# RHIPE Documentation

***Release 0.61***

**Saptarshi Guha**

September 08, 2010

# CONTENTS

Contents:

CHAPTER

ONE

# INSTALLATION

RHIPE is an R package, that can be downloaded at this website. To install the user needs to

- Set an environment variable `$HADOOP` that points to the Hadoop installation directory. It is expected that `$HADOOP\bin` contains the Hadoop shell executable `hadoop`.
- A version of Google's Protocol Buffers (here) greater than 2.3.0

Once the package has been downloaded the user can install it via

```
R CMD INSTALL Rhipe_version.tar.gz
```

where `version` is the latest version of RHIPE. The source is under version control at GitHub .

This needs to be installed on *all* the computers: the one you run your R environment and all the task computers. Use RHIPE is much easier if your filesystem layout (i.e location of R, Hadoop, libraries etc) is identical across all computers.

## 1.1 Tests

In R

```
library(Rhipe)
```

should work successfully.

```
rhwrite(list(1,2,3),"/tmp/x")
```

should successfully write the list to the HDFS

```
rhread("/tmp/x")
```

should return a list of length 3 each element a list of 2 objects.

and a quick run of this should also work

```
map <- expression({
  lapply(seq_along(map.values),function(r){
    x <- runif(map.values[[r]])
    rhcollect(map.keys[[r]],c(n=map.values[[r]],mean=mean(x),sd=sd(x)))
  })
})
## Create a job object
z <- rhmr(map, ofolder="/tmp/test", inout=c('lapply','sequence'),
          N=10,mapred=list(mapred.reduce.tasks=0),jobname='test')
```

```
## Submit the job
rhex(z)
## Read the results
res <- rhread('/tmp/test')
colres  <- do.call('rbind', lapply(res,"[[",2))
colres
       n      mean        sd
 [1,]  1 0.4983786        NA
 [2,]  2 0.7683017 0.2937688
 [3,]  3 0.5936899 0.3425441
 [4,]  4 0.3699087 0.2666379
 [5,]  5 0.5179839 0.4060244
 [6,]  6 0.6278925 0.2952608
 [7,]  7 0.4920088 0.2785893
 [8,]  8 0.4592598 0.2674592
 [9,]  9 0.5734197 0.1928496
[10,] 10 0.4942676 0.2989538
```

CHAPTER

TWO

# INTRODUCTION

Massive data sets have become commonplace today. Powerful hardware is readily available with a terabyte of hard drive storage costing less than $150 and computers with many cores a norm. Today, the moderately adventurous scientist can connect two computers to form a distributed computing platform. Languages and software tools have made concurrent and distributed computing accessibly to the statistician.

It is important to stress that a massive data set is not just a single massive entity that needs to be stored across multiple hard drives but rather the size of the data created during the steps of an analysis. A 'small' 14 GB data set can easily become 190 GB as new data structures are created, or where multiple subsets /transformations are each saved as different data sets. Large data sets can come as they are or grow big because of the nature of the analysis. No analyst wants her research to be restricted because the computing infrastructure cannot keep up with the size or complexity.

## 2.1 Hadoop

Hadoop is an open source programming framework for distributed computing with massive data sets using a cluster of networked computers. It has changed the way many web companies work, bringing cluster computing to people with little knowledge of the intricacies of concurrent/distributed programming. Part of the reason for its success is that it has a fixed programming paradigm. It somewhat restricts what the user can parallelize but once an algorithm has been written the 'MapReduce way', concurrency and distribution over a cluster comes for free.

It consists of two components: the Hadoop Distributed Filesystem and Hadoop MapReduce. They are based on the Google Filesystem and Google MapReduce respectively. Companies using these include Amazon, Ebay, New York Times, Facebook to name a few. The software can be downloaded from here .

## 2.2 Hadoop Distributed Filesystem

The Hadoop Distributed Filesystem (HDFS) sits on top of the file system of a computer (called the local filesystem). It pools the hard drive space of a cluster or heterogenous computers (e.g. different hardware and operating systems) and provides a unified view to the user. For example, with a cluster of 10 computers each with 1TB hard drive space available to Hadoop, the HDFS provides a user 10 TB of hard drive space. A single file can be bigger than maximum size on the local filesystem e.g. 2TB files can be saved on the HDFS. The HDFS is catered to large files and high throughput reads. Appends to files are not allowed. Files written to the HDFS are chunked into blocks, each block is replicated and saved on different cluster computers. This provides a measure of safety in case of transient or permanent computer failures. When a file is written to the HDFS, the client contacts the *Namenode*, a computer that serves as the gateway to the HDFS. It also performs a lot of administrative tasks, such as saving the mapping between a file and the location of its block across the cluster and so on. The Namenode tells the client which Datanodes (the computers that make up the HDFS) to store the data onto. It also tells the client which Datanodes to read the data from when a read request is performed. See (*A schematic of the Hadoop File System*) for an graphical outline of the file copy operation to the HDFS.

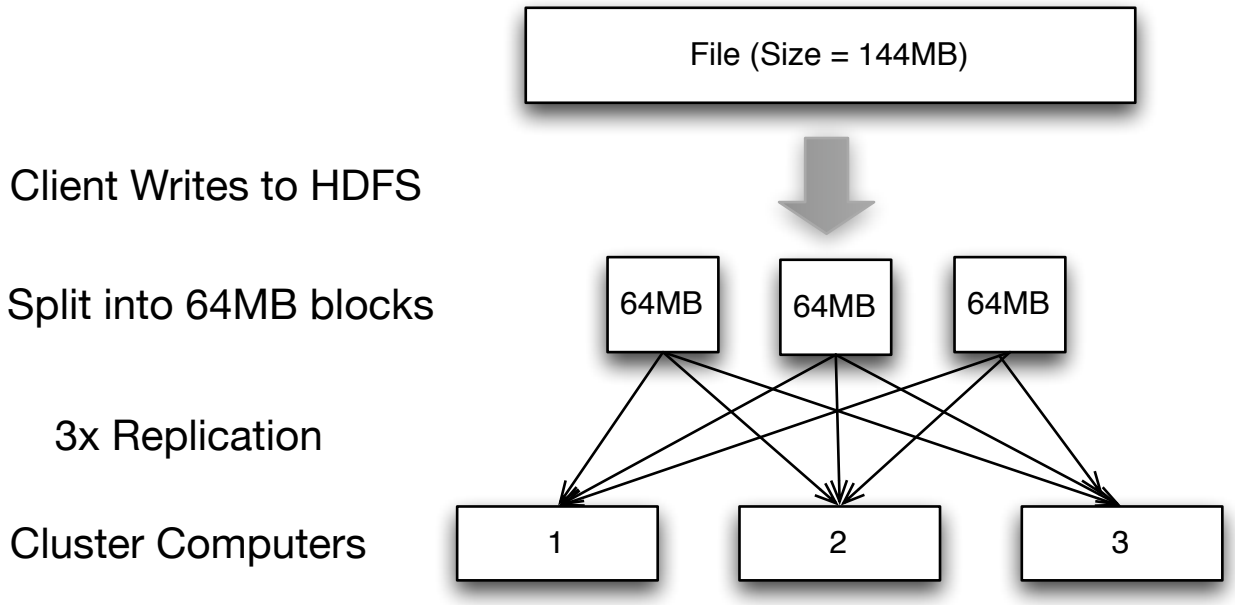


Figure 2.1: A schematic of the Hadoop File System

## 2.3 Hadoop MapReduce

Concurrent programming is difficult to get right. As Herb Sutter put it:

> ... humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code.

A statistician attempting concurrent programming needs to be aware of race conditions, deadlocks and tools to prevent this: locks, semaphores, and mutually exclusive regions etc. An approach suggested by Sutter et al ([STLa]) is to provide programming models not functions that force the programmer to approach her algorithms differently. Once the programmer constructs the algorithm using this model, concurrency comes for free. The MapReduce programming model is one example. Correctly coded Condor DAGS are another example.

MapReduce ([MapRed]) consists of several embarrassingly parallel splits which are evaluated in parallel. This is called the Map. There is a synchronization guard where intermediate data created at the end of the Map is exchanged between nodes and another round of parallel computing starts, called the Reduce phase. In effect large scale simulation trials in which the programmer launches several thousands of independent computations is an example of a Map. Retrieving and collating the results (usually done in the R console) is an example of a manual reduce.

In detail, the input to a MapReduce computation is a set of $N$ *key,value* pairs. The $N$ pairs are partitioned into $S$ arbitrary *splits*. Each split is a unit of computation and is assigned to one computing unit on the cluster. Thus the processing of the $S$ splits occurs in parallel. Each split is processed by a user given function $M$, that takes a sequence of input key,value pairs and outputs (one or many) intermediate key,value pairs. The Hadoop framework will partition the intermediate values by the intermediate key. That is intermediate values sharing the same intermediate key are grouped together. Once the map is complete, the if there are $M$ distinct intermediate keys, a user given function $R$, will be given an intermediate key and all intermediate values associated with the same key. Each processing core is assigned a subset of intermediate keys to reduce and the reduction of the $M$ intermediate keys occurs in parallel. The function $R$, takes an intermediate key, a stream of associated intermediate values and returns a final key,value pair or pairs.

The R programmer has used MapReduce ideas. For example, the `tapply` command splits a vector by a list of factors. This the map equivalent: each row of the vector is the value and the keys are the distinct levels of the list of factors.

The reduce is the user given function applied to the partitions of the vector. The `xyplot` function in `lattice` takes a formula e.g. $F \sim Y|A * B$, subsets the the data frame by the cartesian product of the levels of $A$ and $B$ (the map) and displays each subset (the reduce). Hadoop MapReduce generalizes this to a distributed level.

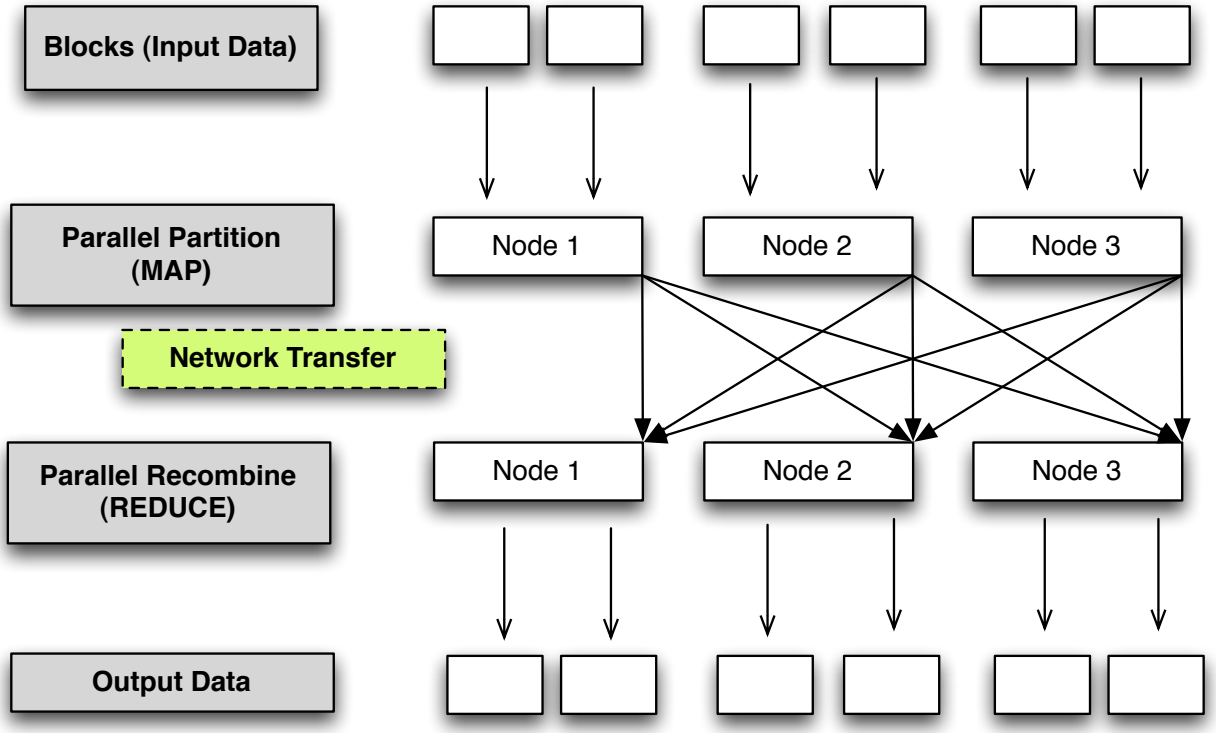


Figure 2.2: An overview of Hadoop MapReduce

### 2.3.1 Examples

Two examples, one for quantiles and another for correlation will be described.

#### Approximate Quantile

Let $X$ be a column of numbers. This can be arbitrarily large (e.g. hundreds of gigabytes). The objective is to find the quantiles of the $X$. Let each number be a key. For discrete data e.g. ages (rounded to years), count data, the number of unique numbers in a data set is generally not large. For continuous data it can be many billions. In this case, we need to discretize this. Care is needed before discretization. Discretions is equivalent to binning and reduces the number of unique data points. For example, do not round to the 5'th decimal place if the data points are the same for the first 5 decimal places!

For this example, let us assume the data is discrete (so no need for rounding). The goal is to compute the frequency table of the data and use that to compute the quantiles (see [HynFan])

```
for line in line_of_numbers:
   for number in tokenize line by [[:space:]]:
      write_key_value(number,1)
```

The input data is partitioned into a splits of many lines of numbers. The above code is applied to these splits in parallel. The intermediate keys are the unique numbers and each has a list of *1*'s associated with it. Hadoop will sort the keys

by the number (not neccassirly by the quantity of the number, it depends on the programming framework) and assign the aggregation computation of the associated values for the different unique keys to different processing cores in the reduce phase. The reduce logic is as

```
1 for number in stream_of_unique_numbers:
2   sum=0
3   while has_more_values?()==TRUE:
4     sum=sum+get_new_value()
5   end while
6   write_key_value(number, sum)
```

Notice the intermediate keys (the value of the number) and the final key (see last line above) are the same. The unique numbers are partitioned. Thus the stream in line 1 is stream of a subset. The different subsets are processed on different compute cores. Note, the reduce code sums the *1*'s in a *while* loop rather than loading them all into one gigantic array and adding the array. There can be too many *1*'s to fit into core. This where the MapReduce implementation shines: big data. The algorithm finally outputs the distinct numbers of $X$ and the counts. This can be sorted and used to compute the quantiles. This algorithm is also used to compute word frequencies for text document analysis.

### Correlation

To compute the correlation of a text file of *N* rows and *C* columns, we need the sum, sums of squares of each column and sum of unique pairs of columns. The intermediate keys and final keys are the same: the column and column pair identifiers. The value will be the sum of columns, their sum of squares,the cross products and the number of entries.

We need to iterate over lines, tokenize, and compute the relevant column sums and pairwise cross products.

```
1  for text_line in stream_of_text_lines
2     tokenized_line = tokenize text_line by [[:space:]]
3     for i=1 to C:
4        rhcollect( (i,i), (n=1,sum=tokenized_line[i],ssq=tokenized_line[i]^2))
5        for j=i+1 to C:
6            rhcollect( (i,j), (crossprod=
7                               tokenized_line[i]* tokenized_line[j]))
8        end for
9      end for
10 end for
```

The intermediate keys produced at the end of the map nodes are pairs $(i, j)$ where $1 \leq i \leq C, i \leq j \leq C$ . The values are the original value (the value for row *i* and column *j*), its square and crossproduct. The *n* is just a *1*. By adding the values for this we obtain the total number of rows. Inserting this *1* is wasteful, since it is redundantly being passed around for all the keys - we could compute the number of rows in another MapReduce job.

We need to sum this:

```
1  for identifier in stream_of_identifiers:
2      ## identifier is a colum pair
3      sum = empty tuple
4      while has_more_values?()==TRUE:
5          sum = sum + get_new_value()
6          # get_new_value() will return
7          # a tuple of length
8          #     = 3 (if identifier is (i,i)
9          #     = 1 (if identifier is (i,j) i <> j
10     end while
11     rhcollect(identifier, sum)
```

The output will be a set of pairs or triplets according as the key is $(i, i)$ or $(i, j)$.

### Computing an Inner Join

We have a text file **A** of *N* columns and **B** with *M* columns. Both have a common column **index**. In **A**, it is unique, with one row per level of **index**. **B** is a repeated measurement data set, with the levels of **index** repeated many times. **B** also contains a column called **weight**. We need to compute the following operation, for every unique value of **index** in **A**, compute the mean value of **weight**. We need to join the two data sets on **index** and compute the number of observations and sum of **weight** for unique values **index** and save a data set which consist of only those values of **index** found in **A**.

In SQL, this is

```
select
       index,
       mean(weight) as mweight
from A inner join B
on A.index=B.index
group by A.index
```

1. Compute the number of observations and sum of weights aggregated by levels of **index** for **B**. This computes the values for all levels of **index** in **B** not just those found in **A**. Call this computed data set **B'**

2. Merge **B'** and **A** .This might be wasteful since we compute for all levels in **B**, however if the summarized data set **B'** will be used often, then this is a one time cost.

Summarizing **B** to **B'**

```
#map
for line in lines:
    index  =  get column corresponding to "index" from line
    weight =  get column corresponding to "weight" from line
    rhcollect(index,(1,weight))

#reduce
for index in stream_of_indices:
    ## identifier is a colum pair
    sum = empty tuple
    while has_more_values?()==TRUE:
        sum = sum + get_new_value()
    rhcollect(index, sum) #total number of obs, sum of weight
```

To merge, we map each index value found in **A** to a TRUE value and each value found in **B'** to a tuple (number of observations and sum of weights). If a value of **index** is found in *both* there will be two intermediate values for that value of **index**. If instead the value of **index** exists in exactly one of **A** and **B** there will be exactly one intermediate value. If there are two values, one is a dummy (the TRUE) and the pseudo-code retains the value whose length is 2 (the tuple).

```
for index in stream_of_indices:
    count = 0
    information = NULL
    while has_more_values?()==TRUE:
        count = count + 1
        temp = get_next_value()
        if length of temp == 2:
            information = temp
    end while
    if count == 2:
        rhcollect(index, information)
```

### 2.3.2 Combiners : An Optimization

The Hadoop framework, sends all the intermediate values for a given key to the reducer. The intermediate values for a given key are located on several compute nodes and need to be shuffled (sent across the network) to the node assigned the processing of that intermediate key. This involves a lot of network transfer. Some operations do not need access to all of the data (intermediate values) i.e they can compute on subsets and order does not matter i.e associative and commutative operations. For example, the minimum of 8 numbers $min(x_1, x_2, \ldots, x_n) = min(min(x_1, x_2), min(x_3, \ldots, x_5), min(x_6, \ldots, x_8))$

The reduction occurs on just after the map phase on a subset of intermediate values for a given intermediate keys. The output of this is sent to the reducer. This greatly reduces network transfer and accelerates the job speed.

