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

- Small updates to tutorial dates in schedule
- (Note: you can check the schedule in LumiNUS > Module Overview > Ov)

Week	Date	Topics	Tutorial	Due Dates
1	12 Aug	Overview and Introduction		
2	29 Aug	MapReduce - Introduction		
3	26 Aug	MapReduce and Relational Databases		
4	2 Sep	MapReduce and Data Mining	Tutorial: Hadoop	Assignment 1 released
5	9 Sep	NoSQL Overview 1		
6	16 Sep	NoSQL Overview 2		
Recess				
7	30 Sep	Apache Spark 1	Tutorial: NoSQL & Spark	Assignment 1 due, Assignment 2 released (3 Oct)
8	7 Oct	Apache Spark 2		
9	14 Oct	Large Graph Processing 1	Tutorial: Graph Processing	
10	21 Oct	Large Graph Processing 2		
11	28 Oct	Stream Processing	Tutorial: Stream Processing	Assignment 2 due (31 Oct)
12	4 Nov	Deepavali – No Class		
13	11 Nov	**Test**		

Announcements

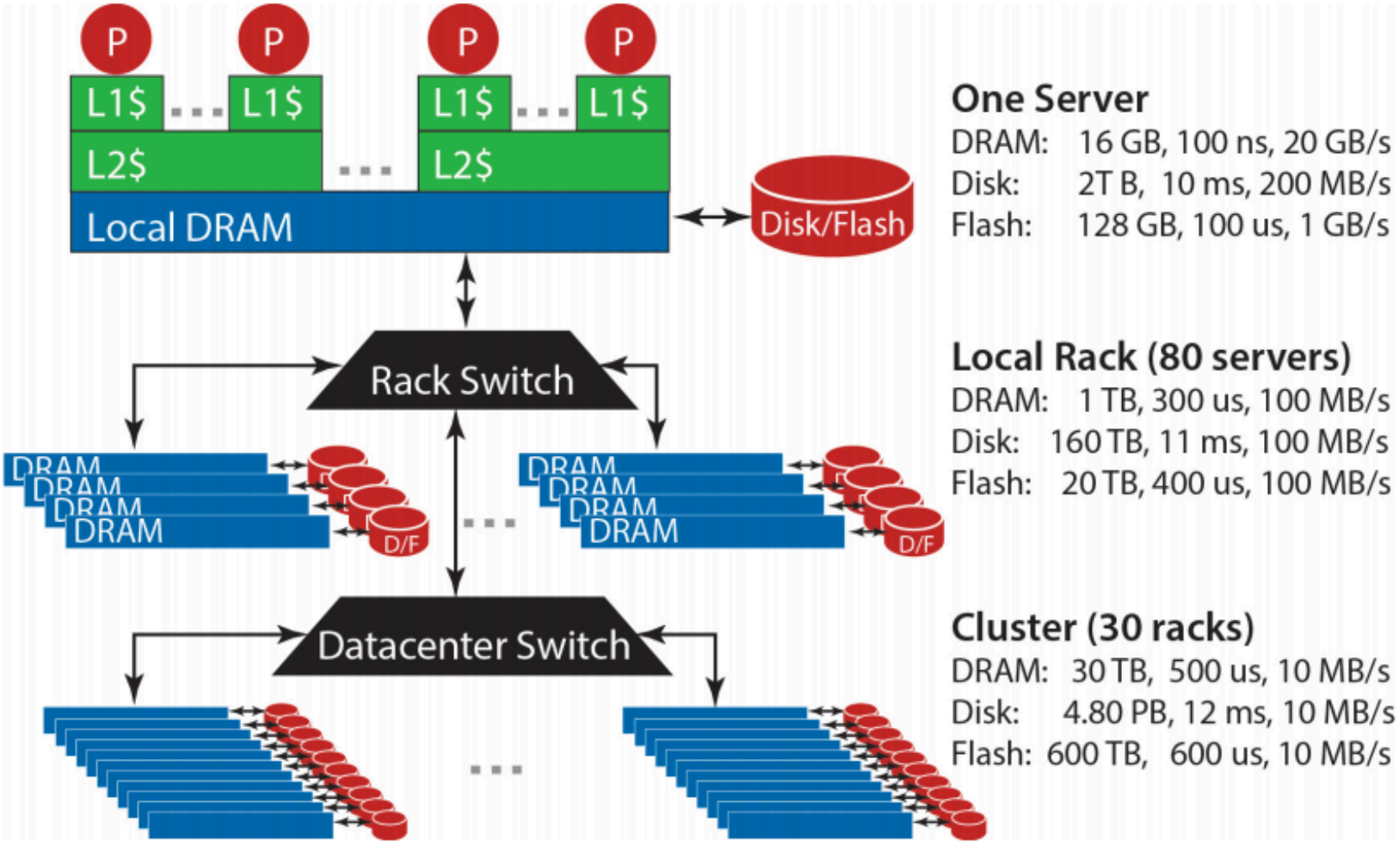
- There will soon be a survey in LumiNUS asking if you want to take the test in-class.
- This is just so I know what size of room is needed to be booked. The survey is anonymous and not binding, but still try to answer it accurately.

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.

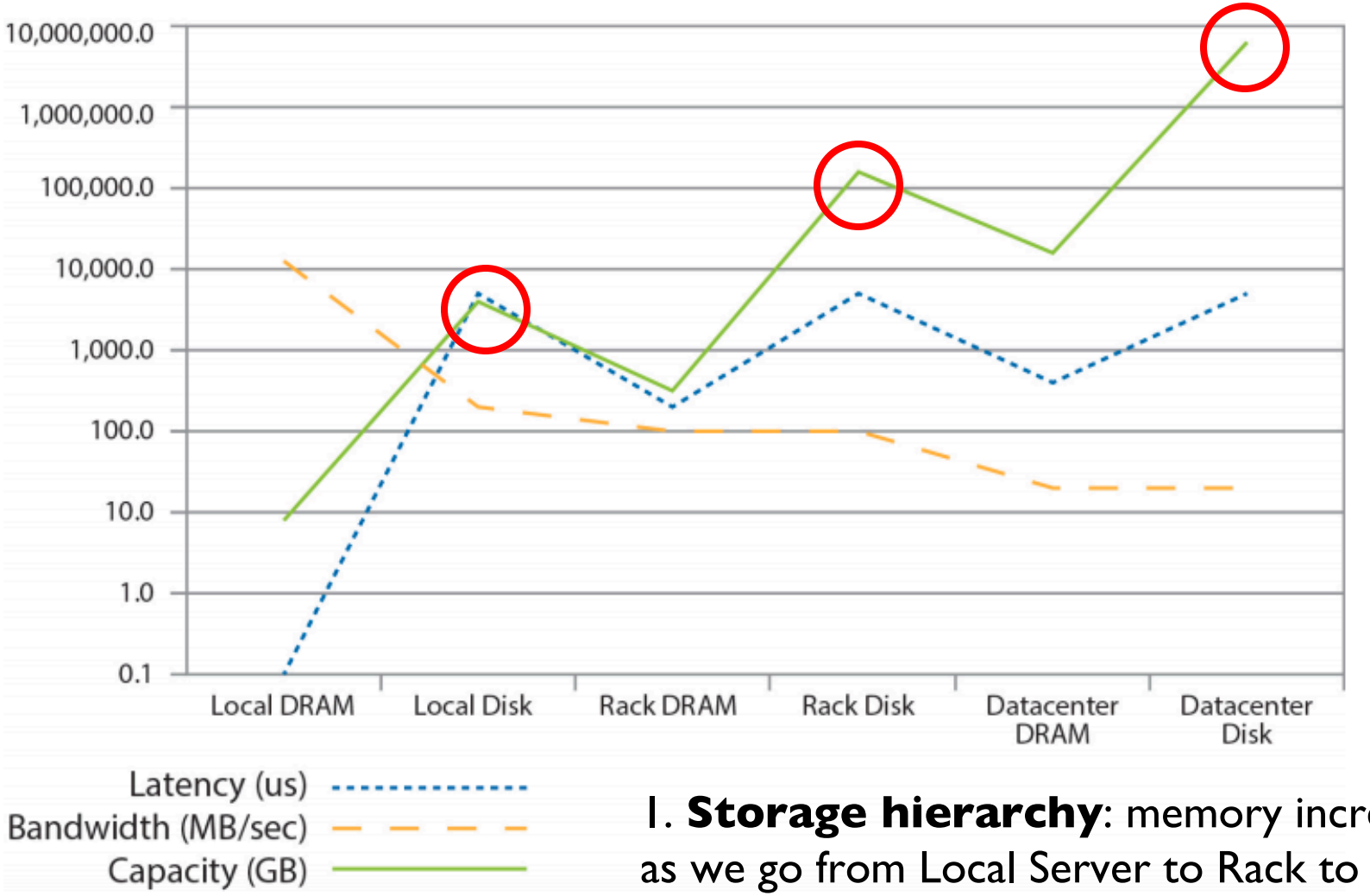
Credit: Katie Hempenius (via Jake Archibald)

Recap: Storage Hierarchy



Source: Barroso and Urs Hölzle (2013)

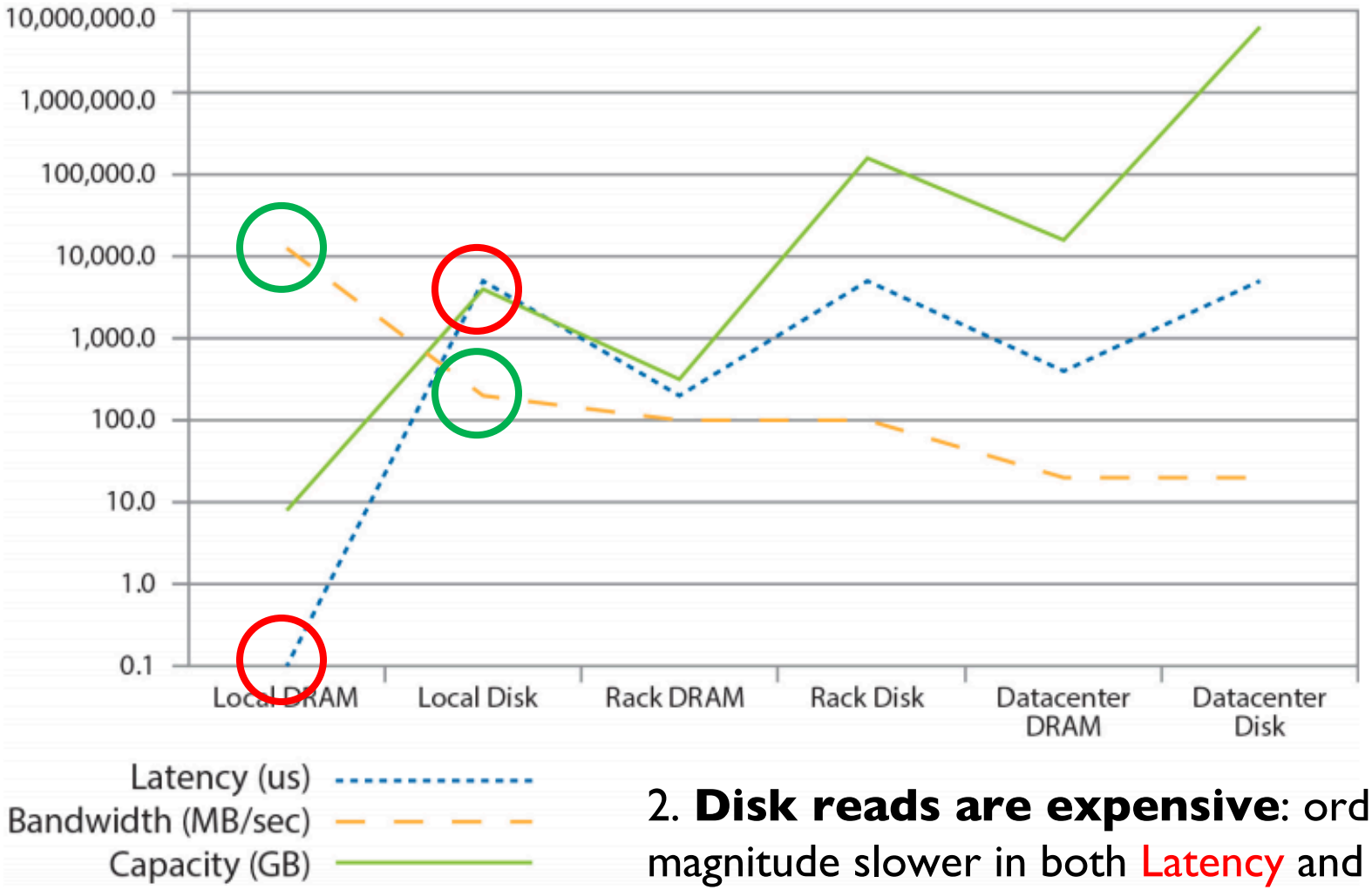
Recap: The Cost of Moving Data Around Data Center



1. **Storage hierarchy**: memory 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: The Cost of Moving Data Around Data Center



2. **Disk reads are expensive**: orders of magnitude slower in both Latency and Bandwidth. Latency of disk reads is even more than from Rack / Datacenter DRAM.

