CS4225/CS5425 Big Data Systems for Data Science

MapReduce

Bingsheng He
School of Computing
National University of Singapore
hebs@comp.nus.edu.sg

School of Computing

Learning Outcomes

- Abstractions for MapReduce
- Programming with MapReduce

1. MapReduce
   a. **Basic MapReduce**
   b. Partition and Combiner
   c. Examples

Source: Google

Typical Big Data Problem

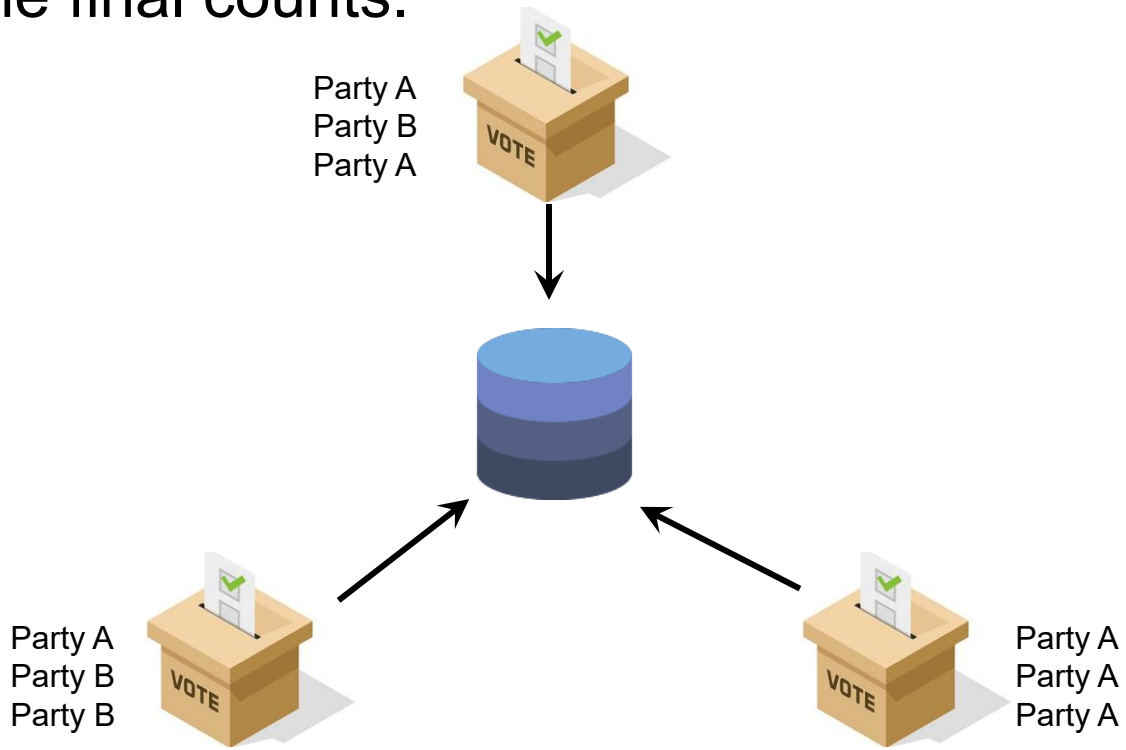
- Iterate over a large number of records
- Extract something of interest from each — Map
- Shuffle and sort intermediate results — Shuffle
- Aggregate intermediate results — Reduce
- Generate final output

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

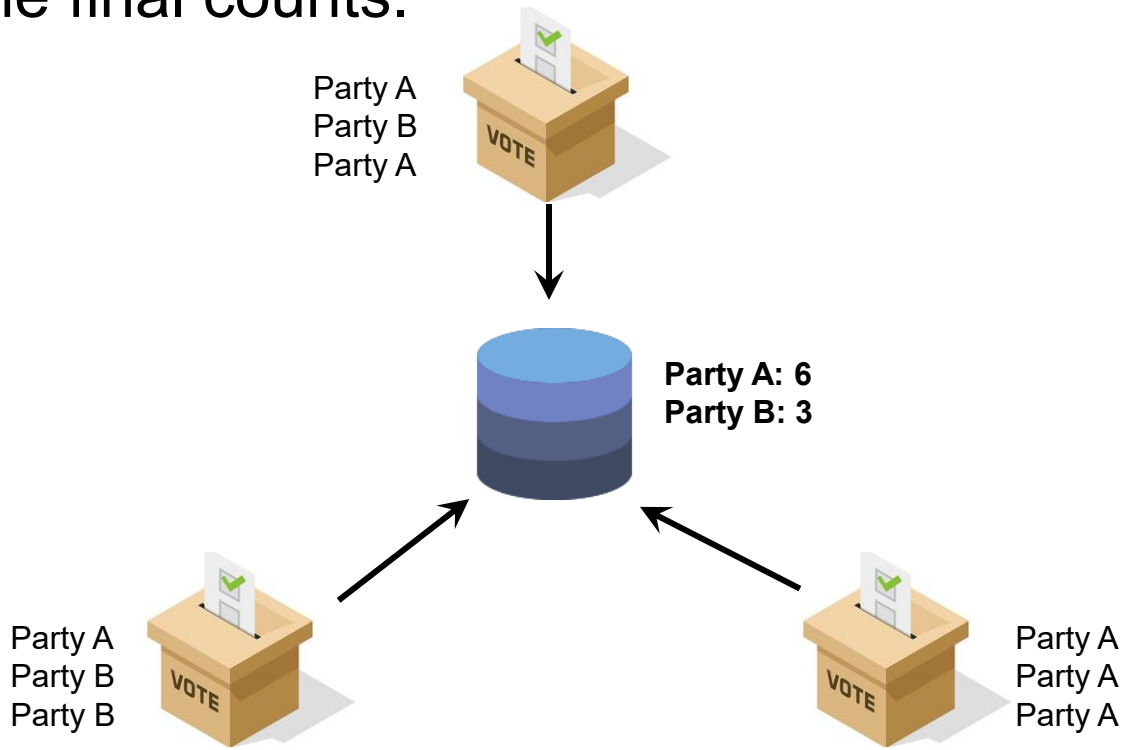
Basic Example: Tabulating Election Results from Multiple Polling Sites

- Imagine you are hired to develop the software for Singapore's election counting system.
- You need to **aggregate** vote counts from multiple stations into the final counts.



Basic Example: Tabulating Election Results from Multiple Polling Sites

- Imagine you are hired to develop the software for Singapore's election counting system.
- You need to **aggregate** vote counts from multiple stations into the final counts.



Tabulating Election Results: MapReduce

Map

Shuffle

Reduce

Party A
Party B
Party A

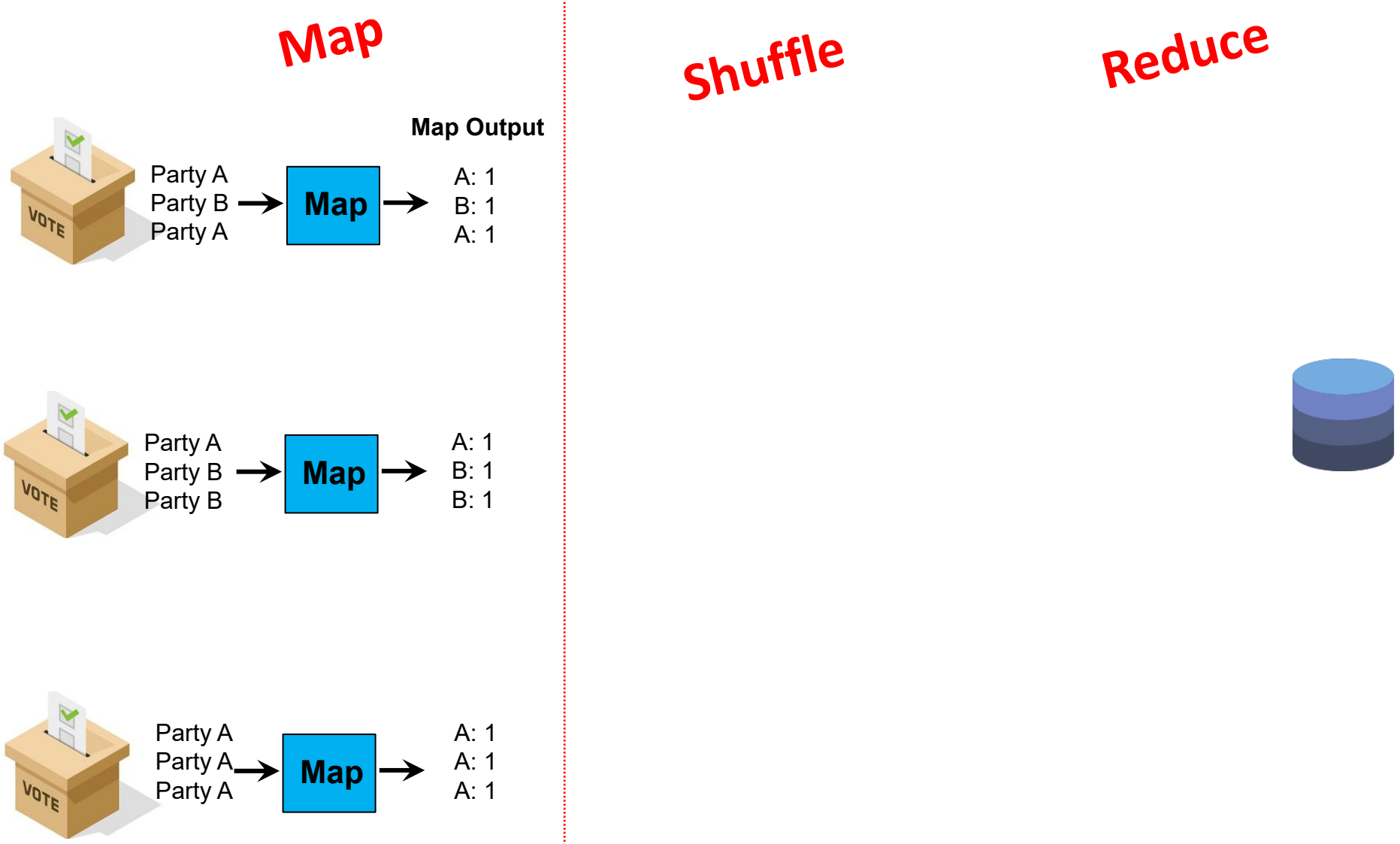
1 Input
"**split**"