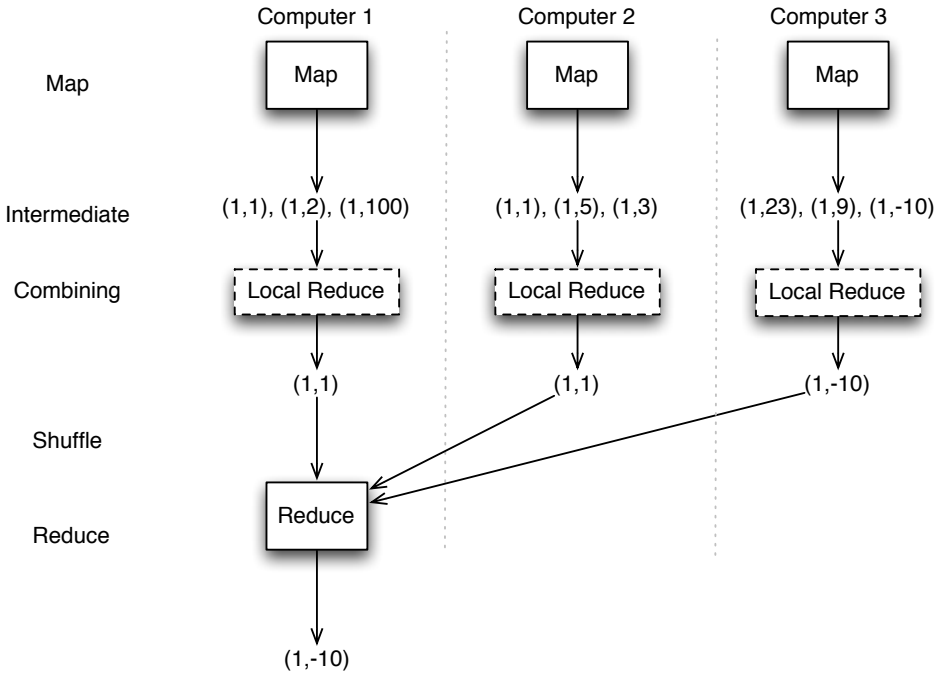


Figure 2.3: A MapReduce job using a combiner (the minimum operator). We consider intermediate values for a single key.

The examples in this section outlined some algorithms that work with MapReduce. Using RHIPE, there are ways to optimize the above code e.g. instead of processing one line at a time use vector operations. Also, RHIPE calls the code with R lists containing the input the keys and values. The streams in the Reduce are replaced by lists of intermediate values and the R code is called repeatedly with the list filled with new elements. This will be explained in the Airline data example (see *Airline Dataset*)

## 2.4 R and Hadoop Integrated Programming Environment

The R and Hadoop Integrated Programming Environment is R package to compute across massive data sets, create subsets, apply routines to subsets, produce displays on subsets across a cluster of computers using the Hadoop DFS and Hadoop MapReduce framework. This is accomplished from within the R environment, using standard R programming idioms. For efficiency reasons, the programming style is slightly different from that outlined in the previous section.

The native language of Hadoop is Java. Java is not suitable for rapid development such as is needed for a data analysis environment. Hadoop Streaming bridges this gap. Users can write MapReduce programs in other languages e.g.

Python, Ruby, Perl which is then deployed over the cluster. Hadoop Streaming then transfers the input data from Hadoop to the user program and vice versa.

Data analysis from R does not involve the user writing code to be deployed from the command line. The analyst has massive data sitting in the background, she needs to create data, partition the data, compute summaries or displays. This need to be evaluated from the R environment and the results returned to R. Ideally not having to resort to the command line.

RHIPE is just that.

- RHIPE consist of several functions to interact with the HDFS e.g. save data sets, read data created by RHIPE MapReduce, delete files.
- Compose and launch MapReduce jobs from R using the command `rhmr` and `rhex`. Monitor the status using `rhstatus` which returns an R object. Stop jobs using `rhkill`
- Compute *side effect* files. The output of parallel computations may include the creation of PDF files, R data sets, CVS files etc. These will be copied by RHIPE to a central location on the HDFS removing the need for the user to copy them from the compute nodes or setting up a network file system.
- Data sets that are created by RHIPE can be read using other languages such as Java, Perl, Python and C. The serialization format used by RHIPE (converting R objects to binary data) uses Googles Protocol Buffers which is very fast and creates compact representations for R objects. Ideal for massive data sets.
- Data sets created using RHIPE are *key-value* pairs. A key is mapped to a value. A MapReduce computations iterates over the key,value pairs in parallel. If the output of a RHIPE job creates unique keys the output can be treated as a external-memory associative dictionary. RHIPE can thus be used as a medium scale (millions of keys) disk based dictionary, which is useful for loading R objects into R.

**Future Work**

I plan on incorporating input and output bridges between RHIPE and HBase.

In summary, the objective of RHIPE is to let the user focus on thinking about the data. The difficulties in distributing computations and storing data across a cluster are automatically handled by RHIPE and Hadoop.

CHAPTER

THREE

# AIRLINE DATASET

The Airline data set consists of flight arrival and departure details for all commercial flights from 1987 to 2008. The approximately 120MM records (CSV format), occupy 120GB space. The data set was used for the Visualization Poster Competition, JSM 2009. The winning entries can be found here . To quote the objectives

> "The aim of the data expo is to provide a graphical summary of important features of the data set. This is intentionally vague in order to allow different entries to focus on different aspects of the data, but here are a few ideas to get you started:
>
> - When is the best time of day/day of week/time of year to fly to minimise delays?
> - Do older planes suffer more delays?
> - How does the number of people flying between different locations change over time?
> - How well does weather predict plane delays?
> - Can you detect cascading failures as delays in one airport create delays in others? Are there critical links in the system?"

In this chapter, I will demonstrate RHIPE code samples to create similar graphics found in the winning entries [SAS] and [FLUSA]

## 3.1 Copying the Data to the HDFS (or a *Distributed Downloader*!)

The Airline data can be found at this site . In this example, we download the data sets for the individual years and save them on the HDFS with the following code (with limited error checks)

```
library(Rhipe)
map <- expression({
  msys <- function(on){
    system(sprintf("wget  %s --directory-prefix ./tmp 2> ./errors",on))
    if(length(grep("(failed)|(unable)",readLines("./errors")))>0){
      stop(paste(readLines("./errors"),collapse="\n"))
    }}

  lapply(map.values,function(x){
    x=1986+x
    on <- sprintf("http://stat-computing.org/dataexpo/2009/%s.csv.bz2",x)
    fn <- sprintf("./tmp/%s.csv.bz2",x)
    rhstatus(sprintf("Downloading %s", on))
    msys(on)
    rhstatus(sprintf("Downloaded %s", on))
    system(sprintf('bunzip2 %s',fn))
```

```
17      rhstatus(sprintf("Unzipped %s", on))
18      rhcounter("FILES",x,1)
19      rhcounter("FILES","_ALL_",1)
20    })
21  })
22  z <- rhmr(map=map,ofolder="/airline/data",inout=c("lapply"), N=length(1987:2008),
23            mapred=list(mapred.reduce.tasks=0,mapred.task.timeout=0),copyFiles=TRUE)
24  j <- rhex(z,async=TRUE)
```

A lot is demonstrated in this code. RHIPE is loaded via the call in line 1. A MapReduce job takes a set of input keys, in this case the numbers 1987 to 2008. It also takes a corresponding set of values. The parameter `inout` in line 22 tells RHIPE how to convert the input the data to key, value pairs. If the input file is a binary file but `inout` specifies *text* as the input, RHIPE will not throw an error but provide very unexpected key,value pairs. `inout` in this case is *lapply*, which treats the numbers 1 to `N` (in line 22) as both keys and values.

These *N* key,value pairs are partitioned into *splits*. How they are partitioned depends on the value of `inout[1]`. For text files (`inout[1]='text'`), the data is split into roughly equi-length blocks of e.g. 128MB each. A CSV text file will have approximately equal number of lines per block (not necessarily). RHIPE will launch R across all the compute nodes. Each node is responsible for processing a the key,value pairs in its assigned splits.

This processing is performed in the `map` argument to `rhmr`. The `map` argument is an R expression. Hadoop will read key,value pairs, send them to RHIPE which in turn buffers them by storing them in a R list: *map.values* and *map.keys* respectively. Once the buffer is full, RHIPE calls the `map` expression. The default length of *map.values* (and *map.keys*) is 10,000 [1].

In our example, N is 22. The variables *map.values* and *map.keys* will be lists of numbers 1 to 22 and strings "1" to "22" respectively. The entries need not be in the order 1 to 22.

`rhmr` is a call that packages the MapReduce job which is sent to Hadoop. It takes an input folder which can contain multiple files and subfolders. All the files will be given as input. If a particular file cannot be understood by the input format (e.g. a text file given to `inout[1]='sequence'`), RHIPE will throw an error.

The expression downloads the CSV file, unzips its, and stores in the folder *tmp* located in the current directory. No copying is performed. The current directory is a temporary directory on the local filesystem of the compute node, **not** on the HDFS. Upon successful completion of the split, the files stored in *tmp* (of the current directory) will be copied to the output folder specified by `ofolder` in the call to `rhmr`. Files are copied **only if** `copyFiles` is set to TRUE (in line 23).

Once a file has been downloaded, we inform Hadoop of our change in status, via `rhastatus`. The figure *Example of rhstatus* displays the various status of each of the 22 splits (also called Tasks)

Once a file has been downloaded, we increment a **distributed count**. Counts belong to families, a single family contains many counters. The counter for group *G* and name *N* is incremented via a call to `rhcounter`. We increment a counter for each of the 22 files. Since each file is downloaded once, this is essentially a flag to indicate successful download. A count of files downloaded is tracked in *Files/_ALL_* .

The operation of Hadoop is affected by many options, some of which can be found in *Options For RHIPE*. Hadoop will terminate splits (Tasks) after 10 minutes if they do not invoke `rhstatus` or return. Since each download takes approximately 30 minutes (the minimum is 4 minutes, the maximum is 42 minutes, the mean is 30 minutes), Hadoop will kill the tasks. We tell Hadoop to not kill long running tasks by setting *mapred.task.timeout* to 0. We do not to need to reduce our results so we set *mapred.reduce.tasks* to 0. Output from the map is written directly to the output folder on the HDFS. We do not have any output. These options are passed in the `mapred` argument.

The call to `rhex` launches the job across Hadoop. We use the `async` argument to return control of the R console to the user. We can monitor the status in two ways

- Print the return value of `rhex`. The name of the job can be changed by giving a value to `jobname` in the call to `rhmr`. The same information can be found at the Hadoop job tracker.

[1] This can be changed by the user, see *Options For RHIPE*.

**Hadoop map task list for job_201007281701_0051 on spica**

**All Tasks**

| Task | Complete | Status | Start Time | Finish Time | Errors | Counters |
|---|---|---|---|---|---|---|
| task_201007281701_0051_m_000000 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1987.csv.bz2 | 28-Jul-2010 22:25:41 | 28-Jul-2010 22:30:38 (4mins, 57sec) | | 7 |
| task_201007281701_0051_m_000001 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1988.csv.bz2 | 28-Jul-2010 22:25:41 | 28-Jul-2010 22:44:12 (18mins, 31sec) | | 7 |
| task_201007281701_0051_m_000002 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1989.csv.bz2 | 28-Jul-2010 22:25:41 | 28-Jul-2010 22:44:07 (18mins, 26sec) | | 7 |
| task_201007281701_0051_m_000003 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1990.csv.bz2 | 28-Jul-2010 22:25:42 | 28-Jul-2010 22:45:55 (20mins, 13sec) | | 7 |
| task_201007281701_0051_m_000004 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1991.csv.bz2 | 28-Jul-2010 22:25:42 | 28-Jul-2010 22:45:48 (20mins, 5sec) | | 7 |
| task_201007281701_0051_m_000005 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1992.csv.bz2 | 28-Jul-2010 22:25:42 | 28-Jul-2010 22:45:32 (19mins, 50sec) | | 7 |
| task_201007281701_0051_m_000006 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1993.csv.bz2 | 28-Jul-2010 22:25:44 | 28-Jul-2010 22:45:37 (19mins, 53sec) | | 7 |
| task_201007281701_0051_m_000007 | 100.00% | Unzipped http://stat-computing.org/dataexpo/2009/1994.csv.bz2 | 28-Jul-2010 22:25:44 | 28-Jul-2010 22:45:27 (19mins, 43sec) | | 7 |
| task_201007281701_0051_m_000008 | 100.00% | Downloading http://stat-computing.org/dataexpo/2009/1995.csv.bz2 | 28-Jul-2010 22:25:44 | | | 3 |
| task_201007281701_0051_m_000009 | 100.00% | Downloading http://stat-computing.org/dataexpo/2009/1996.csv.bz2 | 28-Jul-2010 22:25:44 | | | 3 |

Figure 3.1: Example of `rhstatus`

```
> j
RHIPE Job Token Information
---------------------------
URL: http://spica:50030/jobdetails.jsp?jobid=job_201007281701_0053
Name: 2010-07-28 23:33:44
ID: job_201007281701_0053
Submission Time: 2010-07-28 23:33:45
State: RUNNING
Duration(sec): 11.702
Progess
       pct numtasks pending running complete failed
map      0       22       1      21        0      0
reduce   0        0       0       0        0      0

> j
RHIPE Job Token Information
---------------------------
URL: http://spica:50030/jobdetails.jsp?jobid=job_201007281701_0053
Name: 2010-07-28 23:33:44
ID: job_201007281701_0053
Submission Time: 2010-07-28 23:33:45
State: RUNNING
Duration(sec): 56.417
Progess
       pct numtasks pending running complete failed
map      1       22       0      22        0      0
reduce   0        0       0       0        0      0
```

- By calling `rhstatus`, giving it the value returned from `rhex` or the job ID (e.g. *job_201007281701_0053*).

```
> a <- rhstatus(j) ## or rhstatus("job_201007281701_0053")
> a$state
[1] "RUNNING"
> a$duration
[1] 902.481
> a$counters
$counters
```

```
$counters$`Job Counters `
Launched map tasks
                22

$counters$FileSystemCounters
   FILE_BYTES_READ HDFS_BYTES_WRITTEN
         127162942          127162942

$counters$`"FILES"`
 1987.0 "_ALL_"
      1       1

$counters$`Map-Reduce Framework`
 Map input records    Spilled Records Map output records
                22                  0                  0

$counters$job_time
[1] 902.481
```

This distributed download took 45 minutes to complete, 15 seconds more than the longest running download (2007.csv.bz2). A sequential download would have taken several hours.

**Note:** It is important to note that the above code is mostly boiler plate. There is almost no lines to handle distribution across a cluster or task restart in case of transient node failure. The user of RHIPE need only consider how to frame her argument in the concepts of MapReduce.

## 3.2 Converting to R Objects

The data needs to be converted to R objects. Since we will be doing repeated analyses on the data, it is better to spend time converting them to R objects making subsequent computations faster, rather than tokenizing strings and converting to R objects for every analysis.

A sample of the text file

```
1987,10,23,5,1841,1750,2105,2005,PS,1905,NA,144,135,NA,60,51,LAX,SEA,954,NA,NA,0,NA,0,...
1987,10,24,6,1752,1750,2010,2005,PS,1905,NA,138,135,NA,5,2,LAX,SEA,954,NA,NA,0,NA,0,...
...
...
```

The meaning of the columns can be found here . Rather than store the entire 120MM rows as one big data frame, it is efficient to store it as rectangular blocks of $R$ rows and $M$ columns. We will not store all the above columns only the following:

- Dates: day of week, date, month and year (1,2,3, and 4)
- Arrival and departure times: actual and scheduled (5,6,7 and 8)
- Flight time: actual and scheduled (12 and 13)
- Origin and Destination: airport code, latitude and longitude (17 and 18)
- Distance (19)
- Carrier Name (9)

Since latitude and longitude are not present in the data sets, we will compute them later as required. Carrier names are located in a different R data set which will be used to do perform carrier code to carrier name translation.

We will store the data set as blocks of $5000 \times 5$ rows and columns. These will be the values. Every value must be mapped to a key. In this example, the keys (indices) to these blocks will not have any meaning but will be unique.