Source: Barroso and Urs Hölzle (2013)

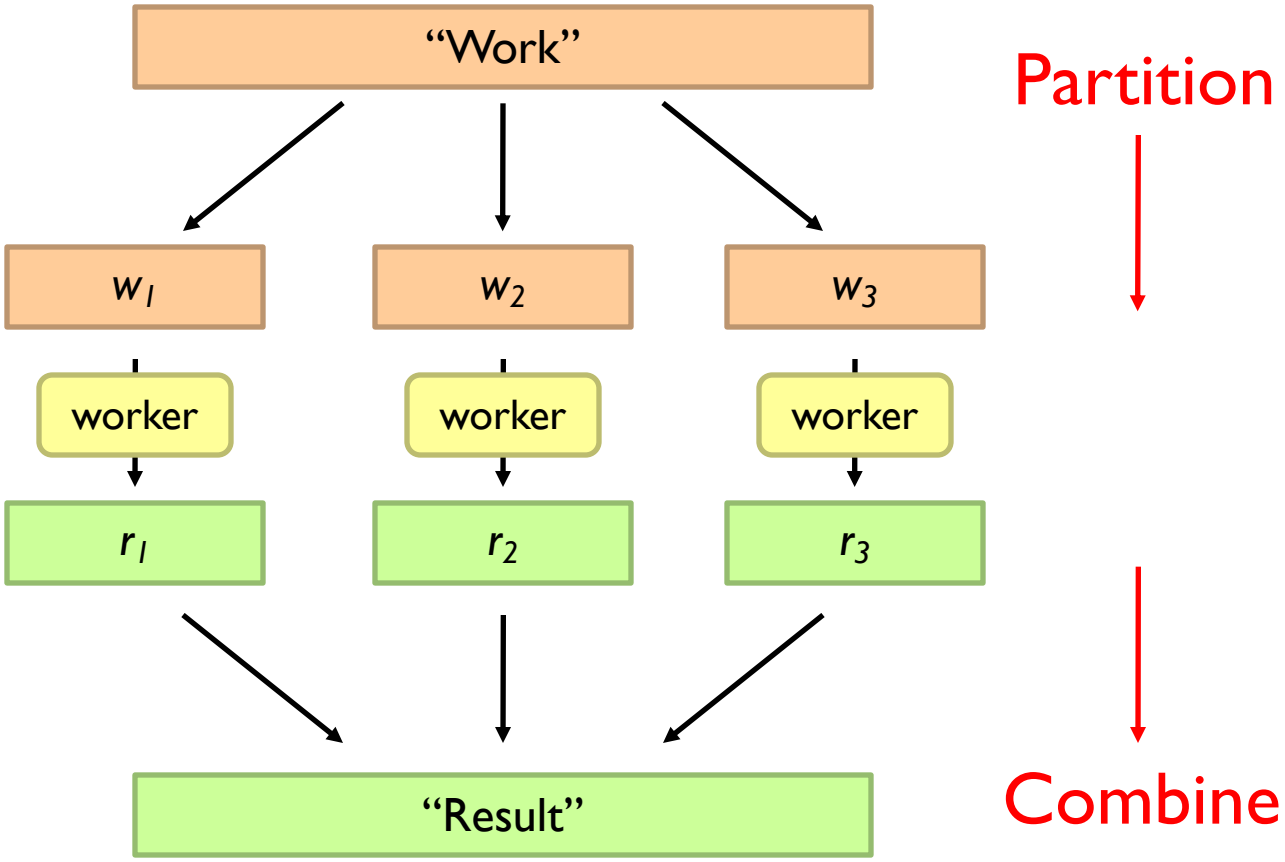
1. **MapReduce**
   a. **Motivation**
   b. Basic MapReduce
   c. Partition and Combiner
   d. Examples
2. Hadoop File System

Source: Google

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- Infrastructure: cloud + data centers
  - Cluster of commodity nodes
  - Commodity network (ethernet) to connect them
- Solution:
  - Parallelization + divide and conquer

Divide and Conquer



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?

Challenge 1: Machine Failures

- One server may stay up 3 years (1,000 days)
- If you have 1,000 servers, expect to loose 1/day
- People estimated Google had ~1M machines in 2011
  - 1,000 machines fail every day!

Challenge 2: Synchronization

- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know when workers need to communicate partial results
  - We don't know the order in which workers access shared data
- Thus, we need (note: not required knowledge for class)
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

Challenge 3: Programming Difficulty

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

The datacenter *is* the computer

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - What's the "instruction set" 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

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

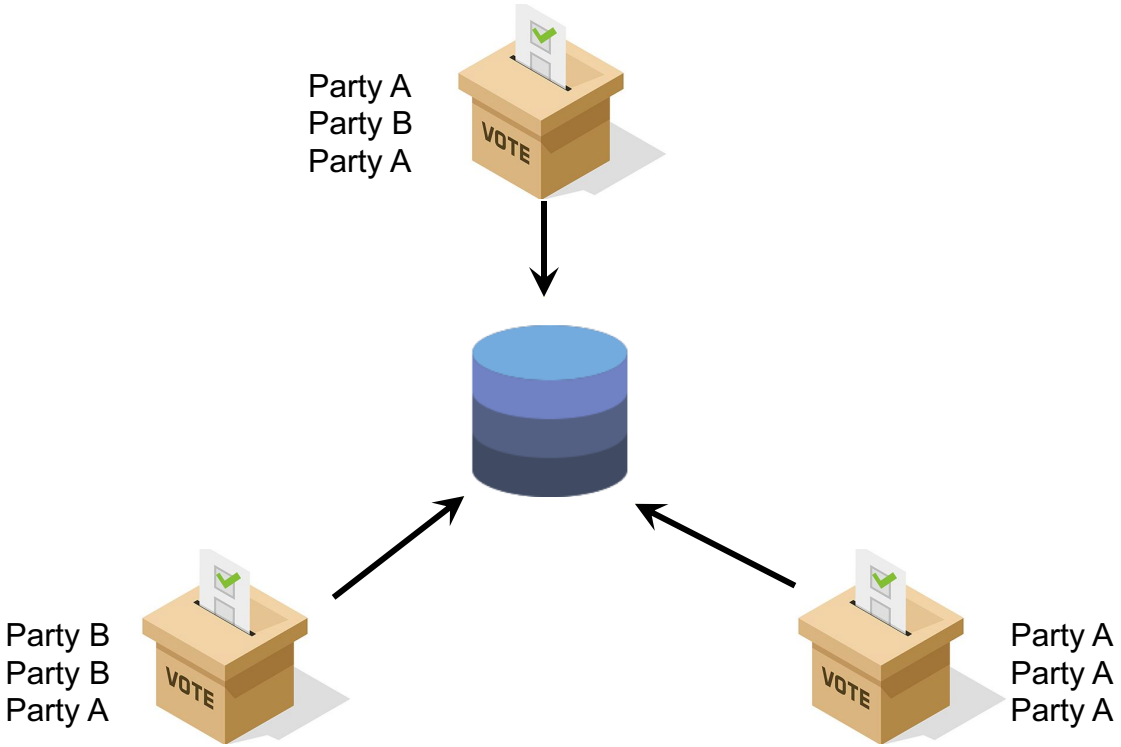
(Dean and Ghemawat, OSDI 2004)

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

Source: Google

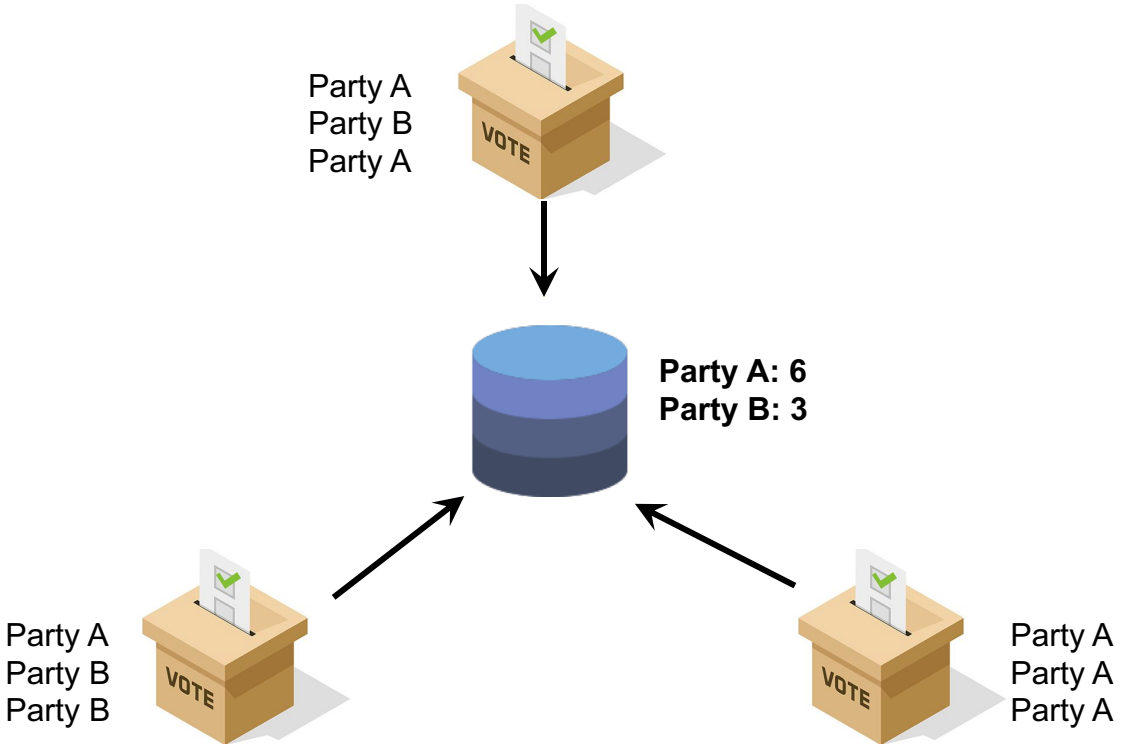
Basic Example: Tabulating Election Results from Multiple Polling Sites

