Mu Sigma

Introduction to RHadoop

Do The Math

Chicago, IL
Bangalore, India
www.mu-sigma.com

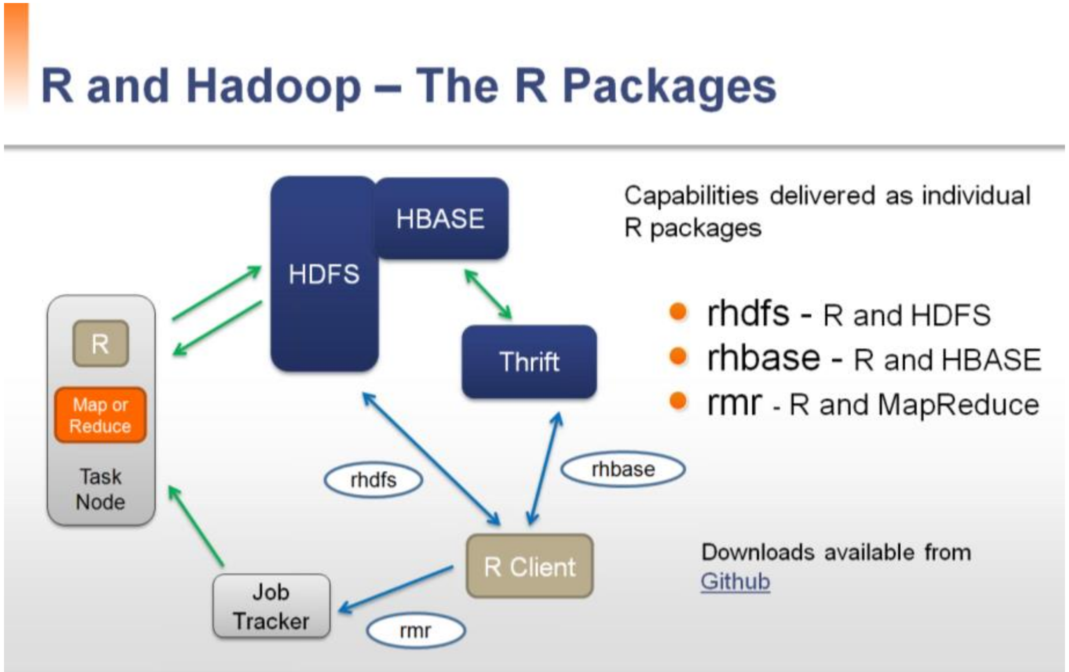
Jan - 2013

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix
  A: Useful links
  B: Overview of R functions used in this session
  C: More on Rhipe vs. RHadoop

R Packages to use for Map-Reduce

- Open source APIs for the Hadoop framework allowing users to define and run map-reduce jobs in R
- **RHadoop**
  - Maintained by Revolution Analytics
  - Designed for R programmers
  - Allows for more intuitive map-reduce programming
  - Easy syntax; uses functions
- **Rhipe**
  - Maintained by Saptarshi Guha
  - Uses Google's Protocol buffers for (faster and compact) serialization
  - Based on Hadoop streaming source
  - More complicated syntax; uses expressions

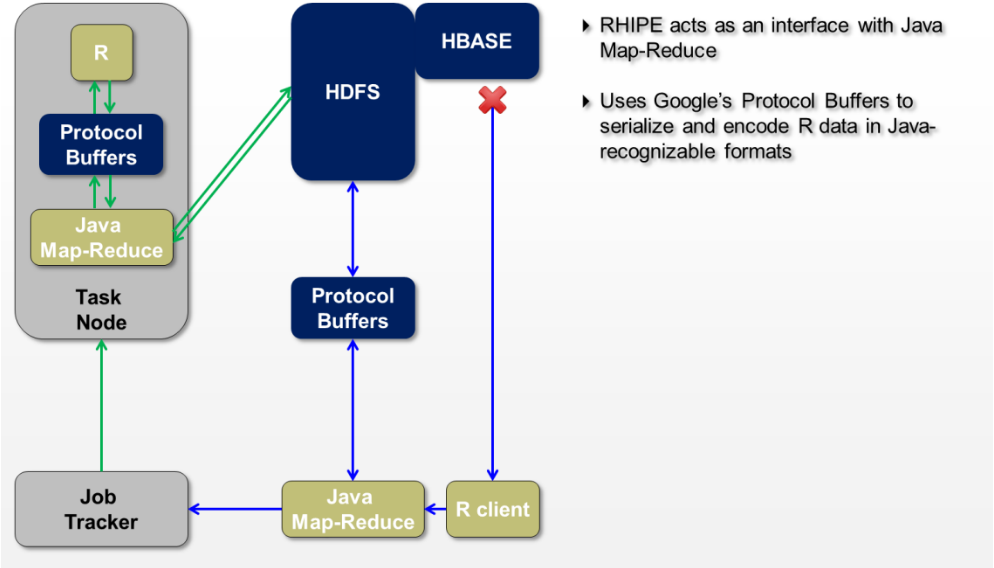


R and Hadoop – The R Packages

Capabilities delivered as individual R packages

- **rhdfs** - R and HDFS
- **rhbase** - R and HBASE
- **rmr** - R and MapReduce

Downloads available from Github



- RHIPE acts as an interface with Java Map-Reduce
- Uses Google's Protocol Buffers to serialize and encode R data in Java-recognizable formats

Each R package for Hadoop has advantages and disadvantages: We will proceed using RHadoop by Revolutions Analytics

- Hadoop streaming allows users to submit scripts written in most languages as the logic to be executed in the mappers and reducers
  - Kludgy alternative, lots of boilerplate code for ensuring inputs and outputs are serialized and de-serialized properly
  - Notoriously hard to debug

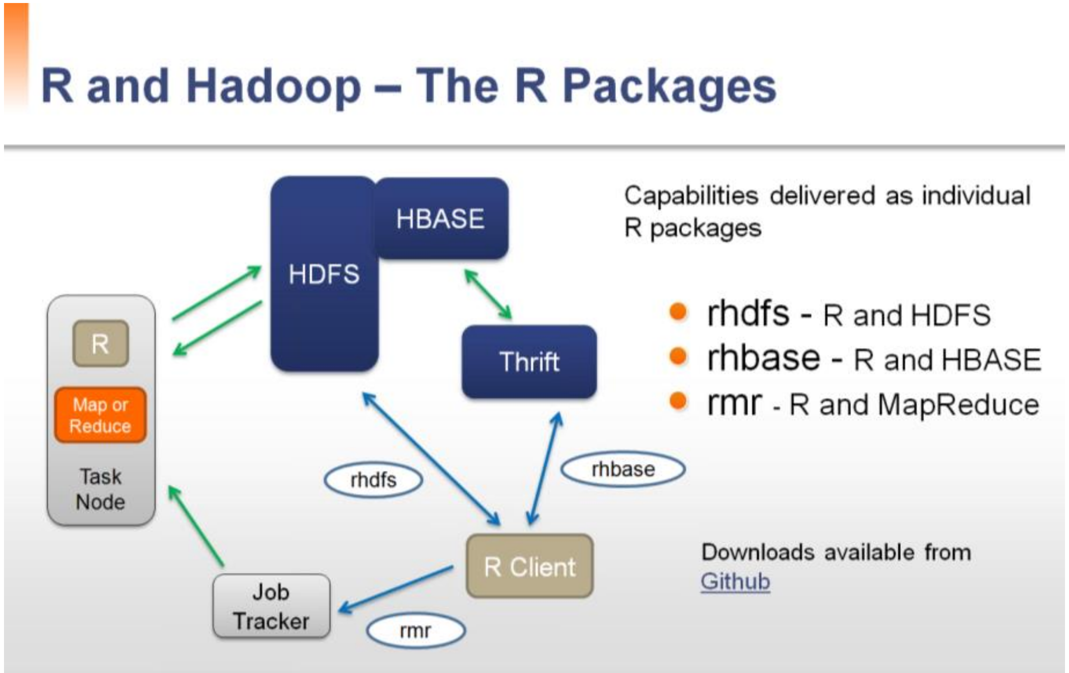
- Rhipe – an R package written and maintained by Saptarshi Guha, it provides an R-interface to Hadoop streaming and uses Google’s protocol buffers for serializing
  - Better than streaming since we can write and check logic in a running session of R
  - However, coding style is not congruent with R; it feels weird to use
  - Syntactical complexity increases with complexity of logic being implemented

