



Code Assignment 1

- Individual assignment.
- One TA is assigned to a group of 60-70 students so that you will have a better assistance.
 - Group 1 (TA: CHEN XIHAO) 70 students
 - Group 2 (TA: CHEN JIQING) 70 students
 - Group 3 (TA: Lim Jun Heng Edward) 70 students
 - Group 4 (TA: QUEK ZHI HENG) 70 students
 - Group 5 (TA: Chew Cheng Yap) 69 students
 - Group 6 (TA: Glenn Lim Jun Wei) 70 students
- Edward will present the details of the assignment to you in the tutorial after this lecture.
- Submission to Canvas

☰ ▾ Assignments	
☰ 	CodeAssignmentOne-VideoSubmission Due 27 Feb at 23:59 8 Pts
☰ 	CodeAssignmentOne-CodeSubmission Due 27 Feb at 23:59 17 Pts

Recap: Writing MapReduce Programs

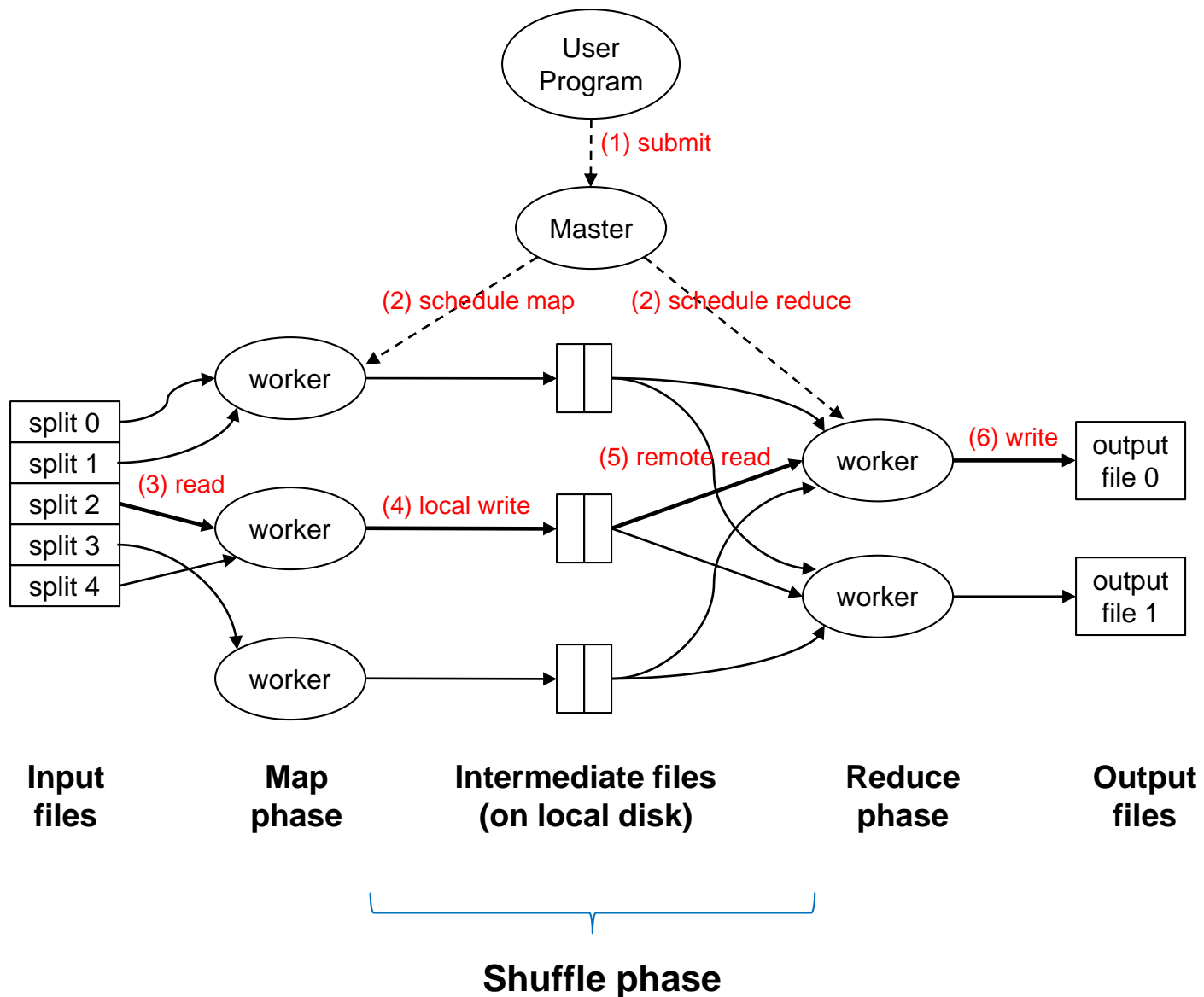
- Programmers specify two functions:

