Advanced Analytics in R

– SEMINAR WINTER SEMESTER 2014/2015 –

Big Data Analytics with R and Hadoop

– SEMINAR PAPER –

Submitted by:
Nicolas Pröllochs

Advisor:
Prof. Dr. Dirk Neumann

Contents

Abstract

The statistical software R is a powerful and flexible tool that allows the development of custom algorithms. As a drawback for Big Data analysis, R mainly involves in-memory analysis. In contrast, the Hadoop framework stores the data in its own file system which allows fast, parallel and scalable data processing. This feature in combination with the integration of the MapReduce programming model overcomes the in-memory problems of sole R analysis. In addition, Hadoop is open source and allows for Big Data analysis at low computational costs. While several programming frameworks for Hadoop exist, the RHadoop collection is particularly suited for the R environment. The individual packages of this collection offer the possibility to access and utilize Hadoop directly from within R and allow to write custom MapReduce algorithms in a familiar manner.

1 Introduction

The Hadoop framework is an open source project which is designed to support large scale data processing. As its main benefit, it integrates the MapReduce programming model which allows fast processing of large datasets with a parallel, distributed algorithm. In connection with the Hadoop distributed file system (HDFS), the Hadoop framework is particular suited for Big Data analysis. Although the Hadoop implementation is written in Java, it offers a streaming API which opens up implementation alternatives for other environments. While several programming frameworks for Hadoop exist, the RHadoop collection is particularly suited for the R environment. In this regard, it offers the possibility to access and utilize Hadoop directly from within R and thus, allows to write custom MapReduce algorithms in a familiar manner.

The RHadoop collection consists of several individual packages. Thereby, the `rmr2` package is the foundation of the Hadoop integration into R. It provides the MapReduce functionality and automatically translates the R code into the required Java code. Consequently, this package provides a minimal standalone environment to utilize Hadoop from within R. The RHadoop collection additionally includes the `rhdfs` package and the `rhbase` package which allow file management and database management for the Hadoop distributed file system. A brief summary of the three packages in the RHadoop collection is given in the following.

rmr2 Allows to write MapReduce algorithms with less code and thus, allows to access to MapReduce programming paradigm to work on large datasets directly from within R. For this purpose, it sets an abstraction layer on top of the Hadoop implementation which allows to focus on the data analysis.

rhdfs Provides functions to access the file management of the HDFS directly from within R.

rhbase Provides functions for database management of the HBase distributed database from within R (not covered in this paper).

The purpose of this paper is to supply an introduction of how to utilize the Hadoop framework from within R. Therefore, we provide a brief explanation of the MapReduce algorithm and present the two main packages of the RHadoop collection. Furthermore, we highlight our presentation using practical coding examples in order to illustrate how to utilize the Hadoop framework in the R environment.

Consequently, this paper is structured as follows. Section 2 provides a brief introduction to the MapReduce algorithm which is the foundation for the Hadoop implementation. Subsequently, Section 3 provides some installation notes and a general overview of how to access the Hadoop framework. In the following, Section 4 introduces the RHadoop collection, i. e. the required packages to access Hadoop from within R. Finally, Section 5 illustrates a practical RHadoop programming example.

2 MapReduce

As its main contribution, the Hadoop framework utilizes the MapReduce programming model for large scale data processing. In fact, the MapReduce paradigm allows for efficient processing of large amounts of data with a parallel, distributed algorithm. As a further advantage, the MapReduce system manages all communications and data transfers between various parts of the systems and provides for redundancy and fault tolerance. As visualized in Figure 1, the MapReduce algorithm is divided into two basic steps. First, the `Map()` procedure performs filtering and sorting of the data. Second, the `Reduce()` procedure performs a summary operation.

