Big Data Analytics with R

Distributed File Systems &
MapReduce Programming Model

Yihuang K. Kang

Introduction

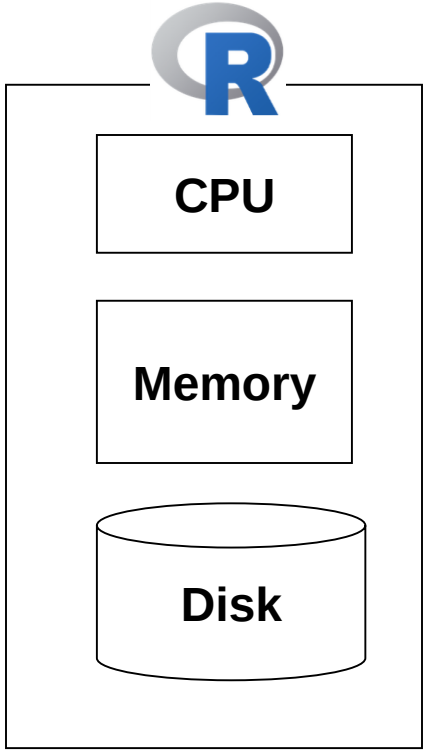
- In this unit, we will be discussing the most popular big data analytics framework—*Apache Hadoop* and its *MapReduce* programming models.

- Hadoop provides a set of tools that cope with distributed storage and computing systems. Its MapReduce provide a programming model that simplifies the implementations of distributed data processing tasks. Users only need to specify a *Map*() function that processes key-value pairs of data, and a *Reduce*() function that merges/aggregates all intermediate values associated with the same intermediate key.

- We will be using *RHadoop*, an set of R packages that allows you to easily get access to files/folders in distributed file systems, and implement MapReduce steps that process big datasets.

Single Node Computing

- We have been using R to do almost everything in one machine.



Data Management

Statistical Analysis

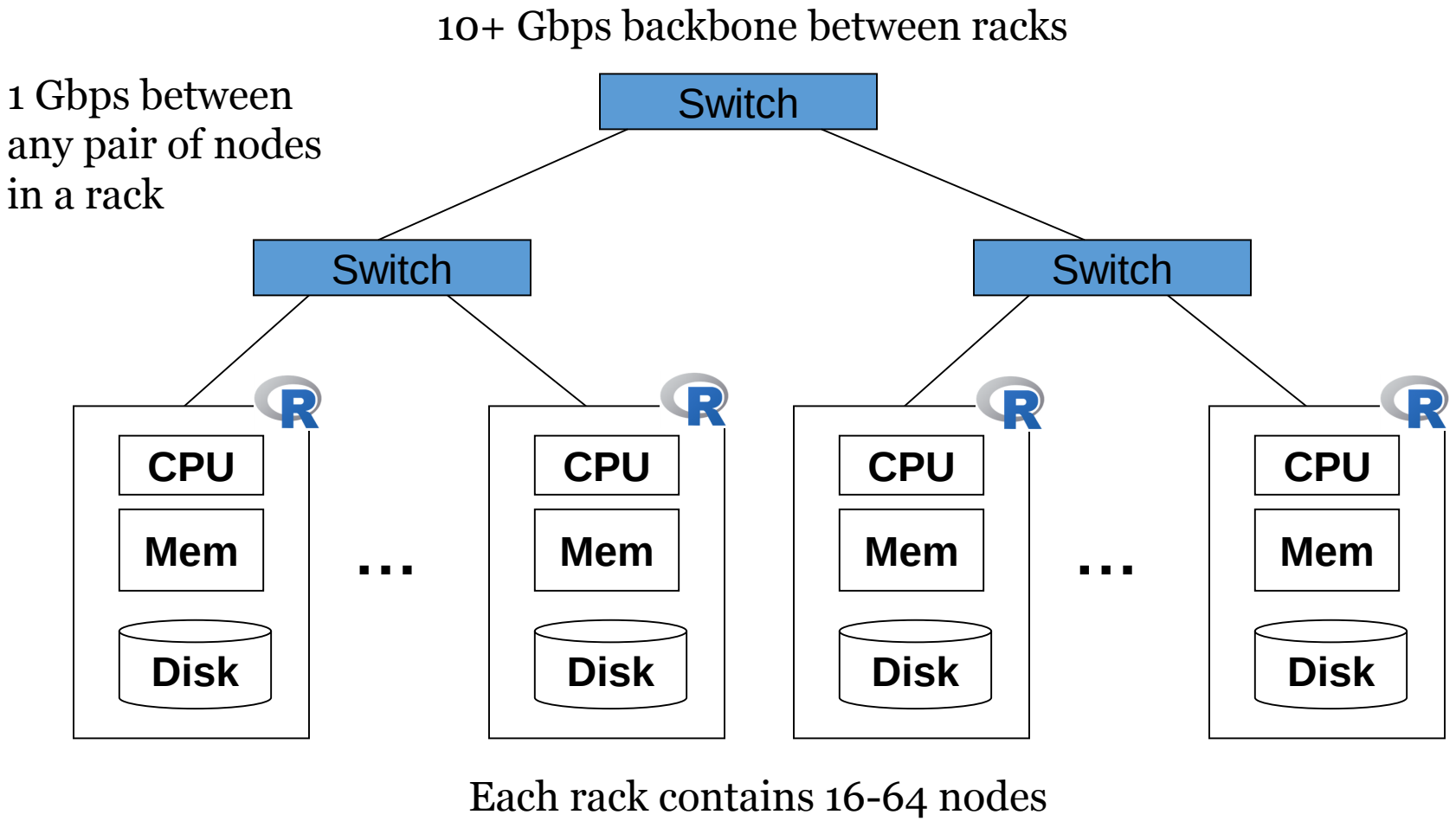
Visualization

⋮

- However, what if our dataset is too big to be processed and analyzed in one single machine?

Cluster Computing

Cluster Computing(cont.)



Problems

- How do you distribute computation tasks? say, fit a simple linear model?
- How can we make it easy to write distributed programs that run on a cluster of machines?
- How do you handle machines failures and recoveries?

Solutions (possible!)

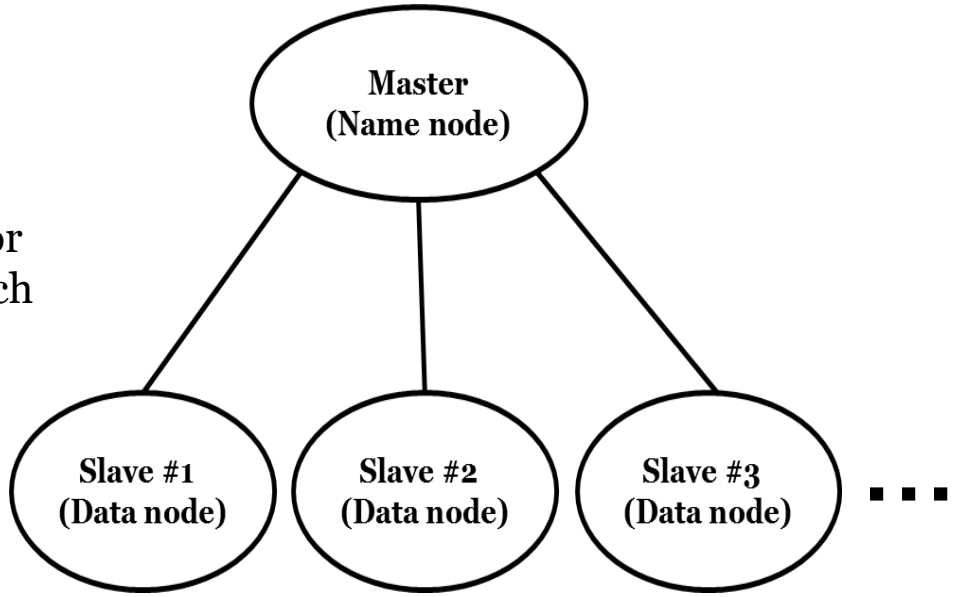
- Although recently considered outdated & inefficient, Google's distributed file systems and programming model, *Google File System*(GFS) and the MapReduce, have addressed the problems of distributed computations and processing failure recovery.

- Open source implementations of the GFS and MapReduce—*Hadoop Distributed File System* (HDFS) and MapReduce of Hadoop framework have been released for years. Hadoop is now considered one of the most popular frameworks that handle the problems of processing massive data.