Party A
Party B
Party B

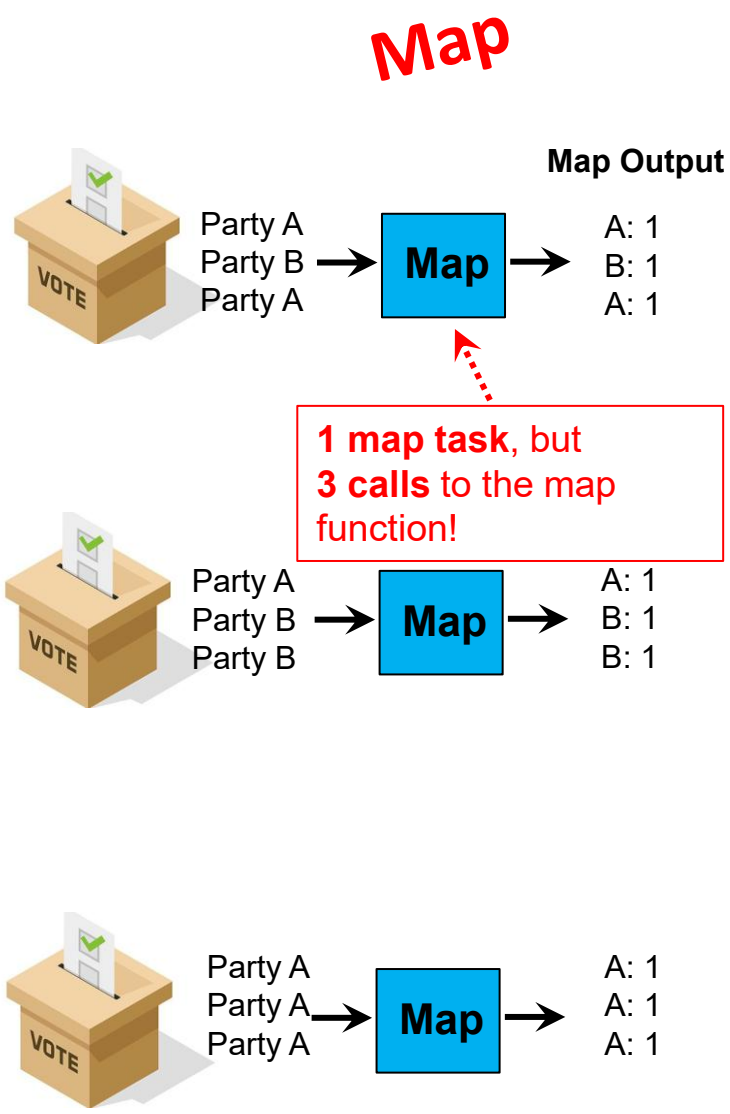
Party A
Party A
Party A

Party A: 6
Party B: 3

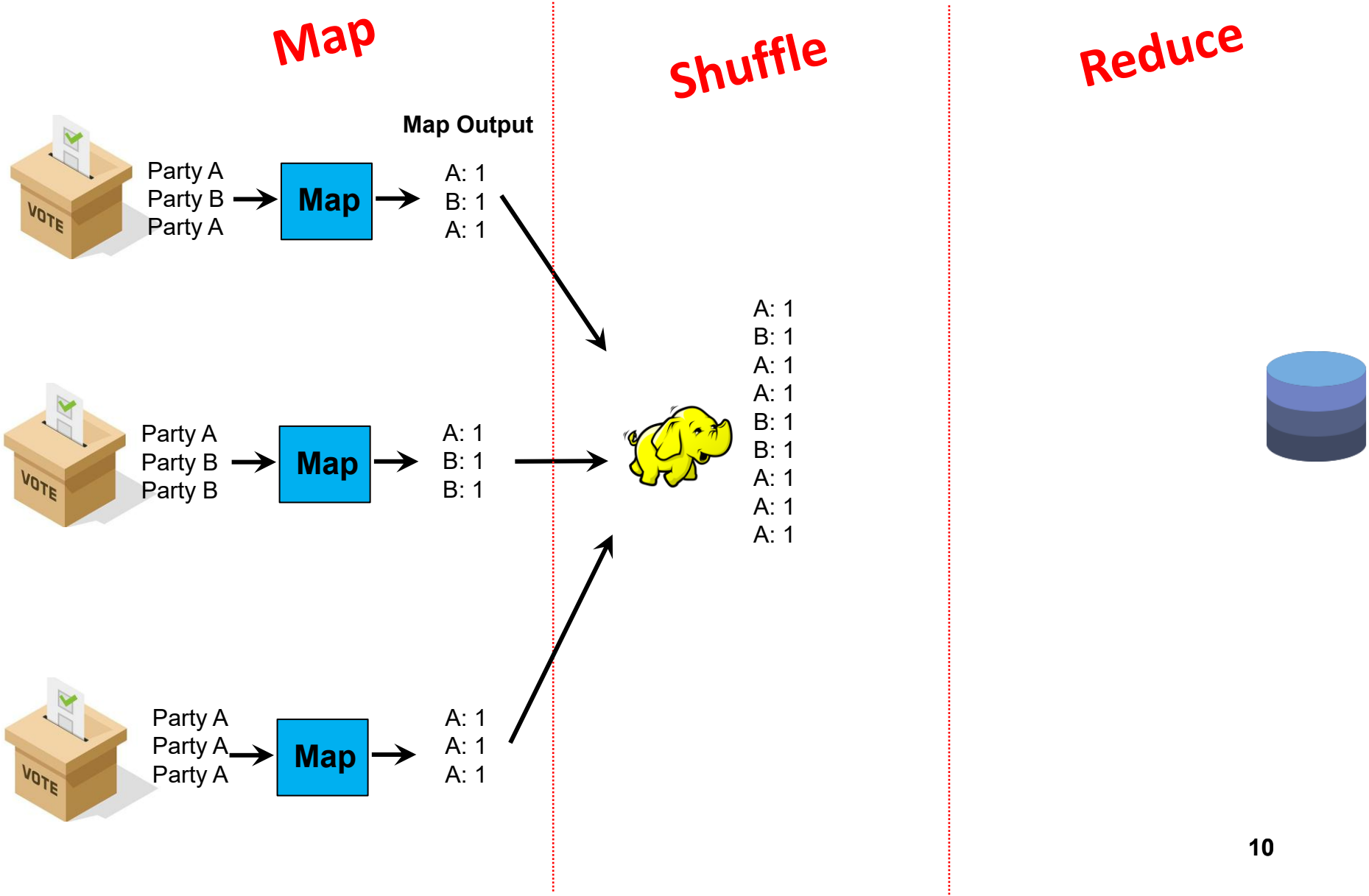
Tabulating Election Results: MapReduce



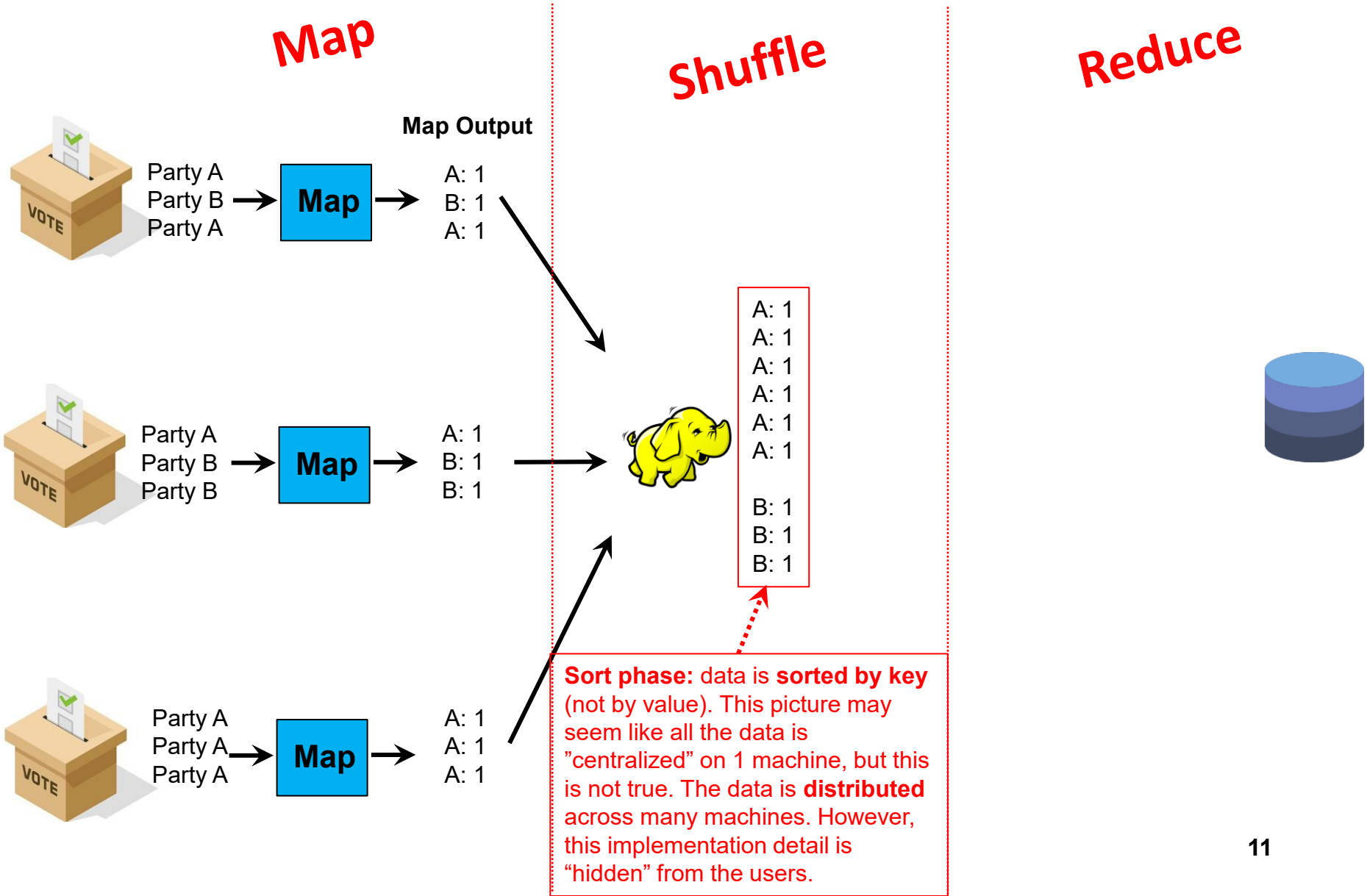
Tabulating Election Results: MapReduce



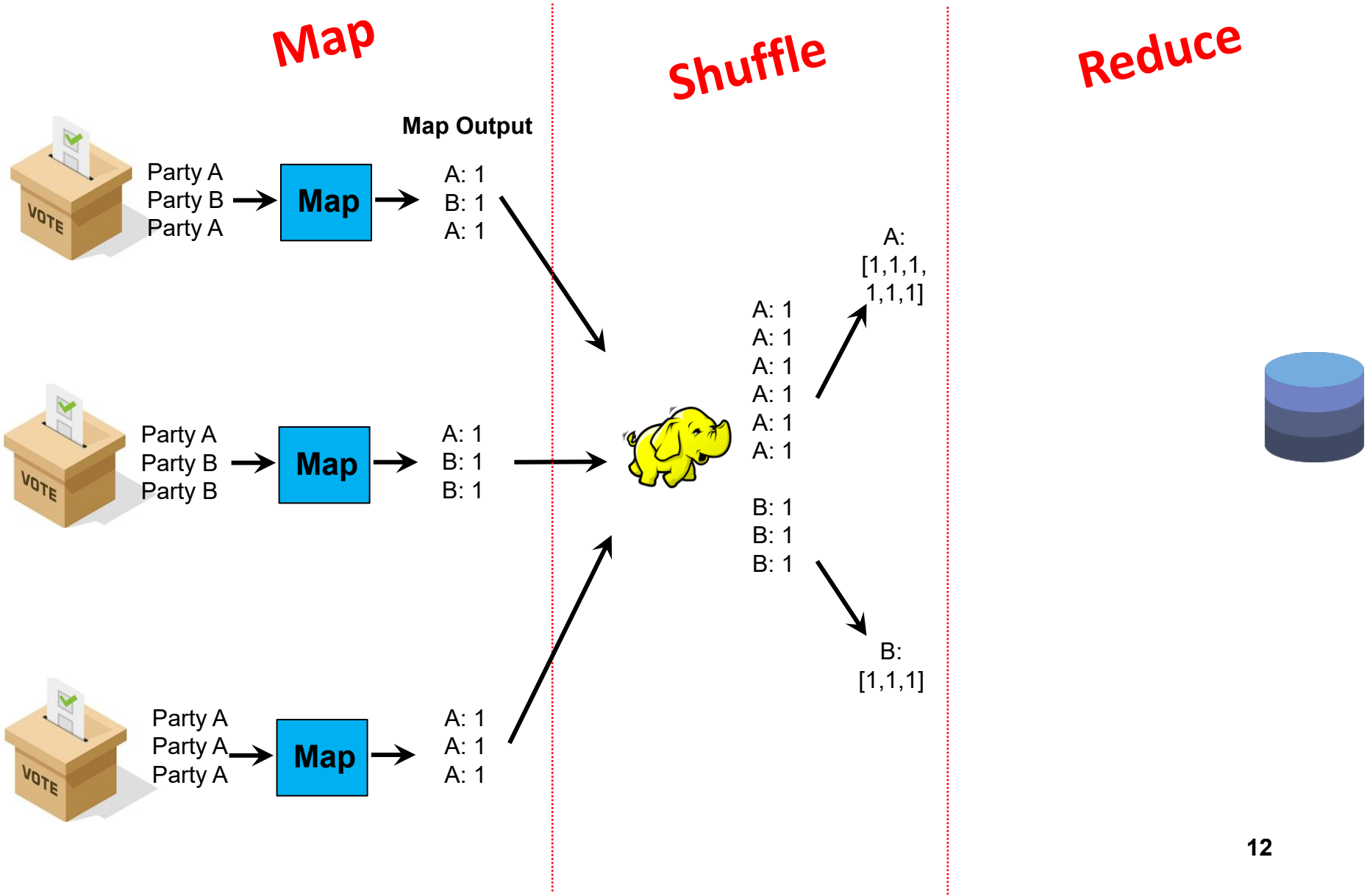
Tabulating Election Results: MapReduce



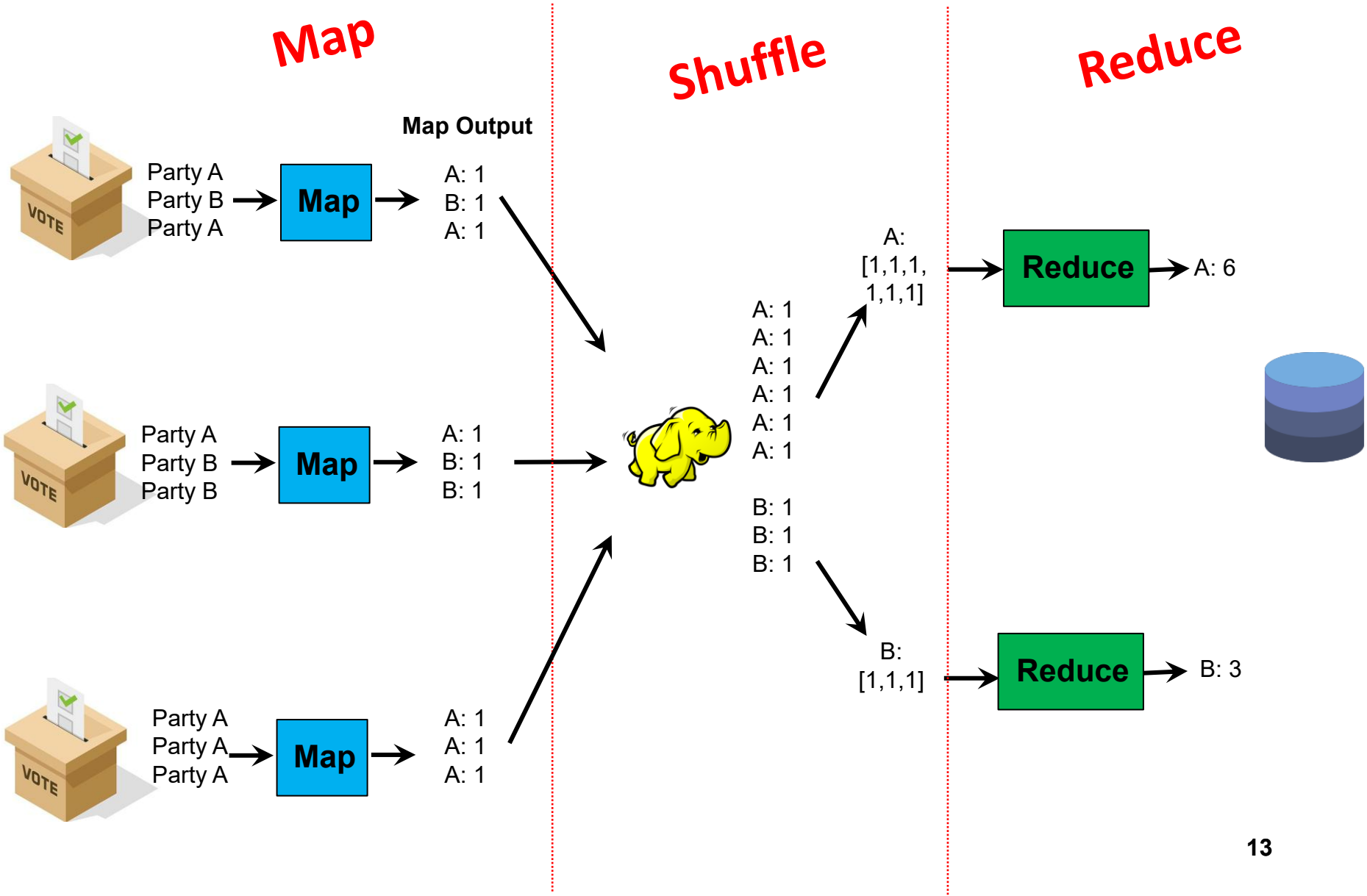
Tabulating Election Results: MapReduce



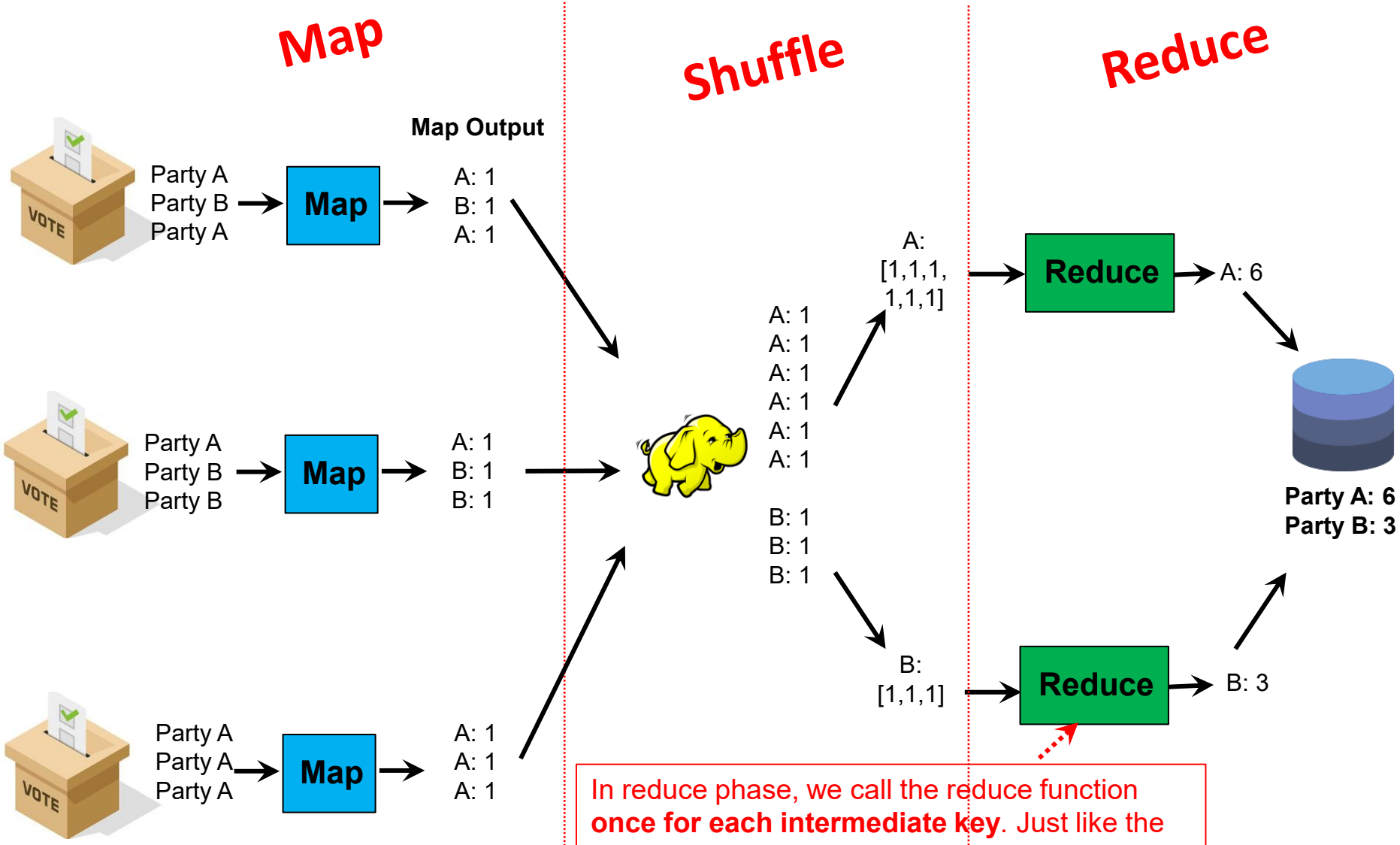
Tabulating Election Results: MapReduce



Tabulating Election Results: MapReduce



Tabulating Election Results: MapReduce



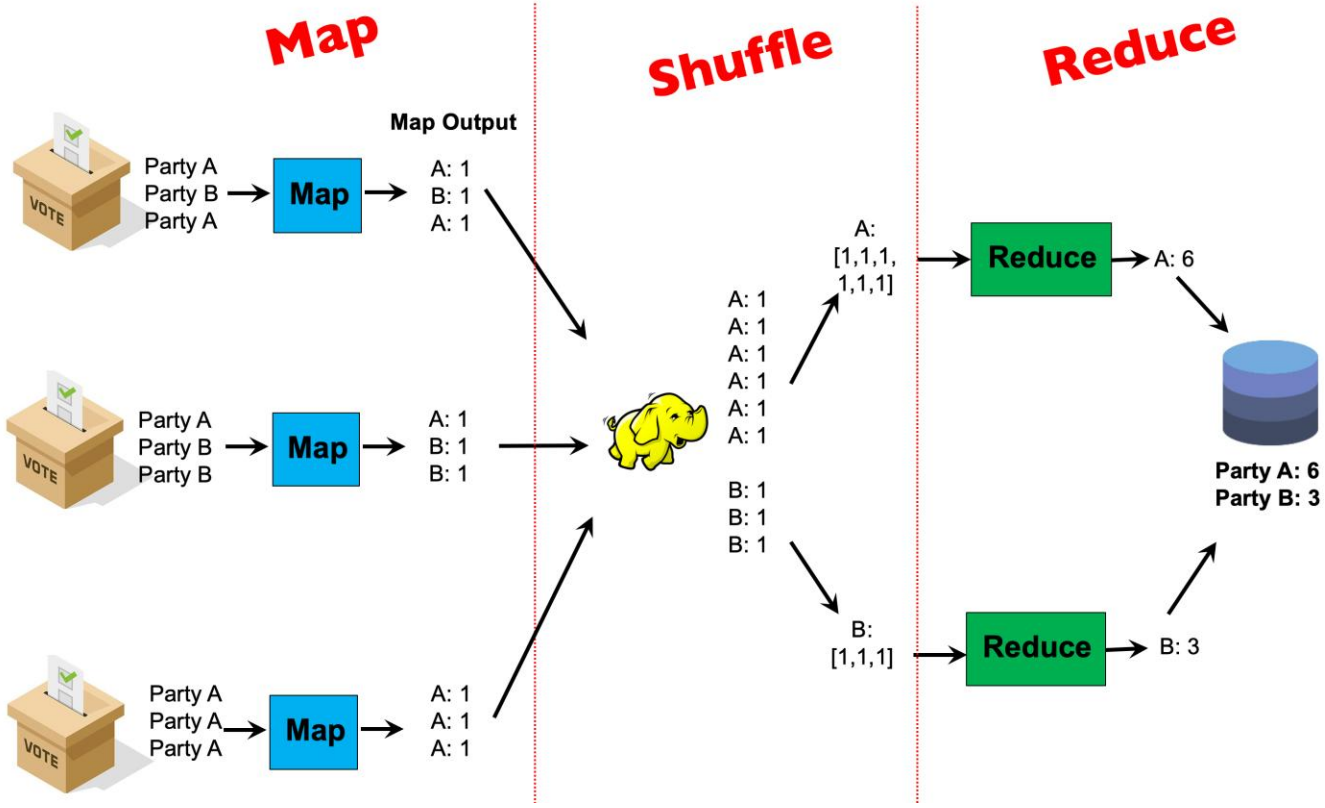
Writing MapReduce Programs

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

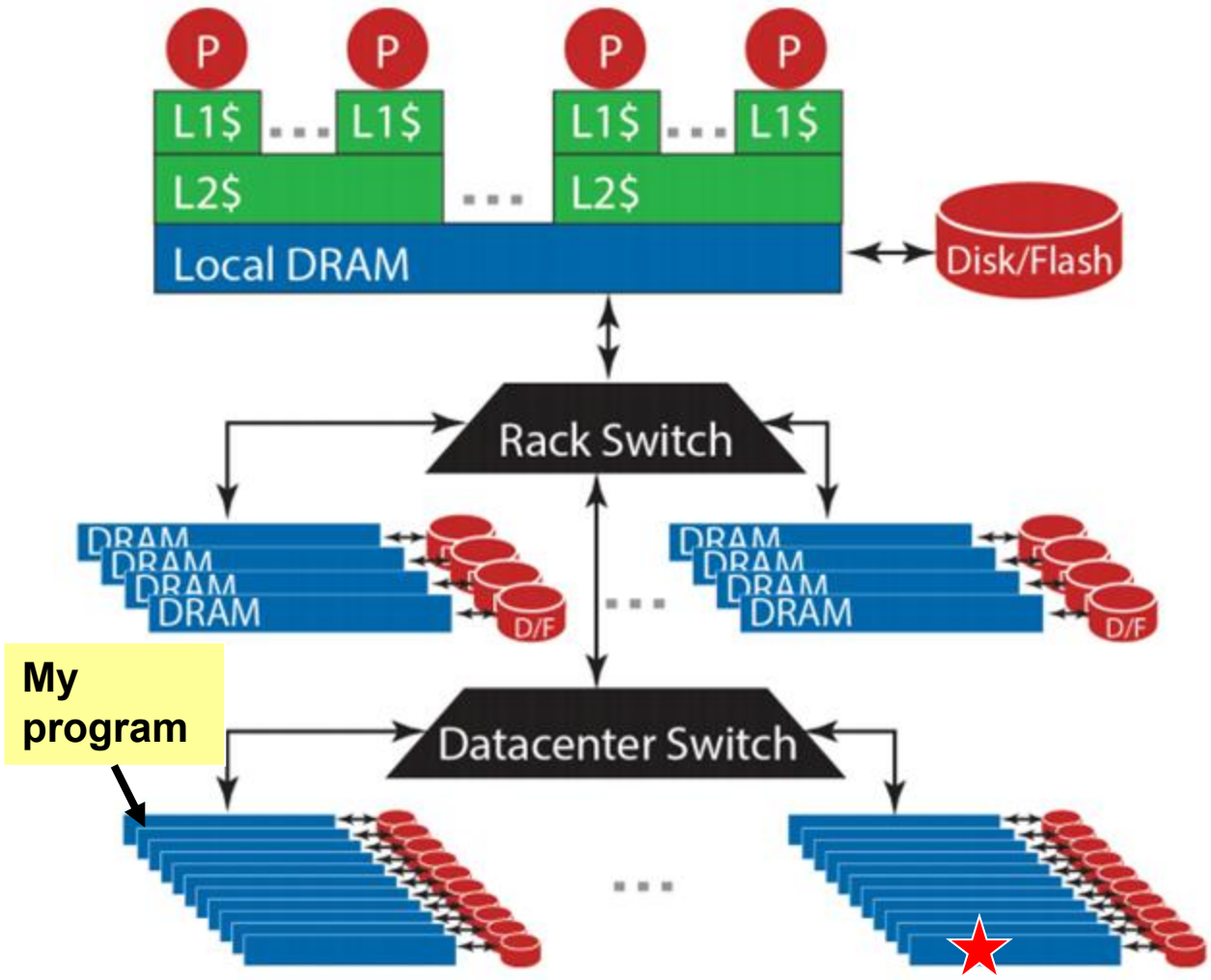
  map $(k_1, v_1) \rightarrow \text{List}(k_2, v_2)$

  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 reduce task



[Recap] Data Accesses within Data Center (But in Different Racks)

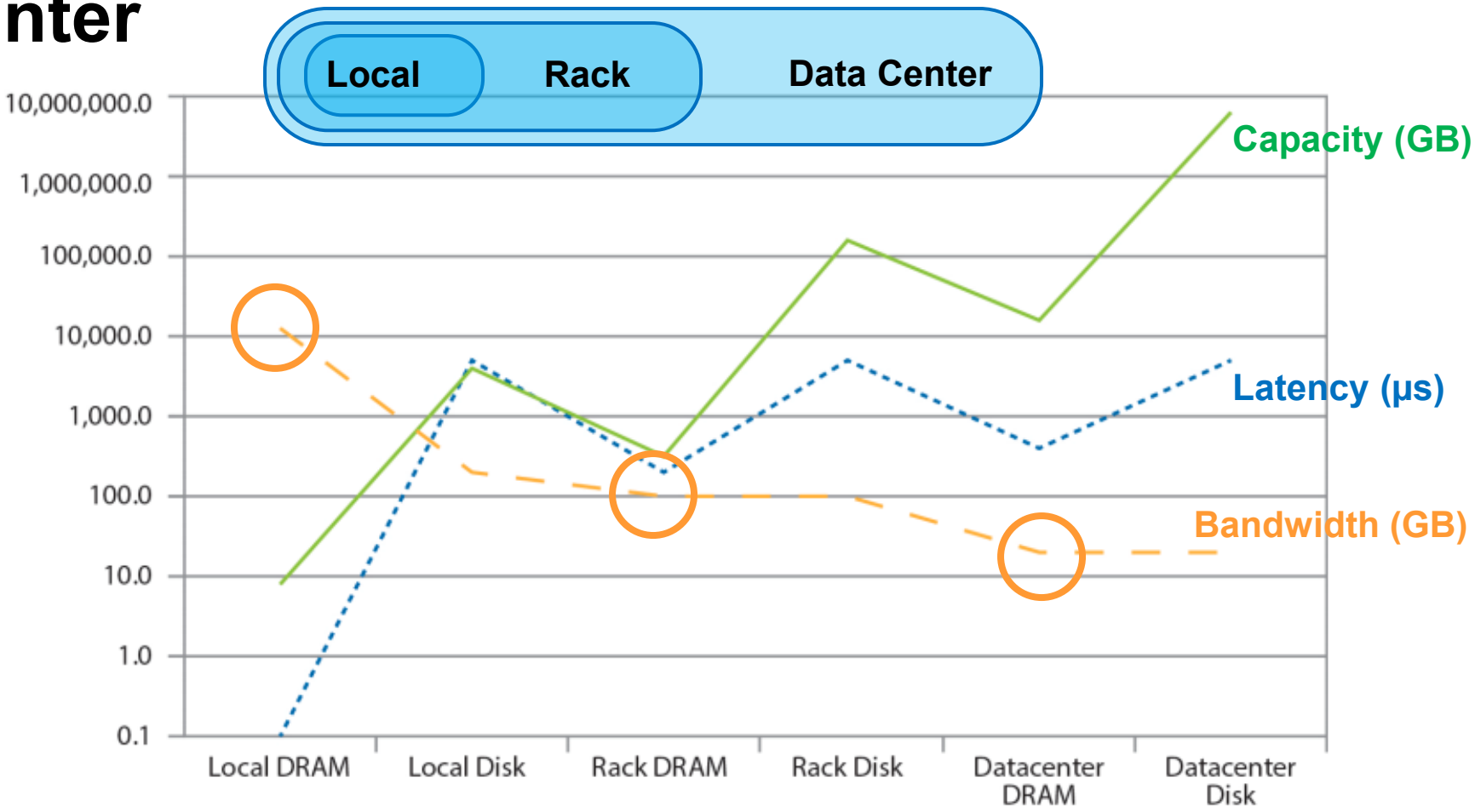


Datacenter switch (per port):
- Latency=500us
- Bandwidth=10MB/sec

Accessing the DRAM on the another machine in different rack:

- **Latency= ~(500*2+300*2)=1600 us**
- **Bandwidth= ~10MB/sec**

[Recap] Speed of Moving Data Around Data Center



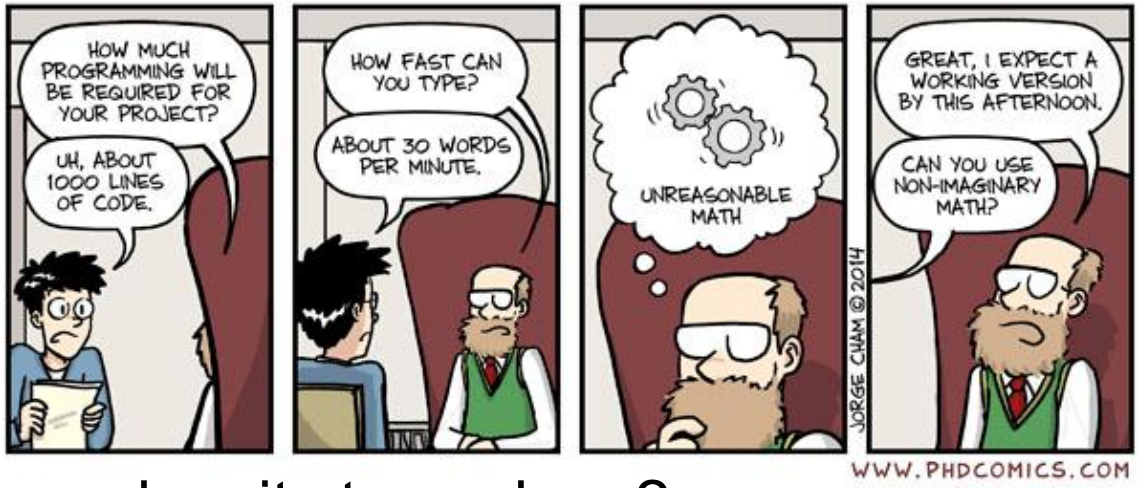
5. **Costs increase over the storage hierarchy**: **bandwidth decreases** as we go from Local to Rack to Datacenter.