From a logical view, the map and reduce functions are both defined with respect to data that is structured in key-value pairs. The map function takes one key-value pair (k_i,v_i) from one specific type as input data and returns a list of pairs of a different type, formally

$$Map(k_1,v_1) \rightarrow list(k_2,v_2). \tag{1}$$

In this manner, the map function is applied in parallel to every pair in the input dataset. This process leads to a list of pairs for each call. Afterwards, all pairs with the same key from all lists are collected and grouped together, resulting in one group for each key. Subsequently, the reduce function is applied to each group, resulting in a collection of values, formally

$$Reduce(k_2,list(v_2)) \rightarrow list(v_3). \tag{2}$$

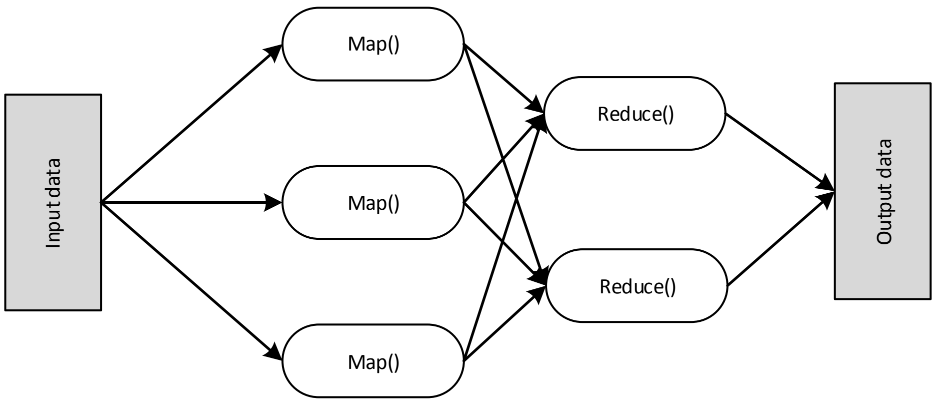


Figure 1: The MapReduce process.

Finally, the collection of all calls of the reduce function leads to the result list. Thus, the MapReduce framework transforms a list of key-value pairs into a list of values. For more clarity, the following enumeration lists up the individual steps of the MapReduce algorithm.

Input split	Splits the MapReduce job into small tasks where each task analyses a dataset chunk.
Map	The map function is computed for any task and produces a key-value pair.
Shuffle	Data associated with the same key is grouped and moved to the same place.
Reduce	The reduce function is computed for any key.
Output	The outputs from the Reduce processes are collected, producing the results.

3 Installation Notes

The Hadoop installation[1] is placed on a virtual machine running on Ubuntu 14.04 with 2 cores at 2.1 GHz, 4 GB RAM and 120 GB of storage. An overview of the installation properties of Hadoop is given in Table 1.

Software	Access	Note
Ubuntu 14.04	`is-hadoop-01.vwl.privat`	VM
Data Node	`is-hadoop-01.vwl.privat:50070`	Browser
RStudio Server	`is-hadoop-01.vwl.privat:8787`	Browser

Table 1: The installation properties of the Hadoop implementation.

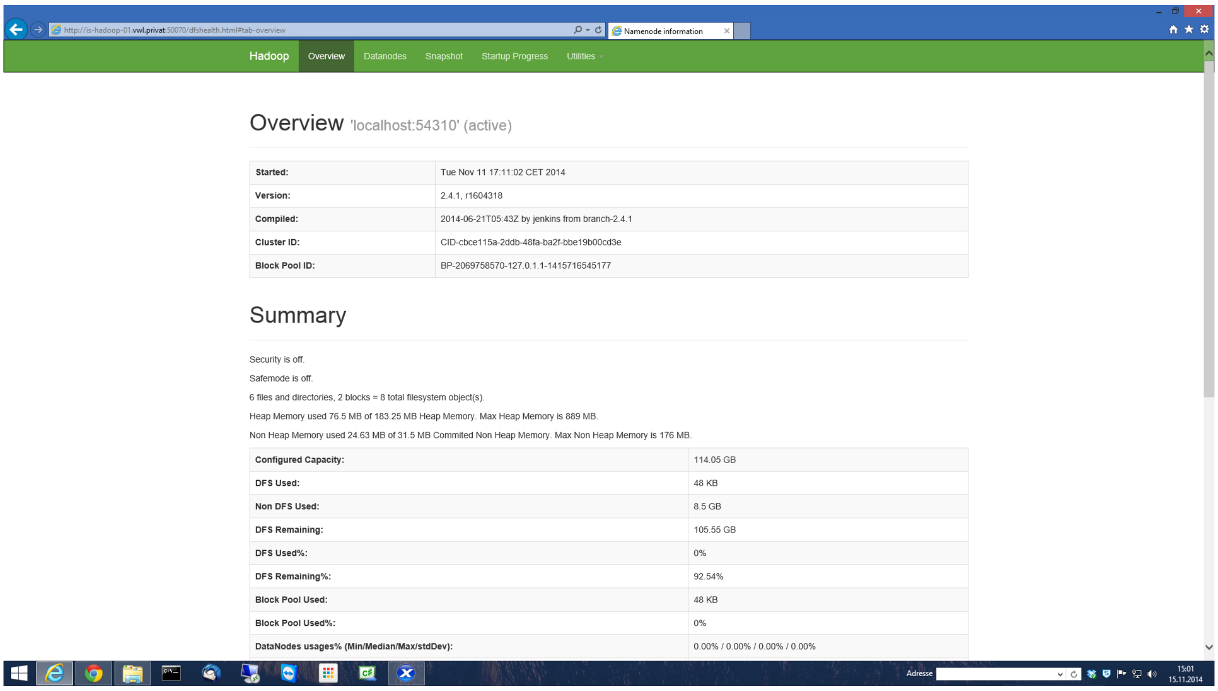
A status report of the Hadoop framework is available from the browser via `is-hadoop-01.vwl.privat:50070`. Furthermore, this browser interface (Figure 2) allows to access log files and to browser Hadoop distributed file system (HDFS).