The key is the first scheduled departure time. The format of the data is a *Sequence File*, which can store binary representations of R objects.

```
setup <- expression({
  convertHHMM <- function(s){
    t(sapply(s,function(r){
      l=nchar(r)
      if(l==4) c(substr(r,1,2),substr(r,3,4))
      else if(l==3) c(substr(r,1,1),substr(r,2,3))
      else c('0','0')
    })
  )}
})
map <- expression({
  y <- do.call("rbind",lapply(map.values,function(r){
    if(substr(r,1,4)!='Year') strsplit(r,",")[[1]]
  }))
  mu <- rep(1,nrow(y));yr <- y[,1]; mn=y[,2];dy=y[,3]
  hr <- convertHHMM(y[,5])
  depart <- ISOdatetime(year=yr,month=mn,day=dy,hour=hr[,1],min=hr[,2],sec=mu)
  hr <- convertHHMM(y[,6])
  sdepart <- ISOdatetime(year=yr,month=mn,day=dy,hour=hr[,1],min=hr[,2],sec=mu)
  hr <- convertHHMM(y[,7])
  arrive <- ISOdatetime(year=yr,month=mn,day=dy,hour=hr[,1],min=hr[,2],sec=mu)
  hr <- convertHHMM(y[,8])
  sarrive <- ISOdatetime(year=yr,month=mn,day=dy,hour=hr[,1],min=hr[,2],sec=mu)
  d <- data.frame(depart= depart,sdepart = sdepart
                  ,arrive = arrive,sarrive =sarrive
                  ,carrier = y[,9],origin = y[,17]
                  ,dest=y[,18],dist = y[,19], year=yr, month=mn, day=dy
                  ,cancelled=y[,22], stringsAsFactors=FALSE)
  d <- d[order(d$sdepart),]
  rhcollect(d[c(1,nrow(d)),"sdepart"],d)
})
reduce <- expression(
    reduce = {
      lapply(reduce.values,function(i)
             rhcollect(reduce.key,i))}
    )
mapred <- list(rhipe_map_buff_size=5000)
z <- rhmr(map=map,reduce=reduce,setup=setup,inout=c("text","sequence")
          ,ifolder="/airline/data/",ofolder="/airline/blocks",mapred=mapred,orderby="numeric")
rhex(z)
```

The `setup` expression is loaded into the R session *once* for every split. Remember a split can consist of many *map.values* that need to be processed. For text files as input, a split is 128MB or whatever your Hadoop block size is. Lines 12-14, iterate over the lines and tokenizing them. The first line in each downloaded file is the column year which must be ignored (see line 13). The lines of text are aggregated using `rbind` and time related columns converted to *datetime* objects. The data frame is sorted by scheduled departure and saved to disk indexed by the range of scheduled departures in the data frame. The size of the value (data frame) is important. RHIPE will can write any sized object but cannot read key,values that are more than 256MB. A data frame of 5000 rows and 8 columns fits very well into 256MB. This is passed to Hadoop in line 37.

Running R across massive data can be illuminating. Without the calls to `ISOdatetime`, it is **much** faster to complete.
**Sorted keys** A reduce is not needed in this example. The text data is blocked into data frames and written to disk. With 128MB block sizes and each block a split, each split being mapped by one R session, there 96 files each containing several data frames. The reduce expression writes each incoming intermediate value (a data frame) to disk. This is called an *identity reducer* which can be used for

1. For map file indexing. The intermediate keys are sorted. In the identity reduce, these keys are written to disk in sorted order. If the output format (`inout[2]`) is *map*, the output can be used as an external memory hash table. Given a key, RHIPE can use Hadoop to very quickly discover the location of the key in the sorted (by key) output data and return the associated value. Thus even when no reduce logic is required the user can provide the identity reduce to create a queryable Map File from the map output.

2. Intermediate keys are sorted. But they can be sorted in different ways. RHIPE's default is *byte ordering* i.e the keys are serialized to bytes and sorted byte wise. However, byte ordering is very different from semantic ordering. Thus keys e.g. 10,-1,20 which might be byte ordered are certainly not numerically ordered. RHIPE can numerically order keys so that in the reduce expression the user is guaranteed to receive the keys in sorted numeric order. In the above code, we request this feature in line 38. Numeric sorting is as follows: keys A and B are ordered if $A < B$ and of unit length or or $A[i] < B[i], 1 \leq i \leq min(length(A), length(B))$ [2]. For keys $1, (2,1), (1,1), 5, (1,3,4), (2,1), 4, (4,9)$ the ordering is $1, (1,1), (1,3,4), (2,1), (2,1), 4, (4,9), 5$ Using this ordering, all the values in a given file will be ordered by the range of the scheduled departures. Using this custom sorter can be slower than the default byte ordering. Bear in mind, the keys in a *part* file will be ordered but keys in one *part* file need not be less than those in another *part* file. To achieve ordering of keys set *orderby* in the call to `rhmr` to one of *bytes* (default), *integer*, *numeric* (for doubles) or *character* (alphabetical sorting) in the `mapred` argument to `rhmr`. If the output format is *sequence*, you also need to provide a reducer which can be an identity reducer. Note, if your keys are discrete, it is best to use *integer* ordering. Values of NA can throw of ordering and will send *all* key,values to one reducer causing a severe imbalance.

```
reduce = expression({
  reduce={ lapply(reduce.values,function(r) rhcollect(reduce.key,r)) }
})
```

3. To decrease the number of files. In this case decreasing the number of files is hardly needed, but it can be useful if one has more thousands of splits.

In situations (1) and (3), the user does not have to provide the R reduce expression and can leave this parameter empty. In situation (2), you need to provide the above code. Also, (2) is incompatible with Map File outputs (i.e `inout[2]` set to *map*). Case (2) is mostly useful for time series algorithms in the reduce section e.g. keys of the form $(identifier, i)$‘ where *identifier* is an object and *i* ranges from *1* to $n_{identifier}$. For each key, the value is sorted time series data. The reducer will receive the values for the keys $(identifier, i)$‘ in the order of *i* for a given *identifier*. This also assumes the user has partitioned the data on *identifier* (see the `part` parameter of `rhmr`: for this to work, all the keys $(identifier, i)$‘ with the same *identifier* need to be sent to the same reducer). For an example see *Streaming Data?*.

A sample data frame (last 4 columns removed):

```
                 depart              sarrive      carrier origin dest dist cancelled
2880 1988-05-01 01:02:01 ... 1988-05-01 01:59:01      DL    SLC  SEA  689         0
3770 1988-05-01 01:10:01 ... 1988-05-01 02:13:01      DL    JAX  FLL  318         0
2137 1988-05-01 01:10:01 ... 1988-05-01 01:59:01      DL    TPA  PBI  174         0
```

## 3.3 Demonstration of using Hadoop as a Queryable Database

*Sightly artificial*: store all Southwest Airlines information indexed by year,month,and day. Each (year, month, day) triplet will have all flight entries that left on that day. Using the above data set as the source, the Southwest lines are selected and sent to the reducer with the (year, month,day) key. All flights with the same (year, month) will belong to the same file. Given a (year, month,day) triplet, we can use the Map File output format to access the associated flight information in seconds rather than subsetting using MapReduce.

```
1 map <- expression({
2   h <- do.call("rbind",map.values)
3   d <- h[h$carrier=='WN',,drop=FALSE]
```

[2] A similar ordering exists for character vectors (`NA` not.. allowed). Specify by setting `orderby="character"` in the call to `rhmr`

```
4     if(nrow(d)>0){
5       e <- split(d,list(d$year,d$month,d$mday))
6       lapply(e,function(r){
7         k <- as.vector(unlist(r[1,c("year","month","mday")]))   ## remove attributes
8         rhcollect(k, r)
9       })
10    }
11  })
12  reduce <- expression(
13      pre = { collec <- NULL },
14      reduce = {
15        collec <- rbind(collec, do.call("rbind",reduce.values))
16        collec <- collec[order(collec$depart),]
17      },
18      post = {
19        rhcollect(k, collec)
20      }
21      )
22  z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","map")
23            ,ifolder="/airline/blocks/",ofolder="/airline/southwest"
24            ,mapred=list(rhipe_map_buff_size=10))
25  rhex(z)
```

Attributes are removed in line 8, for otherwise we have to retrieve a data frame with a data frame with column names and row names instead of a more convenient numeric vector. The map expression combines the individual data frames. Each data frame has 5000 rows, hence *rhipe_map_buff_size* is set to 10 for a combined data frame of 50000 rows in line 32. This is crucial. The default value for *rhipe_map_buff_size* is 10,000. Binding 10,000 data frames of 5000 rows each creates a data frame of 50MN rows - too unwieldy to compute with in R (for many types of operations). Data frames for Southwest Airlines (carried code=WN) are created and emitted with the call to `rhcollect` in line 15. These are combined in the reduce since data frames for the same (year, month,day) triplet can be emitted from different map expressions. Since this is associative and commutative we use a combiner. The output format (`inout[[2]]`) is *map*, so we can access the flights for any triplet with a call to `rhgetkey` which returns a list of key,value lists.

```
1  > a <- rhgetkey(list(c(88,2,17)),"/airline/southwest")
2  > a[[1]][[1]]
3  [1] 93  0  1
4  > head(a[[1]][[2]][,1:9])
5                      depart                  sarrive carrier origin dest dist cancelled
6  23648  1993-01-01 00:00:01 ... 1993-01-01 13:35:01      WN    RNO  LAS  345         1
7  20714  1993-01-01 07:20:01 ... 1993-01-01 08:40:01      WN    SFO  SAN  447         0
8  37642  1993-01-01 07:25:01 ... 1993-01-01 10:15:01      WN    OAK  PHX  646         0
9  316110 1993-01-01 07:30:01 ... 1993-01-01 08:30:01      WN    OAK  BUR  325         0
```

## 3.4 Analyses

We compute some summaries and displays to understand the data.

### 3.4.1 Top 20 cities by total volume of flights.

What are the busiest cities by total flight traffic. JFK will feature, but what are the others? For each airport code compute the number of inbound, outbound and all flights.

```
1  map <- expression({
2    a <- do.call("rbind",map.values)
```

```
  inbound <- table(a[,'origin'])
  outbound <- table(a[,'dest'])
  total <- table(unlist(c(a[,'origin'],a['dest'])))
  for(n in names(total)){
    inb <- if(is.na(inbound[n])) 0 else inbound[n]
    ob <- if(is.na(outbound[n])) 0 else outbound[n]
    rhcollect(n, c(inb,ob, total[n]))
  }
})
reduce <- expression(
    pre={sums <- c(0,0,0)},
    reduce = {
      sums <- sums+apply(do.call("rbind",reduce.values),2,sum)
    },
    post = {
      rhcollect(reduce.key, sums)
    }
    )
mapred$rhipe_map_buff_size <- 15
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/volume"
          ,mapred=mapred)
rhex(z,async=TRUE)
```

The code is straightforward. I increased the value of *rhipe_map_buff_size* since we are doing summaries of columns. The figure *Log of time to complete vs log of rhipe_map_buff_size.* plots the time of completion vs the mean of three trials for different values of *rhipe_map_buff_size*. The trials set *rhipe_map_buff_size* to 5,10,15,20,25 and 125. All experiments (like the rest in the manual) were performed on a 72 core cluster across 8 servers with RAM varying from 16 to 64 GB.

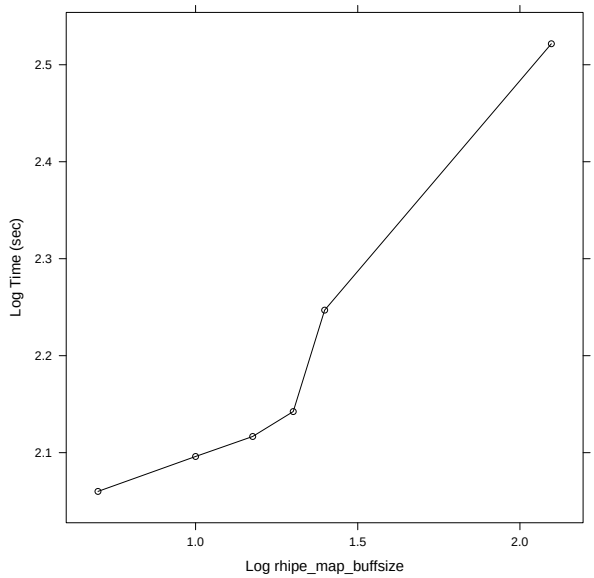


Figure 3.2: Log of time to complete vs log of *rhipe_map_buff_size*.

Read the data into R and display them using the `lattice` library.

```
counts <- rhread("/airline/volume")
aircode <- unlist(lapply(counts, "[[",1))
count <- do.call("rbind",lapply(counts,"[[",2))
results <- data.frame(aircode=aircode,
                      inb=count[,1],oub=count[,2],all=count[,3]
                      ,stringsAsFactors=FALSE)
results <- results[order(results$all,decreasing=TRUE),]
results$airport <- sapply(results$aircode,function(r){
nam <- ap[ap$iata==r,'airport']
if(length(nam)==0) r else nam
})
library(lattice)
r <- results[1:20,]
af <- reorder(r$airport,r$all)
dotplot(af~log(r[,'all'],10),xlab='Log_10 Total Volume',ylab='Airport',col='black')
```

There are 352 locations (airports) of which the top 20 serve 50% of the volume (see *Top 20 airports by volume of all flights.*)

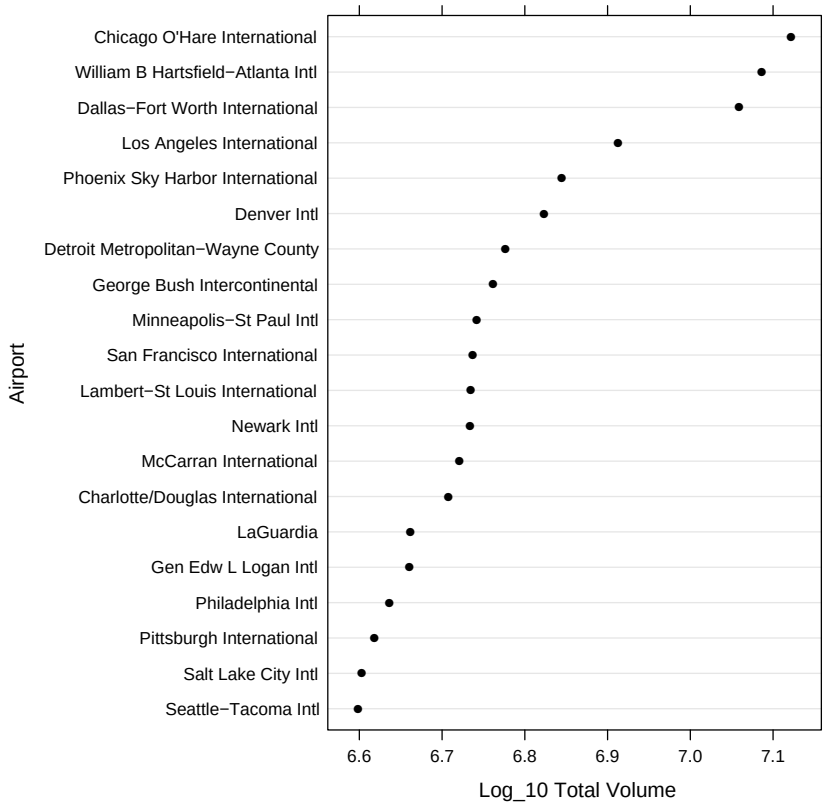


Figure 3.3: Top 20 airports by volume of all flights.

### 3.4.2 Carrier Popularity

Some carriers come and go, others demonstrate regular growth. In the following display, the log base 10 volume (total flights) over years are displayed by carrier. The carriers are ranked by their median volume (over the 10 year span).

As mentioned before, RHIPE is mostly boilerplate. Notice the similarities between this and previous examples (on a side note, to do this for 12GB of data takes 1 minute and 32 seconds across 72 cores and all the examples, except the download and conversion to R data frames, in the manual are less than 10 minutes)

```
## To create summaries
map <- expression({
  a <- do.call("rbind",map.values)
  total <- table(years=a[,'year'],a[,'carrier'])
  ac <- rownames(total)
  ys <- colnames(total)
  for(yer in ac){
    for(ca in ys){
      if(total[yer,ca]>0) rhcollect(c(yer,ca), total[yer,ca])
    }
  }
})
reduce <- expression(
    pre={sums <- 0},
    reduce = {sums <- sums+sum(do.call("rbind",reduce.values))},
    post = { rhcollect(reduce.key, sums) }
    )

mapred <- list()
mapred$rhipe_map_buff_size <- 5
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/carrier.pop"
          ,mapred=mapred)
z=rhex(z)
```

This is the RHIPE code to create summaries. We need to extract the data from Hadoop and create a display

```
a <- rhread("/airline/carrier.pop")
head(a)
[[1]]
[[1]][[1]]
[1] "90" "AA"

[[1]][[2]]
[1] 711825


[[2]]
[[2]][[1]]
[1] "90" "AS"

yr <- as.numeric(unlist(lapply(lapply(a,"[[",1),"[[",1)))
carrier <- unlist(lapply(lapply(a,"[[",1),"[[",2))
count <- unlist(lapply(a,"[[",2))
results <- data.frame(yr=yr,carcode=carrier,count=count,stringsAsFactors=FALSE)
results <- results[order(results$yr,results$count,decreasing=TRUE),]
carrier <- read.table("~/tmp/carriers.csv",sep=",",header=TRUE,
                      stringsAsFactors=FALSE,na.strings="XYZB")
results$carrier <- sapply(results$carcode,function(r){
  cd <- carrier[carrier$Code==r,'Description']
  if(is.na(cd)) r else cd
})
results$yr <- results$yr+1900
carr <- reorder(results$carrier,results$count, median)
xyplot(log(count,10)~yr|carr, data=results,xlab="Years", ylab="Log10 count",col='black'
```

```
         ,scales=list(scale='free',tck=0.5,cex=0.7),layout=c(2,8),type='l'
        ,par.strip.text = list(lines = 0.8,cex=0.7),cex=0.5,
        panel=function(...){
          panel.grid(h=-1,v=-1)
          panel.xyplot(...)
        })
```

The graph is displayed above.

### 3.4.3 Proportion of Flights Delayed

Does this proportion increase with time? Consider the display with proportion of flights delayed in a day across the years. Each year a panel. 22 panels. A flight is delayed if the delay is greater than 15 minutes.