- New large-scale data processing frameworks, such as *Apache Spark* & *H2o*, and their new *DataFrame* & *Datasets* APIs have been proposed to deal with such distributed data analytics tasks. Just like working with local data frames in R or Python (e.g. *pandas*), we can now manipulate and analyze big datasets with these APIs in more elegant and efficient ways.

Hadoop Distributed File System

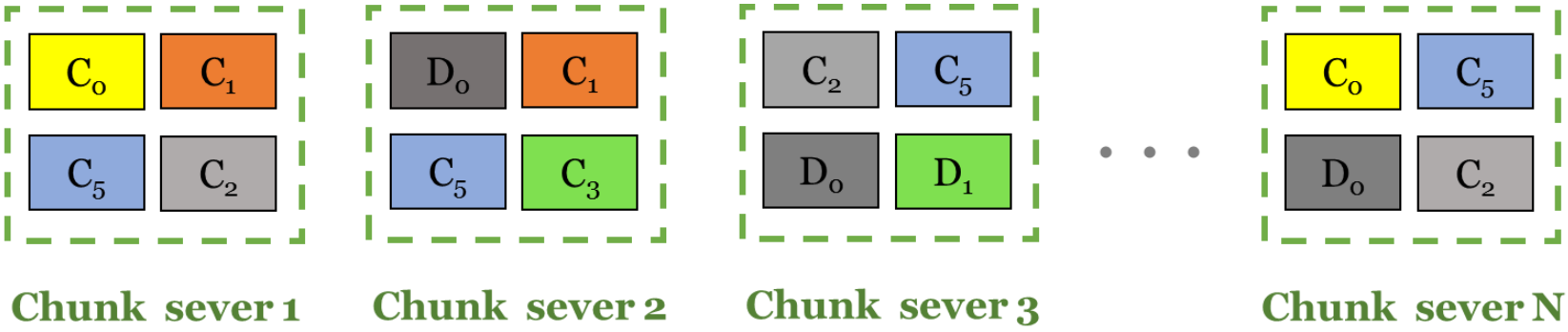
- **Chunk servers (data nodes)**
  - File is split into contiguous chunks.
  - Each chunk /block is 64MB (128MB for Hadoop 2.0+) by default, which is much larger than the block if typical file system (e.g. 4 KB for Linux)
  - Each chunk replicated 3x and kept in data nodes.
- **Master node (name nodes)**
  - Stores metadata about where files are stored.
  - Initialize data process tasks.
  - If it is down, everything is gone!
  - Might be replicated (secondary name nodes).



- **Client library for file access**
  - Talks to Master to find chunk servers .
  - Connects directly to chunk servers to access data.

Hadoop Distributed File System(cont.)

- Data nodes also serve as computing servers. Hadoop can bring distributed computations directly to each chunk of the data file. Data kept in "chunks" (or formally called "blocks") are replicated and spread across machines. Such architectures also provide seamless recovery from disk or machine failures.



- You can check current files (and locations of their blocks) in HDFS via CM's Hadoop name node portal: http://hdp.cm.nsysu.edu.tw:50070/explorer.html (not available outside of CM).

Working with files in HDFS

- To work with HDFS, you must have a pre-installed Apache Hadoop server (with YARN for Hadoop 2.0) in either *pseudo-distributed* mode (single-node cluster for testing) or *distributed* mode (multi-node cluster). Those of you who have access to CM Hadoop server (*hadoop.cm.nsysu.edu.tw*) can simply login to the server via secure shell with port number 22 or via RStudio Server Terminal.
- HDFS provides a file system shell and a set of commands very similar to the systems of most Unix-like systems (e.g. bash).

```
# Use your command line tools/terminals. For MS Windows user, you
# need an SSH client software, e.g. putty
$ ssh hdp.cm.nsysu.edu.tw -p 22 # if you're OSX/Linux/BSDs users.
# List all file in "/" of HDFS of CM Hadoop server
$ hadoop fs -ls /
Found 3 items
drwxr-xr-x   - hadoop supergroup          0 2016-10-21 01:35 /home
drwxrwxrwx   - hadoop supergroup          0 2016-10-20 10:08 /tmp
drwxr-xr-x   - hadoop supergroup          0 2016-10-21 01:35 /user
```

Working with files in HDFS(cont.)

- To start using Hadoop to analyze your big datasets, you must put your data files in the HDFS first. (You might now upload your files to our server via RStudio or SSH, if you have access to SSH). Enter the following command in your terminal to copy files/folders from local file system to the HDFS (or the other way around). Note that you need to have read & write permissions to the folders in HDFS.

```
# E.g. You have a plain text file, text.txt, and would like
# to upload it to /home/myfolder (yours should be /home/"yourAccount").
# Remember to check out the name of your own home folder at CM server)
$ hadoop fs –put text.txt /home/myfolder
# From HDFS to local file system
$ hadoop fs –get /home/myfolder/text.txt ./test_from_HDFS.txt
```

- You can actually do most of the abovementioned operations in R.

```
library("rhdfs"); hdfs.init() # Initializing connection to the HDFS
hdfs.put("test.txt", "/home/myfolder/test_2.txt")
hdfs.ls("/home/myfolder")
hdfs.cat("/home/myfolder/test.txt")
```

Working with files in HDFS(cont.)

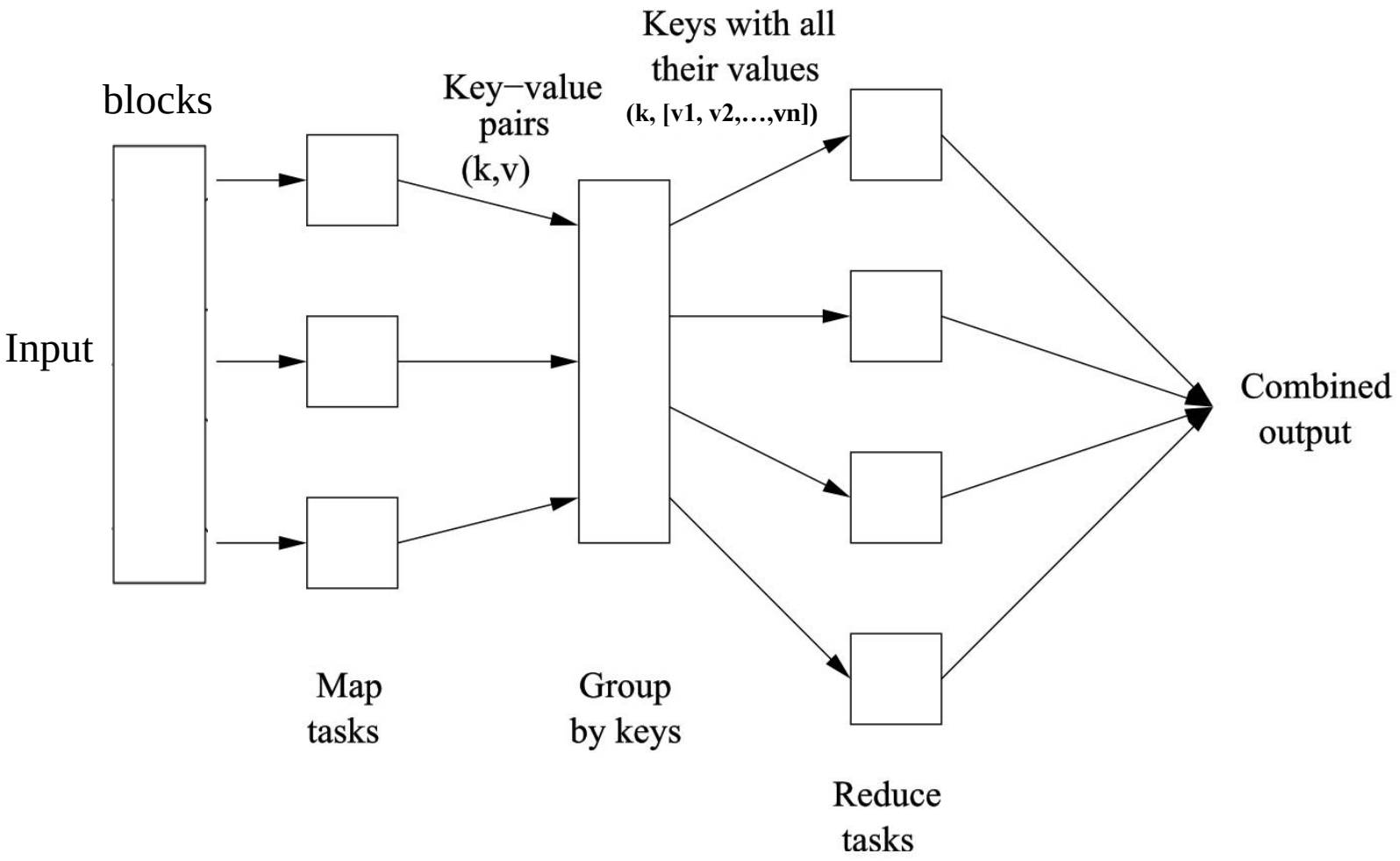
- R package *rhdfs*, *rmr2*, *plyrmr* of RHadoop provide a set of R functions that allow us to directly manipulate and analyze all kinds of data files in HDFS.