- 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

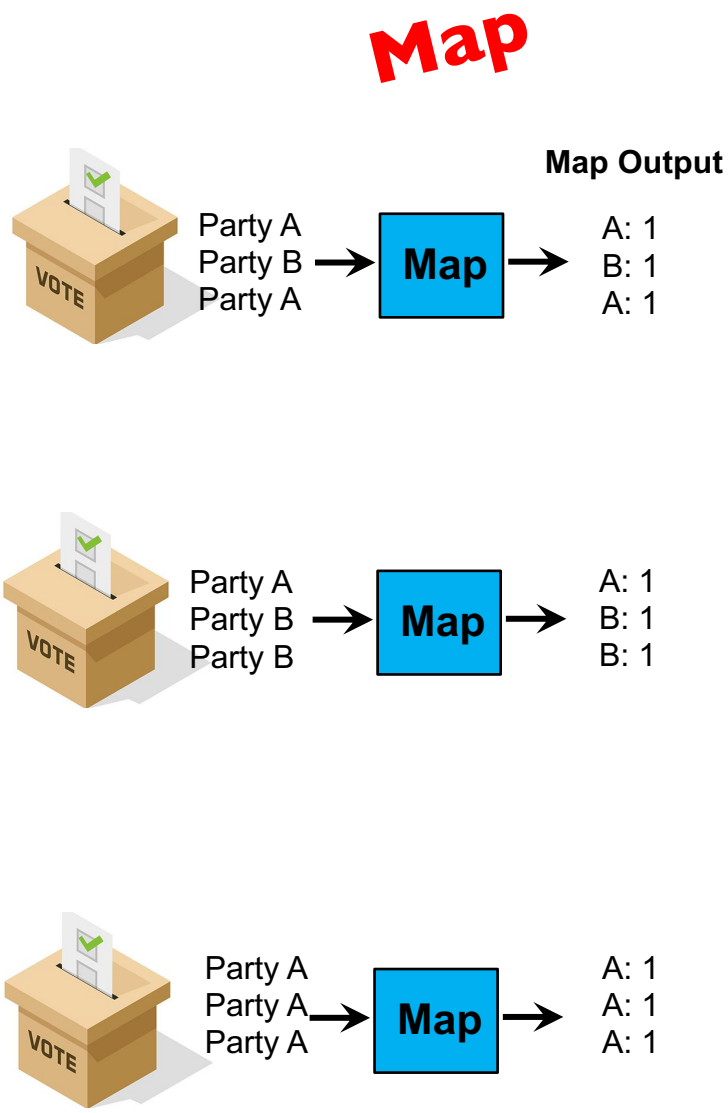
Party A
Party B
Party A

Party A
Party B
Party B

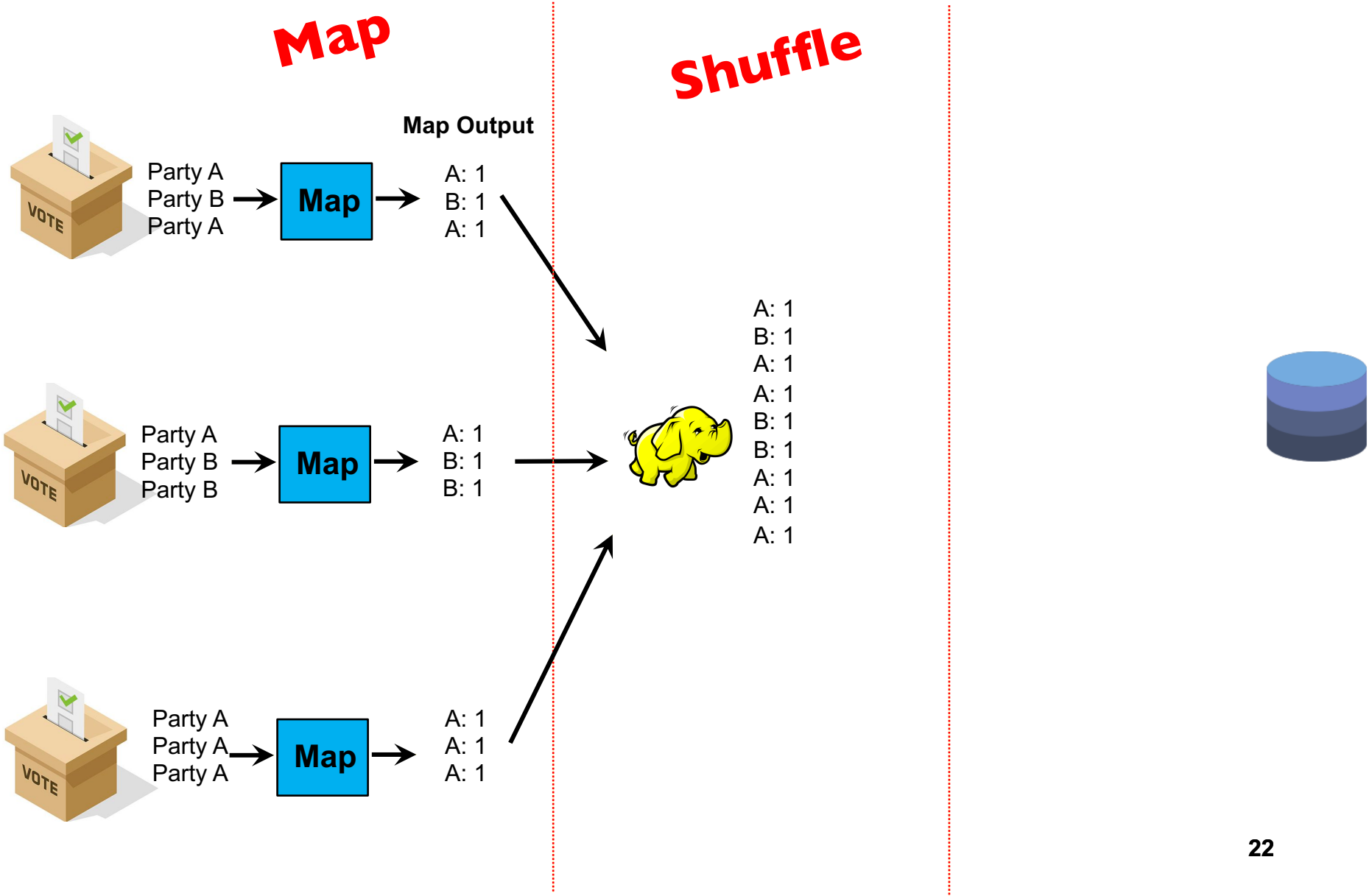
Party A
Party A
Party A

Party A: 6
Party B: 3

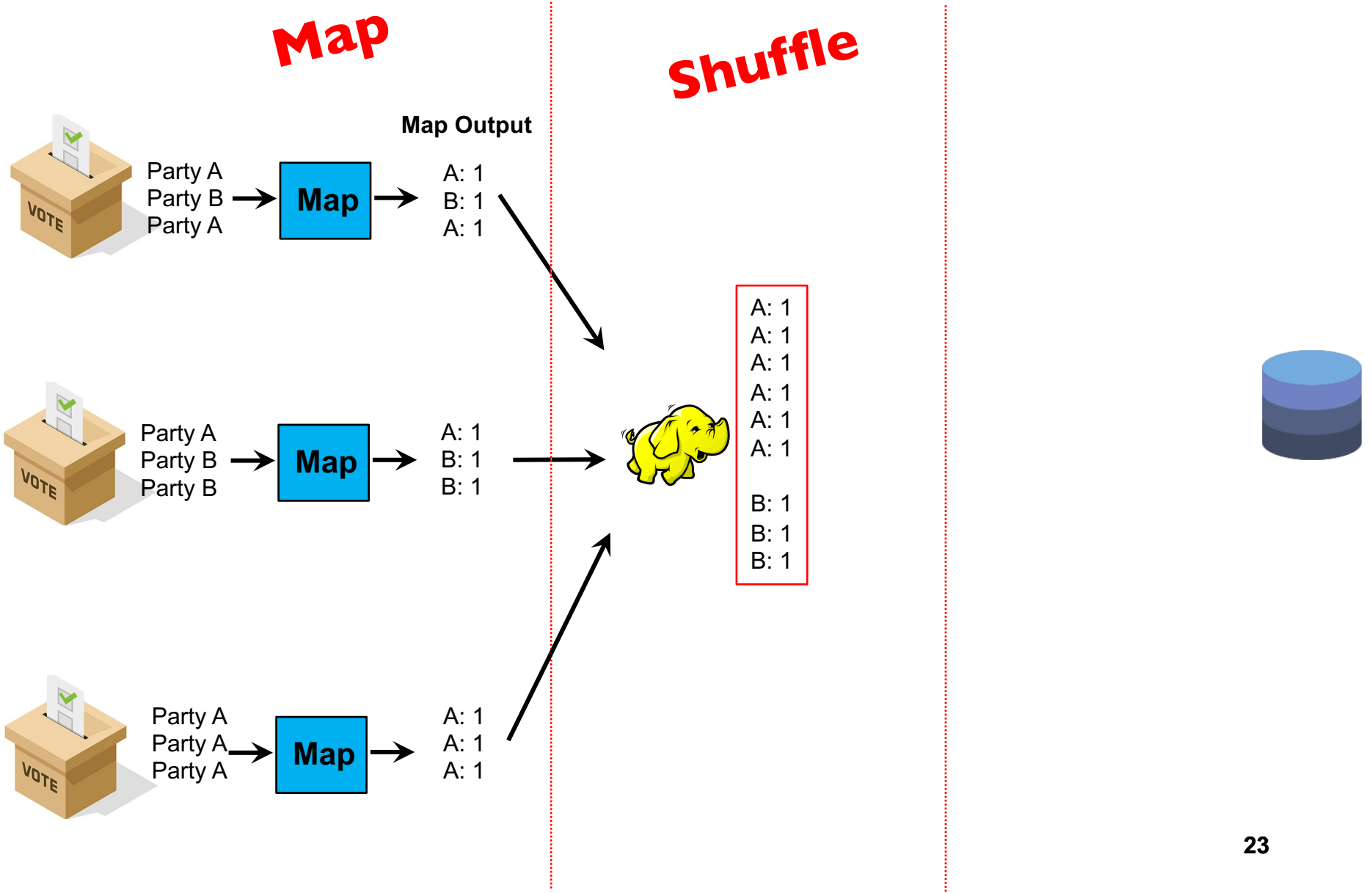
Tabulating Election Results: MapReduce



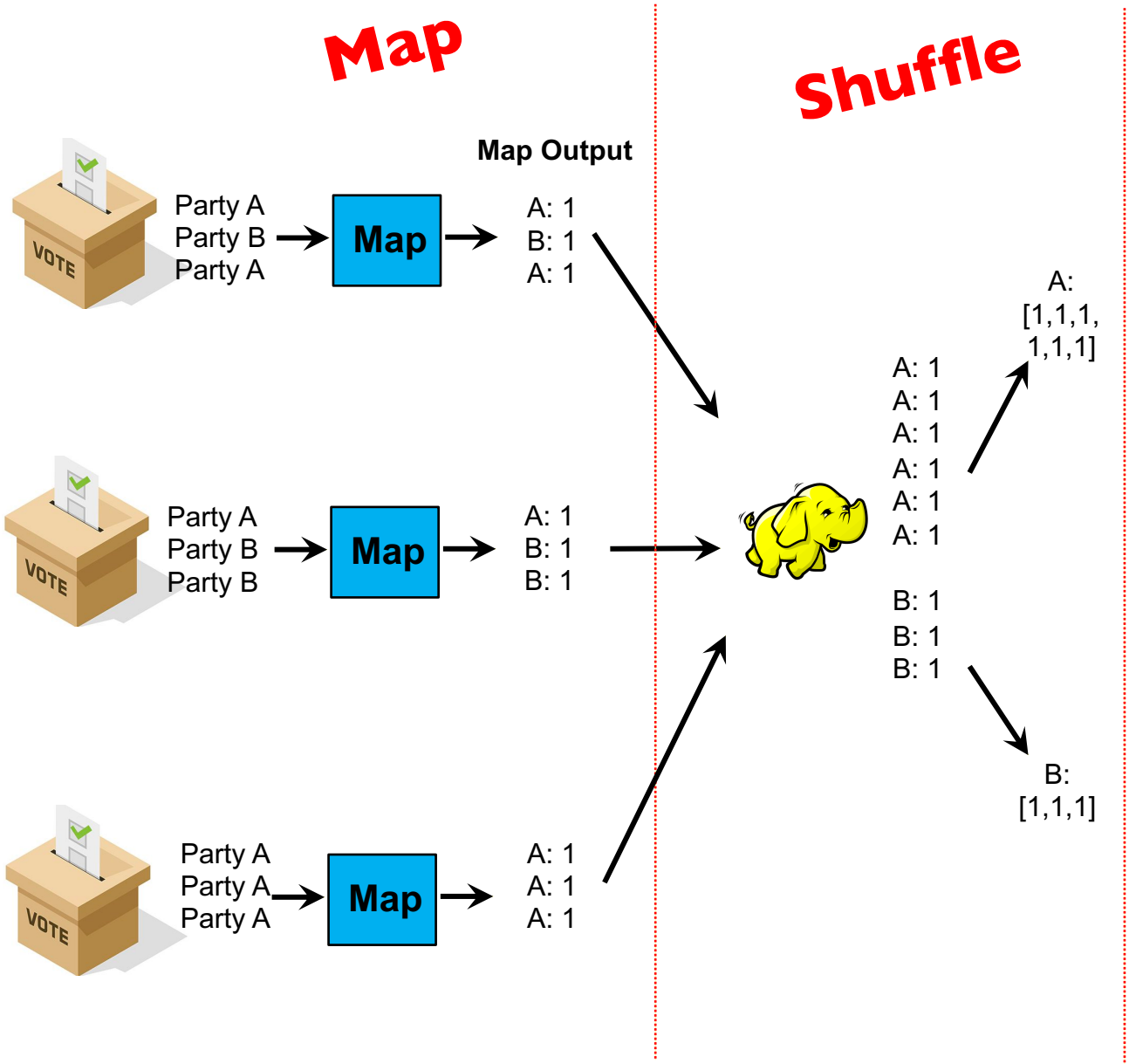
Tabulating Election Results: MapReduce



Tabulating Election Results: MapReduce

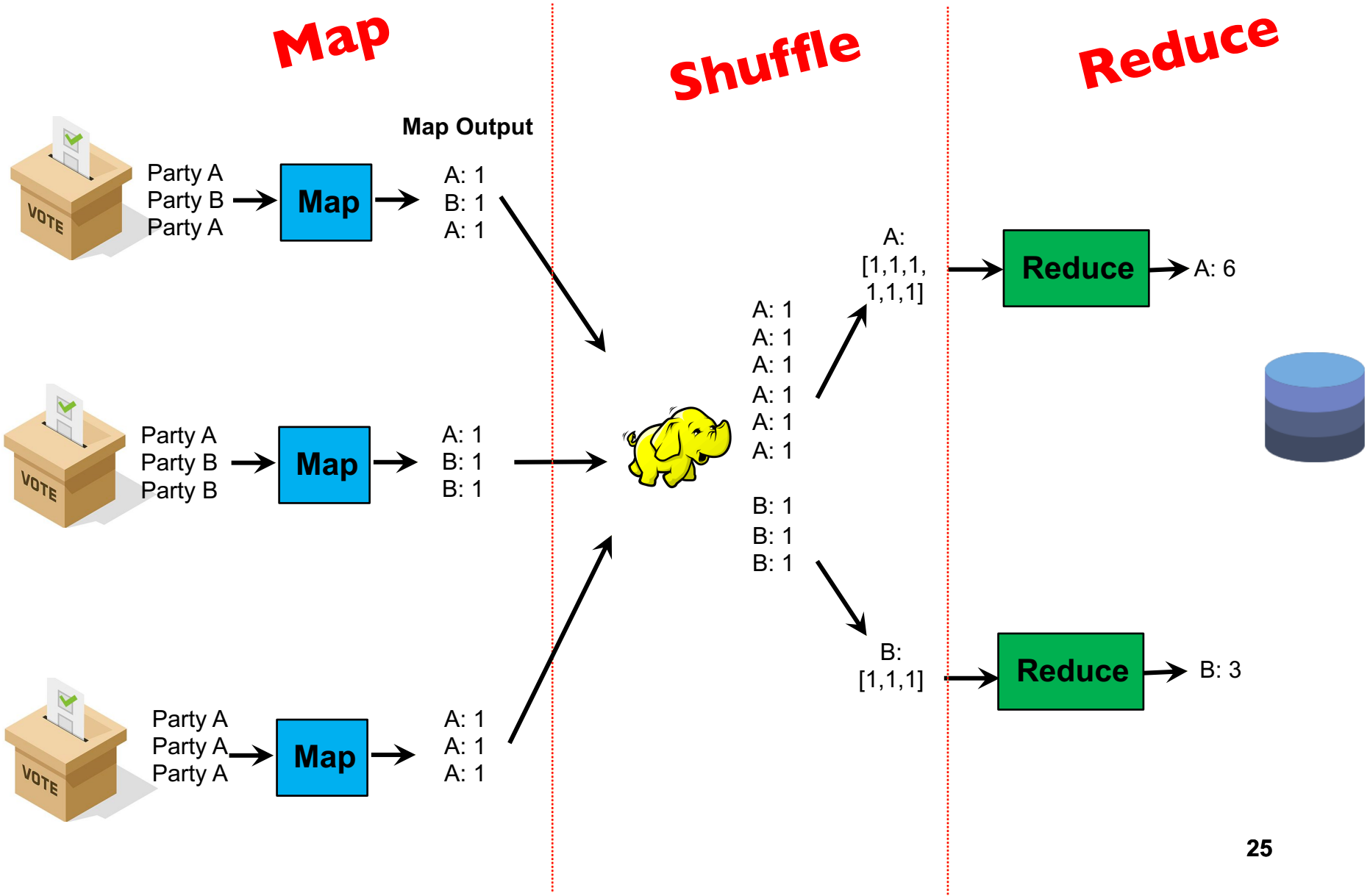