It is clear that proportion increases in the holidays (the ends of the panels). The code for this comes after the figures.

```
map <- expression({
  a <- do.call("rbind",map.values)
  a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
  a <- a[!is.na(a$delay.sec),]
  a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
  e <- split(a,list(a$year,a$yday))
  lapply(e,function(r){
    n <- nrow(r); numdelayed <- sum(r$isdelayed)
    rhcollect(as.vector(unlist(c(r[1,c("year","yday")]))), c(n, numdelayed))
  })
})
reduce <- expression(
    pre={sums <- c(0,0)},
    reduce = {sums <- sums+apply(do.call("rbind",reduce.values),2,sum)},
    post = { rhcollect(reduce.key, sums) }
    )

mapred <- list()
mapred$rhipe_map_buff_size <- 5
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/delaybyyear"
          ,mapred=mapred)
z=rhex(z)

b <- rhread("/airline/delaybyyear")
y1 <- do.call("rbind",lapply(b,"[[",1))
y2 <- do.call("rbind",lapply(b,"[[",2))
results <- data.frame(year=1900+y1[,1],yday=y1[,2],
                      nflight=y2[,1],ndelay=y2[,2])
results$prop <- results$ndelay/results$nflight
results <- results[order(results$year,results$yday),]
```

*STL decomposition of proportion of flights delayed* is the STL decomposition of $p$ (the proportion of flights delayed). The *seasonal* panel clearly demonstrates the holiday effect of delays. They don't seem to be increasing with time (see *trend* panel).

```
prop <- results[,'prop']
prop <- prop[!is.na(prop)]
tprop <- ts(log(prop/(1-prop)),start=c(1987,273),frequency=365)
pdf("~/tmp/propdelayedxyplot.pdf")
plot(stl(tprop,s.window="periodic"))
dev.off()
```

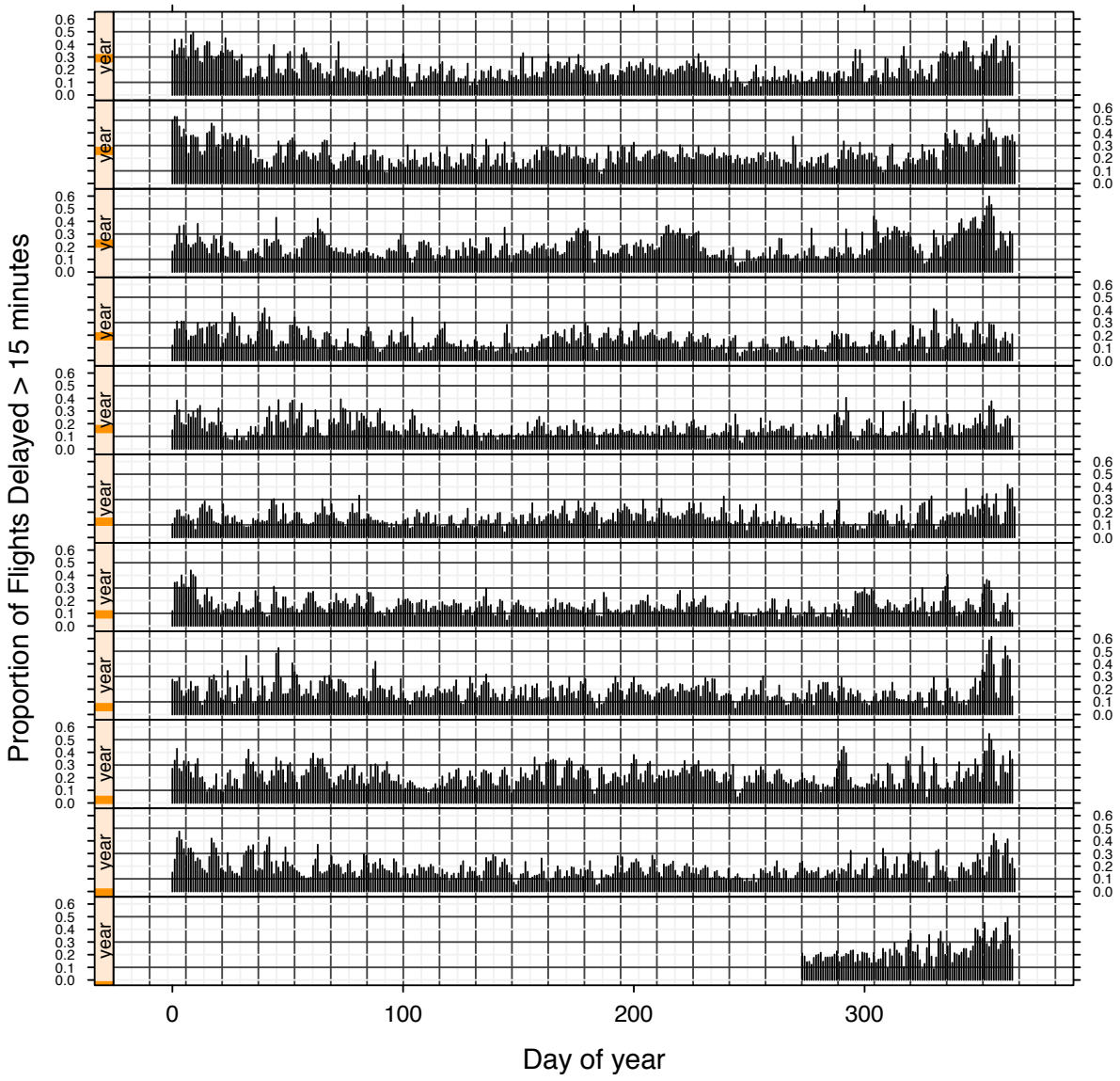


Figure 3.4: Proportion of flights delayed

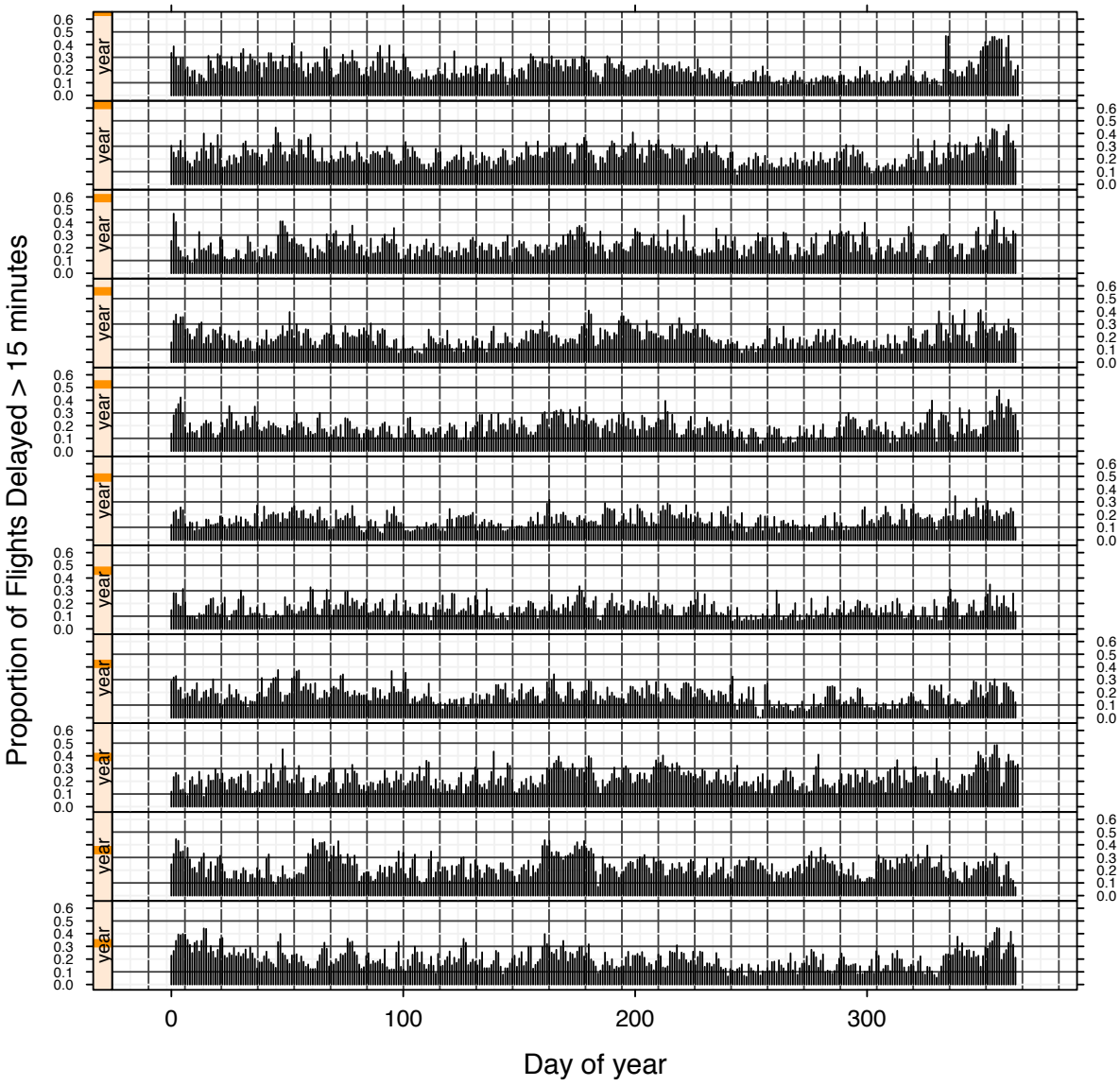


Figure 3.5: Proportion of flights delayed (cont'd)

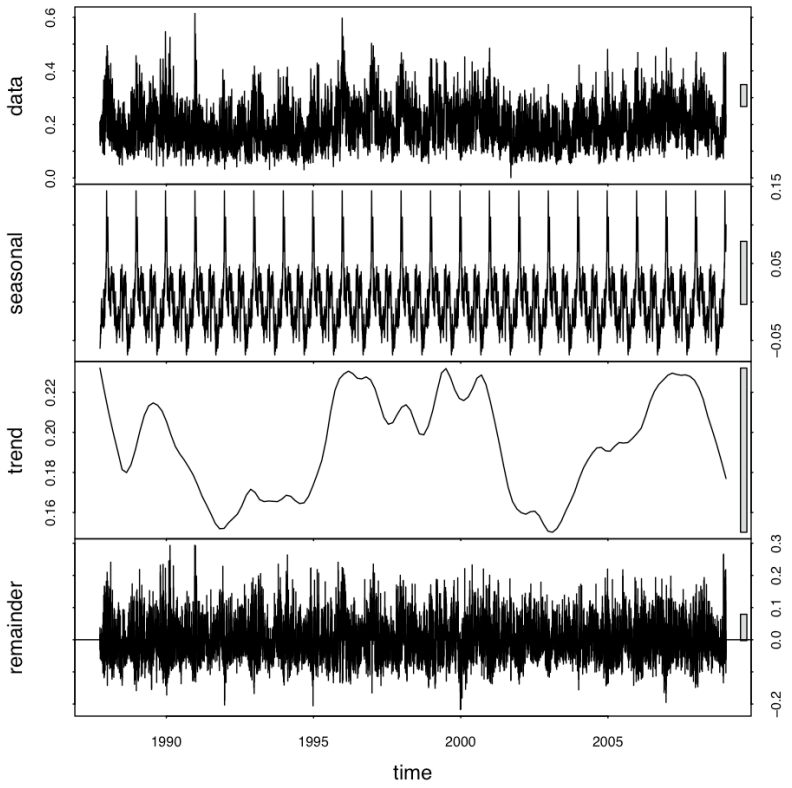


Figure 3.6: STL decomposition of proportion of flights delayed

There is similar seasonality for weekly behavior. The figure *Proportion of flights delayed by day of week.* displays proportion of flights delayed by day of week. The code for this identical to the previous one except we split on `a$wday` and the key is `r[1,c("wday")]`. It appears Thursdays and Fridays can be quite trying.

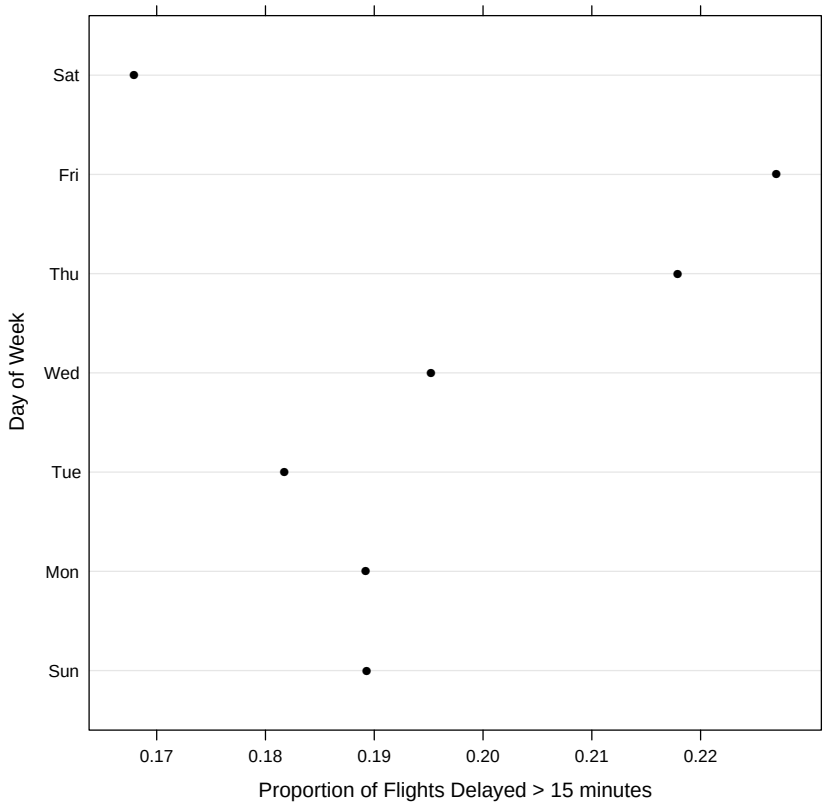


Figure 3.7: Proportion of flights delayed by day of week.

Does the delay proportion change with hour? It appears it does (see *Proportion of flights delayed by hour of day*). The hours are scheduled departure times. *Why are so many flights leaving in the wee hours (12-3) delayed?*

The code to create *Proportion of flights delayed by hour of day* is

```
map <- expression({
  a <- do.call("rbind",map.values)
  a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
  a <- a[!is.na(a$delay.sec),]
  a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
  a$hrs <- as.numeric(format(a[,'sdepart'],"%H"))
  e <- split(a,a$hrs)
  lapply(e,function(r){
    n <- nrow(r); numdelayed <- sum(r$isdelayed)
    rhcollect(as.vector(unlist(c(r[1,c("hrs")]))), c(n, numdelayed))
  })
})
reduce <- expression(
    pre={sums <- c(0,0)},
    reduce = {sums <- sums+apply(do.call("rbind",reduce.values),2,sum)},
    post = { rhcollect(reduce.key, sums) }
```

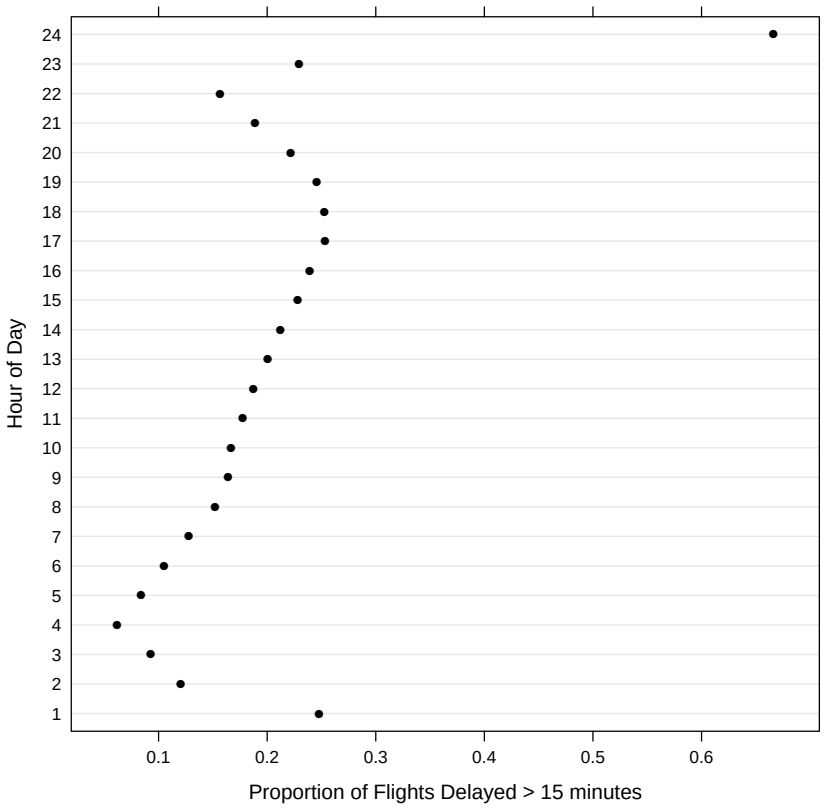


Figure 3.8: Proportion of flights delayed by hour of day

```
    )


mapred <- list()
mapred$rhipe_map_buff_size <- 5
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/delaybyhours"
          ,mapred=mapred)
z=rhex(z)
```

### 3.4.4 Distribution of Delays

Summaries are not enough and for any sort of modeling we need to look at the distribution of the data. So onto the quantiles of the delays. We will look at delays greater than 15 minutes. To compute *approximate* quantiles for the data, we simply discretize the delay and compute a frequency count for the unique values of delay. This is equivalent to binning the data. Given this frequency table we can compute the quantiles.

