

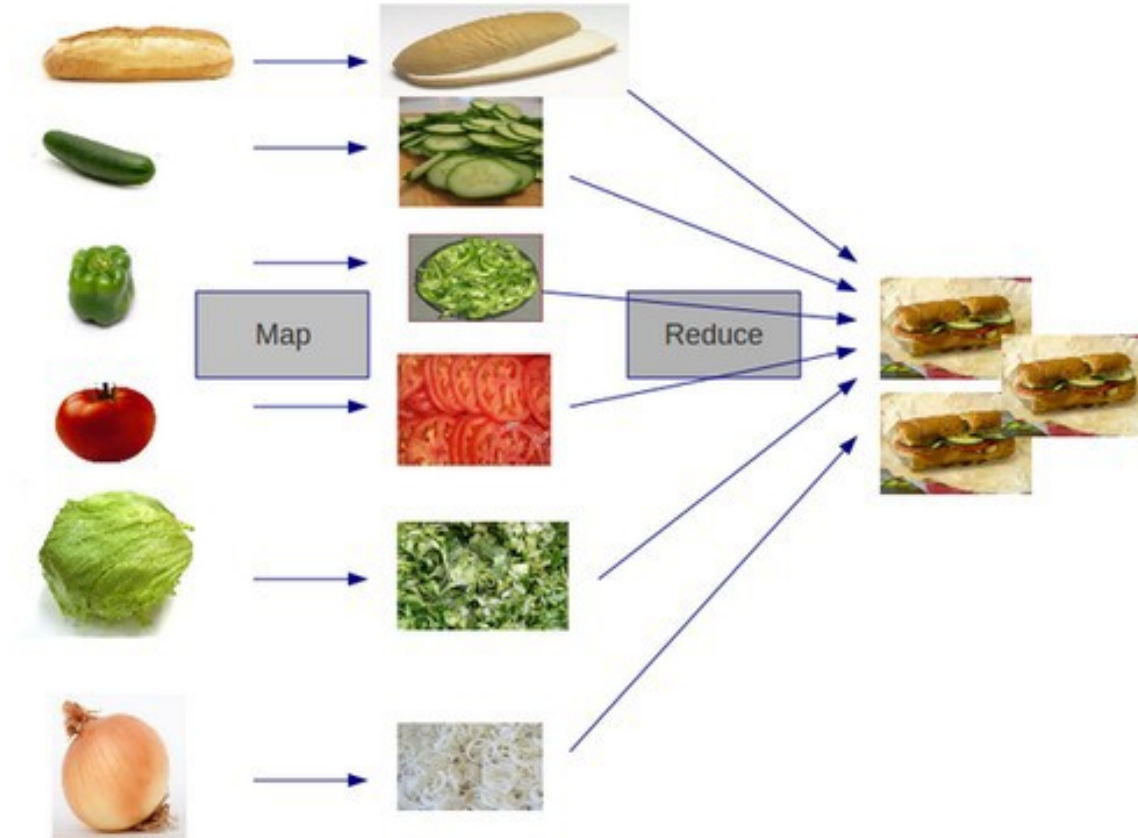
MapReduce

A Processing Framework for Massive Parallelism

MapReduce

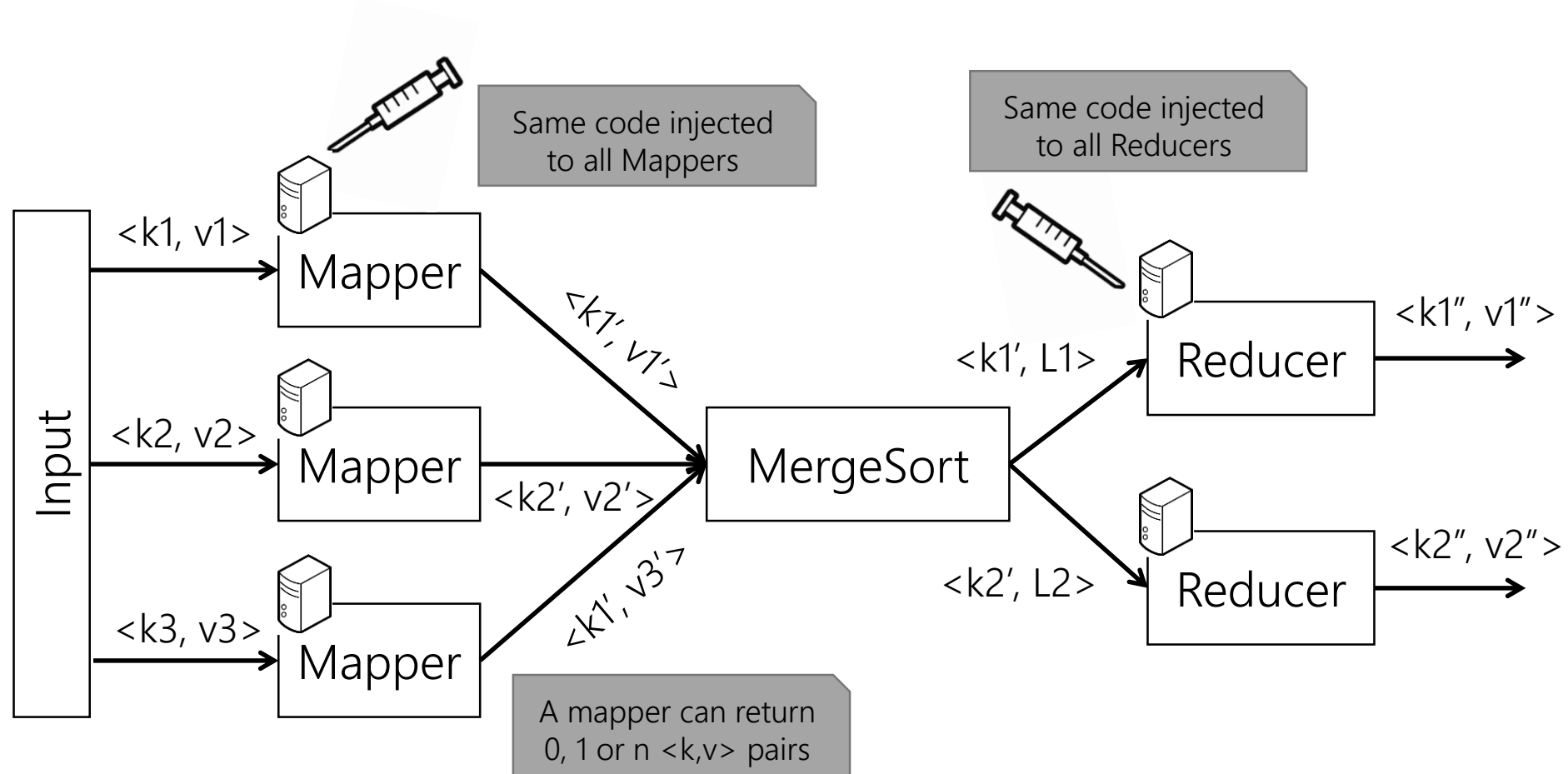
- A piece of history...
 - Appeared together with Google File System (GFS) as a processing framework
 - Its main objective is to enable massive parallelism over distributed data (in GFS)
<https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- Designed to meet the following requirements
 - Exploit distributed systems and provide **full distributed-transparency** for the end-user
 - Send the queries to data (i.e., query-shipping instead of data-shipping for exploiting the data locality principle)
 - Support parallelism and hide its complexity
 - Independent data (typically collected from the web)
 - Without references to other pieces of data
 - No joins
 - Exploit petabytes of data in batch mode
 - No transactions
 - Failure resilience
 - Cope with failures without aborting
- Inspired in functional programming