Tabulating Election Results: MapReduce

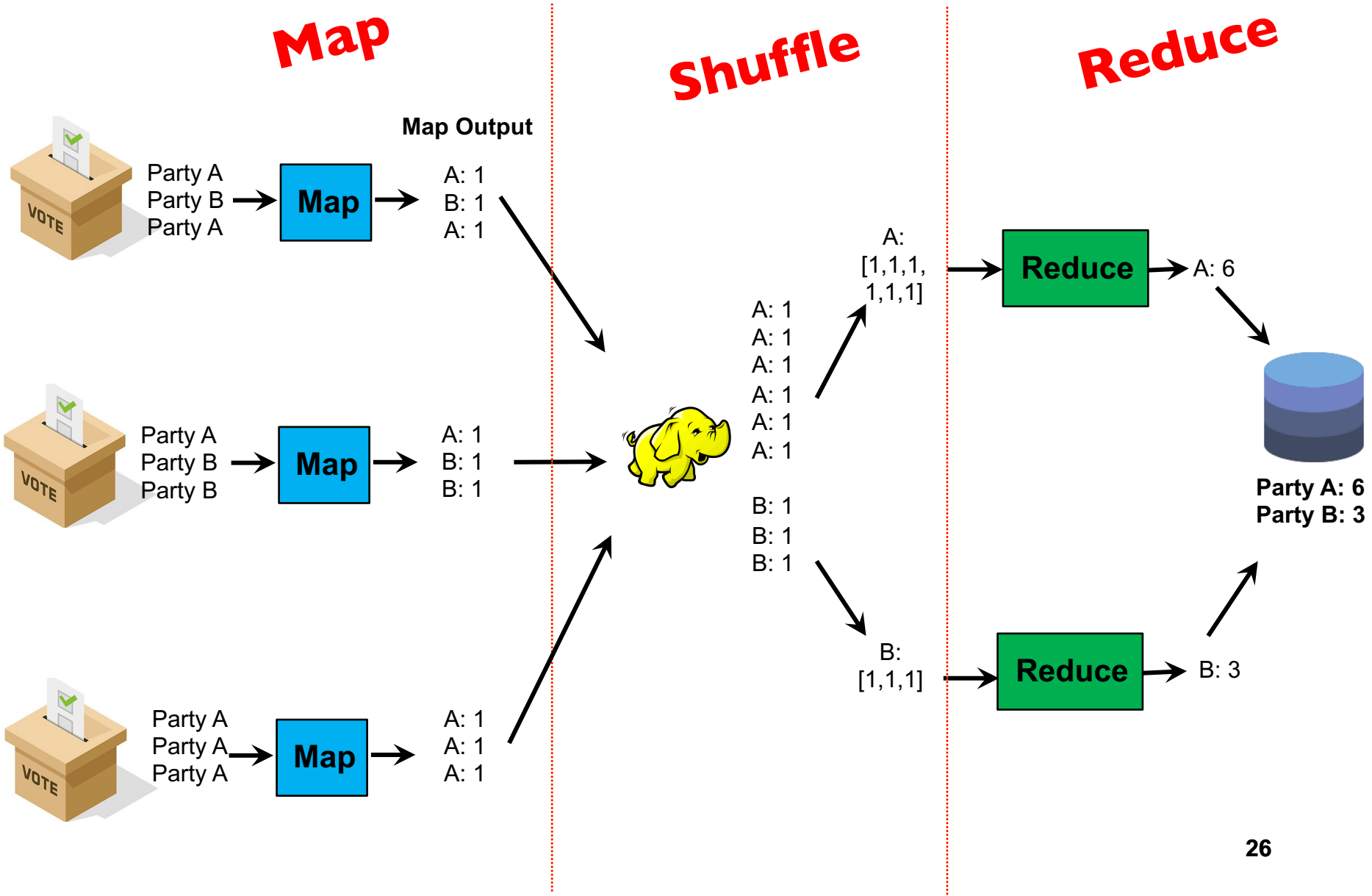


B: 4

Tabulating Election Results: MapReduce



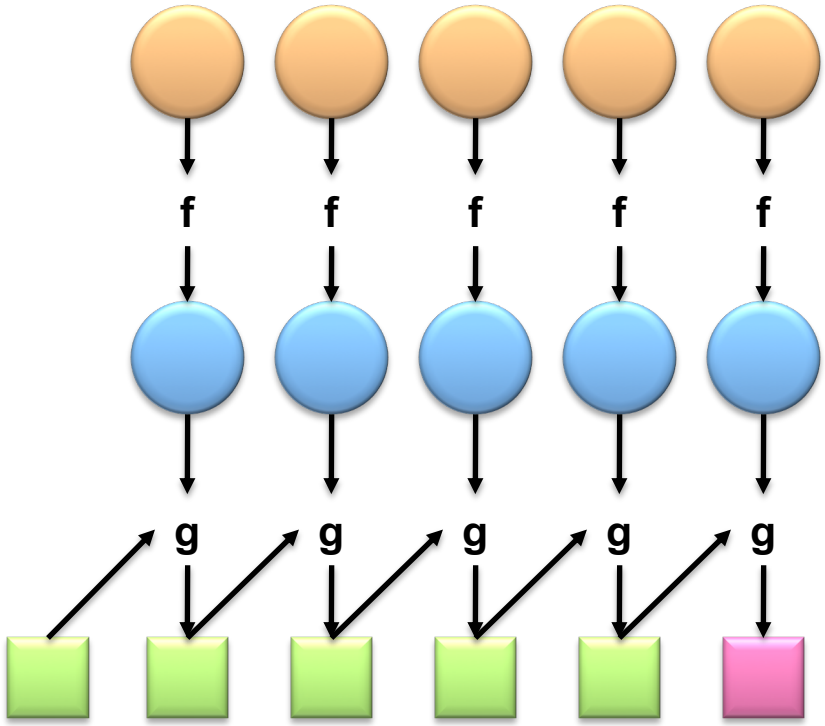
Tabulating Election Results: MapReduce



Roots in Functional Programming

Map

Fold



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 sent to the same reducer

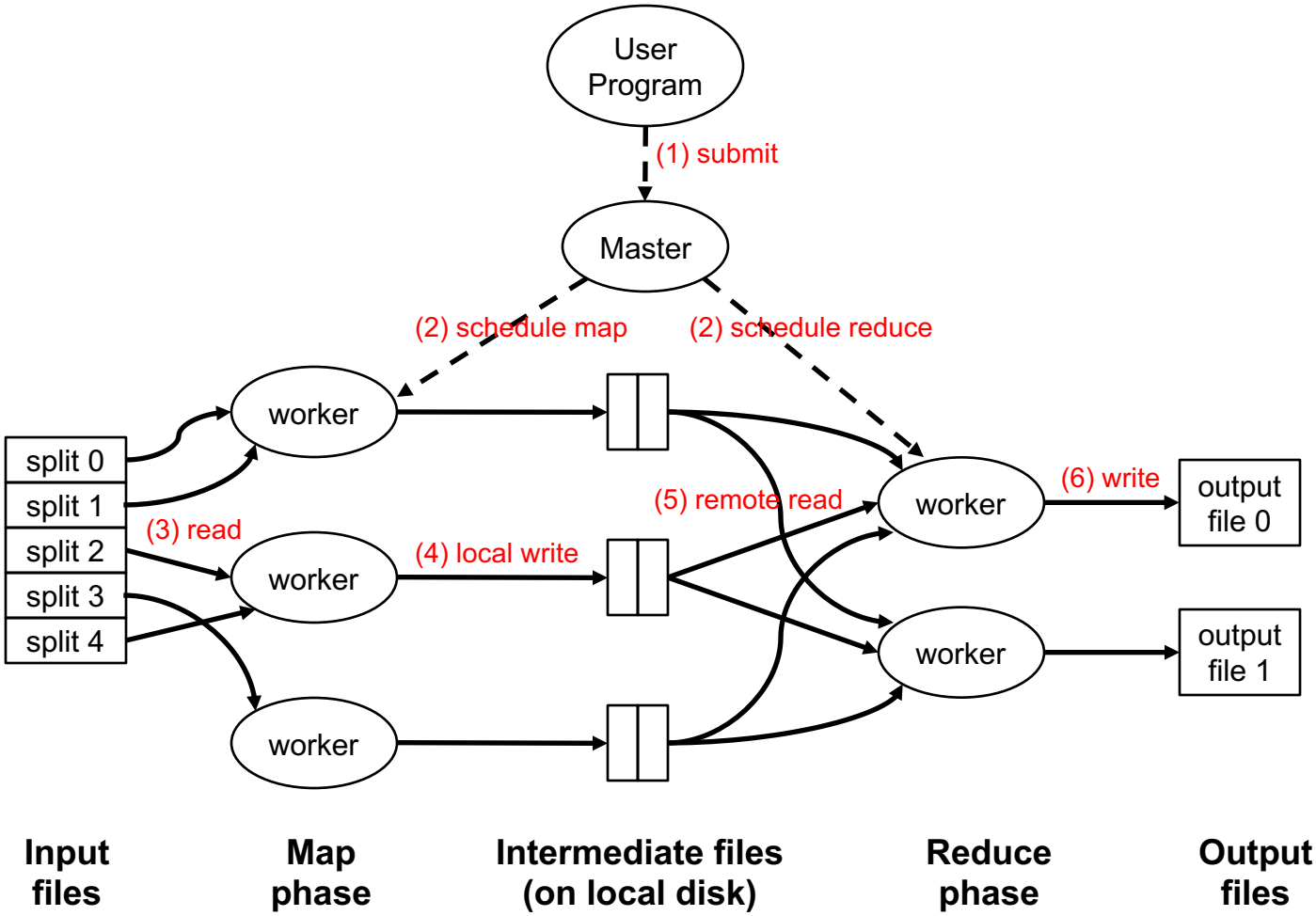
- The execution framework handles the challenging issues…

  - 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?