- RHadoop – an R package written and maintained by Revolutions Analytics, it provides another R-interface to Hadoop streaming and used JSON for serializing
  - Insulates the coder from Java options; very R-centric
  - Coding style is congruent with R; completely functional coding style; more ‘comfortable’ for R coders
  - Syntactical complexity does not increase with complexity of logic as much as Rhipe
  - Can pass many native R objects directly as keys and values
  - JSON is a more popular and widely supported

Rhadoop is a collection of three components for interacting with the Hadoop ecosystem

- `rmr` – This package provides the means for submitting map-reduce jobs
  - Provides the `mapreduce` function for submitting jobs to Hadoop
  - Provides important functions like `keyval`, `to.mapper`, `to.reducer`, `to.dfs`, `from.dfs`, and `equijoin`
- `rhdfs` – This package provides the means to “talk” to the Hadoop Distributed File System (HDFS)
  - Provides important utility functions for interacting with the filesystem like `hdfs.file`, `hdfs.read`, `hdfs.write`, `hdfs.line.reader`, `hdfs.chmod`, `hdfs.chown`, `hdfs.move`, `hdfs.rename`, `hdfs.mkdir`, and `hdfs.ls`

- `rhbase` – This package provides the means to “talk” to HBASE and access stored within
  - Out of immediate scope since this package only allows for interaction with HBASE tables
  - Not currently possible to define map-reduce jobs to run over HBASE tables
  - Package still under active development



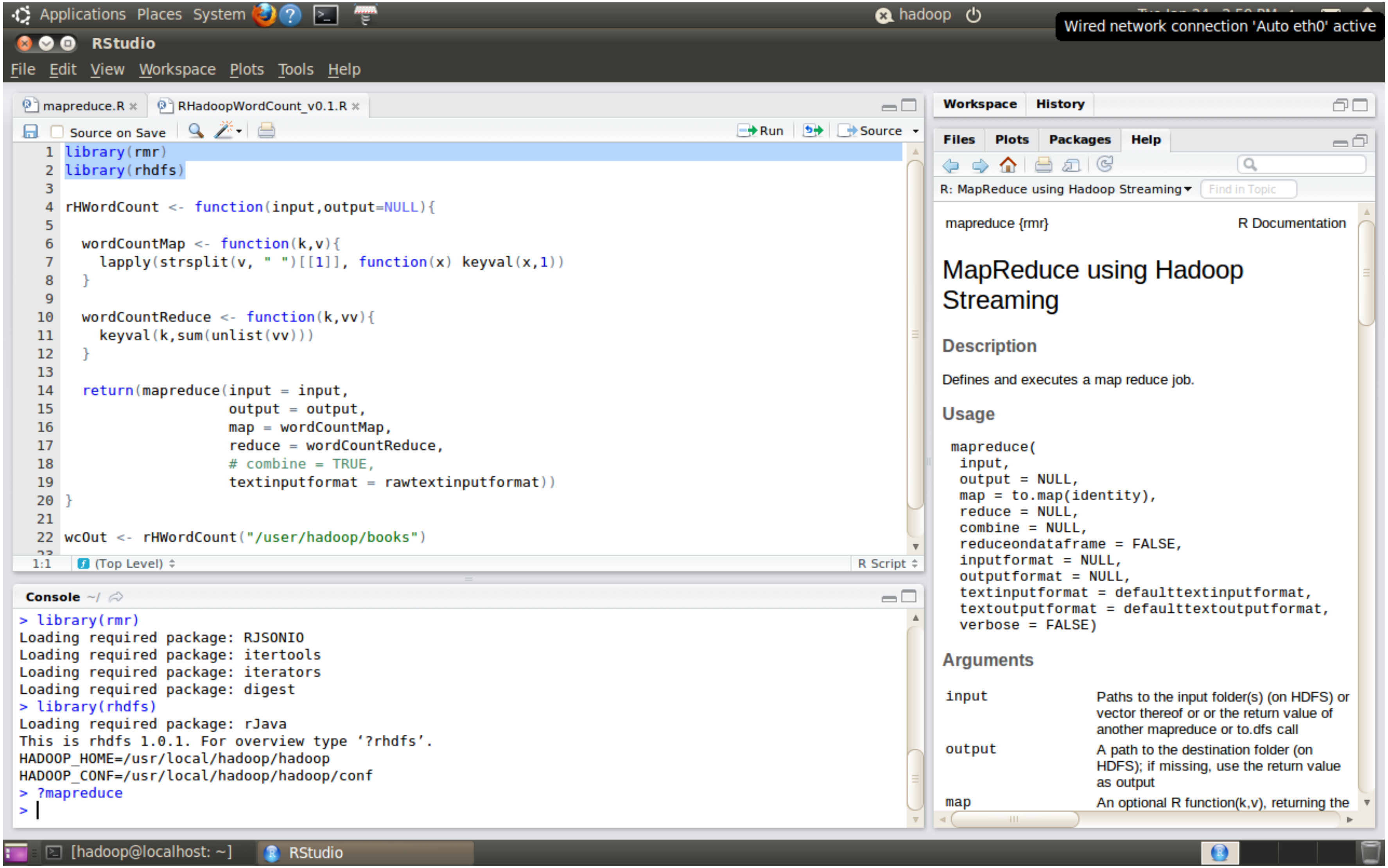
R and Hadoop – The R Packages

Capabilities delivered as individual R packages

- rhdfs - R and HDFS
- rhbase - R and HBASE
- rmr - R and MapReduce

Downloads available from Github

Using Rstudio to run map-reduce in R



```r
library(rmr)
library(rhdfs)

rHWordCount <- function(input,output=NULL){

  wordCountMap <- function(k,v){
    lapply(strsplit(v, " ")[[1]], function(x) keyval(x,1))
  }

  wordCountReduce <- function(k,vv){
    keyval(k,sum(unlist(vv)))
  }

  return(mapreduce(input = input,
                   output = output,
                   map = wordCountMap,
                   reduce = wordCountReduce,
                   # combine = TRUE,
                   textinputformat = rawtextinputformat))
}

wcOut <- rHWordCount("/user/hadoop/books")
```

```
> library(rmr)
Loading required package: RJSONIO
Loading required package: itertools
Loading required package: iterators
Loading required package: digest
> library(rhdfs)
Loading required package: rJava
This is rhdfs 1.0.1. For overview type '?rhdfs'.
HADOOP_HOME=/usr/local/hadoop/hadoop
HADOOP_CONF=/usr/local/hadoop/hadoop/conf
> ?mapreduce
> 
```

mapreduce {rmr} R Documentation

MapReduce using Hadoop Streaming

Description

Defines and executes a map reduce job.

Usage

```
mapreduce(
  input,
  output = NULL,
  map = to.map(identity),
  reduce = NULL,
  combine = NULL,
  reduceondataframe = FALSE,
  inputformat = NULL,
  outputformat = NULL,
  textinputformat = defaulttextinputformat,
  textoutputformat = defaulttextoutputformat,
  verbose = FALSE)
```

Arguments

`input`	Paths to the input folder(s) (on HDFS) or vector thereof or or the return value of another mapreduce or to.dfs call
`output`	A path to the destination folder (on HDFS); if missing, use the return value as output
`map`	An optional R function(k,v), returning the

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix
  - A: Useful links
  - B: Overview of R functions used in this session
  - C: More on RHIPE vs. RHadoop

Function keyval: Generating a Key-value pair in R

- The function `keyval` is the R equivalent to the pseudo-code “emit” function seen in the last session; it takes two arguments as its input

Syntax

```
keyval(key, val)
keys(kv)
values(kv)
```

Example – Generate a key value pair of first 10 integers and their squares.

```
keyval(1:10, (1:10)^2)
```

- `key`: (any object) the desired key(s)
- `val`: (any object) the desired value(s) associated with the key(s)
- `kv`: A key-value pair

- **keyval returns one or more key-value pairs.**
- A key value object should be always considered vectorized, meaning that it defines a collection of key-value pairs. For the purpose of forming key-value pairs, the length of an object is considered its number of rows when defined, that is for matrices and data frames, or its R length otherwise.