- Another exciting part of RHadoop is that you can write functions to read and write *any R objects* from or to the local file system and HDFS!

```
# Write R object to HDFS
to.dfs(mtcars, "/home/myfolder/mtcars.RData")
# Read R object from HDFS (native R object)
from.dfs("/home/myfolder/mtcars.RData",format = "native")
```

- You may notice that *from.dfs*() have transformed *mtcars* data into a key-value R object with a NULL as the key. Don't worry about this now. As we discussed previously, we'll be working with key-value pairs of objects throughout this class. In RHadoop, such objects are used as the return values for the Map and Reduce tasks.

MapReduce – a review



MapReduce – a review(cont.)

- MapReduce is similar to the split-apply-combine process we discussed previously, but it could also parallelize tasks on a cluster of machines. Programmers simply specify *Input*, *Map*, *Reduce,* and *Output* steps, then Hadoop will handle the rest.

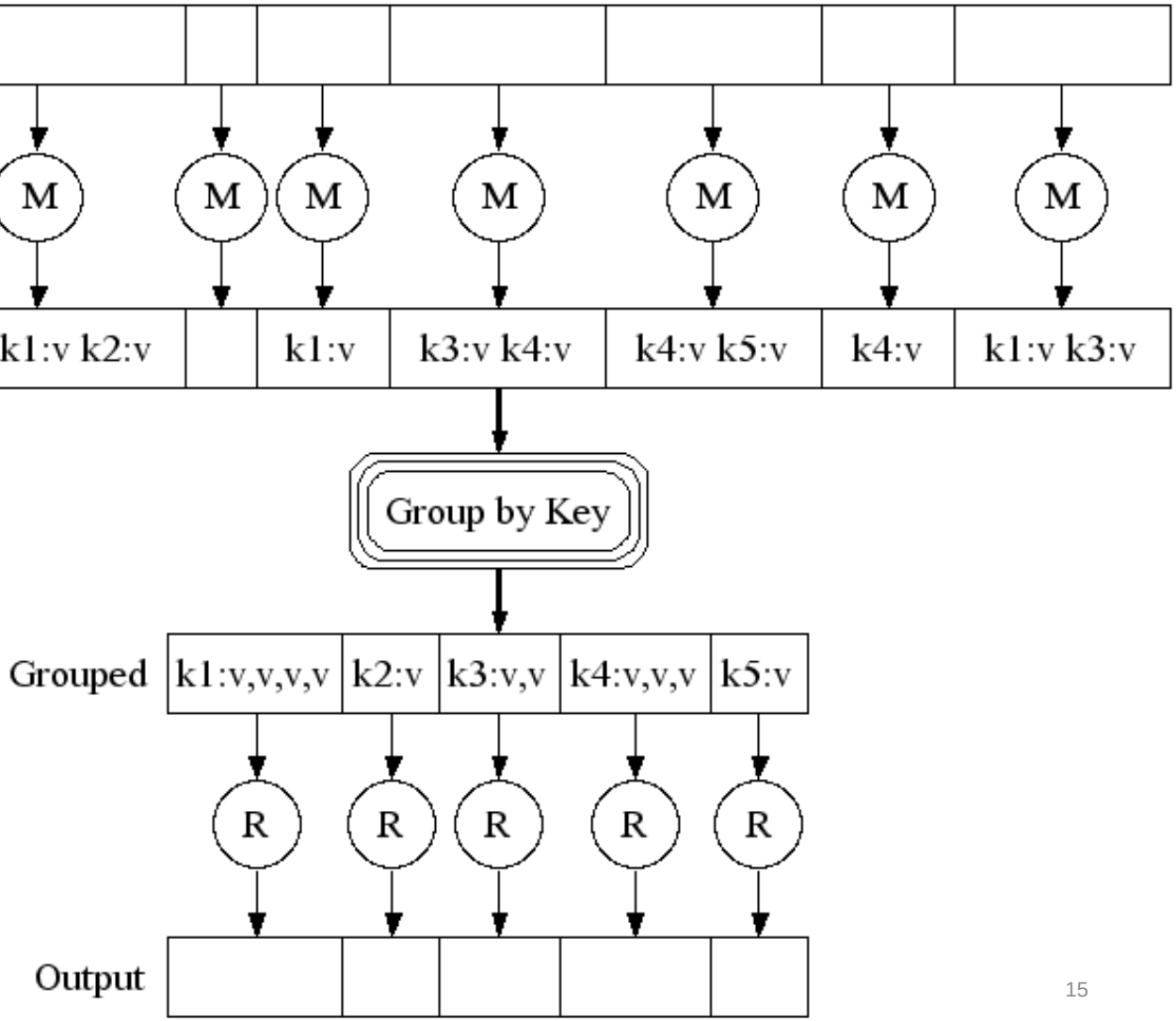
- The *input/output* of a MapReduce task could be a file, an R object, or a set of key-value pairs of R objects. The *Map(k, v)* takes a key-value pair data from the *input*, split the data by the key, and returns a new set of key-value pairs (k, v). There is one Map call for every (k,v) pair. On the other hand, the *Reduce(k, [*v_1, v_2 , … , v_n*])* takes values *v* with the same key *k* generated by previous *Map* step, reduces *[*v_1, v_2 , … , v_n*]* together, processes it in order, then returns a new aggregated result key-value (*k, V*). There is one Reduce call per unique key *k*.

MapReduce – a diagram

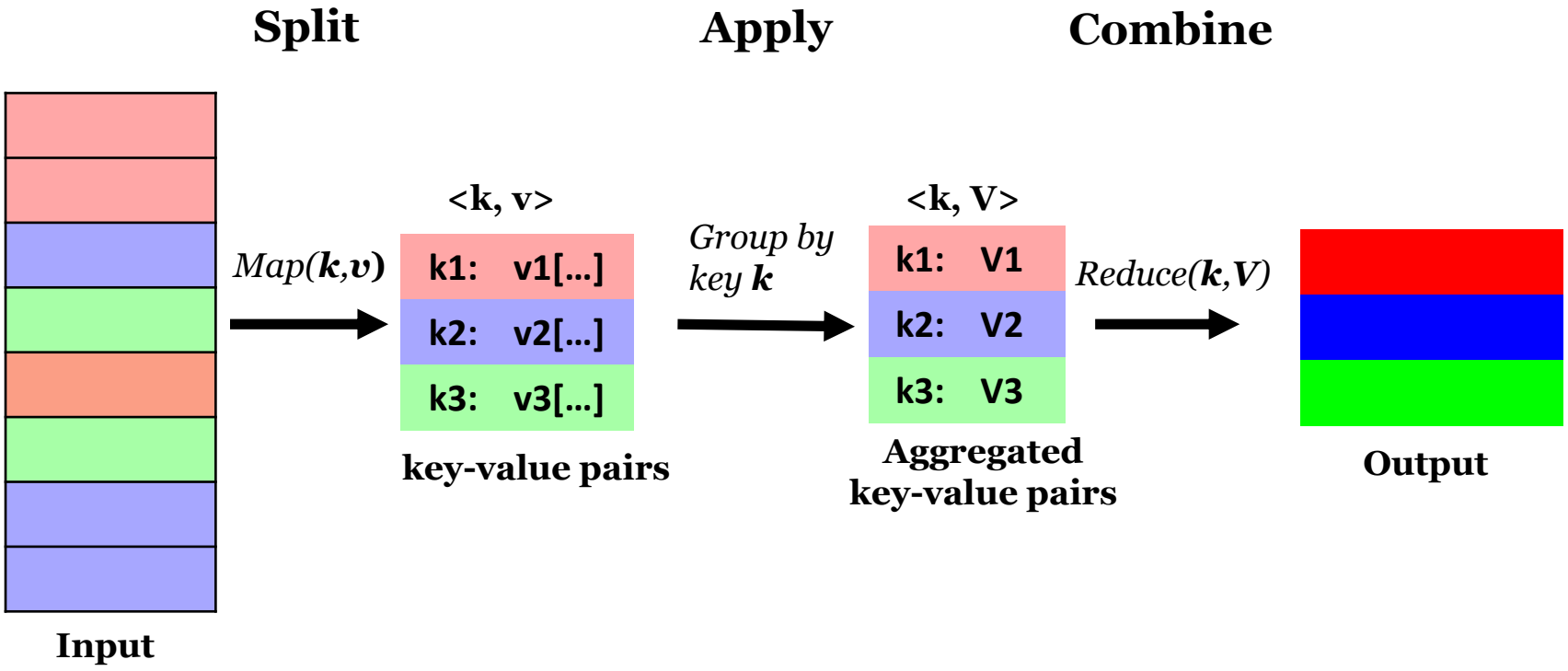
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