Chain production



By Mohamed Nabeel

The MapReduce framework



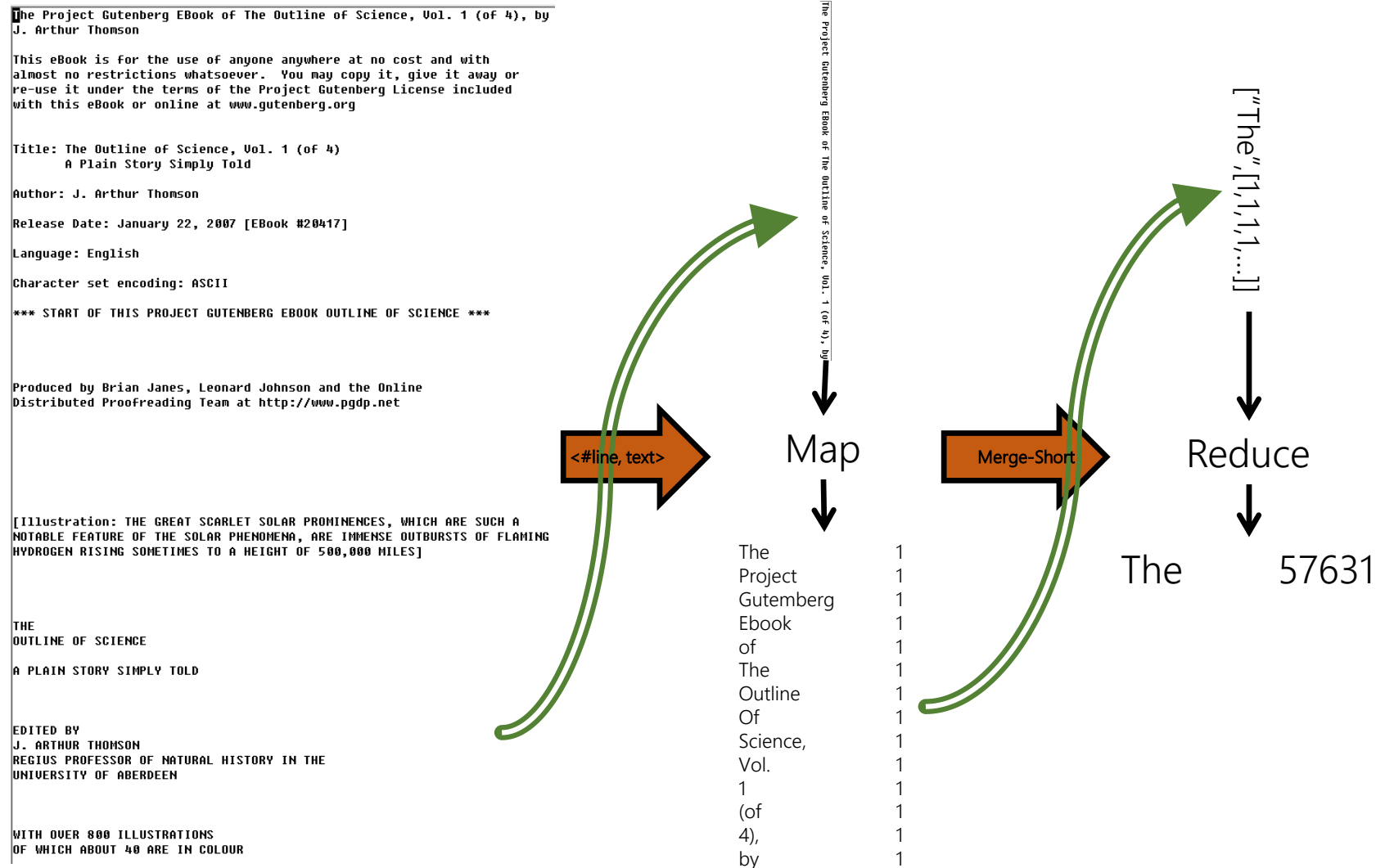
The MapReduce framework in detail

1. Input: read input from a DFS
2. Map: for each input $\langle \text{key}_{\text{in}}, \text{value}_{\text{in}} \rangle$
 - generate zero-to-many $\langle \text{key}_{\text{map}}, \text{value}_{\text{map}} \rangle$
3. Partition: assign sets of $\langle \text{key}_{\text{map}}, \text{value}_{\text{map}} \rangle$ to reducer machines
4. Shuffle: data are shipped to reducer machines using a DFS
5. Sort&Merge: reducers sort their input data by key
6. Reduce: for each key_{map}
 - the set $\text{value}_{\text{map}}$ is processed to produce zero-to-many $\langle \text{key}_{\text{red}}, \text{value}_{\text{red}} \rangle$
7. Output: writes the result of reducers to the DFS

