CS4225/CS5425 Big Data Systems for Data Science

MapReduce

Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

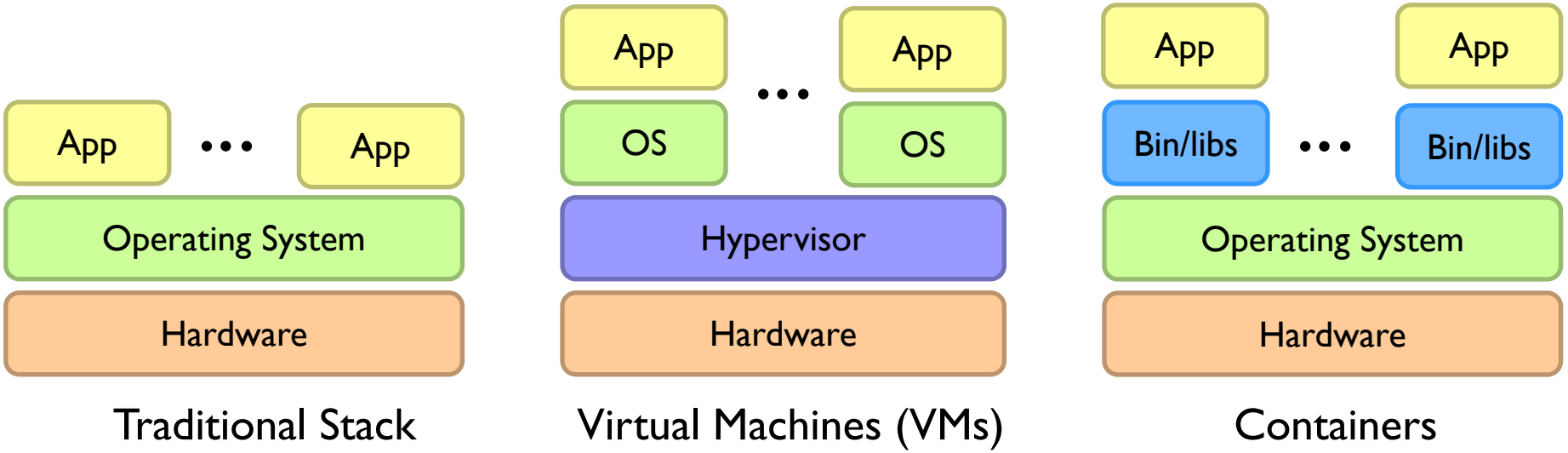
School of Computing

Announcements

- No lecture next week
- My office hours: Fri 2pm - 3pm in COM3-02-22, or Zoom (https://nus-sg.zoom.us/j/83513662217?pwd=RTNDU1dtY0VKVGx0YmpvLzF1T1ZsQT09)
- Tutorials will also be recorded (links will be on Canvas home page, same with the lectures)

Week	Date	Topics	Tutorial	Due Dates
1	18 Aug	Overview and Introduction		
2	25 Aug	MapReduce - Introduction		
3	1 Sep	Polling Day Public Holiday		
4	8 Sep	MapReduce and Databases	Tutorial: MapReduce	Assignment 1 released
5	15 Sep	NoSQL Overview 1		
6	22 Sep	NoSQL Overview 2	Tutorial: NoSQL	
Recess				
7	6 Oct	Apache Spark 1		
8	13 Oct	Apache Spark 2	Tutorial: Spark	Assignment 1 due (15 Oct 11.59pm), Assignment 2 released
9	20 Oct	Stream Processing 1		
10	27 Oct	Stream Processing 2	Tutorial: Stream Processing	
11	3 Nov	Large Graph Processing 1		
12	10 Nov	NUS Well-Being Day		
13	17 Nov	Large Graph Processing 2	Tutorial: Graph Processing	Assignment 2 due (19 Nov 11.59pm)
	29 Nov	**Final Exam**		

Review: Virtualization and Containers



- **Virtual Machines**: enable sharing of hardware resources by running each application in an isolated virtual machine.
  - **High overhead** as each VM has its own OS.
- **Containers**: enable lightweight sharing of resources, as applications run in an isolated way, but still share the same OS.
  - A container is a **lightweight software package** that encapsulates an application and its environment.

Recap: Bandwidth vs Latency

- **Bandwidth**: maximum amount of data that can be transmitted per unit time (e.g. in GB/s)
- **Latency**: time taken for 1 packet to go from source to destination (*one-way*) or from source to destination back to source (*round trip*), e.g. in ms
- When transmitting a *large* amount of data, bandwidth tells us roughly how long the transmission will take.
- When transmitting a very *small* amount of data, latency tells us how much delay there will be.
- **Throughput** is similar to bandwidth, but instead of referring to capacity, it refers to the rate at which some data was *actually transmitted* across the network during some period of time.

Low Bandwidth

High Bandwidth

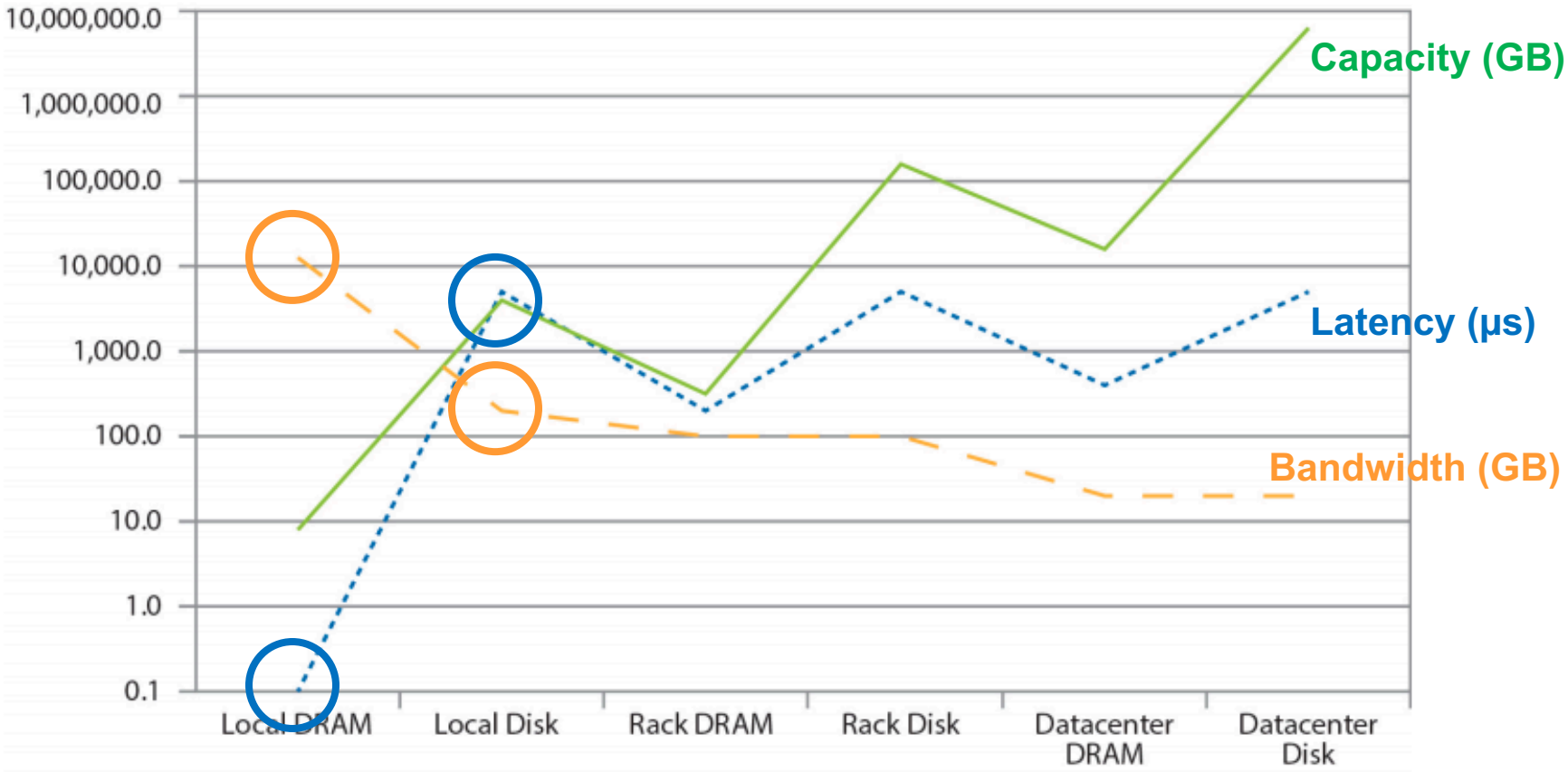
Low Latency

High Latency

Adding many lanes to a highway: increases bandwidth, but does not decrease latency

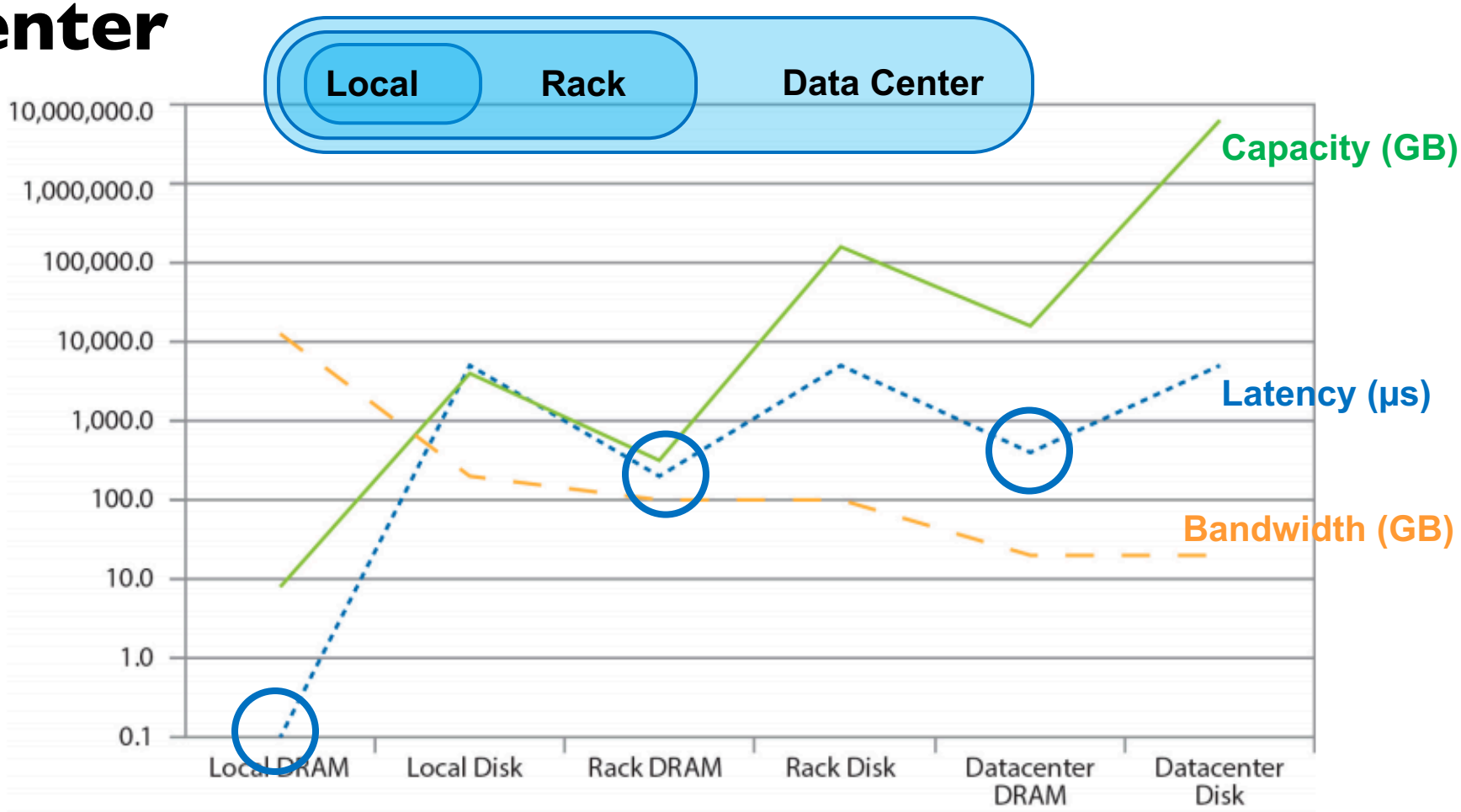
Credit: Katie Hempenius (via Jake Archibald)

Recap: Cost of Moving Data Around Data Center



Disk reads are much more expensive than DRAM, both in terms of with **higher latency** and **lower bandwidth**.

Recap: Cost of Moving Data Around Data Center



Storage hierarchy: capacity increases as we go from Local Server to Rack to Datacenter. But, there are communication costs too (in Latency + Bandwidth)

Source: Barroso and Urs Hölzle (2013)

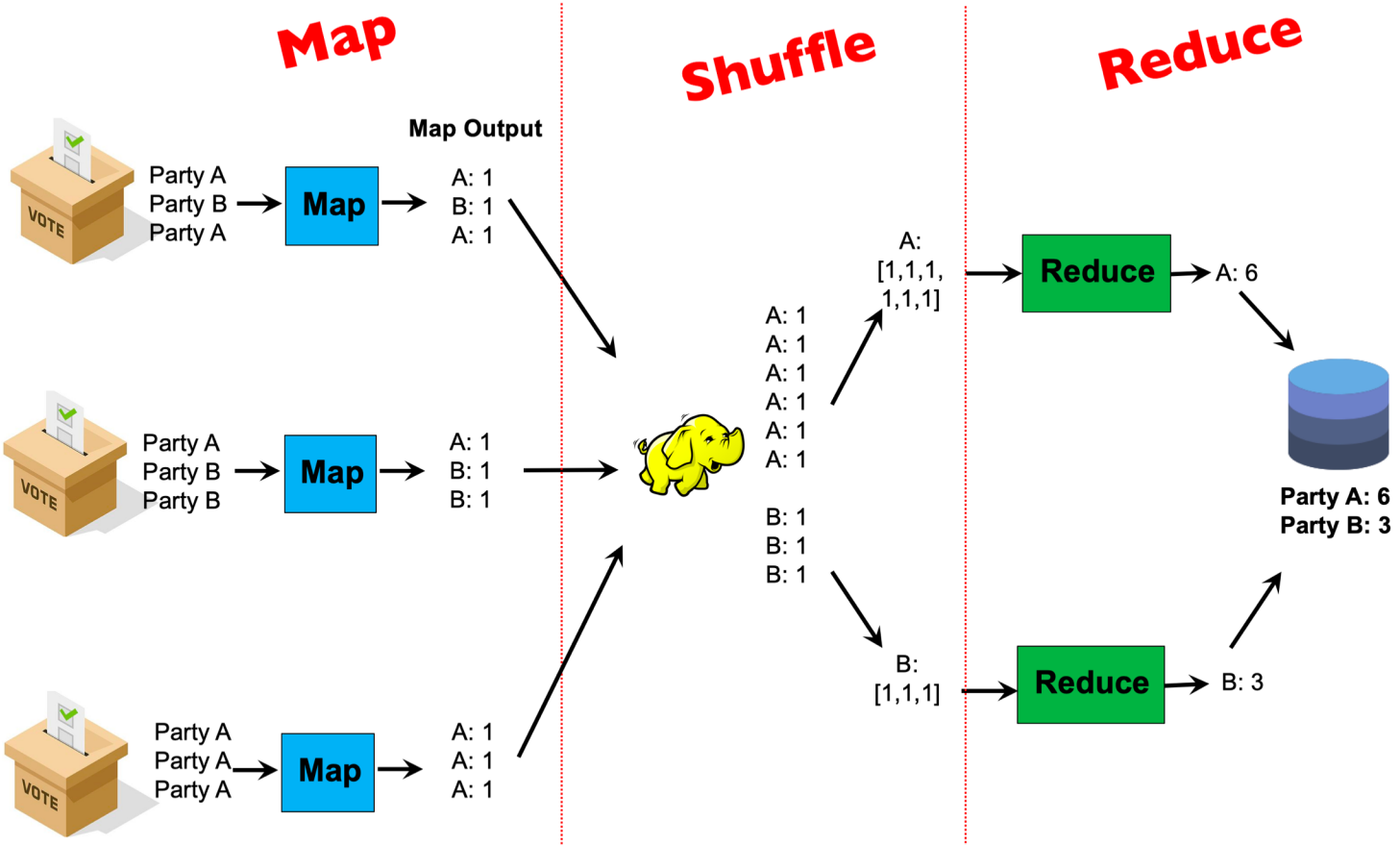
Recap: Writing MapReduce Programs

- **Typical Interface:** Programmers specify two functions:

  map $(k_1, v_1) \rightarrow \text{List}(k_2, v_2)$

  reduce $(k_2, \text{List}(v_2)) \rightarrow \text{List}(k_3, v_3)$

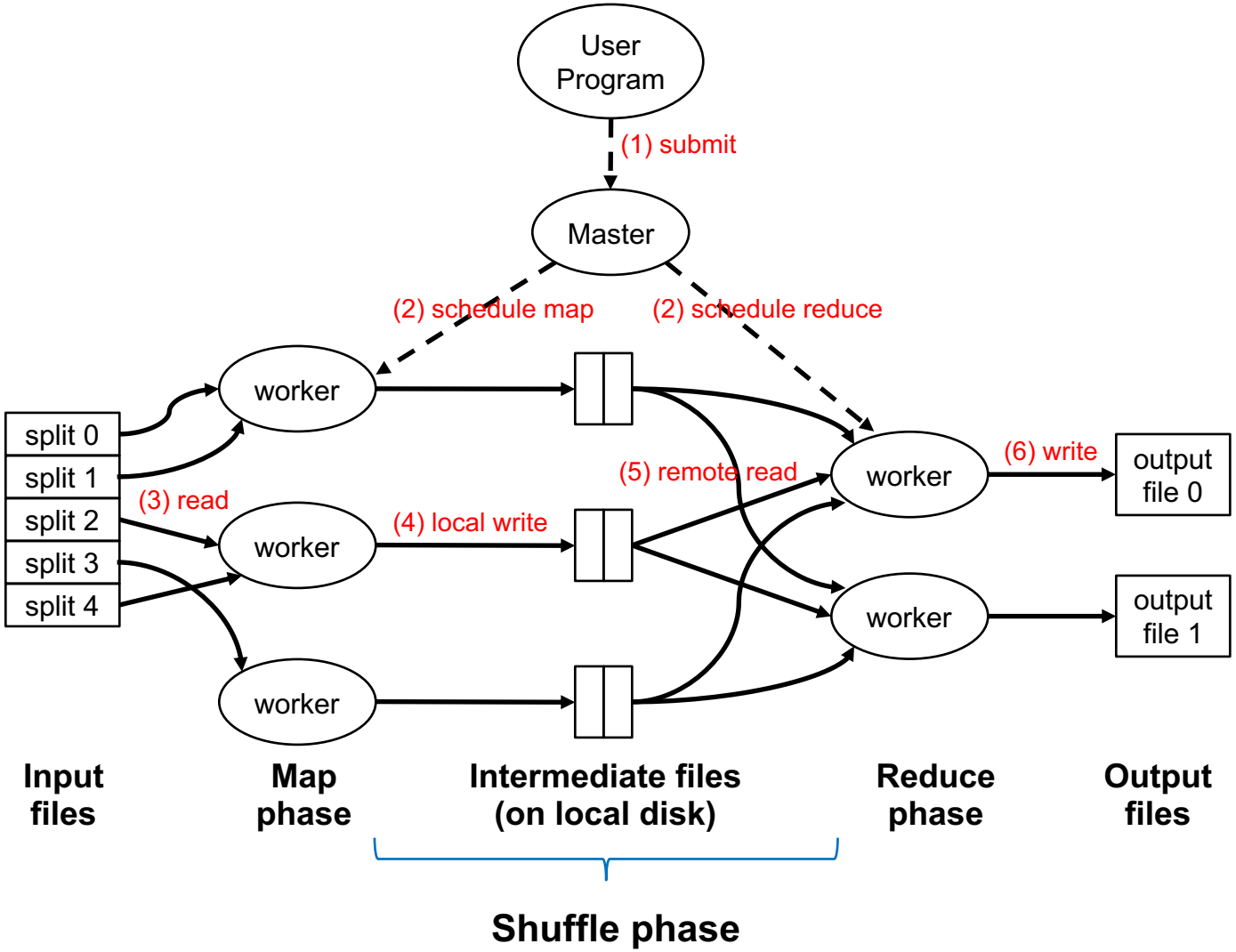
  - All values with the same key are sent to the same reducer



1. **MapReduce**
   a. **Basic MapReduce**
   b. Partition and Combiner
   c. Examples
   d. Secondary Sort
2. Hadoop File System

Source: Google

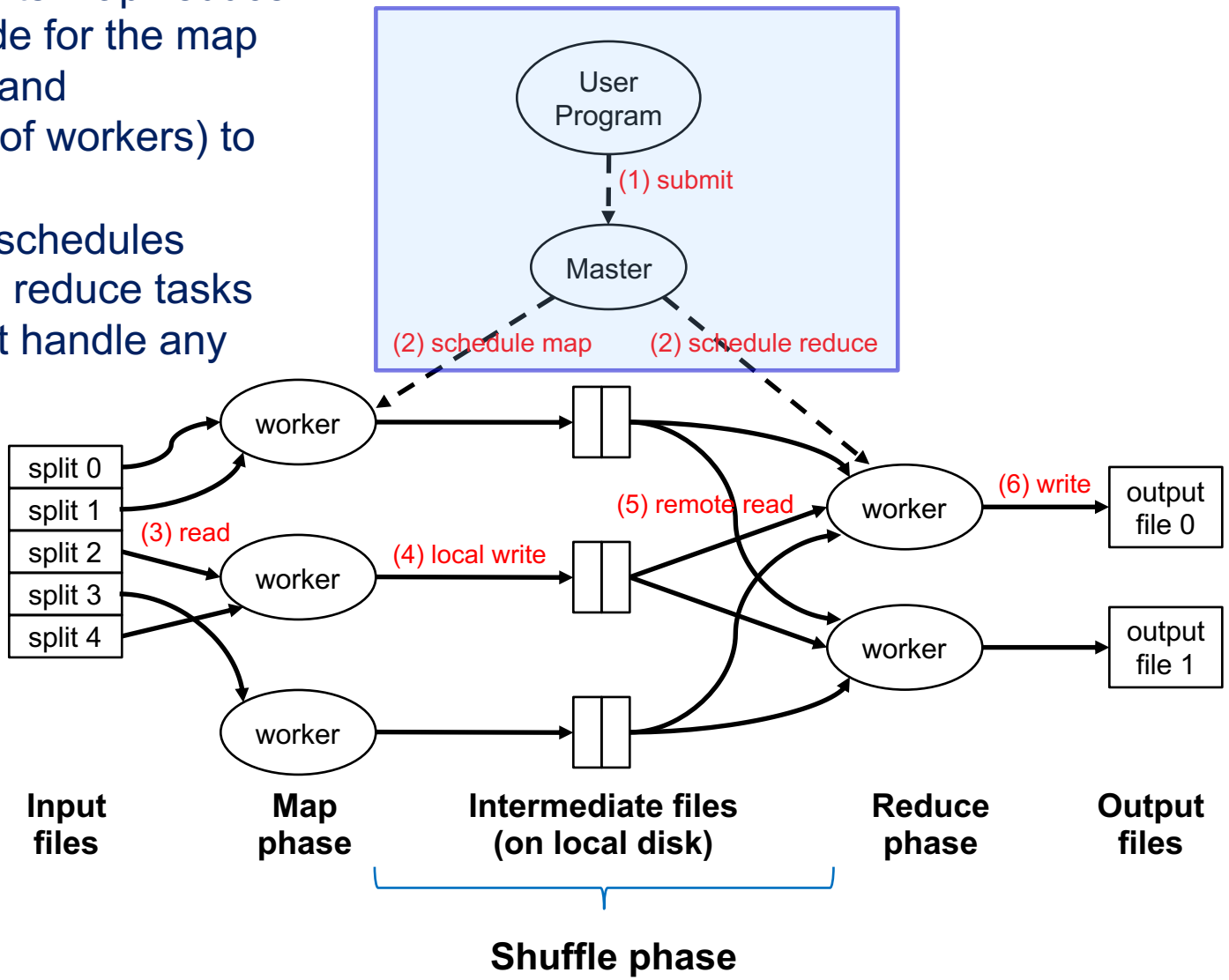
MapReduce Implementation



MapReduce Implementation

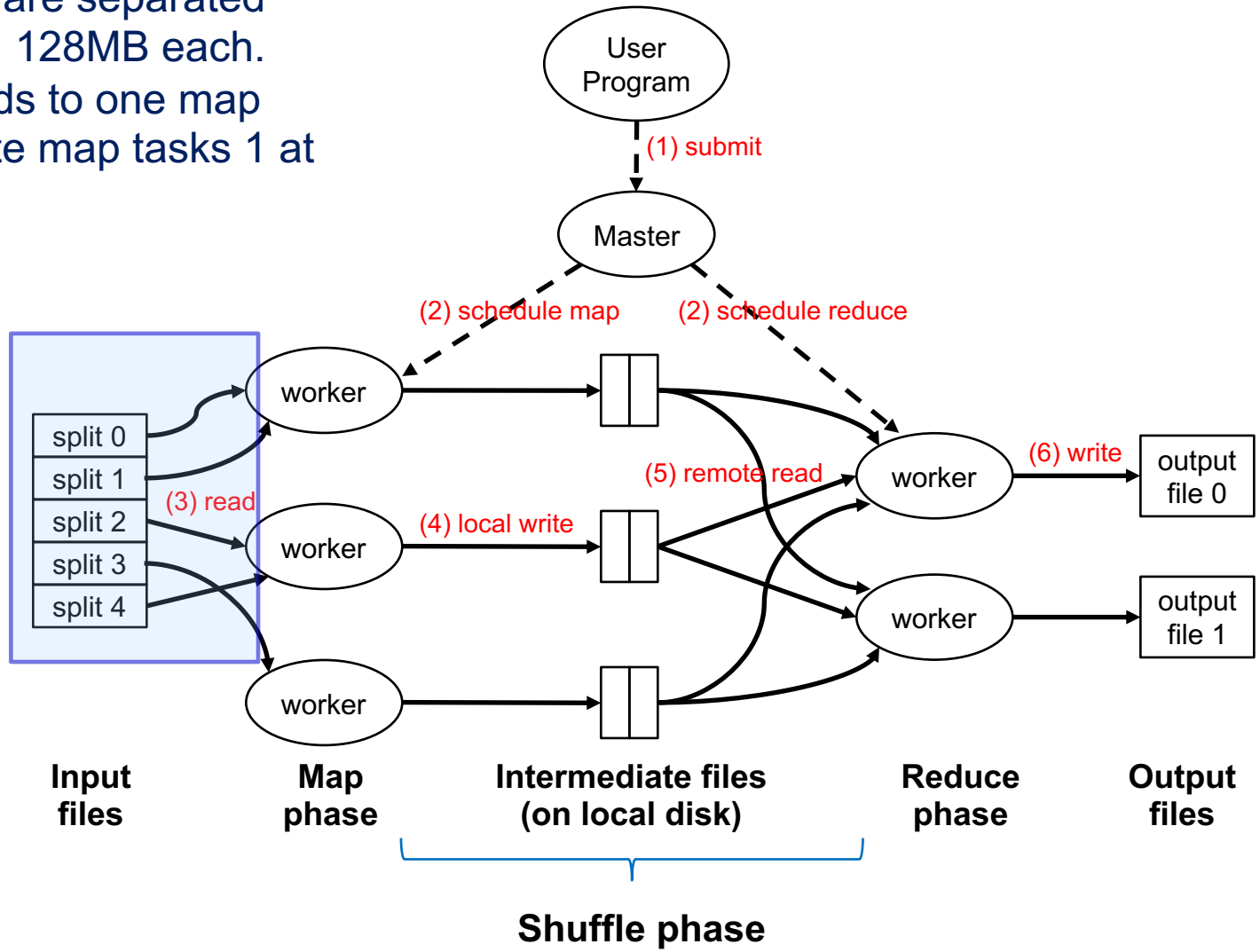
(1) Submit: user submits MapReduce program (including code for the map and reduce functions) and configuration (e.g. no. of workers) to Master node

(2) Schedule: Master schedules resources for map and reduce tasks (Note: Master does not handle any actual data)



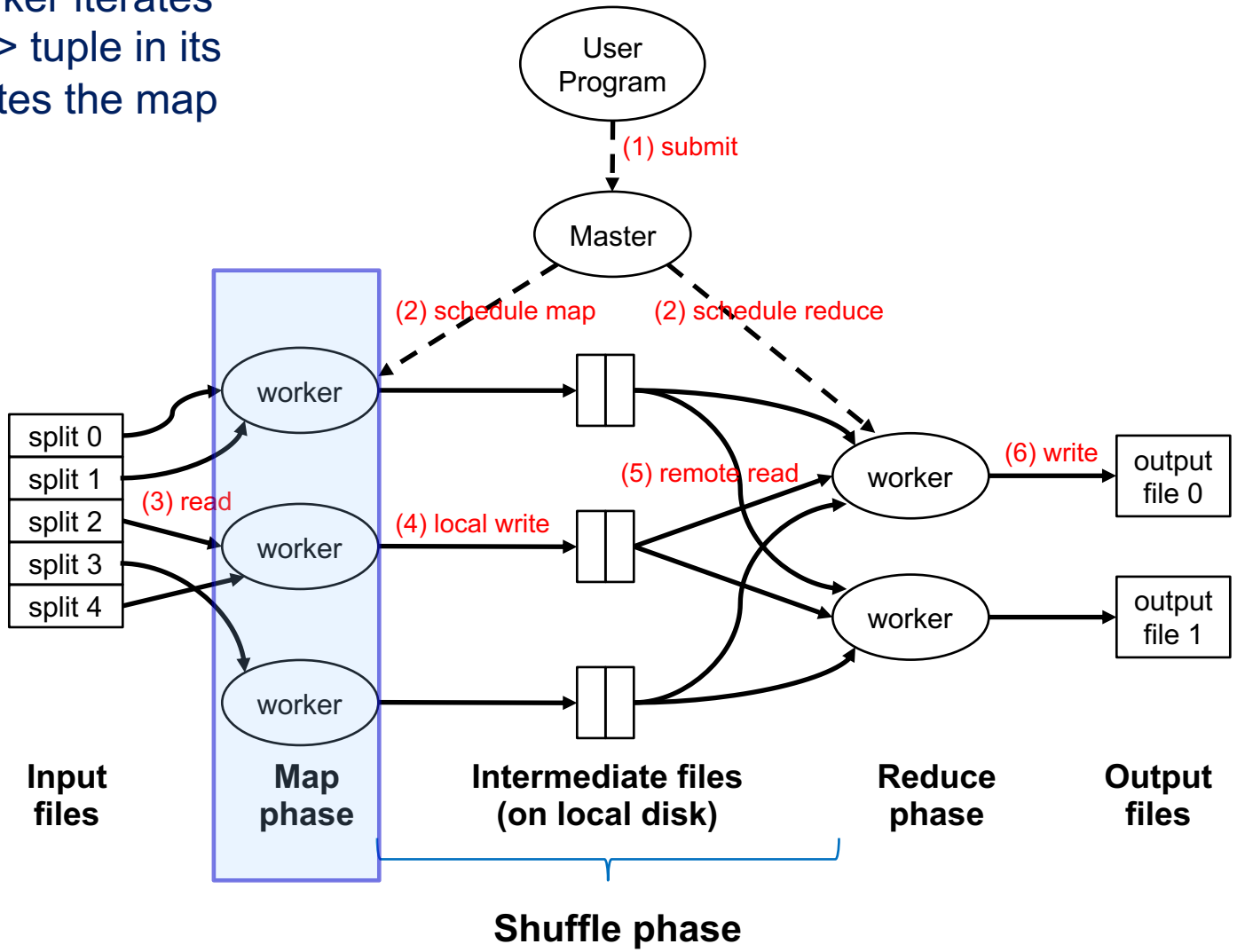
MapReduce Implementation

(3) Read: Input files are separated into "splits" of around 128MB each. Each split corresponds to one map task. Workers execute map tasks 1 at a time.