Source: Barroso and Urs Hölzle (2013)

[Recap] “Big Ideas” of Massive Data Processing in Data Centers

- Scale “out”, not “up”
  - scale ‘out’ = combining many cheaper machines; scale ‘up’= increasing the power of each individual machine
  - Also called ’horizontal’ vs ‘vertical’ scaling
- Seamless scalability
  - E.g. if processing a certain dataset takes 100 machine hours, ideal scalability is to use a cluster of 10 machines to do it in about 10 hours.
- Move processing to the data
  - Clusters have limited bandwidth: we should move the task to the machine where the data is stored
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable

[Recap] Challenges



- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die/fail?

[Recap] The datacenter *is* the computer

- It's all about the right level of abstraction
  - Moving beyond the single machine architecture
  - What's the "instruction set" (or "API") of the datacenter computer?
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
  - No need to explicitly worry about reliability, fault tolerance, etc.
- Separating the *what* from the *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

[Recap] Writing MapReduce Programs

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

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

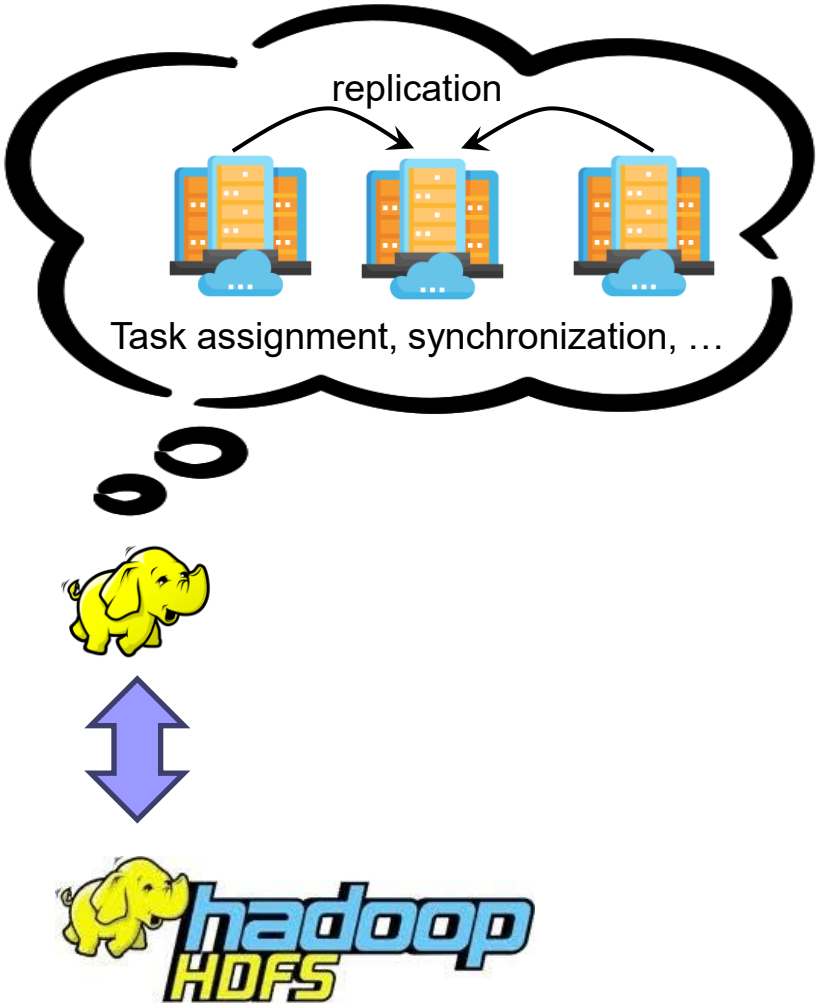
  reduce $(k_2, \text{List}(v_2)) \rightarrow \text{List}(k_3, v_3)$