MapReduce "Runtime"

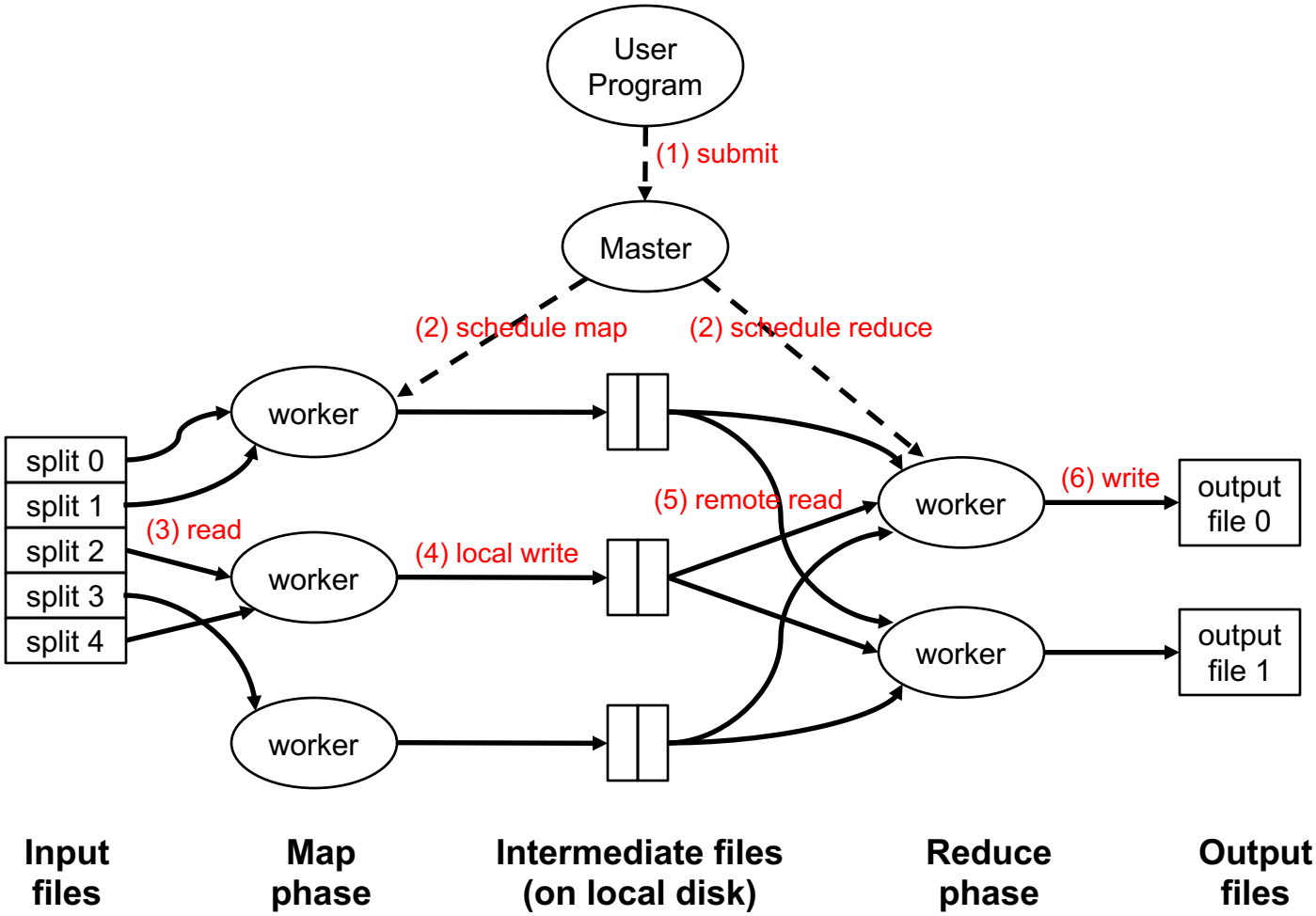
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

MapReduce Implementation



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?

Adapted from (Dean and Ghemawat, OSDI 2004)