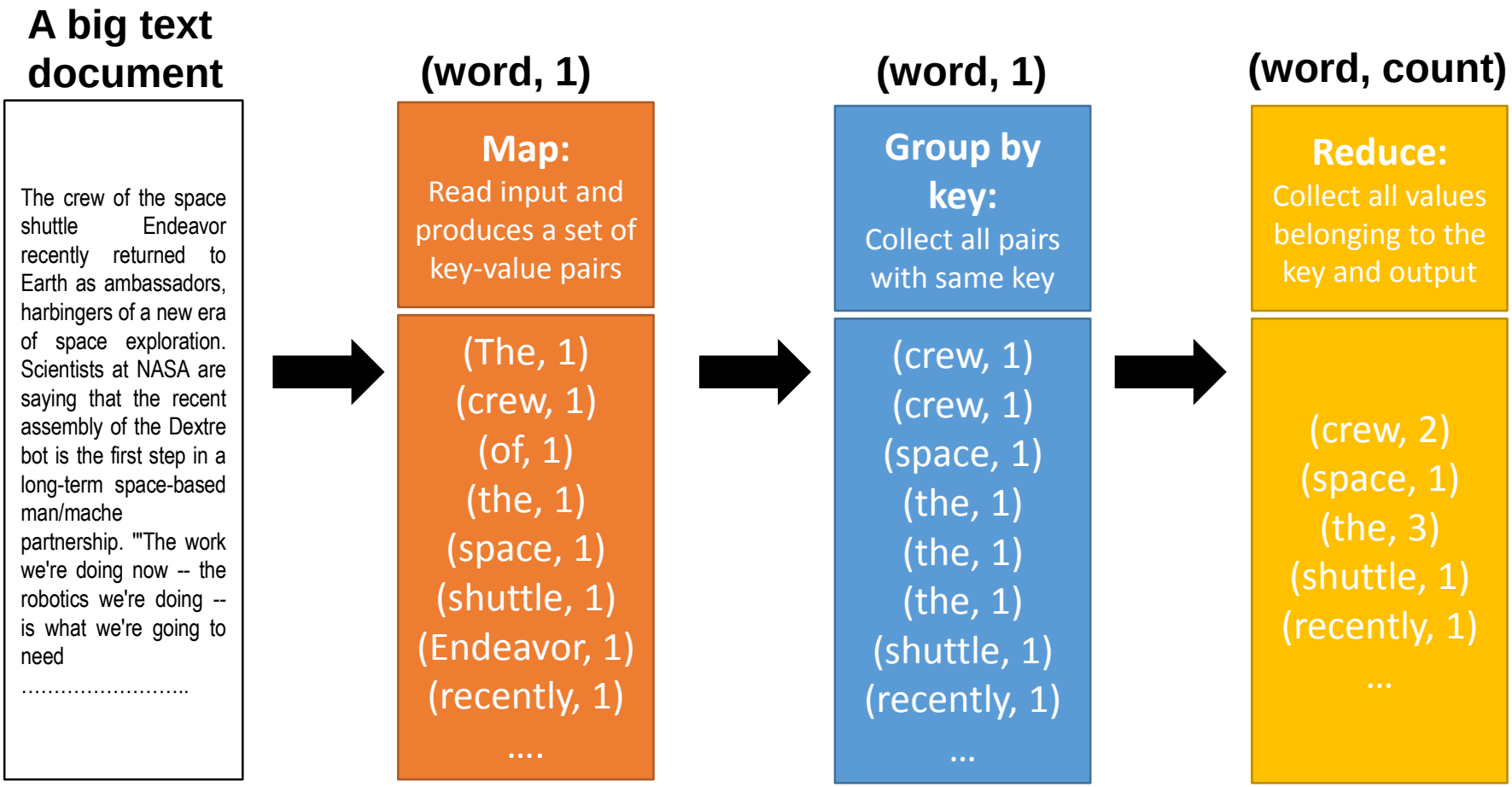
Reduce:
Collect all values belonging to the key and output



MapReduce & Split-Apply-Combine



Word counting example



Word counting example(cont.)

```
map(key, value):
// key: document name
// value: text or line of the document
 for each word w in value:
     return(w, 1)

reduce(key, values):
// key: a word
// values: vector with a set of "1"s for the same key
  return(key, sum(values))
```

Word counting in R

```
# We can easily implement the wordCount() in a few lines of R codes
line = "I am a student. You are a student.
        He is a student. We are all students!"
# Save the "line" object as a plain text file to the HDFS
to.dfs(line, output='/home/myfolder/small_doc.txt', format="text")

# Wordcount(). Remove (split by) punctuations, spaces, and digits.
wordcount = function(input, output = NULL,
                     pattern = '[[:punct:][:space:][:digit:]]+'){
  mapreduce(input = input, output = output, input.format = "text",
            map = function(k, lines)
                 keyval(unlist( strsplit(lines,split = pattern)),1),
            reduce = function(word, counts)
                 keyval(word, sum(counts)))
}

wordcount("/home/myfolder/small_doc.txt",
          output = "/home/myfolder/small_doc_wc.RData")

from.dfs("/home/myfolder/small_doc_wc.RData")
```

Try it!

- Please write a function *findLongestWords*() in MapReduce manner to find the longest word(s) in a given big plain text file (hint: you may borrow some idea from the *wordcount*()).

findLongestWords()

```
line = "I am a student.
You are a student.
He is a student. We are all students!"
to.dfs(line, output = '/home/myfolder/small_doc.txt',format="text")

findLongestWord = function(input, output = NULL,
                      pattern = '[[:punct:][:space:][:digit:]]+'){
  mapreduce(input = input, output = output, input.format = "text",
            map = function(k, lines) {
              words = unlist( strsplit(lines,split = pattern));
              maxLenWord = words[nchar(words) == max(nchar(words))];
              # the longest word(s) in one line of the text
              return(keyval(1, maxLenWord )  );
            },
            reduce = function(k, v)
               keyval(k, Reduce(function(w1,w2){
                   ifelse( (nchar(w1) > nchar(w2)),w1 ,w2 )}, v ) )
  )
}
findLongestWord(input  = "/home/myfolder/small_doc.txt",
                output = "/home/myfolder/small_doc_lw.RData")
from.dfs("/home/myfolder/small_doc_lw.RData") # check the result
```

MapReduce on R data frame

- The *input* of the MapReduce process can also be an R data frame. Just upload your R data frame to HDFS using *to.dfs*().

```
# Save mtcars as native R object to the HDFS
to.dfs(mtcars,"/home/myfolder/mtcars.RData", format = "native")

# We can also check the content of the R data frame
from.dfs("/home/myfolder/mtcars.RData",format = "native")
$key
NULL

$val
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
...
```

- The *mtcars* in the HDFS is now a key-value R list object with "NULL" as the key and the whole data frame as the value.

