Introduction to Distributed File Systems & MapReduce Programming Model

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Thank Y. Kang (MIS/NSYSU) & T. Yuan (Math/NSYSU) for providing slides.

Topic Overview

- Distributed File Systems
- Hadoop Distributed File Systems
- MapReduce Programming Model
- MapReduce R examples

Distributed Filesystem

- When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines.

- File systems that manage the storage across a network of machines are called distributed filesystems.

- Google's distributed file systems and programming model, Google File System (GFS, 2003) and the MapReduce, have addressed the problems of distributed computations and processing failure recovery.

- The system takes care of input data, distributes data across the computer network, processes it in parallel, and finally combines its output into a single file to be aggregated later.

- Open source implementations: Hadoop Distributed File System (HDFS, 2006 (Java)) 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.

The Design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

- Very large files
  "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size.

- Streaming data access
  HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern.

- Commodity hardware
  Hadoop doesn't require expensive, highly reliable hardware.

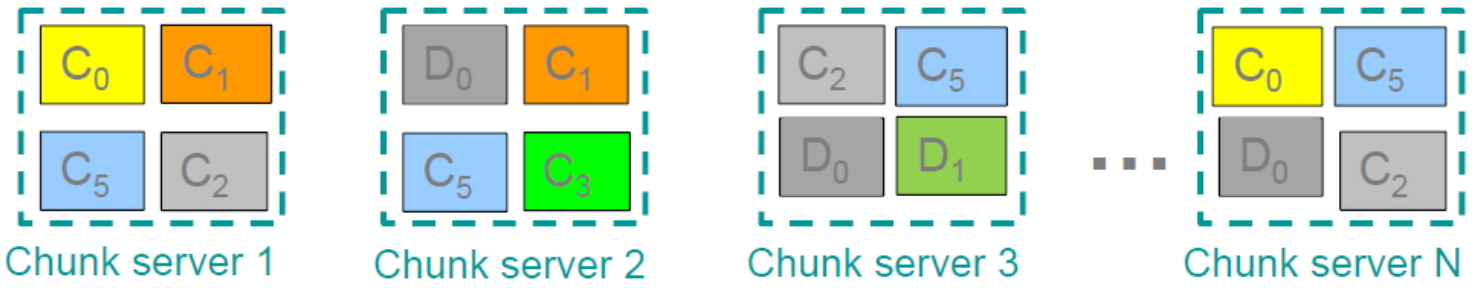
- Hadoop is not good at

  1. Low-latency data access.
  2. Lots of small files.
  3. Multiple writers, arbitrary file modifications

Hadoop Distributed File System

- Master node (name node(s))
  1. Stores metadata about where files are stored.
  2. Initializes data process tasks.
  3. If it is down, everything is gone! Might be replicated (secondary name node).

- Chunk servers (data nodes)
  1. HDFS is a much larger unit. (128 MB by default, general disk blocks 512 bytes).
  2. The reason is to minimize the cost of seeks.
  3. To insure against corrupted blocks and machine failure. Each chunk replicated 3x and kept in data nodes.
  4. Datanodes are the workhorses of the filesystem.
  5. Data kept in "chunks" (or formally called "blocks") are replicated and spread across machines.



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).

- Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well.

- YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code.

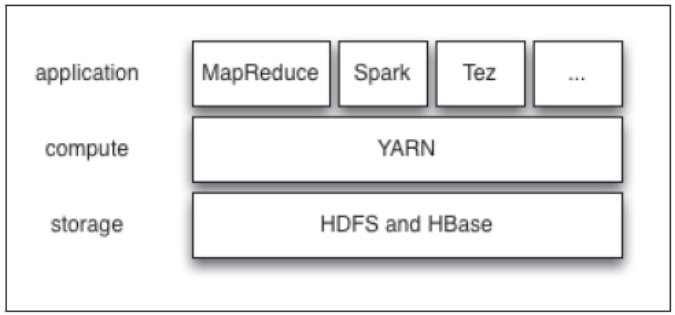


Figure 4-1. YARN applications

Working with files in HDFS –cont.

- HDFS provides a file system shell and a set of commands very similar to the systems of most Unix-like systems (e.g. bash).
- List all file in "/user/huangcs" of HDFS of

```
fisher/home/huangcs: hadoop fs -ls /user/huangcs
Found 3 items
-rw-r--r--   1 huangcs supergroup        631 2017-04-18 14:26 dup.RData
-rw-r--r--   1 huangcs supergroup         14 2017-04-18 17:01 matrix.csv
-rw-r--r--   1 huangcs supergroup        420 2017-04-18 14:54 mtcars.RData
```

- You cannot directly browse HDFS from terminal using cat or similar commands. HDFS is a logical file system and does not directly map to Unix file system.
- When you browse HDFS, you are getting your directory structure from namenode and actual data from datanodes.

Working with files in HDFS –cont.

- Create a directory "TestDir"

```
fisher/home/huangcs: hadoop fs -mkdir TestDir
fisher/home/huangcs: hadoop fs -ls /user/huangcs
Found 4 items
drwxr-xr-x   - huangcs supergroup          0 2017-04-21 15:30 TestDir
-rw-r--r--   1 huangcs supergroup        631 2017-04-18 14:26 dup.RData
-rw-r--r--   1 huangcs supergroup         14 2017-04-18 17:01 matrix.csv
-rw-r--r--   1 huangcs supergroup        420 2017-04-18 14:54 mtcars.RData
```

- To start using Hadoop to analyze your big datasets, you must put your data files in the HDFS first.

```
fisher/home/huangcs: hadoop fs -put small_doc.txt /user/huangcs
fisher/home/huangcs: hadoop fs -ls /user/huangcs
Found 5 items
drwxr-xr-x   - huangcs supergroup          0 2017-04-21 15:30 TestDir
-rw-r--r--   1 huangcs supergroup        631 2017-04-18 14:26 dup.RData
-rw-r--r--   1 huangcs supergroup         14 2017-04-18 17:01 matrix.csv
-rw-r--r--   1 huangcs supergroup        420 2017-04-18 14:54 mtcars.RData
-rw-r--r--   1 huangcs supergroup         74 2017-04-21 15:47 small_doc.txt
fisher/home/huangcs: hadoop fs -cat small_doc.txt
I am a student. You are a student.
He is a student. We are all students!
```

- From HDFS to local file system

```
fisher/home/huangcs: hadoop fs -get /user/huangcs/small_doc.txt ./FromHDFS_doc.txt
```

Working with files in HDFS –cont.

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

```
library("rhdfs"); hdfs.init()
#Initializing connection to the HDFS
hdfs.put("small_doc.txt", "/user/huangcs/small_doc.txt")
hdfs.ls("/user/huangcs")
hdfs.cat("/user/huangcs/small_doc.txt")
```

R Hadoop

- RHadoop is a great open source software framework of R for performing data analytics with the Hadoop platform via R functions.

- RHadoop has been developed by **Revolution Analytics**, which is the leading commercial provider of software and services based on the open source R project for statistical computing

- 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.

