Report on MapReduce: Simplified Data Processing on Large Clusters

The problem statement comes due to the present scenario of large-scale computing and to change from Traditional single Node Architecture to cluster Architecture
In traditional Single Node, Architecture is not sufficient for large-scale data
For example: Suppose if we take 10 billion web pages , 10kB/page => 100TB
The computer reads 50 MB/sec from the disk
Time to read the web:2 million sec = 23 days = 1 month
And also in cluster architecture conducting large-scale computing on commodity Hardware is difficult and takes care of messy details like **Parallelization, Fault-Tolerance, Data distribution, Load balancing**
And copying data over a network also takes time.
The solution to all this comes from the paper that is **MapReduce**
The properties of Mapreduce are
1)Brings **Computation** close to data
2)Store replicas of files for **Reliability**
3)And follows Google's data manipulation model
4)Elegant way to work with large-scale computation
5)Storage infrastructure: Distributed File System

MapReduce: Map + Reduce

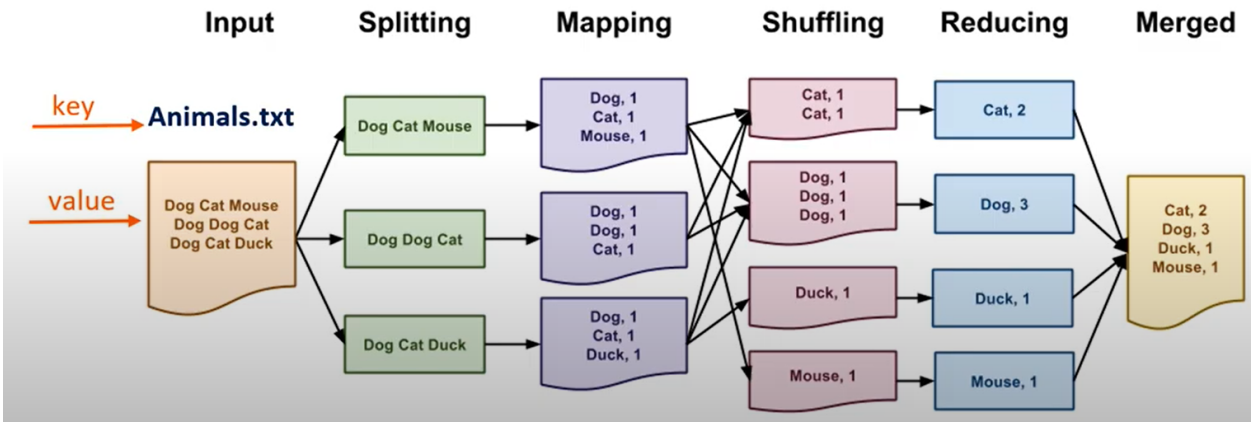
map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

The computation takes a set of input key/value pairs and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.
Map(k,v) => <k' ,v'>
1)Written by the user
2)And scans input file(1 record at a time)
Key => filename
Value => file content
3)Produces a set of intermediate(key, value) pairs.