RStudio Server can be accessed from the browser via `is-hadoop-01.vwl.privat:8787`. This browser environment provides the same graphical interface as the common local RStudio environment (Figure 3). Before starting a session, RStudio requires username and password which makes it possible to close a session and continue at a later point. Furthermore, this opens the possibility to access RStudio and Hadoop with multiple user accounts at the same time. As an advantage of this multi-user setup, each user is able to upload local files to his home directory using the upload button in the graphical interface of RStudio.

4 Access within R

This section provides a step-by-step instruction of how to utilize Hadoop from within the statistical software R. In this regard, we provide a brief introduction to the required R packages from the RHadoop collection. As depicted in Section 3, the R interface for the Hadoop implementation can be accessed using RStudio Server at `is-hadoop-01.vwl.privatonport8787`. This serves as the initial point for the following package descriptions.

1 Installation instructions available from `http://www.bogotobogo.com/Hadoop/BigData_hadoop_Install_on_ubuntu_single_node_cluster.php`.



Overview 'localhost:54310' (active)

Started:	Tue Nov 11 17:11:02 CET 2014
Version:	2.4.1, r1604318
Compiled:	2014-06-21T05:43Z by jenkins from branch-2.4.1
Cluster ID:	CID-cbce115a-2ddb-48fa-ba2f-bbe19b00cd3e
Block Pool ID:	BP-2069758570-127.0.1.1-1415716545177

Summary

Security is off.

Safemode is off.

6 files and directories, 2 blocks = 8 total filesystem object(s).

Heap Memory used 76.5 MB of 183.25 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 24.63 MB of 31.5 MB Commited Non Heap Memory. Max Non Heap Memory is 176 MB.

Configured Capacity:	114.05 GB
DFS Used:	48 KB
Non DFS Used:	8.5 GB
DFS Remaining:	105.55 GB
DFS Used%:	0%
DFS Remaining%:	92.54%
Block Pool Used:	48 KB
Block Pool Used%:	0%
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%

Figure 2: The Hadoop status interface, accessible via `is-hadoop-01.vwl.privat:50070`.

