another operation to store results in binary format using `saveAsSequenceFile(PATH)`. A sequence file is file that can be read by parallel processes. It is also possible to compress the result before storage. The exact binary format of the file is from type of Java `org.apache.hadoop.io.Writable.`

> **Note:** Remember that Spark is running on top of Java Virtual Machine (JVM) and is using java. In execution time, Spark converts all the python or scala code to binaries and executed on top of JVM.

---

17. Apache Hive

18. Apache Spark

19. Apache Hadoop

20. Any file sources like local files, files on Hadoop HDFS, Amazon S3, Google Storage, Hypertable, Hbase)

21. Spark Python API Docs and RDD Programming Guide

22. RDD: A Resilient Distributed Dataset (RDD), the basic abstraction in Spark

23. RDD: A Resilient Distributed Dataset (RDD), the basic abstraction in Spark

24. The complete code of this example as a PySpark Script

25. BZIP2 file compression

26. Prefer "reduceByKey" over groupByKey

1.9. Lazy Evaluations

Spark uses a technique called ”Lazy Evaluation”, which means when an execution of a data processing is not needed, Spark will not run it. The data processing will only start when the result of the processing is required. For example, the code in Listing 1.46 will not trigger any data processing because the result of it is not requested by the code. It will only be requested and start running when, for example, there is an action operation like `collect()`, `count()`, `top()`, or any others.

Listing 1.46: FlatMap Transformation Operation

```
# Lazy Evaluation Example. Spark will do nothing when the following code is entered.
# Spark will just read the code, parse it and do nothing.
>>> rdd = sc.parallelize([2, 3, 4])
>>> newRDD=rdd.flatMap(lambda x: range(1, x))
```

1.10. Spark Installations Standalone vs. Cluster Mode

We can install and use Spark system in two different functional modes:

1. **Standalone installation mode.** In cases of standalone installation, we install Spark on a single machine (like a laptop) and it will run in a single process. In a stand-alone single machine installation, Spark runs a single process and executes all the tasks within the single process. Within a single process, Spark can share variables between multiple running threads so that the threads can access the shared variable/s (For example a counter variable or a result that your computation is depending on). In contrast, when running Spark in multiple processes, only data can be sent to the other process and no variables can be shared.

   Mostly Spark programmers have a local installation on their laptops to develop Spark programs fast and debug it (You can also implement your code directly on a cluster, but this causes some additional expenses for running cluster machines). It is important to note that your program may behave differently when it is running on a standalone Spark installation or a cluster installation. A Spark program that can run well in cluster mode can also run well on a standalone but not vice versa because a program that runs well in multiple processes can also run in a multithreaded single process. A spark programmer needs to develop the code in standalone mode, debug it is less expensive on a small machine with small size data, then run the program on a larger cluster and test it on a Big Dataset.

2. **Cluster Installation mode.** The cluster installation is the principal installation mode of Spark system to process Big Data. A Spark cluster mode installation runs multiple processes. Spark runs distributed processes on master node and worker nodes. Spark processes work by sending to each other batches of data inputs and result outputs in a multiple process setting.

1.10.1. Spark Driver Process

Every Spark application comprises a driver program that contains application primary function, defines distributed datasets and launches various parallel operations on a cluster. Each driver has its own resources of CPU cores and memory RAM.

1.10.2. Spark Executor Process

Every Spark application includes a set of executors that get their input data bath, the data processing function, and return the result output results. Each executor has its computation resources of CPU cores and memory RAM size. Spark includes multiple configuration parameters that you can configure the resources (CPU cores and RAM size) dedicated to the executors and drivers (See Spark configuration documentations).

> **Note:** It is possible to install Spark in cluster mode on a single machine like a single laptop, but mostly we install a standalone mode on a laptop because its ease installation.

1.11. Practice Examples

In this section, you can find two example data wrangling examples that you can use to train and improve your Spark coding abilities.

1.11.1. Practice Example: Flight Data Wrangling

The data for this example practice is flights data from different US airports. We have the following two data sets about ”Flights” and ”Airports”.

Flights dataset: Data is a large comma separate file containing 13 different fields. It includes flight year, month, day, day of the week (1-7 1 is Monday and 7 Sunday), Flight number, Airplane tail number (identifier for the airplane), origin airport and destination airport, scheduled departure time, departure time, departure delay (negative values indicate earlier outgoing flights), canceled (if the flight is canceled or not).

The header of the CSV file is:

$$YEAR, MONTH, DAY, DAY\_OF\_WEEK, AIRLINE, FLIGHT\_NUMBER, TAIL\_NUMBER, ORIGIN\_AIRPORT, DESTINATION\_AIRPORT, SCHEDULED\_DEPARTURE, DEPARTURE\_TIME, DEPARTURE\_DELAY, CANCELLED$$

You can download the file from here:

- Download from Web

Airport dataset: Another CSV file includes data about airports in USA. The file contains IATA identifier, airpot, city, state, country, latitude and longitude.

$$IATACODE, AIRPORT, CITY, STATE, COUNTRY, LATITUDE, LONGITUDE$$

You can download the file from here:

- Download from Web

Answer the following questions:

- **Question 1:** Find a list of all origin Airports. Store the list of all origin airports in a single file.
- **Question 2:** Find a list of (Origin, Destination) pairs.
- **Question 3:** Which airport had the largest departure delay in January?
- **Question 4:** Which airline carrier had the largest delay on weekends (Saturdays and Sundays)?
- **Question 5:** Which airport has the most cancellation of flights?
- **Question 6:** What are the flights cancellation percentage ratio for each carrier? Provide a printout.
- **Question 7:** Find the largest departure delay for each carrier
- **Question 8:** Find the largest departure delay for each carrier for each month.
- **Question 9:** For each carrier find the average departure delay.
- **Question 10:** For each carrier find the average departure delay for each month.
- **Question 11:** Which date of the year has the highest rate of flight cancellations? (You should calculate the rate of flight cancellation by dividing number of canceled flights by total number of flights.)

- **Question 12:** Calculate the number of flights to each destination state for each carrier, for which state do they have the largest average delay? You will need the airline and airport data sets for this question.

> **Solution:** You can find the solutions of this practice example on the Big Data Analytics Github Repository

1.12. Summary

In this module we have learned:

- About the major problems of Big Data Processing
- Usage of parallelization using multiple processes and threads
- Main architecture of Hadoop and Spark
- Different data Spark processing operations using Spark RDD data structure

In the next module we we will learn:

- How to run bulk operations in PySpark by using python Numpy library with `PySpark RDDs`
- What a Spark Dataframe is and how it can run Big Data processing pipelines
- Different Data Types in Spark Dataframes
- Spark data processing using Spark Dataframes

1.13. Further Reading References

- Main Spark Website and Documentation
- Spark Developer Resources
- Advanced Apache Spark Training - Sameer Farooqui (Databricks) (The 5-hour video on Youtube)
- Reynold Xin - Apache Spark and Scala — Scala Symposium 2017
- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, Berkeley, CA, USA, 10-10.
- SPARK Tutorial by Reynold Xin (Databricks). From DataFrames to Tungsten A Peek into Spark's Future. 2015
- Matei Zaharia. The State of Spark, and Where We're Going Next. 2013
- Book: Learning Spark by Matei Zaharia, Patrick Wendell, Andy Konwinski, Holden Karau. 2015
- Book: Spark in Action by Chris Fregly Manning 2015.

Boston University Metropolitan College