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

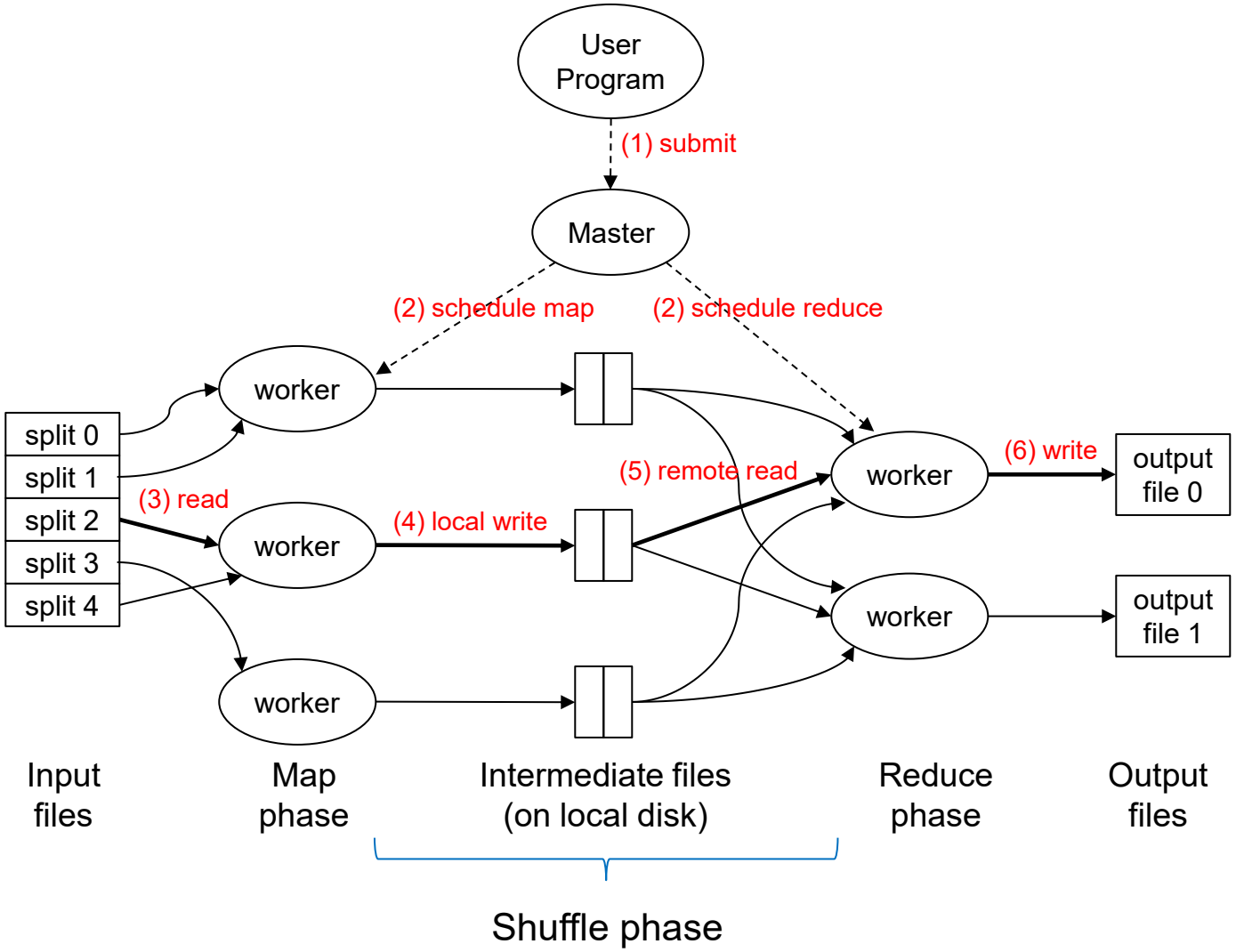
- The execution framework handles challenging issues of executing these functions over a large cluster

MapReduce Execution Framework

- Handles **scheduling**
  - Assigns workers to map and reduce tasks
- Handles **shuffle phase**
  - Data is shuffled and sorted in between map and reduce phases
- Handles **synchronization**
  - Executes tasks in appropriate order
- Handles **faults**
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system, HDFS (later)

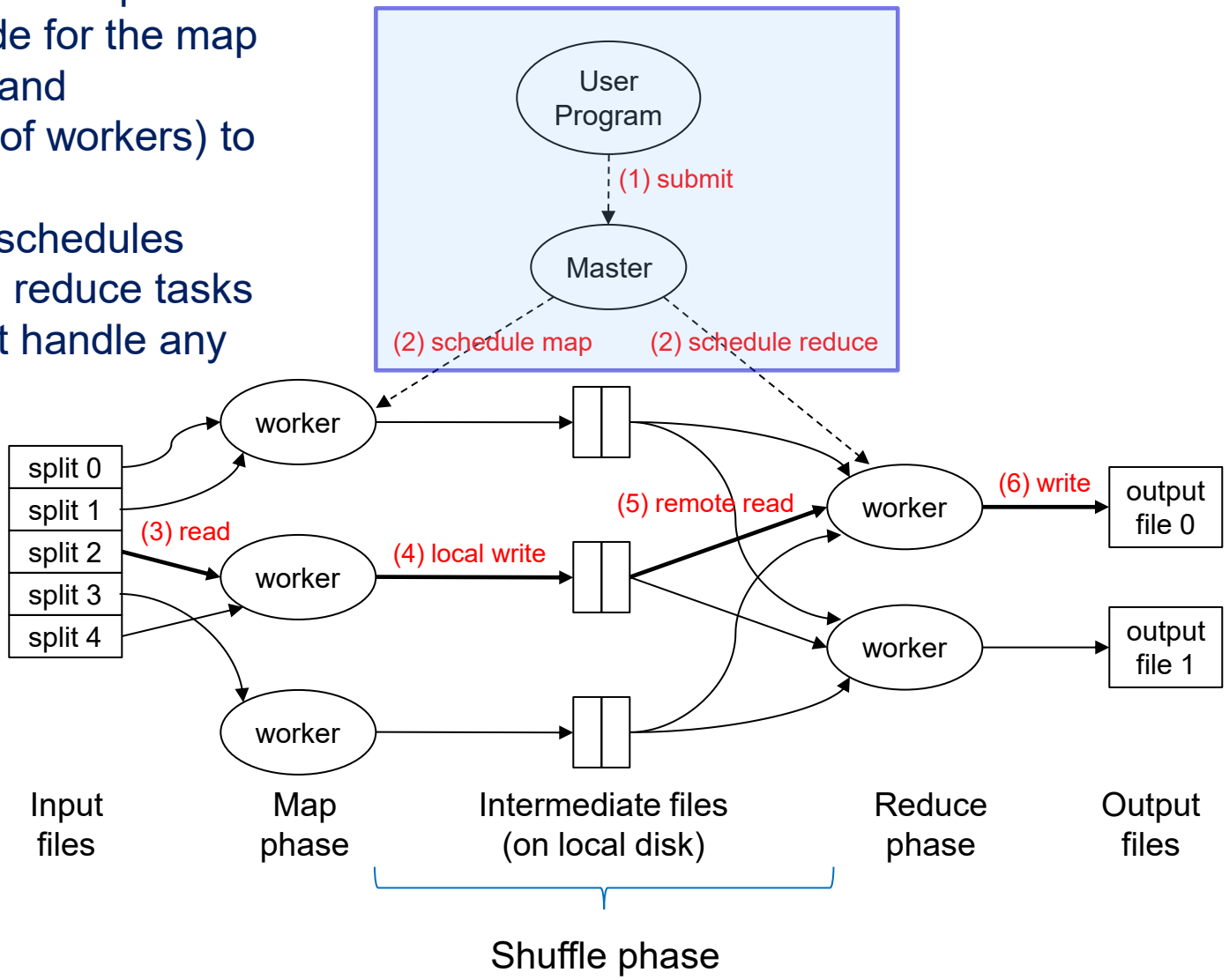


MapReduce Implementation



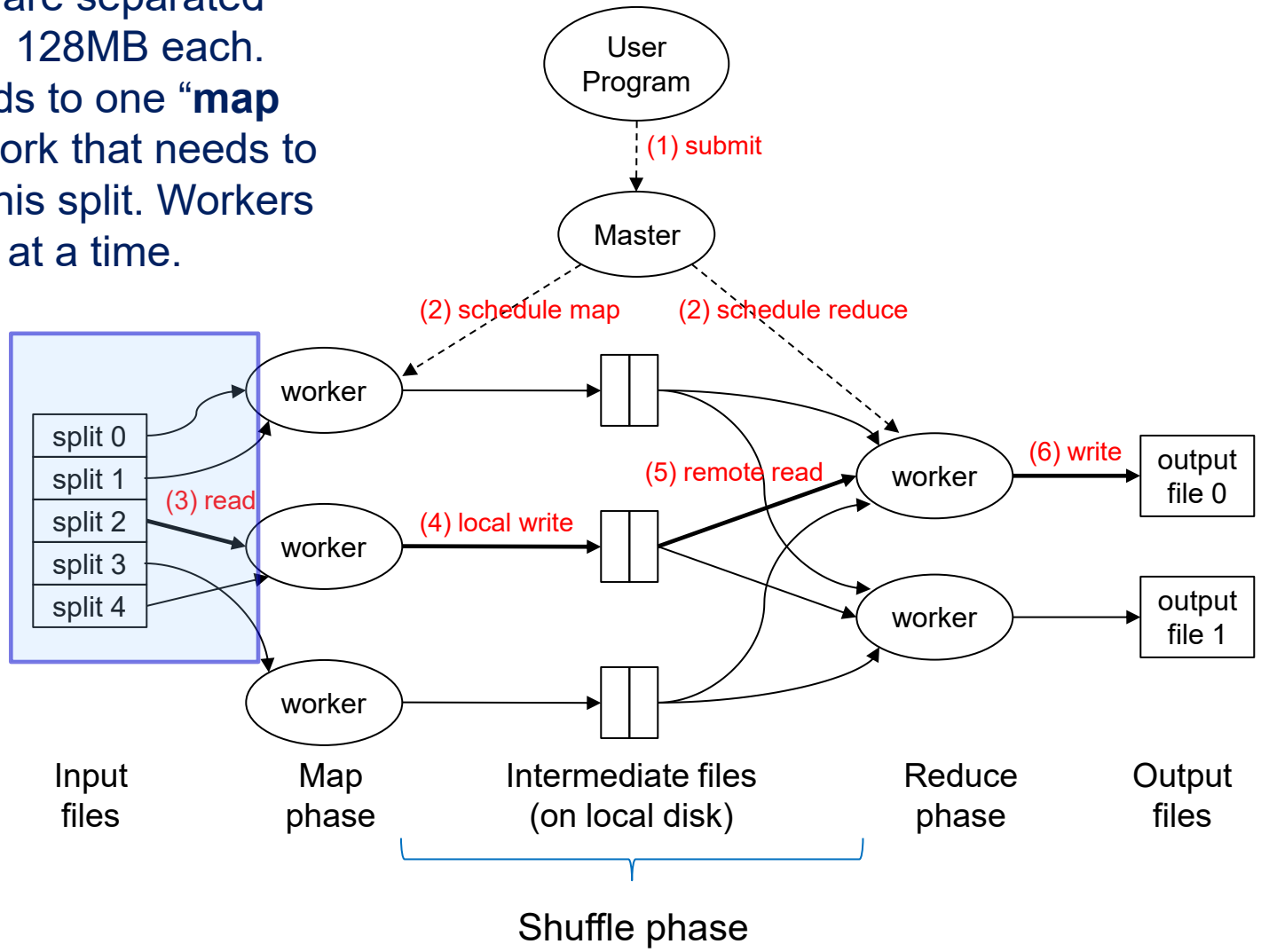
MapReduce Implementation

(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)



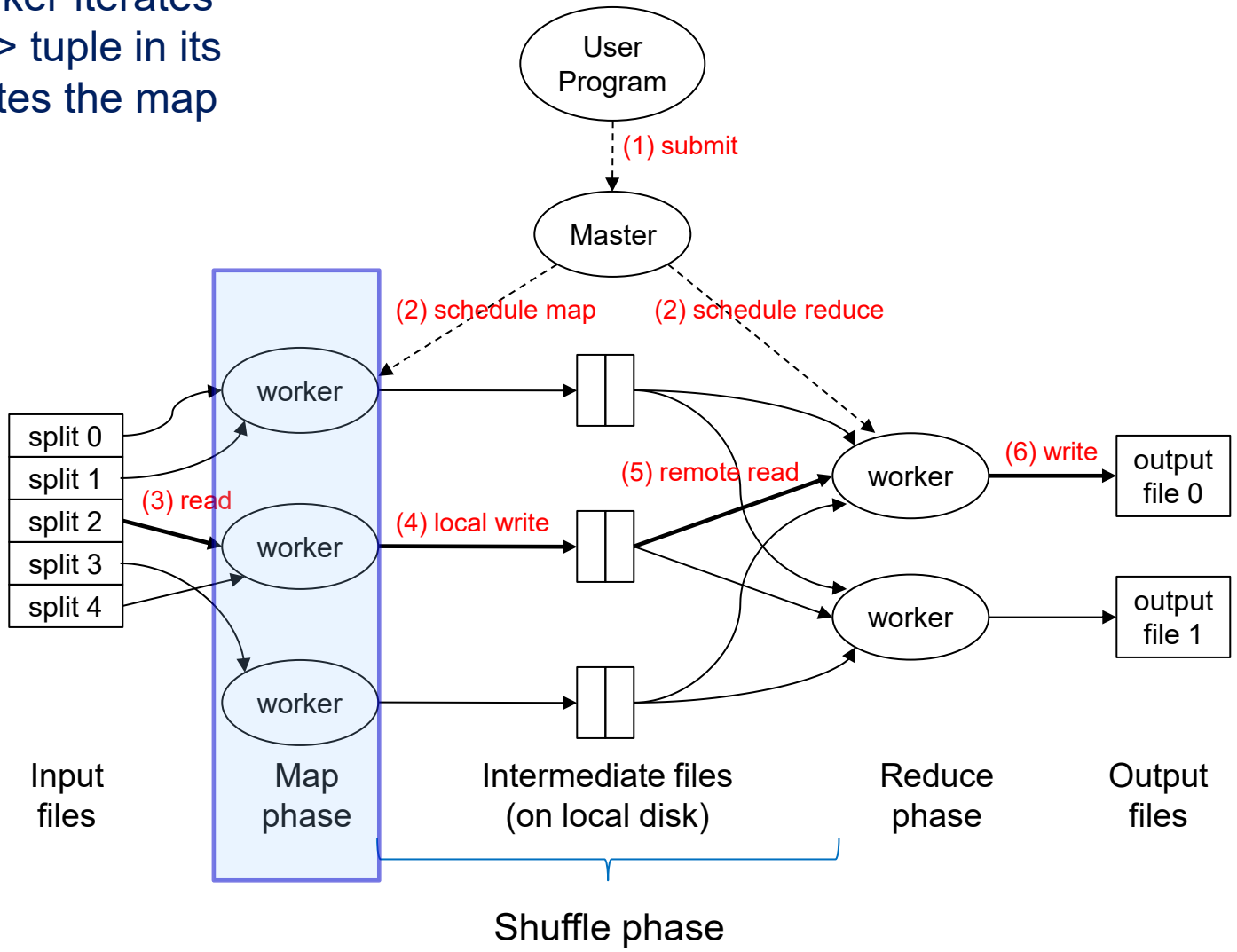
MapReduce Implementation

(3) **Read**: Input files are separated into "splits" of around 128MB each. Each split corresponds to one "**map task**", which is the work that needs to be done to process this split. Workers execute map tasks 1 at a time.