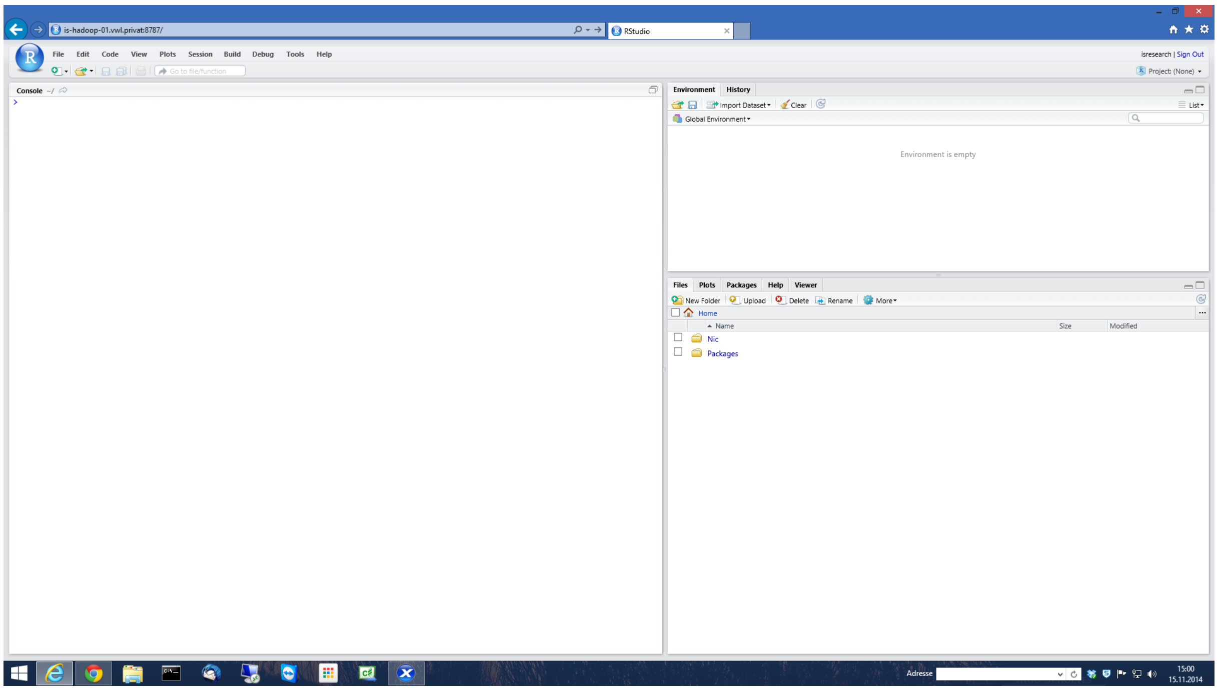


Figure 3: The RStudio Server interface, accessible via `is-hadoop-01.vwl.privat:8787`.

4.1 The `rmr2` Package

The `rmr2` package is the foundation for the integration of Hadoop into R. In fact, it provides the MapReduce function and thus, allows to write custom MapReduce jobs directly from within R. For this purpose, the package automatically translates the R code into the required Java code for the Hadoop framework. Furthermore, it also allows basic access to the Hadoop distributed file system (HDFS). As a further advantage, this package allows the MapReduce function to access all local R variables. Consequently, the `rmr2` package provides a minimal standalone environment to utilize Hadoop from within R. In order to write custom MapReduce algorithms for Hadoop, the MapReduce function of the `rmr2` package takes the following inputs.

- Map function
- Reduce function
- HDFS path of input file
- HDFS path of output file (optional)

All of the functions, i. e. the `map` function and `reduce` function take a key-value matrix as input data and furthermore, also store their respective output in key-value form. The input and output path arguments refer to files on the Hadoop distributed file system. As mentioned before, the `rmr2` package also provides rudimentary functions to access the HDFS. This functionality is implemented using the following two functions.

to.dfs()	Stores data on the HDFS
from.dfs()	Retrieves data from the HDFS

The following examples illustrate the integration of the `rmr2` package into R. First of all, the `rmr2` package requires access to the streaming API and to the Hadoop command in the Linux installation folder. In R, these variables are stored using two environment variables, namely *HADOOP_CMD* and *HADOOP_STREAMING* which are given in the following code snippet.

```
Sys.setenv(HADOOP_CMD = "/usr/local/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING = "/usr/local/hadoop/share/hadoop/tools/lib/
    hadoop-streaming-2.4.1.jar")
```

This setting enables us to load the `rmr2` package to access Hadoop from within R by

```
library(rmr2)
```