MapReduce – Filtering

- Let's do some daily data management tasks on big datasets. For simplicity, here we assume that the *mtcars* data is a big R data frame.

```
# Similar to SQL "where" statements, here we filter out some
# rows/observations. Keep only those with "am == 1"
mapreduce(input = "/home/myfolder/mtcars.RData",
          output = "/home/myfolder/mtcars_am_1.RData",
          input.format = "native",
          map = function(k, v) keyval(key = NULL,
                                      val = v[v$am == 1,]) )
from.dfs("/home/myfolder/mtcars_am_1.RData") # Check the result.
# Similar to SQL "select", here we select some columns we need.
mapreduce(input = "/home/myfolder/mtcars.RData",
          output = "/home/myfolder/mtcars_wt_mpg.RData",
          input.format = "native",
          map = function(k, v) keyval(key = NULL,
                                      val = v[,c("wt","mpg")]) )
```

- Note that we set "key = NULL" to get the whole data instead of splitting/mapping it into pieces, and thus we only need the *Map*() step!

MapReduce – Filtering(cont.)

- The *Reduce*() makes aggregation tasks easier. Consider the following example.

```
# Group the following data by first two characters of "x1".
# How many rows with "x1" beginning with "aa", "bb", or "cc"?
twochr = data.frame(x1=c('aa11','aa35','bb23','bb34','cc23','bb33'),
                    x2=c(1,1,2,3,4,4))
to.dfs(twochr, "/home/myfolder/twochr.RData")

# Using substr() to get first 2 characters of the "x1" as the key
from.dfs( mapreduce(input = "/home/myfolder/twochr.RData",
        map = function(k, v) keyval(key = substr(v$x1,
                                    start = 1, stop = 2), val = v ) ,
        reduce=function(k, V) keyval(k, val = nrow(V))
))
...
$key
[1] "aa" "bb" "cc"

$val
[1] 2 3 1
```

MapReduce – Distinct/Unique Values

- In some cases, we need *Reduce*() to aggregate and reduce the output. Here is an example to find unique *gear* values in *mtcars*.

```
# Find distinct gear values from mtcars
mapreduce(input = "/home/myfolder/mtcars.RData",
          output = "/home/myfolder/mtcars_distinct_gear.RData",
          map = function(k, v) keyval(key = v$gear, 1))
$key
 [1] 4 4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 5 5 5 5 5
$val
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

# Delete existing file
hdfs.del("/home/myfolder/mtcars_distinct_gear.RData")
mapreduce(input = "/home/myfolder/mtcars.RData",
          output = "/home/myfolder/mtcars_distinct_gear.RData",
          map = function(k, v) keyval(key = v$gear, 1),
          reduce = function(k, v) keyval(key = k, 1))
$key
[1] 3 4 5
$val
[1] 1 1 1
```

MapReduce – Sorting

- Another common data manipulation is to sort a big dataset. We can do it in both *Map*() and *Reduce*(). Notice that, here, we must have a rough idea about how many Map tasks will be generated. Too many Maps requires more memory in namenode, whereas too few Maps imposes high computation on a few datanodes.

```
# Sort by gear and mpg
from.dfs(mapreduce(input = "/home/myfolder/mtcars.RData",
          input.format = "native",
          map = function(k, v) keyval( v$gear, val = v),
          reduce = function(k, v) keyval(k, v[order(v$mpg),])))
# Using composite key also works, too.
from.dfs(mapreduce(input = "/home/myfolder/mtcars.RData",
     map = function(k, v) keyval( v[, c("gear", "mpg")], val = v),
     reduce = function(k, v) keyval(k, v)))
```

- We may also use composite keys or more then one single MapReduce process, a chain a multiple MapReduces, to cope with complicated sorting tasks. Check out *Terasort Algorithm* for more efficient sorting techniques.

MapReduce – Remove Duplicates

- Consider a more complicated examples with the *Reduce*() step. Here we'd like to remove duplicate records in a big R data frame.

```
# Create & upload a data frame with duplicates
dup = data.frame(x1=c('a','a','b','b','c','c'),
                 x2=c(1,1,2,3,4,4) )
to.dfs(dup, "/home/myfolder/dup.RData")
# Remove duplicates
from.dfs( mapreduce( input = "/home/myfolder/dup.RData",
                     map = function(k, v)
                        keyval(key = v, val = 1 ),
                     reduce = function(k, V)
                        keyval(key = k, val = sum(V))))
```

- Notice that, in *Map*(), we use the entire record as the key—a long composite key that may generate huge amount of Map tasks (i.e. up to *n* Map tasks for *n* records) ! Later in *Reduce*(), the number of identical records are returned as the aggregated values.

MapReduce – Join

- In the case that we'd like to merge two tables/datasets horizontally like what we did in single machine, we can simply use *rmr2::equijoin*(), instead of writing multiple complicated MapReduce functions.

```
# create 2 temporary data frames, A & B
A = data.frame(id=c(1,3,5), val=c('a','x','c'));
B = data.frame(id=c(3,5,6), val=c('x','y','z'));
to.dfs(kv = A, output = "/home/myfolder/A.RData");
to.dfs(kv = B, output = "/home/myfolder/B.RData");
# Inner join
from.dfs( equijoin(left.input = "/home/myfolder/A.RData",
          map.left = function(k,v) keyval(v$id, v),
          right.input = "/home/myfolder/B.RData",
          map.right = function(k,v) keyval(v$id, v),
          outer = NULL))
# Specify outer = "left", "right", or "full" for other outer joins
```

- You may notice that *equijoin()* have done all the implementations of MapReduce tasks for you. The inner join only need 1 MapReduce process, whereas left/right/outer joins require 3 MapReduce processes.

MapReduce – Concatenation

- Another data merging tasks is to concatenate multiple datasets vertically. It is relative easy, as we're not doing any further data aggregations.

```
# We here simply specify multiple inputs
from.dfs(mapreduce(
   input = c("/home/myfolder/A.RData","/home/myfolder/B.RData"),
   map = function(k, v) keyval(NULL, v)))

# How about some aggregation tasks? Say how many val = 'x'?
from.dfs(mapreduce(
   input = c("/home/myfolder/A.RData","/home/myfolder/B.RData"),
   map = function(k, v) keyval(v$id, v),
   reduce = function(k, v) keyval(k, sum(v$val == 'x'))
))
```

- Note that the variable names and their orders must be the same. Otherwise, we may get errors and columns in your result dataset would be renamed.

MapReduce – Concatenation(cont.)

- A more practical problem you may encounter is *small file problem* in HDFS, i.e. you have many files significantly smaller than HDFS block size (64 or 128 MB for Hadoop 2.0+), and they requires large amount of memory to process. A rule-of-thumb to help estimate memory use of your Hadoop namenode is that, for each block (even a very small file requires one block), namenode need at least 150 bytes to keep tracking on a block in the HDFS.

- What is even worse is that, *Map*() typically processes a block at a time, which means it takes much more time on processing 1,000 64KB files (1,000 blocks) than processing one single 64MB file (1 block)!

- One common solution is to replace these small files with one or a few *Hadoop Archive* (HAR) files.

MapReduce – Concatenation(cont.)

- Similar to TAR file on most Unix-like systems, an HAR packs many small files into a large file with metadata/index files that describe its contents. In HDFS, an HAR works like a file but looks like a "folder" with many binary part-* files.
- Consider a simple application that we'd like to concatenate multiple CSV files into an HAR-like file in HDFS.

```
# Let's say we have many CSV file with the same column names,
# "a","b","c", "d", "e" in /home/myfolder/many_csv
mapreduce( input = hdfs.ls("/home/myfolder/many_csv")$file,
   input.format = make.input.format("csv", mode = "text",
                  sep = ",", col.names = letters[1:5],
                  skip = 1, stringsAsFactors = F),
   output = "/home/myfolder/output.har",
   map = function(k, v) keyval(NULL, v)
)
```

- Also check out here for more customized *rmr2* input/output data formats.