The distribution of the delay in minutes does not change significantly over months.

```
map <- expression({
  a <- do.call("rbind",map.values)
  a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
  a <- a[!is.na(a$delay.sec),]
  a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
  a <- a[a$isdelayed==TRUE,] ## only look at delays greater than 15 minutes
  apply(a[,c('month','delay.sec')],1,function(r){
        k <- as.vector(unlist(r))
        if(!is.na(k[1])) rhcollect(k,1) # ignore cases where month is missing
      })
})
reduce <- expression(
    pre={sums <- 0} ,
    reduce = {sums <- sums+sum(unlist(reduce.values))},
    post = { rhcollect(reduce.key, sums) }
    )
mapred <- list()
mapred$rhipe_map_buff_size <- 5
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/quantiledelay"
          ,mapred=mapred)
z=rhex(z)
b <- rhread("/airline/quantiledelay")
y1 <- do.call("rbind",lapply(b,"[[",1))
count <- do.call("rbind",lapply(b,"[[",2))
results <- data.frame(month = y1[,1], n=y1[,2], count=count)
results <- results[order(results$month, results$n),]
results.2 <- split(results, results$month)

discrete.quantile<-function(x,n,prob=seq(0,1,0.25),type=7){
  sum.n<-sum(n)
  cum.n<-cumsum(n)
  np<-if(type==7) (sum.n-1)*prob+1 else sum.n*prob+0.5
  np.fl<-floor(np)
  j1<-pmax(np.fl,1)
  j2<-pmin(np.fl+1,sum.n)
  gamma<-np-np.fl
  id1<-unlist(lapply(j1,function(r) seq_along(cum.n)[r<=cum.n][1]))
```

```
39    id2<-unlist(lapply(j2,function(r) seq_along(cum.n)[r<=cum.n][1]))
40    x1<-x[id1]
41    x2<-x[id2]
42    qntl<-(1-gamma)*x1+gamma*x2
43    qntl
44  }
45
46  DEL <- 0.05
47  results.3 <- lapply(seq_along(results.2),function(i){
48    r <- results.2[[i]]
49    a <- discrete.quantile(r[,2],r[,3],prob=seq(0,1,DEL))/60
50    data.frame(month=as.numeric(rep(names(results.2)[[i]],length(a))),prop=seq(0,1,DEL),qt=a)
51  })
52  results.3 <- do.call("rbind",results.3)
53  results.3$month <- factor(results.3$month,
54                            label=c("Jan","Feb","March","Apr","May","June",
55                              "July","August","September","October","November","December"))
56  xyplot(log(qt,2)~prop|month, data=results.3,cex=0.4,col='black',
57         scales=list(x=list(tick.number=10),y=list(tick.number=10)),
58         layout=c(4,3),type='l',
59         xlab="Proportion",ylab="log_2 delay (minutes)",panel=function(x,y,...){
60           panel.grid(h=-1,v=-1);panel.xyplot(x,y,...)
61         }
62  )
```

We can display the distribution by hour of day. The code is almost nearly the same. Differences are in line 8, where the `hrs` is used as the conditioning. But the results are more interesting. The delay amounts increase in the wee hours (look at panel 23,24,1,2 and 3)

```
1   map <- expression({
2     a <- do.call("rbind",map.values)
3     a$delay.sec <- as.vector(a['arrive'])-as.vector(a['sarrive'])
4     a <- a[!is.na(a$delay.sec),]
5     a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
6     a <- a[a$isdelayed==TRUE,] ## only look at delays greater than 15 minutes
7     a$hrs <- as.numeric(format(a['sdepart'],"%H"))
8     apply(a[,c('hrs','delay.sec')],1,function(r){
9           k <- as.vector(unlist(r))
10          if(!is.na(k[1])) rhcollect(k,1)
11        })
12  })
13  reduce <- expression(
14      pre={sums <- 0} ,
15      reduce = {sums <- sums+sum(unlist(reduce.values))},
16      post = { rhcollect(reduce.key, sums) }
17      )
18
19  mapred <- list()
20  mapred$rhipe_map_buff_size <- 5
21  z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
22            ,ifolder="/airline/blocks/",ofolder="/airline/quantiledelaybyhour"
23            ,mapred=mapred)
24  z=rhex(z)
```

The distribution of delay times by airports. This could be analyzed for several airports, but we take the top 3 in terms of volumes. In this display, the quantiles of log_2 of the delay times (in minutes) for inbound and outbound for 4 different airports is plotted. The airports are in order of median delay time. Of note, the median delay time for Chicago (ORD) and San Francisco (SFO) is greater flying in than out (approximately an hour). For both Chicago and

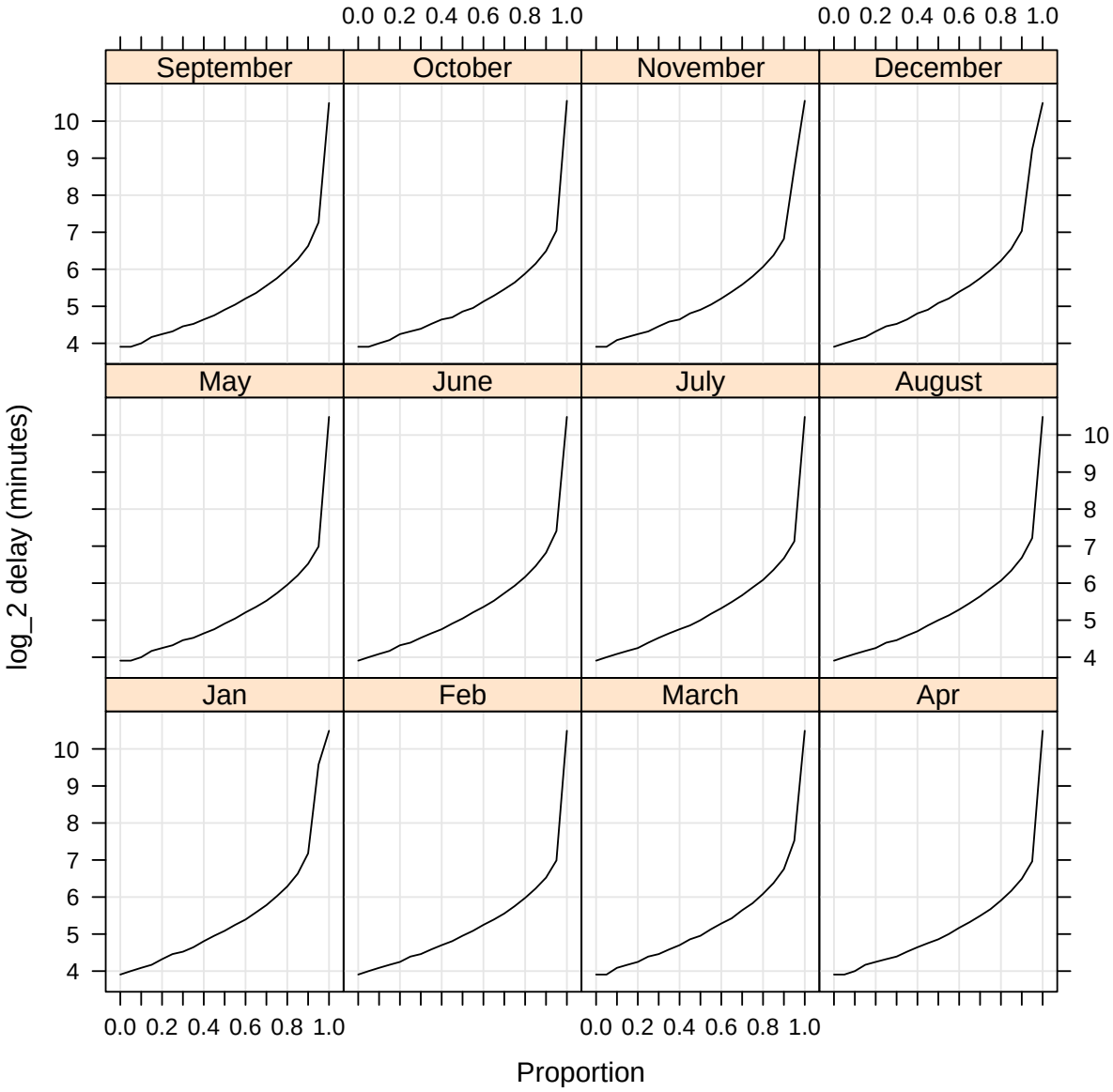


Figure 3.9: Quantile of minute delay (for delay > 15 minutes) across months

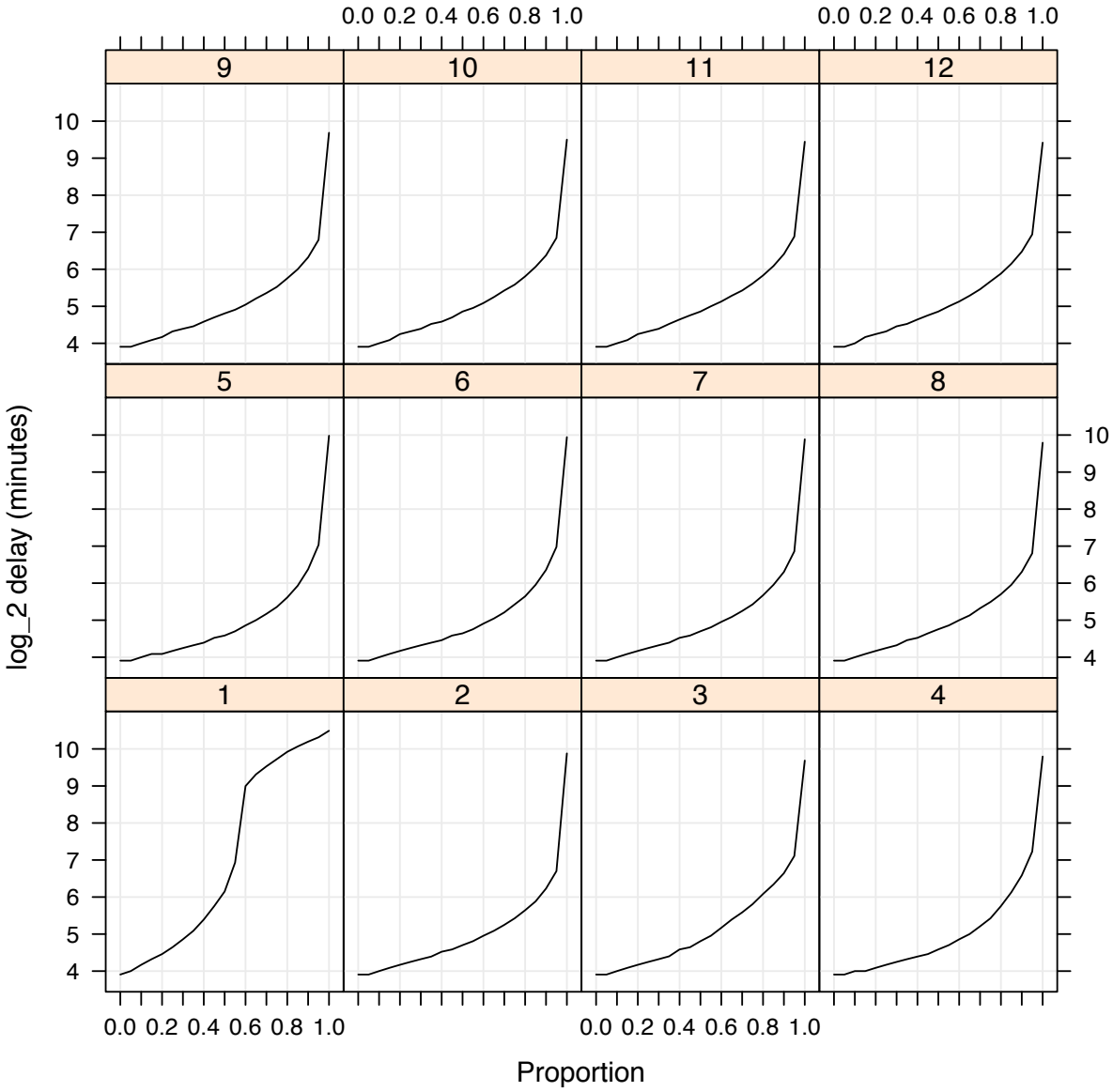


Figure 3.10: Quantile of minute delay (for delay > 15 minutes) by hour of day

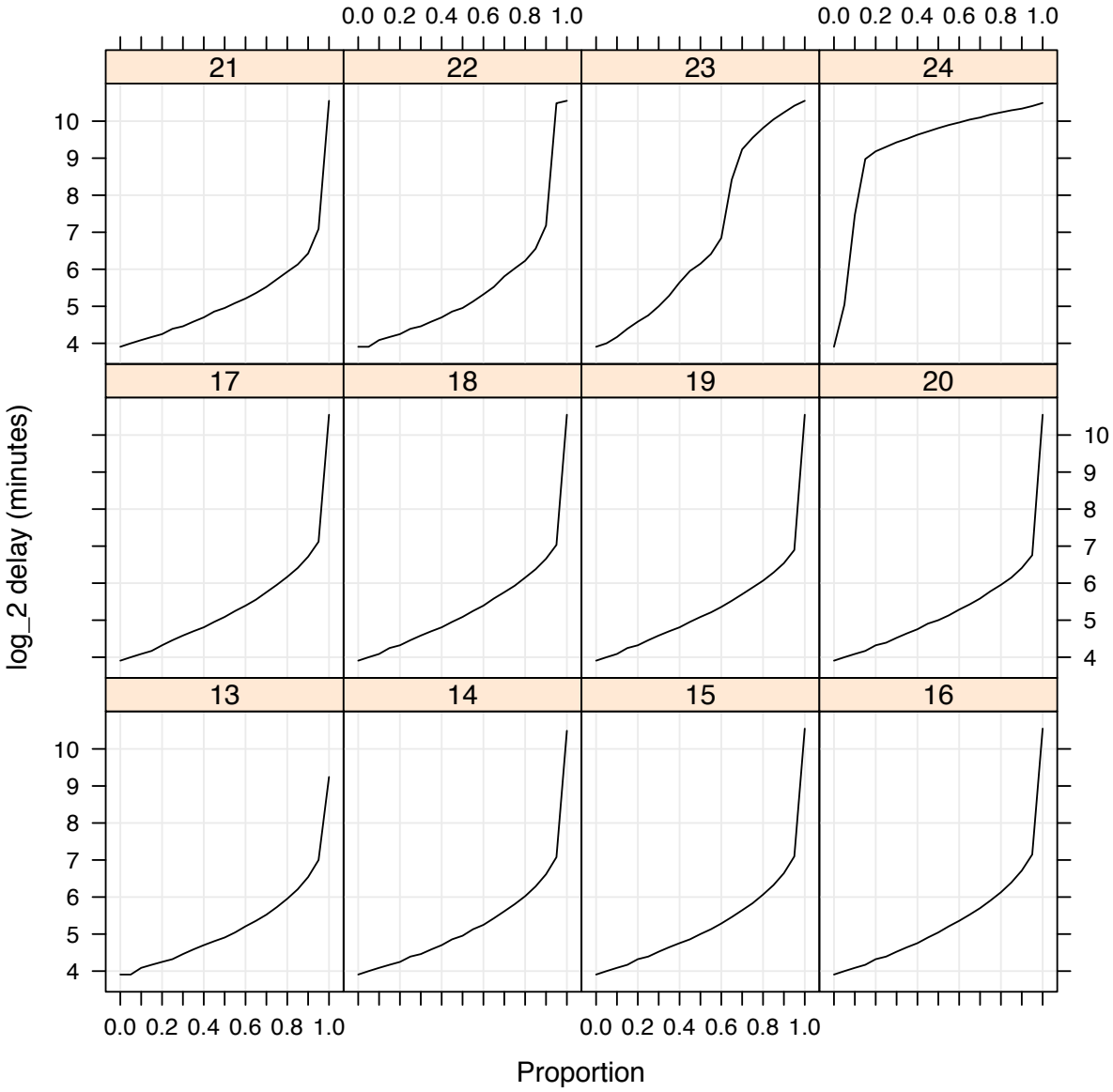


Figure 3.11: Quantile of minute delay (for delay > 15 minutes) by hour of day (cont'd)

Dallas Fort Worth (DFW), the 75th percentile of inbound delays is greater than that for outbound. *Quantile of minute delay for inbound and outbound for 4 different airports. Dotted red lines are 25%,50% and 75% uniform proportions.* displays these differences.

```
 map <- expression({
   cc <- c("ORD","SEA","DFW","SFO")
   a <- do.call("rbind",map.values)
   a <- a[a$origin %in% cc| a$dest %in% cc,]
   if(nrow(a)>0){
     a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
     a <- a[!is.na(a$delay.sec),]
     a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
     a <- a[a$isdelayed==TRUE,]
     for(i in 1:nrow(a)){
       dl <- a[i,'delay.sec']
       if(a[i,'origin'] %in% cc) rhcollect(data.frame(dir="outbound",ap=a[i,"origin"]
                                                      ,delay=dl,stringsAsFactors=FALSE),1)
       if(a[i,'dest'] %in% cc) rhcollect(data.frame(dir="inbound",ap=a[i,"dest"]
                                                    ,delay=dl,stringsAsFactors=FALSE),1)
     }
   }
 })
reduce <- expression(
     pre={sums <- 0} ,
     reduce = {sums <- sums+sum(unlist(reduce.values))},
     post = { rhcollect(reduce.key, sums) }
     )
 mapred <- list()
 mapred$rhipe_map_buff_size <- 5
 z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
           ,ifolder="/airline/blocks/",ofolder="/airline/inoutboundelay"
           ,mapred=mapred)
 z=rhex(z)
```

### 3.4.5 Carrier Delays

Is there a difference in carrier delays? We display the time series of proportion of delayed flights by carrier, ranked by carrier.

```
## For proportions and volumes
map <- expression({
  a <- do.call("rbind",map.values)
  a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
  a <- a[!is.na(a$delay.sec),]
  a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
  a$hrs <- as.numeric(format(a[,'sdepart'],"%H"))
  e <- split(a,a$hrs)
  lapply(e,function(r){
    n <- nrow(r); numdelayed <- sum(r$isdelayed)
    rhcollect(as.vector(unlist(c(r[1,c("carrier")]))), c(n, numdelayed))
  })
})
reduce <- expression(
    pre={sums <- c(0,0)},
    reduce = {sums <- sums+apply(do.call("rbind",reduce.values),2,sum)},
    post = { rhcollect(reduce.key, sums) }
    )
```

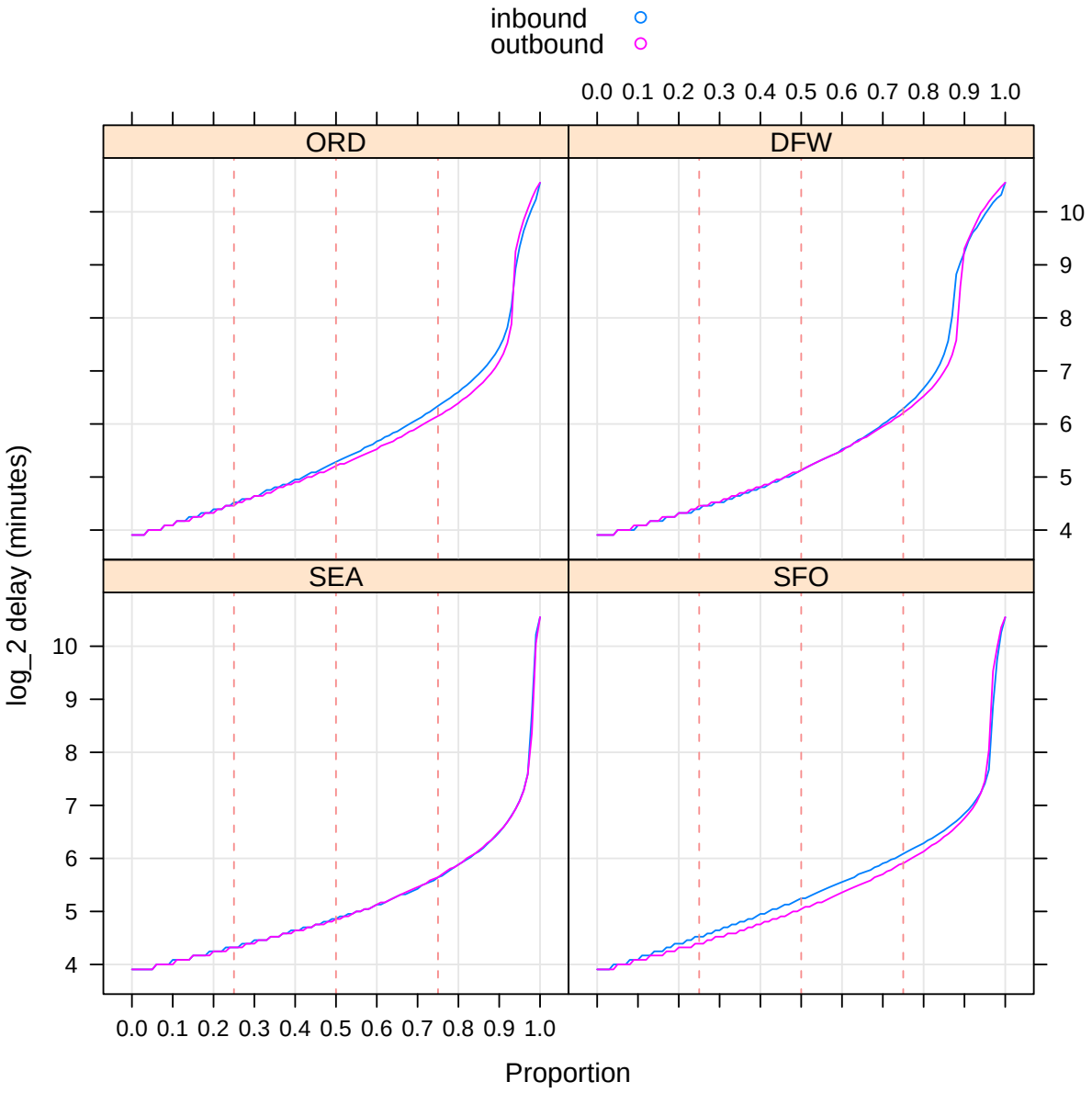


Figure 3.12: Quantile of minute delay for inbound and outbound for 4 different airports. Dotted red lines are 25%,50% and 75% uniform proportions.