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

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

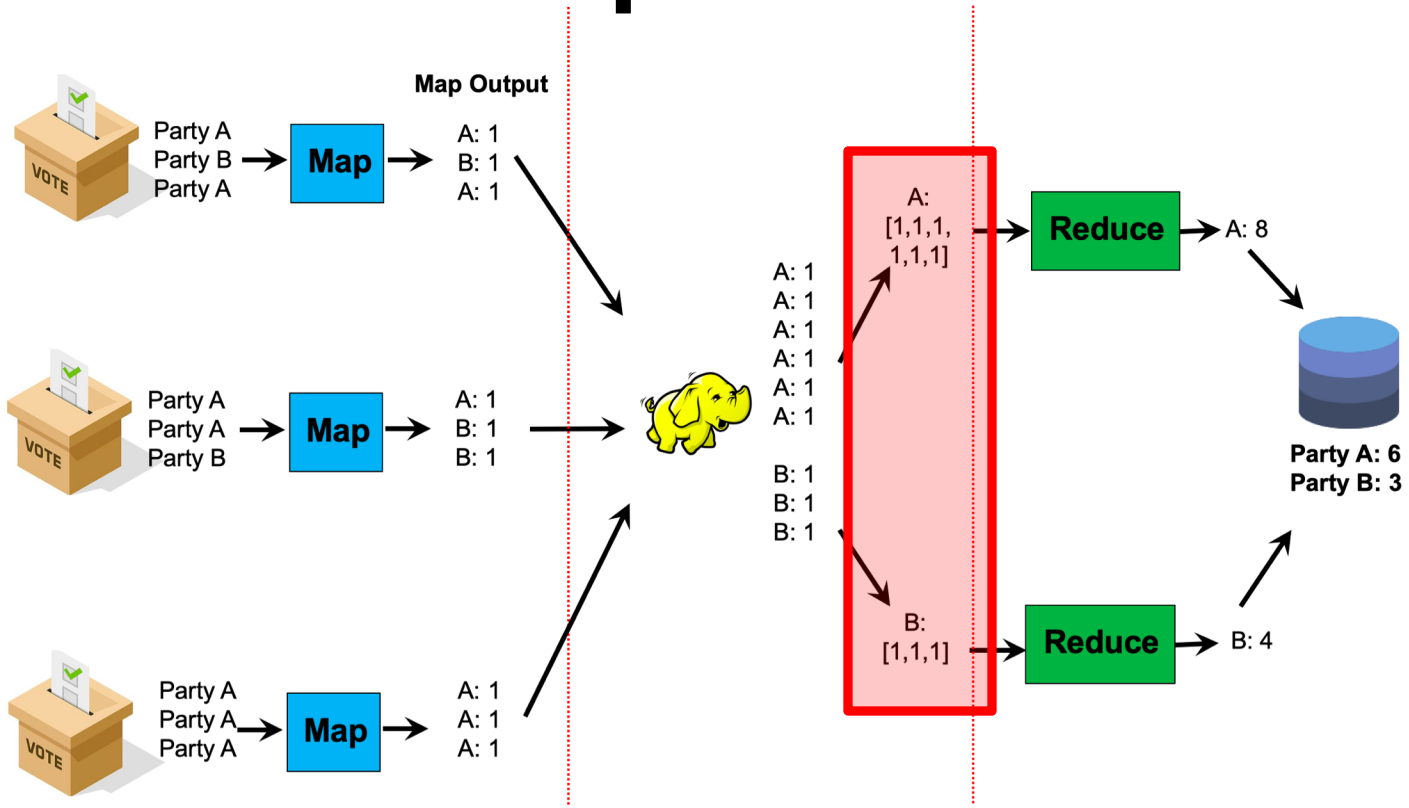
  partition

  - Used to decide which reducer will handle each key

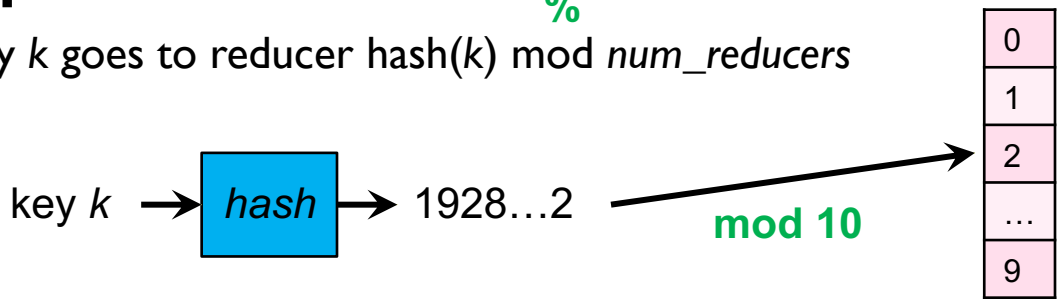
  combine

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

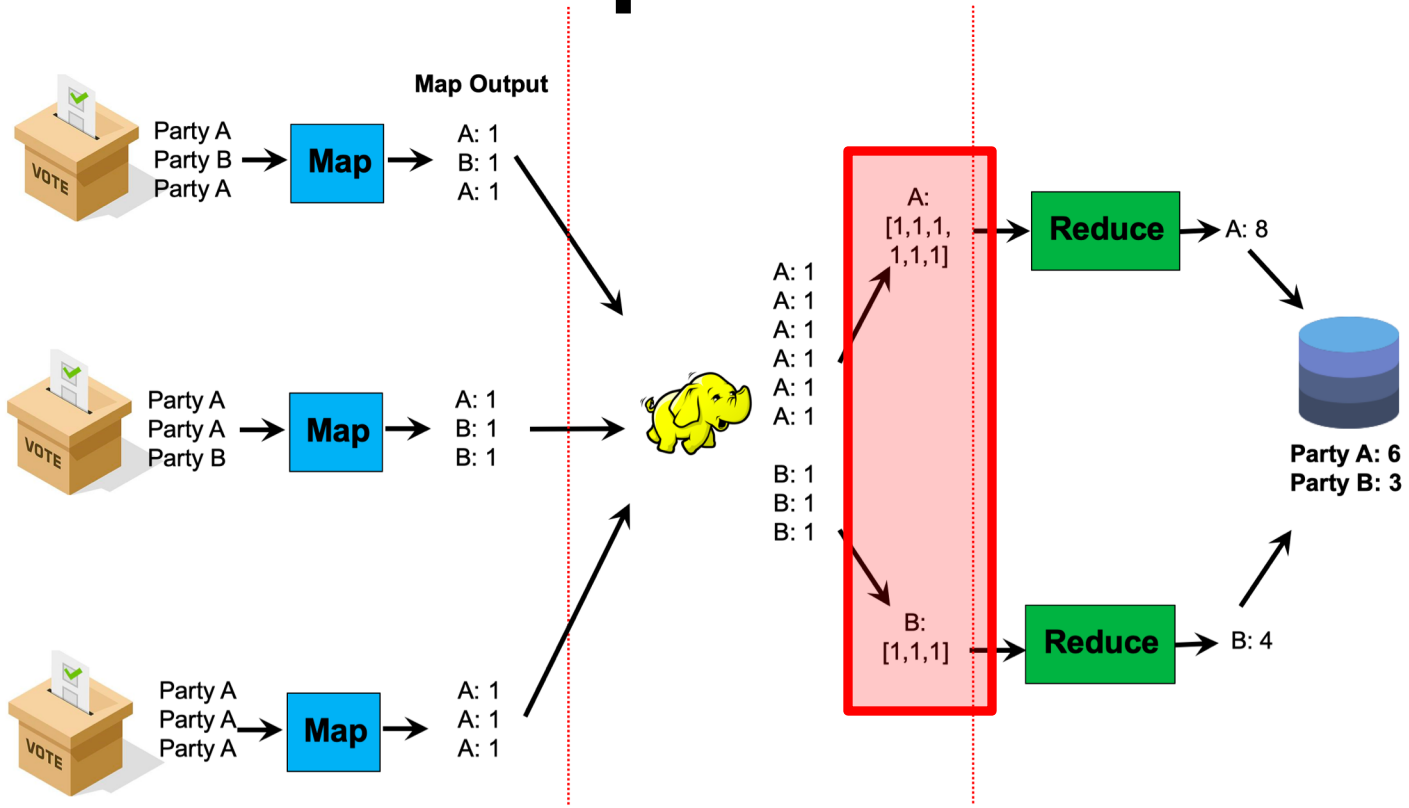
Partition Step



- 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

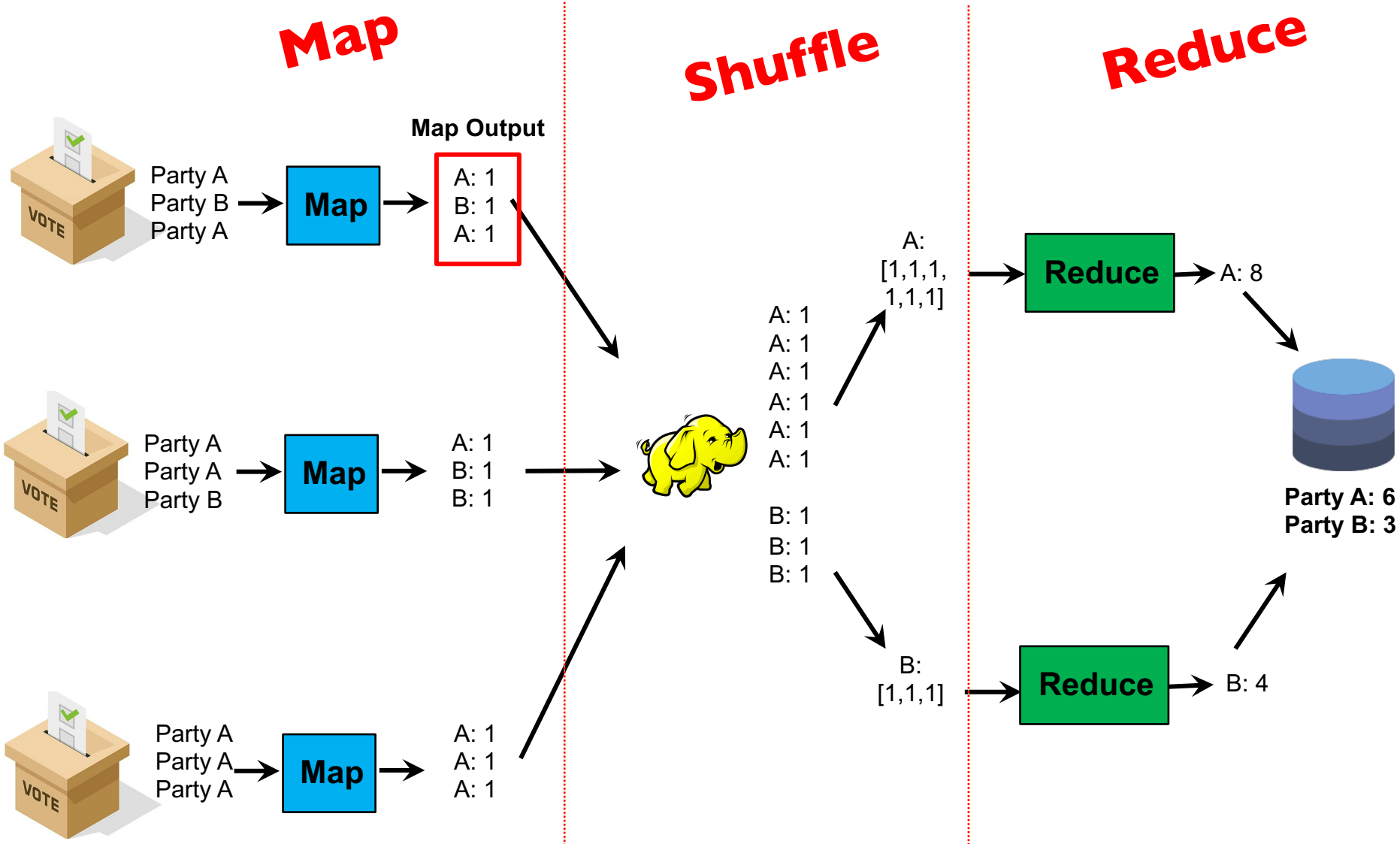


- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a **hash function**

  %

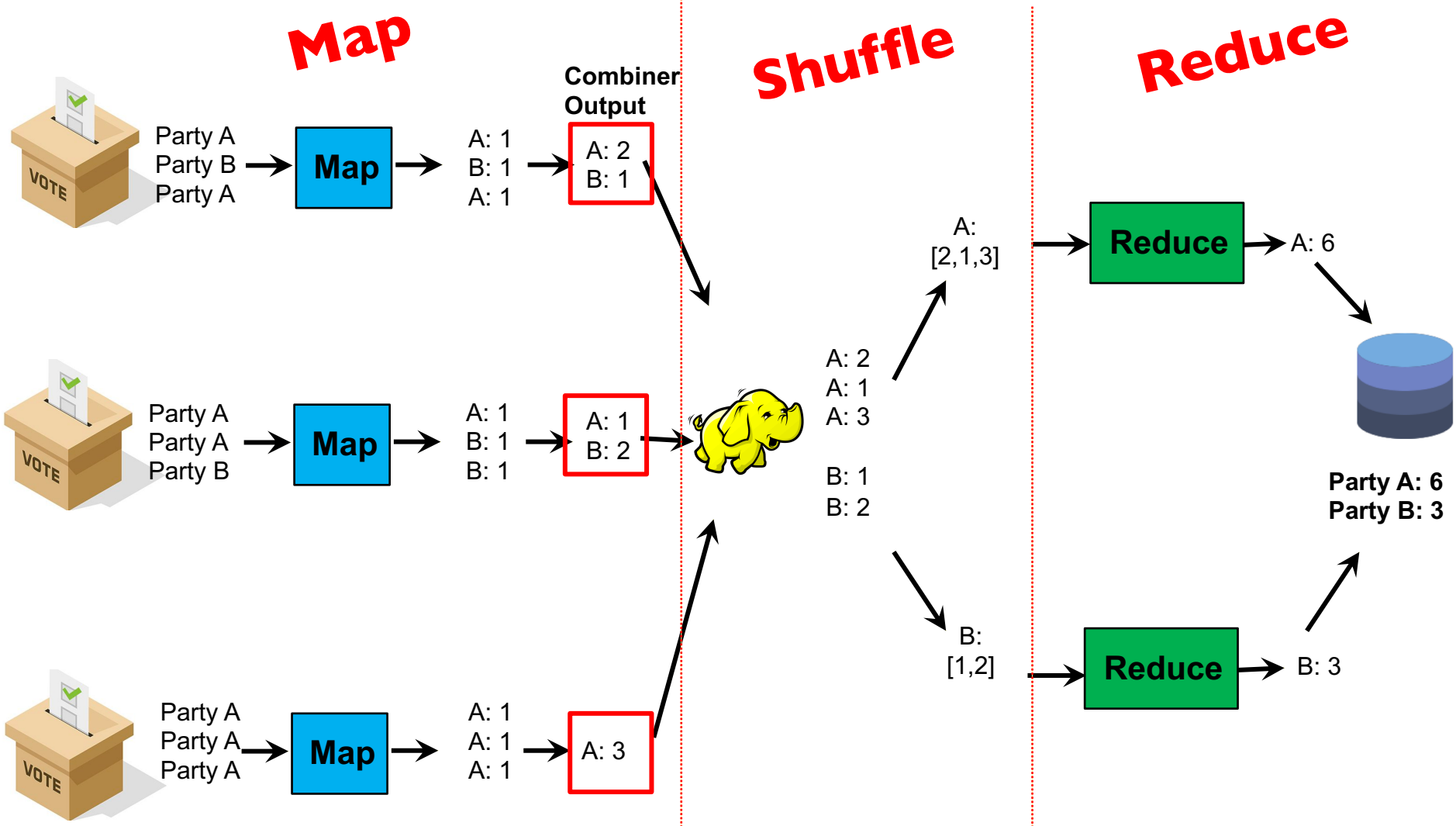
  - e.g., key *k* goes to reducer hash(*k*) mod *num_reducers*
- User can optionally implement a custom partition, e.g. to better spread out the load among reducers (if some keys have much more values than others)

Combiner Step



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

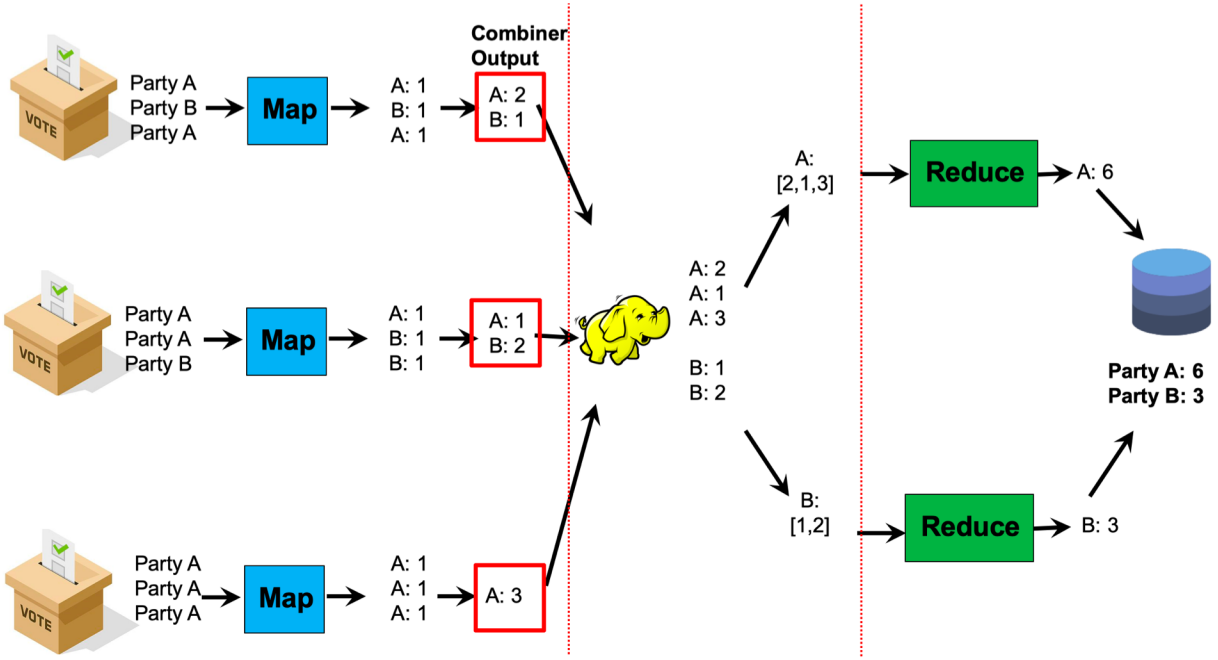
Combiner Step



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

Correctness of Combiner

- The user must 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 "average" or "minus"?



Correctness of Combiner

- The user must 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. Motivation
   b. Basic MapReduce
   c. Partition and Combiner
   d. Examples
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 the amount of I/Os in hard disk and network**
  - Minimize disk I/O; sequential vs. random.
  - Minimize network I/O; bulk send/recvs vs. many small send/recvs
- **Memory working set** of each task/worker
  - Large memory working set -> high probability of failures.
- Guidelines are applicable to Hadoop, Spark, …

Word Count: Version 0

The mapper emits an intermediate key-value pair for each word in an input document. The reducer sums up all counts for each word.

```
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 }
```

Slow, no combiners,
emit (a, 1), (a, 1), (a, 1,
.....

What's the key problem of this program?

What's the impact of combiners?

Word Count: Version 1

The mapper builds a histogram of all words in each input document before emitting key-value pairs for unique words observed.

```
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 }
```

Are combiners still of any use?