Functions map and reduce: Mappers and reducers in R

- Map and reduce functions in RHadoop take two input arguments each

```
map = function(key, value){
      …
      return(keyval(key, value))
}

reduce = function(key, values){
      …
      return(keyval(key, value))
}
```

  - The mapper must take as input a (`key, value`) pair
  - The reducer must take as input a pair consisting of a `key` and list of all it’s associated `values`
  - **Both must return key-value pairs**

Function mapreduce: rmr function that submits map-reduce jobs

- The `mapreduce` function submits any specified logic as a streaming job to Hadoop

```
mapreduce(input, output=NULL, map=to.map(identity), reduce=NULL, combine=NULL,
     input.format = "native", output.format = "native",
     backend.parameters = list(), verbose = TRUE)
```

  - `input` & `output` – character strings representing the paths of the input and output data – for multiple inputs, the inputs argument is specified as a character vector containing all input paths
  - `map` & `reduce` – the names of the mapper and reducer functions
  - `combine` – the name of the combiner function (if separate from the reducer) or `TRUE` (in case the reducer can be used) – `NULL` by default
  - `input.format` & `output.format` – specify the formats to be used to read in the input data and write the final output data – default to “`native`”, R’s native serialization format.
  - `backend.parameters` – Specify additional, backend-specific options, e.g. to set the number of mappers or reducers. It is recommended not to use this argument to change the semantics of mapreduce.
  - `verbose` – a boolean option to run Hadoop in verbose mode
  - **Returns a string containing the location of the output on the HDFS**

Functions to.map and to.reduce: rmr utility functions

- `to.map` and `to.reduce` are two utility functions provided by `rmr` that allow users to easily convert existing functions into mappers and reducers

```
to.map(function1, function2){
     …
     return(mapper(key, value))
}

to.reduce(function1, function2){
     …
     return(reducer(key, values))
}
```

  - If only `function1` is specified, it is applied to the input key and the input value individually.
  - If `function1` and `function2` are specified then it is applied `function1` to the input key and `function2` to the input value
  - **Return functions that act as mappers/reducers**

Functions to.map and to.reduce: rmr utility functions continued

- If one wished to simply apply a log transformation to the some column of input data, no reducer is really required (since # of input rows = # of output rows)

- The identity function in R is ideally suited to act as reducer in such cases; this is done by simply using the `identity` function as an input to the `to.reduce` function to produce an identity reducer

```
to.reduce(identity)
```

- Similarly, in cases where reducing is not required, like aggregations, we can simply use the `identity` function as an input to the `to.map` function to produce an identity mapper

```
to.map(identity)
```

Functions to.dfs and from.dfs: rmr utility functions for testing

- It is often useful to test map-reduce logic on some test data; the utility functions `to.dfs` and `from.dfs` allow users to write to and read from the HDFS

```
to.dfs(<a list of key-value pairs>){
     …
     return(<a reference to the list on the HDFS>)
}

from.dfs(<a list of key-value pairs on the HDFS>){
      …
     return(<an in-memory list read off the input on the HDFS>)
}
```

- `to.dfs` can be used to write sample data from local memory to the HDFS, using syntax like

```
input = to.dfs(lapply(1:10, function(i) keyval(i, i^2)))
```

- `from.dfs` can then be used to read the outputs of the `mapreduce` call from the HDFS into local memory using syntax like

```
output = from.dfs(mapreduce(…))
Or a previously stored mapreduce result
output = from.dfs('/user/hadoop/lorem.txt')
```

Functions equijoin: rmr utility function to join two input datasets

- Many map-reduce jobs involve the relational joining of two input datasets; the `equijoin` function performs such tasks

```
equijoin(left.input, right.input, input, output, outer, map.left, map.right, reduce,
     reduce.all){
     …
     return(<a reference to the output on the HDFS>)
}
```

- `left.input` & `right.input` – The left and right side inputs to the join – `NULL` by default
- `input` – The only input in case of a self join; mutually exclusive with the previous two – `NULL` by default
- `output` – a character representing the path where output should be written to – `NULL` by default (in which case output is written to a temporary location on the HDFS)
- `outer` – the type of outer join to be performed – takes one of three defaults values (as character strings), `"left"`, `"right"` and `"full"`
- `map.left` & `map.right` – The functions to apply to each record from the left and right inputs, they follow the same conventions as any map function – the returned key becomes the join key
- `reduce` – the function to be applied for each key on the lists of associated values produced by `map.left` and `map.right`; – **returns 0 or more key-value pairs like any reduce function**
- `reduce.all` – function to be applied to each triple comprising a key, the left and right values associated with that key – **returns 0 or more key-value pairs like any reduce function**
- **Returns a string containing the location of the output on the HDFS**

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix
  A: Useful links
  B: Overview of R functions used in this session
  C: More on RHIPE vs. RHadoop

The rhdfs package provides utilities for interacting with HDFS

- Most functions provided by the `rhdfs` package will not be of frequent use within a map-reduce context – however, the important functions are summarized here for completeness' sake

- `hdfs.file`, `hdfs.read`, `hdfs.write` and `hdfs.line.reader` are provided to enable users to read from and write to files in the HDFS
  - `hdfs.file` is used to open a connection to a file; it takes two primary character arguments, the first specifying the path of the file that is to be read from/written to, the second specifies the mode, `"r"` for reading and `"w"` for writing
  - `hdfs.read` is used to read from files in the HDFS; it takes as input three arguments, the first being an open connection to a file (returned by `hdfs.file`), the second an integer specifying the number of bytes to be read and the third specifying the position to read from
  - `hdfs.write` is used to write to a file on the HDFS; it takes as primary input two arguments; the first one specifying the R object to be written and the second one specifying an open connection to a file
  - `hdfs.close` is used to close a connection opened using `hdfs.file`
  - `hdfs.line.reader` is used to read lines from a file on the HDFS; it takes as input two primary arguments, the first a character specifying the path of the file to be read and the second an integer argument specifying the number of lines to be read

```
hdfs.line.reader(path="/user/hadoop/TrainingDatasets/RHadoop/IntroductionToRHadoop/R
HadoopBooksData.txt/Dracula.txt", n=25)
```

- Apart from `hdfs.line.reader`, the remaining functions all perform byte reads and writes – not immediately important

The rhdfs package provides utilities for interacting with HDFS continued

- `hdfs.chmod` and `hdfs.chown` are provided to set access levels and ownership for files and directories in the HDFS
  - `hdfs.chmod` takes two primary character arguments, the first specifying the path of the file/directory for which permissions need to be changed and the second specifying the permissions
  - `hdfs.chown` takes three primary character arguments, the specifying the path of the file/directory for which ownership needs to be changed, the second specifying the new owner and the third specifying the group of the new owner

- `hdfs.copy`, `hdfs.move`, `hdfs.rename`, `hdfs.rm`, `hdfs.mkdir`, and `hdfs.ls` allow the user to browse the HDFS, move, modify and delete files
  - `hdfs.copy`, `hdfs.move` and `hdfs.rename` all take as primary input two character arguments in a “from” - “to” syntax
  - `hdfs.copy` copies the contents of the “from” argument to the “to” argument; `hdfs.move` moves them (think Ctrl+C vs. Ctrl+X)
  - `hdfs.rename` changes the name of the object (file/directory) in the “from” argument to the “to” argument
  - `hdfs.rm` takes as primary input one character string specifying the object to be deleted
  - `hdfs.mkdir` and hdfs.ls take as primary input one character argument specifying a path on the HDFS
  - `hdfs.mkdir` creates the path specified
  - `hdfs.ls` lists the contents of the path specified

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix

  A: Useful links

  B: Overview of R functions used in this session

  C: More on RHIPE vs. RHadoop

There are many options of reading and writing data provided for by RHadoop

- Most inputs to map-reduce begin life as an input file in some common format – tab-delimited, comma-separate, etc

- However, the outputs of such jobs are not constrained to be in the same format; since reducers write out key-value pairs, it makes sense for the user to be able to define these as R objects – the JSON format supports this

- The input and output formats may be of the following types:
  - `"text"`: free text, useful mostly on the input side for NLP type applications
  - `"json"`: one or two tab separated, single line JSON objects per record
  - `"csv"`: comma-separated values
  - `"native"`: (**default**): uses the internal R serialization, offers the highest level of compatibility with R data types
  - `"sequence.typedbytes"`: sequence file (in the Hadoop sense) where key and value are of type typedbytes, which is a simple serialization format used in connection with streaming for compatibility with other hadoop subsystems

- It is very easy to specify custom formats in rmr using `make.input.format()` and `make.output.format()` functions

