COMP5349 – Cloud Computing

Week 4: MapReduce Framework

Dr. Ying Zhou
School of Computer Science

Outline

- **Embarrassingly Parallel Workload**
- MapReduce Programming Model
- Hadoop MapReduce Framework

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Administrative

- We are moving to online teaching from this week
  - The lectures will be recorded and put online before the first lab
    - No real time lecture zooming
    - Please ask question on Ed
  - Labs will run at the scheduled time using zoom
- Essential software
  - Zoom used in lab
  - Git repository to release code
  - Ed for discussion and code challenge
    - Please make sure you have Ed access, if not contact the course coordinator: ying.zhou@sydney.edu.au
  - Web browser and shell window to connect to cloud instance

Administrative

- Important change on assessment schedule
  - We have the approval to move the code challenge from week 6 to week 7
  - There will be a practice on week 5 lab
    - For you to get familiar with the environment
    - For us to check if everything is set up properly
  - We want to give everyone an extra week to adjust to the online environment.

Last Week

- Last week we cover container technology
- Container is described as light weight virtualization
- It uses OS techniques such as namespace and control groups to provide isolation and resource allocation.
- Containers share the kernel with host OS
- Container technology can be used in different scenario
  - Docker used it as a way to package application for easy deployment
- Security might be compromised for other features

MapReduce Motivation

- Want to process lots of data (> 1 TB)
  - Eg. Build inverted word-document index for the whole web
- Want to parallelize across hundreds/thousands of CPUs
- Want to make this easy
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

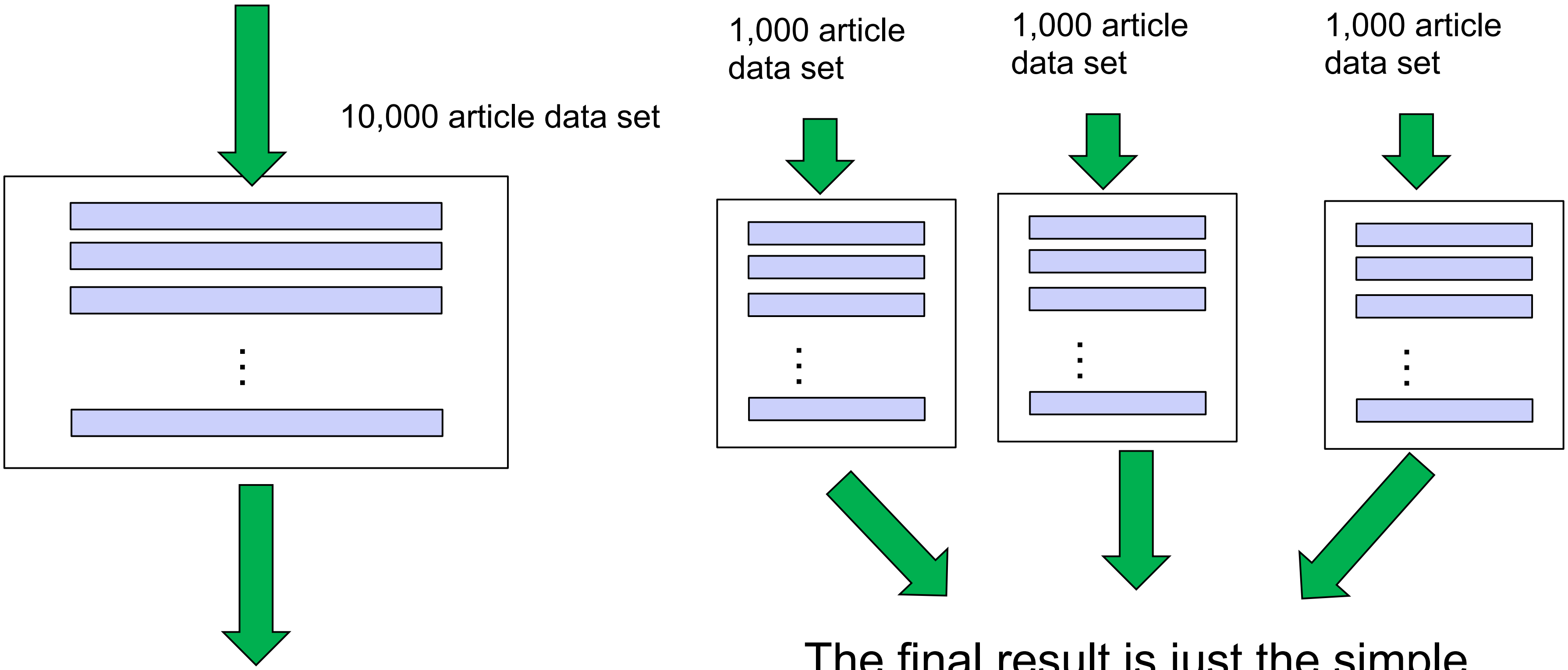
Embarrassingly Parallel Workload

- “In parallel computing, an **embarrassingly parallel** workload or problem (also called *perfectly parallel* or *pleasingly parallel*) is one where *little or no effort* is needed to separate the problem into a number of parallel tasks. This is often the case where there *is* little or no dependency or need for communication between those parallel tasks, or for results between them”

https://en.wikipedia.org/wiki/Embarrassingly_parallel

- Examples:
  - Looking for occurrence of a certain pattern in 10000 articles
  - Sequential processing: go through each article to find the pattern and print out each matching
  - Parallel processing: suppose we have 10 machines, put 1000 articles in each machine, in each machine, do the same sequential processing

Embarrassingly Parallel Workload



10,000 article data set

1,000 article data set

1,000 article data set

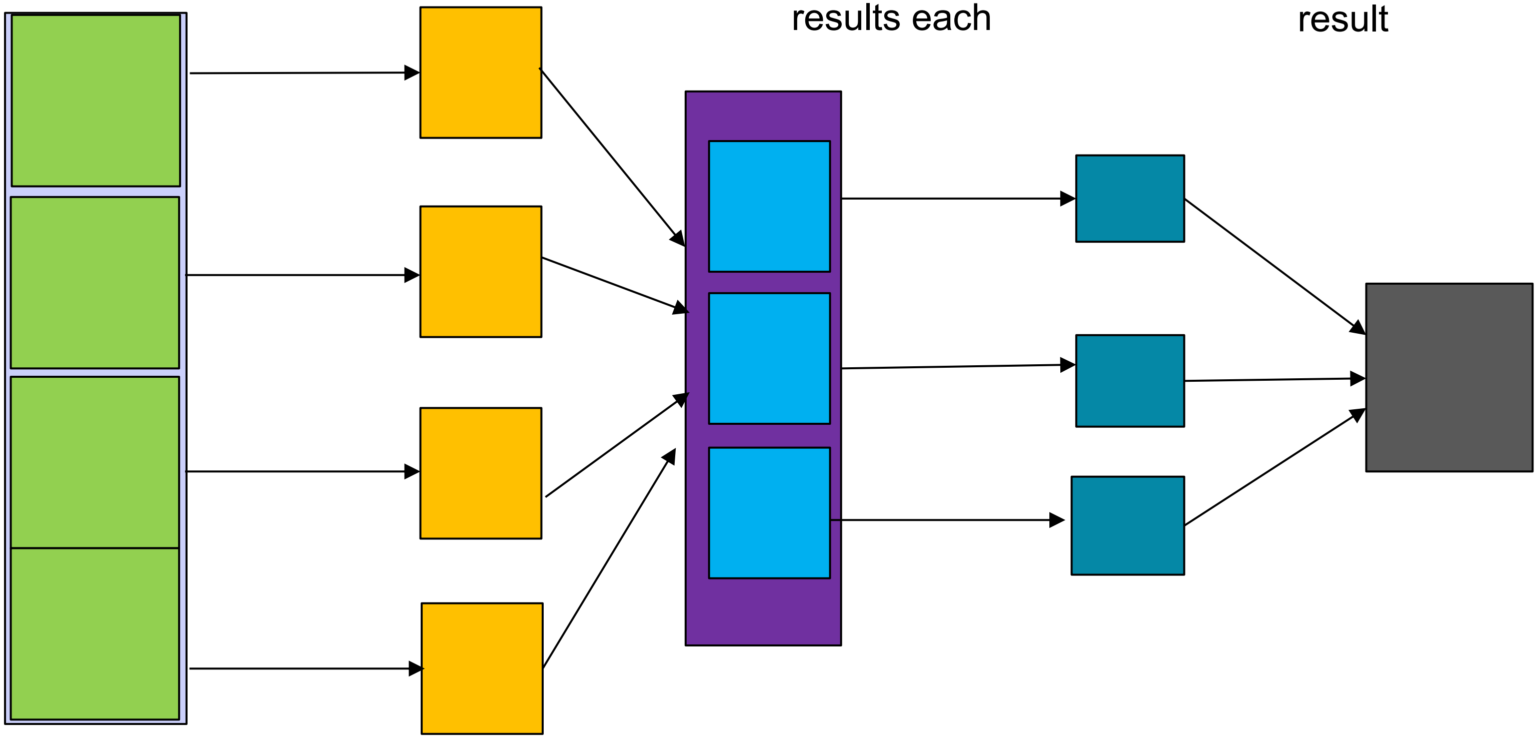
1,000 article data set

Near linear speedup may be achieved.