  1. rhdfs: R package for providing all Hadoop HDFS access to R.
  2. rmr2: R package for providing Hadoop MapReduce interfaces to R.
  3. plyrmr: R package making R work seamlessly with the Hadoop system for the storage and processing of big data on commodity clusters.

R Hadoop–cont.

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

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

- from.dfs() has transformed mtcars data into a key-value R object with a NULL as the key.

- However, this is not the normal way in which big data will enter HDFS; it is normally the responsibility of scalable data collection systems.

```
$key
NULL

$val
                   mpg cyl  disp  hp drat    wt  qsec vs am   ....
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1   ..
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1   ....
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1   ....
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0   ....
...................
```

Map Reduce

- For MapReduce, programmers need to design/migrate applications into two phases: Map and Reduce.

- Programmers have to design Map functions for processing a key-value pair to generate a set of intermediate key-value pairs, and Reduce functions to merge all the intermediate keys.

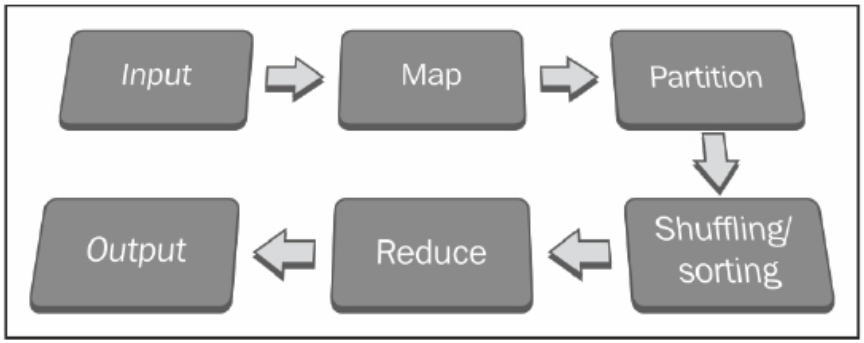
- The key aspect of the MapReduce algorithm is that if every Map and Reduce is independent of all other ongoing Maps and Reduces in the network, the operation will run in parallel on different keys and lists of data.

- The master node of the MapReduce daemon will take care of all the responsibilities of the MapReduce jobs, such as the execution, scheduling of Mappers, Reducers, Combiners, and Partitioners, the completion of the batch job.

Map Reduce–cont.

- This MapReduce paradigm is divided into two phases, Map and Reduce, that mainly deal with key-value pairs of data. The Map and Reduce tasks run sequentially in a cluster, and the output of the Map phase becomes the input of the Reduce phase.

- The process by which the system performs the sort and transfers the map outputs to the reducers as inputs is known as the shuffle.



Hadoop MapReduce

- Hadoop MapReduce processes vast amounts of data (multi-terabyte datasets) in parallel on large clusters (thousands of nodes) of commodity hardware in a reliable and fault tolerant manner.

- The four main stages of Hadoop MapReduce data processing are as follows:

  1. The loading of data into HDFS
  2. The execution of the Map phase
  3. Shuffling and sorting
  4. The execution of the Reduce phase

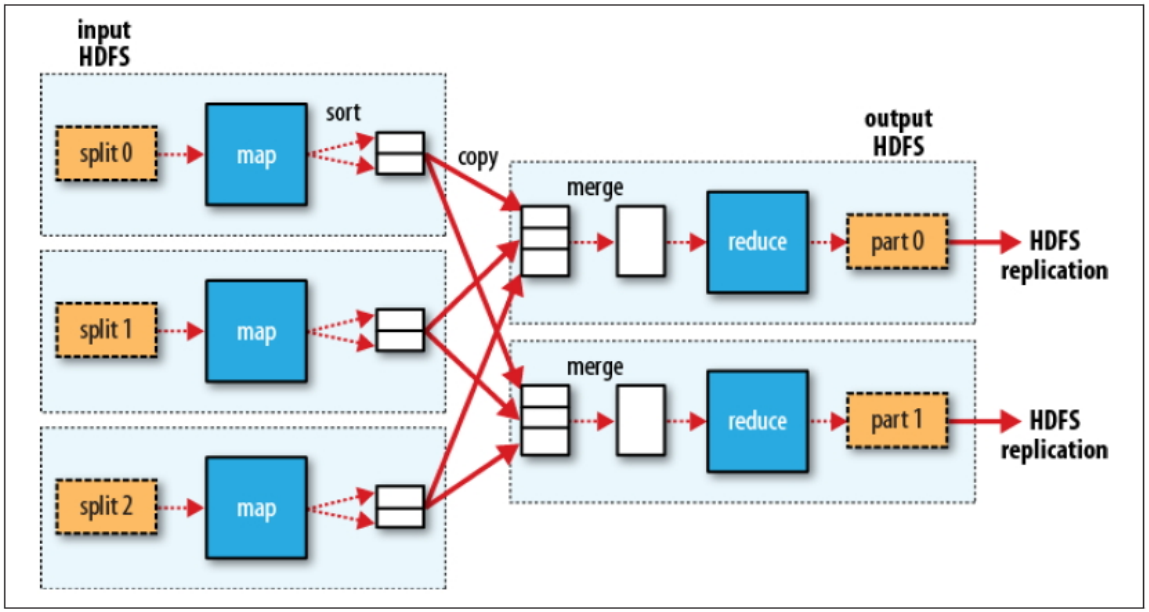


Figure 2-4. MapReduce data flow with multiple reduce tasks

Hadoop MapReduce–cont.

The four main stages of Hadoop MapReduce data processing are as follows:

1. The loading of data into HDFS
   The input dataset needs to be uploaded to the Hadoop directory so it can be used by MapReduce nodes.