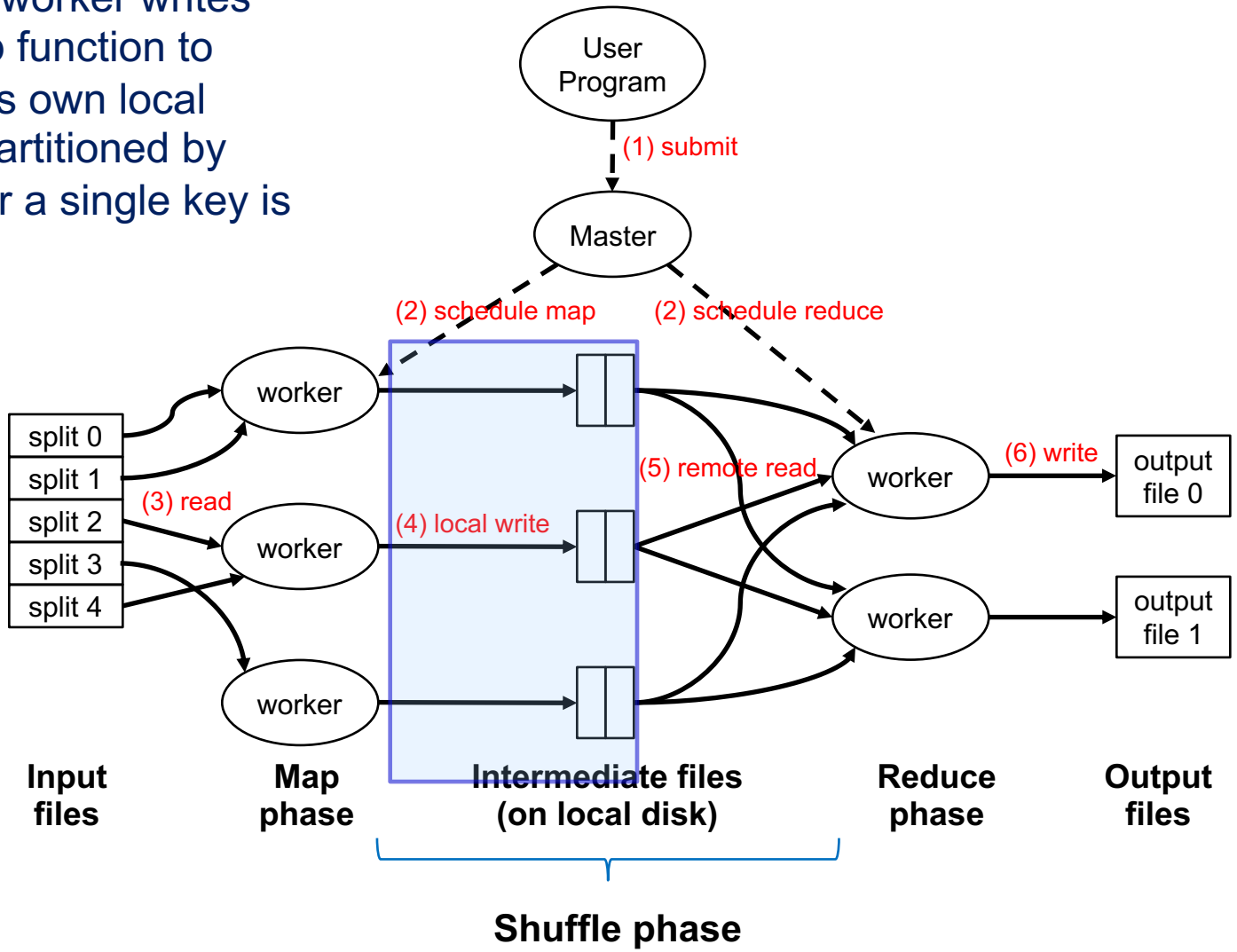
MapReduce Implementation

Map phase: Each worker iterates over each <key, value> tuple in its input split, and computes the map function on each tuple



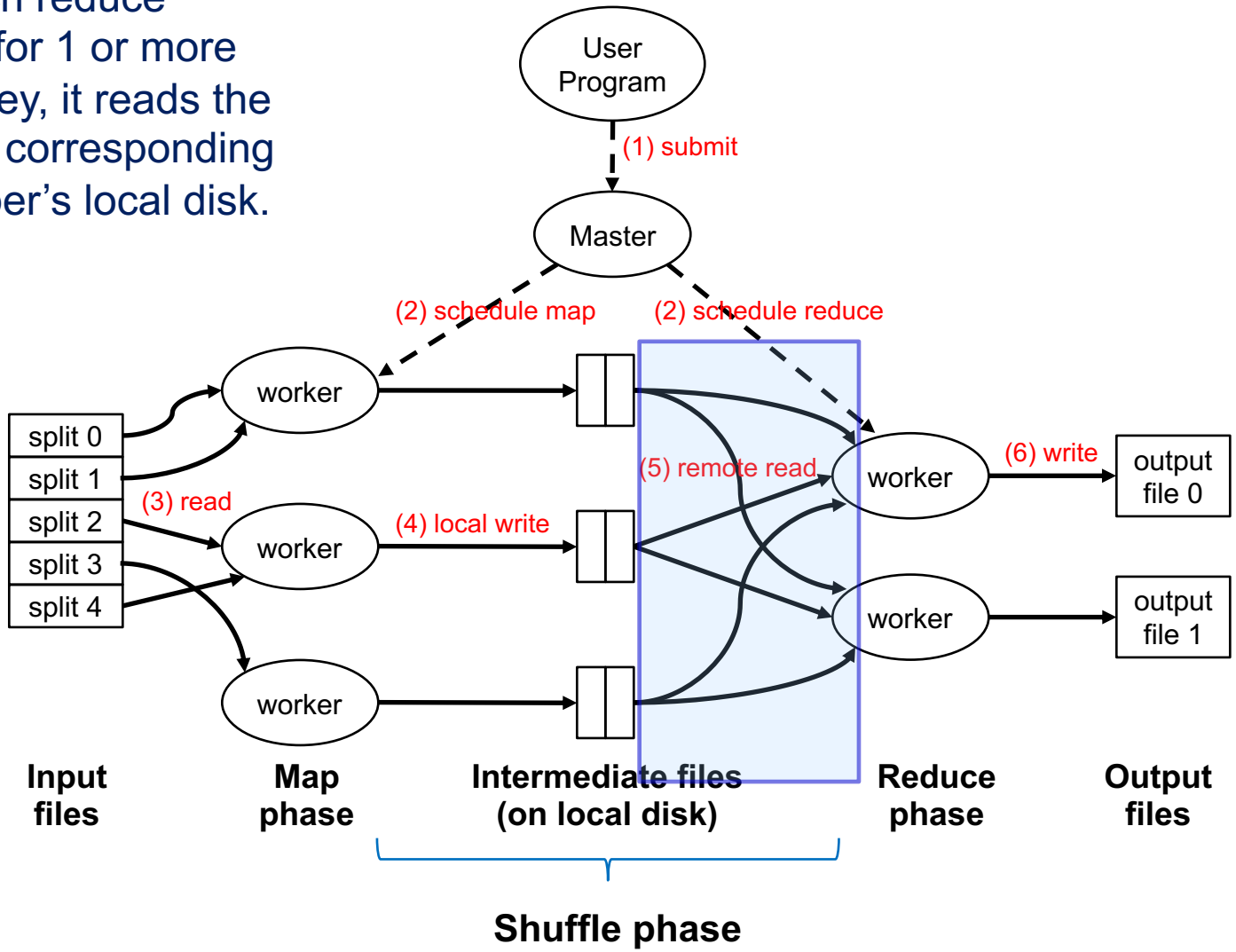
MapReduce Implementation

(4) Local write: Each worker writes the outputs of the map function to intermediate files on its own local disk. These files are partitioned by key (i.e. all the data for a single key is in one partition)



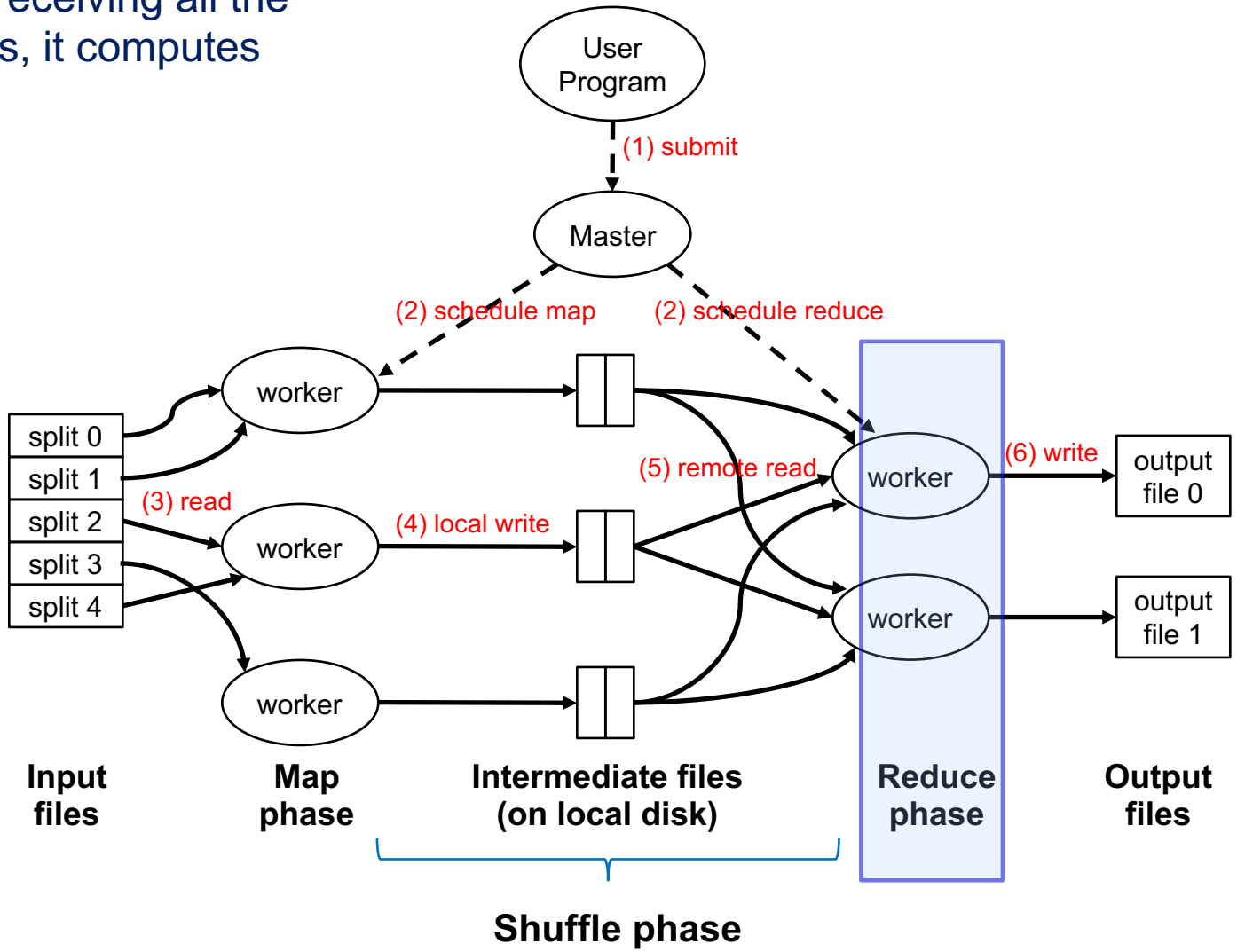
MapReduce Implementation

(4) Remote read: Each reduce worker is responsible for 1 or more keys. For each such key, it reads the data it needs from the corresponding partition of each mapper's local disk.