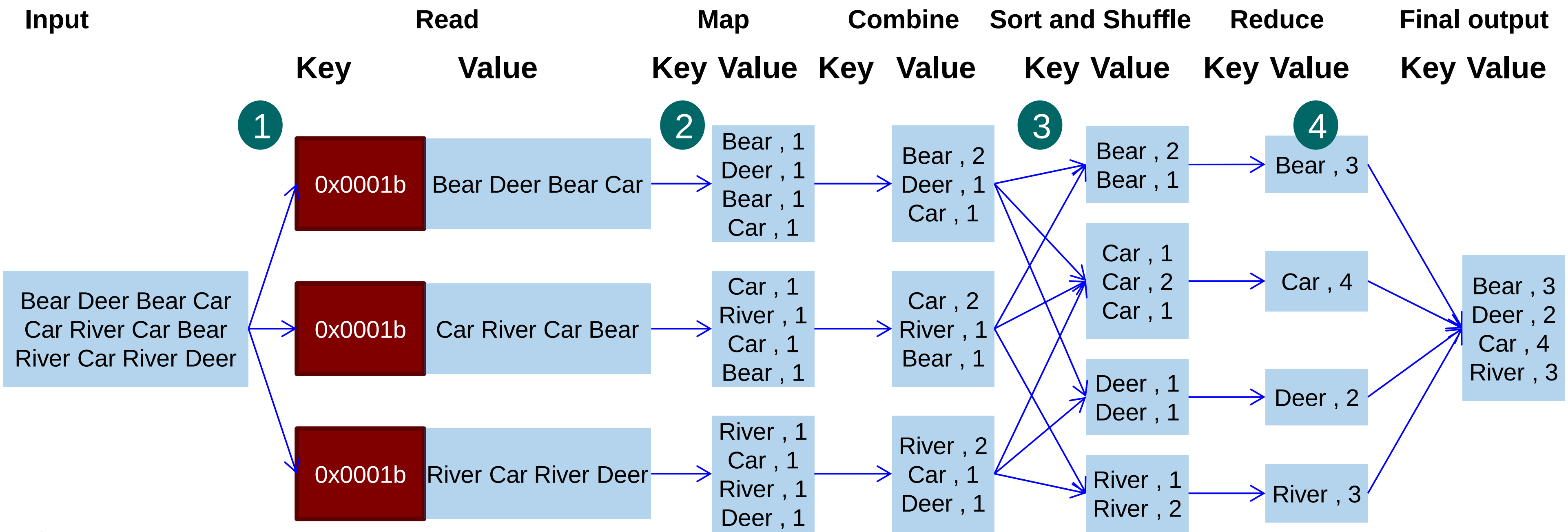
Agenda

- Map-reduce in R - An overview
- The rmr package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix
  - A: Useful links
  - B: Overview of R functions used in this session
  - C: More on RHIPE vs. RHadoop

<u>Example 1</u>: Word count using map-reduce

	Input	Output
map	<K1,V1>	<K2,V2>
reduce	<K2, List(V2)>	<K3,V3>

The over all map-reduce word-count process



① **By default, the line address of every line is they key and value is the contents of the line. These Key-Value pairs will be the input to the map function**

② **Each Mapper will generate new Key-Value pairs based on the map function, In this case new key is the word and value is 1.**

③ **Key-Value pairs will be merged on the same key before sending it to the reducer. In this case, since word is the key, so all the key-value pairs associated with same word are merged**

④ **Finally, the reducer function run on the values associated to same key, and produces the result**

Example 1 continued: Map-reduce word counting in R

- The R code for implementing the word-count map-reduce logic is detailed below [**RHadoopWordCountBooksExampleCode.R**]

```
wc = function(input, output = NULL) {
     map = function(k, v)
              keyval(unlist(strsplit(v, " ")), 1L)

     reduce = function(k, vv)
              keyval(k, sum(vv))

     mrOut = mapreduce(input=input, output=output,
                       map=map, reduce=reduce, combine=TRUE,
                       input.format="text”)
     return(from.dfs(mrOut))
}

wc(input="/user/hadoop/TrainingDatasets/RHadoop/IntroductionToRHadoop/
RHadoopBooksData.txt")
```

NOTE : Descriptions of R functions used in appendix

Example 2: A rough-and-ready equijoin example

- Consider two lists of key-value pairs:
  - This example uses the `to.dfs` function to write to the key-value pair lists of length 9
  - Both lists have keys from 1-9
  - The left input list contains values that are squares of the keys
  - The right input list contains values that are cubes of the keys
  - These lists get joined on their keys in `equijoin`, then `from.dfs` is used to read the joined output back into local memory

Key	Value
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

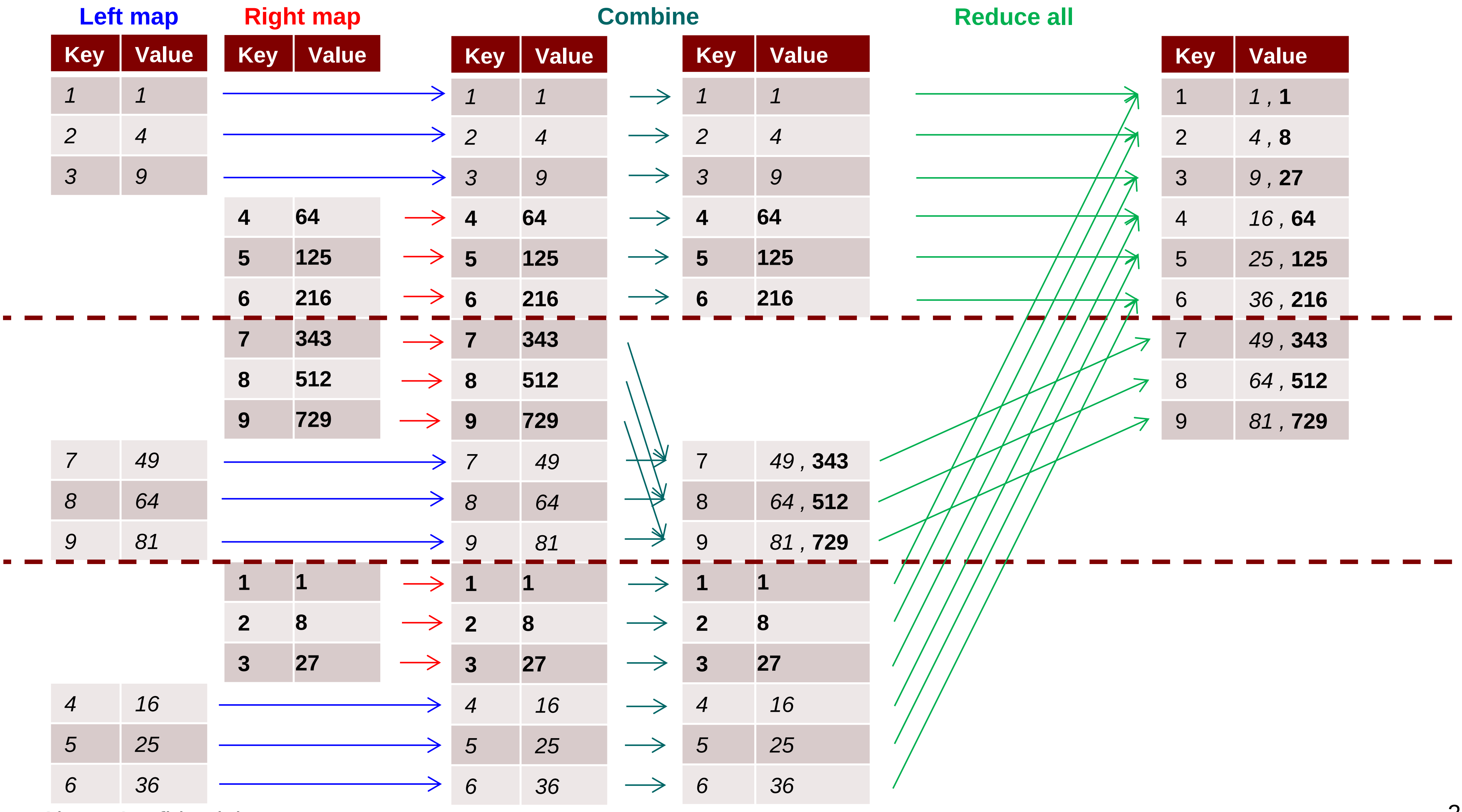
Key	Value
1	1
2	8
3	27
4	64
5	125
6	216
7	343
8	512
9	729

