A Study on Performance Analysis Model for Supporting Big Data Analytics with R

**Abstract**

Over the past years, R has become a popular analytic environment for many domain science fields due to its script style programming language and vast existing scientific packages support. While R was traditionally designed for single computer usage model, many R users begin to seek the possibility of adapting existing R analysis workflow in high performance computing environment due to increasing computational requirement as they are working on big data set. There are a number of efforts that bridge R with new programming models for big data analysis, such as MapReduce and Spark. We have conducted performance comparison studies for utilizing those approaches, including RHadoop, RHIPE, SparkR, and Hadoop streaming on a 48 nodes cluster. In this paper, we report our performance evaluation results, lessons learned and recommend best practices when supporting R for big data analysis with high performance resources.

Keywords: *R, big data, Hadoop, Rhipe, RHadoop, Streaming*

**Introduction**

The amount of data used in applications has been increasing rapidly; R users in the scientific community are attempting to use R with large files. Our aim was to explore the different packages that can be used with R to facilitate big data analysis. We found three ways of combining R with Hadoop: RHadoop, RHIPE, and Hadoop Streaming. We examined the performance of these packages on common big data programs, like wordcount and k-means. We deduced the optimal settings for relatively small files, did a scalability analysis for those settings, and checked the performance for larger files.

· Problem statement, i.e. give a summary of what the benchmark about. What we are measuring, e.g. performance, accuracy, quality , usability etc.

· Motivation, i.e. why we are interested in this project, e.g. enabling practical analysis, computational cost analysis, best practice study,

· Relevant work on benchmarking R/Big data application

· Contribution of this work

**Background**

**R**

R is an open source statistical programming language for performing statistical and predictive analysis, data mining and visualization functions on data. R is the most popular language for these purposes. As the data gets large, issues begin to surface. Large, complex datasets can be structured, semi-structured, or unstructured and typically do not fit into memory. Hence, it is natural to attempt to scale up R computations using Hadoop/Spark.

**Differences/Advantages of each**

**Hadoop**

Apache Hadoop is an open source Java framework for processing and querying large amounts of data on large clusters of commodity hardware. It is the most commonly used framework for Big Data processing. Two main features of Hadoop are HDFS and mapreduce.

**Hadoop Streaming**

Hadoop Streaming enables us to write map and reduce functions in any programming or scripting language that supports reading data from standard input and writing to standard output. R and Python are two common choices for such a language.

The default input format for all hadoop streaming jobs is TextInputFormat, which reads the data one line at a time. (1)

In general we have found Hadoop Streaming to be very efficient, the only drawback being the language of choice. A well detailed evaluation of Hadoop Streaming was given in “More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming.” It concluded the cause of vast overhead in Hadoop streaming to be caused by the linux *pipe()* and Java Read/Write calls.

Python outmaneuvered R in performance during testing with Hadoop streaming.

**RHadoop**

A family of R packages that act as a wrapper for Hadoop and allow the execution of Hadoop jobs without ever leaving the application. Composed of 5 separate packages *ravro*, *plyrmr*, *rmr*, *rhdfs*, and *rhbase*. The main two components are *rhdfs* and *rmr*. Rhdfs primarily responsible for the handling of HDFS operations such as file manipulation, reading and writing, and directory traversal. The main component of RHadoop *rmr* is responsible for the submission of hadoop jobs and passing the hadoop configurations.

RHadoop packages the map and reduce functions and passes them to the Hadoop streaming jar.

**RHIPE**

A single vast package that contains both HDFS and Hadoop management operations. Installation can be a little tricky because of the dependencies, especially Protocol Buffers, but once configured it’s very reliable and stable. It doesn’t depend on the Hadoop streaming jar and comes with a custom inputformat LApplyInputFormat. LApplyInputFormat allows the file processing to be done chunk wise instead of the default line at a time. Not using the standard Hadoop streaming however deprives it of certain functionalities, like the ability to swap out input formats.