MapReduce examples

Word count

Word count example

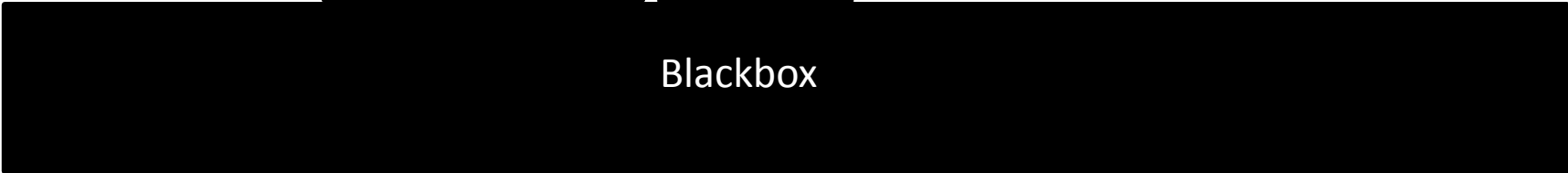


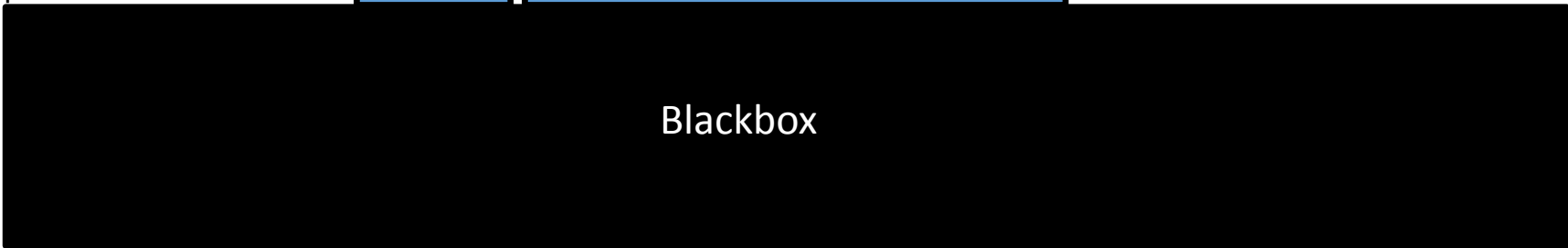
WordCount Code Example

```
public void map(LongWritable key, Text value) {  
    String line = value.toString();  
    StringTokenizer tokenizer = new StringTokenizer(line);  
    while (tokenizer.hasMoreTokens()) {  
        write(new Text(tokenizer.nextToken()), new IntWritable(1));  
    }  
}
```

```
public void reduce(Text key, Iterable<IntWritable> values) {  
    int sum = 0;  
    for (IntWritable val : values) {  
        sum += val.get();  
    }  
    write(key, new IntWritable(sum));  
}
```


WordCount Code Example

```
public void map(Key, Value) {  
     Blackbox  
    write(Key, Value);  
}  
}
```

```
public void reduce(Key, Values) {  
     Blackbox  
    write(Key, Value);  
}
```

MapReduce examples

Common friends

Friends in common example

- In a social network (e.g., Facebook), we aim to compute the friends in common for every pair of users
 - This is a value that does not frequently change, so it can be precomputed
- Friends are stored as

Person \rightarrow [List of friends]

- $A \rightarrow B\ C\ D$
- $B \rightarrow A\ C\ D\ E$
- $C \rightarrow A\ B\ D\ E$
- $D \rightarrow A\ B\ C\ E$
- $E \rightarrow B\ C\ D$

Friends in common – Map task

- For every friend in the list, the mapper will generate a $\langle k, v \rangle$
 - Key: the input key concatenated with one friend in alphabetical order
 - Value: the whole list of friends
- Keys will be sorted, a pair of friends go to the same reducer

A \rightarrow B C D

(A B) \rightarrow B C D

(A C) \rightarrow B C D

(A D) \rightarrow B C D

B \rightarrow A C D E

(A B) \rightarrow A C D E

(B C) \rightarrow A C D E

(B D) \rightarrow A C D E

(B E) \rightarrow A C D E

C \rightarrow A B D E

(A C) \rightarrow A B D E

(B C) \rightarrow A B D E

(C D) \rightarrow A B D E

(C E) \rightarrow A B D E

...