Divide and conquer strategy

The final result is just the simple aggregation of the partial result from each machine

A typical parallel workload execution



Run some processing on the subset parallel in many nodes, each produces a partial result

Further processing can be done on those subset in parallel to produce some partial results each

Aggregate partial results to get final result

Large input

Breakdown in to many smaller subsets

Aggregate partial results to get some intermediate results

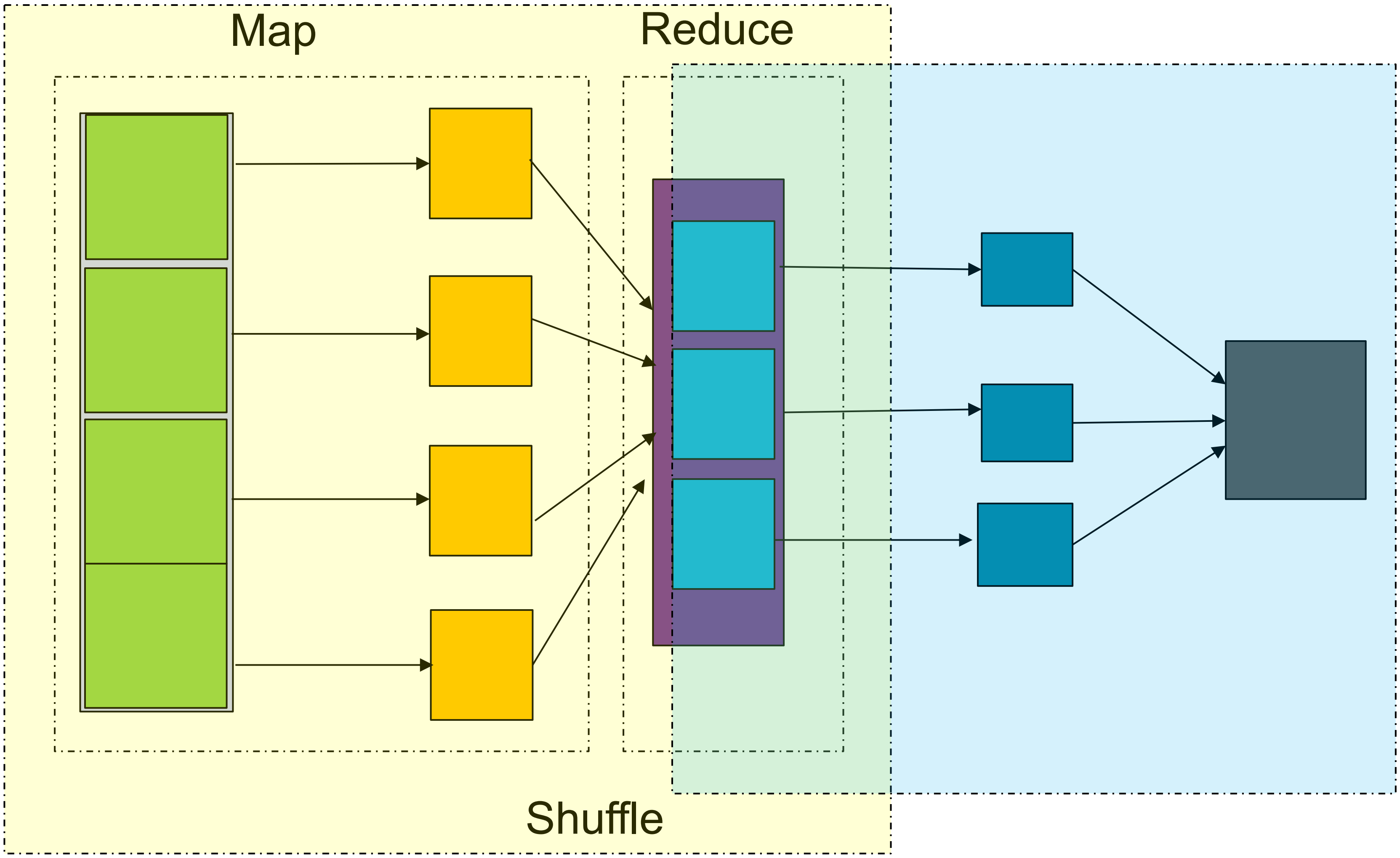
The intermediate results may be broken down into a few subsets

Map, Reduce and Job

Job

Reduce is the phase to aggregate those partial results, we aim to parallel this part as well

Job



Map is the phase aim to produce partial results from each subset in dependently and in parallel

Outline

- Embarrassingly Parallel Workload

- MapReduce Programming Model
  - Functional Programming Basics
  - MapReduce Programming model
    - The key-value concept
    - Word Count Example
    - Execution Overview
    - Combiner Function

- Hadoop MapReduce Framework

Functional Programming

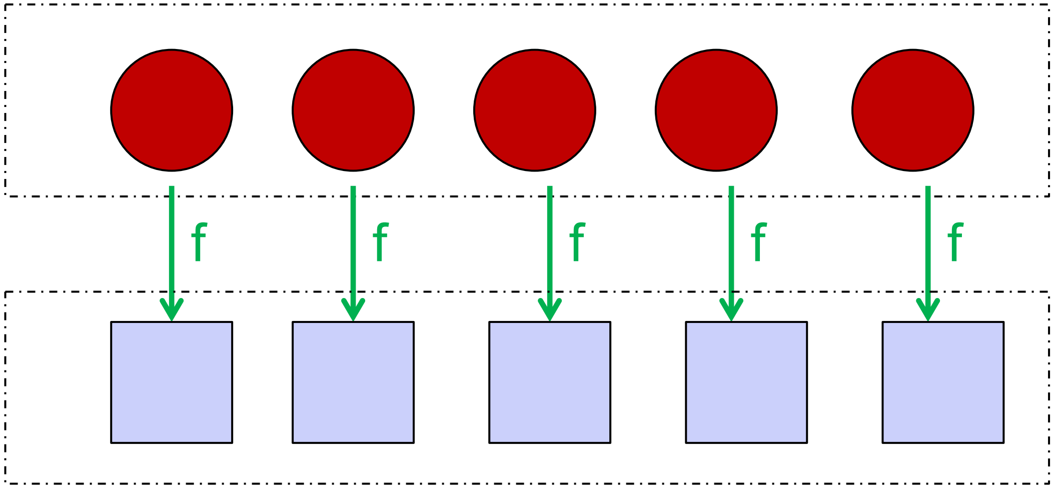
- Most of the languages we learn and use (Java, C#, C, C++,…) belong to **imperative programming**, which is based on von Neumann architecture
  - Emphasising on telling computer what to do in steps
- "**functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions" [-- wikipedia]
  - Lisp, Erlang, F#, Scala etc,

Features of FP

- Functional operations do not modify data structures, they just create new ones
  - No “side effects”
  - Easier to verify, optimize, and parallelize programs
- Higher-order functions, which takes another functions as parameters provide an easy way to handle collection
  - Traditional imperative programming usually relies on a loop structure, visitor pattern, etc. to traverse a collection
  - Some script language, javascript, python, ruby simulate higher-order functions using the closure concept
- Two useful higher-order functions that inspire MapReduce framework are:
  - **map** and **fold**, or **reduce**

Higher-order function-- map

- The map function applies a given function to all elements in a list and returns the result as a new list
  - map f originalList



The original list with five elements

Apply function f to all element

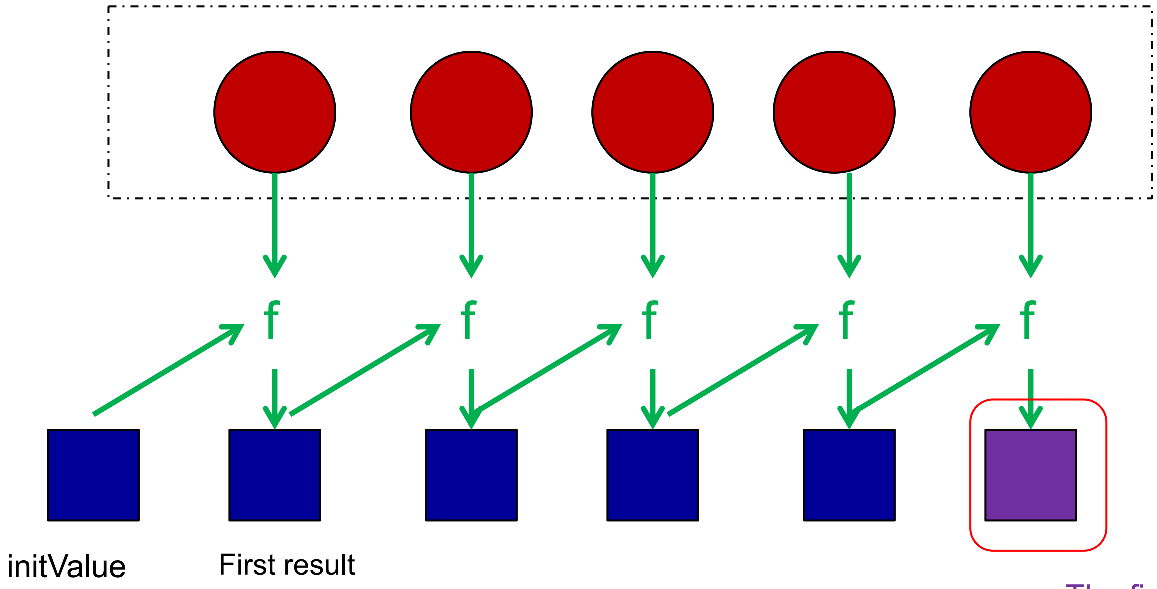
Obtain a new list of five elements

We can easily parallel the execution of function f

The diagram is based on MapReduce lecture slides used in CSE 490H in University of Washington

Higher order function: fold/reduce

- The fold function apply a given function together with an initial value iteratively on list elements; it returns the value obtained from applying the function and initial value to the last element.
  - fold f initValue originalList



The original list with five elements

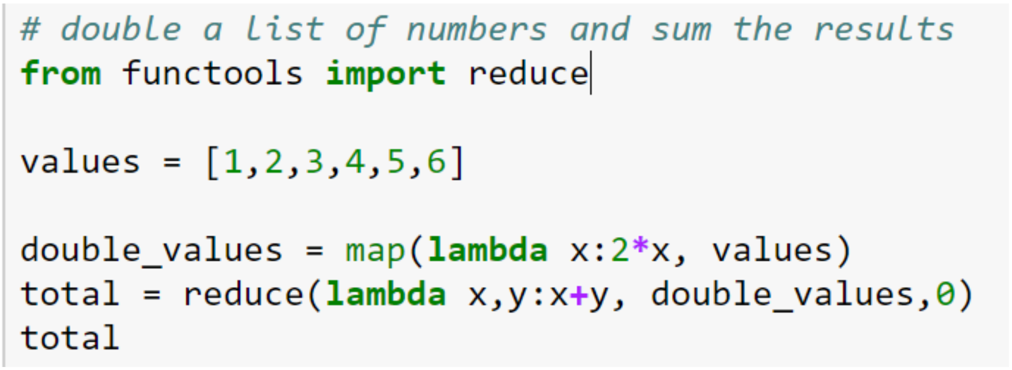
The final result!