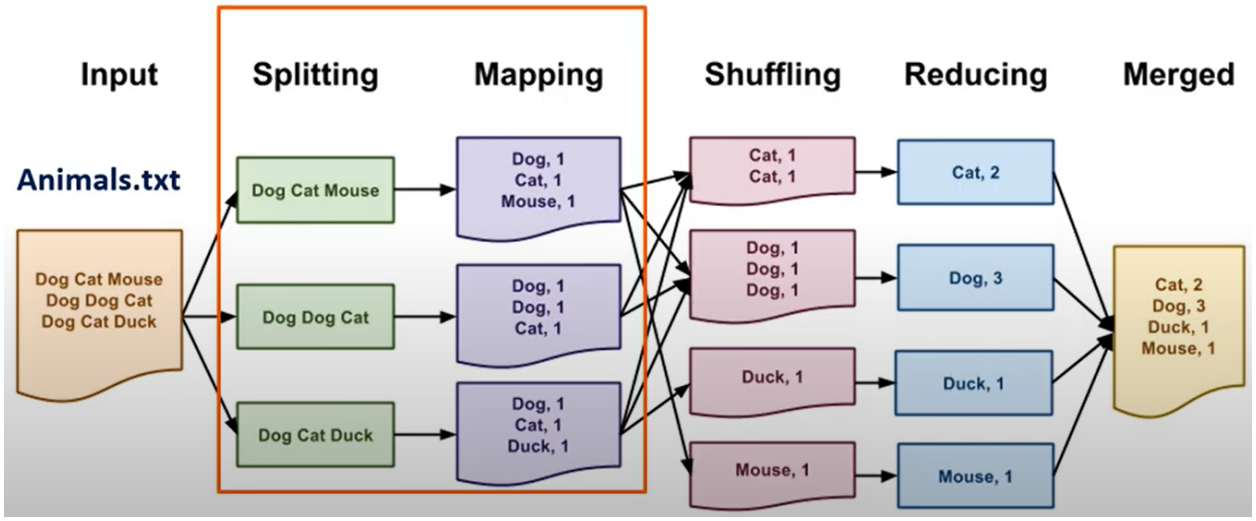
Groups by intermediate key: sort and shuffle
Reduce(k' ,<v'.>) => <k',v''>
1)Also written by the user
2)Accepts an intermediate key and a set of values for that key
3)Produce a smaller set of values for that key
Aggregate, summarize, filter or transform

MapReduce: Word Count Example

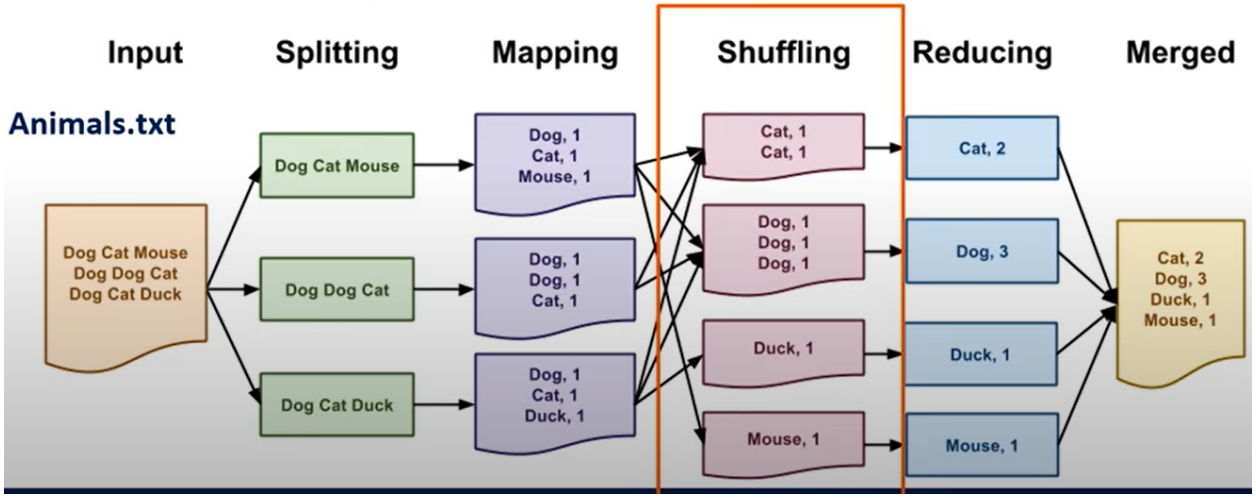
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```



```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```




```
reduce(String key, Iterator values)
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



MapReduce: More Examples:

String Query:

Map: Emits a string if it matches the supplied pattern

Reduce: An identity function that just copies the supplied intermediate data to output
Count URL access frequency:
Map: Outputs <URL, ‘1’>
Reduce: adds together all ‘1’ and emits <URL, total count>
Distributed Sort:
Map:Emits <key,record>
Reduce: Just emits all pairs unchanged(because of ordering guarantees)

MapReduce Implementation Overview:

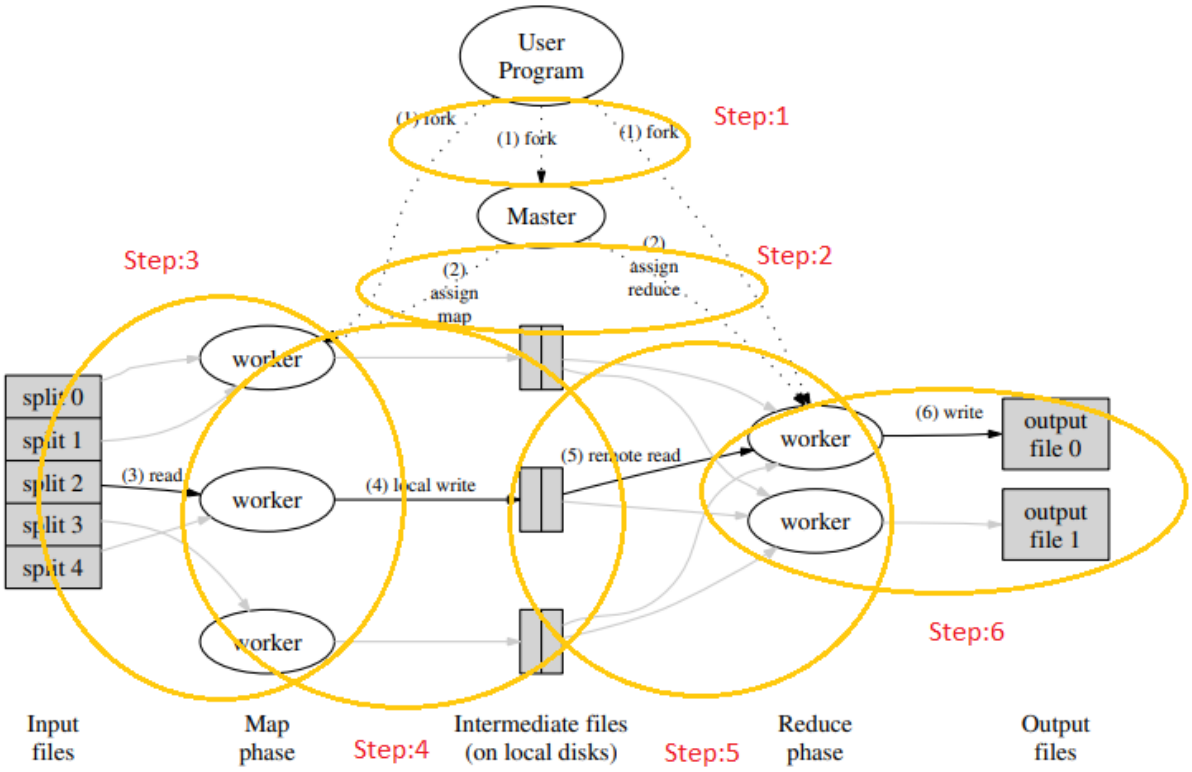


Figure 1: Execution overview

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1)MapReduce library in user program splits the input files into M pieces
It starts up many copies of the program on worker and master machines
2)Master has M map tasks and R reduce tasks to assign.It picks idle workers and Assigns each one a map task or a reduce task
3)Map worker reads contents from one of the splits.It parses key/value pairs from Input and passes each pair to a user-defined Map function.
4)Intermediate key/value pairs produced by mapper are buffered in memory
periodically ,the buffered pairs are written to the local disk.
Locations of buffered pairs on the local disk are passed to master
5)Reduce worker will be notified by master about locations of intermediate key/value pairs, and uses remote procedure calls to read data from mapper's local disk.
A sort/shuffle is done before reducer gets input,so that all occurrences of the same key are grouped together
6)Reduce worker passes each unique intermediate key and the corresponding set of values to Reduce function.Output will be appended to a final output file for This partition
When all map and reduce tasks complete, the MapReduce call in user program returns back to the user code