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

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

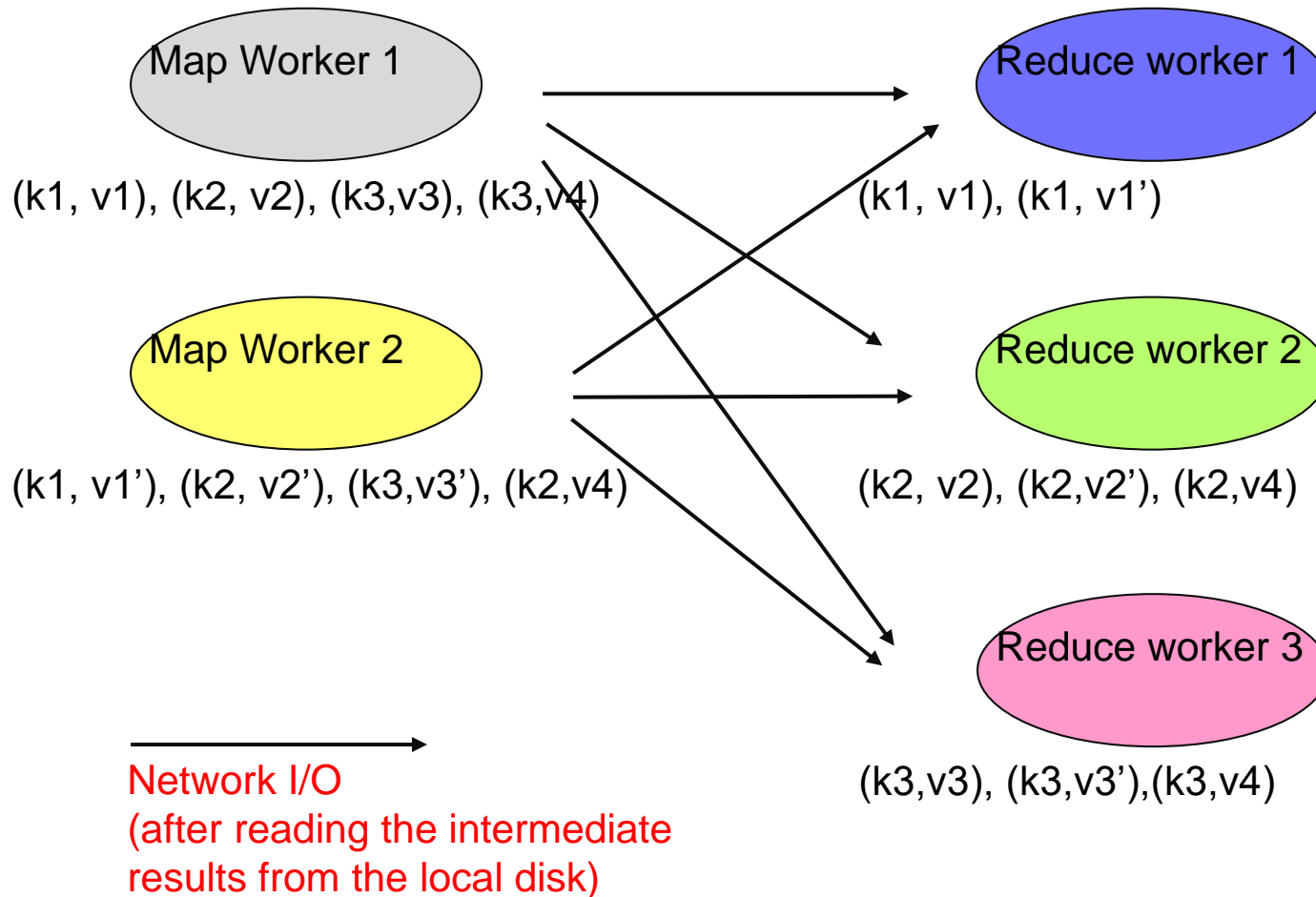
- All values with the same key are sent to the same reducer
- The execution framework handles the challenging issues...
 - How do we assign work units to workers?
 - What if we have more work units than workers?
 - What if workers need to share partial results?
 - How do we aggregate partial results?
 - How do we know all the workers have finished?
 - What if workers die/fail?

Recap: MapReduce Implementation



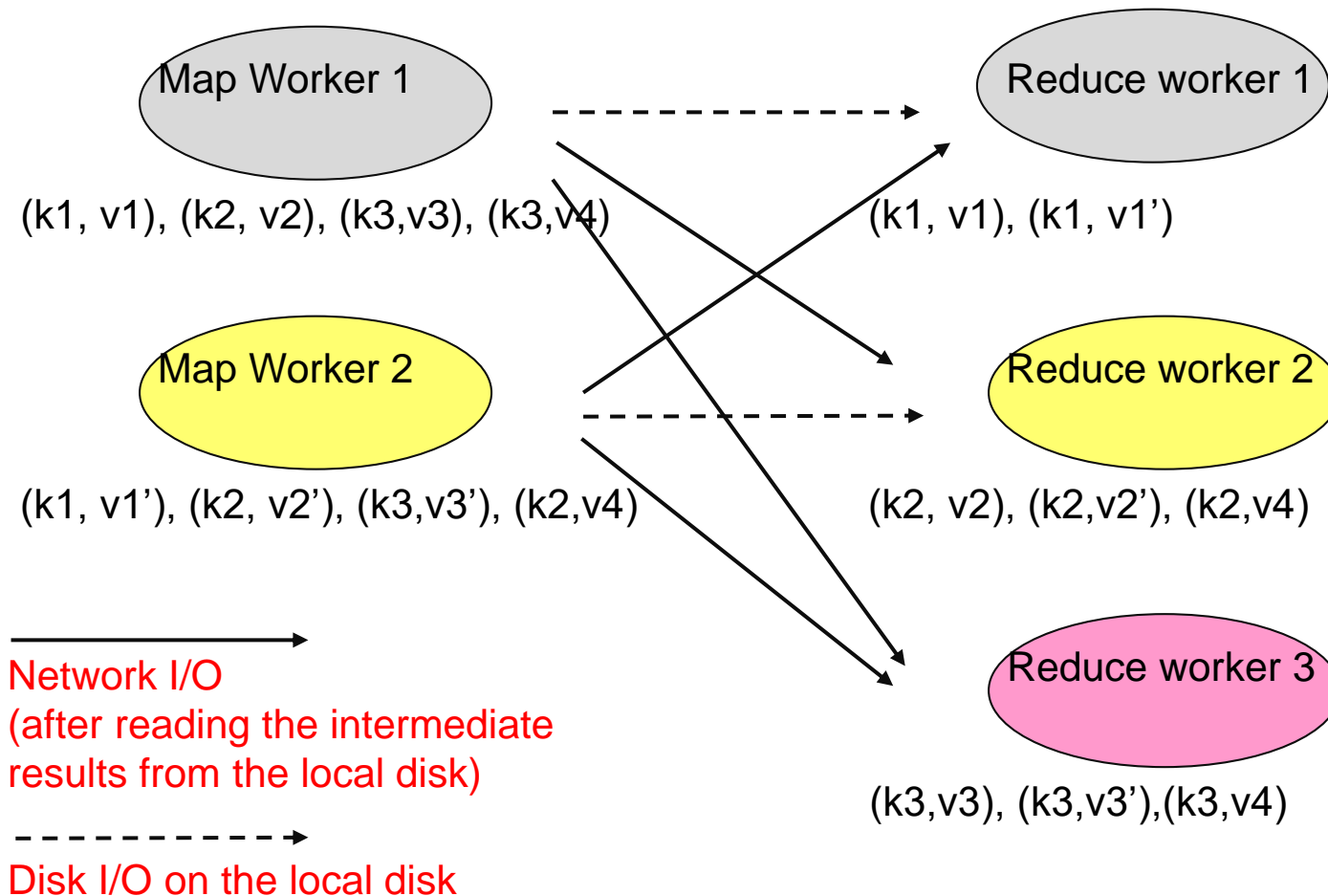
Recap: The amount of network I/O in shuffle