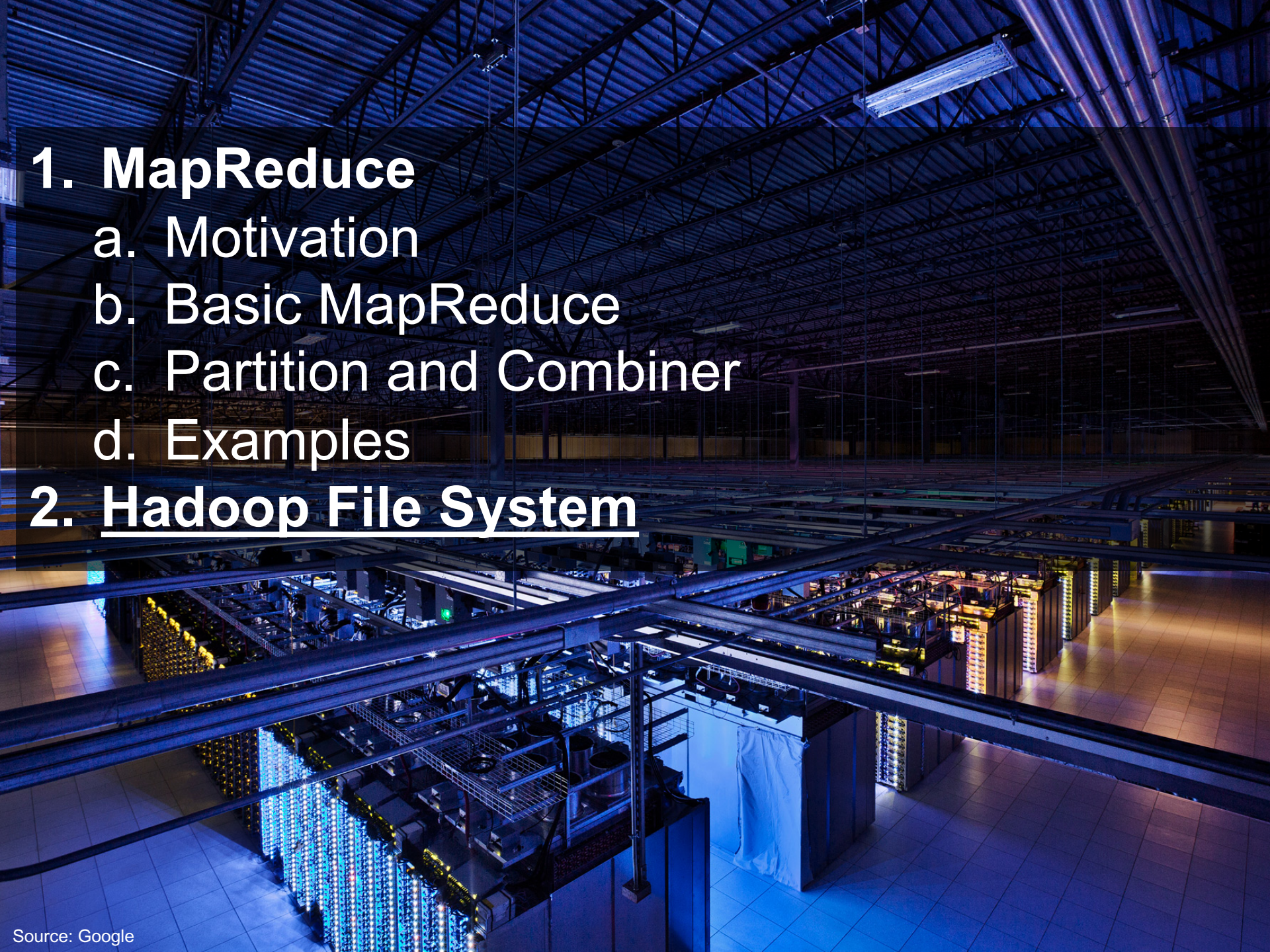
Word Count: Version 2

The mapper builds a histogram of all input documents processed before emitting key-value pairs for unique words observed.

```
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 }
```

Key idea: preserve state across input key-value pairs!

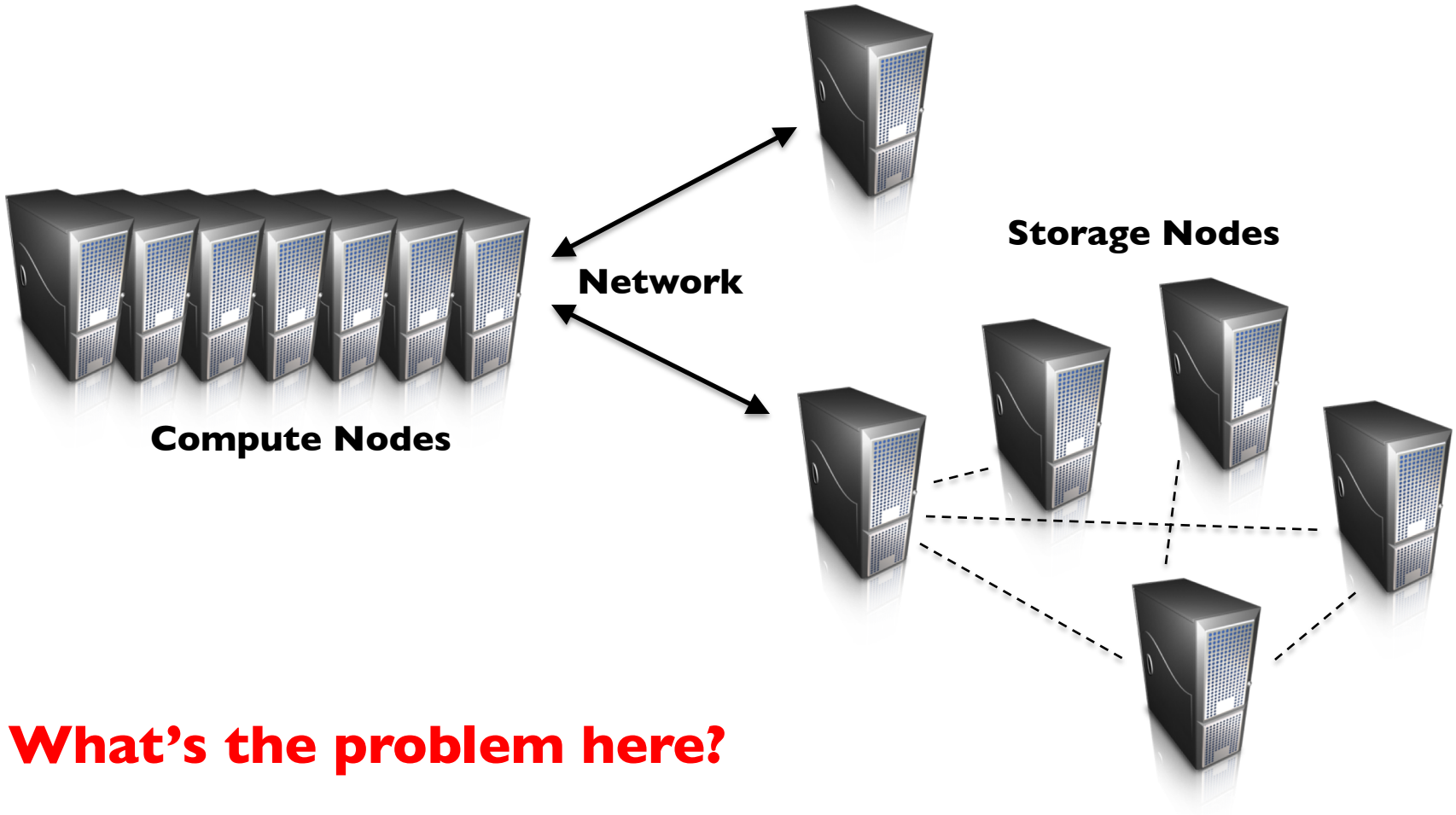
Are combiners still of any use?



1. **MapReduce**
   a. Motivation
   b. Basic MapReduce
   c. Partition and Combiner
   d. Examples
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: 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)

GFS: 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

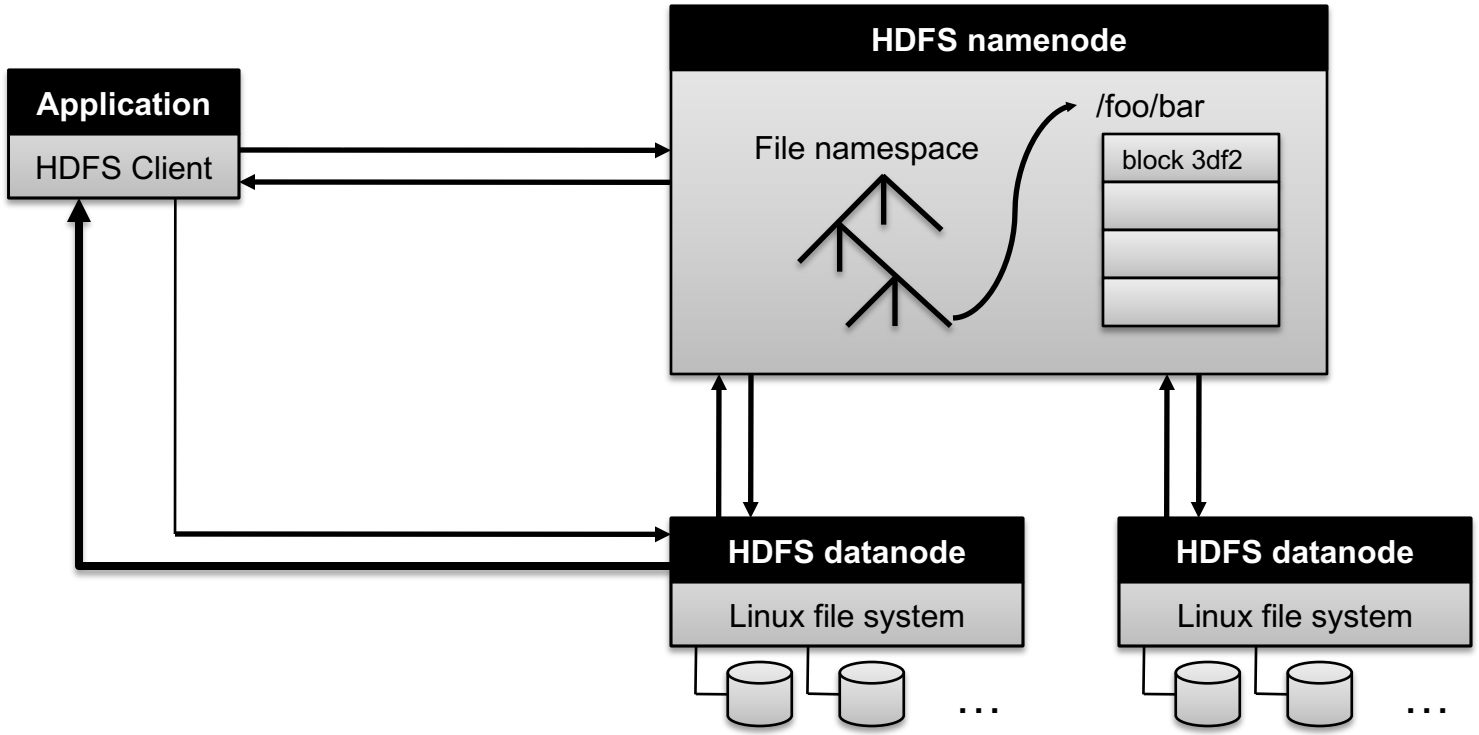
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology…