Fault tolerance:
Map worker failure
Map tasks in-progress at failed worker are reset to idle for rescheduling
Map tasks completed at failed worker are also reset to idle!
Output of mapper is stored on local disk of a failed machine,inaccessible!
Reduce workers are notified when task is rescheduled on another worker
Reduce worker failure
Only in-progress reduce tasks are reset to idle
Reduce task is restarted
Master failure
In practice:MapReduce task is aborted and client is notified
Ideally:Keep periodic checkpoints.If failed, restart from the last checkpointed state

And no of Map and Reduce tasks
M pieces map phase + R pieces reduce phase

Such that M+R much larger than number of workers/nodes in the cluster
Improves dynamic load balancing and speed up recovery from worker failures
R is usually much smaller than M
Constrained by users, output is spread across R files

Refinement for MapReduce:

1.Backup Tasks

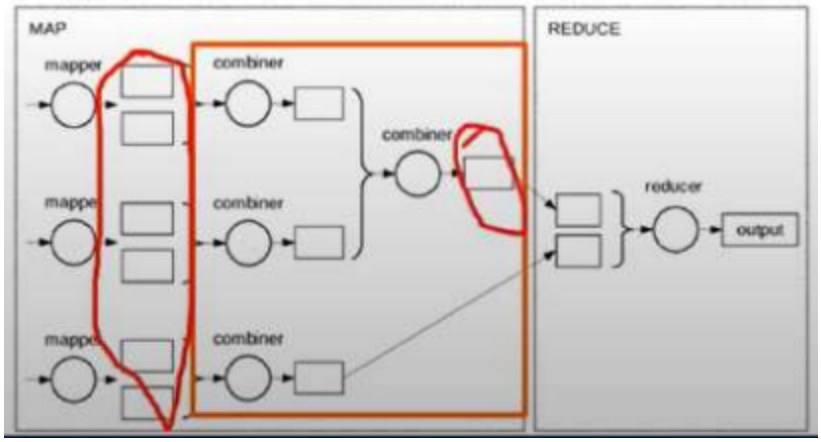
The problem is some slow workers significantly lengthen the task completion time,”straggler”
Bad disk,other task on the machine
No we can do near the end of MapReduce operation, spawn backup copies of in-progress tasks
Whoever finishes first wins
Dramatically shortens job completion time with a little increase in computational resources

2.Partitioning Function

The problem is by default the data are partitioned using hashing hask(key) mod R.
In some cases, user wants to control how keys get partitioned
To make it available we can enable overriding partition function
eg:,hask(hostname(URL)) mod R
Ensures URLs from a host end up in the same output file

3.Combiner Function

The problem is often a Map task will produce many pairs of the form (k,v1),(k,v2),... for the same key k
Eg:,popular words in the word count example
Sending all these counts through network is time consuming
We can pre-aggregating values in the mapper using a combiner
Combine(k,list of v) => <k,count>
Much less data to be copied and shuffled.Save network time.



Results:

In Grep program
Looking for a particular pattern in one terabyte of data
M=15000,R=1
Rate increases as more machines are assigned
Reach peak at over 30 GB/s.
Rate drops as map tasks finish
Takes 150 seconds including a minute of startup to complete the task
And it gives great efficiency