- Case 1: Map Workers are different to Reduce Workers

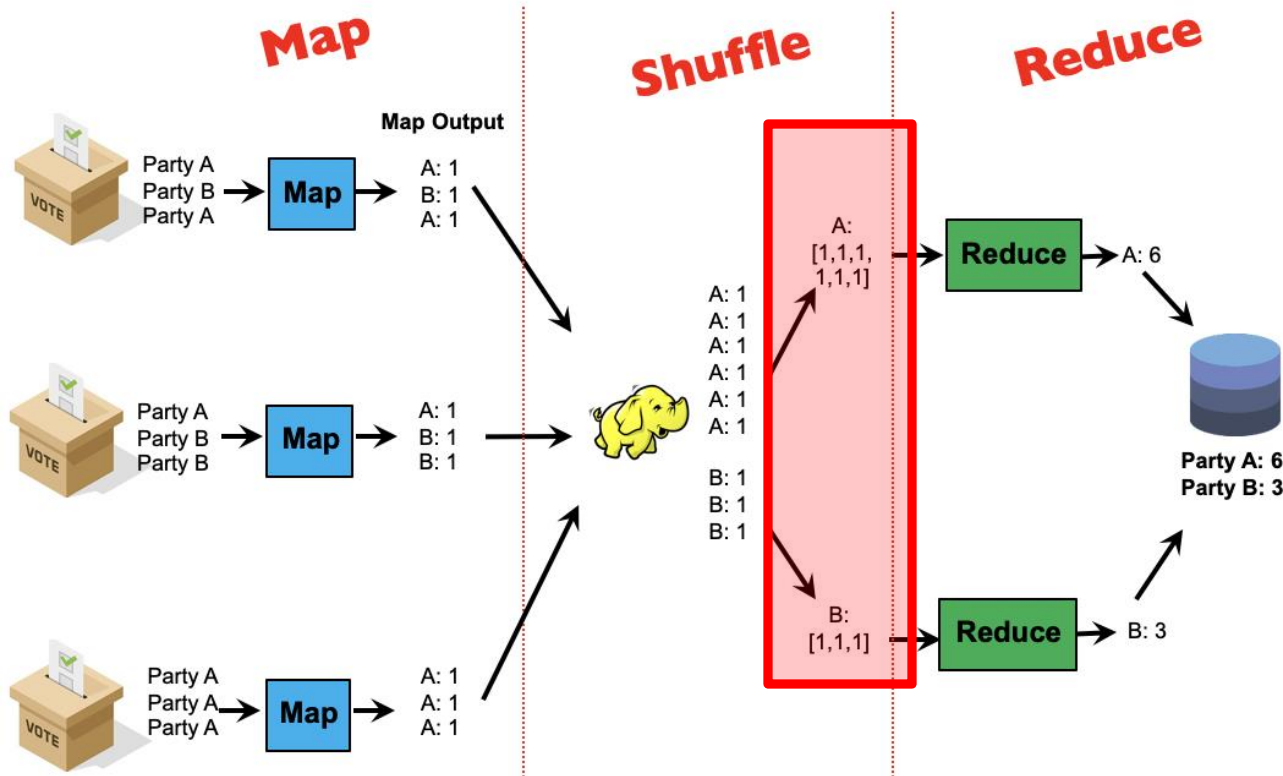


Recap: The amount of network I/O in shuffle

- Case 2: Two map Workers are reused in reduce phase.

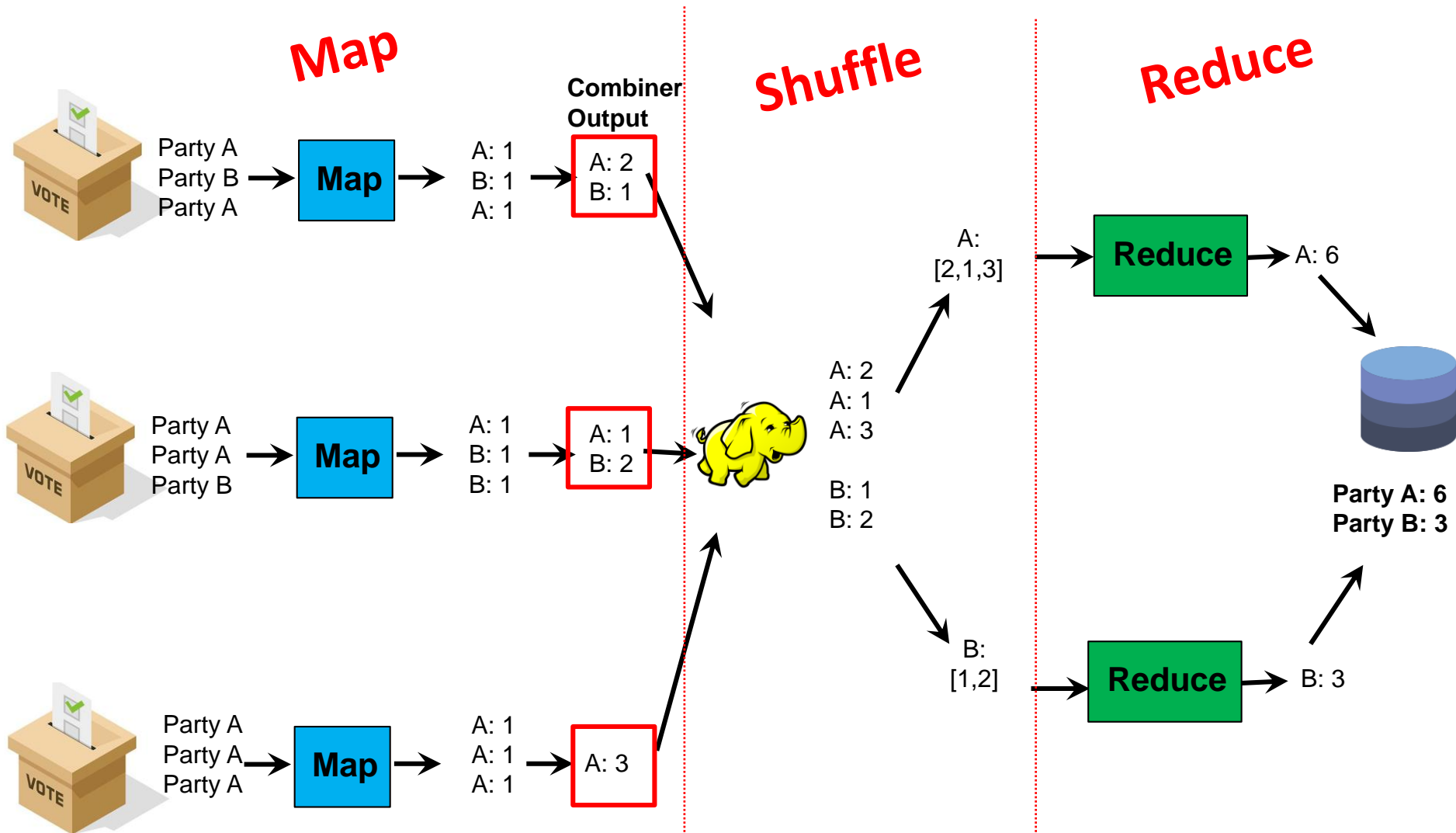


Recap: Partition Step



- 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: $(\text{hash}(k) \bmod \text{num_reducers})$
- 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)