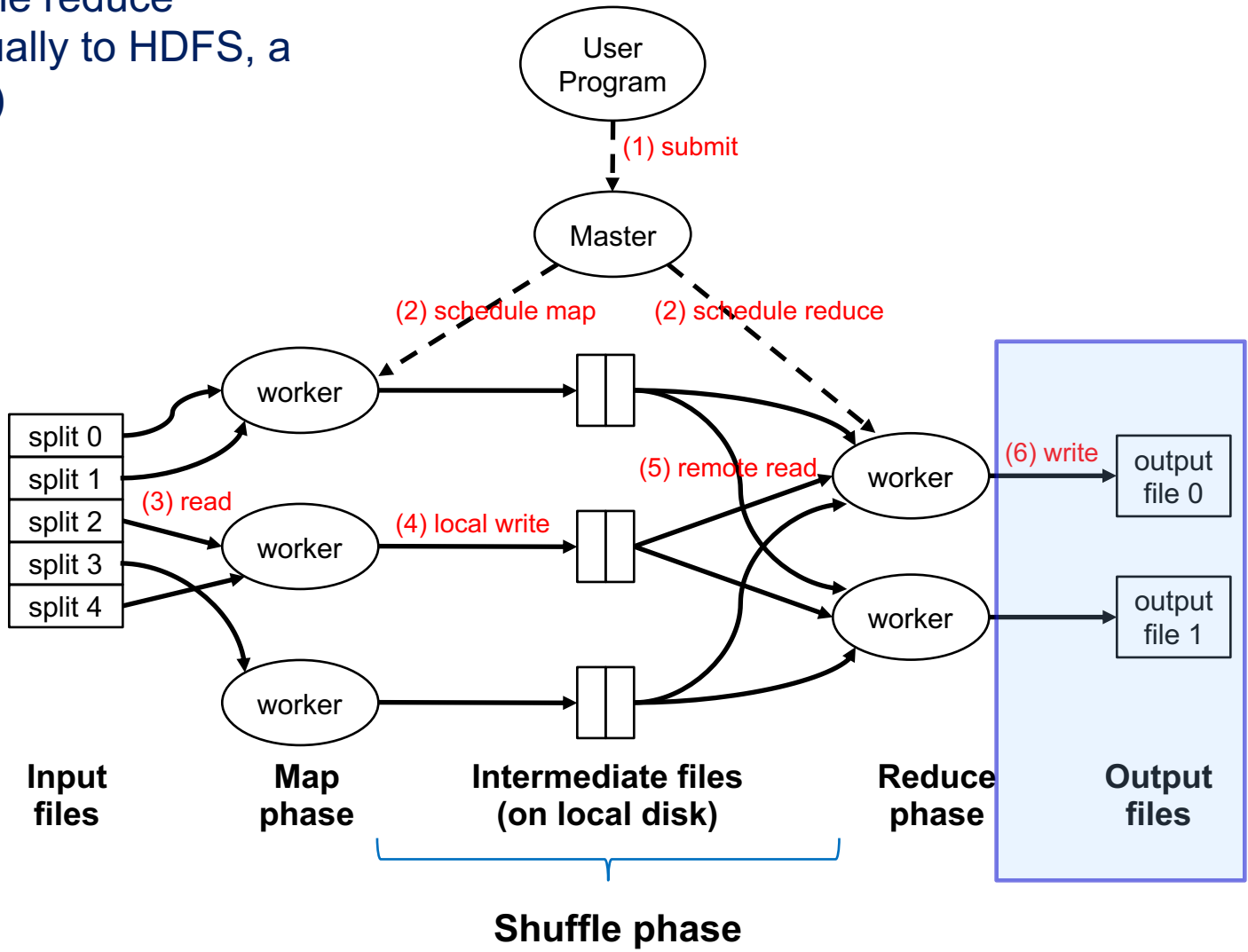
MapReduce Implementation

Reduce phase: After receiving all the needed key value pairs, it computes the reduce function.



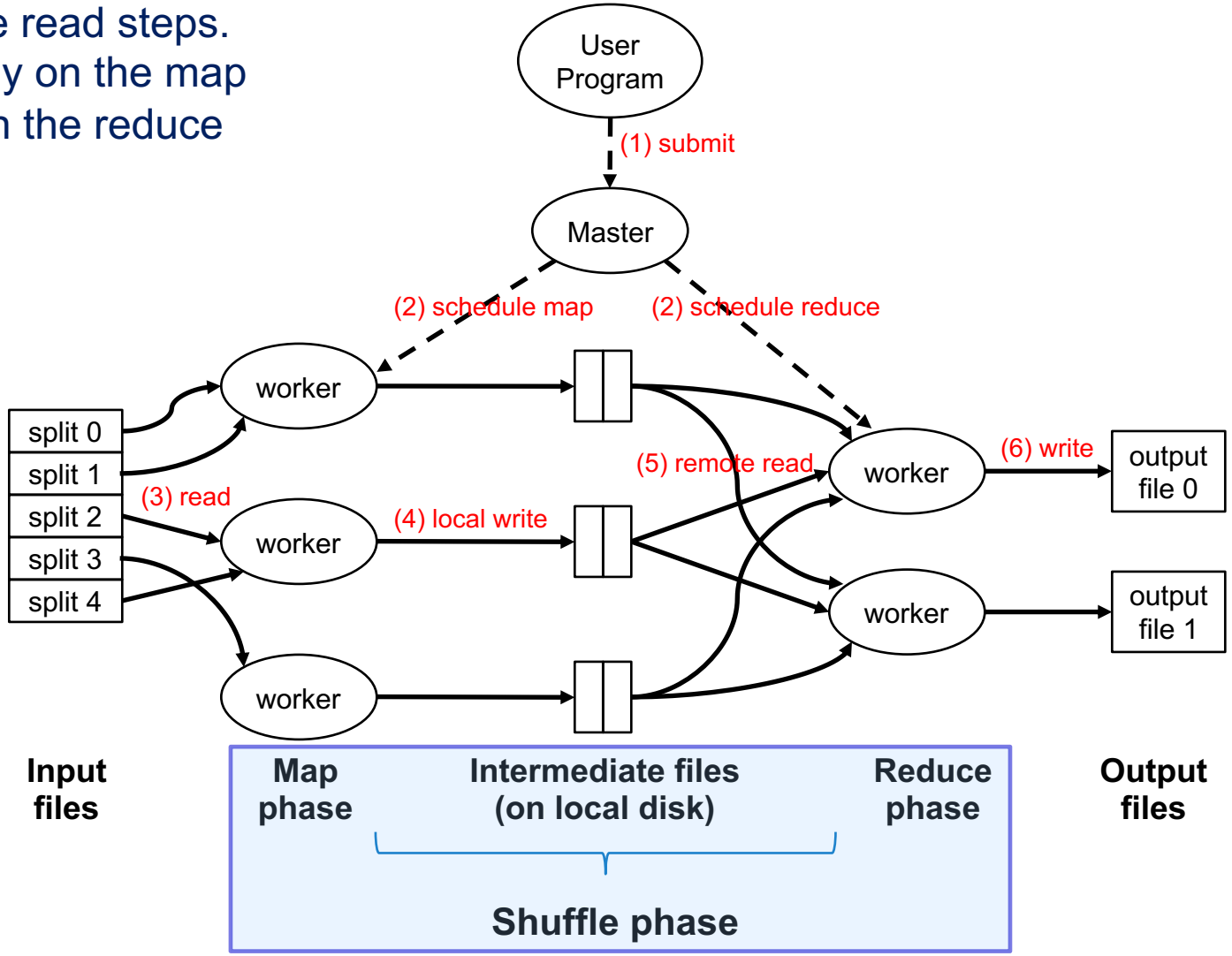
MapReduce Implementation

Write: The output of the reduce function is written (usually to HDFS, a distributed file system)

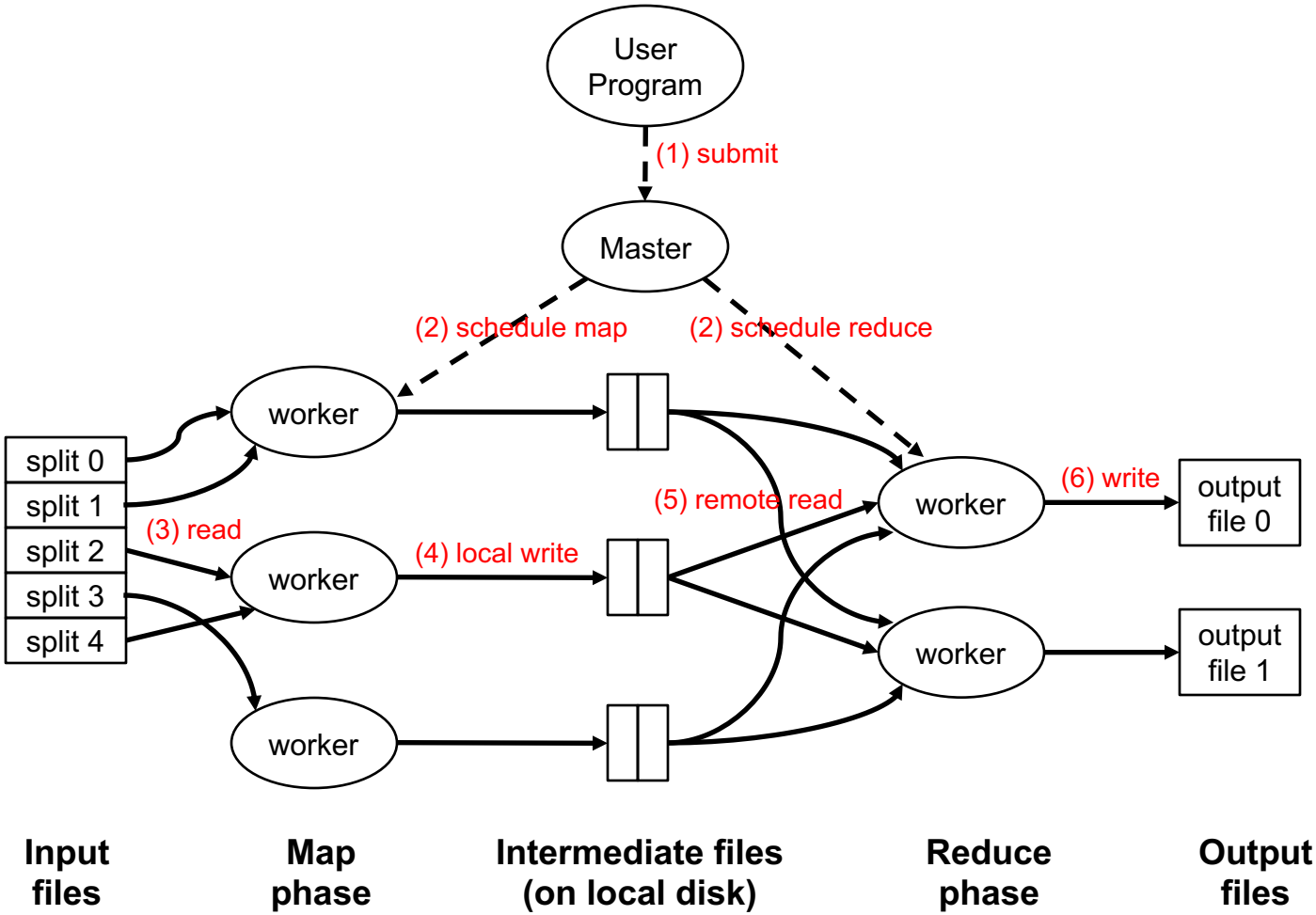


MapReduce Implementation

Clarification of "shuffle phase": the "shuffle phase" is comprised of the local write and remote read steps. Thus, it happens partly on the map workers, and partly on the reduce workers.



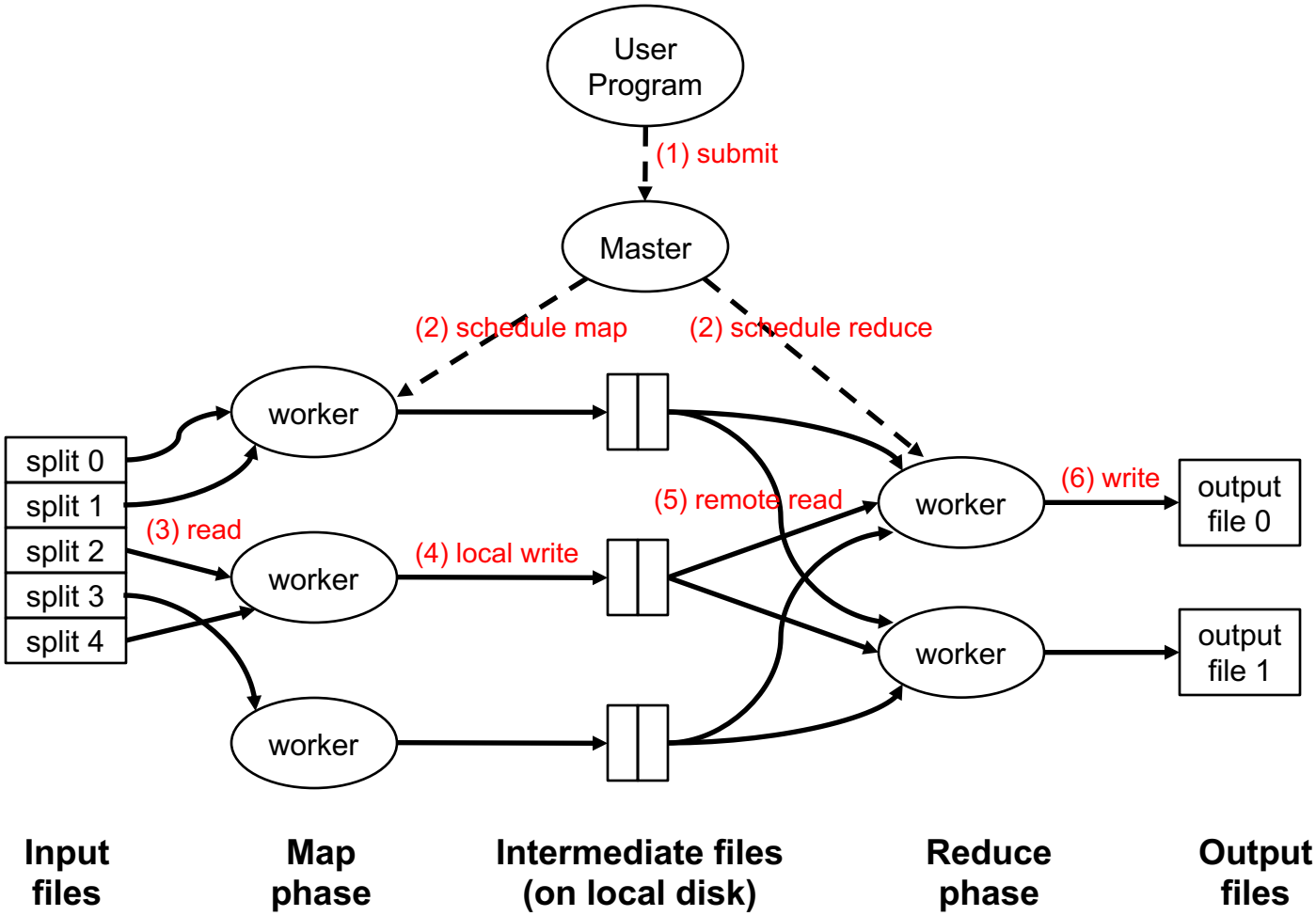
MapReduce Implementation



Q: What disadvantages are there if the size of each split (or chunk) is too big or small?