[**RHadoopEquijoinExampleCode.R**]

```
from.dfs(equijoin(
     left.input = to.dfs(keyval(1:9, (1:9)^2)),
     right.input = to.dfs(keyval(1:9, (1:9)^3))))
```

NOTE : Descriptions of R functions used in appendix

Example 2 continued: A rough-and-ready equijoin example



Left map

Key	**Value**
1	*1*
2	*4*
3	*9*
7	*49*
8	*64*
9	*81*
4	*16*
5	*25*
6	*36*

Right map

Key	**Value**
4	**64**
5	**125**
6	**216**
7	**343**
8	**512**
9	**729**
1	**1**
2	**8**
3	**27**

Combine

Key	**Value**
1	*1*
2	*4*
3	*9*
4	**64**
5	**125**
6	**216**
7	**343**
8	**512**
9	**729**
7	*49*
8	*64*
9	*81*
1	**1**
2	**8**
3	**27**
4	*16*
5	*25*
6	*36*

Key	**Value**
1	*1*
2	*4*
3	*9*
4	**64**
5	**125**
6	**216**
7	*49* , **343**
8	*64* , **512**
9	*81* , **729**
1	**1**
2	**8**
3	**27**
4	*16*
5	*25*
6	*36*

Reduce all

Key	**Value**
1	*1* , **1**
2	*4* , **8**
3	*9* , **27**
4	*16* , **64**
5	*25* , **125**
6	*36* , **216**
7	*49* , **343**
8	*64* , **512**
9	*81* , **729**

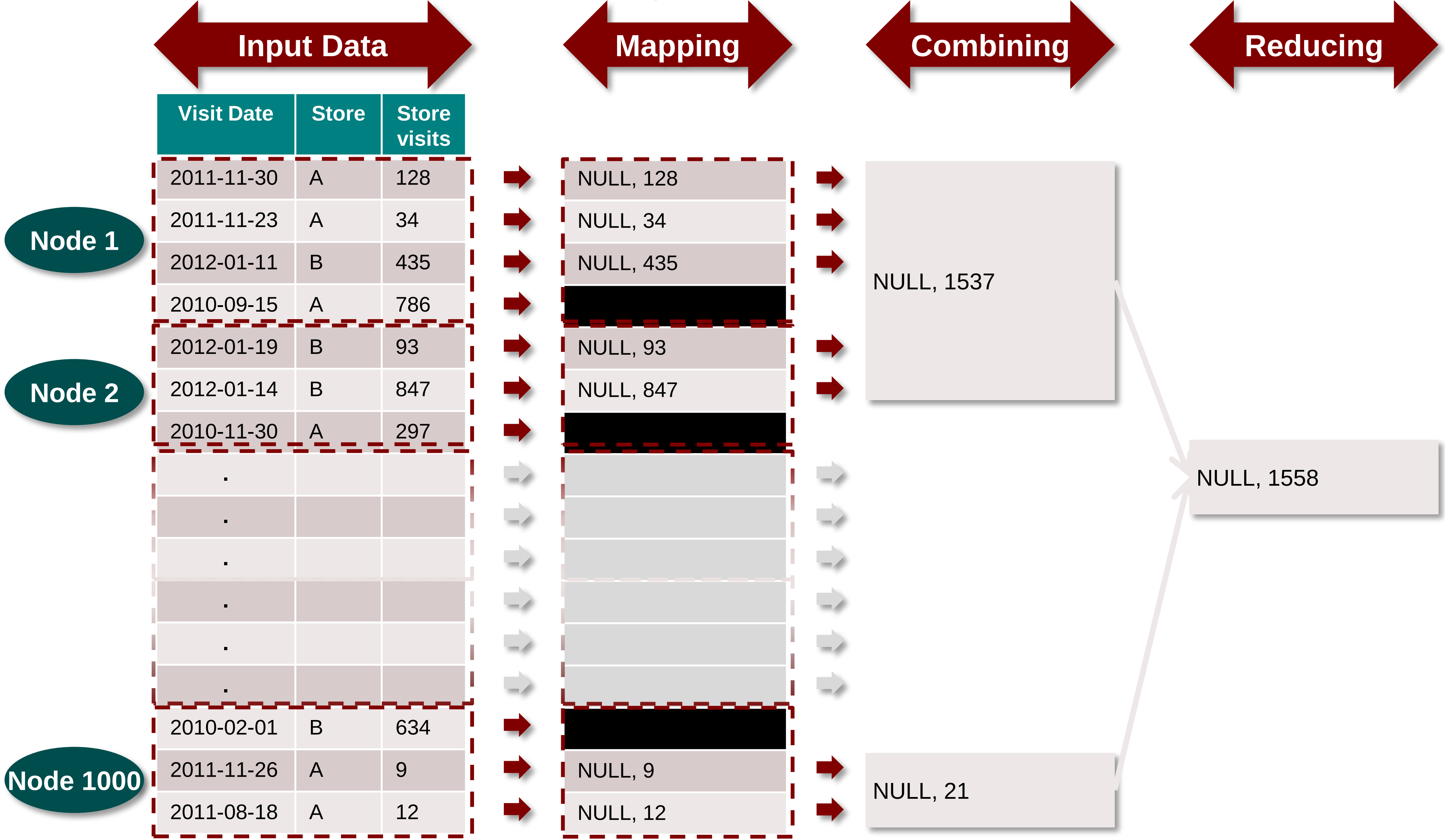
Example 3: Problem statement

- Consider data that tracks store daily visits at all outlets of a retail chain and looks like the following:
  - How would you compute total store visits for all stores after 31 January 2011?
  - How would you compute store-wise total visits after 31 January 2011 for each store?

Store Visits

Visit Date	Store	Store visits
2011-11-30	A	128
2011-11-23	A	34
2012-01-11	B	435
2010-09-15	A	786
2012-01-19	B	93
2012-01-14	B	847
2010-11-30	A	297
2010-07-25	B	218
2011-07-04	B	454
2010-10-19	A	23
2010-02-01	B	634
2011-11-26	A	9
2011-08-18	A	12
…	…	…

Example 3 continued: Filtering and summation steps



Visit Date	Store	Store visits
2011-11-30	A	128
2011-11-23	A	34
2012-01-11	B	435
2010-09-15	A	786
2012-01-19	B	93
2012-01-14	B	847
2010-11-30	A	297
.		
.		
.		
.		
.		
.		
2010-02-01	B	634
2011-11-26	A	9
2011-08-18	A	12