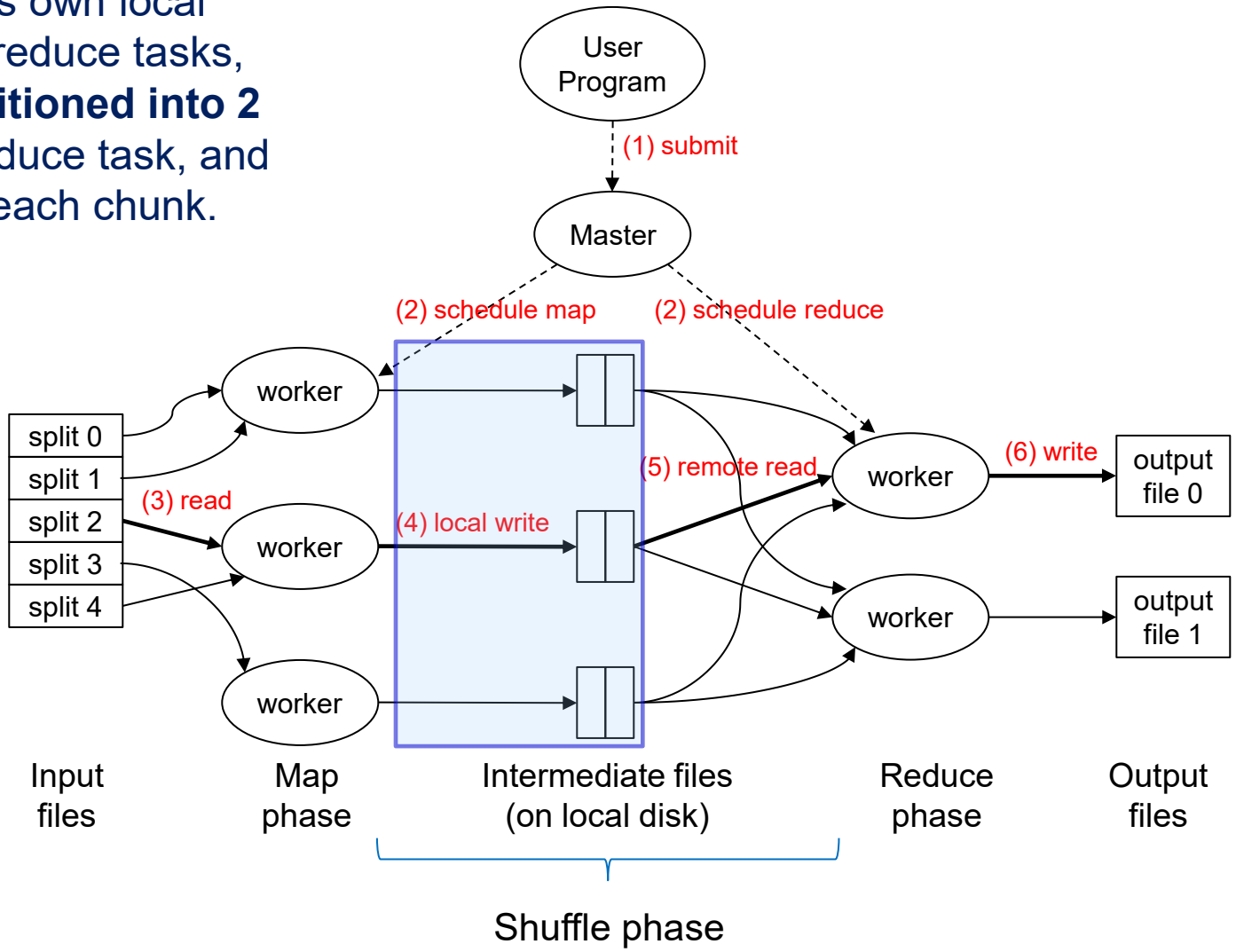
MapReduce Implementation

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



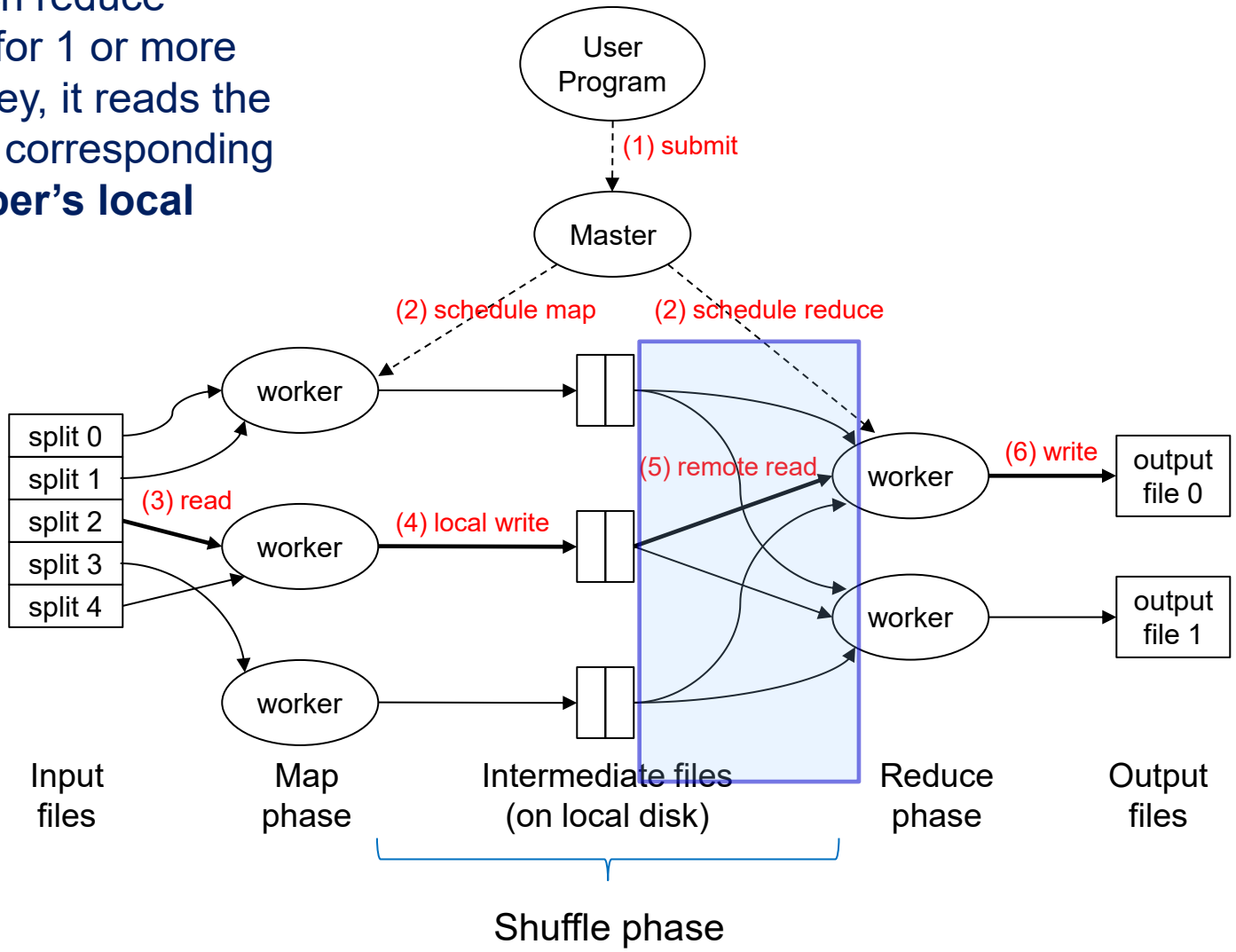
MapReduce Implementation

(4) **Local write**: Each worker writes the outputs of the map function to intermediate files on its own local disk. Here we have 2 reduce tasks, so these files are **partitioned into 2 chunks,** 1 for each reduce task, and **sorted by key** within each chunk.



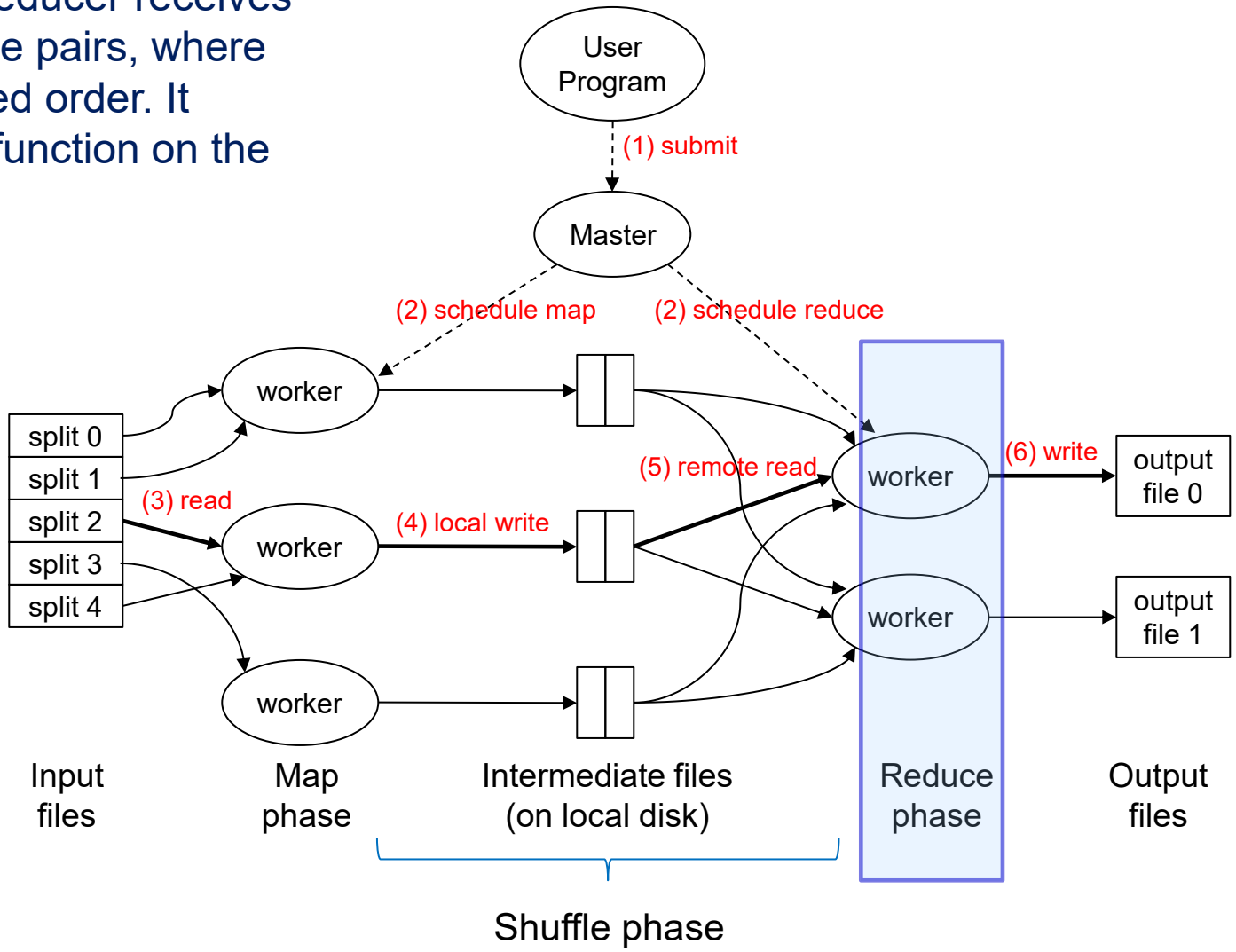
MapReduce Implementation

(5) **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**.



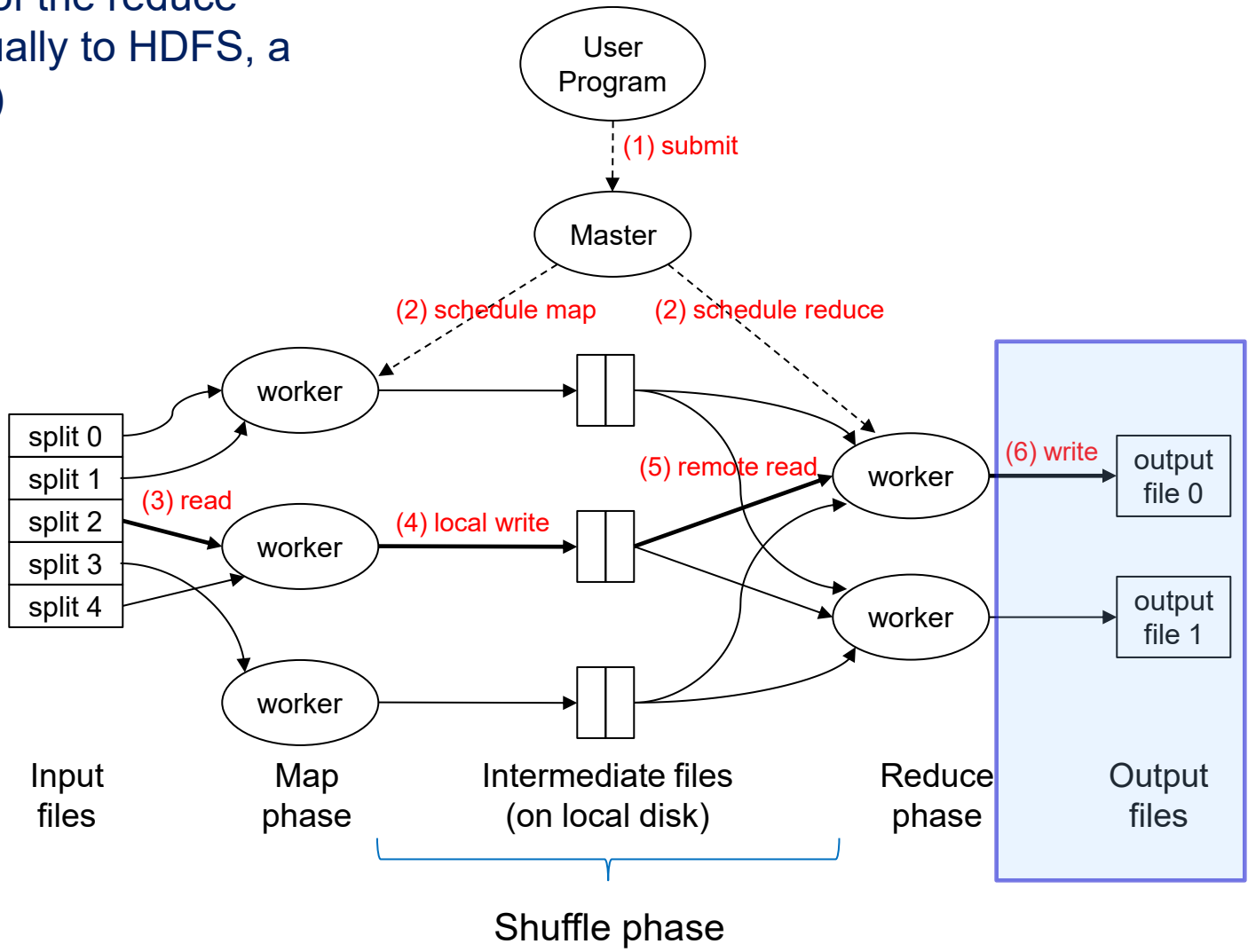
MapReduce Implementation

Reduce phase: The reducer receives all its needed key value pairs, where the keys arrive in sorted order. It computes the reduce function on the values of each key.