Adapted from (Dean and Ghemawat, OSDI 2004)

MapReduce Implementation

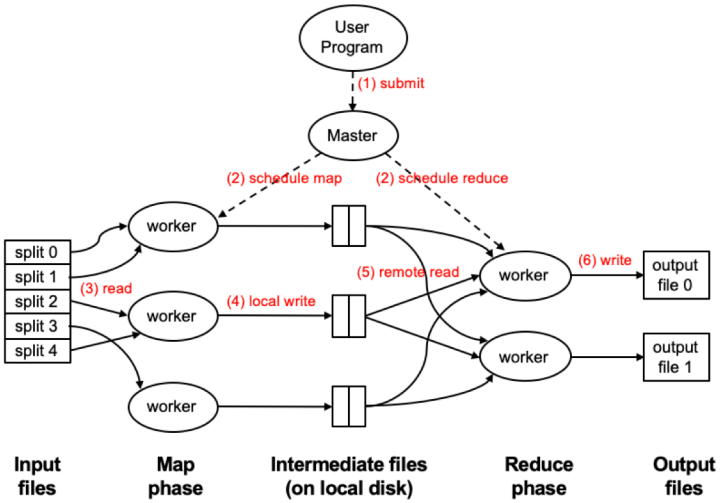


Q: What disadvantages are there if the size of each split (or chunk) is too big or small?
A: Too big: limited parallelism. Too small: high overhead (master node may be overwhelmed by scheduling work)

Adapted from (Dean and Ghemawat, OSDI 2004)

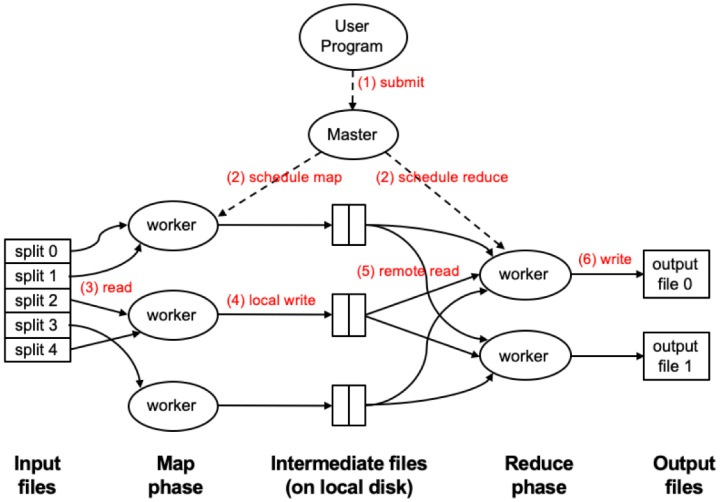
Important Clarifications: Map Task, Mapper, Map Function?

- These terms can get confusing, please be clear on them
- A **worker** is a component of the cluster that performs storage and processing tasks (you can loosely think of it as a physical machine)
- **Map Task** is a basic unit of work; it is typically 128MB. At the beginning the input is broken into splits of 128MB. A map task is a **job** requiring to process one split; not a worker.
- A single worker can handle multiple map tasks. Typically, when a worker completes a map task (e.g. split 0), it is re-assigned to another task (e.g. split 3)



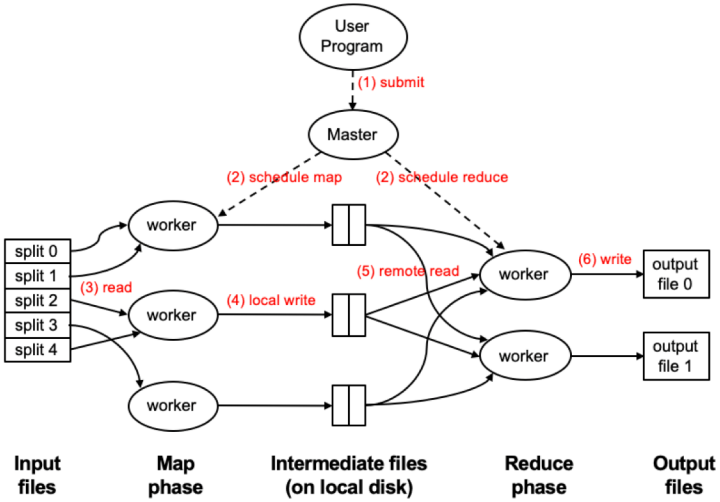
Important Clarifications: Map Task, Mapper, Map Function?

- **A "mapper" or "reducer"** will generally refer to the process executing a map or reduce task, not to physical machines / workers.
- E.g. in this diagram there are 5 map tasks, and thus 5 mappers, but only 3 workers.



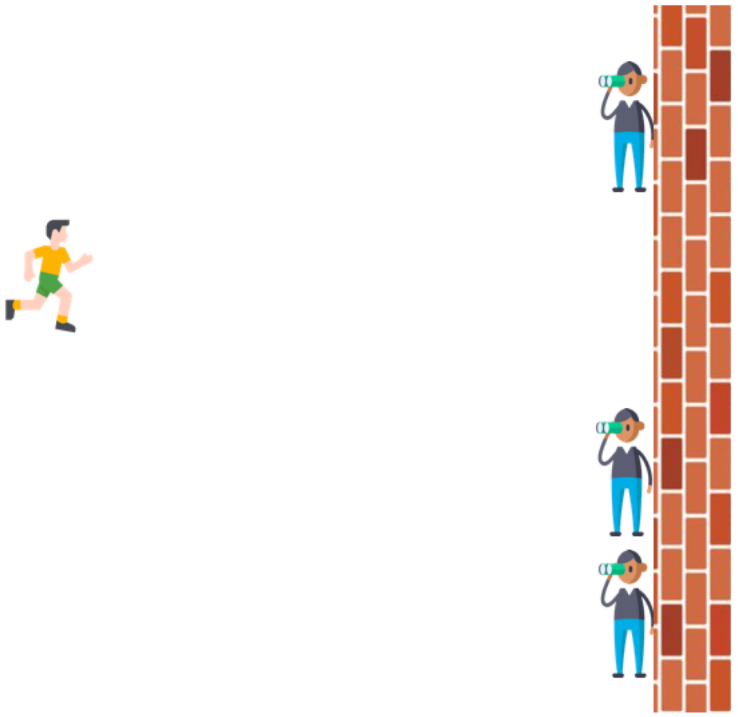
Important Clarifications: Map Task, Mapper, Map Function?

- **"Map Function"** is a single call to the user-defined **map** $(k_1, v_1) \rightarrow \text{List}(k_2, v_2)$ function.
- Note that a single map task can involve many calls to such a map function: e.g. within a 128MB split, there will often be many (key, value) pairs, each of which will produce one call to a map function.



Two more details…

- Barrier between map and reduce phases
  - Necessary, otherwise the reduce phase might compute the wrong answer
  - Note that the shuffle phase can begin copying intermediate data earlier
- If a reduce task handles multiple keys, it will process these keys in sorted order



1. **MapReduce**
   a. Basic MapReduce
   b. Partition and Combiner
   c. Examples
   d. Secondary Sort
2. Hadoop File System

Source: Google

Writing MapReduce Programs

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow \text{List}(k_2, v_2)$

  reduce $(k_2, \text{List}(v_2)) \rightarrow \text{List}(k_3, v_3)$

  - All values with the same key are reduced together

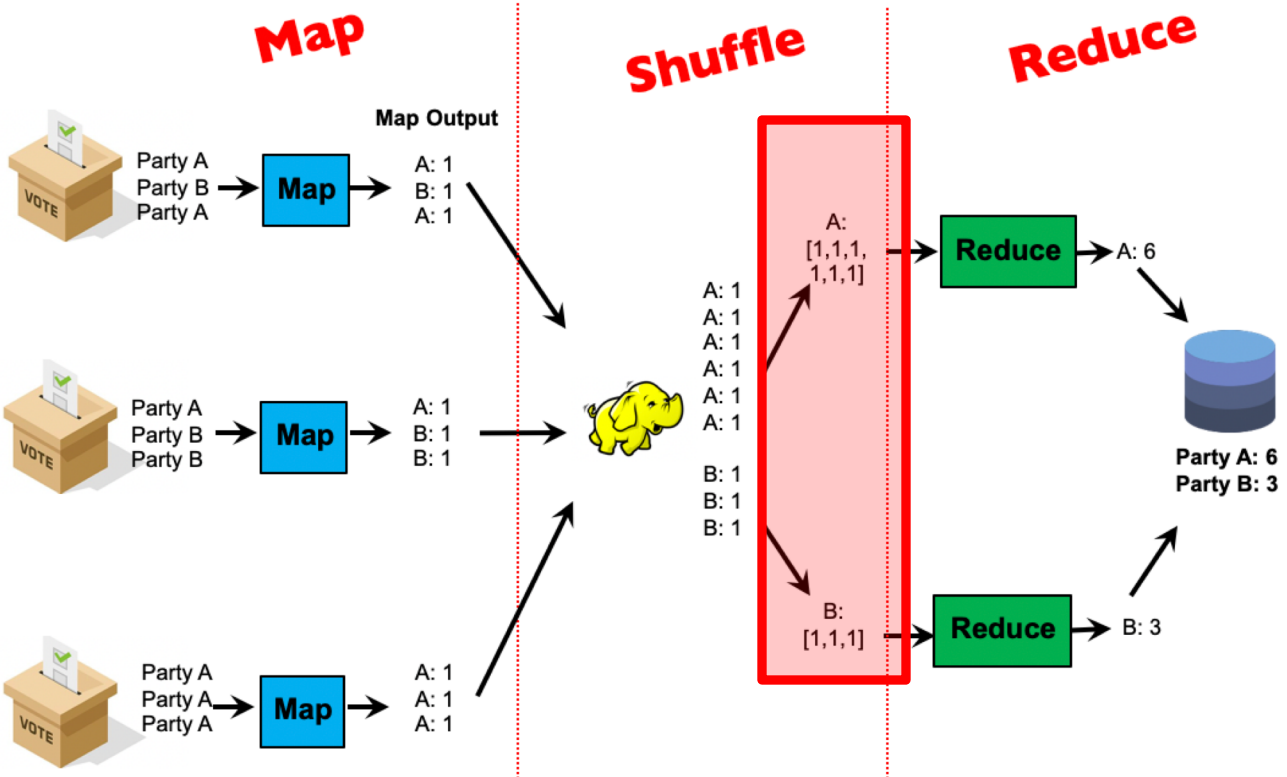
- The execution framework handles everything else…

- Not quite…usually, programmers **optionally** also specify

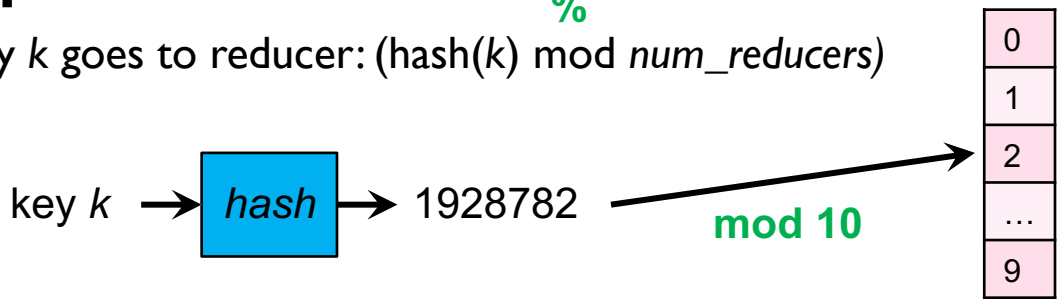
  partition, and **combine** functions

  - These are an optional optimization to reduce network traffic

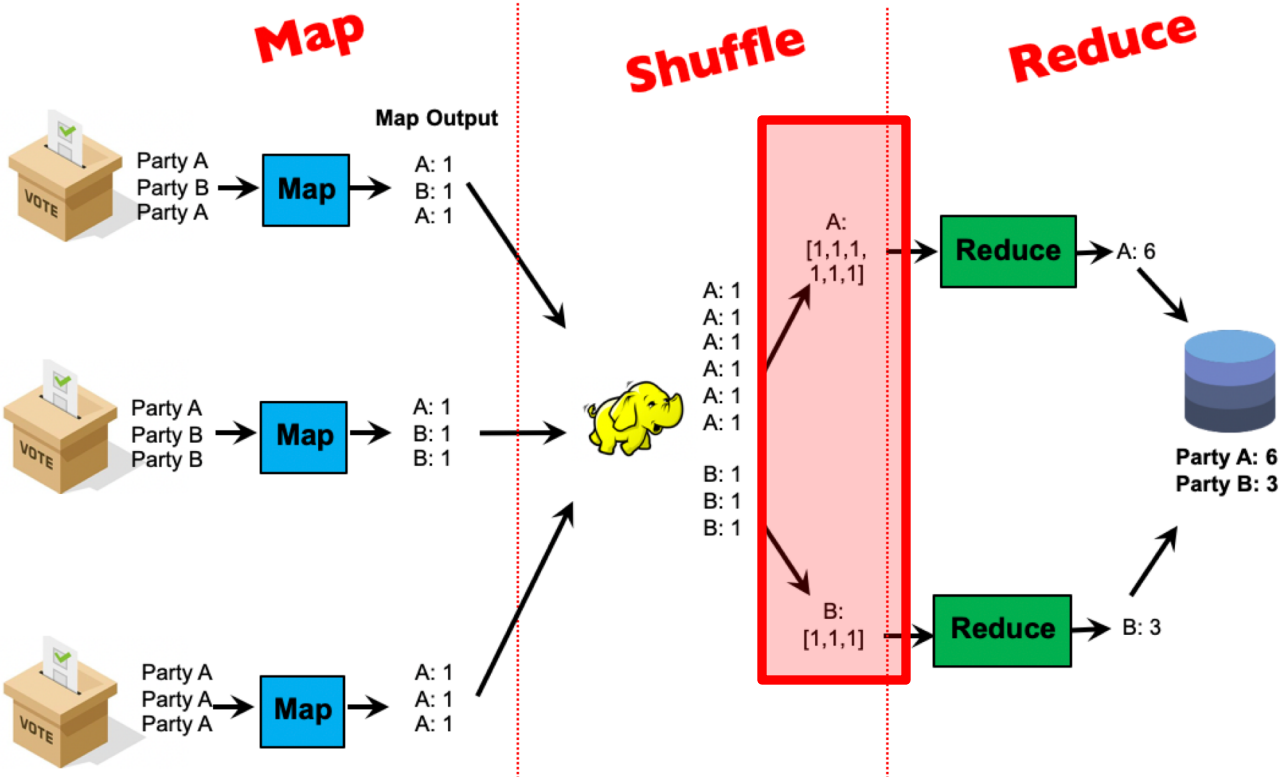
Partition Step



- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a **hash function**
  - e.g., key *k* goes to reducer: (hash(*k*) mod *num_reducers*)
-