MapReduce – Matrix Transpose

- With R's powerful data management functions, MapReduce can do much more than just simple data filtering and merging! Here is an example of matrix transpose.

```
# Let's say we have a matrix in plain text CSV
from.dfs("/home/myfolder/matrix.csv", format="csv")
...
     V1
1 1,2,3
2 4,5,6
from.dfs(mapreduce(
  input = "/home/myfolder/matrix.csv",
  input.format = make.input.format(format = "csv",
                  mode = "text",  sep = ","),
# Emit index of each cell (of each row) as the key
  map = function(k, v) keyval( 1:length(v), unclass(v)),
  reduce = function(k, V) keyval(NULL,rbind(unlist(V)))
))
...
     V11 V12
[1,]   1   4
[2,]   2   5
[3,]   3   6
```

MapReduce – Crosstab

- Checking out frequency tables or crosstabs might be your first step to analyze categorical data. Let's do it in MapReduce manner to distribute the computations to a cluster of computers.

```
# "mtcars" again for example
to.dfs(mtcars, "/home/myfolder/mtcars.RData", format="native")
# Function to get crosstab given two categorical variables, x & y
crosstab_MR = function(dfs_data, x, y, ylevels){
  mapreduce( input = dfs_data,
    map = function(k, v){
      # Output values of "x" as the keys
      return(keyval(key = v[,x], val = v[, y] ));
    },
    reduce = function(k, v){
      tab = rbind(table(factor(v,levels=ylevels) ));
      rownames(tab) = k;  return(keyval(key=k, val=tab));
    }
)}
from.dfs(crosstab_MR("/home/myfolder/mtcars.RData",
                     x = 'am', y = 'gear', ylevels = c(3,4,5)));
```

Try it!

- As we discussed previously, the *five numbers* are simple summary statistics often used to check distributions of continuous variables. Take the *mtcars* dataset as the example again, the following R statement computes five numbers for all variables.

```
lapply(mtcars, FUN=fivenum)
$mpg
[1] 10.40 15.35 19.20 22.80 33.90

$cyl
[1] 4 4 6 8 8
...
```

Please write a MapReduce function *fivenum_MR*() that calculates the five numbers of a given set of variables in a big dataset (the input of your function could be a plain CSV or R native dataset).

Chain of MapReduce

- Combinations of multiple MapReduce steps are practically used to solve complex tasks. Such *chains of MapReduce* make your programs even more powerful! Let's get back to previous sorting problem again, for example. Here we combine two Maps and one Reduce steps.

```
# Sort "mtcars" by given two variables
chainMRSort = function(input, output, byVar1, byVar2){
  mapreduce( input = mapreduce( input = input,
                        map = function(k, v) keyval(v[, byVar1], v)),
# Output composite key (col1, col2). Then sort by the key.
# Note that the composite key is therefore a row of a "data frame".
             map = function(k, v)
                keyval( data.frame(col1=k, col2=v[, byVar2]), v),
             reduce = function(k,v)
                keyval(k,v), output = output)
}
chainMRSort("/home/myfolder/mtcars.RData",
            "/home/myfolder/chainSort1.RData", "am", "mpg")
from.dfs("/home/myfolder/chainSort1.RData")
```

How many Map and Reduce jobs?

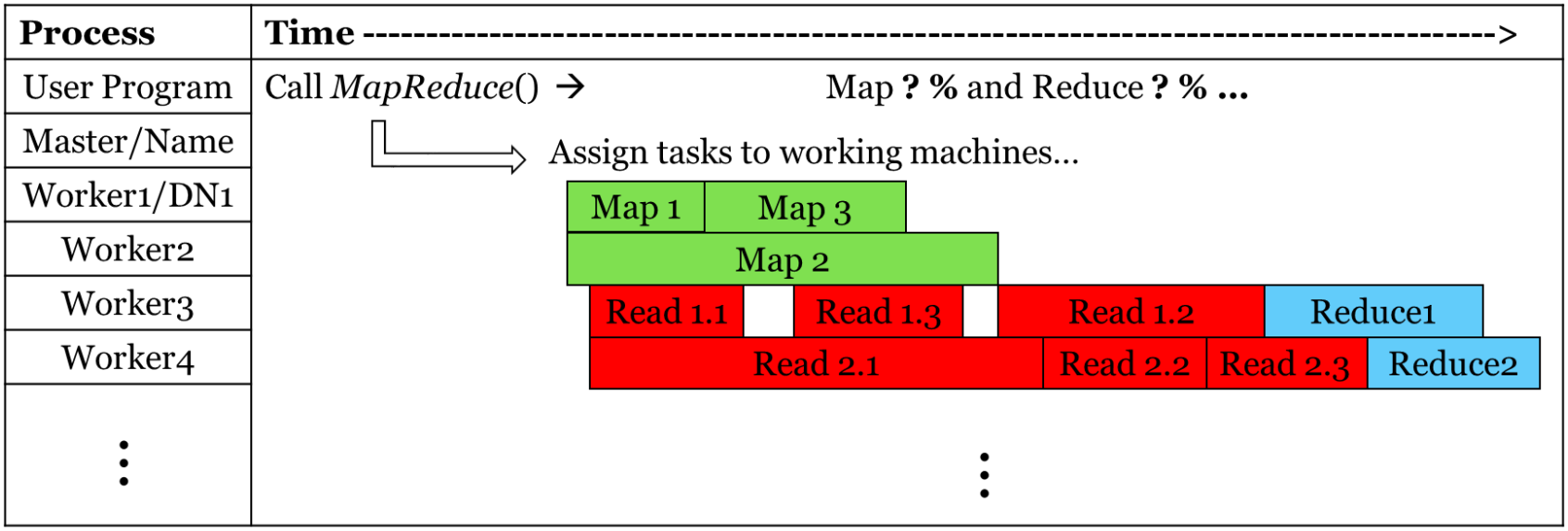
- We have been trying to distributing/parallelizing computations of our data processing tasks in the MapReduce models. The first and crucial step is to find better "keys" that help spread tasks to multiple computing nodes—the key to determine the numbers of Map and Reduce jobs. So, *how to choose the best key*?

- The answer to it is "*it depends*". The best MapReduce coding example in textbooks may not work as great as it in your situations. For example, matrix operations require significant amount of memory. Some MapReduce examples suggest striping vectors and splitting tasks into large numbers of Maps (so that each Map can fit in memory of each computing node). However, too many Maps results in high latency before your MapReduce task actually begins, as Namenodes need too much memory and spend too much time on locating blocks in the cluster.

- So, as a rule of thumb, let's say there're m Maps and r Reduce jobs, making m much larger than the number of nodes in the cluster is a good start (and the r should be smaller than m). Depending on CPU cores & main memory in each node, your MapReduce task analysis should balance between resource consumption and latency in each computing steps.

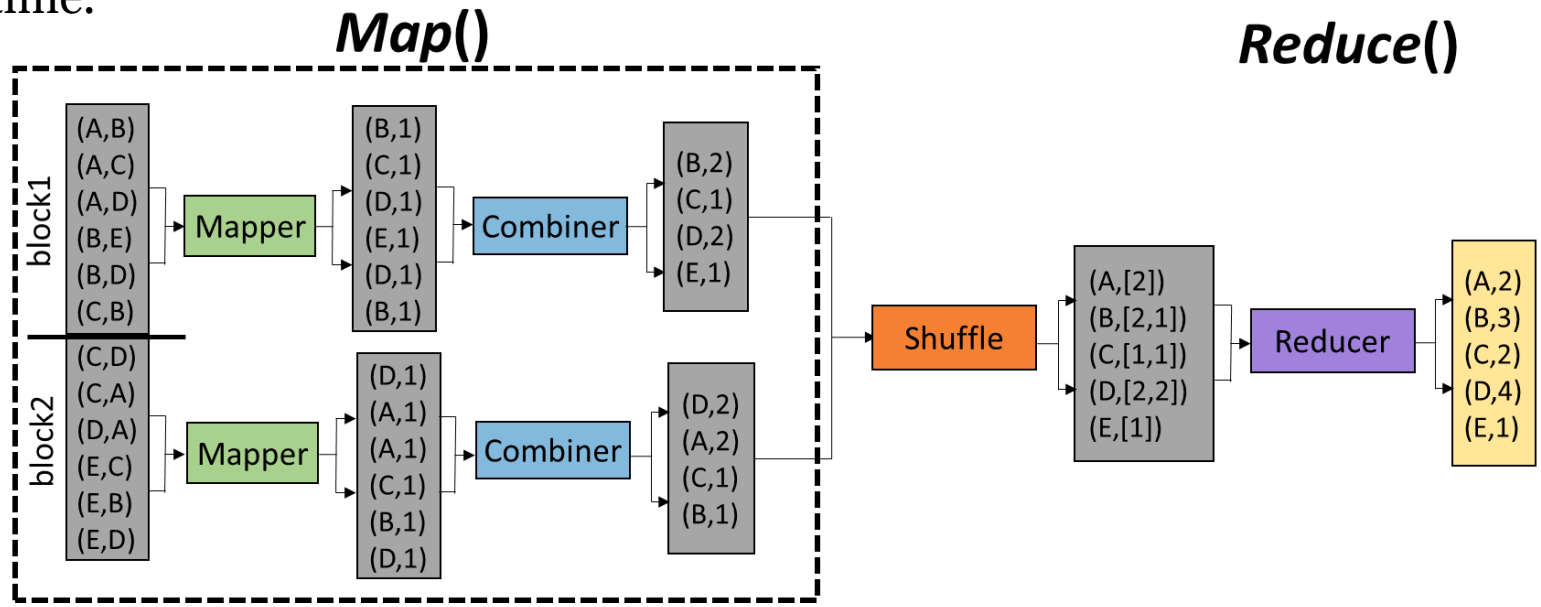
MapReduce Analysis—Finer Granularity of Maps

- As suggested previously (and also in many textbooks), in most cases, the finer the granularity of Maps, the better the load balancing. Another advantage is that it can minimize time for fault recovery (errors in MapReduce can be detected quickly because of less turnaround time for smaller Map), and can do pipeline shuffling with Map execution (i.e. nodes can do Map and Reduce tasks simultaneously without waiting for the finishes of other Maps).