A key benefit of RHIPE is the serialization and deserialization of R data structures allowing the user to call rhcollect(*key*, [*list, object*, *data.frame*, *matrix*]). As of 0.75 RHIPE is capable of serializing scalar vectors such as integers, characters, numerics, logicals, complex, and raw. Lists of scalar vectors and attributes of objects. Rhipe can also serialize data frames, factors, and matrices. With this knowledge we can easily emit many R data structures and easily read them using the reducer.

· Background on R and the packages involved in this study (Rhipe, Rhadoop, Streaming(R& Python)

· Background on Hadoop/spark, especially streaming aspect, (Spark doesn’t have streaming)

· Background on relevant work in performance benchmarking

**Methods**

The “hello world” program for Hadoop: *wordcount* was used for the benchmarking.

Initially we ran tests on a relatively small file (~10GB) with different combinations of mappers and reducers with and without a combiner. We expected to determine the optimal parameters using the above technique. We then performed scalability testing using the parameters found in the small file tests. Finally we performed tests on a very large file (~390 GB) for the packages that were scalable to test for resource exhaustion.

The first test allowed us to find the optimal mapper/reducer ratio for the data set used, Google N-gram library(3). Then we used the ratio and increased the file sizes to test the scalability of the package, we then performed testing on large file sizes.

These are the mapper and reducer functions used in the wordcount testing.

**RHadoop**

wordcount =

function(input, output = NULL, pattern = " "){

wc.map = function(., lines) {

keyval(unlist(strsplit(

x = lines,

split = pattern, fixed = TRUE)),1)}

wc.reduce= function(word, counts ) {

keyval(word, sum(counts))}

mapreduce(

input = input,

output = output,

input.format = "text",

map = wc.map,

reduce = wc.reduce,

in.memory.combine = combiner,

combine = combiner

)}

wordcount(“book.txt”)

**RHIPE**

mapper <- expression( {  
 keys <- unlist(strsplit(unlist(map.values), split=' '))  
 value <- 1  
 lapply(keys, FUN=rhcollect, value=value)  
} )

reducer <- expression(  
 pre = {  
 running\_total <- 0  
 },  
 reduce = {  
 running\_total <- sum(running\_total, unlist(reduce.values))  
 },  
 post = {  
 rhcollect(reduce.key, running\_total)  
 })  
(4)

**R+Streaming**

**Mapper.R**

#! /usr/bin/Rscript  
con <- file("stdin", open = "r")  
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0) {  
 line <- trimWhiteSpace(line)  
 words <- splitIntoWords(line)  
for (w in words)  
 cat(w, "\t1\n", sep="")  
}  
close(con)

**Reducer.R**

#! /usr/bin/Rscript

env <- new.env(hash = TRUE)  
con <- file("stdin", open = "r")  
  
count <-0  
cur\_word<-""  
  
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0){  
 split <- splitLine(line)  
 word <- split$word  
 if(cur\_word!=word){  
 cat(cur\_word,"\t",count,"\n",sep="")   
 cur\_word=word  
 count=split$count  
 }else{  
 count=count+split$count  
 }  
}  
cat(cur\_word,"\t",count,"\n",sep="")  
close(con)

**Python+Streaming**

**Mapper.py**

#!/usr/bin/env python  
  
import sys  
  
for line in sys.stdin:  
 line = line.strip()  
 words = line.split()  
 for word in words:  
 print '%s\t%s' % (word, 1)

**Reducer.py**

#!/usr/bin/env python  
from operator import itemgetter  
import sys  
current\_word = None  
current\_count = 0  
word = None

for line in sys.stdin:  
 line = line.strip()  
 word, count = line.split('\t', 1)  
 try:  
 count = int(count)  
 except ValueError:  
 continue

if current\_word == word:  
 current\_count += count  
 else:  
 if current\_word:  
 print '%s\t%s' % (current\_word, current\_count)  
 current\_count = count  
 current\_word = word

if current\_word == word:  
 print '%s\t%s' % (current\_word, current\_count)

· Describing the benchmark and tests completed, e.g. word count, k-means, sorting, data set used etc. and the characteristics of different tests e.g. I/O bound, CPU bound, network bound etc. (Wordcount and kmeans)

· Describing approaches used in testing, e.g. Rhadoop, RHIPE, Spark R, Hadoop Streaming, Python code as a reference of comparsion.