Recap: Combiner Step



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

Recap: Distributed File System

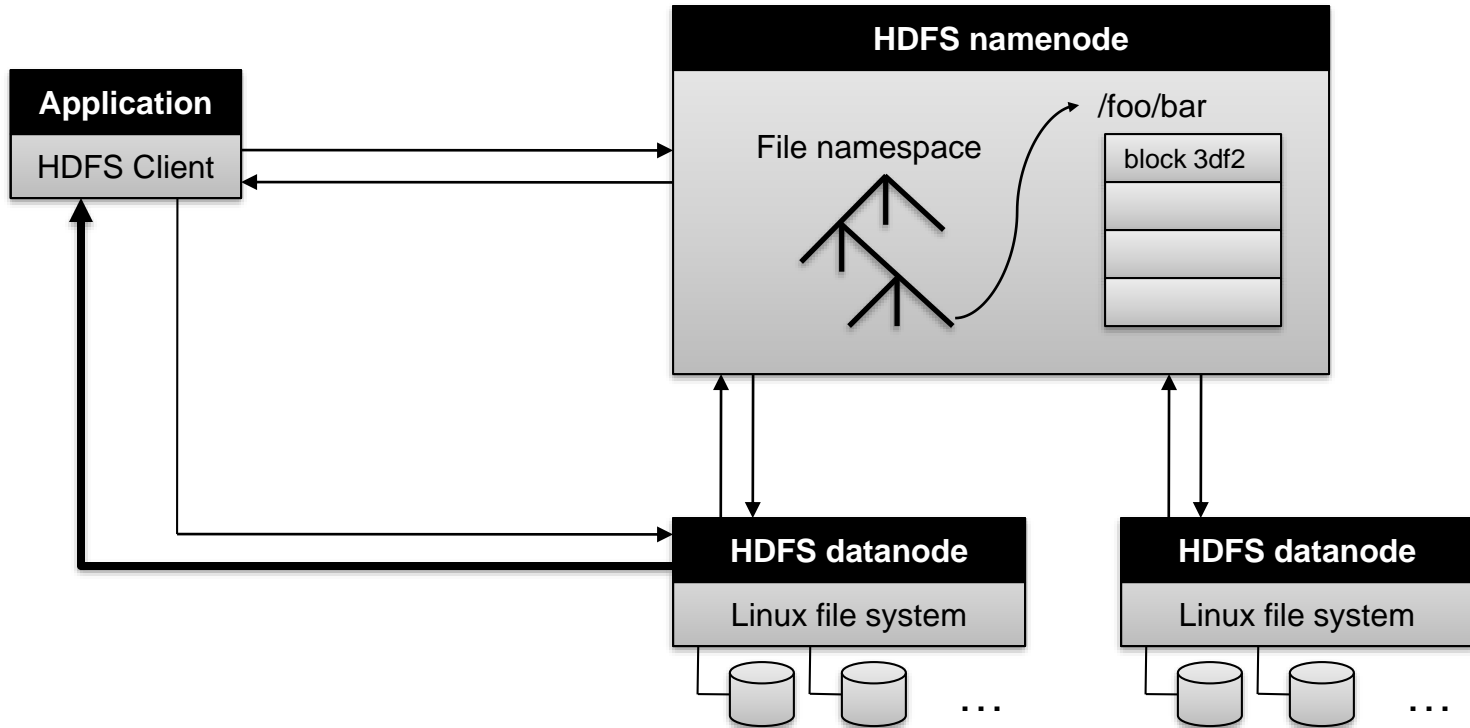
- Don't move data to workers... move workers to the data!
 - Store data on the local disks of nodes in the cluster
 - Start up the workers on the node that has the data local
- A distributed file system is the answer
 - GFS (Google File System) for Google's MapReduce
 - HDFS (Hadoop Distributed File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

Recap: 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)

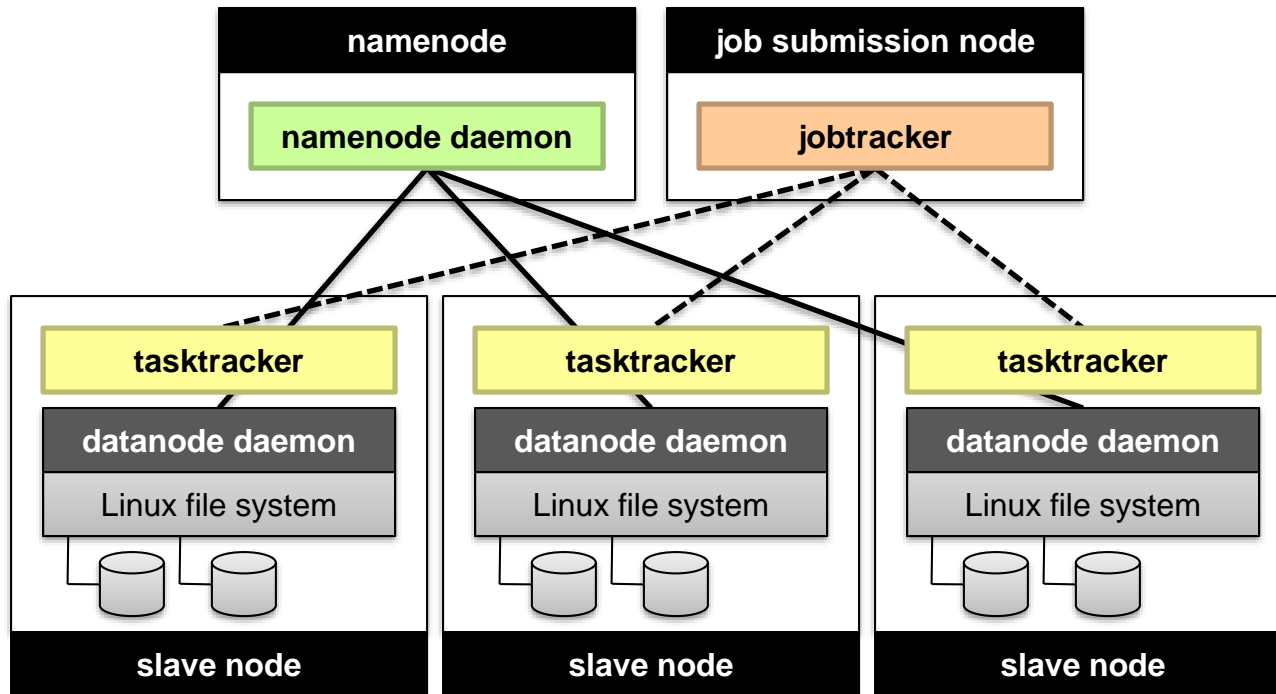
Recap: HDFS Architecture



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.