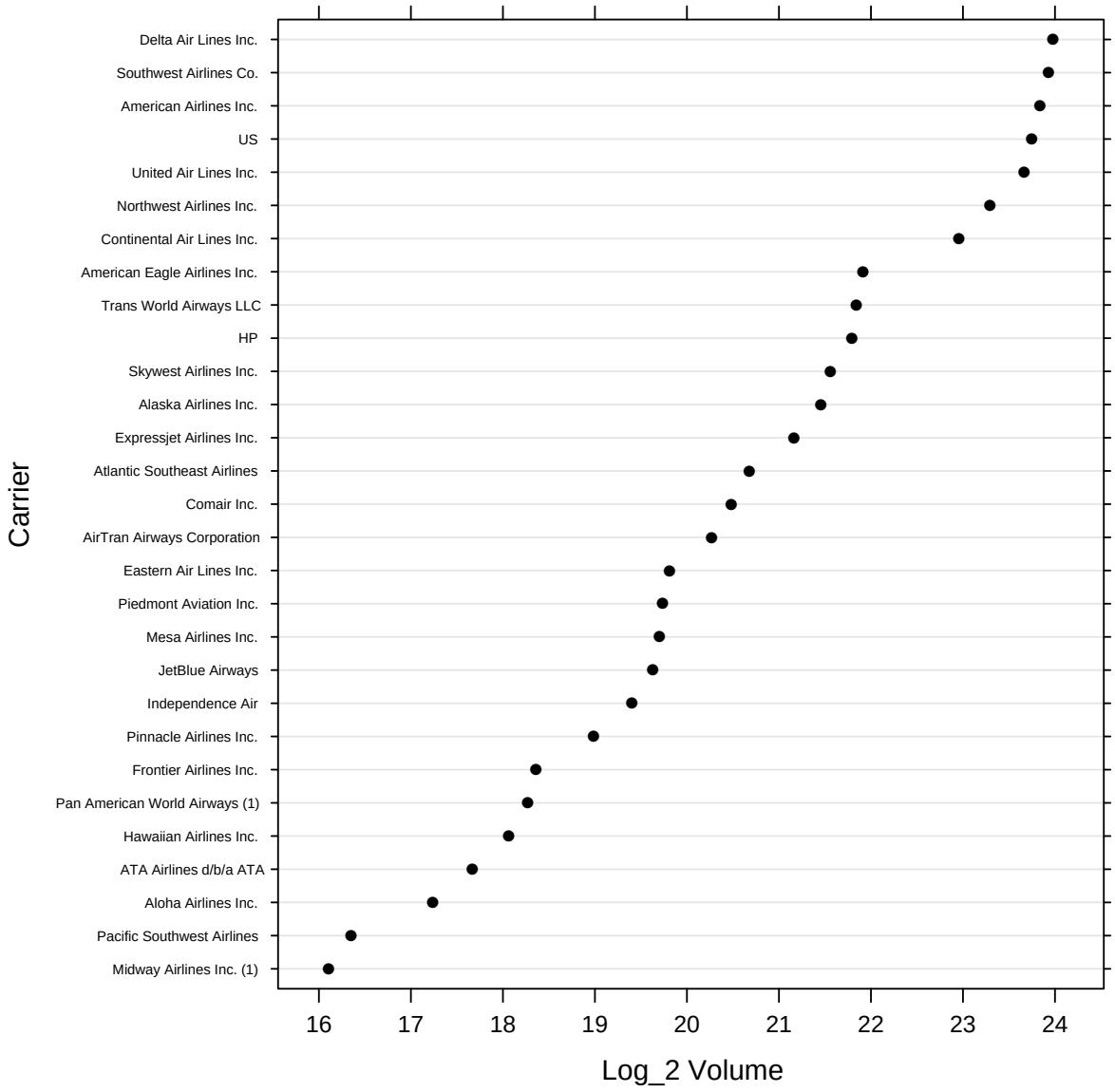


Figure 3.13: Log base 2 volume of flights by carrier

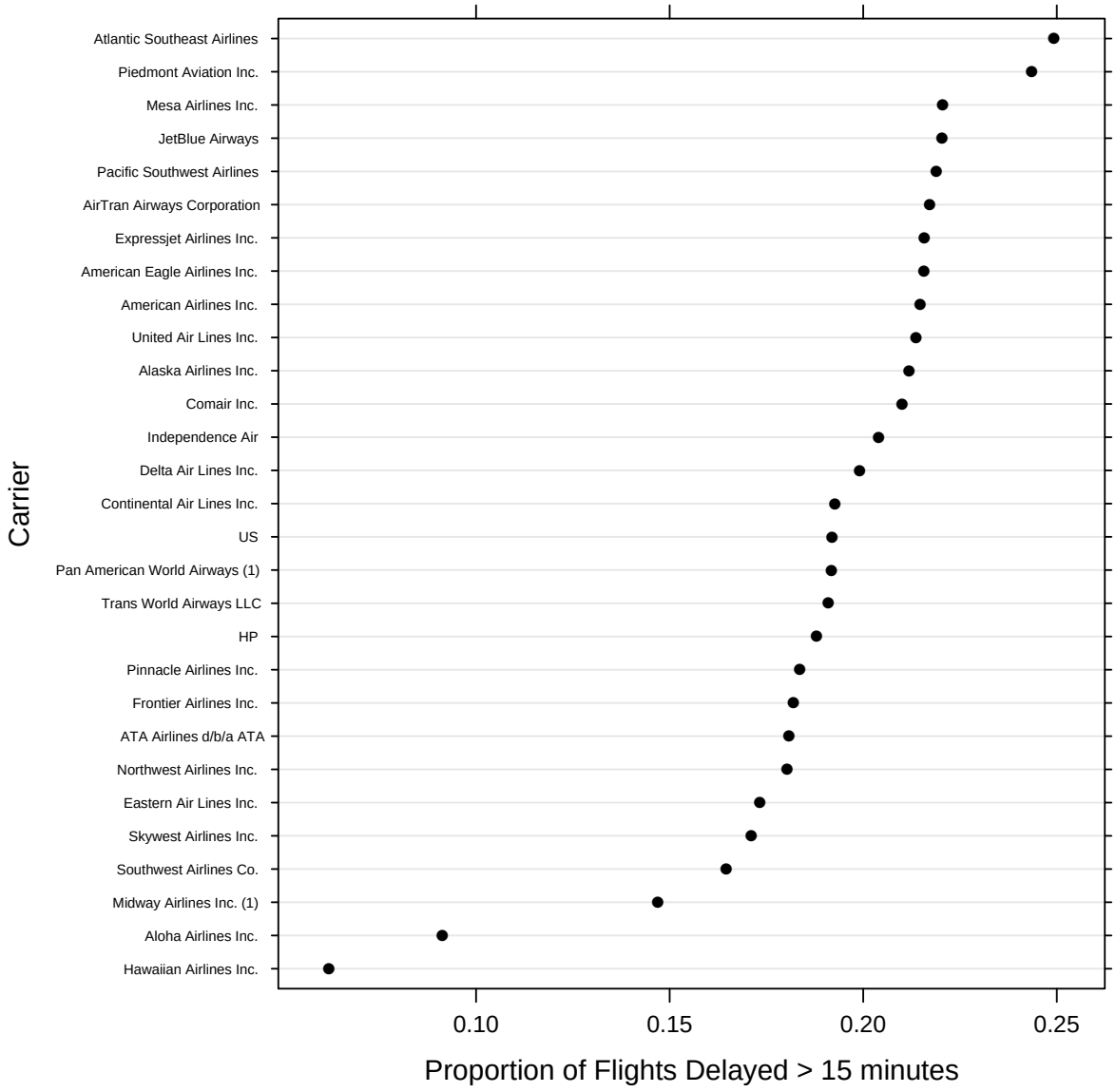


Figure 3.14: Proportion of flights delayed by carrier. Compare this with the previous graph.

### 3.4.6 Busy Routes

Which are busy the routes? A simple first approach (for display purposed) is to create a frequency table for the unordered pair *(i,j)* where *i* and *j* are distinct airport codes. Displays this over the US map.

```
map <- expression({
  a <- do.call("rbind",map.values)
  y <- table(apply(a[,c("origin","dest")],1,function(r){
    paste(sort(r),collapse=",")
  }))
  for(i in 1:length(y)){
    p <- strsplit(names(y)[[i]],",")[[1]]
    rhcollect(p,y[[1]])
  }
})
reduce <- expression(
    pre={sums <- 0},
    reduce = {sums <- sums+sum(unlist(reduce.values))},
    post = { rhcollect(reduce.key, sums) }
    )
mapred <- list()
mapred$rhipe_map_buff_size <- 5
mapred$mapred.job.priority="VERY_LOW"
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/ijjoin"
          ,mapred=mapred)
z=rhex(z)

##Merge results
b=rhread("/airline/ijjoin")
y <- do.call("rbind",lapply(b,"[[",1))
results <- data.frame(a=y[,1],b=y[,2],count=
           do.call("rbind",lapply(b,"[[",2)),stringsAsFactors=FALSE)
results <- results[order(results$count,decreasing=TRUE),]
results$cumprop <- cumsum(results$count)/sum(results$count)
a.lat <- t(sapply(results$a,function(r){
  ap[ap$iata==r,c('lat','long')]
}))
results$a.lat <- unlist(a.lat[,'lat'])
results$a.long <- unlist(a.lat[,'long'])
b.lat <- t(sapply(results$b,function(r){
  ap[ap$iata==r,c('lat','long')]
}))
b.lat["CBM",] <- c(0,0)
results$b.lat <- unlist(b.lat[,'lat'])
results$b.long <- unlist(b.lat[,'long'])

head(results)
   a   b  count     cumprop    a.lat     a.long    b.lat     b.long
1 ATL ORD 145810 0.001637867 33.64044  -84.42694 41.97960  -87.90446
2 LAS LAX 140722 0.003218581 36.08036 -115.15233 33.94254 -118.40807
3 DEN DFW 140258 0.004794083 39.85841 -104.66700 32.89595  -97.03720
4 LAX SFO 139427 0.006360250 33.94254 -118.40807 37.61900 -122.37484
5 DFW IAH 137004 0.007899200 32.89595  -97.03720 29.98047  -95.33972
6 DTW ORD 135772 0.009424311 42.21206  -83.34884 41.97960  -87.90446
```

Using the above data, the following figure draws lines from ORD (Chicago) to other destinations. The black points are the airports that handle 90% of the total air traffic volume. The grey points are the remaining airports. The flights from Chicago (ORD) are color coded based on volume carried e.g. red implies those routes carry the top 25% of traffic

in/out of ORD.

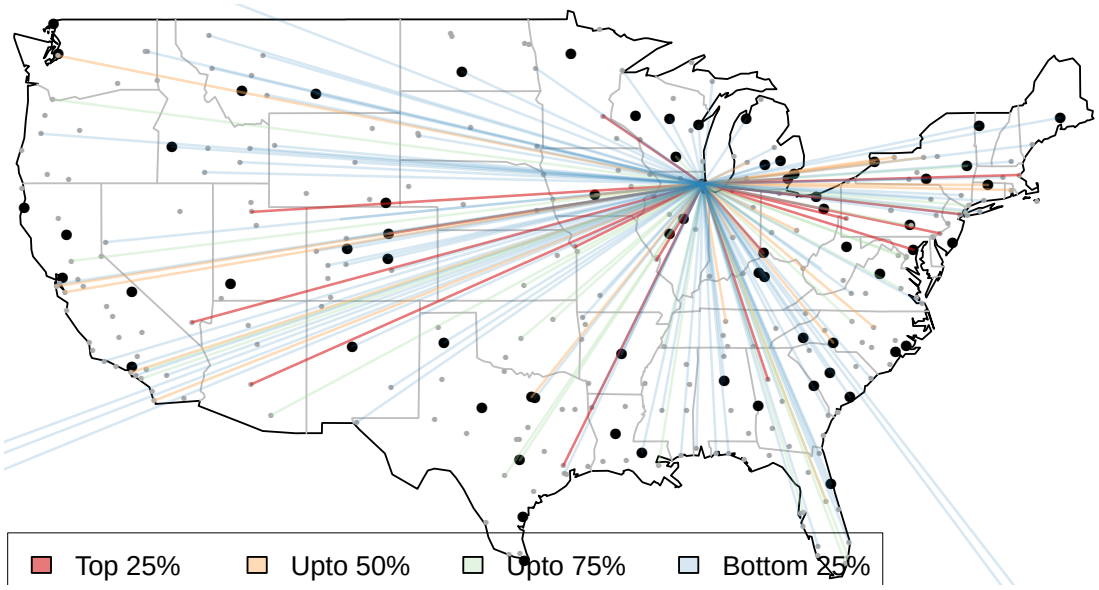


Figure 3.15: Flights in and out of Chicago color coded by % cumulative contribution.

## 3.5 Streaming Data?

Some algorithms are left associative in their operands $t_1, t_2, \ldots, t_n$ but not commutative. For example a streaming update algorithm that computes the inter-arrival times of time series data for different levels of a categorical variable. That is, the triangular series $t_{k,1}, t_{k,2}, \ldots, t_{k,n_k}$ where $k$ takes the levels of a categorical variable $C$ (which takes the values $1, 2, 3, \ldots, m$). The input are pairs $(i, j), i \in \{1, 2, \ldots, m\}, j \in \{t_{ik}\}$ . In the following code, the data structure $F$ is updated with the *datastructure* contained in the values (see the map). The *datastructures* are indexed in time by

the *timepoint* - they need to be sent to the reducer (for a given level of the categorical variable *catlevel*) in time order. Thus the map sends the pair `(catlevel, timepoint)` as the key. By using the `part` parameter (see line 39) *all* the data structures associated with the `catlevel` are sent to the same R reduce process. This is vital since all the component R expressions in the reduce are run in the same process and namespace. To preserve numeric ordering we insist on the special map output key class (see line 38). With this special key class, we cannot have a map output format. In the reduce, the setup expression `redsetup` is run upon R startup (the process assigned to several keys and their associated values). Then for each new intermediate key `(catlevel, timepoint)`, it runs the `pre`, `reduce` and `post`. The lack of a `post` is because we have exactly one intermediate value for a given key (assuming the time points for a category are unique). The `redclose` expression is run when all keys and values have been processed by the reducer and R is about to quit.

```
1  map <- expression({
2    lapply(seq_along(map.values),function(r){
3      catlevel <- map.keys[[r]] #numeric
4      timepoint <- map.values[[r]]$timepoint #numeric
5      datastructure <- map.values[[r]]$data
6      key <- c(catlevel,timepoint)
7      rhcollect(key,datastructure)
8    })
9  })
10 redsetup <- expression({
11   currentkey <- NULL
12 })
13 reduce <- expression(
14     pre={
15       catlevel <- reduce.key[1]
16       time <- reduce.key[2]
17       if(!identical(catlevel,currentkey)){
18         ## new categorical level
19         ## so finalize the computation for
20         ## the previous level e.g. use rhcollect
21         if(!identical(currentkey,NULL))
22           FINALIZE(F)
23         ## store current categorical level
24         currentkey <- catlevel
25         ## initialize computation for new level
26         INITIALIZE(F)
27       }
28     },
29     reduce={
30       F <- UPDATE(F, reduce.values[[1]])
31     })
32 redclose <- expression({
33   ## need to run this, otherwise the last catlevel
34   ## will not get finalized
35  FINALIZE(F)
36 })
37 rhmr(..., combiner=FALSE,setup=list(reduce=redsetup),cleanup=list(reduce=redclose),
38      orderby="numeric",
39      part=list(lims=1,type='numeric'))
```

## 3.6 Simple Debugging

Consider the example code used to compute the delay quantiles by month (see *Delay Quantiles By Month* ). We can use `tryCatch` for some simple debugging. See the error in line 7, there is no such variable `isdelayed`

```
map <- expression({
  tryCatch({
    a <- do.call("rbind",map.values)
    a$delay.sec <- as.vector(a[,'arrive'])-as.vector(a[,'sarrive'])
    a <- a[!is.na(a$delay.sec),]
    a$isdelayed <- sapply(a$delay.sec,function(r) if(r>=900) TRUE else FALSE)
    a <- a[isdelayed==TRUE,] ## only look at delays greater than 15 minutes
    apply(a[,c('month','delay.sec')],1,function(r){
      k <- as.vector(unlist(r))
      if(!is.na(k[1])) rhcollect(k,1) # ignore cases where month is missing
    })
  },error=function(e){
    e$message <- sprintf("Input File:%s\nAttempt ID:%s\nR INFO:%s",
                 Sys.getenv("mapred.input.file"),Sys.getenv("mapred.task.id"),e$message)
    stop(e) ## WONT STOP OTHERWISE
  })
})
  reduce <- expression(
    pre={sums <- 0} ,
    reduce = {sums <- sums+sum(unlist(reduce.values))},
    post = { rhcollect(reduce.key, sums) }
    )
mapred <- list()
mapred$rhipe_map_buff_size <- 5
z <- rhmr(map=map,reduce=reduce,combiner=TRUE,inout=c("sequence","sequence")
          ,ifolder="/airline/blocks/",ofolder="/airline/quantiledelay"
          ,mapred=mapred)
z=rhex(z)
```

Produces a slew of errors like

```
10/08/04 00:41:20 INFO mapred.JobClient: Task Id : attempt_201007281701_0273_m_000023_0, Status : FA
java.io.IOException: MROutput/MRErrThread failed:java.lang.RuntimeException:
R ERROR
=======
Error in `[.data.frame`(a, isdelayed == TRUE, ) : Input File:
Attempt ID:attempt_201007281701_0273_m_000023_0
R INFO:object "isdelayed" not found
```

It can be very useful to provide such debugging messages since R itself doesn't provide much help. Use this to provide context about variables, such printing the first few rows of relevant data frames (if required). Moreover, some errors don't come to the screen instead the job finishes successfully (but very quickly since the R code is failing) but the error message is returned as a counter. The splits succeed since Hadoop has finished sending its data to R and not listening to for errors from the R code. Hence any errors sent from R do not trigger a failure condition in Hadoop. This is a RHIPE design flaw. To compensate for this, the errors are stored in the counter *R_ERROR*.