· Describing the hardware resources used for testing, (Rustler, 51 nodes with 117GB of RAM)

· Describing the parameter of consideration used with test.

**Results**

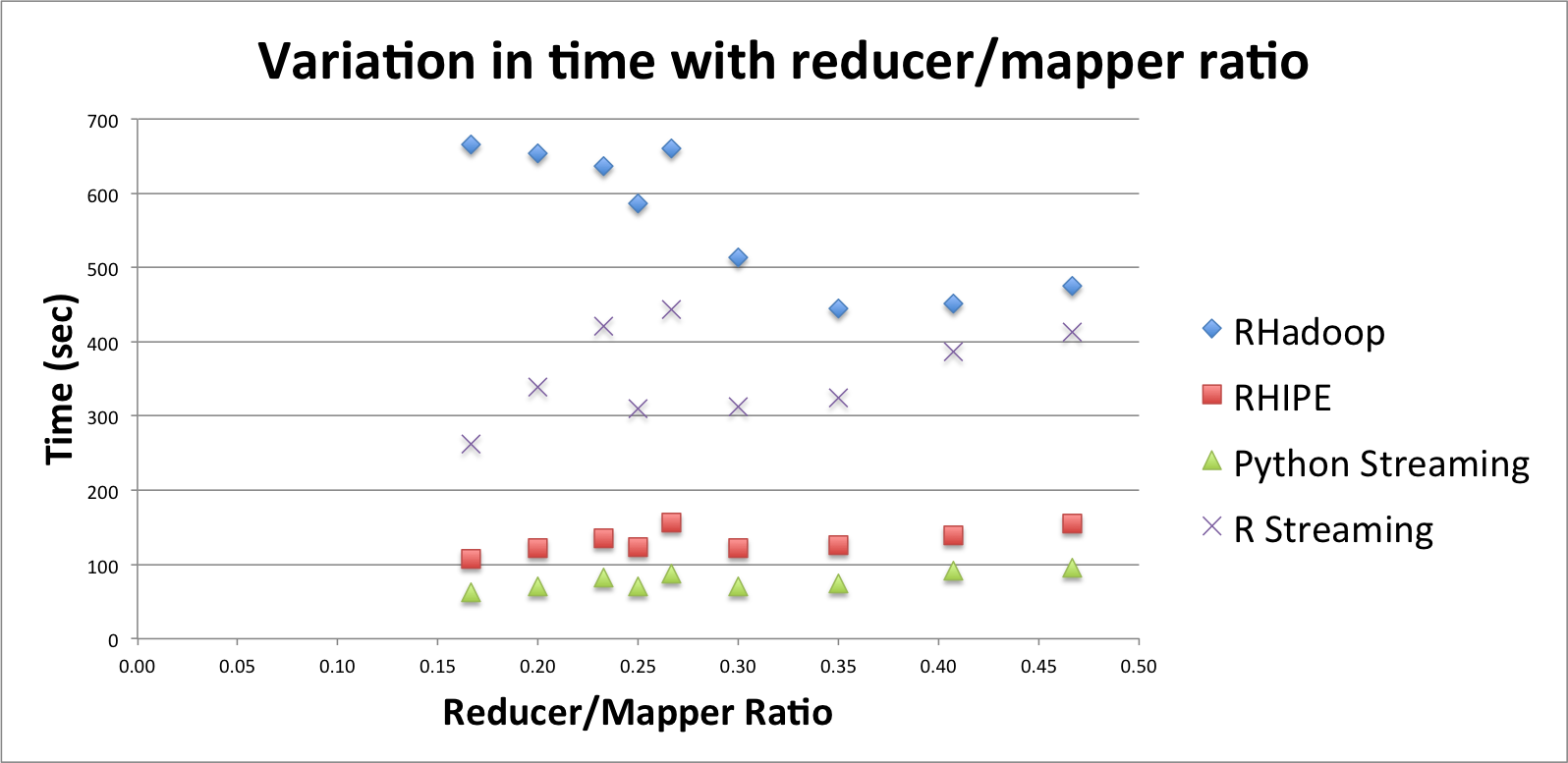


Figure 1:

· Parameter selection, e.g. number of mapper and reducer, block size used with each data set, explanation on why those are better set of parameter etc.