As previously mentioned, the `rmr2` package provides two basic commands to store and retrieve objects from the HDFS. First, the `to.dfs()` command stores a big data object on the HDFS among an arbitrary variable name. Second, the object can be retrieved using the `from.dfs()` command. In both cases, the brackets embrace the desired variable name. The following example stores a vector consisting of the numbers 1–10 among the variable name *input* on the HDFS. Consequently, the *output* variable retrieves the data from the HDFS.

```
input = to.dfs(1:10)
output = from.dfs(input)
output
$key
NULL

$val
 [1]  1  2  3  4  5  6  7  8  9 10
```

The core application of the rmr2 package is the `mapreduce()` function that allows to write custom MapReduce algorithms. The following code provides a simple example that demonstrates how to exploit RHadoop for data analysis. For this purpose, we utilize GDP data from the year 2012 to calculate the number of countries that have a greater or lower GDP than Germany. We start by storing the GDP data[2] in a variable *gdp* while we store the GDP of Germany among a variable *gdp_germany*. In addition, we send the GDP data to the HDFS via the `to.dfs()` command using the following code snippet.

```
gdp <- read.csv("GDP.csv", sep = ";")
head(gdp)
  Economy  Mio_USD
1     USA 16800000
2     CHN  9240270
3     JPN  4901530
4     DEU  3634823
5     FRA  2734949
6     GBR  2521381

gdp_germany = gdp[4,2]

gdp.values <- to.dfs(gdp)
```

Afterwards, we are able to define the map function and the reduce function. In our case, the map function compares the current input GDP value with the GDP of Germany and emits a key-value pair $(less, 1)$, if the current GDP value is smaller than the GDP of Germany and a key-value pair $(greater, 1)$ otherwise. The following code snippet demonstrates the specification of the required functions.

```
gdp.map.fn <- function(k,v) {
  key <- ifelse(v[,2] < gdp_germany, "less", "greater")
  keyval(key, 1)
}

gdp.reduce.fn <- function(k,v) {
  keyval(k, length(v))
}
```

Consequently, the map function, reduce function and the input stored on the HDFS allows to run the MapReduce process by calling the following function.

2 GDP data available from http://data.worldbank.org/data-catalog/GDP-ranking-table

```
gdp.calc <- mapreduce(input = gdp.values,
                      map = gdp.map.fn,
                      reduce = gdp.reduce.fn)
```

Subsequently, R initiates a Hadoop streaming job to process the data using the MapReduce algorithm.

```
...
14/11/27 04:25:13 INFO mapreduce.Job:  map 100% reduce 100%
14/11/27 04:25:13 INFO mapreduce.Job: Job job_local311308709_0001 completed
    successfully
...
```

If the MapReduce job is completed succesfully, the results can be retrieved using the `to.dfs()` command in the following code snippet. We note that the output contains two possible key values indicating whether a country has a greater or lower GDP in comparison to Germany.

```
from.dfs(gdp.calc)

$key
[1] "less"    "greater"

$val
[1] 188   4
```

Evidently, there are 188 countries having a greater GDP than Germany and respectively, 4 countries that have a lower GDP than Germany.

4.2 The `rhdfs` Package

The `rhdfs` package provides basic connectivity to the HDFS which allows programmers to browse, read, write and modify files stored in the HDFS. This functionality is provided by the well-known Linux shell commands, e. g. `ls, rm, mkdir`, etc. In order to utilize the `rhdfs` package in R, we have to set up the required environment variables and need to initialize the HDFS using the `hdfs.init()` function. The following code provides a starting point to utilize the `rhdfs` package.

```
Sys.setenv(HADOOP_CMD="/usr/local/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING="/usr/local/hadoop/share/hadoop/tools/lib/hadoop-
    streaming-2.4.1.jar")
library(rhdfs)
hdfs.init()
```

After setting up the package, we are able to access the HDFS. The following code snippet allows to create and browse a new directory.

```
hdfs.mkdir("/examples")
hdfs.ls("/")

  permission  owner      group size          modtime      file
1 drwxr-xr-x hduser supergroup    0 2014-11-25 14:37 /examples
```

```
2 drwxr-xr-x hduser supergroup    0 2014-11-25 12:41      /tmp
3 drwxr-xr-x hduser supergroup    0 2014-11-25 12:38     /user
```