Unfortunately, RHIPE does not offer much in the way of debugging. To run jobs locally that is, Hadoop will execute the job in a single thread on one computer, set `mapred.job.tracker` to *local* in the `mapred` argument of `rhmr`. In this case, `shared.files` cannot be used and `copyFiles` will not work.

CHAPTER

FOUR

# TRANSFORMING TEXT DATA

This chapter builds on the *Airline Dataset*. One foreseeable use of RHIPE is to transform text data. For example,

1. Subset Southwest Airline and Delat Airways information to create a new set of text files, one with only Southwest and the other with Delta.
2. Transform the original text data to one with fewer columns and some transformed e.g. Airport Codes to full names.

We'll cover both examples.

## 4.1 Subset

The text data looks like

```
1987,10,23,5,1841,1750,2105,2005,PS,1905,NA,144,135,NA,60,51,LAX,SEA,954,NA,NA,0,NA,0,...
1987,10,24,6,1752,1750,2010,2005,PS,1905,NA,138,135,NA,5,2,LAX,SEA,954,NA,NA,0,NA,0,...
...
...
```

The carrier name is column 9. Southwest carrier code is *WN*, Delta is *DL*. Only those rows with column 9 equal to *WN* or *DL* will be saved.

```
map <- expression({
  ## Each element of map.values is a line of text
  ## this needs to be tokenized and then combined
  tkn <- strsplit(unlist(map.values),",")
  text <- do.call("rbind",tkn)
  text <- text[text[,9] %in% c("WN","DL"),,drop=FALSE]
  if(nrow(text)>0) apply(text,1, function(r) rhcollect(r[9], r))
})
```

`rhcollect` requires both a key and value but we have no need for the key. So NULL is given as the key argument and *mapred.textoutputformat.usekey* is set to FALSE so that the key is not written to disk. RHIPE quotes strings, which we do not want (nothing is quoted), so *rhipe_string_quote* is set to '' and *mapred.field.separator* is "," since the original data is comma separated. A partitioner is used to send all the Southwest flights to one file and Delta to another.

```
z <- rhmr(map=map,ifolder="/airline/data/2005.csv",ofolder="/airline/southdelta",
          ,inout=c("text","text"),orderby="char",
          part=list(lims=1,type="string"),
          mapred=list(
           mapred.reduce.tasks=2,
            rhipe_string_quote='',
```

```
7               mapred.field.separator=",",
8               mapred.textoutputformat.usekey=FALSE))
9   rhex(z)
```

The output, in one file is

```
2005,1,5,3,1850,1850,2208,2025,WN,791,N404,258,155,207,103,0,BDL,...
2005,1,5,3,810,810,1010,940,WN,824,N784,180,150,155,30,0,BDL,...
2005,1,5,3,1430,1325,1559,1435,WN,317,N306SW,89,70,61,84,65,BDL,...
2005,1,5,3,705,705,830,815,WN,472,N772,85,70,57,15,0,BDL,...
```

and the other

```
2005,12,22,4,1652,1655,1815,1837,DL,901,N109DL,...
2005,12,22,4,1825,1825,1858,1848,DL,902,N932DL,...
2005,12,22,4,1507,1511,1641,1649,DL,903,N306DL,...
```

## 4.2 Transformations

Convert each airport codes to their name equivalent. Airport codes can be found at the JSM website . When working with massive data, repeatedly used operations need to be as fast as possible. Thus we will save the airport code to airport name as a hash table using the `new.env` function. Airport codes (origin and destination) are in columns 17 and 18. The `setup` expression loads this data set and creates a function that does the mapping.

```
1   airport <- read.table("~/tmp/airports.csv",sep=",",header=TRUE,stringsAsFactors=FALSE)
2   aton <- new.env()
3   for(i in 1:nrow(airport)){
4     aton[[ airport[i,"iata"] ]] <- list(ap=airport[i,"airport"],latlong=airport[i,c("lat","long")])
5   }
6   rhsave(aton,file="/tmp/airports.Rdata")
7
8   setup <- expression({
9     load("airports.Rdata")
10    co <- function(N){
11      sapply(text[,N],function(r){
12        o <- aton[[ r[1] ]]$ap
13        if(is.null(o)) NA else sprintf('"%s"',o)
14      })
15    }
16  })
```

The map will use the `aton` dictionary to get the complete names which are quoted (in line 13 above). Removing the `sprintf` makes it much faster.

```
1   map <- expression({
2     tkn <- strsplit(unlist(map.values),",")
3     text <- do.call("rbind",tkn)
4     text[,17] <- co(17)
5     text[,18] <- co(18)
6     apply(text,1,function(r){
7       rhcollect(NULL,r)
8     })
9   })
10
11  z <- rhmr(map=map,ifolder="/airline/data/2005.csv",ofolder="/airline/transform",
12            ,inout=c("text","text"),
13            shared=c("/airport/airports.Rdata"),
```

```
14            setup=setup,
15            mapred=list(
16              mapred.reduce.tasks=0,
17              rhipe_string_quote='',
18              mapred.field.separator=",",
19              mapred.textoutputformat.usekey=FALSE))
20  rhex(z)
```

and this gives us

```
1987,10,28,3,NA,1945,NA,2100,...,"San Francisco International","John Wayne /Orange Co,...
1987,10,29,4,2025,1945,2141,2100,...,"San Francisco International","John Wayne /Orange Co,...
1987,10,30,5,1947,1945,2109,2100,...,"San Francisco International","John Wayne /Orange Co,...
1987,10,1,4,2133,2100,2303,2218,...,"San Diego International-Lindbergh","San Francisco International,
```

CHAPTER

FIVE

# SIMULATIONS

Simulations are an example of task parallel routines in which a function is called repeatedly with varying parameters. These computations are processor intensive and consume/produce little data. The evaluation of these tasks are independent in that there is no communication between them. With $N$ tasks and $P$ processors, if $P = N$ we could run all $N$ in parallel and collect the results. However, often $P << N$ and thus we must either

- Create a queue of tasks and assign the top most task on the queue to the next free processor. This works very well in an heterogeneous environment e.g. with varying processor capacities or varying task characteristics - free resources will be automatically assigned pending tasks. The cost in creating a new task can be much greater than the cost of evaluating the task.
- Partition the $N$ tasks into $n$ splits each containing $\lceil N/n \rceil$ tasks (with the last split containing the remainder). These splits are placed in a queue, each processor is assigned a splits and the tasks in a split are evaluated sequentially.

The second approach simplifies to the first when $n = N$. Creating one split per task is inefficient since the time to create,assign launch the task contained in a split might be much greater than the evaluation of the task. Moreover with $N$ in the millions, this will cause the Jobtracker to run out of memory. It is recommended to divide the $N$ tasks into fewer splits of sequential tasks. Because of non uniform running times among tasks, processors can spend time in the sequential execution of tasks in a split $\sigma$ with other processors idle. Hadoop will schedule the split $\sigma$ to another processor (however it will not divide the split into smaller splits), and the output of whichever completes first will be used.

RHIPE provides two approaches to this sort of computation. To apply the function $F$ to the set $\{1, 2, \ldots, M\}$, the pseudo code would follow as (here we assume $F$ returns a data frame)

```
FC <- expression({
  results <- do.call("rbind",lapply(map.values,F))
  rhcollect(1,results)
})

rhmr(map=FC,ofolder='tempfolder',inout=c('lapply','sequence'),N=M
     ,mapred=list(mapred.map.tasks=1000))

do.call('rbind',lapply(rhread('/tempfolder', mc=TRUE),'[[',2))
```

Here $F$ is applied to the numbers $1, 2, \ldots, M$. The job is decomposed into 1000 splits (specified by `mapred.map.tasks`) each containing approximately $\lceil M/1000 \rceil$ tasks. The expression, $FC$ sequentially applies $F$ to the elements of `map.values` (which will contain a subset of $1, 2, \ldots, M$) and aggregate the returned data frames with a call to `rbind`. In the last line, the results of the 1000 tasks (which is a list of data frames) are read from the HDFS, the data frame are extracted from the list and combined using a call to `rbind`. Much of this is boiler plate RHIPE code and the only varying portions are: the function $F$, the number of iterations $M$, the number of groups (e.g. `mapred.map.tasks`) and the aggregation scheme (e.g. I used the call to `rbind`). R lists can be written to a file

on the HDFS(with `rhwrite`), which can be used as i input to a MapReduce job . All of this could then be wrapped in a single function:

```
rhipe.lapply(function, input, groups=number.of.cores, aggregate)
```

where `function` is $F$, `input` could be a list or maximum trials (e.g. $M$). The parameter `groups` is the number of groups to divide the job into and by default is the number of cluster cores and `aggregate` is a function to aggregate the intermediate results. With this function, the user can distribute the `lapply` command and rely on Hadoop to handle fault-tolerancy and the scheduling of processors in an optimal fashion. The `rhlapply` function is present to do this.

**::** rhlapply(ll, F, ofolder,setup=NULL,readIn = TRUE, N, aggr=NULL,...)

This applies `F` to the elements of `ll`. If provided a value, it will save the results to `ofolder` and the results are returned as a list if `readIn` is TRUE. The value of `N` is passed to `rhwrite` (if `ll` is a list, they will be written to a temporary file). `setup` can be used to load files. The `rhllapply` command takes the arguments of `rhmr` (e.g. `mapred`) and they passed to `rhmr`.

## 5.1 A Note on Random Number Generators

RHIPE does not include parallel random generator e.g. Scalable Parallel Random Number Generators Library and the Rstreams package for R ([ecuyer] and [Masac]). Parallel RNGs can create streams of random numbers that are not correlated across cluster computers (i.e enforce 'statistical independence') and ensure reproducibility of streams for research. RHIPE can guarantee independent streams since each task has a unique identifier obtained from the environment variable *mapred.task.id*. Since the identifier is unique for every task it can be used to seed random number generators. This cannot be used for reproducible results. There is ongoing work to integrate parallel random generator packages for R with RHIPE.

CHAPTER

SIX

# RHIPE FUNCTIONS

RHIPE has functions that access the HDFS from R, that are used inside MapReduce jobs and functions for managing MapReduce jobs.

## 6.1 HDFS Related

### 6.1.1 rhdel - File Deletion

```
rhdel(folders)
```

This function deletes the folders contained in the character vector `folders` which are located on the HDFS. The deletion is recursive, so all subfolders will be deleted too. Nothing is returned.

### 6.1.2 rhls - Listing Files

```
rhls(path, recurse=FALSE)
```

Returns a data frame of filesystem information for the files located at `path`. If `recurse` is TRUE, `rhls` will recursively travel the directory tree rooted at `path`. The returned object is a data frame consisting of the columns: *permission, owner, group, size (which is numeric), modification time*, and the *file name*. `path` may optionally **end** in '*' which is the wildcard and will match any character(s).

### 6.1.3 rhget - Copying from the HDFS

```
rhget(src,dest)
```

Copies the files (or folder) at `src`, located on the HDFS to the destination `dest` located on the local filesystem. If a file or folder of the same name as `dest` exists on the local filesystem, it will be deleted.

### 6.1.4 rhput - Copying to the HDF

```
rhput(src,dest)
```

Copies the local file called `src` (not a folder) to the destination `dest` on the HDFS.

### 6.1.5 rhcp - Copying on the HDFS

```
rhcp(src,dest)
```

Copies the file (or folder) `src` on the HDFS to the destination `dest` also on the HDFS.

### 6.1.6 rhwrite - Writing R data to the HDFS

```
rhwrite(list,dest,N=NULL)
```

Takes a list of objects, found in `list` and writes them to the folder pointed to by `dest` which will be located on the HDFS. The file `dest` will be in a format interpretable by RHIPE, i.e it can be used as input to a MapReduce job. The values of the list of are written as key-value pairs in a SequenceFileFormat format. `N` specifies the number of files to write the values to. For example, if `N` is 1, the entire list `list` will be written to one file in the folder `dest`. Computations across small files do not parallelize well on Hadoop. If the file is small, it will be treated as one split and the user does not gain any (hoped for) parallelization. Distinct files are treated as distinct splits. It is better to split objects across a number of files. If the list consists of a million objects, it is prudent to split them across a few files. Thus if $N$ is 10 and `list` contains 1,000,000 values, each of the 10 files (located in the directory `dest`) will contain 100,000 values.

Since the list only contains values, the keys are the indices of the value in the list, stored as strings. Thus when used as a source for a MapReduce job, the variable `map.keys` will contain numbers in the range $[1, length(list)]$. The variable `map.values` will contain elements of `list`.

### 6.1.7 rhread - Reading data from HDFS into R

```
rhread(files,type="sequence",max=-1,mc=FALSE,buffsize=2*1024*1024)
```

Reads the key,value pairs from the files pointed to by `files`. The source `files` can end in a wildcard (*) e.g. */path/input/p** will read all the key,value pairs contained in files starting with *p* in the folder */path/input/*. The parameter `type` specifies the format of `files`. This can be one of `text`, `map` or `sequence` which imply a Text file, MapFile or a SequenceFile respectively. For text files, RHIPE returns a matrix of lines, each row a line from the text files. Specifying `max` for text files, limits the number of bytes read and is currently alpha quality.

Thus data written by `rhwrite` can be read

using `rhread`. The parameter `max` specifies the maximum number of entries to read, by default all the key,value pairs will be read. Setting `mc` to TRUE will use the the `multicore` [multicore] package to convert the data to R objects in parallel. The user must have first loaded `multicore` via call to library. This often does accelerate the process of reading data into R.

### 6.1.8 rhgetkeys - Reading Values from Map Files

```
rhgetkey(keys, path)
```

Returns the values from the map files contained in `path` corresponding to the keys in `keys`. `path` will contain folders which is MapFiles are stored. Thus the `path` must have been created as the output of a RHIPE job with `inout[2]` (the output format) set to *map*. Also, the saved keys must be in sorted order. This is always the case if

1. *mapred.reduce.tasks* is not zero.
2. The variable *reduce.key* is not modified.
3. `orderby` is not the default (*bytes*) in the call to `rhmr`

A simple way to convert any RHIPE SequenceFile data set to MapFile is to run an identity MapReduce

```
map <- expression({
  lapply(seq_along(map.values),function(i)
    rhcollect(map.keys[[i]],map.values[[i]]))
})
rhmr(map=map,ifolder,ofolder,inout=c("sequence","map"))
```

The `keys` argument is a list of the keys. Keys are R objects and are characterized by their attributes too. So

```
> identical(c(x=1),c(1))
FALSE
```

If the stored key is `c(x=1)` then this call to `rhgetkey` will not work

```
rhgetkey(list(c(1)),path)
```

but this will

```
rhgetkey(list(c(x=1)),path)
```

## 6.2 MapReduce Administration

### 6.2.1 rhex - Submitting a MapReduce R Object to Hadoop

```
rhex(mrobj, async=FALSE,mapred)
```

Submits a MapReduce job (created using `rhmr`) to the Hadoop MapReduce framework. The argument `mapred` serves the same purpose as the `mapred` argument to `rhmr`. This will override the settings in the object returned from `rhmr`. The function returns when the job ends (success/failure or because the user terminated (see `rhkill`)). When `async` is TRUE, the function returns immediately, leaving the job running in the background on Hadoop.

The function returns an object of class *jobtoken*. The generic function `print.jobtoken`, displays the start time, duration (in seconds) and percent progress. This object can be used in calls to `rhstatus`,‘‘rhjoin‘‘ and `rhkill`.

### 6.2.2 rhstatus - Monitoring a MapReduce Job

```
rhstatus(jobid)
```

This returns the status of an running MapReduce job. The parameter `jobid` can either be a string with the format *job_datetime_id* (e.g. *job_201007281701_0274*) or the value returned from `rhex` with the `async` option set to TRUE.