· Comparison among different approaches in R including the performance measurement on the same set of workload under the optimal parameter settings.

· Comparison with non-R/hadoop approach, e.g. vanilla version of R, hadoop streaming with python etc.

**Best practice recommendation**

We have noticed a couple of improvements that should be kept in mind when developing R related Hadoop applications. A significant time reduction can be achieved by replacing regular expressions with fixed characters, in our simple wordcount experiments processing time decreased by approximately 45% even after mapper and reducer optimization was achieved.

Increasing the number of mappers and reducers doesn’t always increase performance and beyond the optimal ratio the time becomes minimally impacted. Increasing mappers does increase the performance but, the continual increase of reducers eventually leads to performance degradation as can be seen in figure(5).

For RHadoop and Hadoop Streaming we can directly specify the mapper and reducer counts using *mapreduce.job.maps* or *mapreduce.job.reduces* parameters. To specify the number of mappers in RHIPE we specify the **chunk size** in bytes *mapred.max.split.size.*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* file\_size / chunk\_size\*1024\*1024 = NUM\_Mappers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

By default the number of mappers is chosen by the HDFS number of blocks, with the default **block-size** being 64MB and in production 128MB. Given this a 1GB file will be stored in 8 blocks, and given 8 mappers. However in RHIPE specifying *mapred.max.split.size* to 64MB will give us 16 mappers. The same can be accomplished in Streaming by specifying *mapreduce.job.maps=16*, however the block-size is the upper bound for chunk size. Hence, we cannot request less than 8 mappers or request the chunk-size to be more than 128MB.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #Splits /alpha 1/chunk\_size\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Comparing Python streaming times to RHIPE, RHadoop, and R Streaming we can conclude that R I/O can be very slow when done line at a time as streaming does by default. Depicted by the slow RHadoop and R+Hadoop Streaming times. RHIPE taking a different approach reads chunks at a time, with speeds closer to Python streaming implying lower I/O overhead. By supplying a custom InputFormat to the Hadoop streaming jar we should be able to achieve comparable speeds.

In RHIPE testing the map and reduce functions does not have to be a cryptic guessing game

* + map.keys<-lapply(irisSplit,”[[“,1)
  + map.values<-lapply(irisSplit,”[[“,2)
  + cat book.txt | ./mapper.py | sort k1,1 | ./reducer.py

**Hadoop Streaming**

$ hadoop jar ${HADOOP\_STREAMING\_JAR} \

-Dmapreduce.job.name="Wordcount-book.txt" \

-Dmapreduce.job.maps=100 \

-Dmapreduce.job.reduces=20 \

-Dmapreduce.map.java.opts=-Xmx11500M \

-Dmapreduce.reduce.java.opts=-Xmx11500M \

-files ./mapper.R,./reducer.R \

-mapper ./mapper.R \

-reducer ./reducer.R \

-combiner ./reducer.R \

-input ./data/book.txt \

-output ./output/book \

**RHIPE**

mapred = list(

mapred.task.timeout=1

, mapred.max.split.size=as.integer(1024\*1024\*block\_size)

, mapreduce.job.reduces=num\_reducers

)

rhipe.results <- rhwatch(

map=mapper, reduce=reducer,

input=rhfmt(input.file.hdfs, type="text"),

output=output.dir.hdfs,

jobname=paste("rhipe\_wordcount\_", 1 ,sep="-"),

mapred=mapred)