initValue for second element

The diagram is based on MapReduce lecture slides used in CSE 490H in University of Washington

Python MapReduce Example

- Simple Python Example



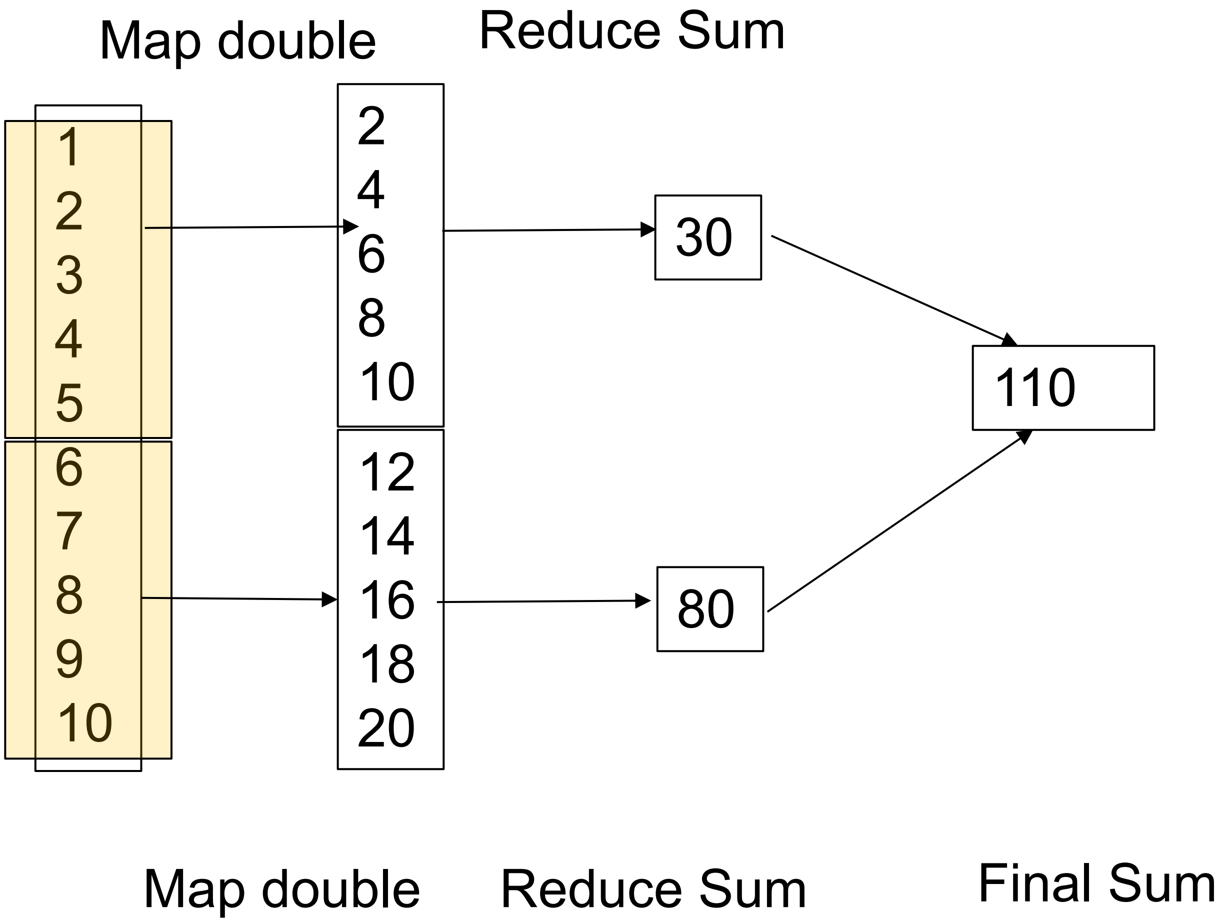
```
# double a list of numbers and sum the results
from functools import reduce

values = [1,2,3,4,5,6]

double_values = map(lambda x:2*x, values)
total = reduce(lambda x,y:x+y, double_values,0)
total
```

- Note this is an example only, using sum() is always preferred than reduce for simple operations

Reduce Function Parallel Execution

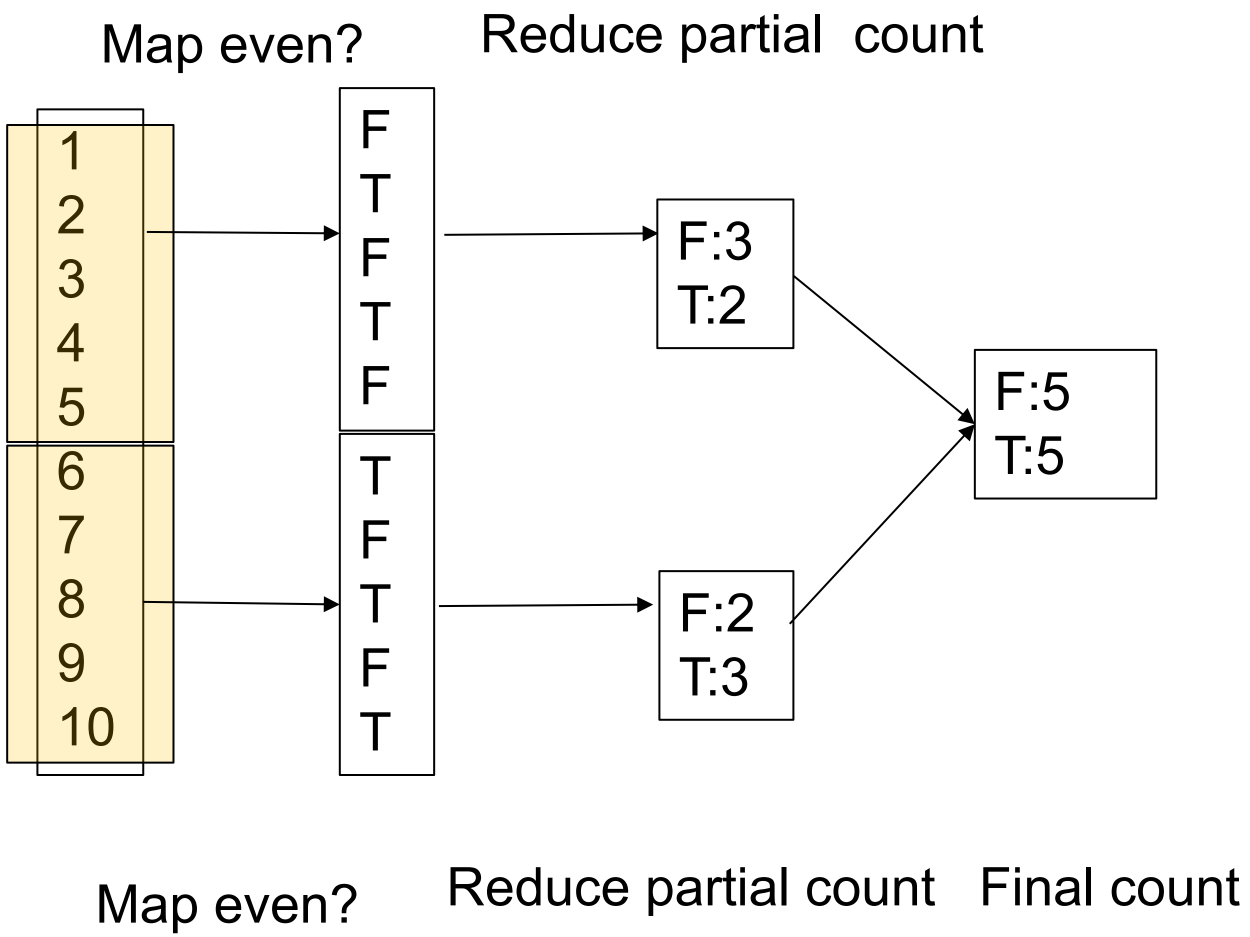
- By default, the **reduce** function is not parallelizable because all elements in a list needs to be visited one by one to produce the final results.
- But in many cases, it is possible to produce the final reduce results by aggregating partial results from a subset
  - E.g. to compute the sum of 10,000 numbers, if we have 10 machines, we can ask each machine to compute the partial sum of 1000 numbers and get the final sum by adding up the partial sum
- The question is, how do we divide the input of reduce into subset, especially if it is the output of a previous map function?