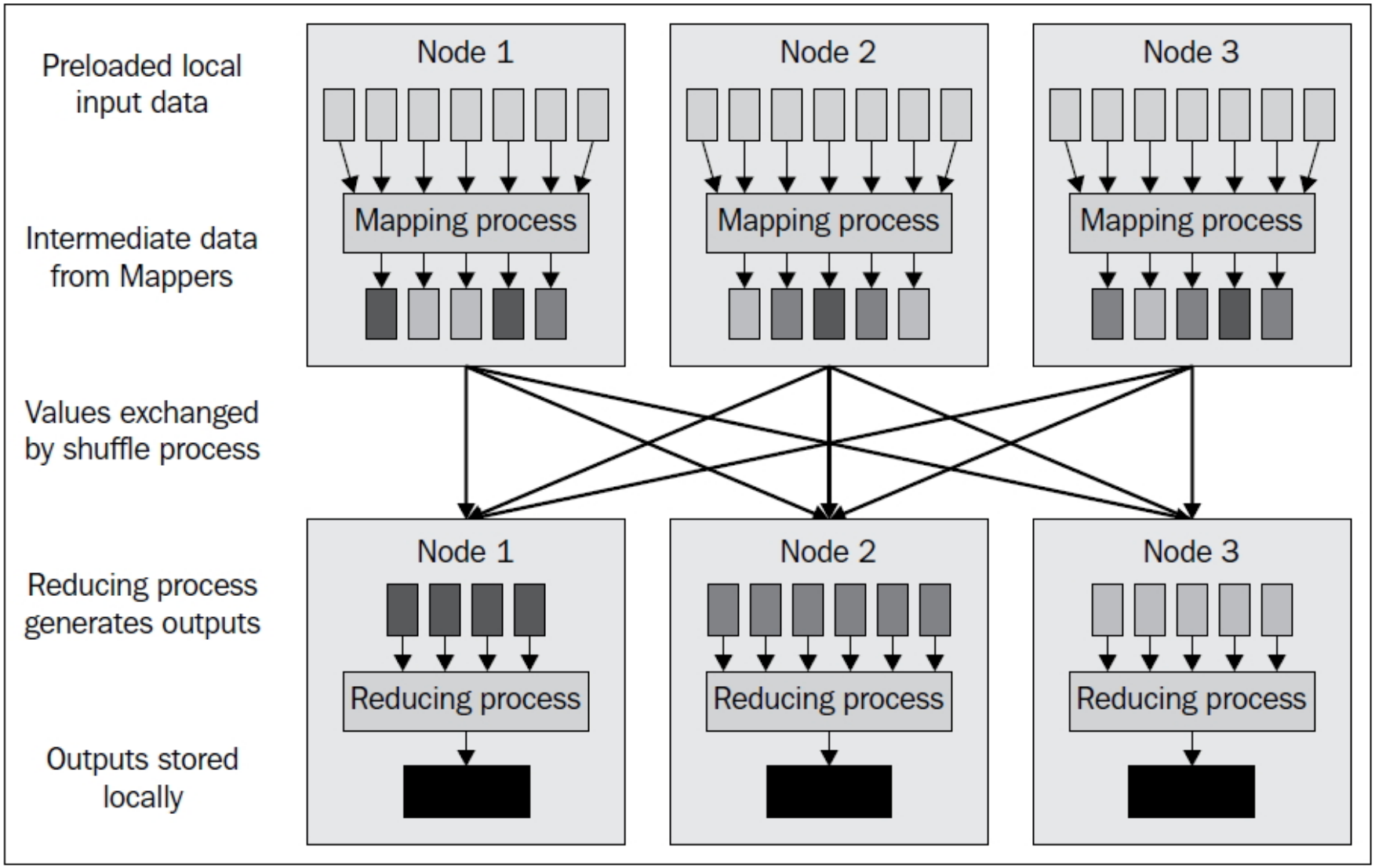
2. The execution of the Map phase
   The Map task reads this input split data as input (key, value) pairs provided to the Mapper method, which then produces intermediate (key, value) pairs.

3. Shuffling and sorting
   To optimize the MapReduce program, this intermediate phase is very important. As soon as the Mapper output from the Map phase is available, this intermediate phase will be called automatically.

4. The execution of the Reduce phase
   The Reducer method receives a list of input values from an input (key, list (value)) and aggregates them based on custom logic, and produces the output (key, value) pairs.

Hadoop MapReduce Dataflow

MapReduce dataflow with multiple nodes in a Hadoop cluster:



MapReduce dataflow

Hello World!

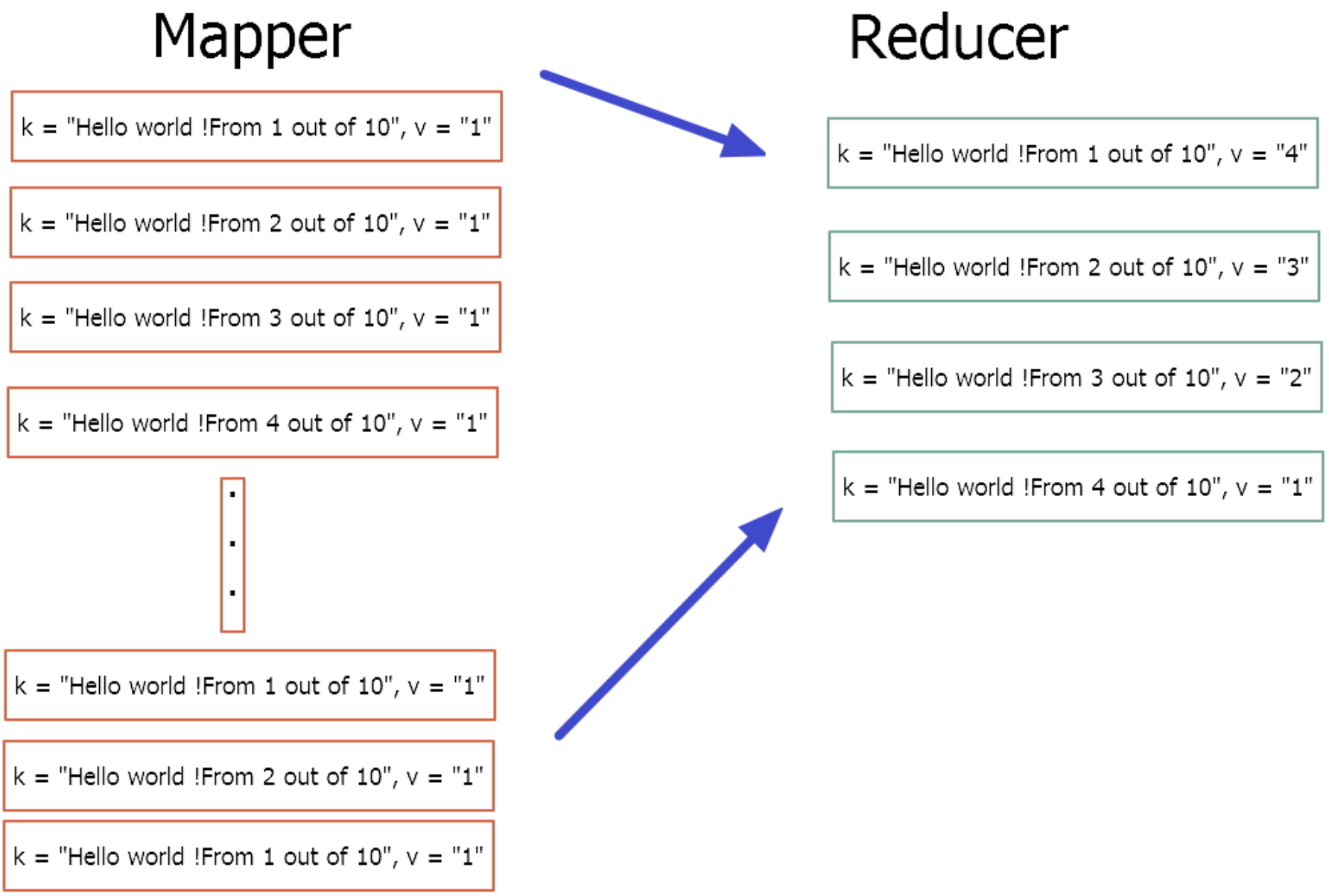
```
# "workers" is a list of functions to save the return values
#  of to.dfs()
workers = list(); numOfWorkers= 10; hello = "Hello world !"
# to.dfs() returns a function that contains the temporary location
# of file in HDFS. Here we consider putting 10 temporary small files
# on HDFS as 10 workers.
# These small files are just key-value pairs:
# 1->hello, 2-> hello, ..., 1 -> hello

kk = c('1','2','3','4','1','2','3','1','2','1');
for(i in 1:numOfWorkers)
    workers[[i]] = to.dfs(keyval(kk[[i]], hello))

kv= from.dfs( mapreduce(input=workers, map = function(k, v){
    k = paste(hello, " From " , k , " out of ", numOfWorkers, sep="")
    return(keyval(k, 1))
    },reduce = function(k, v)(keyval(k, sum(v)))
))
kv
$key
[1] "Hello world ! From 1 out of 10" "Hello world ! From 2 out of 10"
[3] "Hello world ! From 3 out of 10" "Hello world ! From 4 out of 10"

$val
[1] 4 3 2 1
```

Hello World!–cont.