A list of 4 elements:

- the state of the job (one of *START, RUNNING, FAIL,COMPLETE*),
- the duration in seconds,
- a data frame with columns for the Map and Reduce phase. This data frame summarizes the number of tasks, the percent complete, and the number of tasks that are pending, running, complete or have failed.
- In addition the list has an element that consists of both user defined and Hadoop MapReduce built in counters (counters can be user defined with a call to `rhcounter`).

### 6.2.3 rhjoin - Waiting on Completion of a MapReduce Job

```
rhjoin(jobid, ignore=TRUE)
```

Calling this functions pauses the R console till the MapReduce job indicated by `jobid` is over (successfully or not). The parameter `jobid` can either be string with the format *job_datetime_id* or the value returned from `rhex` with the `async` option set to TRUE. This function returns the same object as `rhex` i.e a list of the results of the job (TRUE or FALSE indicating success or failure) and a counters returned by the job. If `ignore` is FALSE, the progress will be displayed on the R console (much like `rhex`)

### 6.2.4 rhkill - Stopping a MapReduce Job

```
rhkill(jobid)
```

This kills the MapReduce job with job identifier given by `jobid`. The parameter `jobid` can either be string with the format *job_datetime_id* or the value returned from `rhex` with the `async` option set to TRUE.

CHAPTER

SEVEN

# PACKAGING A JOB FOR MAPREDUCE

The function `rhmr` discussed below creates the R object that contains all the information required by RHIPE to run a MapReduce job. Within the MapReduce environment, RHIPE provides 3 functions to interact with the Hadoop Framework. These are discussed in *Functions to Communicate with Hadoop during MapReduce*

## 7.1 Creating a MapReduce Object

### 7.1.1 rhmr - Creating the MapReduce Object

```
rhmr(map,reduce=NULL, combiner=FALSE,
   setup=NULL,cleanup=NULL,
   ofolder='',ifolder='',orderby='bytes'
   inout=c("text","text"),mapred=NULL,
   shared=c(),jarfiles=c(),
   partitioner=NULL,copyFiles=F,
   N=NA,opts=rhoptions(),jobname="")
```

The `rhmr` takes the users map and reduce expressions, the input source and output destination and the input/output formats. It returns an object that can submitted to the Hadoop Framework via a call to `rhex`.

**`map`** The map is an R expression (created using the R command `expression`) that is evaluated by RHIPE during the map stage. For each task, RHIPE will call this expression multiple times. If a task consists of $W$ key,value pairs, the expression `map` will be called $\lceil \frac{W}{\text{rhipe\_map\_buffsize}} \rceil$ times. The default value of *rhipe_map_buffsize* is 10,000 and is user configurable. Each time `map` is called, the vectors `map.keys` and `map.values` contain *rhipe_map_buffsize* keys and values respectively. If the objects are large it advisable to reduce the size of *rhipe_map_buffsize* . See the Airline examples where the value was set to 10 (each value was 5000x8 data frame).

**`reduce`** The general form the Reduce phase is best explained with this pseudo code

```
1 while more_keys_available() == TRUE
2   reduce_key <- get_new_key()
3   ...
4   while more_values_for_key_available()==TRUE
5     value <-  get_new_value_for_key()
6     ...
7   end while
8 end while
```

Each Reduce task is a partition of the intermediate keys produced as the output of the Map phase. The above code is run for every Reduce task. RHIPE implements the above algorithm by calling the R expression `reduce$pre` at line 3. In this expression, the user will have the new key present in `reduce.key`. After which RHIPE will

call `reduce$reduce` several times until the condition in line 4 is false. Each time `reduce\$reduce` is called, the vector `reduce.values` will contain a subset of the intermediate map values associated with `reduce.key`. The length of this vector is a default 10,000 but can be changed via the *rhipe_reduce_bufsize* option. Finally when all values have been processed, RHIPE calls `reduce$post` at line 7. At this stage, all intermediate values have been sent and the user is expected to write out the final results. Variables created in `reduce$pre` will be visible in the subsequent expressions. Thus to compute the sum of all the intermediate values,

```
1  reduce <- expression(
2    pre       = { s <- 0 },
3    reduce    = { s <- sum(s,unlist(reduce.values)) },
4    post      = { rhcollect(reduce.key,s) }
5  )
```

`reduce` is optional, and if not specified the map output keys will be sorted and shuffled and saved to disk. Thus it is possible to set `inout[2]` to *map* when the reduce expression is not given to obtain a MapFile. To turn of sorting and shuffling and instead write the map output to disk directly, set `mapred.reduce.tasks` to zero in `mapred`. In this case, the output keys are not sorted and the output format should not be *map*

**`combine`** If set to TRUE, RHIPE will run the `reduce` expression on the output of the `map` expression locally i.e. on the same computer that is running the associated map. For every *io.sort.mb* megabytes of key,value

**.** output from the map, the keys are sorted, and the expression `reduce` will be called for all keys and their associated values. The calling sequence of the elements of `reduce` is the same as above. The only difference is that the expression will not be sent *all* the values associated with the key.

If `combiner` is TRUE, the local reduction *will* be in invoked.

The outputs from the reduce are sorted and shuffled and sent to the Hadoop MapReduce reduce phase. Since the output from `map` is sent to `reduce` and the output from `reduce` is also sent to the `reduce` (during the final reduce phase of Hadoop MapReduce), the `reduce` expression must be able to handle input from the `map` and from `reduce`.

If `combiner` is TRUE, the `reduce` expression will be invoked during the local combine, in which case the output is intermediate and not saved as final output. The `reduce` expression also be invoked during the final reduce phase, in which case it will receive all the values associated with the key (note, these are values outputted when `reduce` is invoked as a combiner) and the output will be committed to the destination folder. To determine in which state `reduce` is running read the environment variable `rhipe_iscombining` which is '1' (also the R symbol `rhipe_iscombining` is equal TRUE) or '0' for the former and latter states respectively.

**`shared`** This is a character vector of files located on the HDFS. At the beginning of the MapReduce job, these files will be copied to the local hard disks of the Tasktrackers (cluster computers on which the compute nodes/cores are located). User provided R code can read theses files from the current directory (which is located on the local hard disk). For example, if */path/to/file.Rdata* is located on the HDFS and shared, it is possible to read it in the R expressions as `load('file.Rdata')`. Note, there is no need for the full path, the file is copied to the current directory of the R process.

**`setup` and `cleanup`** In RHIPE, each task is a sequence of many thousands of key, value pairs. Before running the `map` and `reduce` expression (and before any key, value pairs have been read), RHIPE will evaluate expressions in `setup` and `cleanup`. Each of these may contain the names `map` and `reduce` e.g `setup=list(map=,reduce=)` specific to the `map` and `reduce` expressions. If just an expressions is provided, it will be evaluated before both the Map phase and Reduce phase. The same is true for `cleanup`. Variables created, packages loaded in the `setup` expression will be visible in the `map` and the `reduce` expression but not both since both are evaluated in different R sessions. For an example, see *Streaming Data?*

**`ifolder`** This is a path to a folder on the HDFS containing the input data. This folder may contain sub folders in which case RHIPE use the all the files in the subfolders as input. This argument is optional: if not provided, the user must provide a value for `N` and set the first value of `inout` to `lapply`.

**ofolder** The destination of the output. If the destination already exists, it will be overwritten. This is not needed if there is not output. See *Downloading Airline Data*

**orderby** This is one of *bytes*, *integer* , *numeric* and *character*. The intermediate keys will be ordered assuming the output key in the map is of that type. If not of the type an exception will be thrown. Tuples can be sorted too, see *Tuple Sorting*

**inout** A character vector of one or two components which specify the formats of the input and output destinations. If `inout` is of length one this specifies the input format, the output being NULL (nothing is written) Components can be:

***sequence*** The keys and values can be arbitrary R objects. All the information of the object will be preserved. To extract a single key,value pair from a sequence file, either the user has to read the entire file or compose a MapReduce job to subset that key,value pair.

***text*** The keys, and values are stored as lines of text. If the input is of text format, the keys will be byte offsets from beginning of the file and the value is a line of text without the trailing newline. R objects written to a text output format are written as one line. Characters are quoted and vectors are separated by `mapred.field.separator` (default is space). The character used to separate the key from the value is specified in the `mapred` argument by setting `mapred.textoutputformat.separator` (default is tab). To not output the key, set `mapred.textoutputformat.usekey` to FALSE.

***map*** A map file is actually a folder consisting of sequence file and an index file. A small percentage of the keys in the sequence file are stored in the index file. Using the index file, Hadoop can very quickly return a value corresponding to a key (using `rhgetkey`). To create such an output format, use *map*. Note, the keys have to be saved in sorted order. The keys are sent to the `reduce` expression in sorted order, hence if the user does not modify `reduce.key` a query-able map file will be created. If `reduce.key` is modified, the sorted guarantee does not hold and RHIPE will either throw an error or querying the output for a key might return with empty results. MapFiles cannot be created if `orderby` is not *bytes*.

**copyFiles** Will the files created in the R code e.g. PDF output, be copied to the destination folder, `ofolder`?

**jobname** The name of the job, which is visible on the Jobtracker website. If not provided, Hadoop MapReduce uses the default name *job_date_time_number* e.g. `job_201007281701_0274`

**jarfiles** Optional JARs that need to be used during Hadoop MapReduce. This is used in the case when a user provides a custom InputFormat. Specify the JAR file to handle this InputFormat using this argument and specify the name of the InputFormat in the `mapred` argument.

**opts** RHIPE launches the C engine on the remote computers using the value found in `rhoptions()$opts$runner`. This is created from the local R installation which is possibly different from the Tasktrackers. If this is the case, specify the command that launches the R session via this parameter.

**N** To apply a computation to the numbers $1, 2, \ldots, N$ set `inout[1]` to `lapply` and specify the value of $N$ in this parameter. Set the number of map tasks in `mapred.map.tasks` (hence each task will run approximately $\lfloor \frac{N}{mapred.map.tasks} \rfloor$ computations sequentially).

**partitioner** A list of two names elements: `lims` and `type`. A partitioner forces all keys sharing the same property to be processed by one reducer. Thus, for these keys, the output of the reduce phase will be saved in one file. For example, if the keys were IP addresses e.g. `c(A,B,C,D)` where the components are integers, with the default partitioner, the space of keys will be uniformly distributed across the number of reduce tasks. If it is desired to store all IP addresses with the same first three ordinates in one file (and processed by one R process), use a partitioner as `list(lims=c(1:3), type='integer')`. RHIPE implements partitioners when the key is an atomic vector of the following type: integer, string, and real. The value of `lims` specifies the ordinates (beginning and end) of the key to partition on. The numbers must be positive. `lims` can be a single number. See *Streaming Data?*.

**mapred** Specify Hadoop and RHIPE options in this parameter (a list). For a full list of RHIPE options see *Options For RHIPE* and for Hadoop options go here.

## 7.2 Functions to Communicate with Hadoop during MapReduce

### 7.2.1 rhcollect - Writing Data to Hadoop MapReduce

```
rhcollect(key,value)
```

Called with two R objects. Sends a key,value pair to the Hadoop system. In the Map phase, it will pass it on for reduction if `mapred.reduce.tasks` is not zero (by default it is non zero) or it will be written to disk if `mapred.reduce.tasks` is zero. In the Reduce phase, it will be sent for further reduction if `reduce` is being run as a combiner or it will be written to the final destination if it is being run as the reducer. Don't forget to use this in the Map - if not called,nothing will be sent to the reducer.

### 7.2.2 rhcounter - Distributed Counters

```
rhcounter(group, name, value)
```

Increments (in a safe way i.e. no race conditions) the distributed counter `name` that belongs to family `group` by `value`. Ideally `group` and `name` should be strings, any R object can be sent and it will be converted to its string representation (see *String Representations*)

### 7.2.3 rhstatus - Updating the Status of the Job during Runtime

```
rhstatus(message)
```

Makes the string `message` visible on the Jobtracker website. This also informs Hadoop that the task is still running and it is not to be killed. In the absence of `rhstatus` and if `mapred.task.timeout` is non zero (by default it is 10 minutes) Hadoop will kill the R process.

### 7.2.4 rhsz,rhuz - Functions for Serialization

```
rhsz(object)
rhuz(rawobj)
```

The function `rhsz` serializes an object using RHIPE's binary serialization (see *RHIPE Serialization*). This will return the raw bytes corresponding the serialized object. If the object cannot be serialized, it will be converted to NULL and serialized. `rhuz` takes the bytes and un-serializes, throwing an error if it cannot. These two functions are also available at the R console. RHIPE uses the internals of these functions in `rhcollect` and `rhread`. The maximum size of the serialized object that can be read is 256MB. Larger objects will be written successfully, but when read RHIPE will throw an error. These functions are useful to get an approximate idea of how large an object will be.

CHAPTER

EIGHT

# RHIPE SERIALIZATION

## 8.1 About

The R serialization is verbose. Serialized objects have 22 bytes of header and booleans are serialized to integers. Best performance is achieved in Hadoop when the size of the data exchanged is as small as possible. RHIPE implements its own serialization using Google's Protocol Buffers . A benefit of using this is that the data produced by RHIPE can be read in languages such as Python, C and Java using the wrappers provided on the Google website.

However, a drawback of RHIPE's serialization is that not all R objects can be seamlessly serialized. RHIPE can serialize the following

- Scalar vectors: integers, characters (including UTF8 strings), numerics, logicals, complex and raw. `NA` values are accepted.
- Lists of the above.
- Attributes of objects. RHIPE can serialize data frames, factors, matrices (including others like time series objects) since these are the above data structure with attributes.

Closures, environments and promises cannot be serialized. For example, to serialize the output of `xyplot`, wrap it in a call to `serialize` e.g.

```
rhcollect(key, serialize(xyplot(a~b), NULL))
```

## 8.2 String Representations and TextOutput Format

RHIPE provides string representations of the above objects and is used when the output format in `rhmr` is *text*. The stringfying rules expand all scalar vectors and write them out as a line separated by *mapred.field.separator*. Thus the vector `c(1,2,3)` is written out as *1,2,3* if the value of *mapred.field.separator* is ",". The default value is `SPACE`. Strings are surrounded by *rhipe_string_quote* (default is double quote, to not surround strings set this to ''). Lists have their elements written out consecutively on a single line.

In the text output format, keys are written if *mapred.textoutputformat.usekey* is TRUE (default) and they are separated from the value by *mapred.textoutputformat.separator* (default is `TAB`). The options can be passed to RHIPE in the `mapred` parameter of `rhmr`.

## 8.3 Proto File

```
option java_package = "org.godhuli.rhipe";
option java_outer_classname = "REXPProtos";
message REXP {
  enum RClass {
    STRING = 0;
    RAW = 1;
    REAL = 2;
    COMPLEX = 3;
    INTEGER = 4;
    LIST = 5;
    LOGICAL = 6;
    NULLTYPE = 7;
  }
  enum RBOOLEAN {
    F=0;
    T=1;
    NA=2;
  }

  required RClass rclass = 1 ;
  repeated double  realValue      = 2 [packed=true];
  repeated sint32  intValue       = 3 [packed=true];
  repeated RBOOLEAN booleanValue   = 4;
  repeated STRING  stringValue    = 5;

  optional bytes  rawValue      = 6;
  repeated CMPLX   complexValue   = 7;
  repeated REXP          rexpValue      = 8;

  repeated string attrName = 11;
  repeated REXP  attrValue   = 12;
}
message STRING {
  optional string strval = 1;
  optional bool isNA = 2 [default=false];
}
message CMPLX {
  optional double real = 1 [default=0];
  required double imag = 2;
}
```

CHAPTER

NINE

# RHIPE OPTIONS

| Name | Description | Default |
|---|---|---|
| rhipe_map_buffsize | The length of map.keys and map.values | 10,000 |
| rhipe_reduce_buffsize | The length of reduce.values | 10,000 |
| io.sort.mb | The number of megabytes of intermediate output before it is locally reduced when combiner is true | Hadoop site file |
| mapred.textoutputformat.separator | Character that separates the key and value in text output | [tab] |
| mapred.field.separator | Character that separates the elements of vectors in text output | [space] |
| mapred.textoutputformat.usekey | For text output, if false, the key is not written | TRUE |
| mapred.map.tasks | The number of map tasks to divide the job into | For sequence and text file input, it is derived from the number of blocks. For lapply input, it is derived from *N* in the call to `rhmr` |
| mapred.reduce.tasks | The number of reduce tasks to launch to aggregate the intermediate keys | Hadoop site file |
| mapred.tasktimeout | The number of minutes after which the Tasktracker will terminate the R process. Set to zero, if the job consists of unpredictable long running tasks | 10 min |
| rhipe_stream_buffer | The size of the buffer in KB for transferring data between Java and R | 1MB |
| mapred.input.filename | the name of current input file for the given task, read it using `Sys.getenv` | |

Figure 9.1: Options useful for RHIPE and Hadoop

# BIBLIOGRAPHY


[STLa] Software and the concurrency revolution, H. Sutter and J. Larus, *ACM Queue, Volume 3,Number 7 2005*

[MapRed] MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat,*Communications of the ACM*, 2008

[HynFan] Hyndman, R. J. and Fan, Y. (1996) 'Sample quantiles in statistical packages', *American Statistician*, *50*, 361-365).

[SAS] Congestion in the Sky: Vizualising Domestic Airline Traffic with SAS, Rick Wicklin and Robert Allison, *SAS Institute*. http://stat-computing.org/dataexpo/2009/posters/wicklin-allison.pdf

[FLUSA] Delayed, Cancelled, On-Time, Boarding ... Flying in the USA, Heike Hofmann, Di Cook, Chris Kielion, Barret Schloerke, Jon Hobbs, Adam Loy,Lawrence Mosley, David Rockoff, Yuanyuan Sun, Danielle Wrolstad and Tengfei. Yin, *Iowa State University*, http://stat-computing.org/dataexpo/2009/posters/hofmann-cook.pdf

[ecuyer] rstream: Streams of Random Numbers for Stochastic Simulation,Pierre L'Ecuyer and Josef Leydold, http://cran.r-project.org/web/packages/rstream/index.html

[Masac] Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation, M. Mascagni and A. Srinivasan, *ACM Transactions on Mathematical Software*, pages 436-461,volume 26, 2000

[multicore] http://http://cran.r-project.org/web/packages/multicore/index.html

# INDEX

## S



## T