The double-then-sum workload



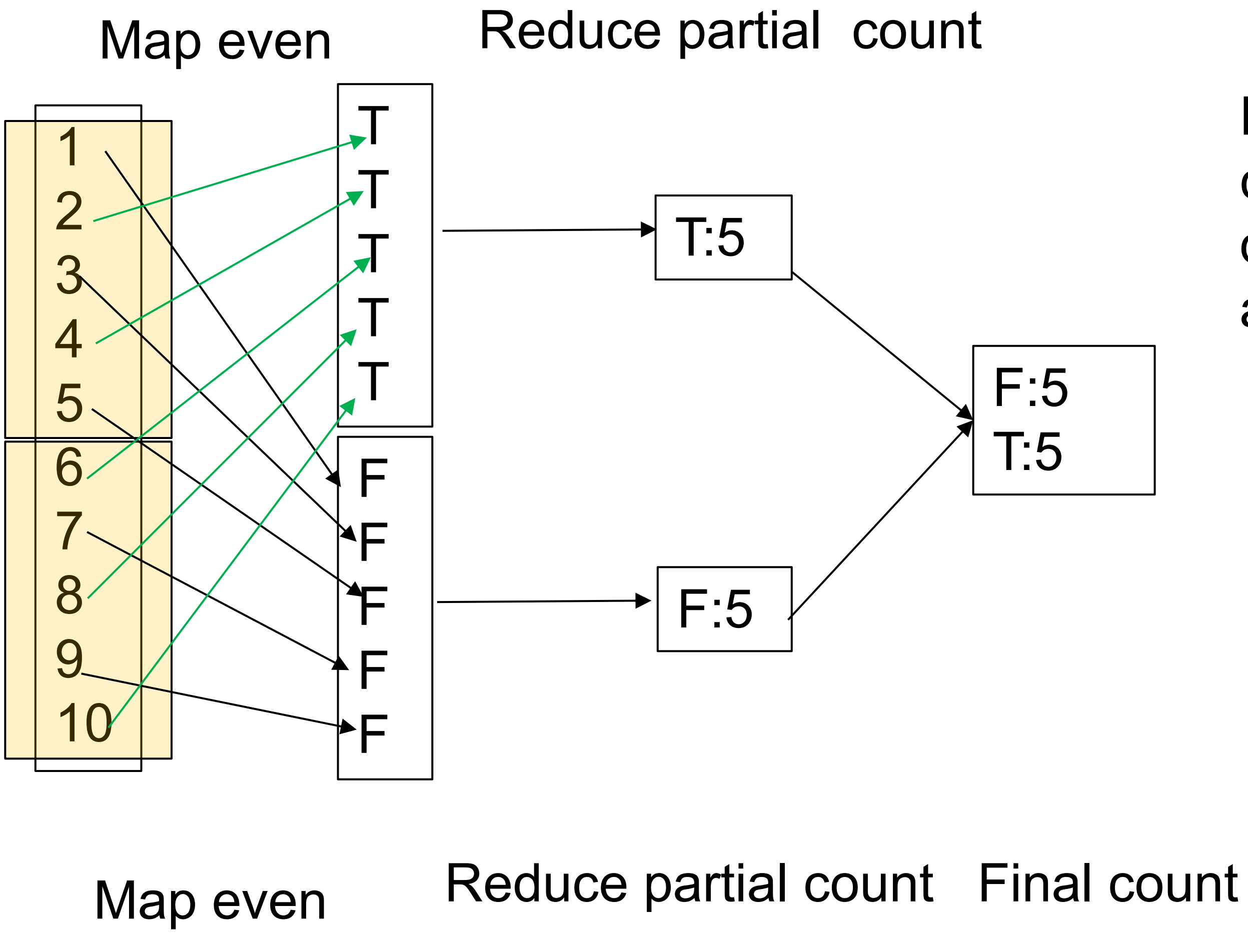
Can we just run one **reduce** on the partial result of one **map**?

Count odd or even number workload



Things would be much easier if we can put all Fs in one group and all Ts in one group

Count odd or even number workload



Reduce computation does not involve any comparison

Organizing map output and presenting them in desirable format can be done by framework

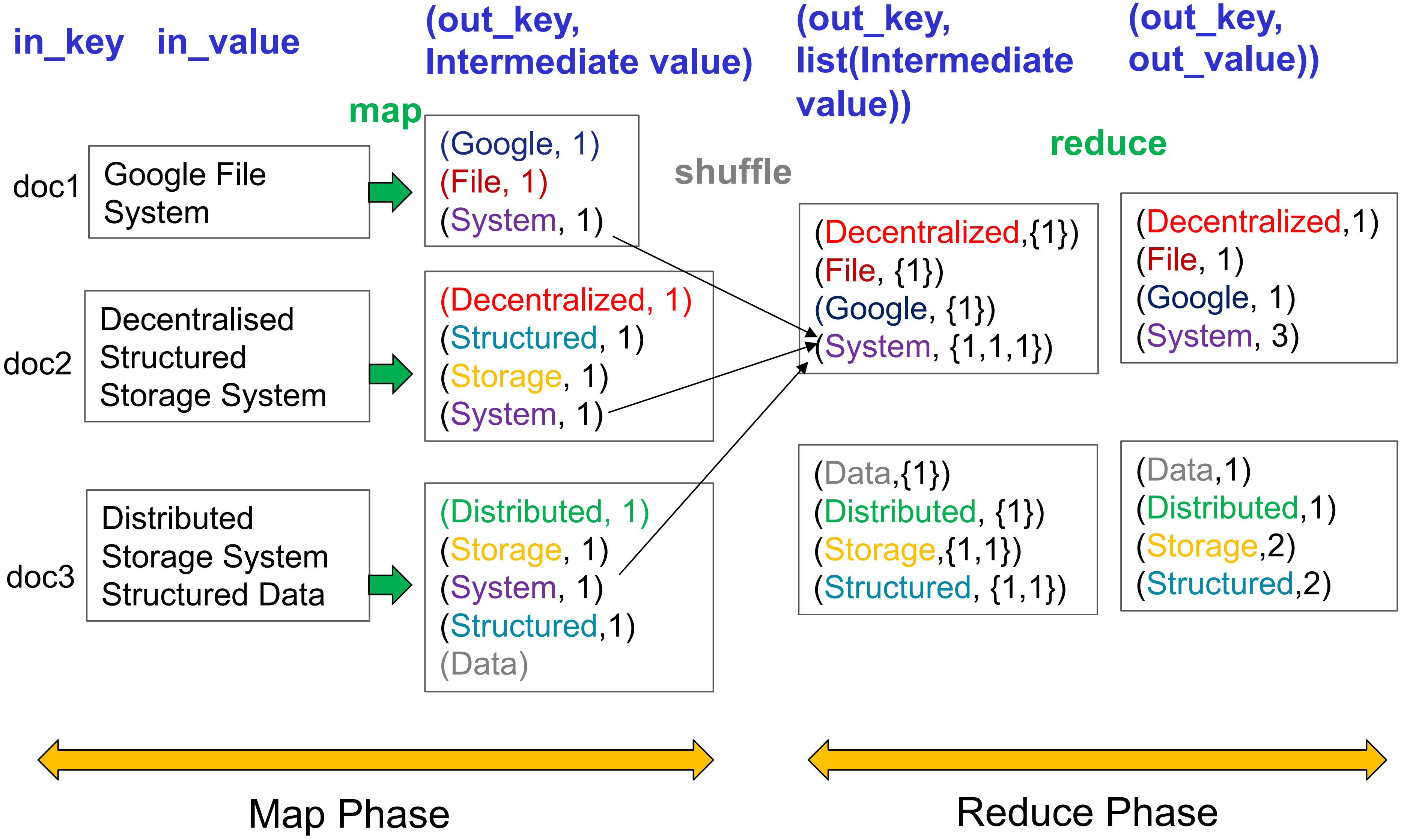
Programming Model

- Inspired by **map** and **fold** in FP
- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
  - map (**in_key**, in_value) -> list(**out_key**, intermediate_value)
    - Processes input key/value pair
    - Produces a list of intermediate pairs
  - reduce (**out_key**, list (intermediate_value)) -> list(**out_key**,out_value)
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values for a given key (usually just one)
- The *key* is used for dividing and grouping reduce input more effectively

Example: Count word occurrences

```
map(String in_key, String in_value):
    // in_key: document name
    // in_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");
                       key value
reduce(String out_key, Iterator intermediate_values):
   // out_key: a word
   // intermediate_values: a list of counts associated with that
   //word
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(out_key, AsString(result));
```

Word Count Example



in_key in_value

(out_key, Intermediate value)

(out_key, list(Intermediate value))

(out_key, out_value))

map

shuffle

reduce

doc1 Google File System

(Google, 1)
(File, 1)
(System, 1)

doc2 Decentralised Structured Storage System

(Decentralized, 1)
(Structured, 1)
(Storage, 1)
(System, 1)

doc3 Distributed Storage System Structured Data

(Distributed, 1)
(Storage, 1)
(System, 1)
(Structured,1)
(Data)

(Decentralized,{1})
(File, {1})
(Google, {1})
(System, {1,1,1})

(Data,{1})
(Distributed, {1})
(Storage,{1,1})
(Structured, {1,1})

(Decentralized,1)
(File, 1)
(Google, 1)
(System, 3)

(Data,1)
(Distributed,1)
(Storage,2)
(Structured,2)

Map Phase

Reduce Phase

What can framework provide?

- The developer only needs to write the two functions:

```
map(String in_key, String in_value):
    for each word w in input_value:
   EmitIntermediate(w, "1");
```

```
reduce(String out_key, Iterator intermediate_values):
    int result = 0;
    for each v in intermediate_values:
   result += ParseInt(v);
    Emit(out_key, AsString(result));
```

- The framework would manage the parallel execution of these functions
  - Split input data into small partitions
  - Run map function on small partitions on available machines in parallel
  - Re-organize the map output to prepare input for reduce function
  - Run reduce functions on its input on available machines in parallel
  - Fault tolerance and other features

MapReduce Execution Overview

Data Locality

Split 0 and 1 locate in the same worker machine, two map tasks are assigned to this worker. Input data is read locally!