NULL, 128
NULL, 34
NULL, 435
NULL, 93
NULL, 847
NULL, 9
NULL, 12

NULL, 1537
NULL, 21

NULL, 1558

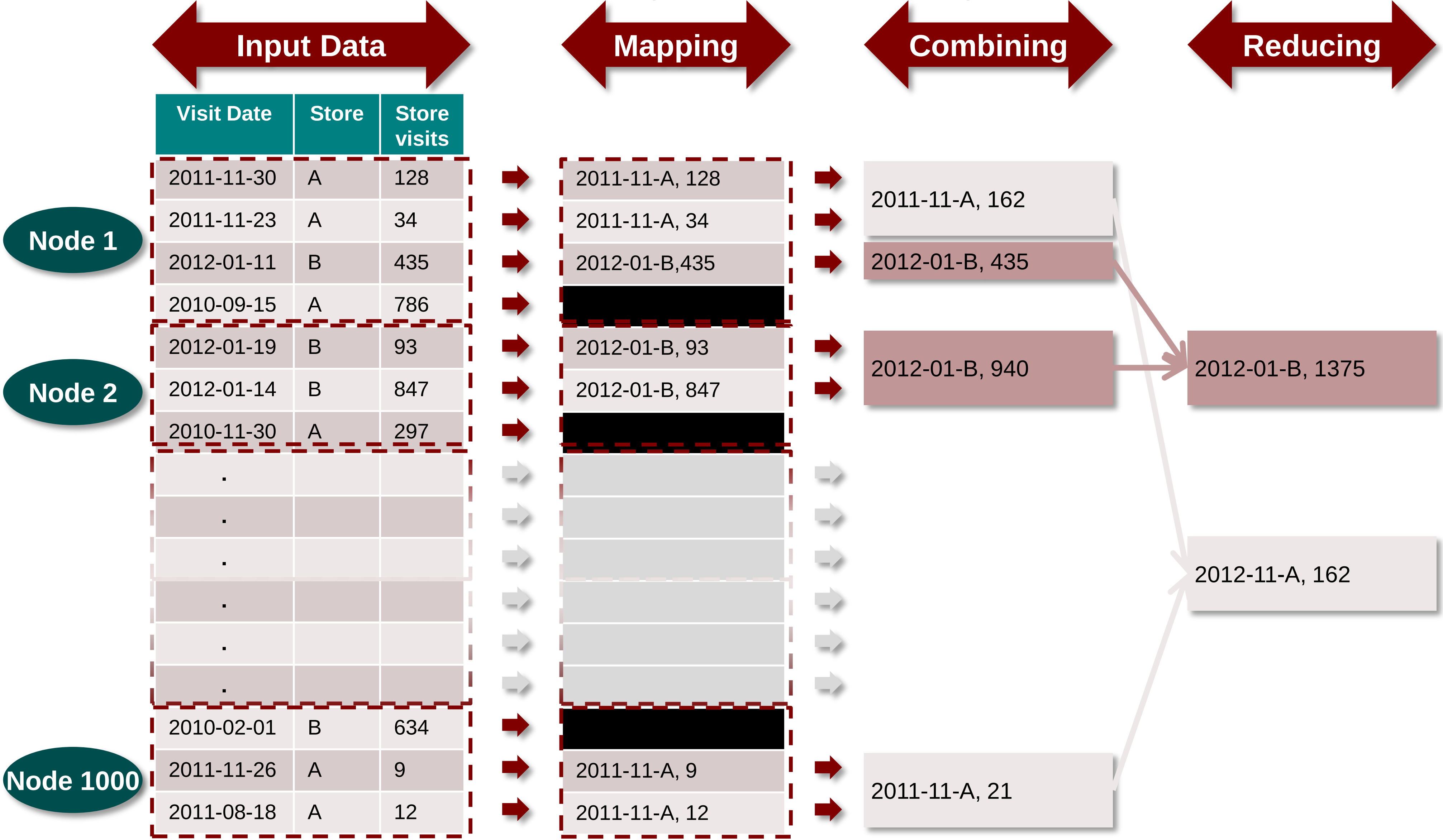
Example 3 continued: Filtering and totaling in R

- Consider the summing examples from the previous session; if the input data were now to be in the form of a comma-separated file, the R code for summing store visits (in column 3) for all rows corresponding to store visits later than 31 January 2011 (in column 1) would be [**RHadoopFilterTotalStoreVisitsExampleCode.R**]

```
total.visits = function(input, output=NULL, date.ref="2011-01-31") {
     map = function(k, v) {
              rows = which(as.POSIXlt(as.character(v[, 1L])) > as.POSIXlt(date.ref))
              keyval(1L, sum(v[rows, 3L]))
     }

     reduce = function(k, vv)
              keyval(k, sum(vv))

     mrOut = mapreduce(input=input, output=output,
              map=map, reduce=reduce, combine=TRUE,
              input.format=make.input.format("csv", sep=","))
     values(from.dfs(mrOut))
}
total.visits("/user/hadoop/TrainingDatasets/RHadoop/IntroductionToRHadoop/
RHadoopStoreVisitsData.csv")
```

NOTE : Descriptions of R functions used in appendix

Example 3 continued: Filtering and computing monthly subtotals



Input Data

Mapping

Combining

Reducing

Node 1

Node 2

Node 1000

Visit Date	Store	Store visits
2011-11-30	A	128
2011-11-23	A	34
2012-01-11	B	435
2010-09-15	A	786
2012-01-19	B	93
2012-01-14	B	847
2010-11-30	A	297
.		
.		
.		
.		
.		
.		
2010-02-01	B	634
2011-11-26	A	9
2011-08-18	A	12

2011-11-A, 128

2011-11-A, 34

2012-01-B,435

2012-01-B, 93

2012-01-B, 847

2011-11-A, 9

2011-11-A, 12

2011-11-A, 162

2012-01-B, 435

2012-01-B, 940

2011-11-A, 21

2012-01-B, 1375

2012-11-A, 162

Example 3: Filtering and subtotaling in R redux – using utility functions in the map-reduce function call

- The R code for computing monthly subtotals of store visits for each store (column two) for visits after 31 January 2011 would be [**RHadoopFilterSubtotalStoreVisitsExampleCode.R**]

```
subtotal.visits = function(input, output=NULL, date.ref="2011-01-31") {
  map = function(k, v) {
    rows = which(as.POSIXlt(as.character(v[, 1L])) > as.POSIXlt(date.ref))
    v = v[rows,]
    ksplit = strsplit(as.character(v[,1L]), "-")
    key = character(length(rows))
    for (i in c(1L:length(rows)))
      key[i] = paste(ksplit[[i]][1L], ksplit[[i]][2L], v[i, 2L], sep="-")
    val = as.integer(v[, 3L])
    keyval(key, val)
  }
  reduce = function(k, vv) keyval(k, sum(vv))

  mrOut = mapreduce(input=input, output=output, map=map, reduce=reduce,
      combine=TRUE, input.format=make.input.format("csv", sep=","))
  return(from.dfs(mrOut))
}
subtotal.visits(input="/user/hadoop/TrainingDatasets/RHadoop/
IntroductionToRHadoop/RHadoopStoreVisitsData.csv")
```

Best practices for rmr2

- Use a **small sample data set** first. Run the code on the actual data set only when there is no doubt that the code is bug free.

- **Test all functions** (input.format, map and reduce) individually in order.

- There are three Hadoop modes, namely standalone (local), pseudo-distributed and distributed. You should **start debugging in standalone mode**. You can use debug to debug your function in this mode. Syntax: `rmr.options(backend="local")`

- **Move on to distributed mode** if your code runs correctly in local. You cannot use debug in this mode. To see the outputs, write to the `stderr()` stream, and check the user logs. Syntax: `rmr.options(backend="hadoop")`.

- The preferred way of processing recursive lists is to **avoid using `unlist()`** and loop through the contents using `lapply()`. This is especially true for non-uniform recursive lists.

- **Reduce early, reduce often** – This is because the data shuffling and sorting takes maximum amount of time, and the process speeds up as the amount of data in this stage is reduced. A combiner does exactly this and should be used whenever possible.

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix

  A: Useful links

  B: Overview of R functions used in this session

  C: More on RHIPE vs. RHadoop

Composing map-reduce jobs in R using RHadoop

- Exercise 1 – write a map-reduce job to perform grouped averaging as seen in the previous session
  - Column 2 contains the grouping variable
  - Column 3 contains the variable to be averaged
  - How can you create and write this data into the HDFS?

- Exercise 2 – write a map-reduce job to perform matrix transposition and multiplication as seen in the previous session
  - Assume the same input matrix of size 5×2
  - How can you create and write this matrix into the HDFS?
  - How can this code be extended to perform OLS?

Agenda

- MapReduce in R - An overview
- The rmr2 package – Map-reduce jobs in R
- The rhdfs package – Interacting with HDFS
- Input/Output formats – Different options for reading and writing data
- Examples for discussion
- Exercises
- Appendix
  - A: Useful links
  - B: Overview of R functions used in this session
  - C: More on Rhipe vs. RHadoop

Useful links

- https://github.com/RevolutionAnalytics/RHadoop
- https://github.com/RevolutionAnalytics/RHadoop/wiki
- https://github.com/RevolutionAnalytics/RHadoop/wiki/Tutorial
- https://github.com/RevolutionAnalytics/RHadoop/wiki/Efficient-rmr-techniques
- https://github.com/RevolutionAnalytics/RHadoop/blob/4efbd435aff3d52cfea116b663100baf637035cc/rmr/pkg/docs/introduction-to-vectorized-API.md
- http://www.revolutionanalytics.com/news-events/free-webinars/2011/r-and-hadoop/
- http://blog.revolutionanalytics.com/2011/09/mapreduce-hadoop-r.html
- http://www.slideshare.net/RevolutionAnalytics/rhadoop-r-meets-hadoop
- https://github.com/saptarshiguha/RHIPE
- http://www.datadr.org/

An overview of R functions used in this session

- `lapply(list, fun)`
  - Applies the function `fun` to each element of `list`
  - `lapply(list(1,2,3) function(x) x^3)` yields a list containing the elements 1, 8 and 27