Recap: Putting everything together...



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



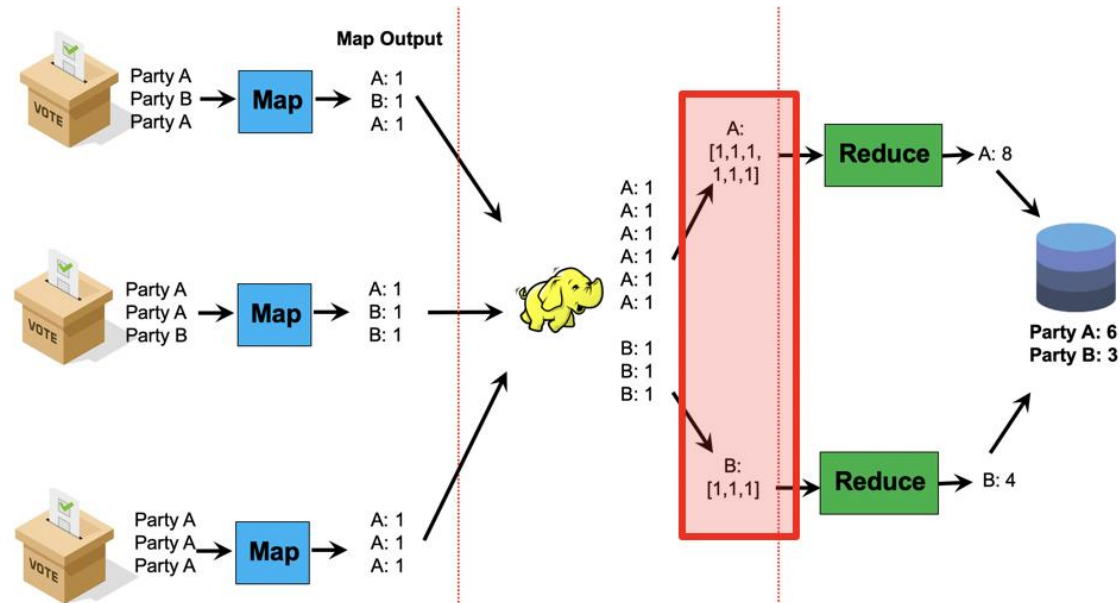
1. The partitioner determines which keys are processed on which mappers.
2. The partitioner can improve performance when some keys are much more common than others.
3. The partitioner is run in between the map and reduce phases.

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 run within the Master node.

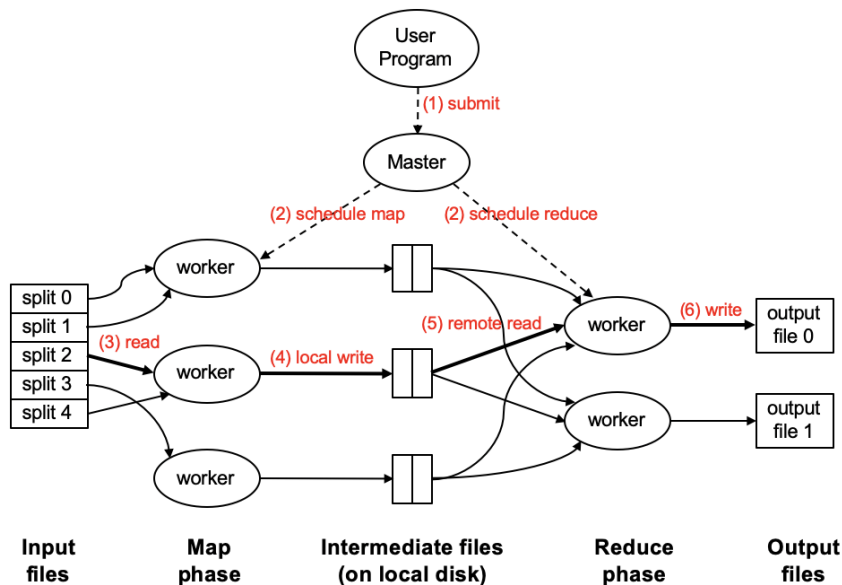
1. True
2. False



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)



CS4225/CS5425 Big Data Systems for Data Science

Performance Analysis of Big Data Systems: MapReduce

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



Learning Outcomes

- Given a big data system program (here MapReduce/Hadoop as an example), you will learn about:
 - How to analyze scalability
 - How to analyze disk I/O and network I/O performance
 - How to analyze memory consumption
 - How to analyze load balance

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 memory requirements / probability of out-of-memory errors.
- **Load balance among tasks**
 - Load imbalance -> long execution time / large memory working set.
- Guidelines are applicable to Hadoop, Spark, ...

A Step-by-Step Performance Analysis Guide

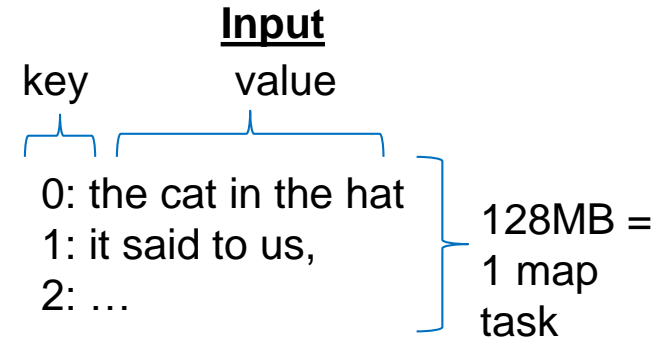
- Scalability analysis
 - **Max** number of Map tasks: Will the number of mapper tasks increase linearly as the input size increases?
 - **Max** number of Reduce tasks: Will the number of reducer tasks increase linearly as the input size increases?
- I/O analysis: input + intermediate results + output
 - The amount of disk I/O from each Map/Reduce task
 - The amount of network I/O from each Map/Reduce task
 - The amount of network I/O in shuffle (= amount of intermediate results from map tasks)
- Memory working set: intermediate results
 - The amount of memory consumption from each map/reduce task
- Load balance: **worst case analysis**
 - The worst execution time for all map/reduce tasks
 - The largest memory working set for all map/reduce tasks

Learn by Example

- We will use word count as an example
- To go through each step of performance analysis
- To identify the performance bottleneck
- To develop solutions to address the performance bottleneck
- Let's assume normal execution (no machine failure or slowdown).