Master Operation

Master stores the state of each map and reduce tasks

It receives intermediate file locations and push them to reduce tasks incrementally

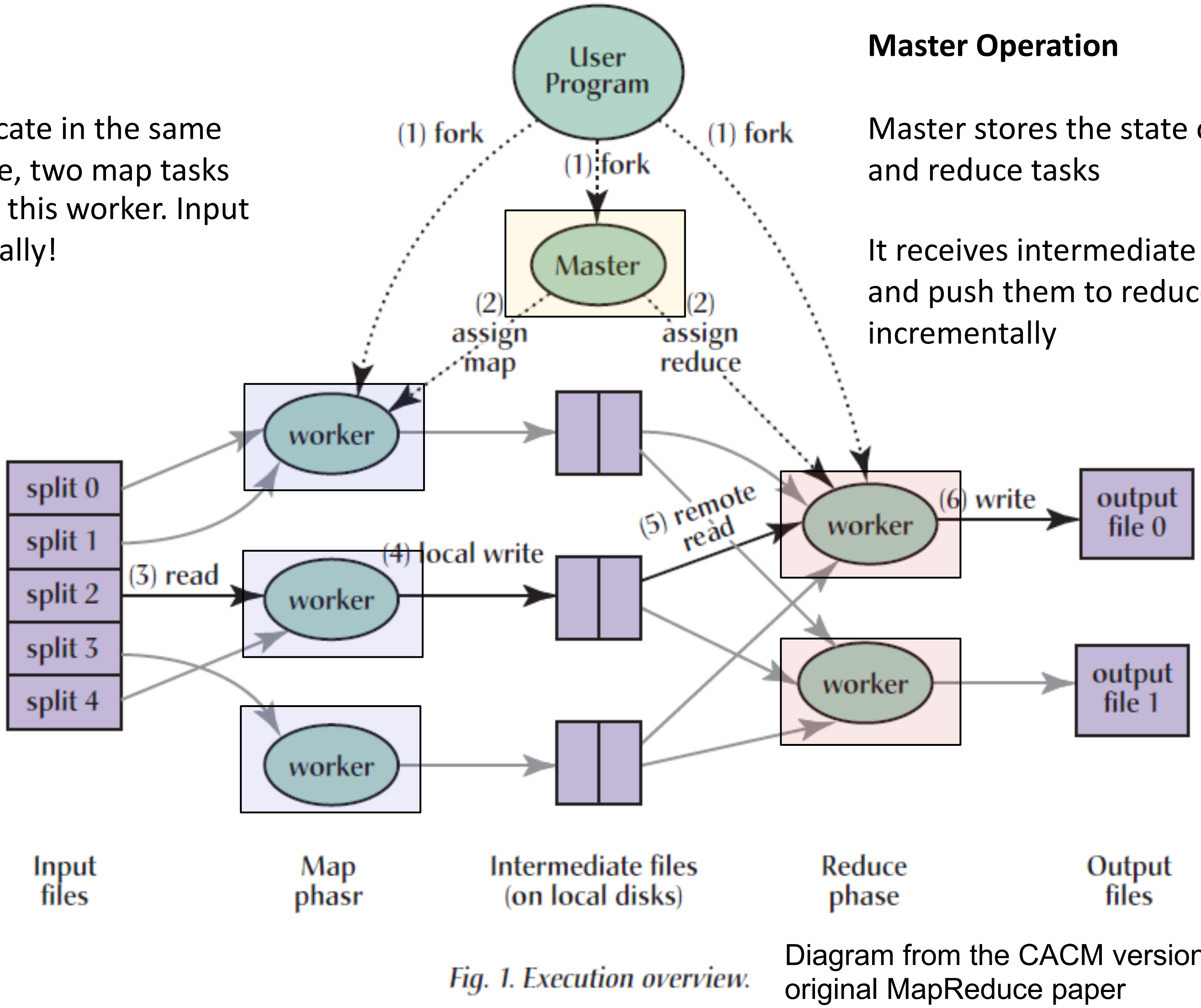


Fig. 1. Execution overview.

Worker

Workers are the machines that will execute either map or reduce function defined by developer

Diagram from the CACM version of the original MapReduce paper

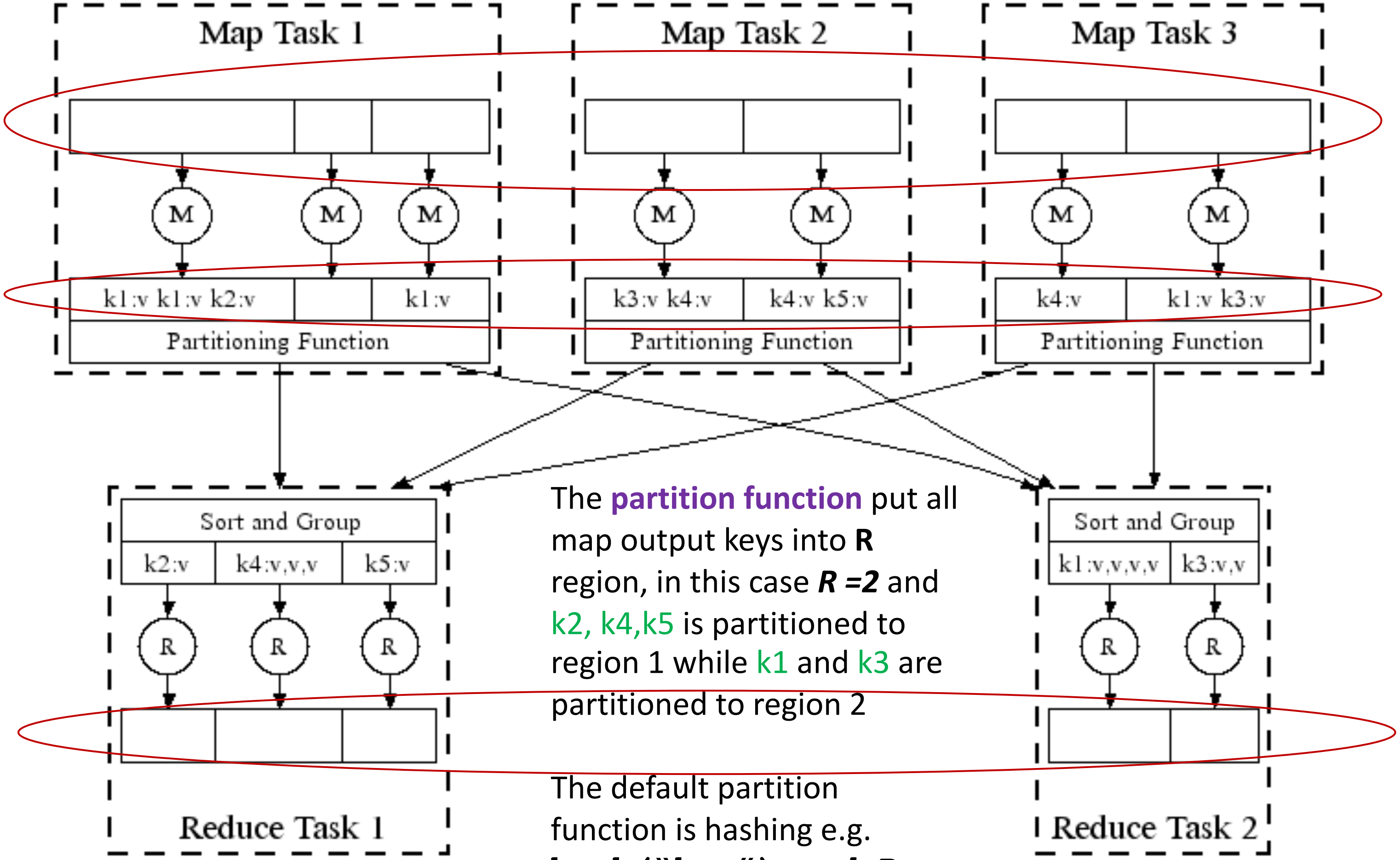
Parallel Execution

Input data stores in GFS

Intermediate results stores in Local disk

The shuffle process uses RPC read

Final result stores in GFS



The **partition function** put all map output keys into **R** region, in this case ***R* =2** and k2, k4,k5 is partitioned to region 1 while k1 and k3 are partitioned to region 2

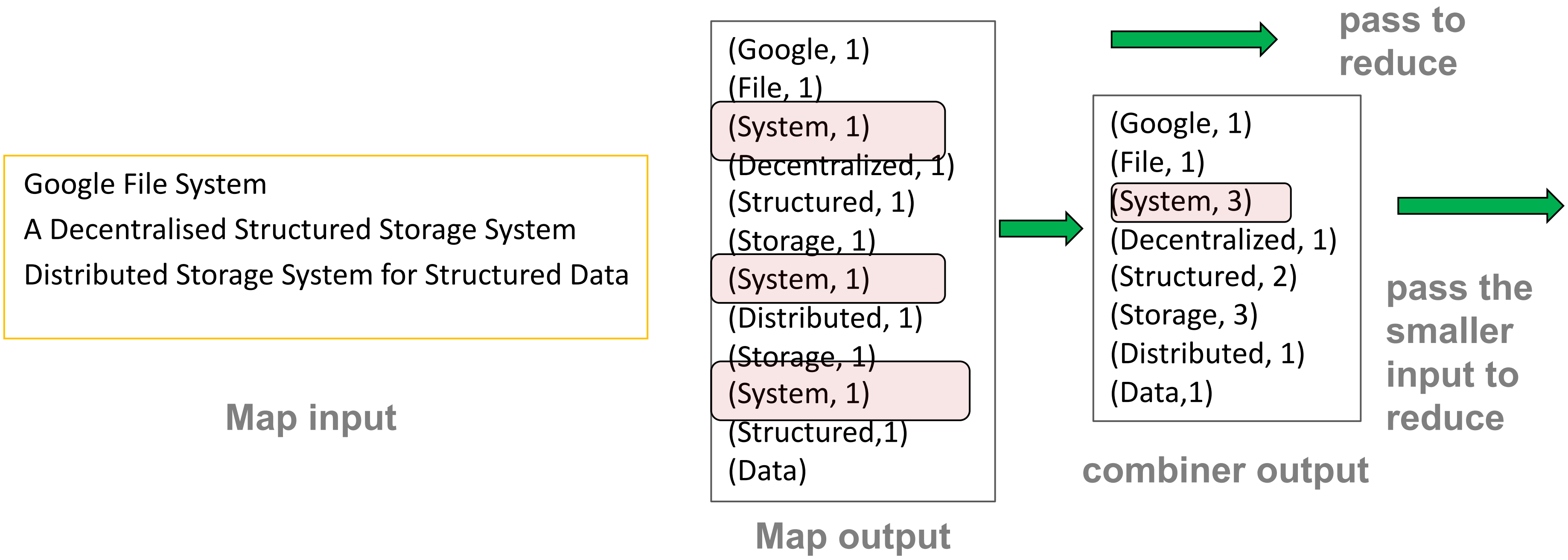
The default partition function is hashing e.g.
`hash("key") mod R`