MapReduce - again

- 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.
- The Reduce(k, (v1, v2, . . . , vn)) takes values v with the same key k generated by previous Map step, reduces (v1, v2, . . . , vn) 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 With R

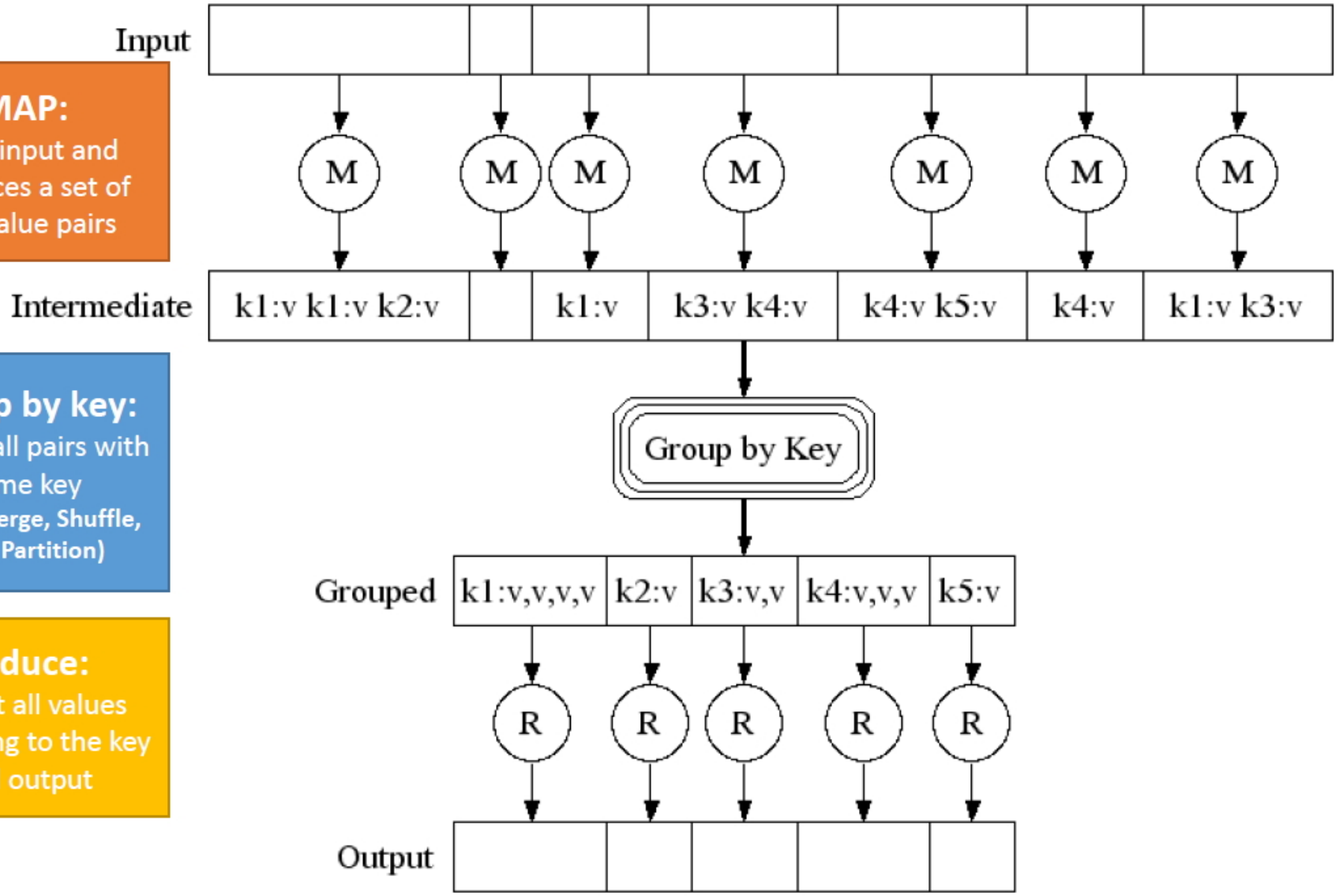
```
# template for mapper function
map = function(k, v){
      key = ...
      val = ...
      return(keyval(key, val))
      }
# template for mapreduce function
mapreduce(
  input = "inputfile",
  output = "outputfile",
  map = function(k, v){
    key = ...
    val = ...
    return(keyval(key, val)
  },
  reduce = function(k, v){
    key = ...
    val = ...
    return(keyval(key, val)
  }
)
```