- `strsplit(string, separator)`
  - Splits the input `string` (or character vector) on the basis of the `separator`
  - Returns a list as long as the input vector; each list element contains a character vectors containing the sub-strings separated by the `separator` argument
  - `strsplit("2011-01-31", "-")` will yield a list of length 1, whose only element will be a character vector containing the values “2011”, “01” and “21”

- `sum(…)`
  - Returns the sum of all input arguments specified in `…`
  - `sum(1,2,3)` will yield 6

- `unlist(list)`
  - Converts the input `list` to a vector
  - `unlist(list(1,2,3))` will yield a numeric vector with the elements 1, 2 and 3

An overview of R functions used in this session continued

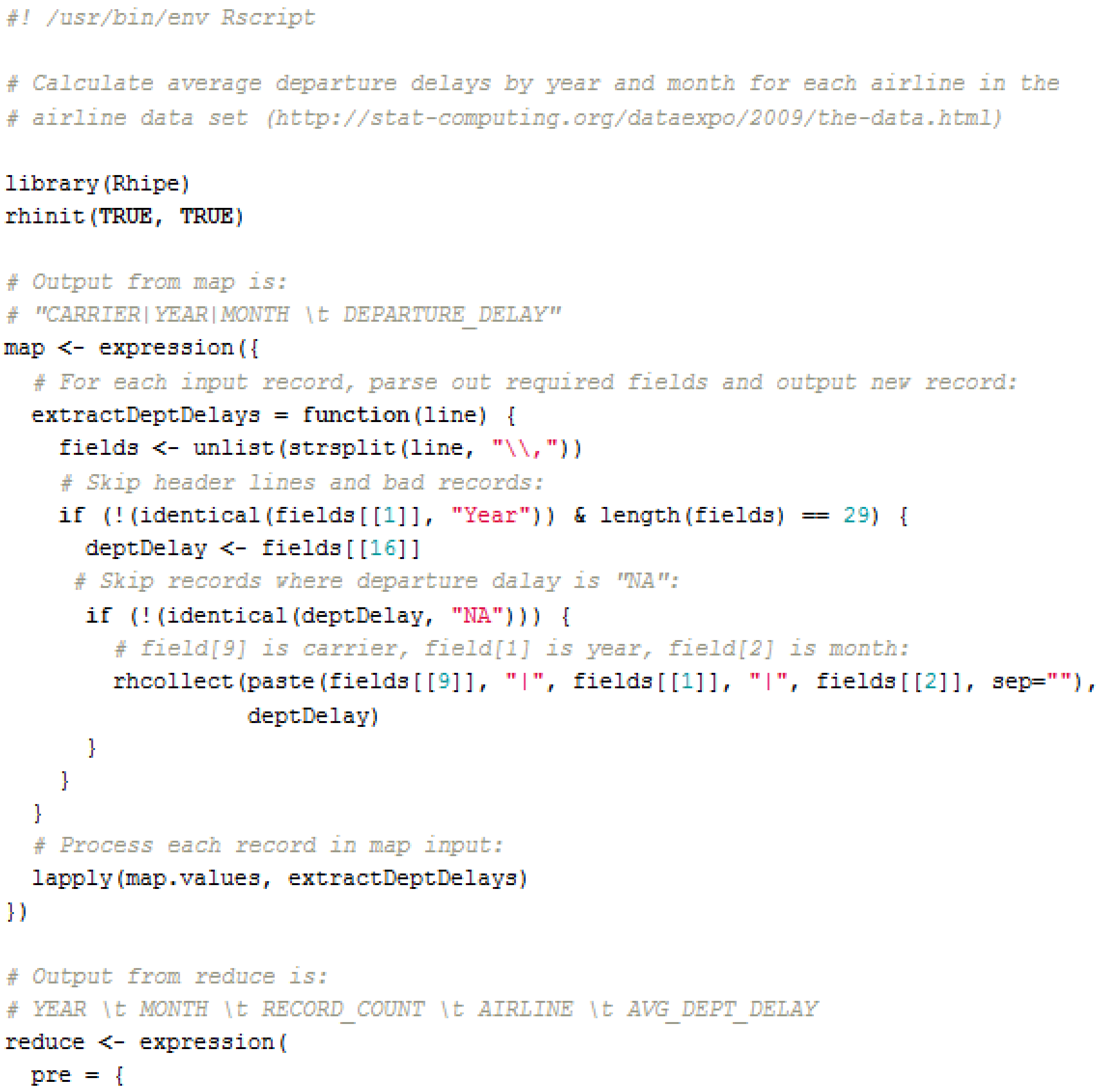
- `as.POSIXlt(character)`
  - Converts input `character` into a date variable, measured in number of seconds elapsed since some origin (this is usually 01-January-1900)
  - Origin varies from system to system in R – usually 1 January 1900
  - `as.POSIXlt("2010-01-01")$year` will yield the number of years elapsed since the origin date
    - » 110 in case the origin is 01-Jan-1900

- `paste(vector/list, collapse, sep)`
  - Acts as a concatenation function;
  - If input is a vector or a list, returns a string with the elements of the vector/list separated by the string specified in `collapse`
  - If input is a set of strings, returns a concatenation of the input strings separated by the string specified in `sep`
  - `paste(list(1, 2), collapse = "-")` yields “1-2”
  - `paste("R", "hadoop", sep = "")` yields “Rhadoop”

To better understand the differences between RHIPE and RHadoop let us consider some sample code snippets

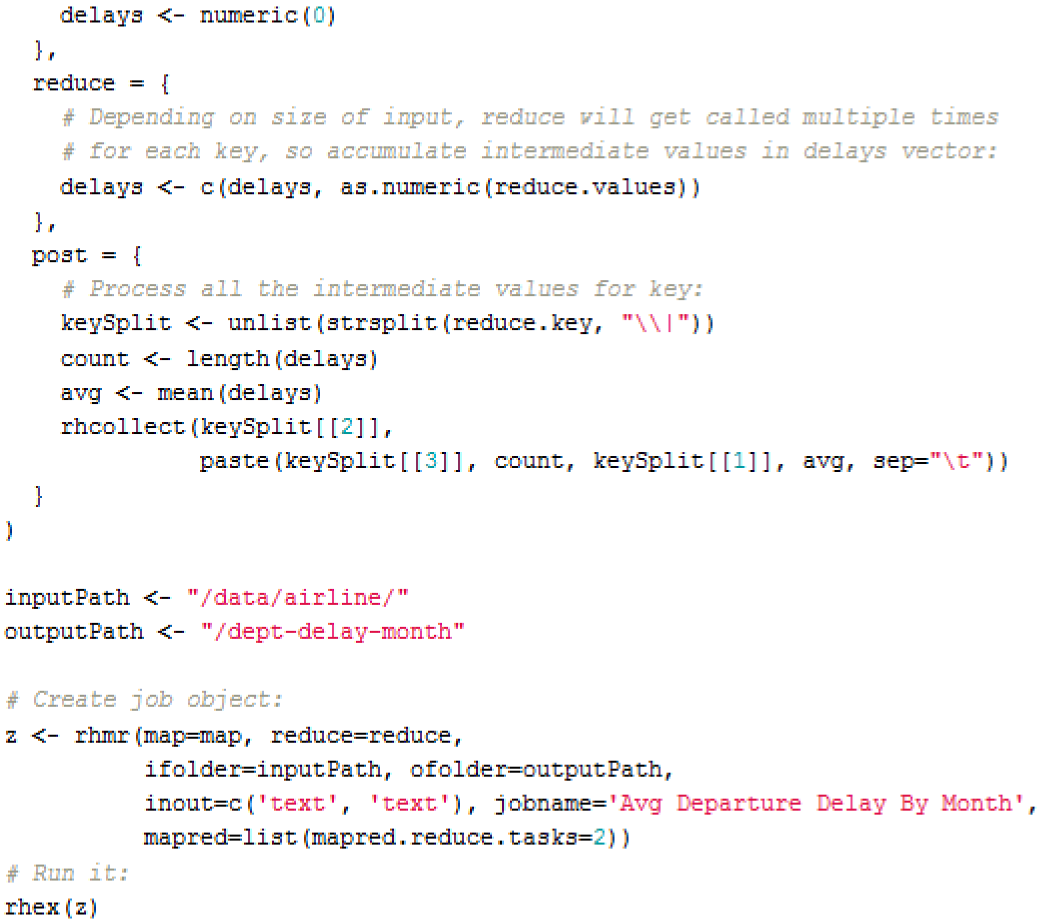
- Based on the airline dataset: http://stat-computing.org/dataexpo/2009/the-data.html
- Dataset contains departure and arrival times for flights between 1987 and 2008
- The code computes average departure delay by year and month for each airline
- Two code snippets implementing the same logic:
  - First code in Rhipe
  - Second code in RHadoop
- Code snippets sourced from
  - https://github.com/jseidman/hadoop-R/blob/master/airline/src/deptdelay_by_month/R/rhipe/rhipe.R
  - https://github.com/jseidman/hadoop-R/blob/master/airline/src/deptdelay_by_month/R/rmr/deptdelay-rmr.R
- Rhipe code is longer, syntactically convoluted
- RHadoop code is shorter and easier to understand