Diagram from the original slides by Jeff Dean and Sanjay Ghemawat

The Combiner Function

- Combiner is an optimization mechanism to minimize the data transferred between the map and reduce tasks
- Combiner function runs on the map side to merge some of the map intermediate result
  - It is like running a reduce function locally on each map task
- The output of the combiner function becomes the input of the reduce function

The Combiner Function



Google File System
A Decentralised Structured Storage System
Distributed Storage System for Structured Data

Map input

(Google, 1)
(File, 1)
(System, 1)
(Decentralized, 1)
(Structured, 1)
(Storage, 1)
(System, 1)
(Distributed, 1)
(Storage, 1)
(System, 1)
(Structured,1)
(Data)

Map output

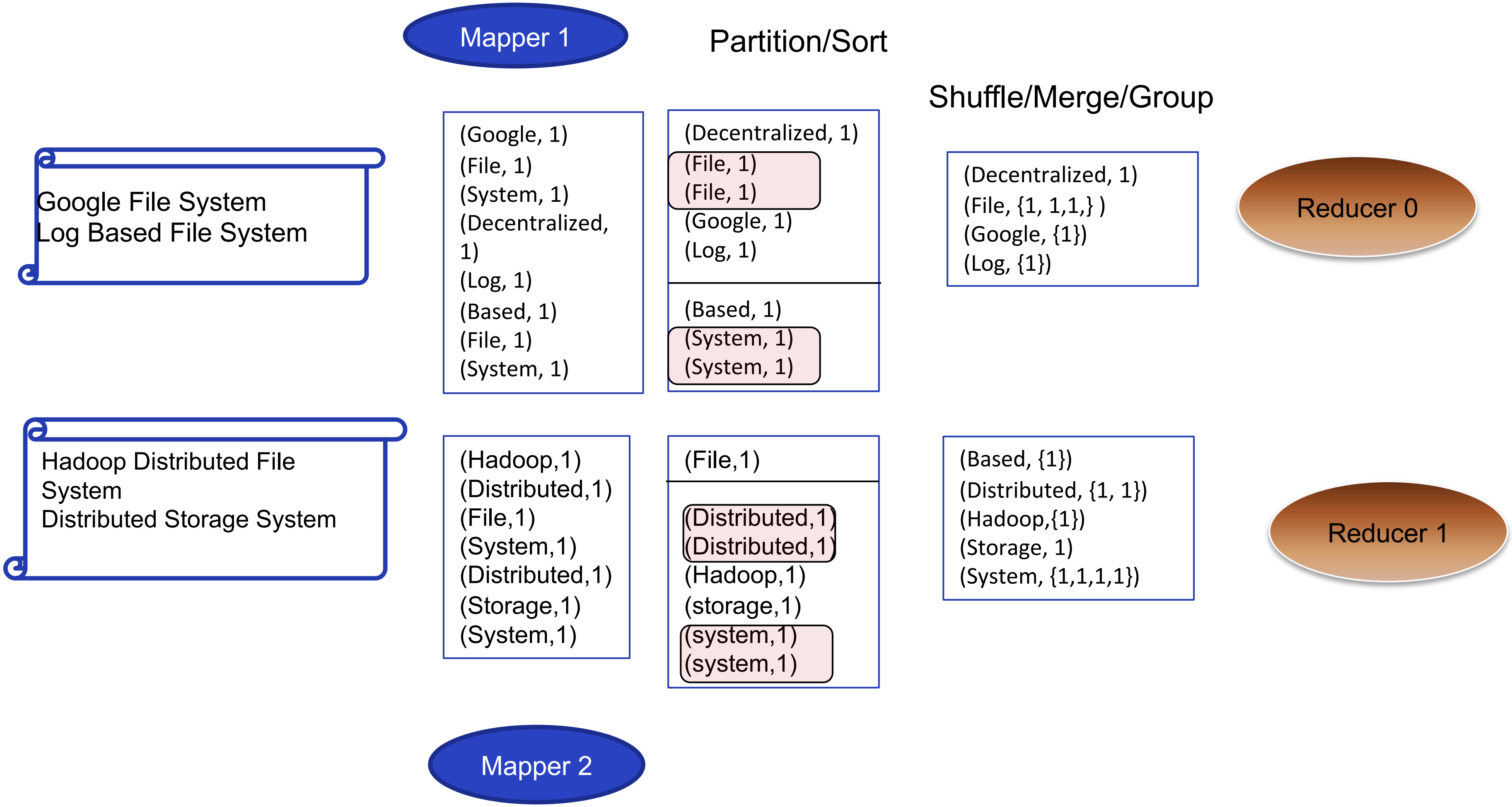
(Google, 1)
(File, 1)
(System, 3)
(Decentralized, 1)
(Structured, 2)
(Storage, 3)
(Distributed, 1)
(Data,1)

combiner output

pass to reduce

pass the smaller input to reduce

Word Count Without Combiner



Google File System
Log Based File System

Hadoop Distributed File System
Distributed Storage System

Mapper 1

(Google, 1)
(File, 1)
(System, 1)
(Decentralized, 1)
(Log, 1)
(Based, 1)
(File, 1)
(System, 1)

Mapper 2

(Hadoop,1)
(Distributed,1)
(File,1)
(System,1)
(Distributed,1)
(Storage,1)
(System,1)

Partition/Sort

(Decentralized, 1)
(File, 1)
(File, 1)
(Google, 1)
(Log, 1)

(Based, 1)
(System, 1)
(System, 1)

(File,1)

(Distributed,1)
(Distributed,1)
(Hadoop,1)
(storage,1)
(system,1)
(system,1)

Shuffle/Merge/Group

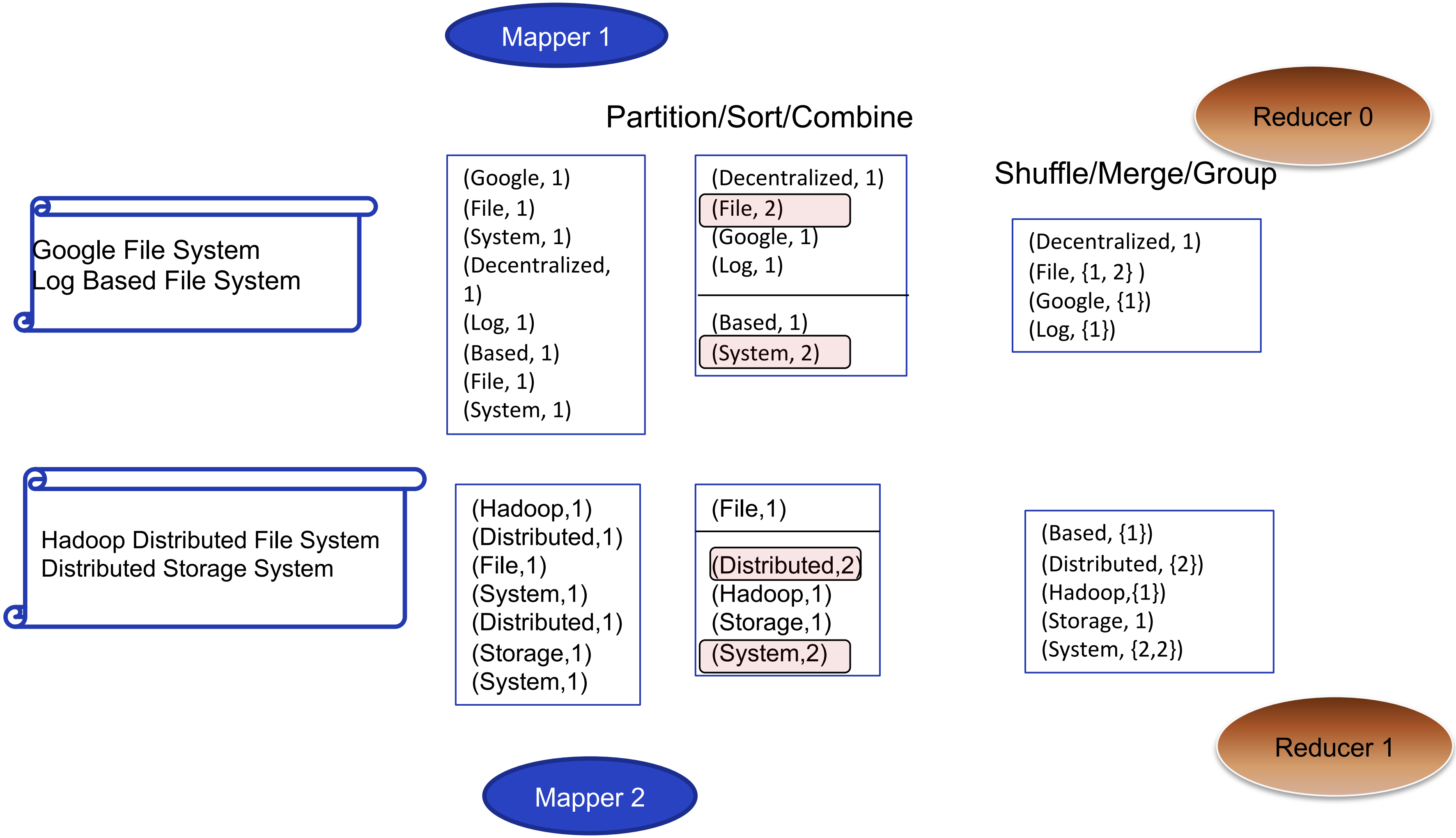
(Decentralized, 1)
(File, {1, 1,1,})
(Google, {1})
(Log, {1})

(Based, {1})
(Distributed, {1, 1})
(Hadoop,{1})
(Storage, 1)
(System, {1,1,1,1})