As the probably most useful function of the `rhdfs` package, the `hdfs.put()` command allows to copy a file from the local filesystem to the HDFS. This function takes two arguments: The first argument provides the local path while the second argument points at the requested path on the HDFS. The following code snippet copies a local text file to a specific path on the HDFS. In addition, the code introduces the `hdfs.read.text.file()` command that allows to read the stored text file from the HDFS.

```
writeLines(text = c("Cat Tiger Horse", "Tiger Cat Dog", "Dog Tiger Horse"),
    sep = '\n', "wordcount.txt")
readLines("wordcount.txt")
[1] "Cat Tiger Horse" "Tiger Cat Dog"   "Dog Tiger Horse"

hdfs.put("wordcount.txt", "/examples/wordcount.txt")
hdfs.ls("/examples")

  permission  owner      group size          modtime                    file
1 -rw-r--r-- hduser supergroup   46 2014-11-25 14:38 /examples/wordcount.txt

hdfs.read.text.file("/examples/wordcount.txt")

[1] "Cat Tiger Horse" "Tiger Cat Dog"   "Dog Tiger Horse"
```

It is noteworthy that the `rhdfs` package provides a huge number of additional commands. Two of them are given in the following code. Thereby, the `hdfs.rename()` command allows to rename a file directly on the HDFS while the `hdfs.rm()` command allows to delete a specific file or directory.

```
hdfs.rename("/examples/wordcount.txt","/examples/renamed.txt")
hdfs.ls("/examples)

  permission  owner      group size          modtime                  file
1 -rw-r--r-- hduser supergroup   46 2014-11-25 14:39 /examples/renamed.txt

hdfs.rm("/examples/renamed.txt")
hdfs.ls("/examples")

NULL
```

5 Example: Word Count

The following example counts the appearance of each in word in a set of documents. Therefore, each document is split into words and each word is counted by the `map` function while each `map` function uses a word as result key. Afterwards, the Hadoop framework automatically puts together all key-value pairs with the same keys, i. e. words. Finally, the `reduce` function sums up all its input values and thus, finds the total appearances of the corresponding word. In this regard, Figure 4 visualizes the word count process of the MapReduce algorithm.