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  }
```

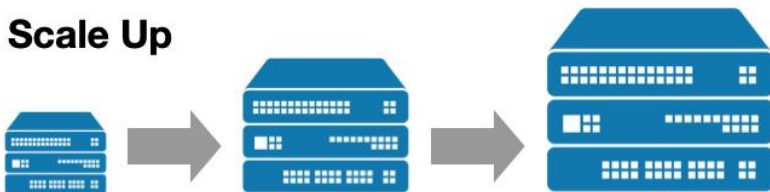


This mapper processes each word one by one, and emits a “1”, to be summed by the reducers.

Recap: Scale “out”, not “up”

- Scaling up = adding further resources, like hard drives and memory, to increase the computing capacity of physical servers.
- Scaling out = adding more servers to your architecture to spread the workload across more machines.

Scale Up

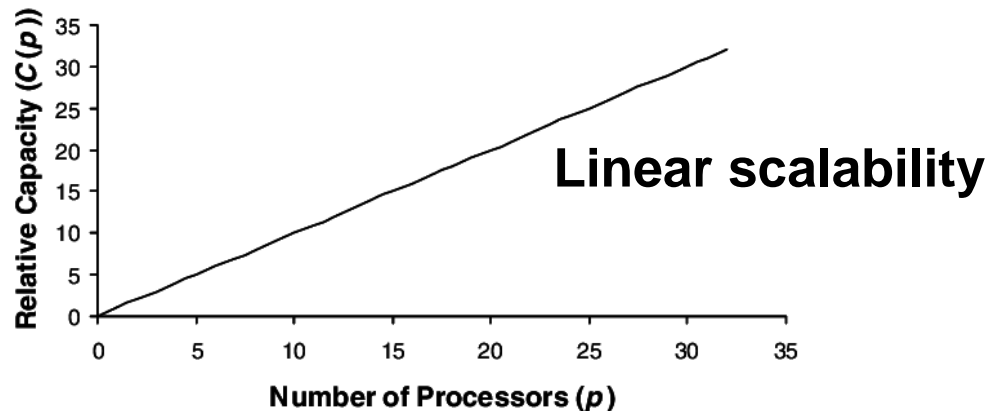


Scale Out



Recap: Seamless Scalability

- Think about aggregated bandwidth
 - 1 machine: 200MB/sec disk, 20GB/sec DRAM
 - 10 machine: 2 GB/sec disk, 200 GB/sec DRAM
 - 100 machine: 20 GB/sec disk, 2 TB/sec DRAM
- If our design scales linearly to #machine, we can trade more machines with better performance.



(Optional/Re-cap): Amdahl's law

- Amdahl's law is a formula that gives the theoretical speedup for executing at fixed workload on a system whose resources could be improved.

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

- S_{latency} is the theoretical speedup of the execution of the whole task;
 - s is the speedup of the part of the task that benefits from improved system resources;
 - p is the proportion of execution time that the part benefiting from improved resources originally occupied.
- Linear scalability: $p=1$, $\Rightarrow S_{\text{latency}} = s$ (*equal to #workers*).

Scalability Analysis

- Assume that one worker can run one Map or Reduce task
- Linear scalability:
 - Given W workers, we can run W tasks at the same time.
 - The key question will be: how many tasks does the job have?
- We calculate:
 - **Max** number of Map tasks
 - **Max** number of Reduce tasks
- We analyze:
 - Will the number of mapper tasks increase linearly as the input size increases?
 - Will the number of reducer tasks increase linearly as the input size increases?

Word Count V0: Scalability Analysis