**RHadoop**

bp =

list(

hadoop =

list(

D = paste("mapred.job.name=", args[[1]], sep=''),

D = "mapreduce.map.memory.mb=11500",

D = "mapreduce.reduce.memory.mb=11500",

D = "mapreduce.map.java.opts=-Xmx11500M",

D = "mapreduce.reduce.java.opts=-Xmx11500M",

D = paste("mapreduce.job.maps=", args[[2]], sep=''),

D = paste("mapred.reduce.tasks=", args[[3]], sep='')

))

rmr.options(backend.parameters = bp);

rmr.options("backend.parameters")

· What you learned and suggestion if others want to use use RHIPE/RHADOOP/Hadoop+R? (Other than choose another language instead)

* Mapper and reduce functions
  + map.keys<-lapply(irisSplit,”[[“,1)
  + map.values<-lapply(irisSplit,”[[“,2)
  + cat book.txt | ./mapper.py | sort k1,1 | ./reducer.py
* RegEx versus fixed
  + RHadoop / R Streaming example times
* Affecting functionality
  + D parameters
  + Mapper/Reducer counts

Considering things like

· suggestions general approach for different workload,

· Suggestions for different types of user, computer developer, domain scientist etc.

· Suggestions on parameters may be used

**Conclusion**

· Summarizing what you have achieved in this project,

· What are the pro and cons of using R for Big data analytics, considering both the performance, flexibility, and development effort etc.

## Reference

**Data Mining Applications with R** <https://books.google.com/books?id=nYpqAAAAQBAJ&pg=PA10&lpg=PA10&dq=rhipe+map.values&source=bl&ots=wL3Xylq-D4&sig=ZpI52s2mEoEipbHKIPEZszRiQ8k&hl=en&sa=X&ei=YYhCVfClKoO0ogTL2oCwDg&ved=0CEsQ6AEwBw#v=onepage&q=rhipe%20map.values&f=false>

(1)<http://hadoop.apache.org/docs/r1.2.1/streaming.html#Streaming+Command+Options>

(2)Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. 2011. More convenient more overhead: the performance evaluation of Hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (RACS '11). ACM, New York, NY, USA, 307-313. DOI=10.1145/2103380.2103444 <http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/2103380.2103444>

(3)

<https://storage.googleapis.com/books/ngrams/books/datasetsv2.html>

(4) Glenn K. Lockwood Github

<https://github.com/glennklockwood/paraR/blob/master/rhipe/wordcount-rhipe.R>

Specify versions of packages we use. The datasets on which the jobs were run

Differences between R and Python/ Advantages of R

Different lines of the same graph for different settings

Time Vs no. of mappers for a particular file for all 4 packages with reducer fixed at the optimal (10, 20, 30, 40, …)

Time Vs no. of reducers for the same file for all 4 packages with no of mappers fixed at optimal. (1, 2, 4, 8 …) (5, 10,15.)

Comment on the optimal ratio between the no. of mappers and reducers. Should one use a combiner? What does in.combine do?

Make a bar graph showing the times of all 4 together for the small file.

Considerable difference between the times with and without regex, Small choices in R matter

Tips for R functions to use

Scalability graphs for all

R Streaming gives around linear scalability, RHIPE also has linear scalability for considerable file size.

RHadoop doesn’t scale

Redo python Streaming time for 285.1 GB. Should have much better than linear scaling.

RHIPE (~ 390 GB)

Largest possible block size: 128 MB: 3167 mappers.

Time reduces as compared to 5058 mappers: Since there is a limit on how many mappers can run concurrently, using a lot of mappers does not necessarily help.

Python Streaming times for large files.

Specify the parameters for all the times we provide

Provide codes ran to achieve the times.

Things to consider:

If RHadoop is just a wrapper around R Streaming, why is it inconsistent?

Try R Streaming without the memory options and see if it is still consistent. Maybe also try RHadoop with the memory options.

Difference between RHIPE and R Streaming?

Reading by block instead of line by line?

R Streaming vs Python Streaming?

Slow I/O in R?

Is RHIPE as good as Python Streaming? If yes, how does it achieve that? Is there a RHIPE for Python?

Test out RHIPE "mapred.map.tasks” for RHIPE