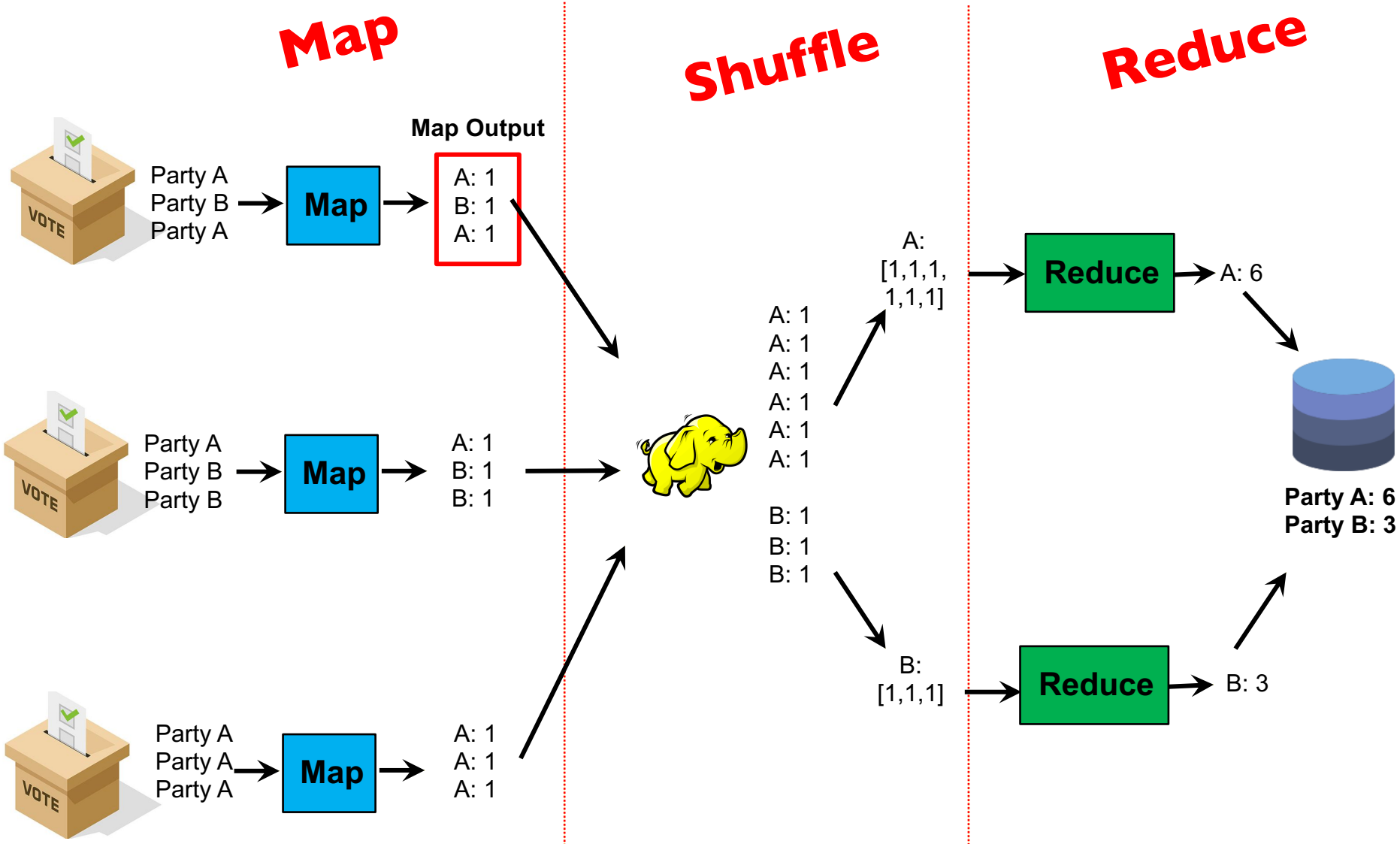


Partition Step



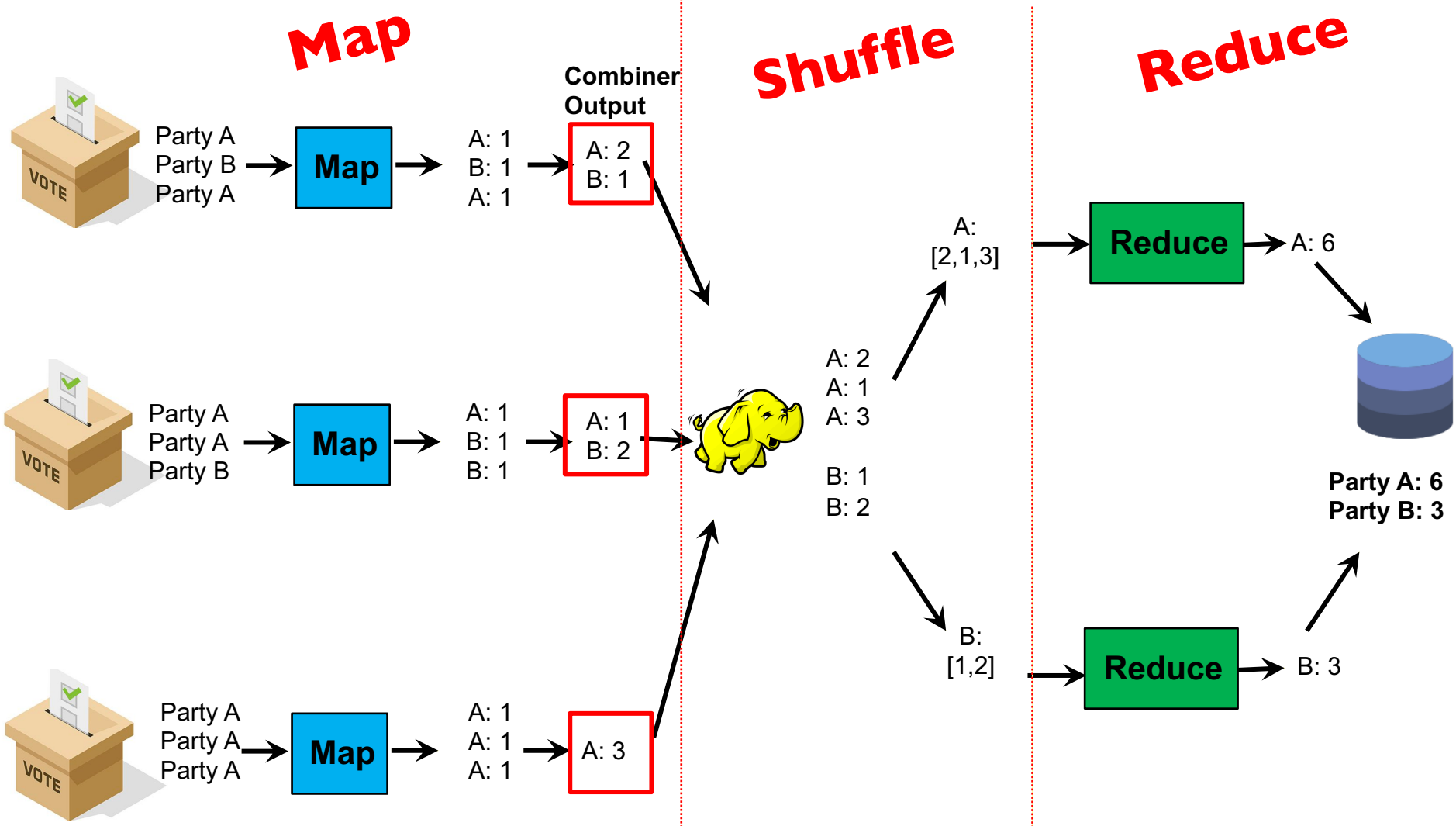
- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a **hash function**
  - e.g., key *k* goes to reducer: (hash(*k*) % *num_reducers*)
- User can optionally implement a custom partition, e.g. to better spread out the load among reducers (if some keys have much more values than others)

Combiner Step



- Writing map output to disk is expensive! Can we reduce disk writes?

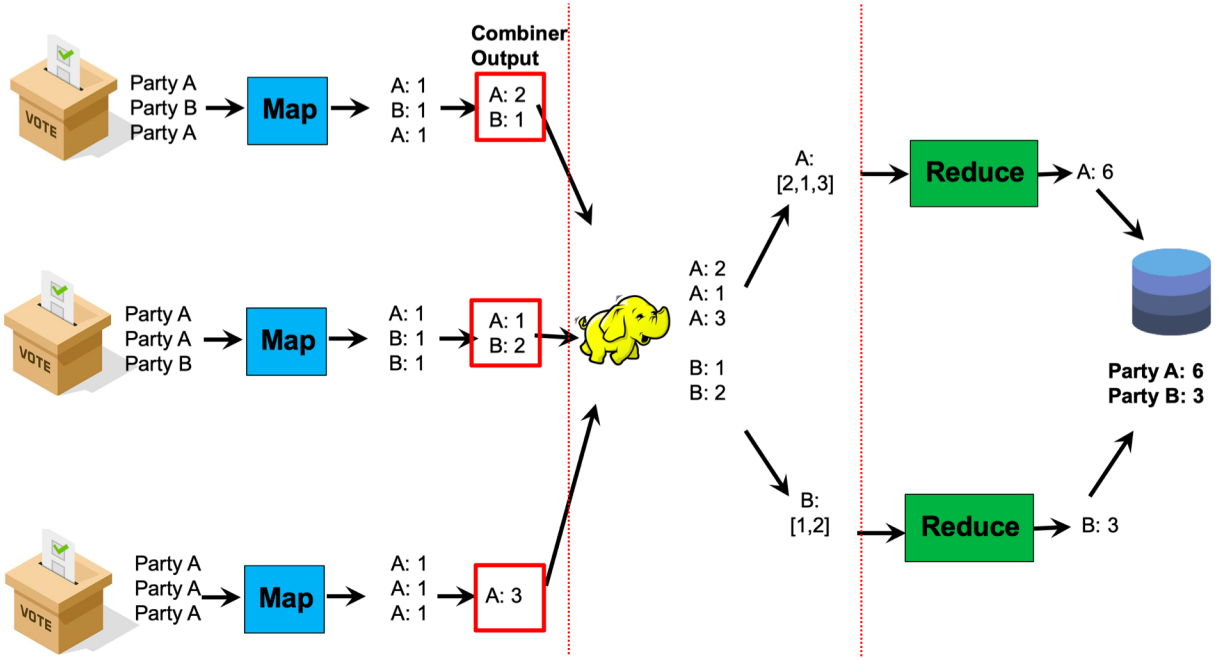
Combiner Step



- Combiners **locally aggregate** output from mappers.
- Combiners are 'mini-reducers': in this example, combiners and reducers are the same function!

Correctness of Combiner

- It is the user's responsibility to ensure that the combiner does not affect the correctness of the final output, whether the combiner runs 0, 1, or multiple times
  - Example: in election example, the combiner and reducer are a "sum" over values with the same key. Summing can be done in any order without affecting correctness:
    - e.g. sum(sum(1, 1), 1, sum(1, 1, 1)) = sum(1, 1, 1, 1, 1, 1) = 6
  - The same holds for "max" and "min"
  - How about "mean" or "minus"?



Correctness of Combiner

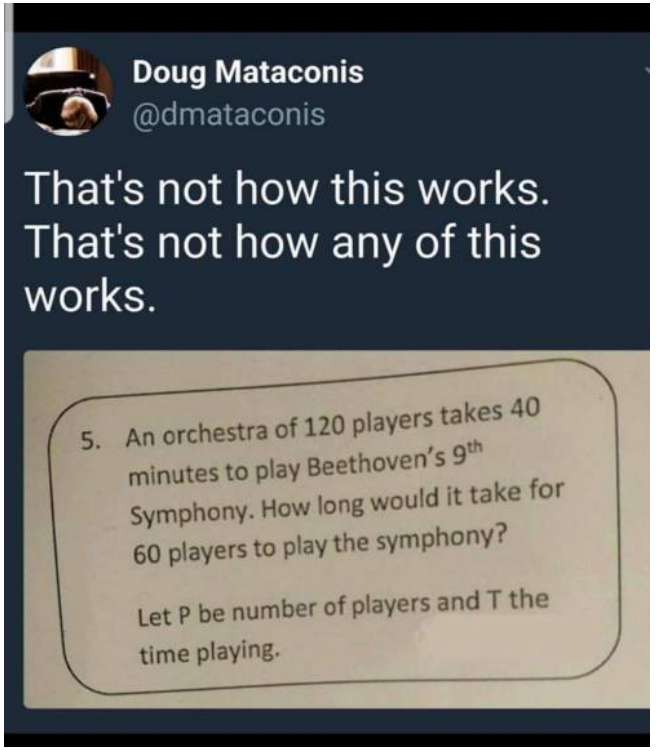
- It is the user's responsibility to ensure that the combiner does not affect the correctness of the final output, whether the combiner runs 0, 1, or multiple times
  - Example: in election example, the combiner and reducer are a "sum" over values with the same key. Summing can be done in any order without affecting correctness:
    - e.g. sum(sum(1, 1), 1, sum(1, 1, 1)) = sum(1, 1, 1, 1, 1, 1) = 6
  - The same holds for "max" and "min"
  - How about "mean" or "minus"?
    - Answer: No! E.g. mean(mean(1, 1), 2) ≠ mean(1, 1, 2).
  - (Optional) In general, it is correct to use reducers as combiners if the reduction involves a binary operation (e.g. +) that is both
    - **Associative:** a + (b + c) = (a + b) + c
    - **Commutative:** a + b = b + a

1. **MapReduce**
   a. Basic MapReduce
   b. Partition and Combiner
   c. Examples
   d. Secondary Sort
2. Hadoop File System

Source: Google

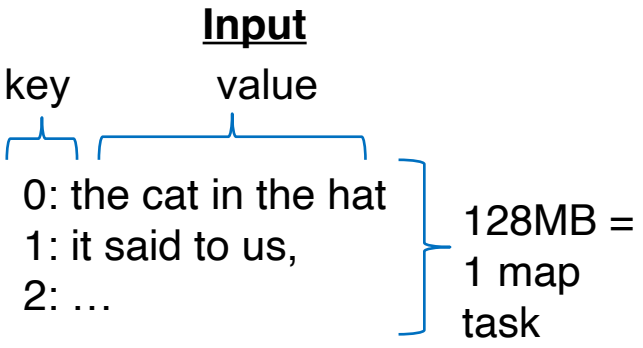
Performance Guidelines for Basic Algorithmic Design

- **Linear scalability:** more nodes can do more work in the same time
  - Linear on data size
  - Linear on computer resources
- **Minimize disk and network I/O**
  - Minimize disk I/O; sequential vs. random.
  - Minimize network I/O; send data in bulk vs in small chunks
- **Reduce memory working set** of each task/worker
  - ”Working set” = portion of memory that is actively being used during algorithm execution
  - Large working set -> high memory requirements / probability of out-of-memory errors.
- Guidelines are applicable to Hadoop, Spark, …



Word Count: Version 0

```
1  class Mapper {
2    def map(key: Long, value: Text) = {
3      for (word <- tokenize(value)) {
4        emit(word, 1)
5    }
6  }
7
8  class Reducer {
9    def reduce(key: Text, values: Iterable[Int]) = {
10     for (value <- values) {
11       sum += value
12     }
13     emit(key, sum)
14   }
15 }
```



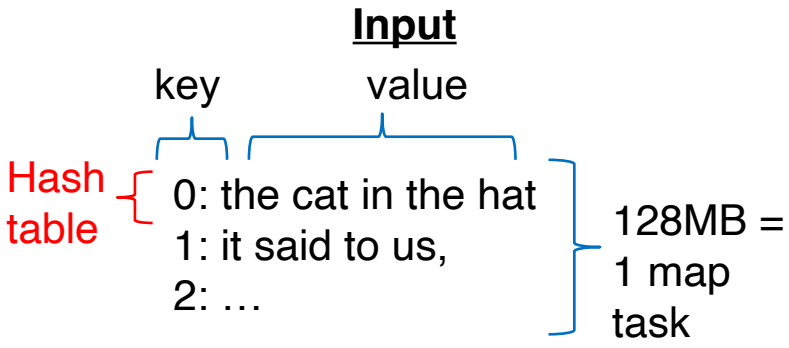
This mapper processes each word one by one, and emits a "1", to be summed by the reducers.