Friends in common – Reduce task

- Reducers receive two lists of friends per pair of people

(A B) → (B C D) (A C D E)

(A C) → (B C D) (A B D E)

(A D) → (B C D) (A B C E)

...

- The reduce function intersects the lists of values and generates the same key

(A B) → (C D)

(A C) → (B D)

(A D) → (B C)

...

- ... when D visits A's profile we can lookup (A D) to see their common friends

Relational algebra in MapReduce

Relational operations: Projection

$$\pi_{a_{i_1}, \dots, a_{i_n}}(T) \Rightarrow \begin{cases} \text{map}(\text{key } k, \text{value } v) \mapsto [(\text{prj}_{a_{i_1}, \dots, a_{i_n}}(k \oplus v), 1)] \\ \text{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Relational operations: Cross Product

$$T \times S \Rightarrow \left\{ \begin{array}{l} \text{map}(\text{key } k, \text{value } v) \mapsto \\ \left\{ \begin{array}{ll} [(h_T(k) \bmod D, k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(0, k \oplus v), \dots, (D-1, k \oplus v)] & \text{if input}(k \oplus v) = S. \end{array} \right. \\ \text{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \left[\begin{array}{l} \text{crossproduct}(T_{ik}, S) \mid \\ T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\ S = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\} \end{array} \right] \end{array} \right.$$

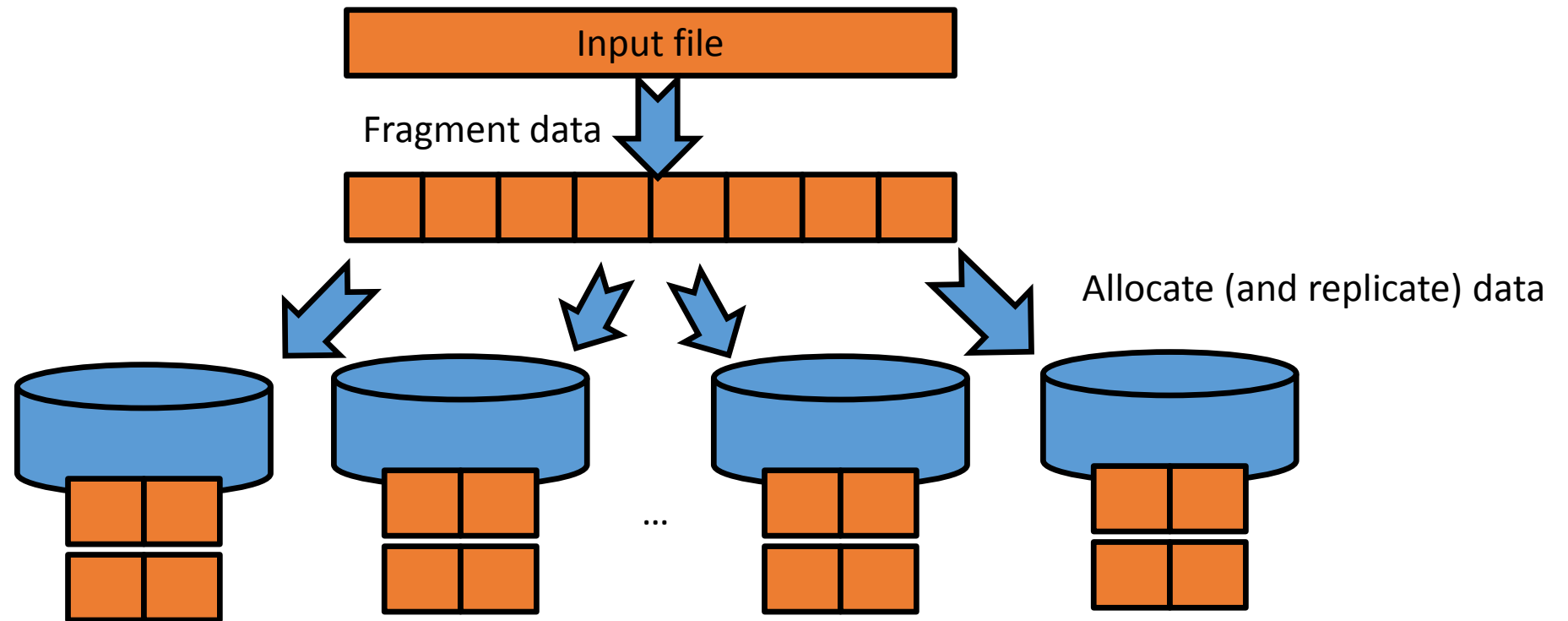
Architecture

MapReduce Functionality

Pre-requisite: Data Loaded into a Distributed System

The **input data must be partitioned into fragments** (i.e., be already distributed) and allocated (**and therefore replicated**) in a distributed system (e.g., HDFS or HBase)