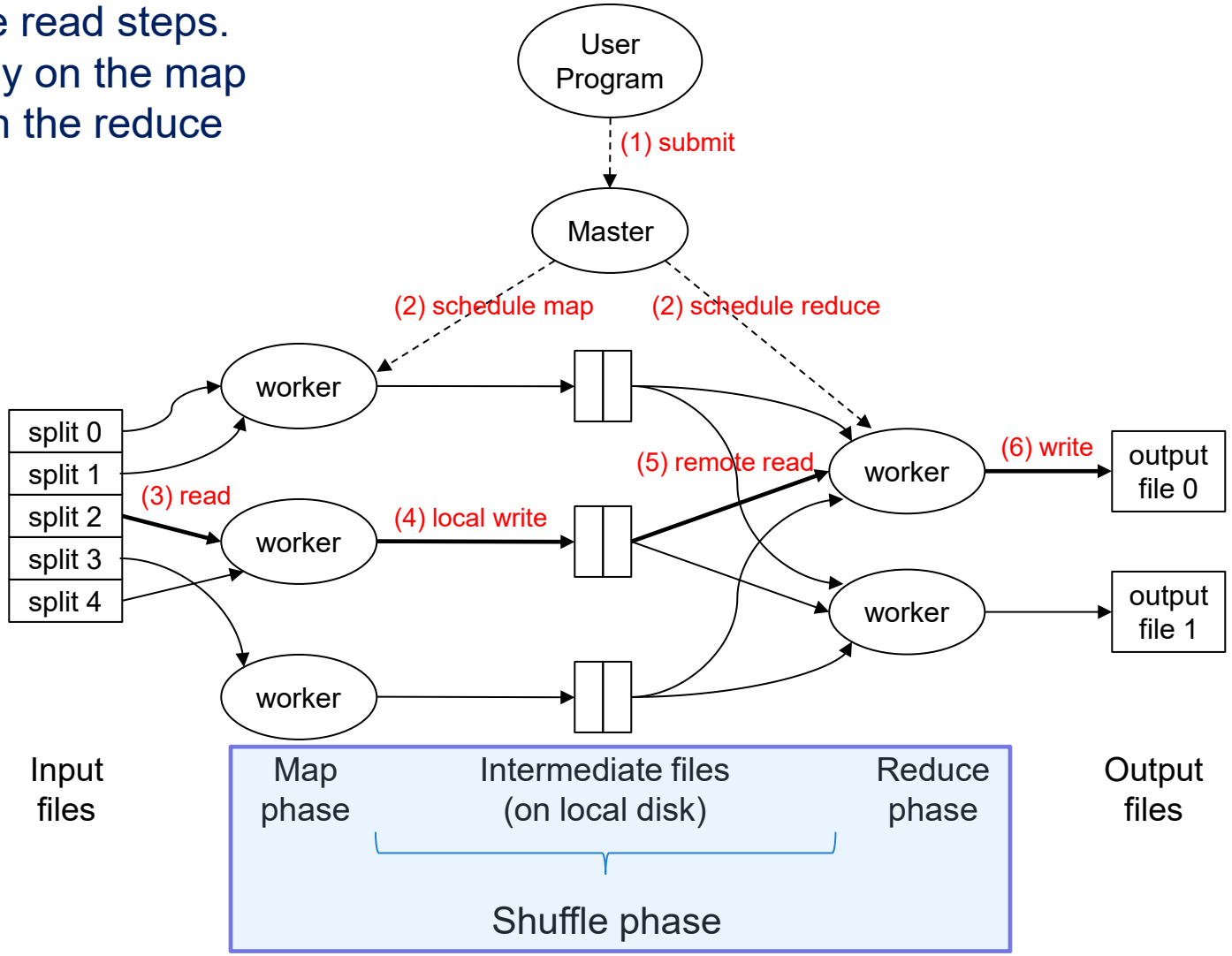
MapReduce Implementation

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

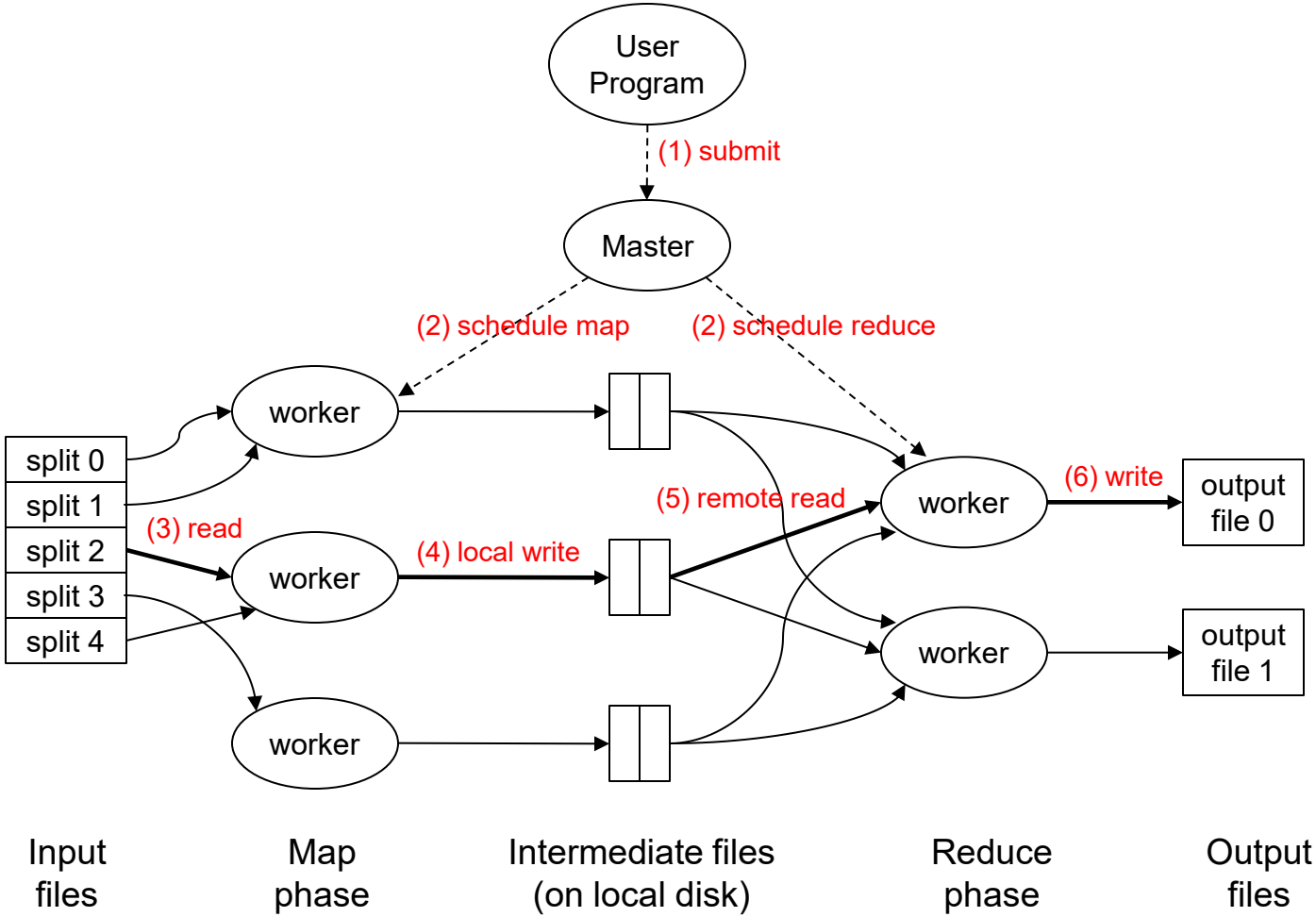


MapReduce Implementation

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.



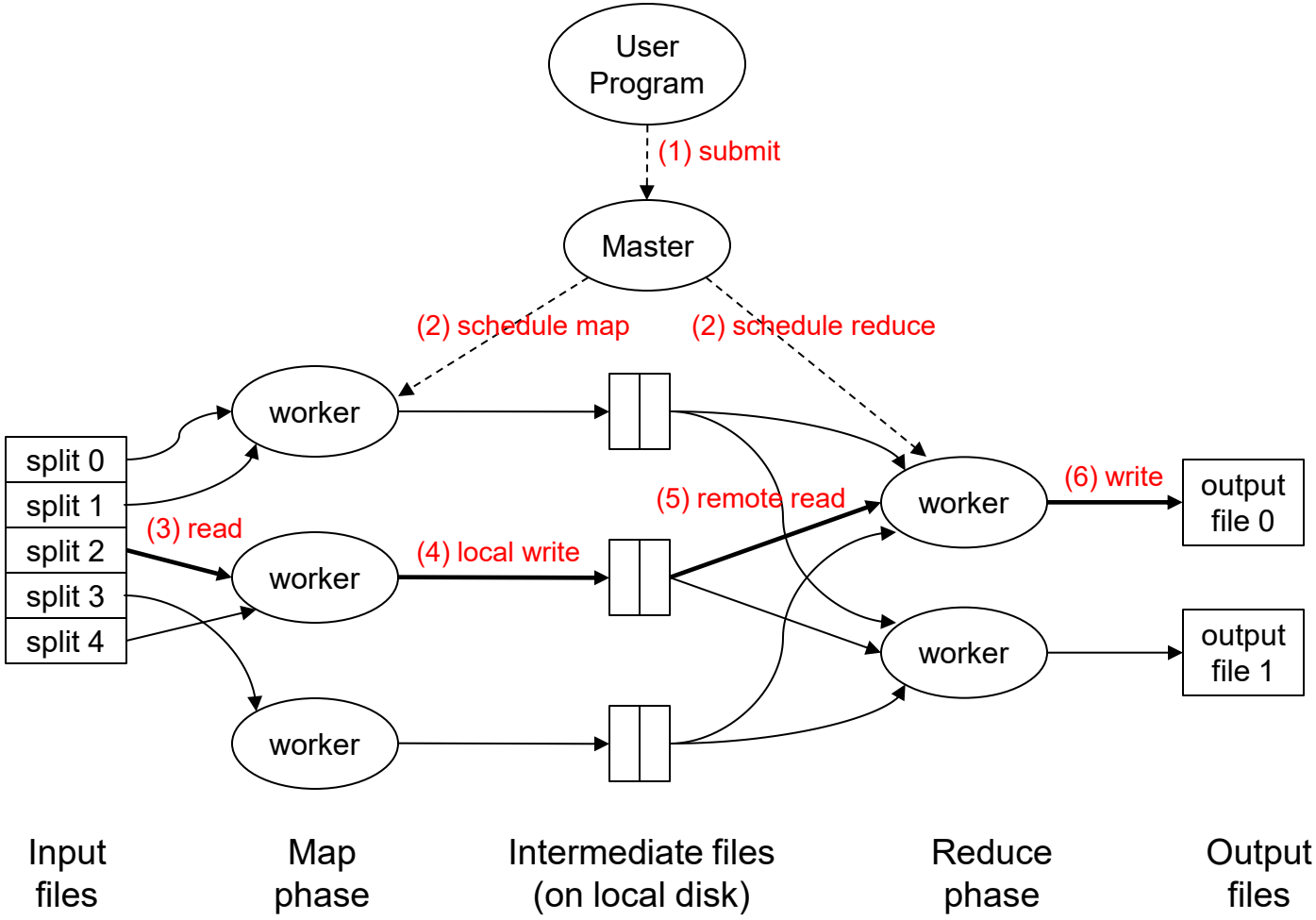
MapReduce Implementation



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

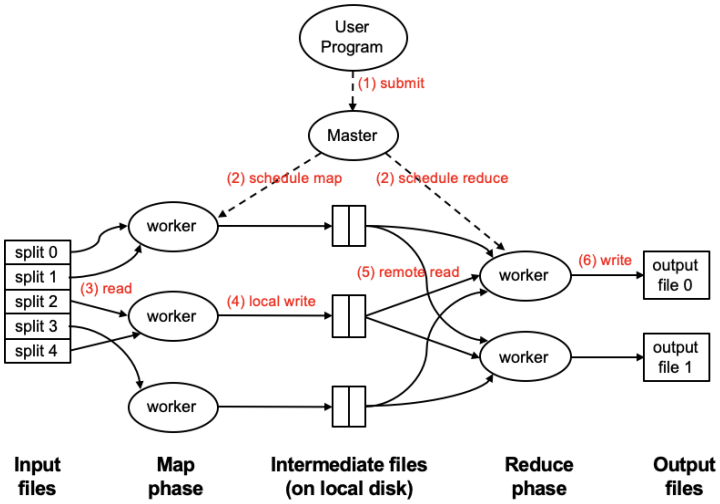


Q: What disadvantages are there if the size of each split (or chunk) is too big or small?
A: Too big: limited parallelism. Too small: high overhead (master node may be overwhelmed by scheduling work)

Adapted from (Dean and Ghemawat, OSDI 2004)

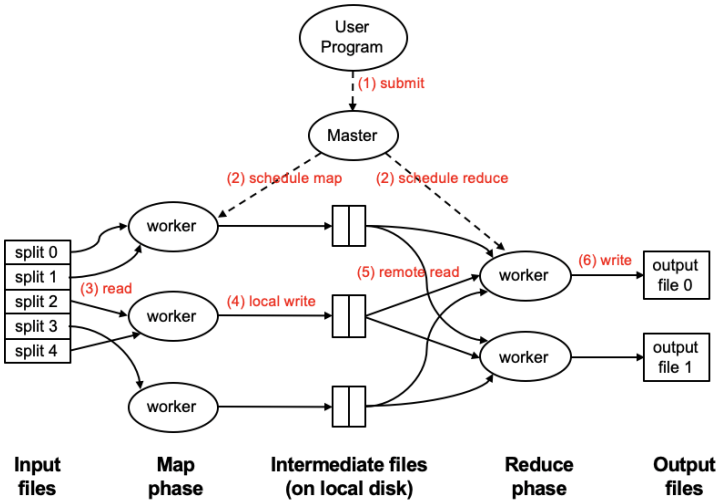
Important Clarifications: Workers, Map Task, Map Function?

- 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)



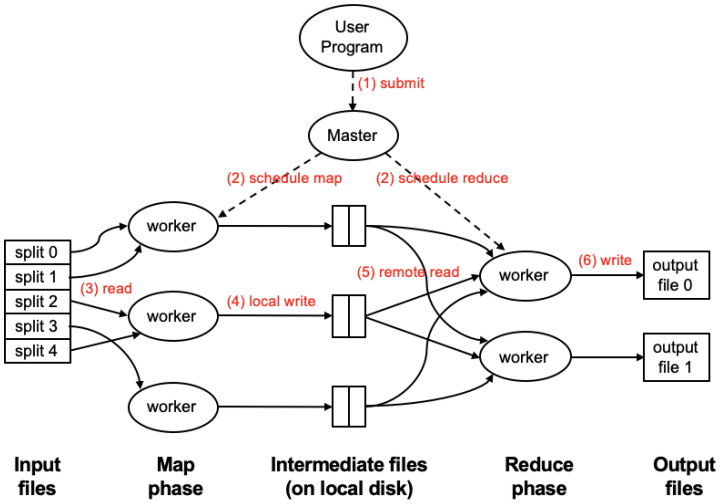
Important Clarifications: Workers, Map Task, Map Function?

- In this diagram there are 5 map tasks, but only 3 workers.
- There are 2 reduce tasks (since the intermediate files are partitioned into 2), and 2 reduce workers.



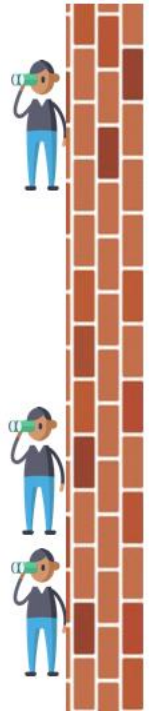
Important Clarifications: Workers, Map Task, Map Function?

- "**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.
- The same is true for reduce.