MapReduce Analysis—Combiners in Maps

- Very often, a Map task will produce many pairs of the form (k, v_1), (k, v_2), … for the same key k. Combining/Pre-aggregating values (v) in Maps before emit/output the key-value pairs may save significant network transmission time.



- Many programming languages implement the combiner by using *reduce*() in Reduce step. But it only works when your *reduce*() is commutative and associative.

MapReduce Analysis—Combiners in Maps(cont.)

- In R, we can easily create such combiner by specifying parameter *combine* as TRUE in *rmr2::mapreduce*().

```
# wordcount() that uses reduce() as the combiner.
wordcount_w_combiner = function(input, output = NULL,
                        pattern = '[[:punct:][:space:][:digit:]]+'){
  mapreduce(input = input, output = output, input.format = "text",
            map = function(k, lines)
                 keyval(unlist( strsplit(lines,split = pattern)),1),
            reduce = function(word, counts)
                 keyval(word, sum(counts)), combine = T
}
```

- Note that you can also *create your own combiners* in Map step instead of using such "default combiner". It is one of the reason that many people consider MapReduce model simple, flexible, and powerful!

MapReduce Analysis—Small data, big computation

- A special but practical application of the MapReduce is to solve "small data, big computation" problems. In such cases, we have small or almost no data to process, but we would simply like the cluster of computers to solve computation-intensive tasks (e.g. simulations). Using H2O or Apache Spark may be a better solution to this problem today, but we can still do it in Hadoop MapReduce.

- Before diving into the MapReduce code, let's review a bit about functional programming of R. We know everything, including a function, is an *object* in R, so we can surely create and manipulate functions just like what we're doing to a string or an integer value .

```
# A vector of functions (it's actually an "R list")
someCalculationList = c(function(x) x^2, function(x) x^3,
  mean, max)
lapply(someCalculationList, FUN = function(f) f(1:10))
```

MapReduce Analysis—Small data, big computation(cont.)

- You may notice (and might also be surprised) that *rmr2::to.dfs*(), without specifying the parameter *output*, returns a "function" that actually returns the temporary file location of your R objects.

```
two_numbers = to.dfs(kv = keyval(key = 1:2,val = c("num1", "num2")))
two_numbers # your output should be different from below.
function ()
{
    fname
}
<bytecode: 0x36e69e0>
<environment: 0x9f19020>
two_numbers()
[1] "/tmp/file43f311d228be"
```

- Hadoop still put such files on the HDFS (i.e. as different blocks on different data nodes), but theses temporary files may be removed by Hadoop later. Interestingly, we can virtually parallelize any kinds of computations if our MapReduce function takes multiple such files (with different key-value pairs) as the input, simply because these files are saved in multiple & different data nodes!

MapReduce Analysis—Small data, big computation(cont.)

- Let's consider a simple example that we'd like to demonstrate the *Central Limit Theory* discussed previously. Now, we'd like to distribute the computations of the "x bars" over a cluster of computers.

```
# "workers" is a list of functions to save the return values of to.dfs()
workers = list(); numOfWorkers = 20; x_bars_per_worker = 50
# to.dfs() returns a function that contains the temporary location of file in
# HDFS. Here we consider putting 20 temporary small files on HDFS as 20
# workers. We'd like each worker to compute 50 x_bars.
# These small files are just key-value pairs: 1-> 50, 2-> 50, ..., 20 -> 50

for(i in 1:numOfWorkers)
   workers[[i]] = to.dfs(keyval(i, x_bars_per_worker))

x_bar_kv = from.dfs( mapreduce(input= hdfs.ls("/home/myfolder/workers")$file,
     map = function(k, v){
     set.seed(k)# different seed to avoid generating identical random numbers
     # An x_bar is the mean of 100 uniformly-generated numbers (between 0 and 1)
     matrix_worker_by_xbars = matrix(runif( v[1] * 100,0,1), ncol = v[1] )
     x_bars = colMeans(matrix_worker_by_xbars)
     return(keyval(1, list(x_bars) ))
     },reduce = function(k, v)(keyval(k,  v))
))
ggplot2::qplot(x = unlist(x_bar_kv$val),geom = "histogram" )
```

MapReduce—A real-world application

- The previous "worker designation", which is virtually to *assign an R task/session to a virtual CPU core*, demonstrates the power of the MapReduce model with R. It allows us to easily distribute any R data analytics tasks to a cluster of machines (or even single machine with multiple virtual CPU cores!).

- Here is another practical example that we'd like to use the concept of *worker assignment* to parallelize a task that builds a random forest model.

```
# To distribute the task of building a random forest model
# that predict "price" in diamond data
library(ggplot2); library(randomForest)
diamonds_df = as.data.frame(diamonds)
# We here convert the dataset into compressed key-value pair data
pryr::object_size(diamonds_df) # ~3.46 MB
# Serialize and compress the dataset
compressed_diamonds_df =
   list(memCompress(serialize(diamonds_df, NULL),type = "gzip" ))
pryr::object_size(compressed_diamonds_df) # ~ 509 KB
```

MapReduce—A real-world application(cont.)

```
workers = character(10) # Number of 10 logical "workers"
for(i in 1:10){
  # Locations of 10 identical datasets.
  workers[i] = paste("/home/myfolder/compress_diamonds_", i, ".RData", sep = "")
  # The key is from 1 to 10. Used as the random seed value
  kv = keyval(i, compressed_diamonds_df ) to.dfs(kv, workers[i])
}
# 10 workers * 10 trees/worker = 100 regression trees in the forest
rf = from.dfs(mapreduce(input = workers,
       map = function(k, v){# decompress the data
           dat = unserialize(memDecompress(v[[1]], type = "gzip"))
          set.seed(k) # set seed to make random forest "random"
          # Compress and return the random forest model object
          return( keyval(k, list(memCompress(serialize(
              randomForest::randomForest(price ~ ., data = dat,
              ntree = 10, na.action = na.omit), NULL),type = "gzip" ))) )
       }, reduce = function(k, v) keyval(k,v)))
# "rf" is now a key (1-10) to value(10 random forests. 10 trees for each)
rfs = list(); rf_val = values(rf)
for(i in 1:10) {rfs[i] = list(unserialize(memDecompress(rf_val[[i]],
    type = "gzip")))}
# Combine 10 forests. The final random forest model has 100 trees
mergedRF = Reduce(f = function(rf1, rf2) randomForest::combine(rf1, rf2), rfs)
```

Package Plyrmr

- For those of you who would like to do some quick-and-dirty data analysis tasks on structured big datasets without bothering with the MapReduce, package *plyrmr* may suit your need.

- The *plyrmr* borrows ideas from many famous R package (e.g. *plyr*, *dplyr*, *reshapes*, etc..), SQL, and Apache Spark. It provides a set of functions that allows you to easily manipulate big datasets in HDFS using similar syntaxes and functions.

- The logic of *plyrmr* is built on top of the MapReduce, but complicated MapReduce steps are hidden from end users. Still, we specify input data source and output data destination. But intermediate data processing logics are replaced by Unix-like pipeline operations and "apply"-like aggregation & vectorized functions.