MapReduce - a diagram

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

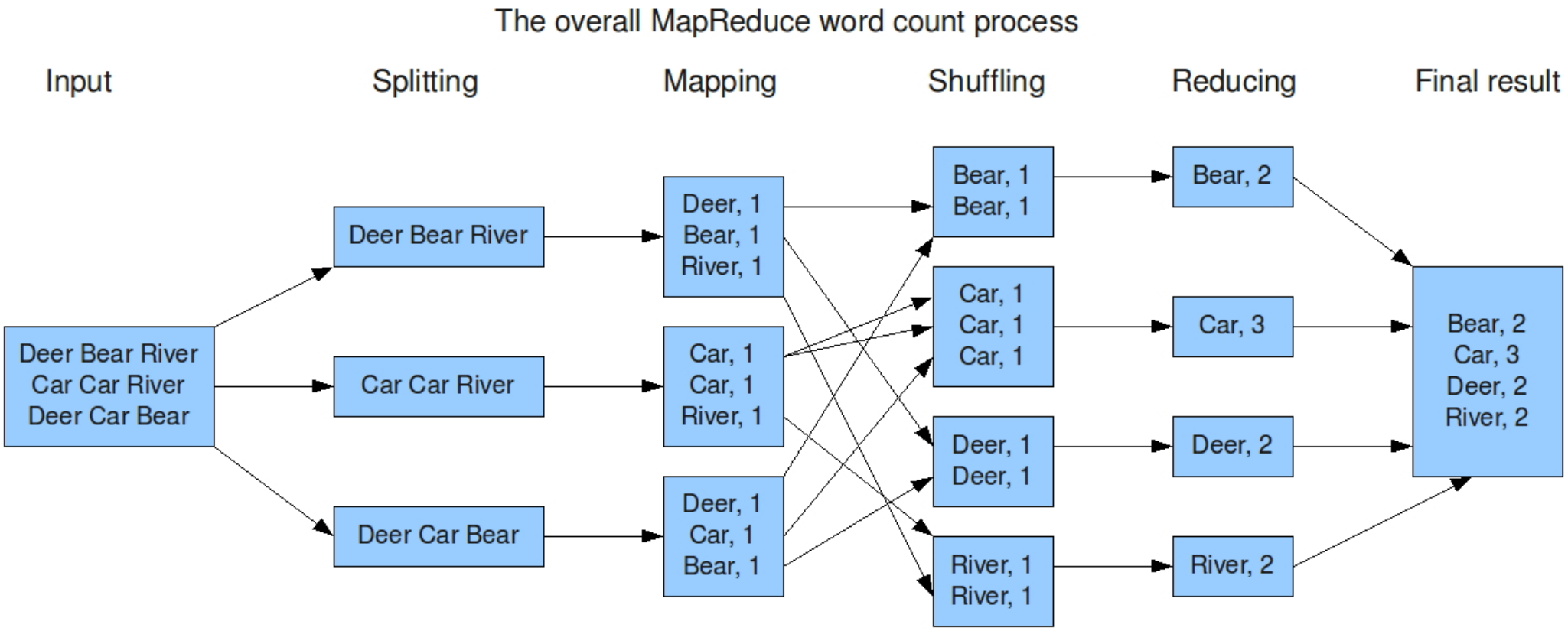
Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



Word Counting Example

The word count program has become a sort of "hello world" of the mapreduce world.



Word Counting in R

- to.dfs() can put the data in a file of your own choosing, but if you don't specify one it will create temp files and clean them up when done.

- It is not possible to write out big data with to.dfs(), to.dfs() is nonetheless very useful for a variety of uses like writing test cases, learning and debugging.

- The return value of to.dfs() is something we call a big data object. It is not in memory, it is some information that helps finding and managing the data. This way you can refer to very large data sets whose size exceeds memory limits.

- from.dfs() can transfer text into a key-value R object with a NULL as the key.

Word Counting in R–cont.

```
# 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='/user/huangcs/small_doc.txt', format="text")

> from.dfs(input = "/user/huangcs/small_doc.txt", format="text")
$key
NULL

$val
[1] "\tI am a student."                    "You are a student."
[3] "He is a student. We are all students!"
```

Word Counting in R–cont.

```
# 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)))
}
# You can also use length(counts)
wordcount("/user/huangcs/small_doc.txt",
           output = "/user/huangcs/small_doc_wc.RData")

from.dfs("/user/huangcs/small_doc_wc.RData")

$key
 [1] ""       "I"      "a"      "He"       "We"         "am"    "is"
 [8] "You"    "all"    "are"    "student" "students"

$val
 [1] 1 1 3 1 1 1 1 1 1 2 3 1
```

Homework: Please write a function findLongestWords() in MapReduce manner to find the longest word(s) in a given big plain text file

MapReduce – Filtering

- Let's do some daily data management tasks on "big" R datasets.

```
# We filter out some rows/observations. Keep only those
# with "am == 1".
mapreduce(input = "/user/huangcs/mtcars.RData",
          output = "/user/huangcs/mtcars_am_1.RData",
          input.format= "native",
          map = function(k, v) keyval(key = NULL,
                                      val= v[v$am== 1,]) )
from.dfs("/user/huangcs/mtcars_am_1.RData") # Check the result.

$key
NULL

$val
                mpg cyl  disp  hp drat    wt  qsec vs am gear  ....
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4
...................
```

MapReduce – Filtering–cont.

```
# 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")]) )
```

```
$val
                     wt  mpg
Mazda RX4         2.620 21.0
Mazda RX4 Wag     2.875 21.0
Datsun 710        2.320 22.8
...........................
```

- Note that we set "key = NULL" to get the whole data instead of splitting/mapping it into pieces.

- mtcars_wt_mpg.RData is a directory.

```
hdfs.ls("/user/huangcs/mtcars_wt_mpg.RData")
```

```
  permission   owner      group size          modtime
1 -rw-r--r-- huangcs supergroup    0 2017-04-26 12:21
2 -rw-r--r-- huangcs supergroup  122 2017-04-26 12:21
3 -rw-r--r-- huangcs supergroup 1700 2017-04-26 12:21
                                    file
1   /user/huangcs/mtcars_wt_mpg.RData/_SUCCESS
2 /user/huangcs/mtcars_wt_mpg.RData/part-00000
3 /user/huangcs/mtcars_wt_mpg.RData/part-00001
```

MapReduce – Understanding Key-Value Pairs

- After filtering, the output (with different keys) is split into two files.
- We have no control of how many files the mapreduce output are.
- In the following case, we have 2 files.

```
mapreduce(input = "/user/huangcs/mtcars.RData",
          input.format= "native",
          output = "/user/huangcs/mtcars_mpg2.RData",
map = function(k, v) keyval( v$mpg, val= 1),
reduce = function(k, v) keyval(k, cbind(k,sum(v))))

from.dfs(mapreduce(input = "/user/huangcs/mtcars_mpg2.RData",
                   input.format= "native",
                   map = function(k,v) {
                       n = length(k);
                       m = nrow(v);
                       keyval(n,m)}))
$key
[1] 17  8
$val
[1] 17  8
```

MapReduce – Understanding Key-Value Pairs–cont.

- Understanding cbind and rbind.
- They bind multiple rows or columns.
- They make the vector as row/column vector when a single vector is inputted.

```
> rbind(1:3, 4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

> rbind(1:3)
     [,1] [,2] [,3]
[1,]    1    2    3

> cbind(1:2,4:5)
     [,1] [,2]
[1,]    1    4
[2,]    2    5

> cbind(1,3)
     [,1] [,2]
[1,]    1    3

> cbind(1:3)
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

MapReduce – Understanding Key-Value Pairs–cont.

- Before reducer, we produce ONLY one key-value pair.
- It is unclear if mapper produces 1 or 32 key-value pairs.
- In reducer, the keyval function output 32 key-value pairs.

```
from.dfs(mapreduce(input = "/user/huangcs/mtcars.RData",
         input.format= "native",
         map = function(k, v) keyval(key = 1,
                                     val= v[,c("wt","mpg")]),
         reduce = function(k, v) {
             n = nrow(v);
             keyval(v, n)}))
$key
           wt  mpg
21      2.620 21.0
21.1    2.875 21.0
22.8    2.320 22.8
21.4    3.215 21.4
18.7    3.440 18.7
...............

$val
 [1] 32 32 32 32 32 32 32 32 32 32 32 32 32 ........
```

MapReduce – Understanding Key-Value Pairs–cont.

- Now, with "key = 1:32", maper produces 32 key-value pairs, i.e. one for each row.

- In reducer, the keyval function output 32 key-value pairs.