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

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

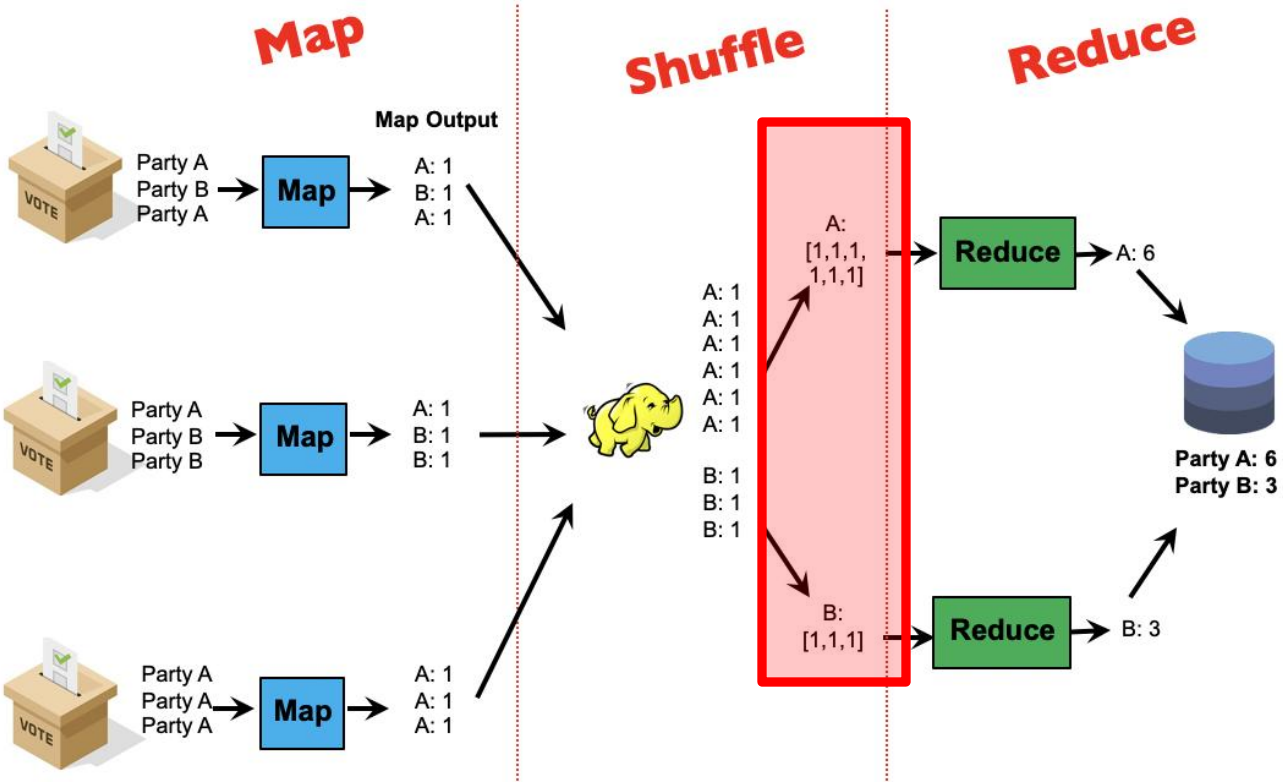
- The execution framework handles everything else…

- Not quite…usually, programmers optionally also specify

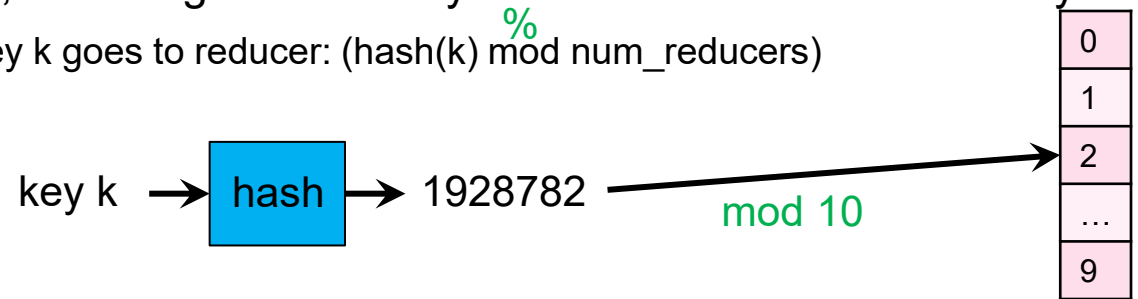
  partition, and combine functions

  - These are an optional optimization to reduce network traffic

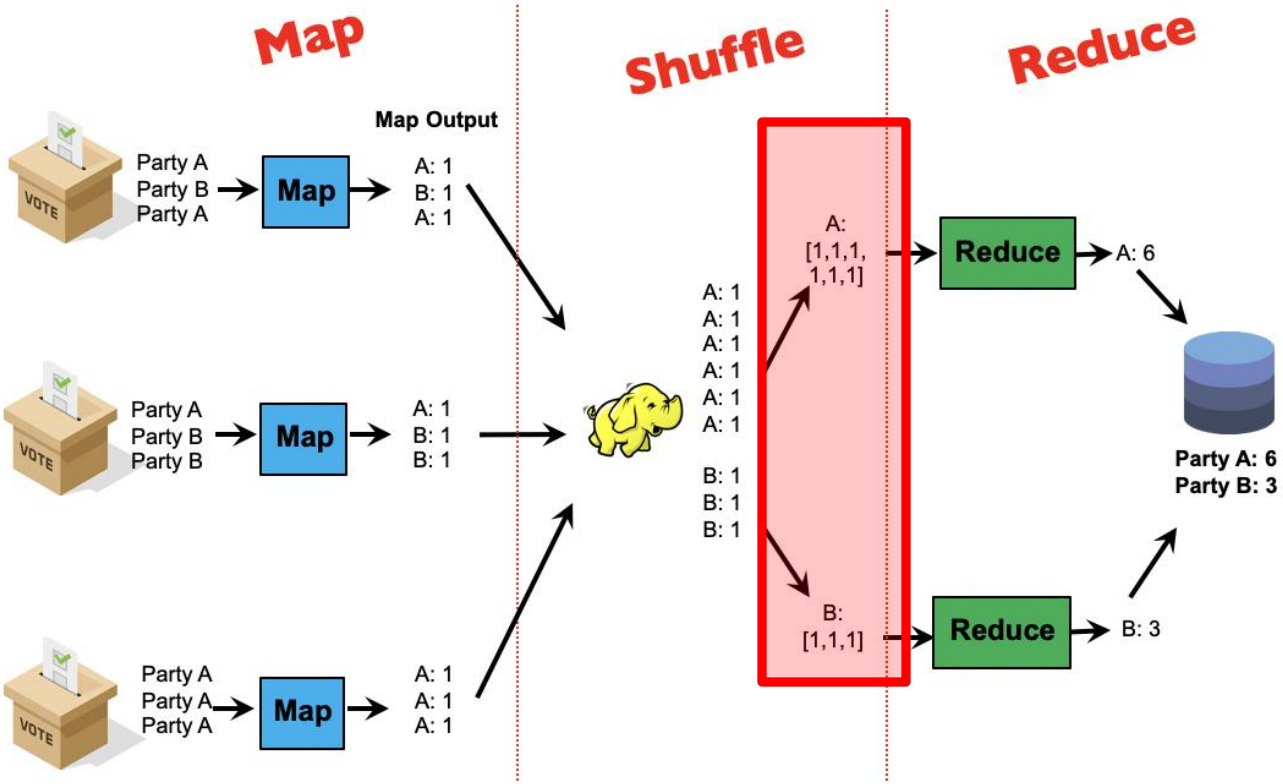
Partition Step



- 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)

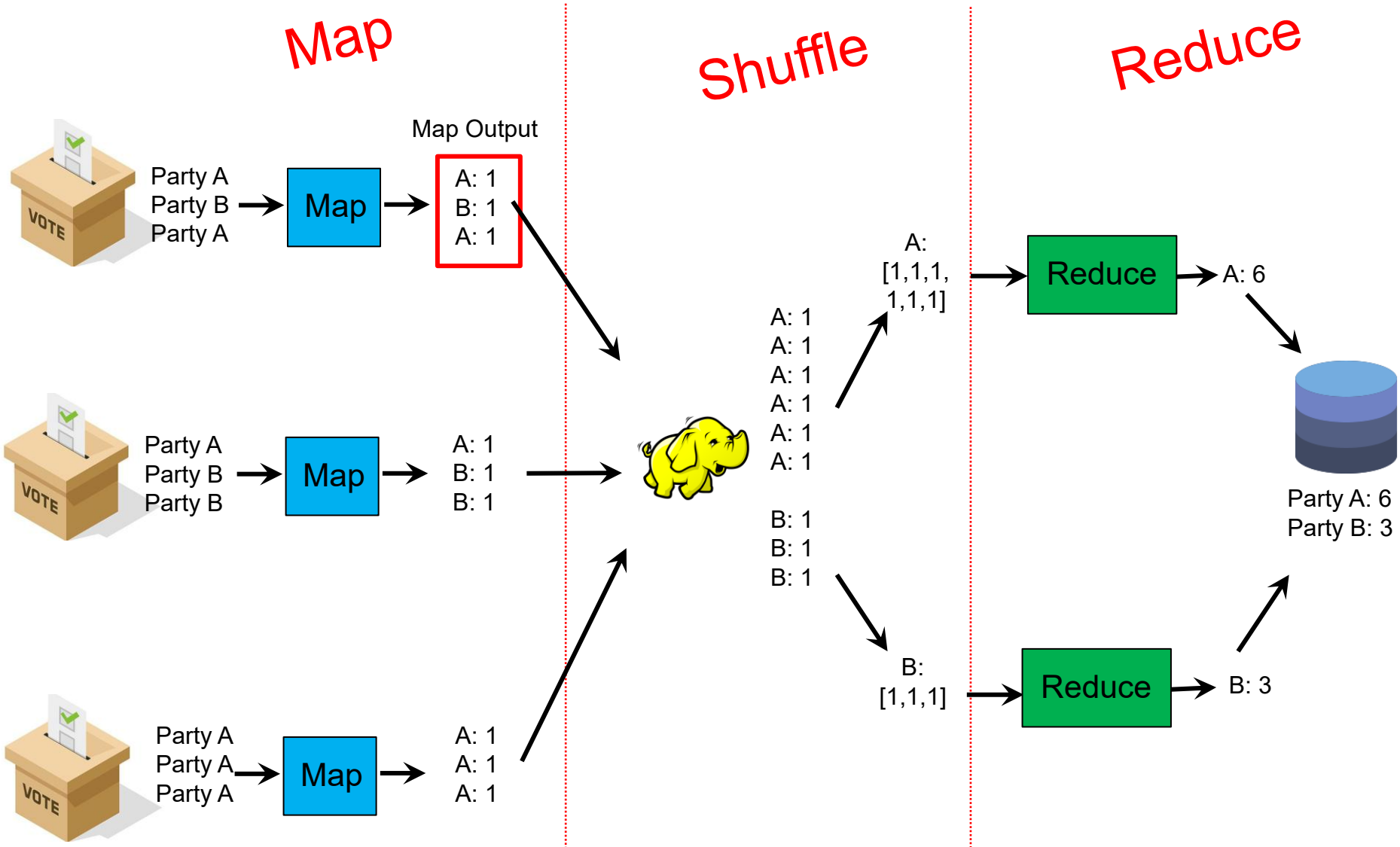


Partition Step



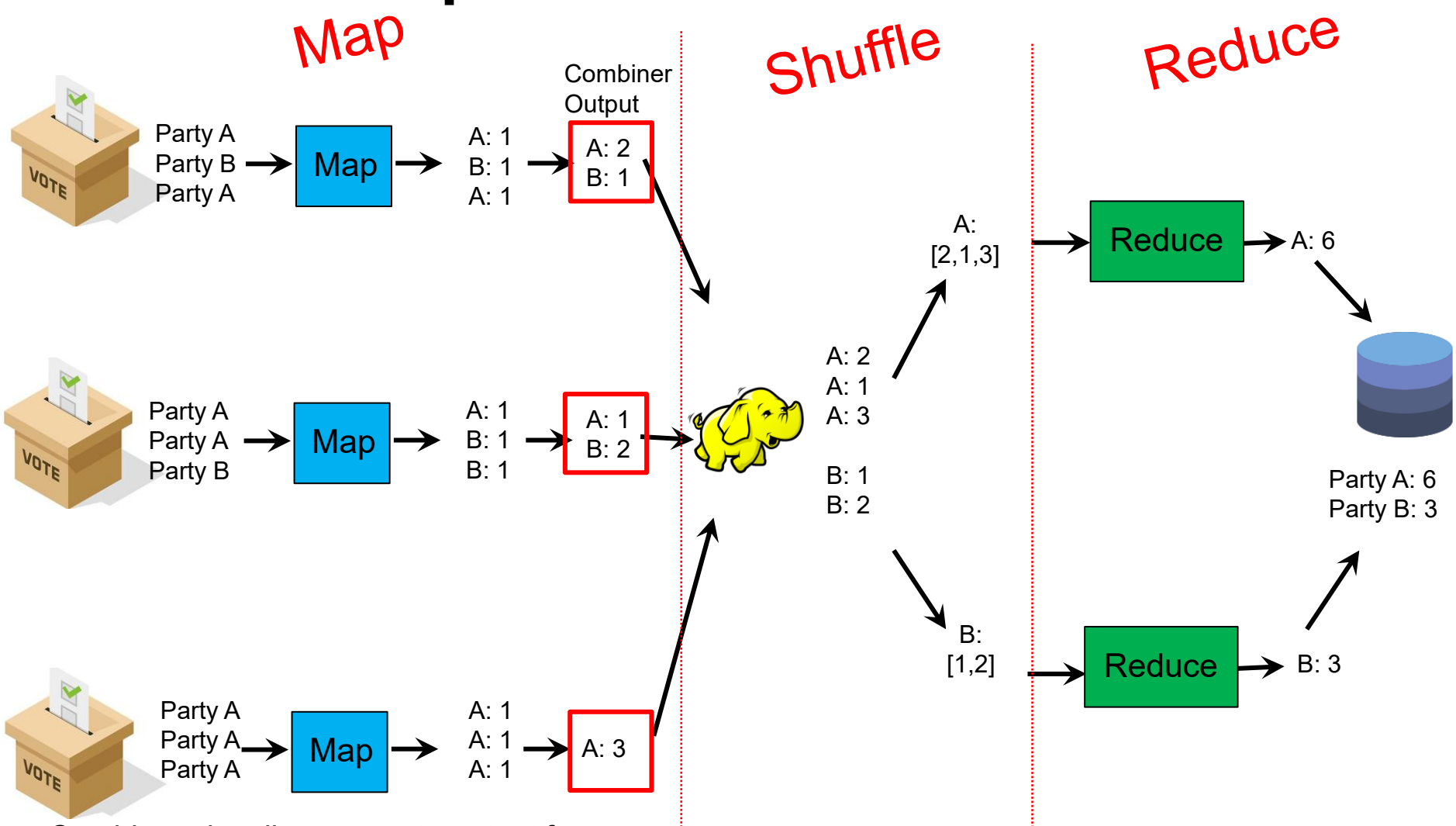
- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a hash function
  - e.g., key k goes to reducer: (hash(k) % num_reducers)
- User can optionally implement a custom partition, e.g. to better spread out the load among reducers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

Combiner Step



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

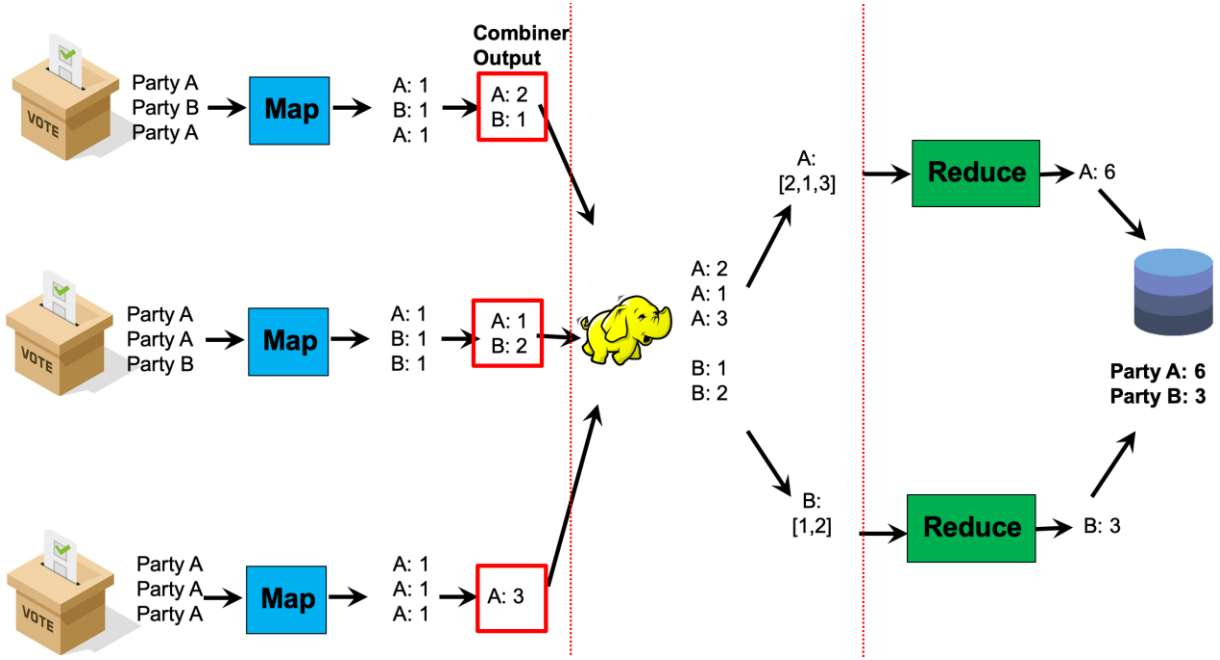
Combiner Step



- 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"?