```
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 }
```

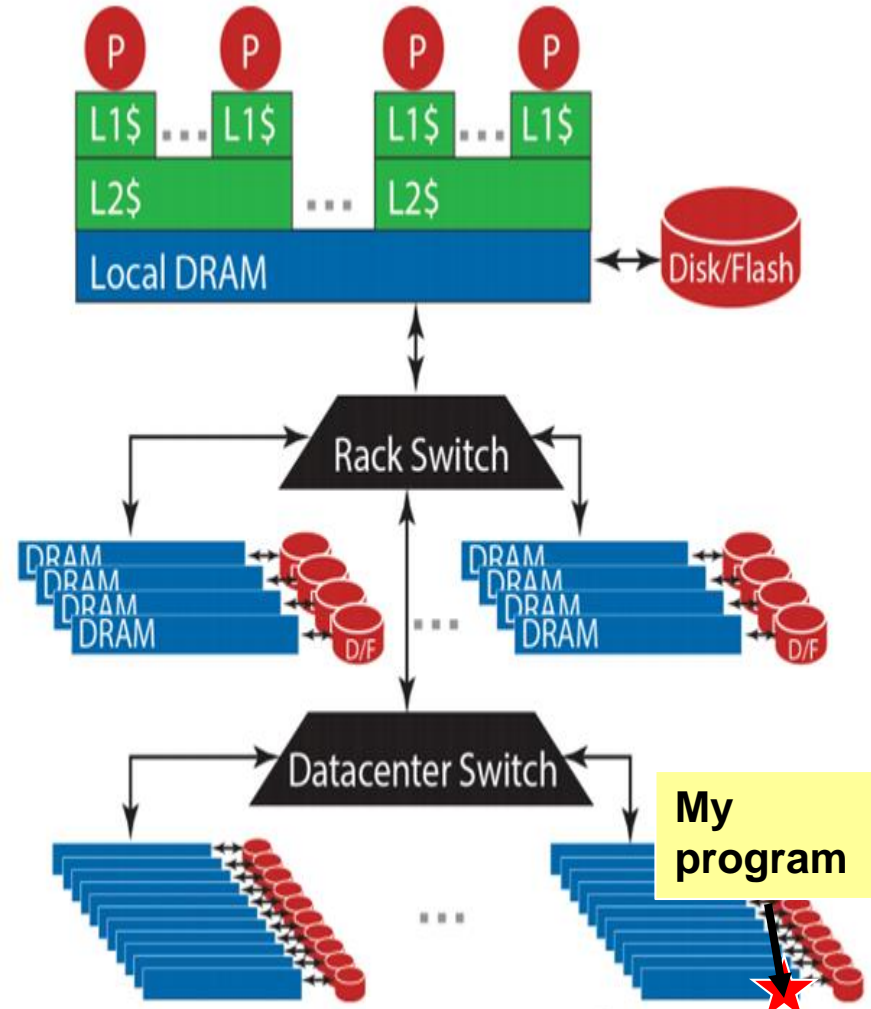
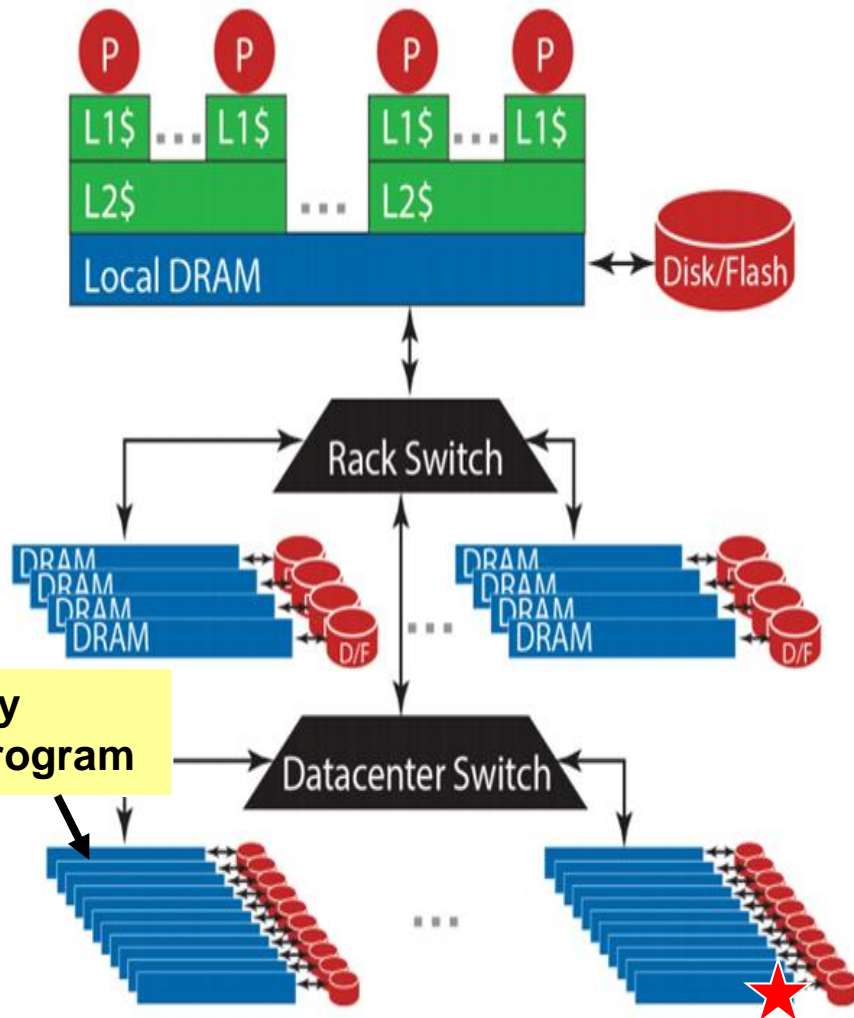
Scalability analysis for Map:

* Max number of map tasks
= input size / chunk size

Scalability analysis for Reduce:

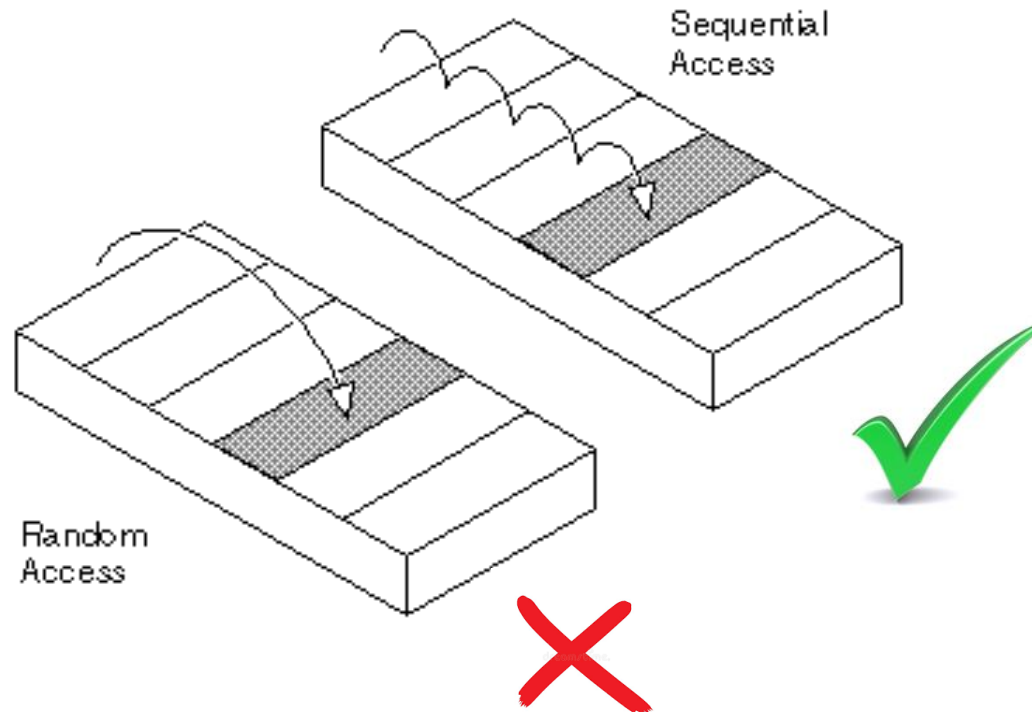
* Max number of reduce tasks
= **number of distinct keys**

Recap: Move processing to the data



Recap: Sequential Accesses vs. Random Accesses

- Take hard disk as an example
- Random access: 10ms for 4KB = 400KB/sec
- Sequential access: 200MB/sec



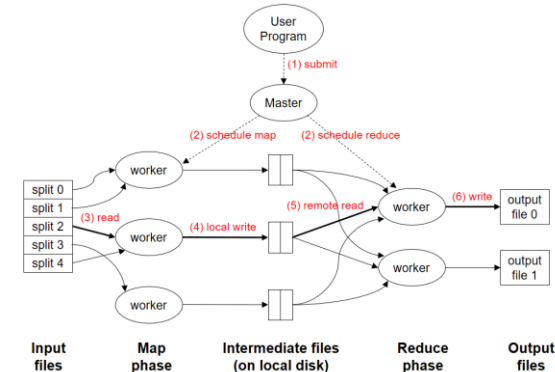
I/O Analysis

- For a Mapreduce job, we have the following I/O components:

- Reading input from HDFS: mainly disk I/O
- Shuffle and sort: Disk and network I/O
- Output: Disk and network I/O

- We calculate:

- The amount of disk I/O from each Map/Reduce task
- The amount of network I/O from each Map/Reduce task
- The amount of network I/O in shuffle (= amount of intermediate results from map tasks)



Word Count V0: I/O Analysis

```
1 class Mapper {  
2     def map(key: Long, value: Text) = {  
3         for (word <- tokenize(value)) {  
4             emit(word, 1)  
5         }  
6     }  
7 }
```

I/O analysis for map task:

- Input Disk I/O= **128MB**
- Intermediate results = very small (i.e., can be ignored)
- Output Disk I/O= **all <word, 1> pairs, #pairs= #words in the chunk**
- Network I/O= very small

I/O analysis for shuffling:

Network I/O = **all <word, 1> pairs, #pairs= #words in the chunk**

Word Count V0: I/O Analysis