What's the key problem of this program?

What's the impact of combiners?

Word Count: Version 1

```
1   class Mapper {
2     def map(key: Long, value: Text) = {
3       val counts = new Map()
4       for (word <- tokenize(value)) {
5         counts(word) += 1
6       }
7       for ((k, v) <- counts) {
8         emit(k, v)
9       }
10    }
11  }
```

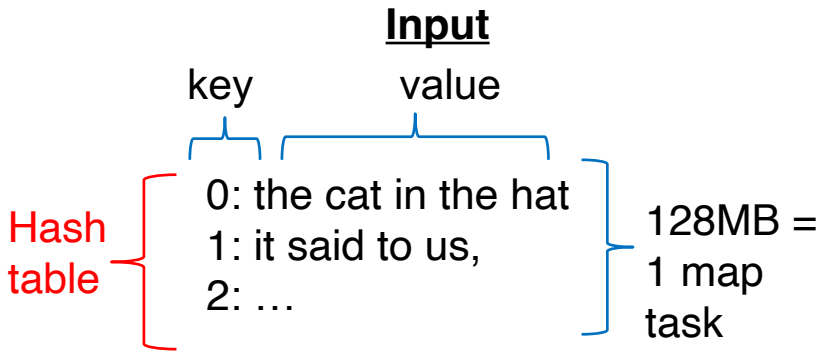


This mapper uses a hash table ("counts") to maintain the words and counts per line (i.e. in each call to the map function). After processing each line it emits the counts for this line.

Are combiners still of any use?

Word Count: Version 2

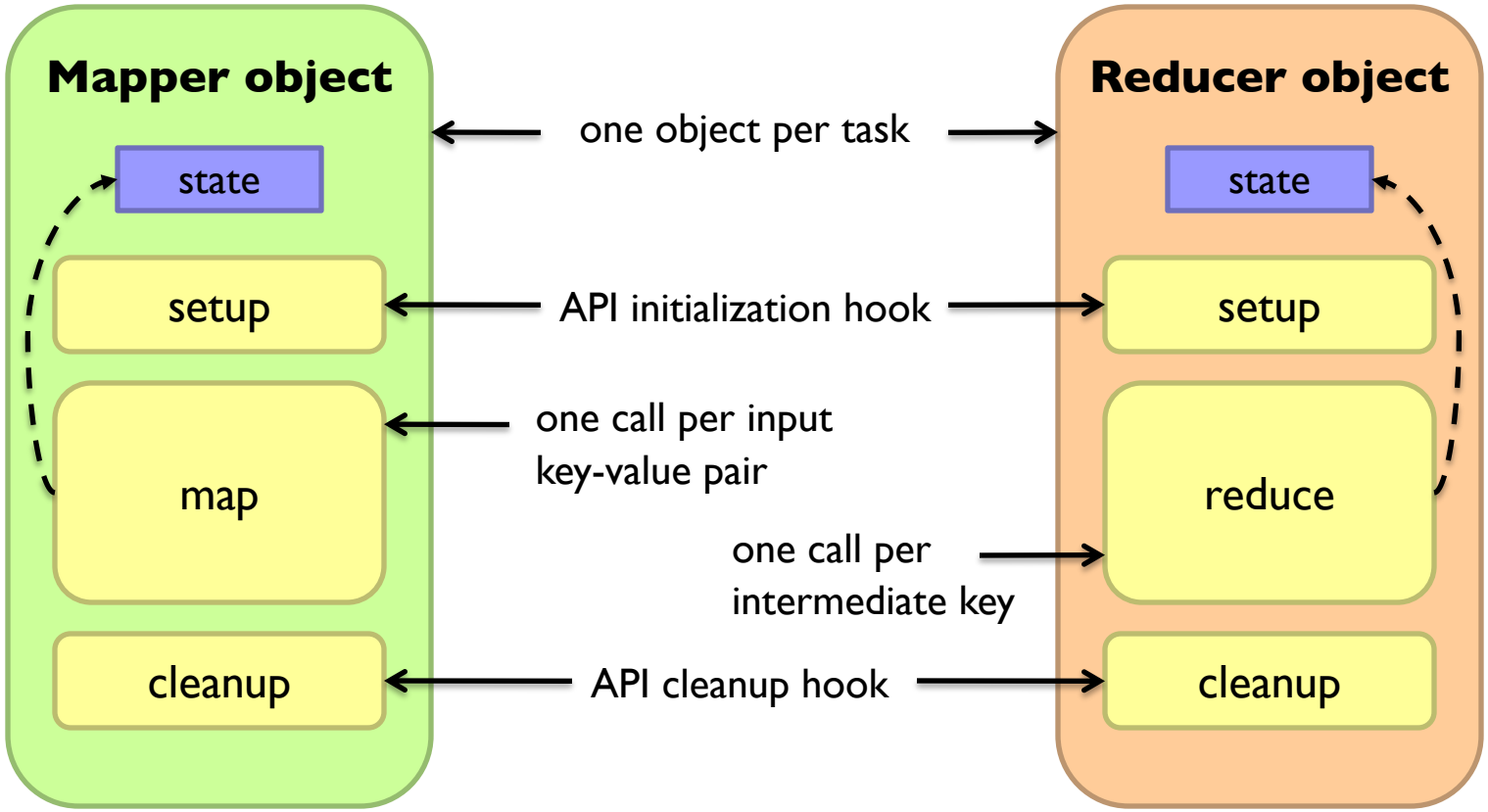
```
1   class Mapper {
2     val counts = new Map()
3
4     def map(key: Long, value: Text) = {
5       for (word <- tokenize(value)) {
6         counts(word) += 1
7       }
8     }
9
10    def cleanup() = {
11      for ((k, v) <- counts) {
12        emit(k, v)
13      }
14    }
15  }
```



This mapper uses a hash table to maintain the words and counts across all lines in a single split.

By aggregating tuples across map tasks, this reduces disk and memory I/O. However, a possible drawback is **increasing the memory working set** (which is proportional to the number of distinct words in a map task)

Preserving State in Map / Reduce Tasks

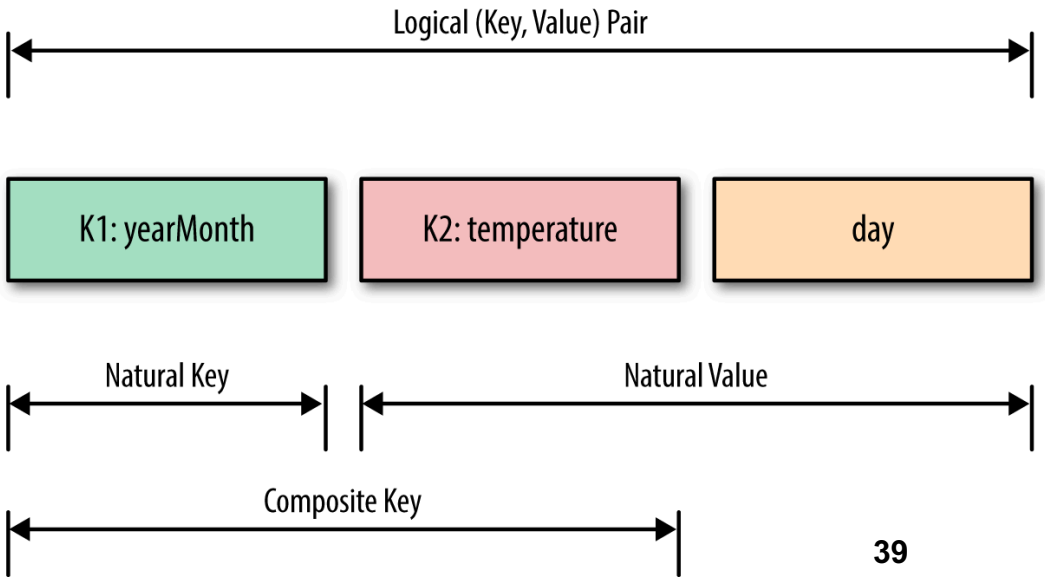


1. **MapReduce**
   a. Basic MapReduce
   b. Partition and Combiner
   c. Examples
   d. Secondary Sort
2. Hadoop File System

Source: Google

Secondary Sort

- **Problem**: each reducer's values arrive unsorted. But what if we want them to be sorted (e.g. sorted by temperature)?
- **Solution**: define a new 'composite key' as (K1, K2), where K1 is the original key ("Natural Key") and K2 is the variable we want to use to sort
  - **Partitioner**: now needs to be customized, to partition by K1 only, not (K1, K2)



Code Example: Composite Key

```
1  import org.apache.hadoop.io.Writable;
2  import org.apache.hadoop.io.WritableComparable;
3  ...
4  public class DateTemperaturePair
5     implements Writable, WritableComparable<DateTemperaturePair> {
6
7      private Text yearMonth = new Text();                  // natural key
8      private Text day = new Text();
9      private IntWritable temperature = new IntWritable(); // secondary key
10
11     ...
12
13     @Override
14     /**
15      * This comparator controls the sort order of the keys.
16      */
17     public int compareTo(DateTemperaturePair pair) {
18         int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
19         if (compareValue == 0) {
20             compareValue = temperature.compareTo(pair.getTemperature());
21         }
22         //return compareValue;     // sort ascending
23         return -1*compareValue;    // sort descending
24     }
25     ...
26 }
```

Compare by **`yearMonth`** first; if tie, compare by **`temperature`**

Code Example: Custom Partitioner

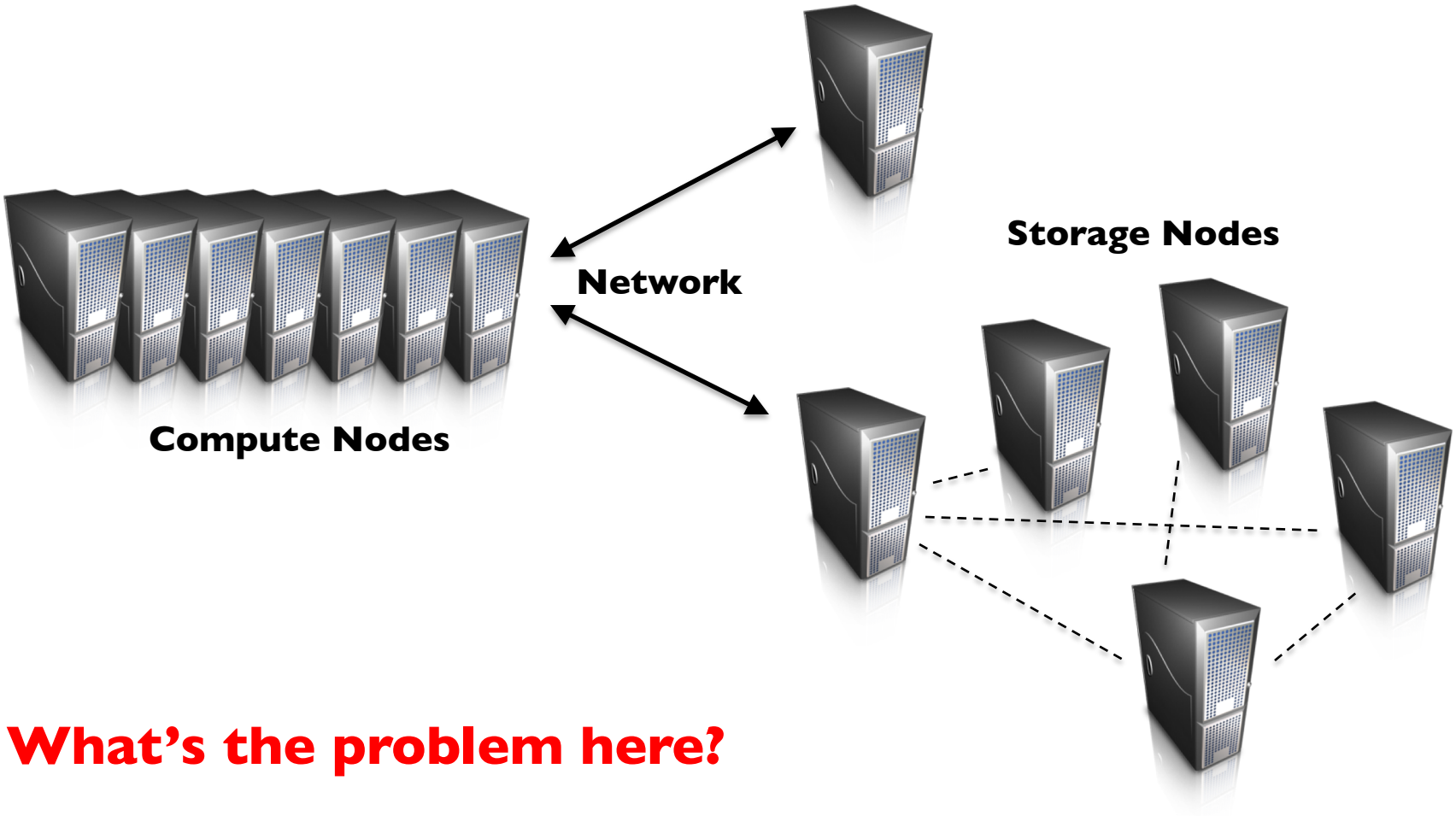
```
public class DateTemperaturePartitioner
    extends Partitioner<DateTemperaturePair, Text> {

     @Override
     public int getPartition(DateTemperaturePair pair,
                             Text text,
                             int numberOfPartitions) {
         // make sure that partitions are non-negative
         return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
      }
}
```

Partition by **`yearMonth`** only (not **`temperature`**)

1. **MapReduce**
   a. Basic MapReduce
   b. Partition and Combiner
   c. Examples
   d. Secondary Sort
2. **Hadoop File System**

Source: Google

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS / HDFS: Assumptions

- Commodity hardware instead of "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
- Files are write-once, mostly appended to
- Large streaming reads instead of random access
  - High sustained throughput over low latency