Plyrmr—%|%(pipe) operator

- The **%|%** is similar to pipe operator used in pipelines of most Unix-like system shell commands. It connects the input, output, and data manipulations to form a chain of data processes in HDFS.
- Let's begin with a simple example that we'd like to know the numbers of rows and columns (i.e. dimension) of an R dataset in HDFS. We would normally write a nested functions as below, which take the result of *plyrmr::input()* as the input data source.

```
# Again, let's say we have the "mtcars" in HDFS
dim(input("/home/myfolder/mtcars.RData", format="native"))
  nrow ncol
1   32   11
```

- We can replace the above statement with the %|%.

```
input("/home/myfolder/mtcars.RData", format="native") %|% dim
# Also remember to check out plyrmr::nrow() and plyrmr::ncol()
```

Plyrmr—%|%(pipe) operator(cont.)

- Consider another example that we'd like those records with *"am ==1"*. The *where*() with multiple %|% can do the trick.

```
input("/home/myfolder/mtcars.RData", format="native") %|%
  where(am == 1) %|%
  output(path = "/home/myfolder/mtcars_am1.RData")
hdfs.ls("/home/myfolder/mtcars_am1.RData")
```

- You may notice that the output file is actually a "folder" in HDFS. It's simply because the *output*() helped you do Map() & Reduce() tasks that created an output directory with metadata that indicate whether the task was done successfully.

```
# Let's check out the output data.
from.dfs("/home/myfolder/mtcars_am1.RData")
...
$val
               mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
```

Plyrmr—*select*()/*where*()/*group*()

- Just like SQL syntax, the *select*(), *where*(), and *group*() can literally do the data filtering and aggregation for you. You can use them as regular R functions with input data sources or combine them with %|% to make your code like pipeline processes.

```
# Built-in iris data
to.dfs(iris, "/home/myfolder/iris.RData")
# Equivalent to "select Sepal.Length, Sepal.Width, Species
# from iris where Species == "setosa")
where(select(input("/home/myfolder/iris.RData"), Sepal.Length,
      Sepal.Width, Species ), Species == "setosa")

# Equivalent to "select am, count(am) from mtcars group by am"
# Note that the nrow() is NOT base::nrow()
input("/home/myfolder/mtcars.RData") %|%
  group(.columns=c("am")) %|% nrow %|%
  output(path="/home/myfolder/mtcars_nrow_by_am.RData")
```

Plyrmr—*bind.cols*() & *rbind*()

- You may wonder if there are functions that help you concatenate big datasets vertically/horizontally without writing the daunting Map() & Reduce() functions. Yes, just like *cbind*() and *rbind*() of package *base*, *plyrmr*::*binds.cols*() merges data vertically by adding columns to existing data, whereas *plyrmr*::*rbind*() combines dataset by concatenating rows.

```
# A new column, kpl, for "km per liter". Keep those with kpl > 10
bind.cols(input("/home/myfolder/mtcars.RData"),
                kpl = mpg * 0.42514) %|% where(.cond = kpl > 10) %|%
  output(path="/home/myfolder/mtcars_kpl_gt10.RData")
from.dfs("/home/myfolder/mtcars_kpl_gt10.RData")

# Put two new car datasets to the HDFS
to.dfs(data.frame(carName = rownames(mtcars), mpg = mtcars[,c("mpg")]),
   "/home/myfolder/car_mpg1.RData")
to.dfs(data.frame(carName = "VW Golf", mpg = 22 ),
   "/home/myfolder/car_mpg2.RData")
# Concatenate two datasets
plyrmr::rbind(input("/home/myfolder/car_mpg2.RData"),
              input("/home/myfolder/car_mpg1.RData")) %|%
  output(path="/home/myfolder/car_mpg.RData")
```

Plyrmr—*transmute()* your data!

- The *plyrmr*::*transmute*(), which borrows ideas from *base*::*transform*(), *dplyr*::*mutate*(), and *dplyr*::*transmute*(), allows you directly manipulate big data by creating new variable/column as the result of almost all R functions applied to your data.

```
# Five numbers of mpg & hp
transmute(input("/home/myfolder/mtcars.RData"),
   fivenum(mpg), fivenum(hp) )

# Equivalent to SQL code: "select * from mtcars where
# mpg >= (select avg(mpg) from mtcars)"
avg_mpg =
as.data.frame(transmute(input("/home/myfolder/mtcars.RData"),
   mean(mpg)))[[1]] # Get average mpg as a numeric value

input("/home/myfolder/mtcars.RData") %|%
   where(.cond = mpg >= avg_mpg)
```

- Note that you might get error messages if the result of your function is not an R vector. You may consider using *list*() or *data.frame*() to encapsulate the result.

Plyrmr—*transmute()* your data!(cont.)

```
# Frequency table
result =
as.data.frame(transmute(input("/home/myfolder/iris.RData"),
   list(table(Species))))
result[[1]]
[[1]]
Species
    setosa versicolor  virginica
        50         50         50
# Normality tests
result =
as.data.frame(transmute(input("/home/myfolder/mtcars.RData"),
   shapiro = list(shapiro.test(mpg)),
   lillie = list(nortest::lillie.test(mpg) ) ))
result$shapiro
[[1]]
       Shapiro-Wilk normality test
data:  mpg
W = 0.94756, p-value = 0.1229
# Linear Model
result =
as.data.frame(transmute(input("/home/myfolder/mtcars.RData"),
        list( lm(formula = mpg ~ hp + am + hp:am))))
result[[1]]
```

Plyrmr—*merge*(), *intersect*(), and *union*()

- In additionally *rmr2::equijoin*(), *plyrmr*::*merge*(), offers the same functionalities to join two big datasets with almost identical syntax.

```
# "A" and "B" datasets again!
A = data.frame(id=c(1,3,5), val=c('a','x','c'));
B = data.frame(id=c(3,5,6), val=c('x','y','z'));
to.dfs(A, "/home/myfolder/A.RData")
to.dfs(B, "/home/myfolder/B.RData")
# Left outer join A and B by "id"
merge(input("/home/myfolder/A.RData"),
      input("/home/myfolder/B.RData"), by = "id", all.x = T) %|%
  output(path="/home/myfolder/AB_inner.RData")
```

- The intersect() and union() are literally used to do set operations on big datasets.

```
# Union
plyrmr::union(input("/home/myfolder/A.RData"),
              input("/home/myfolder/B.RData")) %|%
output("/home/myfolder/AB_union.RData")
```

Plyrmr—*reshape2* you data, again!

- You may wonder whether there's any functions to "reshape" big data like what we're doing on local R Data Frames. The *plyrmr* does implement *melt*() and *dcast*().

```
# Built-in CO2 data
CO2_df = data.frame(lapply(CO2,
         FUN = function(v){ if(is.factor(v)){v = as.character(v)}
                            else {v = v} }), stringsAsFactors = F)
# Some R functions working on files in HDFS do not recognize
# R factor. You may consider converting factor vectors into
# character vectors before uploading your data to the HDFS.
to.dfs(CO2_df, "/home/myfolder/CO2.RData")

# Average CO2 uptake for different Treatments and
# the origin of the plants
plyrmr::dcast(input("/home/myfolder/CO2.RData"),
              formula = Type ~ Treatment,
              value.var = "uptake", fun.aggregate = mean )
```

Plyrmr—*gapply*()

- The *plyrmr*::*gapply*() is often used in combination with *plyrmr*::*group()* to solve complex data aggregation tasks. The *gapply*() applies a function to each group of your grouped input data.

```
# Five numbers for different "am"
group(input("/home/myfolder/mtcars.RData"), am) %|%
  gapply(.f = function(x) list(fivenum(x$mpg)) ) %|%
     output("/home/myfolder/mtcars_am_fivenum.RData")

# Linear models for different "am"
gapply(group(input("/home/myfolder/mtcars.RData"), am),
  function(x) list(lm(mpg ~ wt, data = x)))
```

- Note that if your data is ungrouped (without *group*() in the process), your function will be applied to arbitrary chunks of your data.

Final Comment

- MapReduce along with the distributed file systems simplifies large-scale parallel computations with easy-to-implement and hardware-independent programming models. Users can just focus on the problems without worrying about the details of distributed computing architectures.

- R + Hadoop is a powerful marriage. It allows you do virtually any kinds of data analysis & management tasks (it works great on both small and big data problems!). Users can choose between powerful & flexible MapReduce functions in package *rmr2*, or quick & easy pipeline functions in package *plyrmr*.