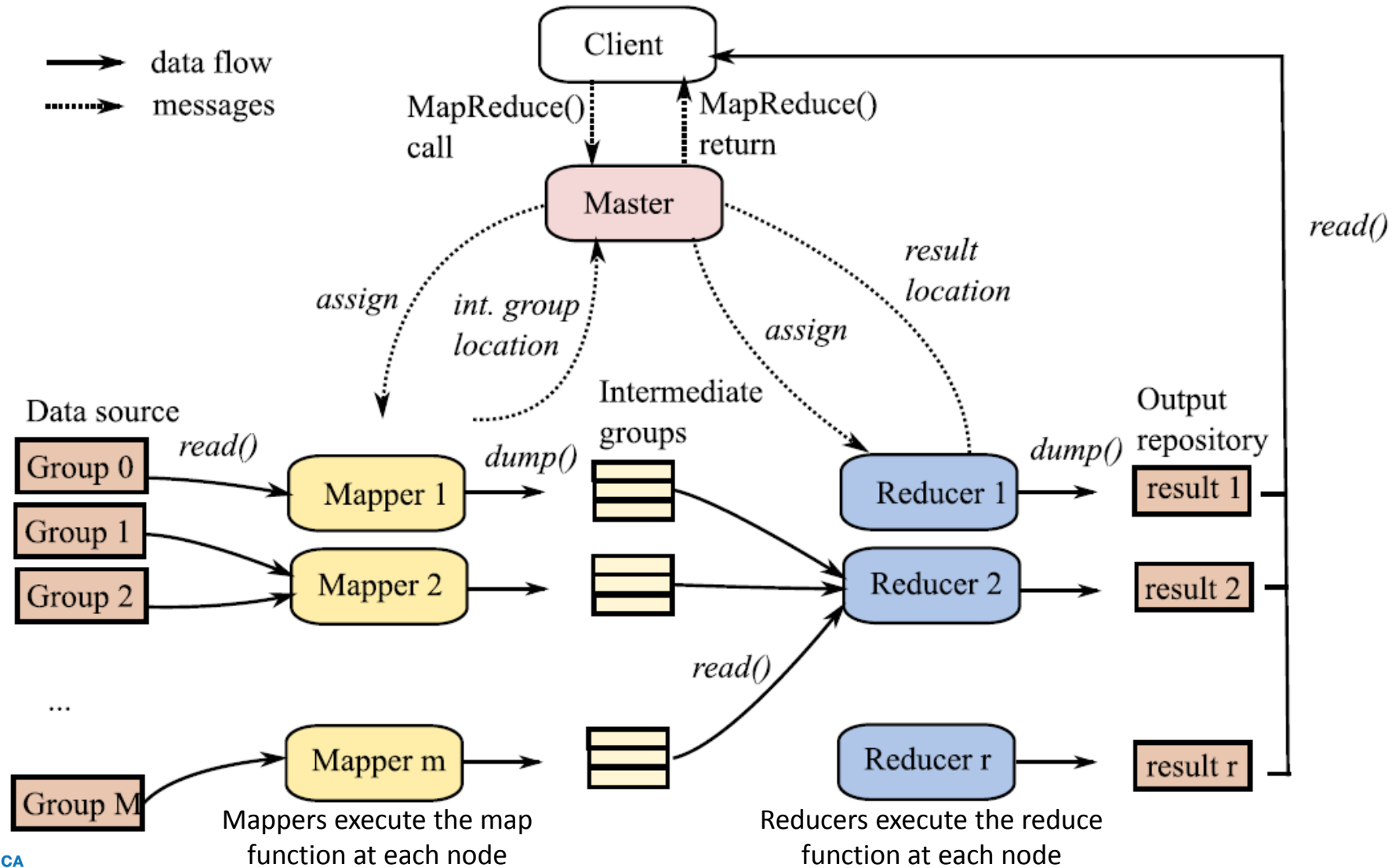
This step is **NOT** performed by MapReduce and to use MapReduce data must be loaded into a distributed storage system (e.g., HDFS or HBase) from where MapReduce will read