Reducer 0

Reducer 1

Word Count With Combiner



Mapper 1

Partition/Sort/Combine

Reducer 0

Shuffle/Merge/Group

Google File System
Log Based File System

(Google, 1)
(File, 1)
(System, 1)
(Decentralized, 1)
(Log, 1)
(Based, 1)
(File, 1)
(System, 1)

(Decentralized, 1)
(File, 2)
(Google, 1)
(Log, 1)

(Based, 1)
(System, 2)

(Decentralized, 1)
(File, {1, 2})
(Google, {1})
(Log, {1})

Hadoop Distributed File System
Distributed Storage System

(Hadoop,1)
(Distributed,1)
(File,1)
(System,1)
(Distributed,1)
(Storage,1)
(System,1)

(File,1)

(Distributed,2)
(Hadoop,1)
(Storage,1)
(System,2)

(Based, {1})
(Distributed, {2})
(Hadoop,{1})
(Storage, 1)
(System, {2,2})

Reducer 1

Mapper 2

Outline

- Embarrassingly Parallel Workload
- MapReduce Programming Model
- Hadoop MapReduce Framework
  - Basic Components
  - Java API
  - Hadoop Streaming

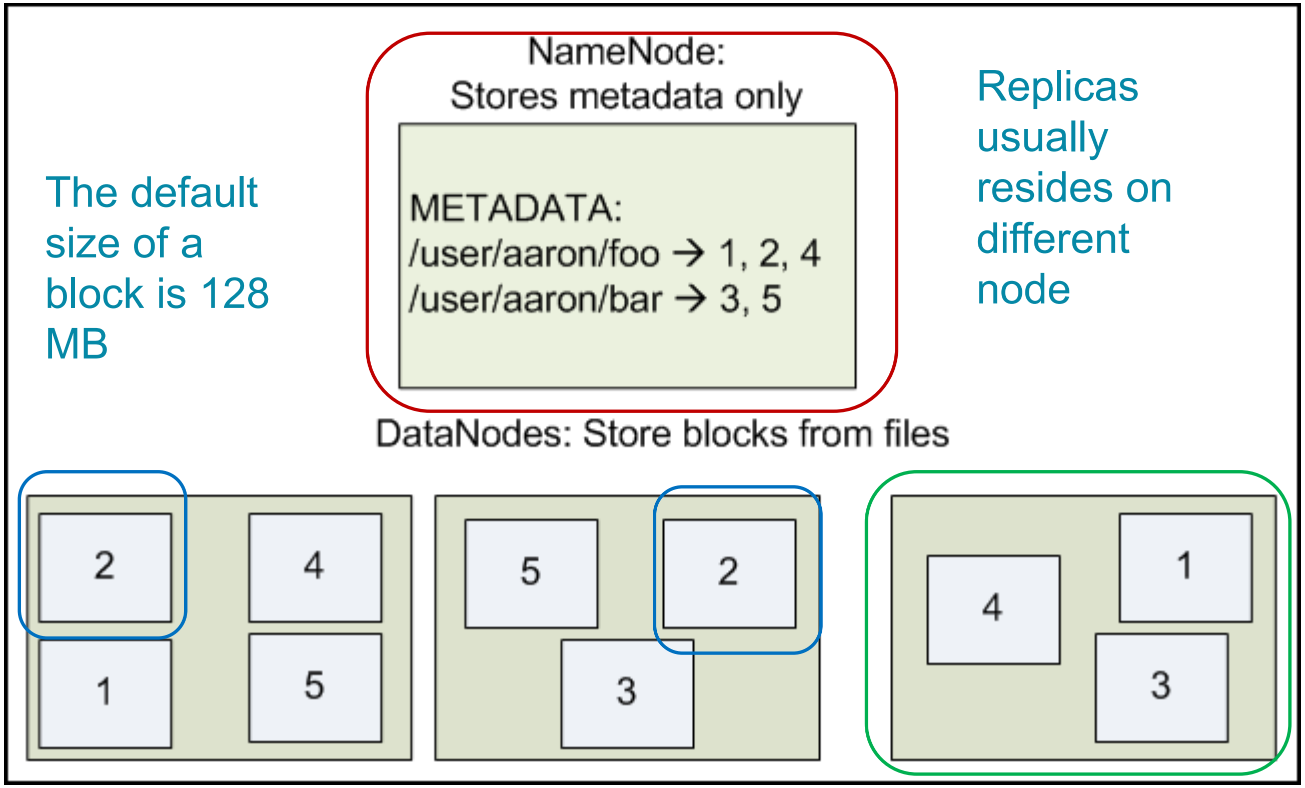
Hadoop Basics

- Hadoop is the open source implementation of Google’s MapReduce Framework
  - It was first released in 2006, containing **HDFS** and **MapReduce** modelled after Google’s GFS and MapReduce
  - Hadoop has grown into a large ecosystem with many supporting components
  - A basic Hadoop installation consists of **HDFS**, **MapReduce** and **YARN**
- Hadoop is written in Java and has native support for Java Applications
- Support for Python application is implemented through Hadoop Streaming
- Hadoop can run on a single machine
  - Standalone mode
  - **Pseudo-distributed mode**
- Production environment runs **full-distributed cluster mode**.

HDFS

- HDFS is a distributed file system modelled after an distributed file system used in Google (GFS)
- It consists of a cluster of nodes
  - One has special role, and is called name node
  - All others are called data nodes, they are responsible for storing files
  - HDFS stores files on a designated location on host file system
- HDFS is designed to store huge files, e.g. files in GB or TB size
  - Large files are divided into smaller blocks of a configured size, e.g. 128M
  - Those blocks are stored in different data nodes and are also replicated.
  - Name node keeps the meta data such as file A consists of block a,b,c, … h, and the are stored in node x, y, z respectively.

HDFS Architecture



The default size of a block is 128 MB

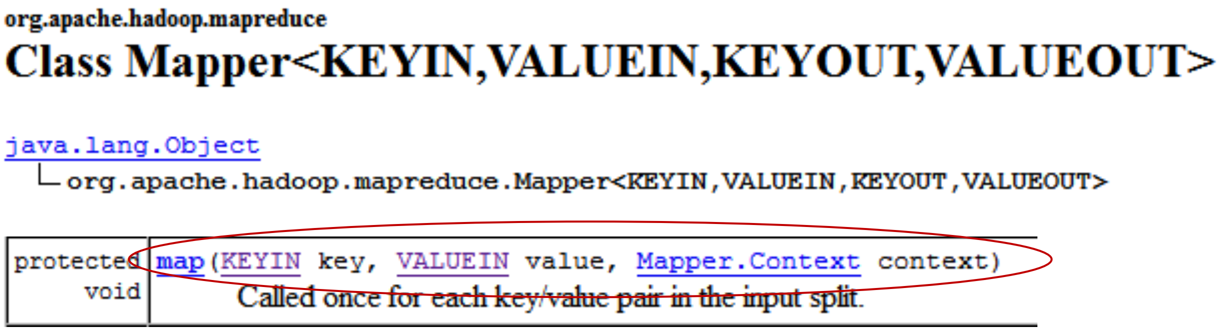
NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

Replicas usually resides on different node

DataNodes: Store blocks from files

Hadoop MapReduce Java API



org.apache.hadoop.mapreduce

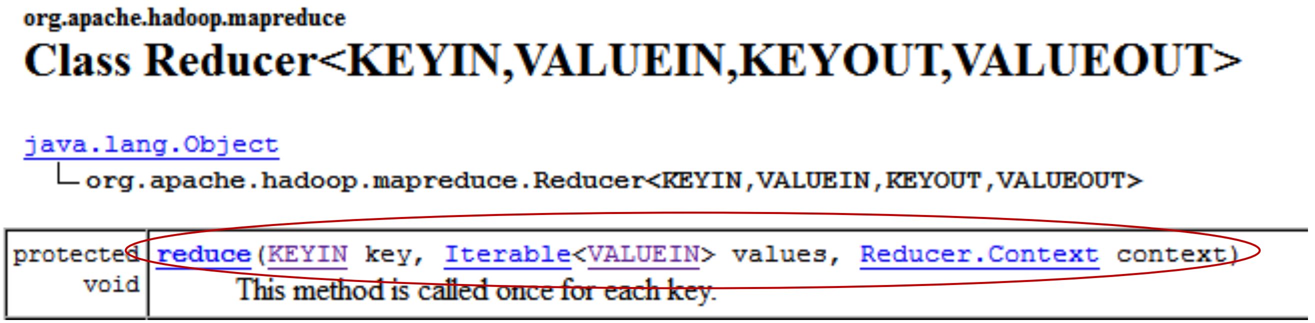
Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

```
java.lang.Object
  └org.apache.hadoop.mapreduce.Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

protected void	map(KEYIN key, VALUEIN value, Mapper.Context context) Called once for each key/value pair in the input split.

the map function

Map task



org.apache.hadoop.mapreduce

Class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

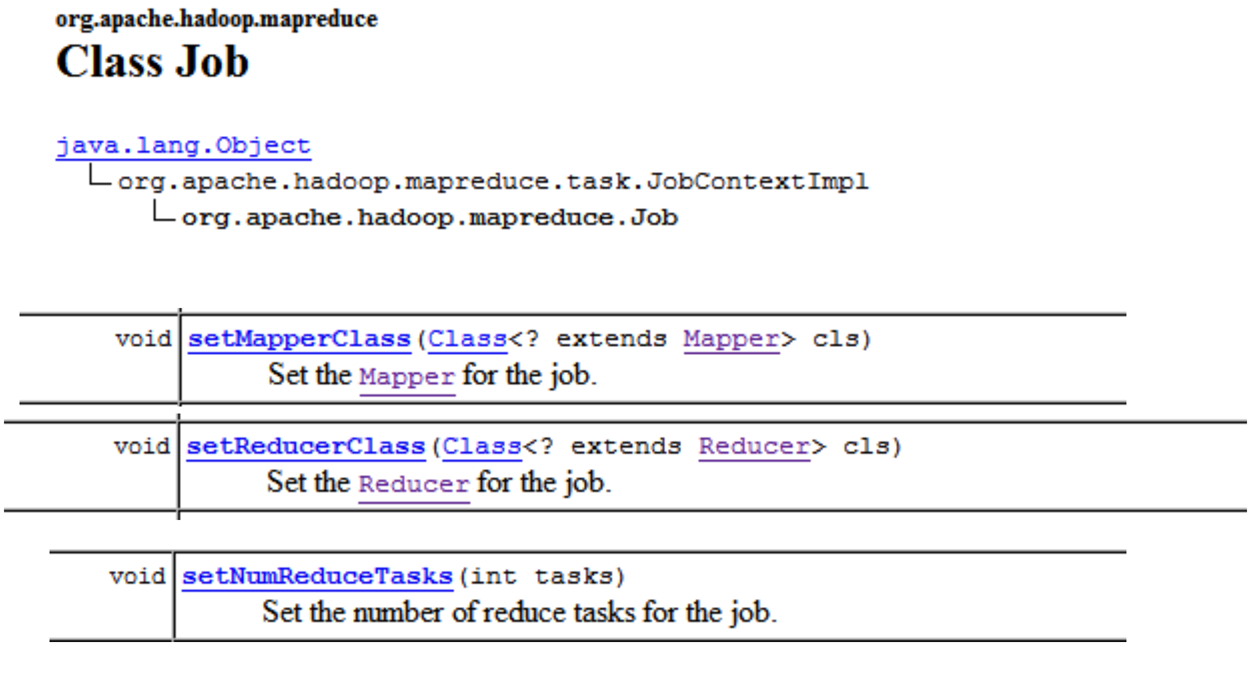
```
java.lang.Object
  └org.apache.hadoop.mapreduce.Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