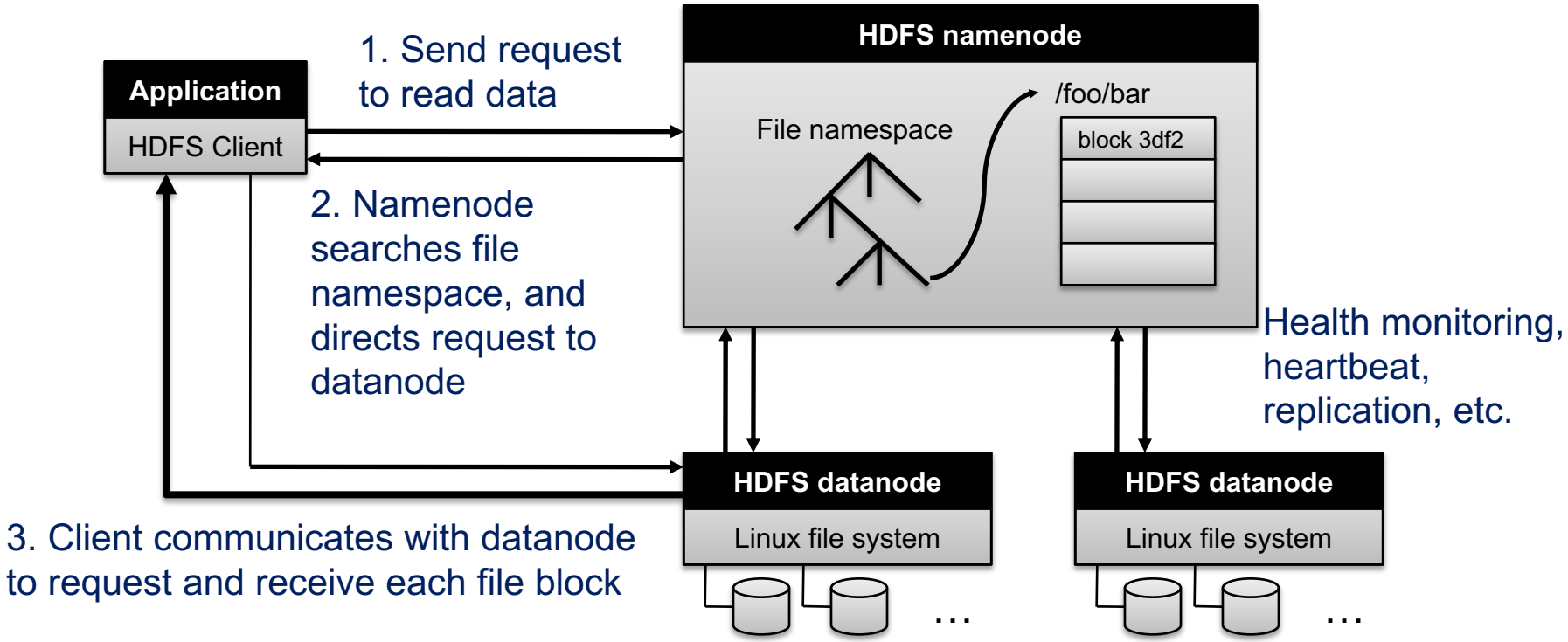
GFS slides adapted from material by (Ghemawat et al., SOSP 2003)

Design Decisions

- Files stored as chunks
  - Fixed size (64MB for GFS, 128MB for HDFS)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management

HDFS = GFS clone (same basic ideas)

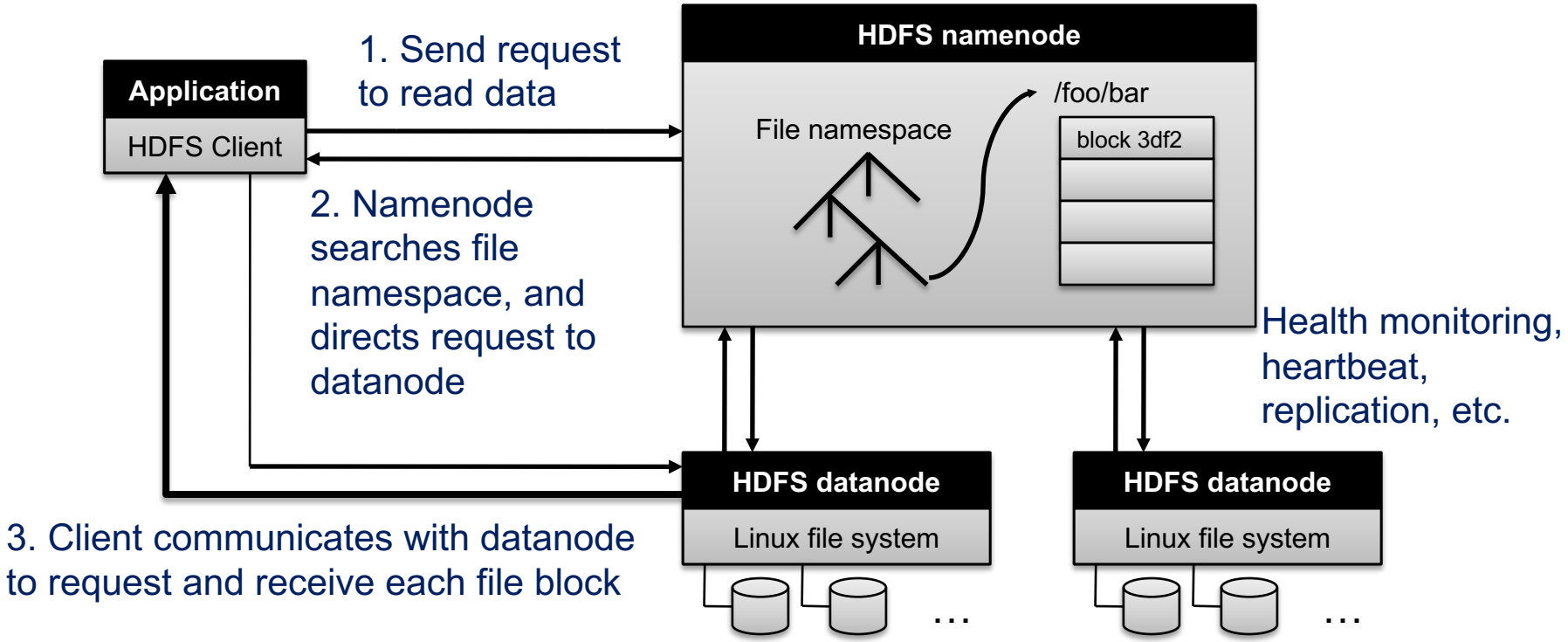
HDFS Architecture



Q: How to perform replication when writing data?

Adapted from (Ghemawat et al., SOSP 2003)

HDFS Architecture



Q: How to perform replication when writing data?

A: Namenode decides which datanodes are to be used as replicas. The 1st datanode forwards data blocks to the 1st replica, which forwards them to the 2nd replica, and so on.

Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc. Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

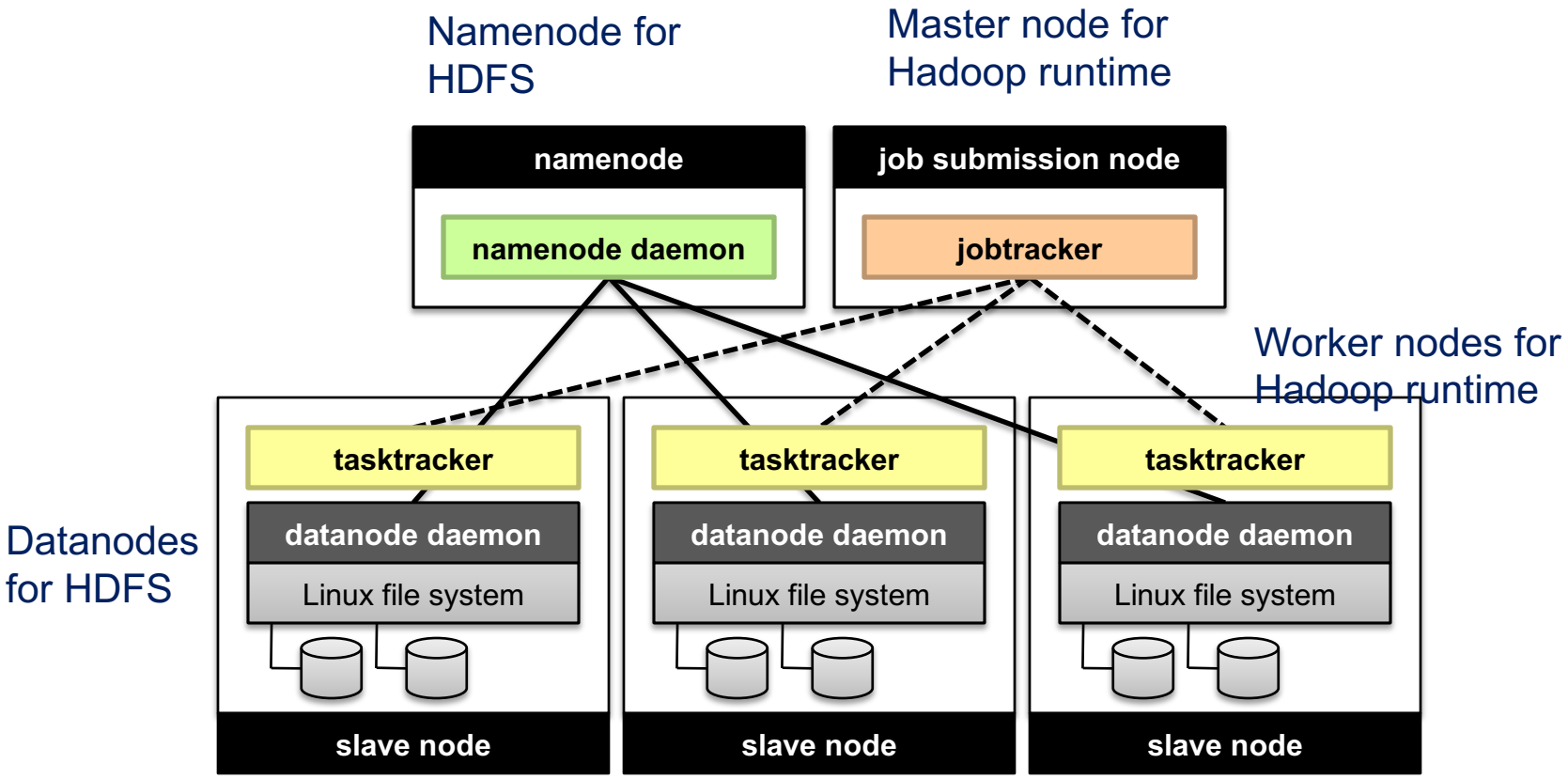
- **Q**: What if the namenode's data is lost?

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc. Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

- **Q**: What if the namenode's data is lost?
- **A**: All files on the filesystem cannot be retrieved since there is no way to reconstruct them from the raw block data. Fortunately, Hadoop provides 2 ways of improving resilience, through backups and secondary namenodes (out of syllabus, but you can refer to Resources for details)

Putting everything together…



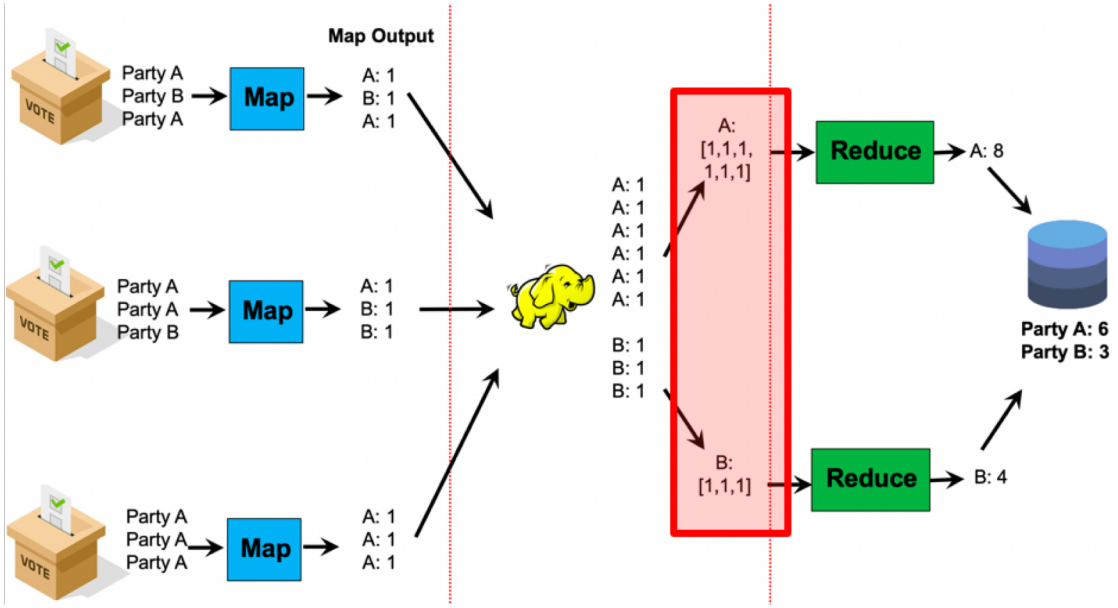
Q: Which statement(s) about Hadooop's partitioner is true?

1. The partitioner determines which keys are processed on which mappers.
2. The partitioner can improve performance when some keys are much more common than others.
3. The partitioner is run in between the map and reduce phases.

Q: Which statement(s) about Hadooop's partitioner is true?

1. The partitioner determines which keys are processed on which mappers.
2. The partitioner can improve performance when some keys are much more common than others.
3. The partitioner is run in between the map and reduce phases.

- **A: 2 and 3**



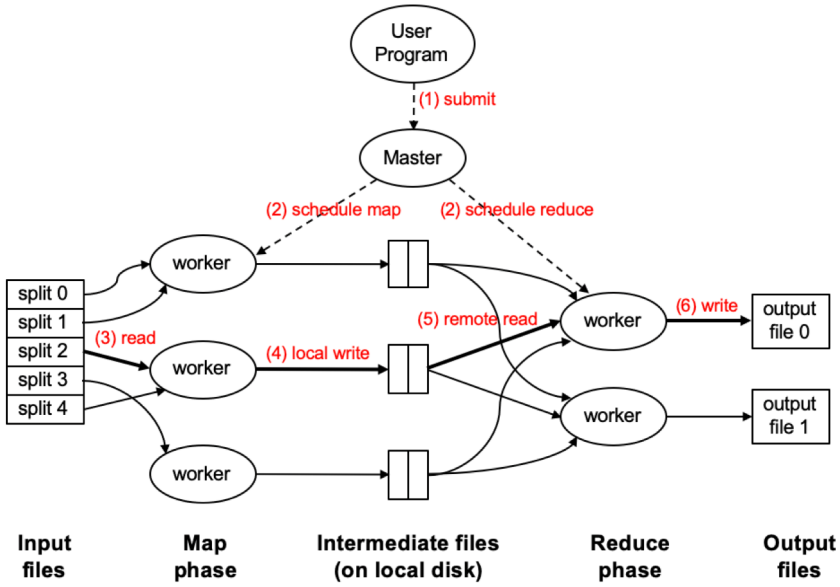
Q: True or false: the Shuffle stage of MapReduce is always run within a single node.

1. True
2. False

Q: True or false: the Shuffle stage of MapReduce is always run within a single node.

1. True
2. False

A: False (it runs in the worker nodes)



Resources

- Hadoop: The Definitive Guide (by Tom White)
- Hadoop Wiki
  - Introduction
    - http://wiki.apache.org/lucene-hadoop/
  - Getting Started
    - http://wiki.apache.org/lucene-hadoop/GettingStartedWithHadoop
  - Map/Reduce Overview
    - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
    - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
  - Eclipse Environment
    - http://wiki.apache.org/lucene-hadoop/EclipseEnvironment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/
- YARN
  - https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

Acknowledgement

- Slides adopted/revised from
  - Jimmy Lin, http://lintool.github.io/UMD-courses/bigdata-2015-Spring/
  - He Bingsheng
- Some slides are also adopted/revised from
  - Claudia Hauff, TU Delft: https://chauff.github.io/
  - Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. Mining of Massive Datasets (2nd ed.). Cambridge University Press. http://www.mmds.org/