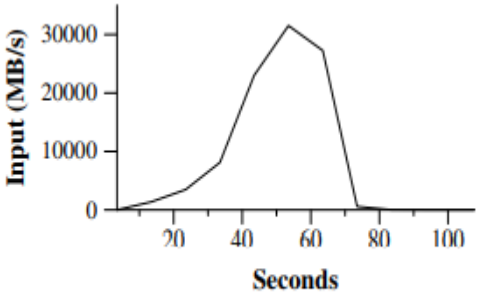
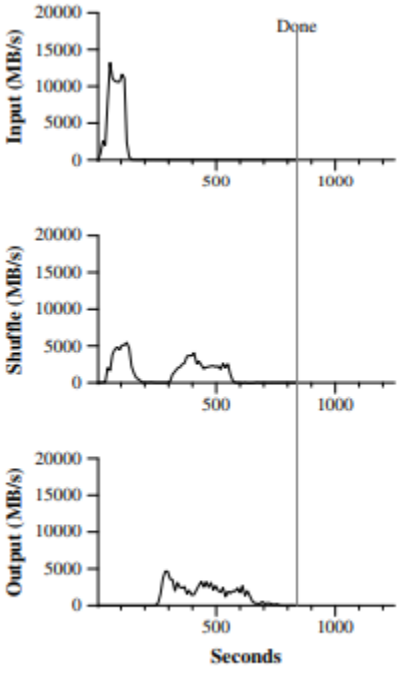


Figure 2: Data transfer rate over time

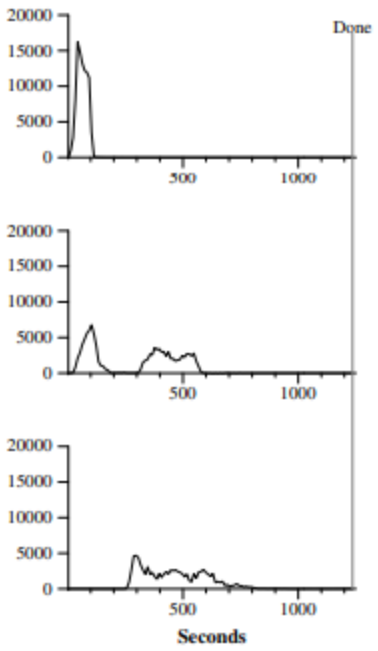
In sort program
Sorts one terabyte of data
M=15000, R=4000
Input rate peaks at 13GB/s
Shuffle is done about 600 seconds into computation
Output finishes about 850 seconds into computation
This program also gives great efficiency



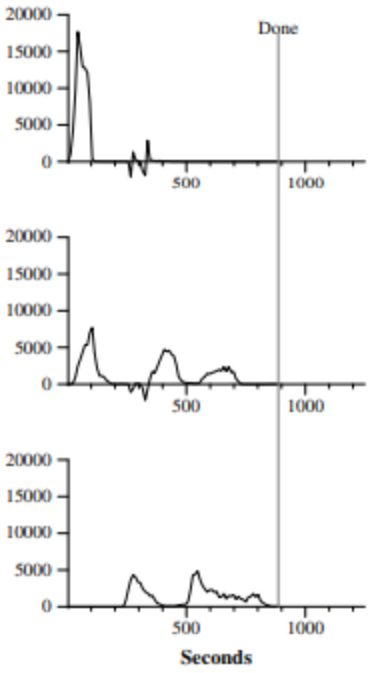
(a) Normal execution

In the case of no backup tasks
  A long tail where hardly any write Activity occurs
 Entire computation takes 1283 seconds vs 850 seconds
 Efficiently get rid of “strigger

Machine Failures
 Intentionally killed 200 out of 1746 worker processes
 Entire computation takes 933 seconds vs 850 seconds
 Quick recovery from failures!



(b) No backup tasks



(c) 200 tasks killed

Conclusion:

MapReduce programming model is easy to use
    Because it will hide all the horrible details!
Large variety of problems are expressible using MapReduce
MapReduce solves large computational problems efficiently
Sending data across the network is expensive
  Locality optimization allows data r/w from local disks

Currently:

Google Dumps MapReduce in Favor of New Hyper-Scale Analytics System