```
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 }
```

I/O analysis for reduce task:

- Input Disk I/O= very small //already counted in shuffling
- Intermediate results = very small
- Output Disk I/O= very small
- Network I/O= very small

Memory Consumption Analysis

- For Map/Reduce function, look for the memory allocation:
 - Variables
 - Intermediate data structures
- We calculate:
 - The total amount of memory consumption by all those variables/data structures.

Word Count V0: Memory Consumption

```
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 }
```

Memory analysis for Map:

*** Memory working set
= very small**

Memory analysis for Reduce:

*** Memory working set
= very small**

Load Balance

- Avoid unevenly overloading some compute nodes while other compute nodes are left idle.
- Cannikin Law (“Buckets effect”)
- Worst case analysis for all map/reduce tasks
 - The worst execution time (sensitive to input)
 - The largest memory consumption



Word Count V0: Worst Case Analysis

```
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 }
```

Worst case analysis for Map:

- * Memory working set
= Very small
- * Execution time
= Parsing the chunk

Worst case analysis for Reduce:

- * Memory working set
= Very small
- * Execution time
= Parsing the entire input
(all documents have the same word)

Summary of Issues in Word Count Version 0

- Scalability

- Max number of reducer tasks= **number of distinct keys**

- I/O

- Map: Output Disk I/O= **all <word, 1> pairs, #pairs= #words in the chunk**
 - Shuffle: Network I/O= **all <word, 1> pairs, #pairs= #words in the chunk**

- Memory working set

- No Issue

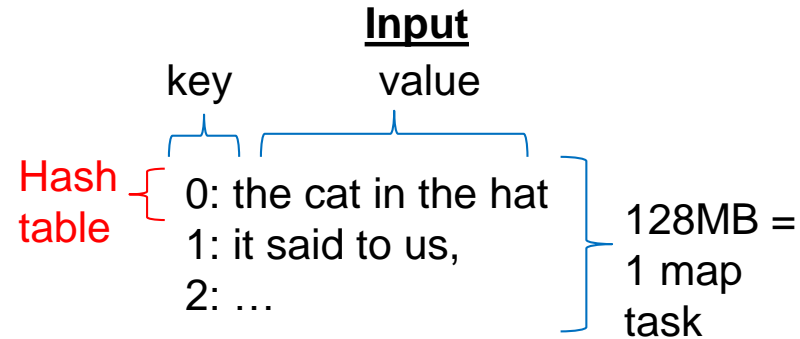
- Load balance

- Reduce: Execution time= Parsing the entire input (all documents have the same word)

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.

Let's conduct performance analysis on Map.

Word Count V1: Scalability Analysis

```
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 }
```

Scalability analysis for Map:

*** Max number of map tasks**

= input size / chunk size

Word Count V1: I/O Analysis

```
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 }
```

I/O analysis for map task:

- Input Disk I/O= **128MB**
- Intermediate results = very small
- Output Disk I/O= **all <word, count> pairs, #pairs= #distinct words in the chunk**
- Network I/O= very small

I/O analysis for shuffling:

Network I/O = all <word, count> pairs, #pairs= #distinct words in the chunk

Word Count V1: Memory Consumption

```
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 }
```

Memory analysis for Map:

* Memory working set

= hash table size

~#distinct words in the chunk

Word Count V1: Worst Case Analysis

```
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 }
```

Worst case analysis for Map:

* Memory working set

= hash table size

~ #words in the chunk (all words are different)

* Execution time

= Parsing the chunk

Summary of Issues in Word Count V1 vs. V0

○ V0:

- I/O
 - Map: Output Disk I/O= all <word, 1> pairs, #pairs= #words in the chunk
 - Shuffle: Network I/O= all <word, 1> pairs, #pairs= #words in the chunk
- Memory working set
 - No Issue
- Load balance
 - No issue in Map

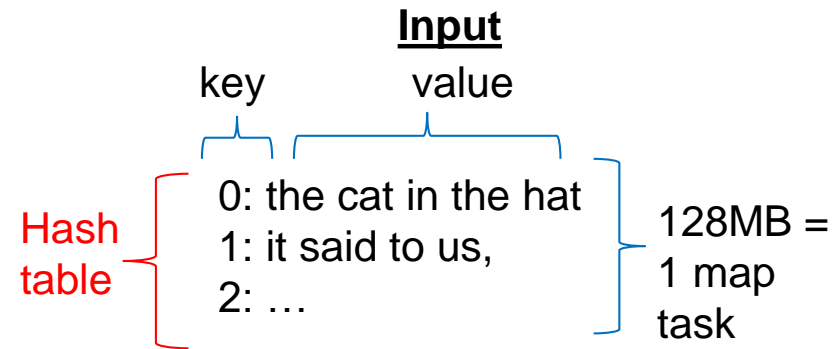
○ V1:

- I/O (much better than V0)
 - Map: Output Disk I/O= all <word, count> pairs, #pairs= #distinct words in the chunk
 - Shuffle: Network I/O = all <word, count> pairs, #pairs= #distinct words in the chunk
- Memory working set (more than V0 but in practice okay)
 - Map: Memory working set= hash table size ~#distinct words in the chunk
- Load balance (more than V0 but in practice okay)
 - Worst case analysis for Map: * Memory working set= hash table size ~ #words in the chunk (all words are different)

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 }
```



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

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

Recap: A Step-by-Step Performance Analysis Guide

- Scalability analysis
 - **Max** number of Map tasks: Will the number of mapper tasks increase linearly as the input size increases?
 - **Max** number of Reduce tasks: Will the number of reducer tasks increase linearly as the input size increases?
- I/O analysis: input + intermediate results + output
 - The amount of disk I/O from each Map/Reduce task
 - The amount of network I/O from each Map/Reduce task
 - The amount of network I/O in shuffle (= amount of intermediate results from map tasks)
- Memory working set: intermediate results
 - The amount of memory consumption from each map/reduce task
- Load balance: **worst case analysis**
 - The worst execution time for all map/reduce tasks
 - The largest memory working set for all map/reduce tasks

In Practice...

- Case 1: a Hadoop job runs very slow
 - Where is the performance bottleneck?
- Case 2: for the same input, a Hadoop job runs well in most cases, but fails sometime
 - OoM?
- Case 3: a Hadoop job runs well in some input, but fails in other input.
 - Worst case analysis?

Take-away

- Big data systems may not automatically bring performance benefits
- Performance analysis and algorithm design are still needed to efficiently and effectively develop various applications.
 - Coming lectures: large relational database and data mining

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

Thank you!



*“Yes, I have made a strategic decision;
I’ve decided to ignore the bad news...”*