RHIPE vs. RHADOOP – RHIPE implementation (1)



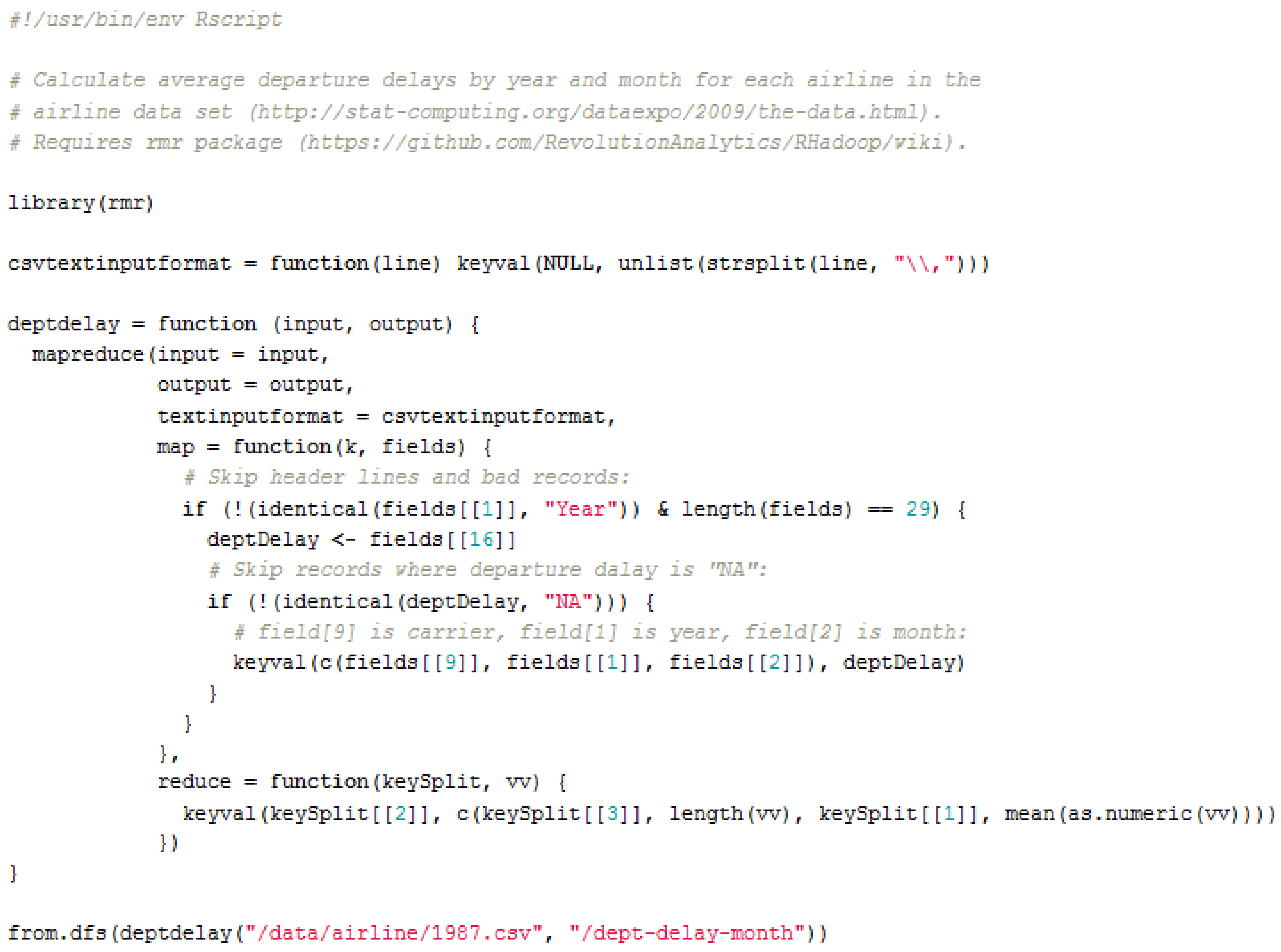
```
#! /usr/bin/env Rscript

# Calculate average departure delays by year and month for each airline in the
# airline data set (http://stat-computing.org/dataexpo/2009/the-data.html)

library(Rhipe)
rhinit(TRUE, TRUE)

# Output from map is:
# "CARRIER|YEAR|MONTH \t DEPARTURE_DELAY"
map <- expression({
  # For each input record, parse out required fields and output new record:
  extractDeptDelays = function(line) {
    fields <- unlist(strsplit(line, "\\,"))
    # Skip header lines and bad records:
    if (!(identical(fields[[1]], "Year")) & length(fields) == 29) {
      deptDelay <- fields[[16]]
     # Skip records where departure dalay is "NA":
      if (!(identical(deptDelay, "NA"))) {
        # field[9] is carrier, field[1] is year, field[2] is month:
        rhcollect(paste(fields[[9]], "|", fields[[1]], "|", fields[[2]], sep=""),
                  deptDelay)
      }
    }
  }
  # Process each record in map input:
  lapply(map.values, extractDeptDelays)
})

# Output from reduce is:
# YEAR \t MONTH \t RECORD_COUNT \t AIRLINE \t AVG_DEPT_DELAY
reduce <- expression(
  pre = {
```

RHIPE vs. RHADOOP – RHIPE implementation (2)



```
    delays <- numeric(0)
  },
  reduce = {
    # Depending on size of input, reduce will get called multiple times
    # for each key, so accumulate intermediate values in delays vector:
    delays <- c(delays, as.numeric(reduce.values))
  },
  post = {
    # Process all the intermediate values for key:
    keySplit <- unlist(strsplit(reduce.key, "\\|"))
    count <- length(delays)
    avg <- mean(delays)
    rhcollect(keySplit[[2]],
              paste(keySplit[[3]], count, keySplit[[1]], avg, sep="\t"))
  }
)

inputPath <- "/data/airline/"
outputPath <- "/dept-delay-month"

# Create job object:
z <- rhmr(map=map, reduce=reduce,
          ifolder=inputPath, ofolder=outputPath,
          inout=c('text', 'text'), jobname='Avg Departure Delay By Month',
          mapred=list(mapred.reduce.tasks=2))
# Run it:
rhex(z)
```

RHIPE vs. RHADOOP – RHADOOP implementation



```
#!/usr/bin/env Rscript

# Calculate average departure delays by year and month for each airline in the
# airline data set (http://stat-computing.org/dataexpo/2009/the-data.html).
# Requires rmr package (https://github.com/RevolutionAnalytics/RHadoop/wiki).

library(rmr)

csvtextinputformat = function(line) keyval(NULL, unlist(strsplit(line, "\\,")))

deptdelay = function (input, output) {
  mapreduce(input = input,
            output = output,
            textinputformat = csvtextinputformat,
            map = function(k, fields) {
              # Skip header lines and bad records:
              if (!(identical(fields[[1]], "Year")) & length(fields) == 29) {
                deptDelay <- fields[[16]]
                # Skip records where departure dalay is "NA":
                if (!(identical(deptDelay, "NA"))) {
                  # field[9] is carrier, field[1] is year, field[2] is month:
                  keyval(c(fields[[9]], fields[[1]], fields[[2]]), deptDelay)
                }
              }
            },
            reduce = function(keySplit, vv) {
              keyval(keySplit[[2]], c(keySplit[[3]], length(vv), keySplit[[1]], mean(as.numeric(vv))))
            })
}

from.dfs(deptdelay("/data/airline/1987.csv", "/dept-delay-month"))
```

Thank You

Chicago, IL
Bangalore, India
Jan 2013
www.mu-sigma.com

Proprietary Information