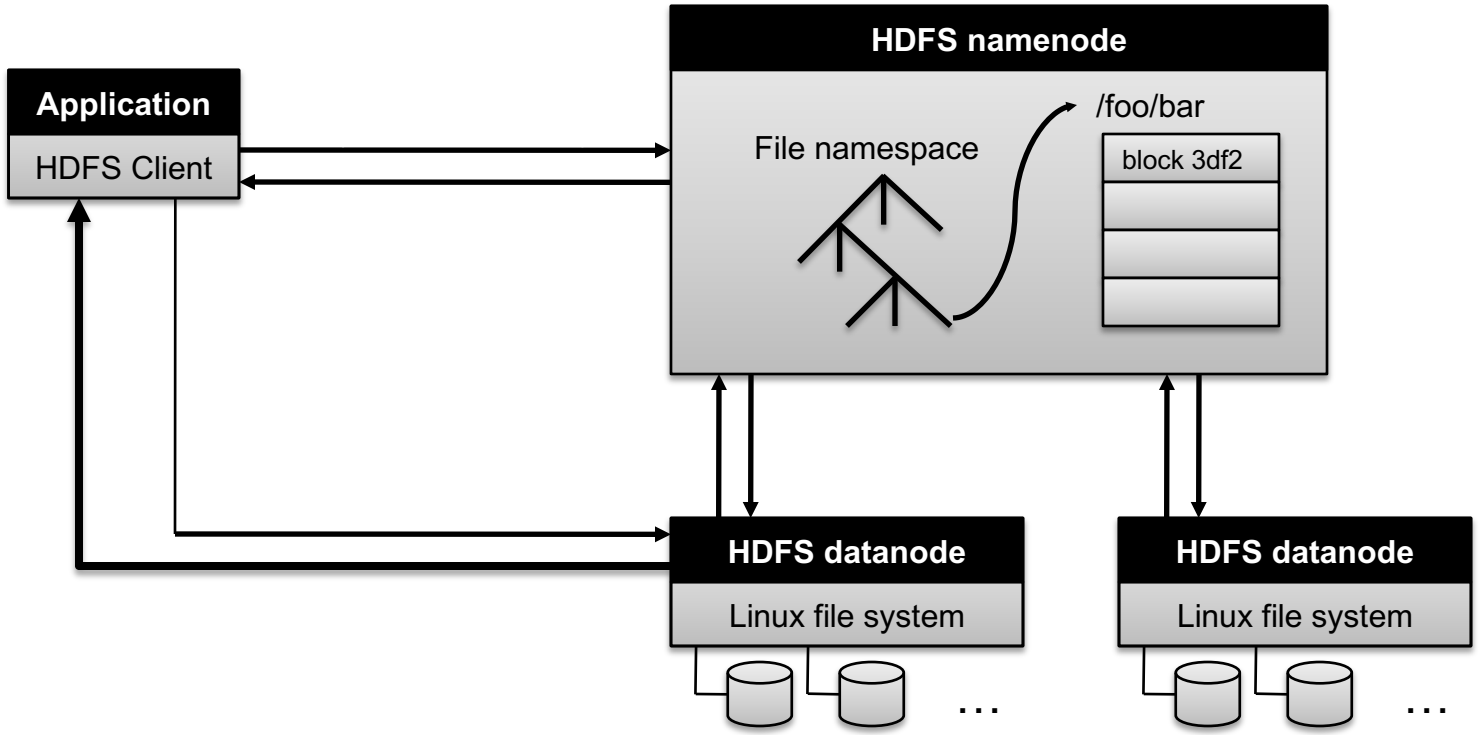
HDFS Architecture



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

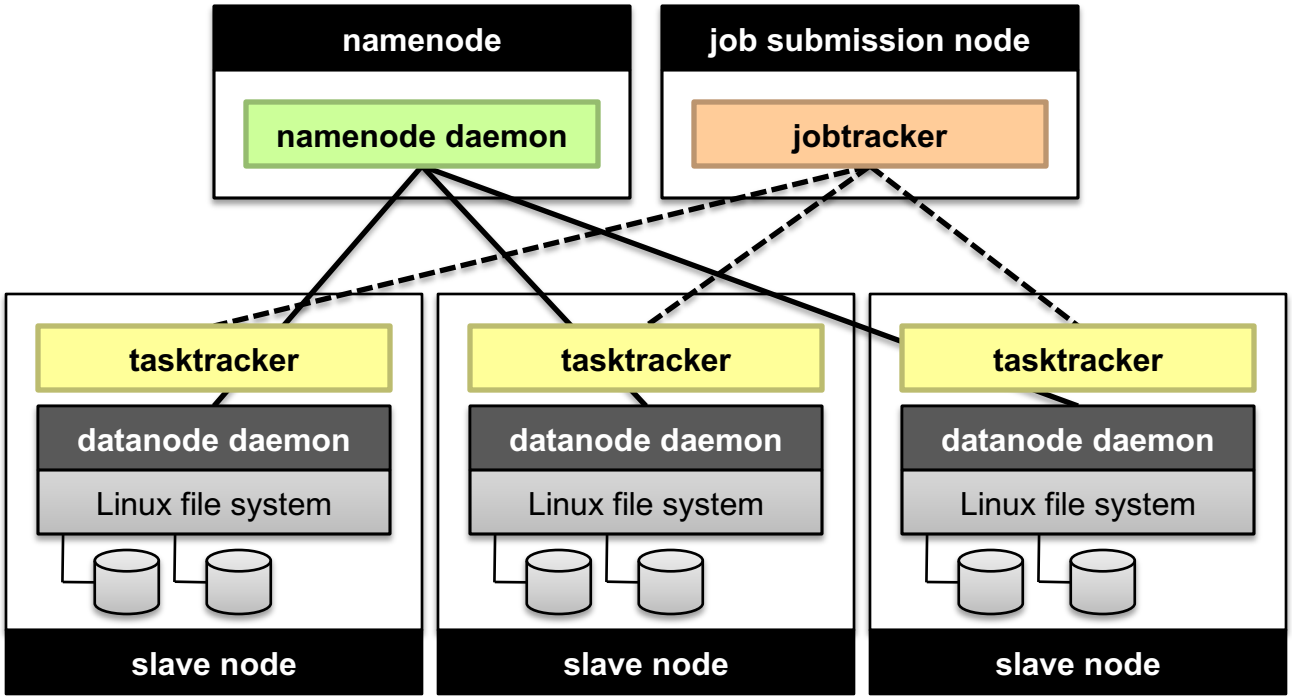
- **Q**: What if the namenode's data is lost?

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…



(Not Quite… We'll come back to YARN later)

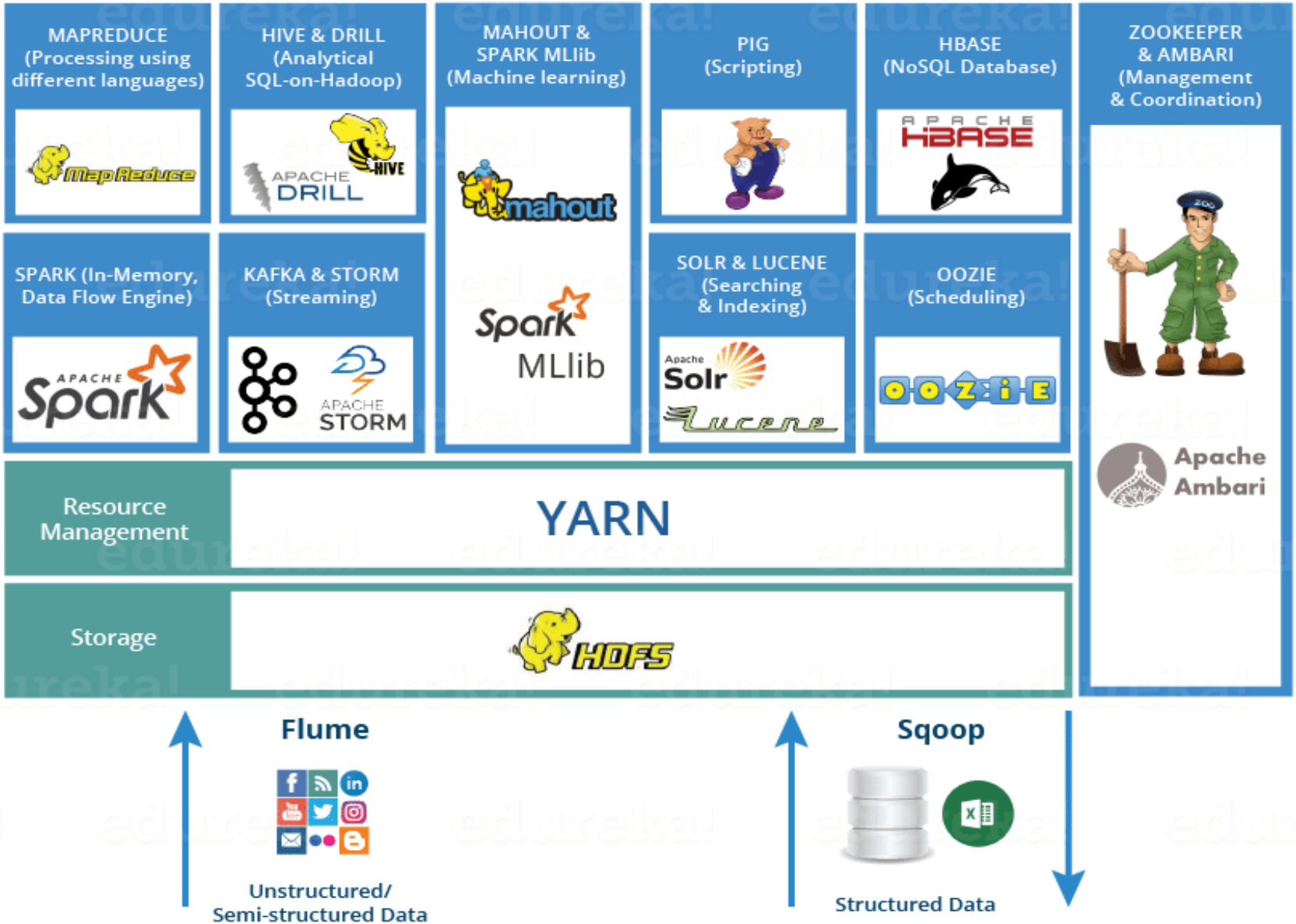
Further Reading

- "Mining Massive Datasets", Chapter 2.1-2.5
- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters
  - http://labs.google.com/papers/mapreduce.html
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
  - http://labs.google.com/papers/gfs.html

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.
  - Coming lectures: large relational database and data mining

Hadoop Ecosystem



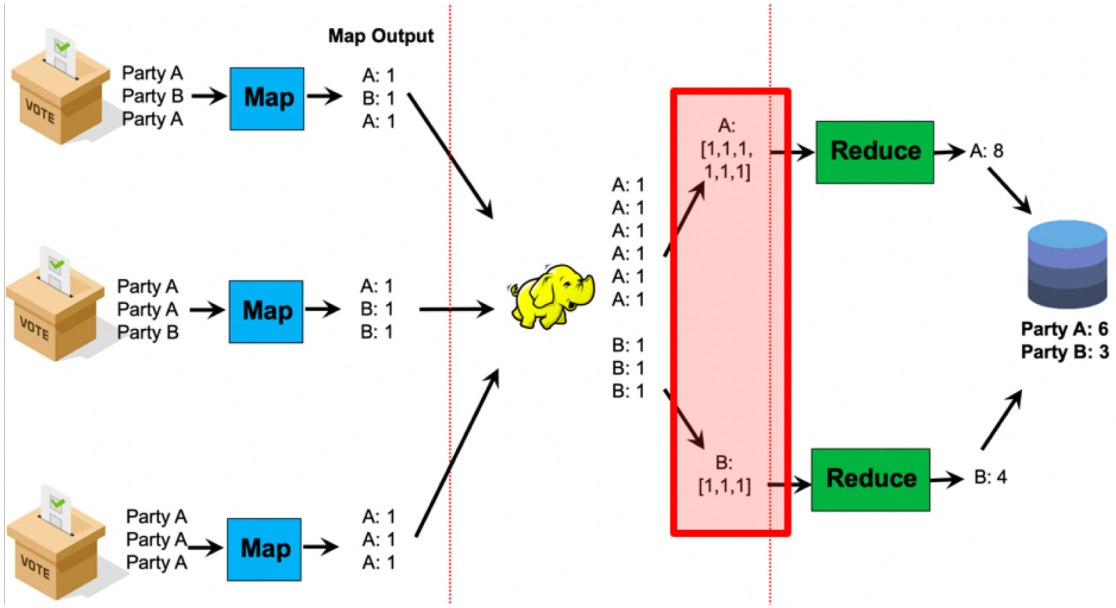
Q: Which statements about Hadooop's partitioner are correct?

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 statements about Hadooop's partitioner are correct?

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:** F, T, T



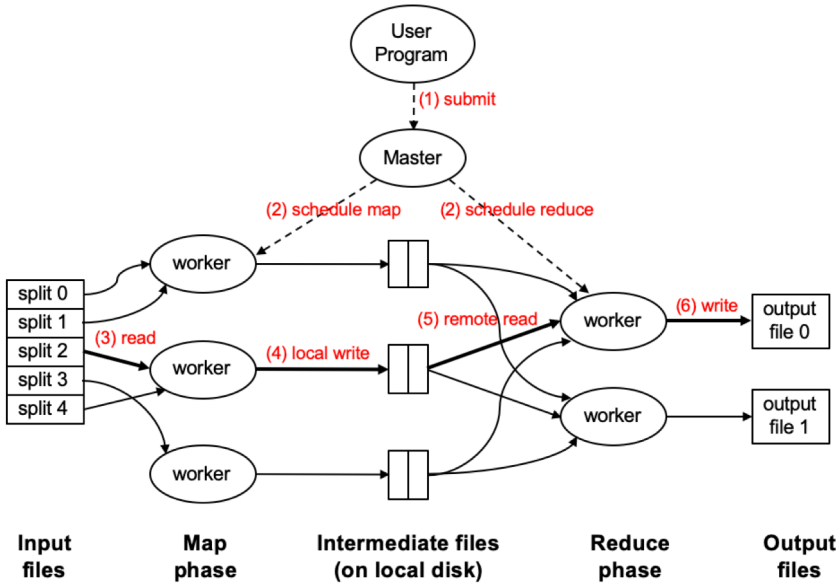
Q: True or false: the Shuffle stage of MapReduce is run within the Master node.

1. True
2. False

Q: True or false: the Shuffle stage of MapReduce is run within the Master 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

Resources

- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
  - http://people.apache.org/dist/lucene/hadoop/nightly/
- Source code from subversion
  - http://lucene.apache.org/hadoop/version_control.html

Acknowledgement

- Much of course slides are adopted/revised from
  - Jimmy Lin, http://lintool.github.io/UMD-courses/bigdata-2015-Spring/
- 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/