```
from.dfs(mapreduce(input = "/user/huangcs/mtcars.RData",
         input.format= "native",
         map = function(k, v) keyval(key = 1:32,
                                     val= v[,c("wt","mpg")]),
         reduce = function(k, v) {
             n = nrow(v);
             keyval(v, n)}))
$key
           wt  mpg
21      2.620 21.0
21.1    2.875 21.0
22.8    2.320 22.8
21.4    3.215 21.4
18.7    3.440 18.7
.................

$val
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 ........................
```

MapReduce – Understanding Key-Value Pairs–cont.

- Now, with "key = t(rep(1, 32))", maper "should" produces 32 key-value pairs, i.e. one for each row. However, after "Combiner", they get aggregated into one key-value pair before entering reducer.

- In reducer, the keyval function output 32 key-value pairs.

- Make sure what your key-value pairs are everywhere.

```
from.dfs(mapreduce(input = "/user/huangcs/mtcars.RData",
          input.format= "native",
          map = function(k, v) keyval(key = t(rep(1, 32)),
                                      val= v[,c("wt","mpg")]),
          reduce = function(k, v) {
              n = nrow(v);
              keyval(v, n)}))
```

```
$key
          wt  mpg
21     2.620 21.0
21.1   2.875 21.0
22.8   2.320 22.8
21.4   3.215 21.4
18.7   3.440 18.7
...............
$val
 [1] 32 32 32 32 32 32 32 32 32 32 32 32 32 ........
```

MapReduce – Filtering–cont.

- The Reduce() makes aggregation tasks easier.
- The Mapper takes one row of input k-value pair at a time, then Combiner aggregates the value-set with the same key value before entering Reducer.

```
# 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, "/user/huangcs/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 – Sorting

- We can do it in both Map() and Reduce().
- Too many Maps requires more memory in namenode.
- Too few Maps imposes high computation on a few datanodes.

```
# Sort by gear and mpg, order(v$mpg) returns the row indices of v
# in the increasing order of "mpg".
from.dfs(mapreduce(
         input = "/user/huangcs/mtcars.RData",
                 input.format= "native",
         map = function(k, v) keyval( v$gear, val= v),
         reduce = function(k, v) keyval(k, v[order(v$mpg),])))
$key
 [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 5 ...

$val
                     mpg cyl  disp ...   qsec vs am gear carb
Cadillac Fleetwood  10.4   8 472.0 ...  17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 ...  17.82  0  0    3    4
Camaro Z28          13.3   8 350.0 ...  15.41  0  0    3    4
Duster 360          14.3   8 360.0 ...  15.84  0  0    3    4
.............
```

MapReduce – Sorting

```
# Using composite key also works, too.
from.dfs(mapreduce(
         input = "/user/huangcs/mtcars.RData",
         map = function(k, v) keyval( v[, c("gear", "mpg")], val= v),
         reduce = function(k, v) keyval(k, v)))
$key
       gear  mpg
10.4      3 10.4
10.4.1    3 10.4
13.3      3 13.3
14.3      3 14.3
.........

$val
                     mpg cyl  disp ...   qsec vs am gear carb
Cadillac Fleetwood  10.4   8 472.0 ...  17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 ...  17.82  0  0    3    4
Camaro Z28          13.3   8 350.0 ...  15.41  0  0    3    4
Duster 360          14.3   8 360.0 ...  15.84  0  0    3    4
...............
```

Homework: Please write a function to sort mtcars.RData according to 3 features, "gear", "mpg" and "disp" in MapReduce manner.

MapReduce – Sorting–cont.

- Assuming we have `"big" data that needed to be sorted, after mapreduce, how do we allocate the "i-th" record from the sorted file?

- Note that we can NOT load the sorted file to R-studio and use R command to find the "i-th" record, not enough memory.

- We could use unix command to allocate the "i-th" record according to Prof. Kang.

MapReduce – Remove Duplicates

- Remove duplicate records in a big R data frame.
- 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.

```
# 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, "/user/huangcs/dup.RData")
# Remove duplicates
from.dfs( mapreduce( input = "/user/huangcs/dup.RData",
                     map = function(k, v)
                        keyval(key = v, val= 1 ),
                     reduce = function(k, V)
                        keyval(key = k, val= sum(V))))
```

```
$key
  x1 x2
1  a  1
2  b  2
3  b  3
4  c  4

$val
[1] 2 1 1 2
```

MapReduce – Remove Duplicates–cont.

- In mapper, we set val=1:6 to see it is infact 6 key-value pairs.

```
# 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 = "/user/huangcs/dup.RData",
                     map = function(k, v)
                        keyval(key = v, val= 1:6 ),
                     reduce = function(k, V)
                        keyval(key = k, val= sum(V))))
```

```
$key
  x1 x2
1  a  1
2  b  2
3  b  3
4  c  4

$val
[1] 3 3 4 11
```

MapReduce – Join

- Merge two tables/datasets horizontally.
- Use rmr2::equijoin(), instead of writing multiple complicated MapReduce functions.

```
# create 2 temporary data frames, A & B (kv, keys will be set to NULL)
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 = "/user/huangcs/A.RData");
to.dfs(kv= B, output = "/user/huangcs/B.RData");
# Inner join
from.dfs( equijoin(left.input= "/user/huangcs/A.RData",
          map.left= function(k,v) keyval(v$id, v),
          right.input= "/user/huangcs/B.RData",
          map.right= function(k,v) keyval(v$id, v),
          outer = NULL))
# Specify outer = "left", "right", or "full" for other outer joins
```

```
$key
[1] 1 3 5

$val
  id val
1  1   a
2  3   x
3  5   c
```

```
$key
[1] 3 5 6

$val
  id val
1  3   x
2  5   y
3  6   z
```

```
$key
[1] 3 5

$val
id.l val.l id.r val.r
1      3     x     3     x
1.1    5     c     5     y
```

MapReduce – Concatenation

- Concatenate multiple data sets vertically.

```
# We here simply specify multiple inputs
from.dfs(mapreduce(
   input = c("/user/huangcs/A.RData","/user/huangcs/B.RData"),
   map = function(k, v) keyval(NULL, v)))
# How about some aggregation tasks? Say how many val= 'x'?
from.dfs(mapreduce(
   input = c("/user/huangcs/A.RData","/user/huangcs/B.RData"),
   map = function(k, v) keyval(v$id, v),
   reduce = function(k, v) keyval(k, sum(v$val== 'x'))
))
```

```
$key
NULL

$val
   id val
1    1   a
2    3   x
3    5   c
1.1  3   x
2.1  5   y
3.1  6   z
```

```
$key
[1] 1 3 5 6

$val
[1] 0 2 0 0
```

MapReduce – Matrix Transpose

```
from.dfs("/user/huangcs/matrix.csv", format="csv")
$key
NULL

$val
     V1
1 1,2,3
2 a,b,c
```

```
from.dfs(mapreduce(
  input = "/user/huangcs/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(as.character(unlist(V))))
))
$key
NULL

$val
     [,1] [,2]
[1,] "1"  "a"
[2,] "2"  "b"
[3,] "3"  "c"
```

MapReduce – Matrix Transpose–cont.