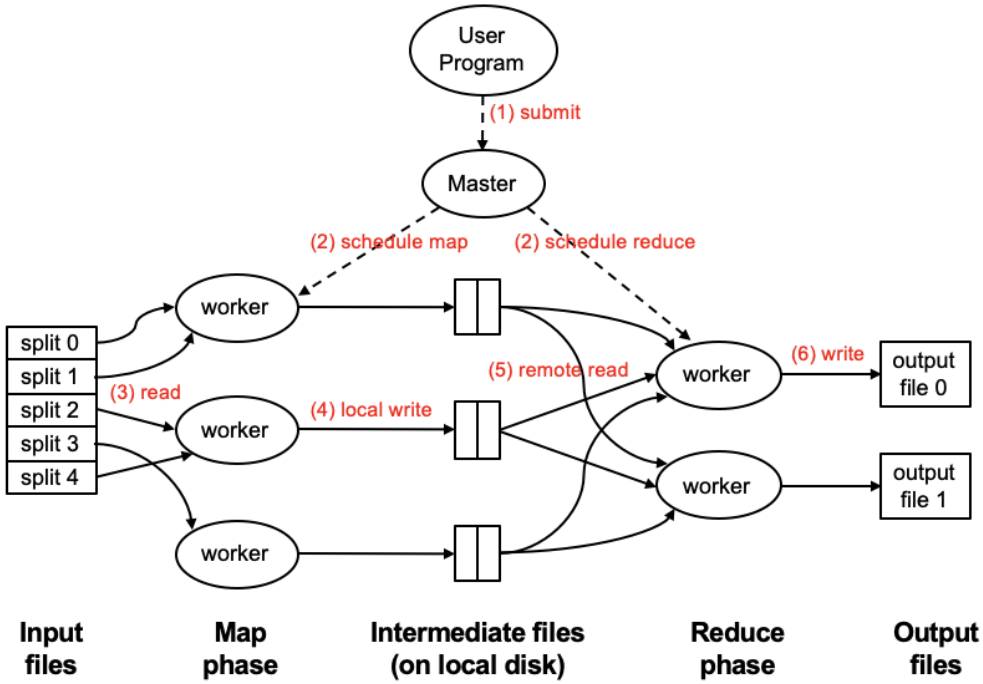


Correctness of Combiner

- 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

Where do Combiner and Partitioner Run?

- **Combiner**: after the Map phase, the tuples are combined by the Combiner. This is done during the **local write** (but before the actual disk write, to save disk I/O)
- **Partitioner**: also runs in the **local write** phase as this stage requires knowing which keys need to go to which reducer.

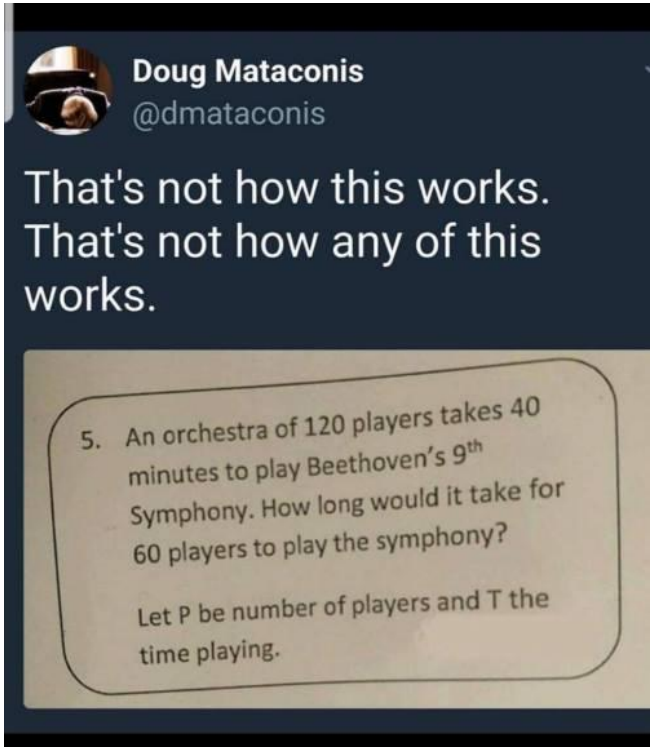


1. MapReduce
   a. Basic MapReduce
   b. Partition and Combiner
   c. **Examples**

Source: Google

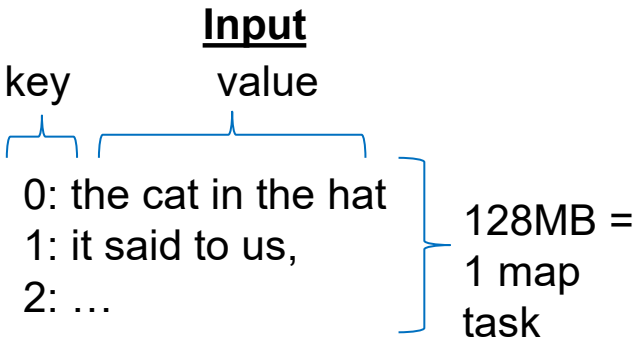
Performance Guidelines for Basic Algorithmic Design

- 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 }
```



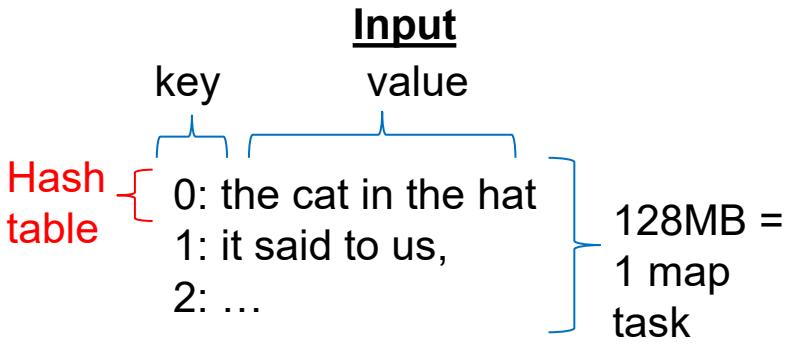
This mapper processes each word one by one, and emits a “1”, to be summed by the reducers.

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  }
```

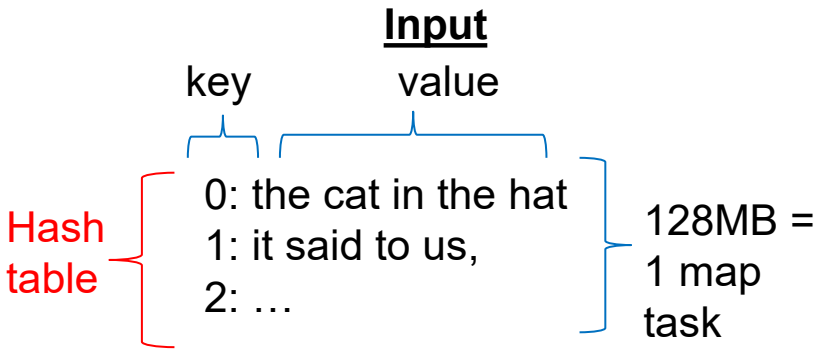


This mapper uses a hash table ("counts") to maintain the words and counts per line (i.e. in each call to the map function). After processing each line it emits the counts for this line.

Are combiners still of any use?

Word Count: Version 2

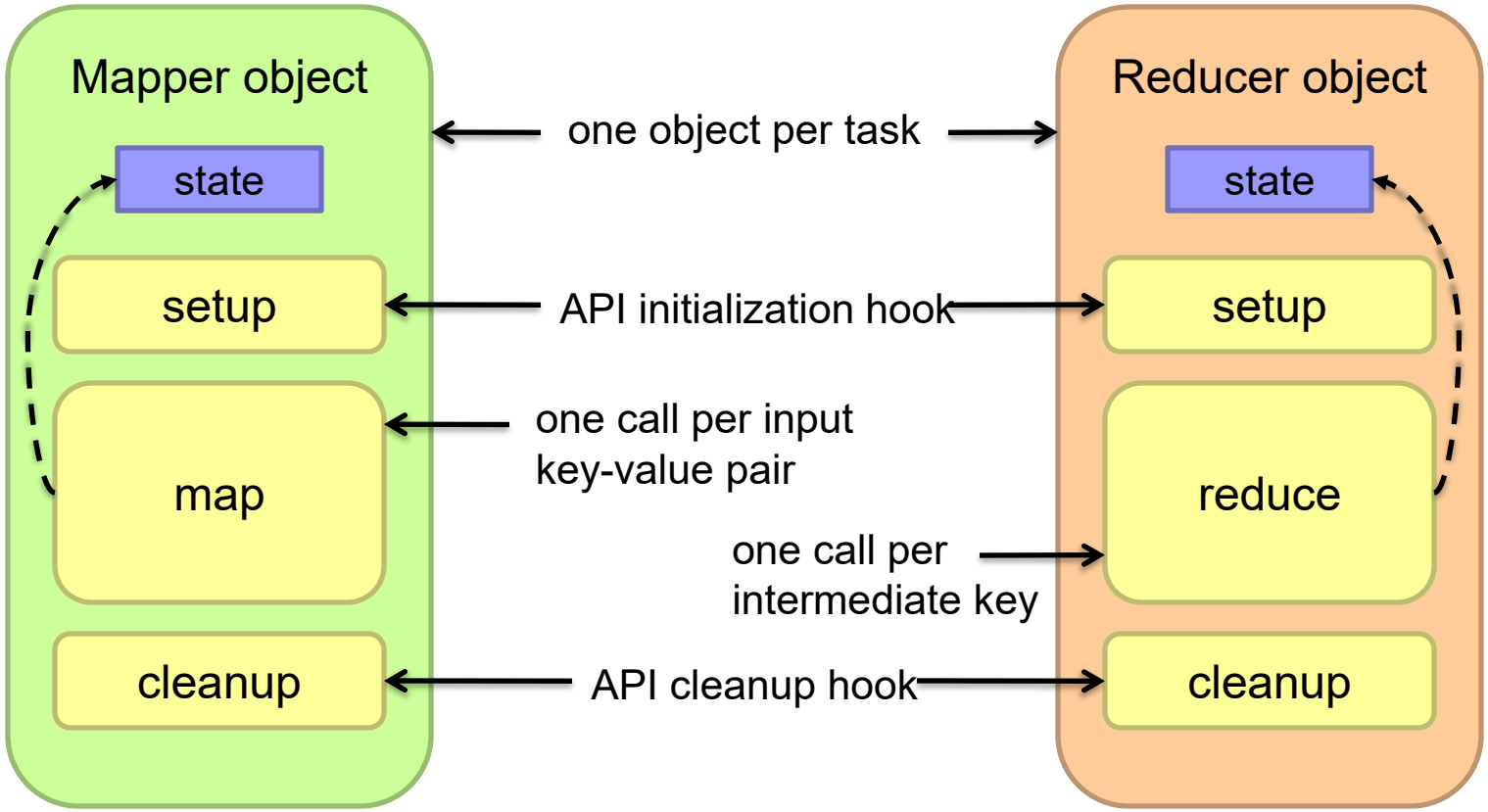
```
1  class Mapper {
2    val counts = new Map()
3
4    def map(key: Long, value: Text) = {
5      for (word <- tokenize(value)) {
6        counts(word) += 1
7      }
8    }
9
10   def cleanup() = {
11     for ((k, v) <- counts) {
12       emit(k, v)
13     }
14   }
15 }
```



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



Q: Which statement(s) about Hadoop's map and reduce phases are true?

1. The number of times the **map function** is called is equal to the number of input splits.
2. The number of times the **reduce function** is called is equal to the number of distinct intermediate keys.
3. The number of **reduce tasks** is equal to the number of intermediate keys.

Q: Which statement(s) about Hadoop's map and reduce phases are true?

1. The number of times the **map function** is called is equal to the number of input splits.
2. The number of times the **reduce function** is called is equal to the number of distinct intermediate keys.
3. The number of **reduce tasks** is equal to the number of intermediate keys.

- **A:** 2.
  The number of:
  - Map function calls: = number of input key-value pairs.
  - Map tasks: = number of input splits.
  - Reduce function calls: = number of distinct intermediate keys.
  - Reduce tasks: specified by the user when configuring the job.

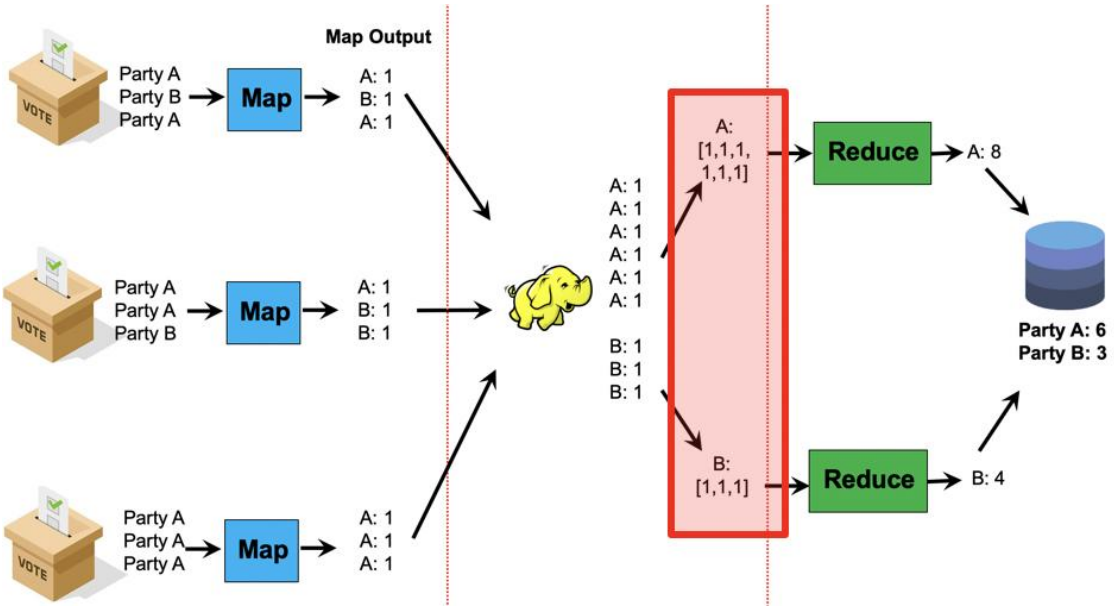
Q: Which statement(s) about Hadoop's partitioner is true?

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 Hadoop'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



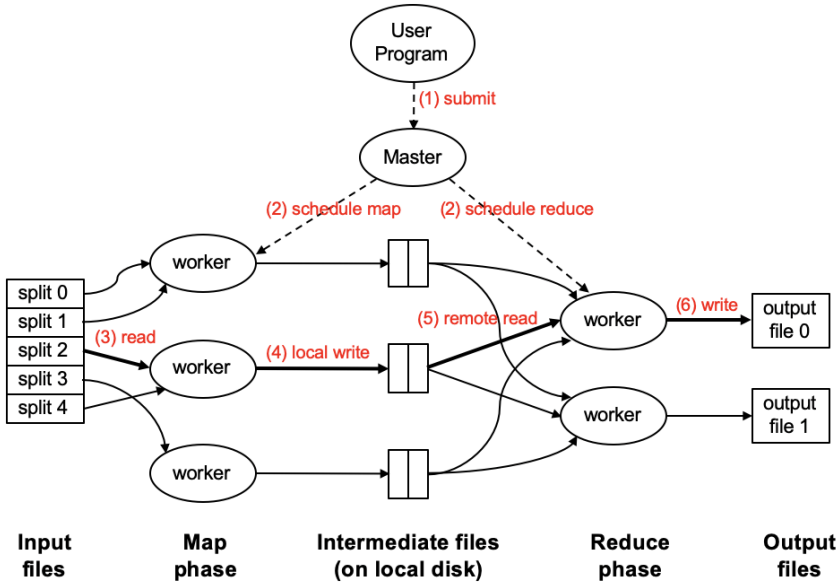
Q: True or false: the Shuffle stage of MapReduce is always run within a single node.

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
  - https://hadoop.apache.org/docs/current/

Take-away

- Big data needs new programing abstractions and runtime systems, rather than conventional approaches in parallelization.
- With the popularity of Hadoop, MapReduce programming framework has become quite common for Big Data.
- As we will see, MapReduce can be used to efficiently and effectively develop various applications such as large databases and data mining.
- Further reading:
  - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004. https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Take-away in the AI Era

- (Why MapReduce still matters when AI can write distributed code)
- 🧠 What AI cannot decide for you (yet)
  - Whether MapReduce is the *right abstraction* for the problem
  - How much data will be shuffled across the network
  - Whether performance is limited by disk, network, or skew
  - Whether an optimization preserves correctness
- **Your design matters (AI can assist)**
  - Choosing the right abstraction
  - Reasoning about data movement
  - Managing skew and parallelism
  - Ensuring correctness under optimization

Acknowledgement

- Slides adopted/revised from
  - Jimmy Lin, http://lintool.github.io/UMD-courses/bigdata-2015-Spring/
  - Bryan Hooi
- 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/