protected void	reduce(KEYIN key, Iterable<VALUEIN> values, Reducer.Context context) This method is called once for each key.

the reduce function

Reduce task

Hadoop MapReduce Java API



org.apache.hadoop.mapreduce

Class Job

```
java.lang.Object
  └org.apache.hadoop.mapreduce.task.JobContextImpl
      └org.apache.hadoop.mapreduce.Job
```

void	`setMapperClass(Class<? extends Mapper> cls)` Set the `Mapper` for the job.
void	`setReducerClass(Class<? extends Reducer> cls)` Set the `Reducer` for the job.
void	`setNumReduceTasks(int tasks)` Set the number of reduce tasks for the job.

The MapReduce Job

……

Java API: The Mapper

KeyIn ValueIn KeyOut ValueOut

```
public static class TagMapper
  extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable ONE = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {

        String[] wordArray = value.toString().split(" ");

        for(String term: wordArray) {
               word.set(term);
               context.write(word, ONE);
        }

  }
}
```

Emit Intermediate result

Google File System

Decentralised Structured Storage System

Distributed Storage System for Structured Data

….

Each line of the input file is feed into the map function as value

Java API: The Reducer

```
public static class IntSumReducer
extends Reducer<Text,IntWritable,Text,IntWritable> {
   private IntWritable result = new IntWritable();

   public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
               sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
   }
}
```

(Google, {1})
(File, {1})
(System, {1,1,1})
(Decentralized,{1})
(Structured, {1,1})
(Storage,{1,1})
(Distributed, {1})
(Data,{1})

The (key, list of values) passed to each reduce function

Each run of the reduce function would write out the result of a particular word

Java API: The Driver

```
public class WordCount{
   public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
        args).getRemainingArgs();
        if (otherArgs.length != 2) {
                System.err.println("Usage: WordCount <in> <out>");
                System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setNumReduceTasks(2);
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TagMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        TextInputFormat.addInputPath(job, new Path(otherArgs[0]));
        TextOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Combiner does “reduce” on local map output

Communication Between Mappers and Reducers

If there is a combiner function, it runs after sort and before disk spilling

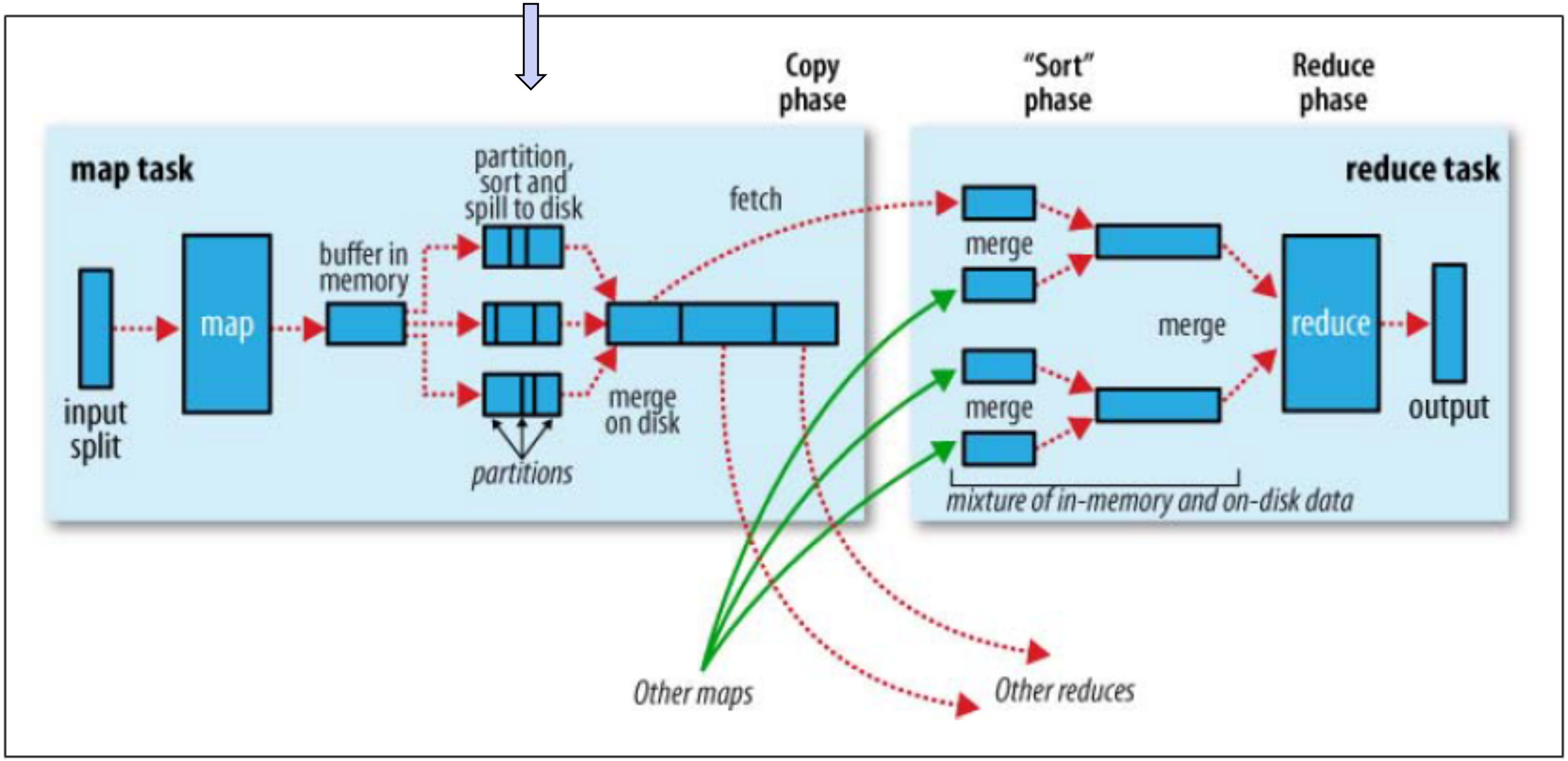


Figure 6-4. Shuffle and sort in MapReduce

Diagram from Tom White, Hadoop, the definitive Guide, O'reilly, 2009, page 163

Hadoop Streaming

- Hadoop streaming is a utility to enable writing MapReduce programs in languages other than Java
  - The utilility itself is packed as a jar file
  - We can specify any **executable** or **script** as mapper/combiner/reducer
- Eg.

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.9.0.jar \
   -input myInputDirs \
   -output myOutputDirs \
   -mapper  mapper.py \
    -reducer  reducer.py \
   -D mapreduce.job.reduces=2 \
   -D mapreduce.job.name= ‘word count’\
   -file mapper.py \
   -file reducer.py
```

How does streaming work

- The Hadoop framework assigns map and reduce tasks to slave nodes as usual
- Each **map** task
  - starts the executable or script in separate **process**,
  - converts the input key value pairs into lines and feed the lines to the stdin of the **process**
    - The **process** read the input line, does map work, and write output line by line to standard out
  - collects output from the stdout of the process and convert each line to key/value pair as map output

How does streaming work (cont’d)

- The framework does partition, shuffle and sort (but not grouping!) to prepare the reduce task input
  - The reduce task input is sorted map output
- Each reduce task
  - Starts the executable or script in separate **process**
  - converts the input key value pairs into lines and feed the lines to the stdin of the **process**
    - The **process** read the input line, does reduce work, and write output line by line to standard out
      - The input line has the format (key, value)
      - Script code needs to identify the boundary of keys (see example in lab code!)
  - collects output from the stdout of the process and convert each line to key/value pair as reduce output

MapReduce Program Design

- Deciding on the number of jobs
- For each job, design the **map** and **reduce** functions
- Each map and reduce **task** (mapper, reducer) will run those functions multiple times depends on the input size
- Combiner is just a **reduce** function running locally on the mapper side to aggregate results locally
- There is a chain of keys that are related
  - Map output key is the input key of reducer if there is no combiner
  - If there is a combiner, map output key is the input key of combiner, the output key of combiner becomes the input key of the reducer

References

- Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04,
- Yahoo! Hadoop Tutorial, Module 4: MapReduce http://developer.yahoo.com/hadoop/tutorial/module4.html
- Tom White, Hadoop, the definitive Guide, O'reilly, 2009
  - Library has online version of the latest edition (4th edition)