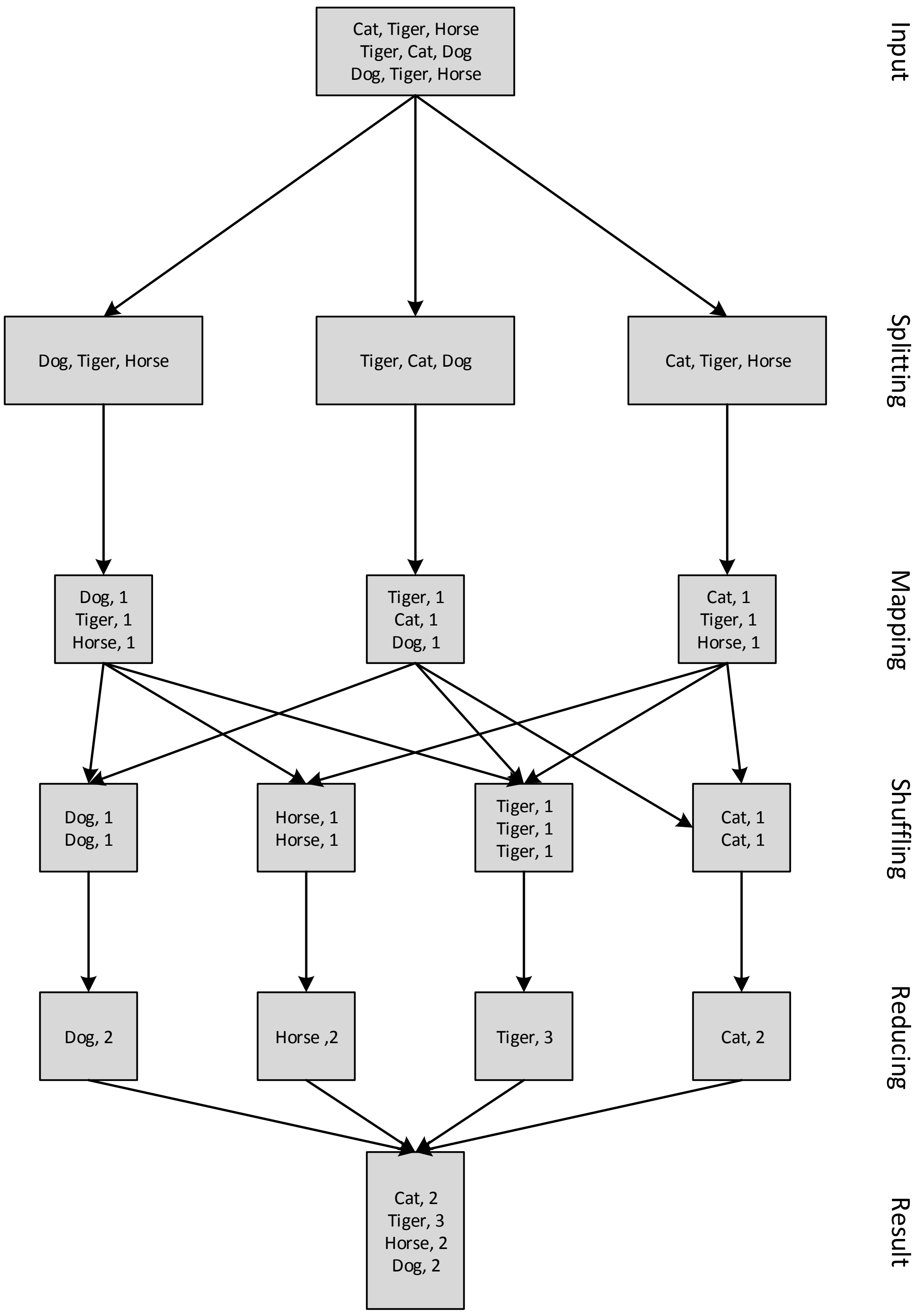


Figure 4: The MapReduce word count process.

To implement this word count task in R, we have to start by defining the required `map` function. Thus, the following code produces a function that splits each documents into words and creates a key-value pair for each single word.

```
wc.map = function(k,v) {
  keyval(
    unlist(
      strsplit(
        x = v,
        split = " ")),
    1)}
```

In a second step, we have to define the `reduce` function which summarizes the results from the `map` function. Therefore, the following reduce function sums up each key-value pair and thus, completes the word count process. Using the `map` function, the `reduce` function and a text file on the HDFS, we are able to submit the MapReduce job using the arguments in the following code.

```
wc.reduce =
  function(word, counts ) {
    keyval(word, sum(counts))}

wc.calc = mapreduce(
      input = "/examples/wordcount.txt",
      input.format = "text",
      map = wc.map,
      reduce = wc.reduce)
```

Finally, we are able retrieve the calculations from the HDFS using the `from.dfs()` command. In this regard, the following code snippet presents the final results of the word count process.

```
result = from.dfs(wc.calc)
result
$key
[1] "Cat"   "Dog"   "Horse" "Tiger"

$val
[1] 2 2 2 3

data.frame(key=keys(result),val=values(result))
    key val
1   Cat   2
2   Dog   2
3 Horse   2
4 Tiger   3
```

6 Summary

Hadoop provides a powerful framework for Big Data analysis. This framework stores the data in a custom file system which allows fast, parallel and scalable data processing. In order to utilize Hadoop from within the statistical software R, the RHadoop collection offers an easy to use open source implementation. As a large benefit, this combination of R and Hadoop allows to overcome the in-memory problems of common R analysis. For this purpose, the `rmr2` package allows to write custom MapReduce algorithms directly from within R and provides automatic translation into Java code which is a requirement for Hadoop. Furthermore, the `rhdfs` package allows to access the Hadoop distributed file system in R using standard Linux shell commands. It is noteworthy that utilizing Hadoop is only efficient on large datasets because the overhead of setting up a MapReduce job is likely higher than the runtime of the job itself. In addition, the current Hadoop installation is based on a single node which does not exploit the full potential of Hadoop. Consequently, future efforts could expand the single node Hadoop cluster using additional slave nodes and thus, significantly improve the performance on Big Data analysis.