- How many key pairs we have from last example ?

```
from.dfs(mapreduce(
  input = "/user/huangcs/matrix.csv",
  input.format= make.input.format(format = "csv",
                                  mode = "text", sep= ","),
  # Emit index of each cell (as row) as the key
  # But after unlist, values are read in columnwise

   map = function(k, v) keyval( 1:length(v), unlist(unclass(v))),


  reduce = function(k, V) keyval(NULL,rbind(as.character(unlist(V))))
))
After mapper
$key
[1] 1 1 2 2 3 3

$val
[1] 1 b a 3 2 c
Levels: 1 2 3 a b c

After reducer
$key
NULL

$val
     [,1] [,2]
[1,] "1"  "b"
[2,] "a"  "3"
[3,] "2"  "c"
```

MapReduce – Matrix Transpose–cont.

- How many key pairs we have from last example ?

```
from.dfs(mapreduce(
  input = "/user/huangcs/matrix.csv",
  input.format= make.input.format(format = "csv",
                                  mode = "text", sep= ","),
  # Emit index of length of the unlist matrix.

 map=function(k, v) keyval( 1:length(unlist(v)), unlist(unclass(v))),

  reduce = function(k, V) keyval(NULL,rbind(as.character(unlist(V))))
))
After mapper
$key
[1] 1 2 3 4 5 6

$val
[1] 1 a 2 b 3 c
Levels: 1 2 3 a b c

After reducer
$key
NULL

$val
     [,1]
[1,] "1"
[2,] "a"
[3,] "2"
[4,] "b"
[5,] "3"
[6,] "c"
```

MapReduce – All Reduce

```
from.dfs("/user/huangcs/mtcars_mpg2.RData")
$key
 [1] 10.4 13.3 14.3 14.7 15.0 15.2 15.5 15.8 16.4 17.3 17.8 ......
$val
         k
 [1,] 10.4 2
 [2,] 13.3 1
 [3,] 14.3 1
 [4,] 14.7 1
............
```

```
mapreduce(input = "/user/huangcs/mtcars_mpg2.RData",
            input.format= "native",
          output = "/user/huangcs/mtcars_mpg22.RData",
          map = function(k,v) keyval(1,v),
          reduce = function(k,v) keyval (v[,1], rbind(unlist(v[,2])))
          )# Note that the results are not ordering.
$key
 [1] 21.5 22.8 24.4 26.0 27.3 30.4 32.4 33.9 10.4 13.3 14.3 ........
$val
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] .....
 [1,]    1    2    1    1    1    2    1    1    2     1     1 .....
 [2,]    1    2    1    1    1    2    1    1    2     1     1 .....
 [3,]    1    2    1    1    1    2    1    1    2     1     1 .....
 [4,]    1    2    1    1    1    2    1    1    2     1     1 .....
............
```

MapReduce – Crosstab

- Checking out frequency tables or crosstabs.

```
# 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("/user/huangcs/mtcars.RData",
                        x = 'am', y = 'gear', ylevels= c(3,4,5)));

#output of mapper
$key
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$val
 [1] 4 4 4 4 4 4 4 5 5 5 5 5 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3
```

MapReduce – Crosstab–cont.

```
# 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("/user/huangcs/mtcars.RData",
                     x = "am", y = "gear", ylevels= c(3,4,5)))
#factor encodes Levels to a vector
#table uses the cross-classifying factors to build a contingency
#table of the counts at each combination of factor levels.
#rbind does row bind within the same k.
#output of reducer
$key
[1] 0 1

$val
   3 4 5
0 15 4 0
1  0 8 5
```

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!

```
# 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("/user/huangcs/mtcars.RData",
            "/user/huangcs/chainSort1.RData", "am", "mpg")
from.dfs("/user/huangcs/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).

How many Map and Reduce jobs? –cont.

- 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.

- 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 (128 MB), 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.

- 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 – Create Virtual Key-Value Pairs

- 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()
[1] "/tmp/file2fa170b5c0fb"
```

- 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 – Computing π

```
# "workers" is a list of functions to save the return values
# of to.dfs()
workers = list(); numOfWorkers= 40; N = 1000;
# number of points for computing pi
# to.dfs() returns a function that contains the temporary location
# of file in HDFS. Here we consider putting 40 temporary small files
# on HDFS as 40 workers.
# These small files are just key-value pairs:
# 1-> 1000, 2-> 1000, ..., 40 -> 1000
for(i in 1:numOfWorkers)
    workers[[i]] = to.dfs(keyval(i, N))

pi_kv= from.dfs( mapreduce(input=workers, map = function(k, v){
# xcoors are the interpolation points of \frac{1}{1+x^2} in [0, 1].
# There are totally N points, and each k-v pair (worker) contains
# N/numOfWorkers points.
# \pi is approximated by \pi = \int_0^1 4/(1+x^2) dx.
      xcoords = (seq(from = k, to = v, by = numOfWorkers) - 0.5)/v;
      f = 4/(1 + xcoords^2);
      return(keyval(1, sum(f)))},
    reduce = function(k, v)(keyval(k, sum(v)))
))
pi_kv[[2]]/N
```

MapReduce – Central Limit Theory

Homework : Please demonstrate the Central Limit Theory in MapReduce manner. Please put 20 temporary small files on HDFS as 20 workers. And let each worker compute $\{\bar{x_i}\}_{i=1}^{50}$, where each $\bar{x_i}$ is the mean of 100 uniformly-generated numbers (between 0 and 1).

Homework : Compute π in MapReduce manner by sampling points with $[0,1]^2$ and comput the probability of points falling with Disk$(x, 0.5)$, $x = [0.5, 0.5]$. The result is not as accurate as using $\pi = \int_0^1 \frac{4}{(1+x^2)}\, dx$, why ?

Linear Regression or Least Square Approximation

Given a data set $D = \{(\mathbf{x}_j, y_j) | (\mathbf{x}_j, y_j) \in \mathbb{R}^{M+1}, j = 1, \ldots, N\}$. We have the following minimization problem,

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^M} \quad \frac{1}{2} \sum_{j=1}^{N} (< \mathbf{x}_j, \boldsymbol{\beta} > - y_j)^2, \quad \text{or}$$

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^M} \quad \frac{1}{2} \|\mathcal{X}\boldsymbol{\beta} - \mathbf{y}\|^2,$$

where $\mathcal{X} \in \mathbb{R}^{N \times M}$, and $N \sim \infty$.

Let $g(\boldsymbol{\beta}) = \frac{1}{2}\|\mathcal{X}\boldsymbol{\beta} - \mathbf{y}\|^2$, to minimize $g(\boldsymbol{\beta})$, we have to find $\boldsymbol{\beta}^*$ such that $\nabla g(\boldsymbol{\beta}^*) = 0$. Now,

$$\begin{aligned} \frac{\partial g(\boldsymbol{\beta})}{\partial \beta_i} &= \partial_{\beta_i} \frac{1}{2} (\mathcal{X}\boldsymbol{\beta} - \mathbf{y}, \mathcal{X}\boldsymbol{\beta} - \mathbf{y}) = (\mathcal{X}\boldsymbol{\beta} - \mathbf{y}, \partial_{\beta_i}(\mathcal{X}\boldsymbol{\beta} - \mathbf{y})) \\ &= (\mathcal{X}\boldsymbol{\beta} - \mathbf{y}, \mathcal{X}e_i) = \left(\mathcal{X}^t\mathcal{X}\boldsymbol{\beta} - \mathcal{X}^t\mathbf{y}, e_i\right), \quad \forall\, i, \end{aligned}$$

i.e. we need to solve $\mathcal{X}^t\mathcal{X}\boldsymbol{\beta}^* = \mathcal{X}^t\mathbf{y}$. This can NOT be done if $N \sim \infty$

Gradient Method (Steepest Descent Method)

Given a smooth function, $g : \mathbb{R}^M \to \mathbb{R}$. We consider an iterative method for $\min_{\boldsymbol{\beta}\in\mathbb{R}^M} g(\boldsymbol{\beta})$ of the form:
Given an initial approximation $\boldsymbol{\beta}^0 \in \mathbb{R}^M$ of the exact solution $\boldsymbol{\beta}$, find successive approximations $\boldsymbol{\beta}^k \in \mathbb{R}^M$, $k = 1, 2, \ldots$ of the form

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k + \alpha_k \mathbf{d}^k, k = 1, 2, \ldots,$$

We denote by $\nabla g (= \mathbf{g}')$ the gradient of $g(\boldsymbol{\beta})$, i.e.

$$\mathbf{g}' = \nabla g = (\frac{\partial g}{\partial \beta_1}, \frac{\partial g}{\partial \beta_2}, \cdots, \frac{\partial g}{\partial \beta_M}).$$

Futher, define the Hessian matrix of g to be the $M \times M$ matrix $\mathbf{g}''$

$$\mathbf{g}'' = \begin{bmatrix} \frac{\partial^2 g}{\partial \beta_1^2} & \cdots & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_M} \\ & \cdots & \\ \frac{\partial^2 g}{\partial \beta_M \partial \beta_1} & \cdots & \frac{\partial^2 g}{\partial \beta_M^2} \end{bmatrix}.$$

Gradient Method–cont.

By Taylor's formula, we have

$$\begin{aligned} g(\boldsymbol{\beta}^{k+1}) &= g(\boldsymbol{\beta}^k) + \alpha_k \left\langle \mathbf{g}'(\boldsymbol{\beta}^k), \mathbf{d}^k \right\rangle + \frac{\alpha_k^2}{2} \left\langle \mathbf{g}''(\xi)\mathbf{d}^k, \mathbf{d}^k \right\rangle \\ &= g(\boldsymbol{\beta}^k) + \alpha_k \left\langle \mathbf{g}'(\boldsymbol{\beta}^k), \mathbf{d}^k \right\rangle + \mathcal{O}(\alpha_k^2) \end{aligned}$$

where ξ lies on the line segment between $\boldsymbol{\beta}^{k+1}$ and $\boldsymbol{\beta}^k$.

It follows that $g(\boldsymbol{\beta}^{k+1}) < g(\boldsymbol{\beta}^k)$ if $\left\langle \mathbf{g}'(\boldsymbol{\beta}^k), \mathbf{d}^k \right\rangle < 0$ and α_k is sufficiently small. We say that $\mathbf{d}^k$ is a descent direction for g. In particular, we choose

$$\mathbf{d}^k = -\mathbf{g}'(\boldsymbol{\beta}^k), \text{ if } \mathbf{g}'(\boldsymbol{\beta}^k) \neq 0.$$

To choose the best α_k we may cosider the following optimization problem

$$g(\boldsymbol{\beta}^k + \alpha_k \mathbf{d}^k) = \min_{\alpha} g(\boldsymbol{\beta}^k + \alpha \mathbf{d}^k),$$

and we shall omit this for the moment.

MapReduce – Gradient Method

From Gradient Method, we need to compute

$$\mathbf{d}^k = -\nabla g(\boldsymbol{\beta}^k), \text{ where } g(\boldsymbol{\beta}) = \frac{1}{2}\sum_{j=1}^{N}(< \mathbf{x}_j, \boldsymbol{\beta} > -y_j)^2.$$

Now,

$$\partial_{\beta_i} g(\boldsymbol{\beta}^k) = \sum_{j=1}^{N}(< \mathbf{x}_j, \boldsymbol{\beta}^k > -y_j)x_{ji}.$$

$$\begin{pmatrix} x_{1i} \\ x_{2i} \\ \vdots \\ \vdots \\ x_{Ni} \end{pmatrix} \cdot \left(\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \mathbf{x}_2 & - \\ & \vdots & \\ & \vdots & \\ - & \mathbf{x}_N & - \end{bmatrix} \begin{bmatrix} \beta_1^k \\ \beta_2^k \\ \vdots \\ \beta_M^k \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_N \end{bmatrix} \right)$$

| sum along
| different k-v pair
↓

Thereore, with each k-v pair, we compute

$$\mathbf{d}^k_{key,v} = -\mathcal{X}_{key,v}{}^t(\mathcal{X}_{key,v}\boldsymbol{\beta}^k - \mathbf{y}_{key,v}).$$

And, $\mathbf{d}^k = \text{Reduce}_{key,v}(\mathbf{d}^k_{key,v})$.

MapReduce – Gradient Method–cont.

```
#We look for a plane z = ax + by + c, with variables as (x, y, 1)
#Preparing Data, with polynomial as ax + by + c = 0.
a = 2; b = 3; c= 5;

#number of samples, x = var1, y = var2, var3 = 1.
data_size<-1000

# var is random variable which comes from a uniform distribution
var1 <- runif(data_size, 1, 10)
var2 <- runif(data_size, 1, 10)
var3 <- rep(1,data_size)

#res is function of x, y, 1  and noise
#res represents given data zi and it is away from
#the plane z = ax + by + c by rnorm(data_size)
res <- a*var1 + b*var2 + c*var3 + rnorm(data_size)

sum_data=cbind(res, var1, var2, var3)

#Output = "sum_data_split.RData"
```

MapReduce – Gradient Method–cont.

```
lr.map = function(k, M){
 y = M[,1]
 x = M[,-1]
 keyval(t(t(seq(1,3,1))),
        unlist(unclass(t(x) %*% ((x %*% beta - y)))))}

lr.reduce = function(k, Z)
    keyval(k, sum(Z))

#initial guess beta_0
beta =  t(t(c(3.1, 4.1, 4)));
alpha = 0.000001;
steps = 20;
for (i in 1:steps) {
        gradient = values(
            from.dfs(mapreduce("/user/huangcs/sum_data_split.RData",
                               map = lr.map,
                               reduce = lr.reduce,
                               combine = T)))
        beta = beta - alpha*t(t(gradient)) }
```

Goodbye MapReduce

I'll be back.

"We" will work on PostgreSQL.