MapReduce Main Components

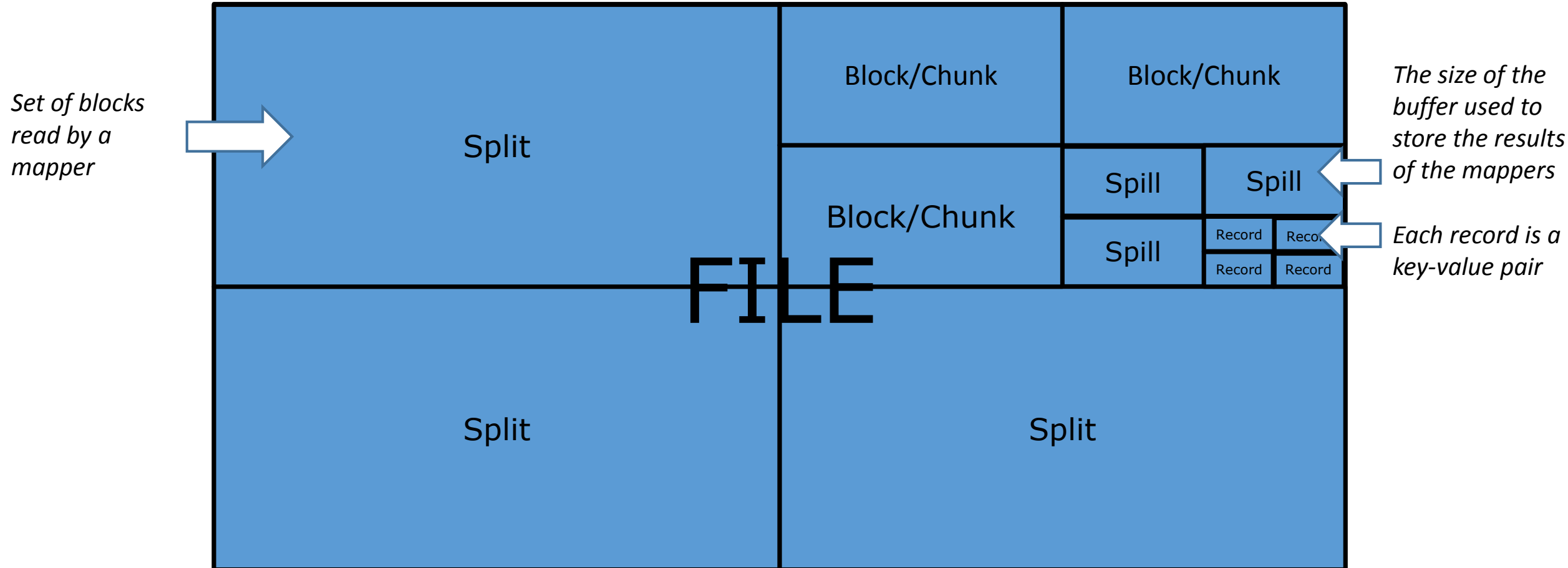
- **Master**: receives the MapReduce (MR) job and schedules it
 - Send the map function to Mappers
 - Send the reduce function to Reducers
 - Monitors the overall execution and detects if a mapper or a reducer fails
 - Reassigns the tasks of a mapper or a reducer to another node upon failure
- **Mapper**: one per node assigned to execute the map function
 - The master assigns a set of blocks to each mapper
 - Executes the map function for each record in the set of blocks assigned
 - Stores the result of all the map executions in intermediate results
- **Reducer**: one per node assigned to execute the reduce function
 - The master assigns a set of intermediate results to each reducer
 - Executes the reduce function to each record in the intermediate results assigned
 - Stores the result of the reduce executions in the output storage specified in the MR job

MapReduce Components: Tasks and Data Flows



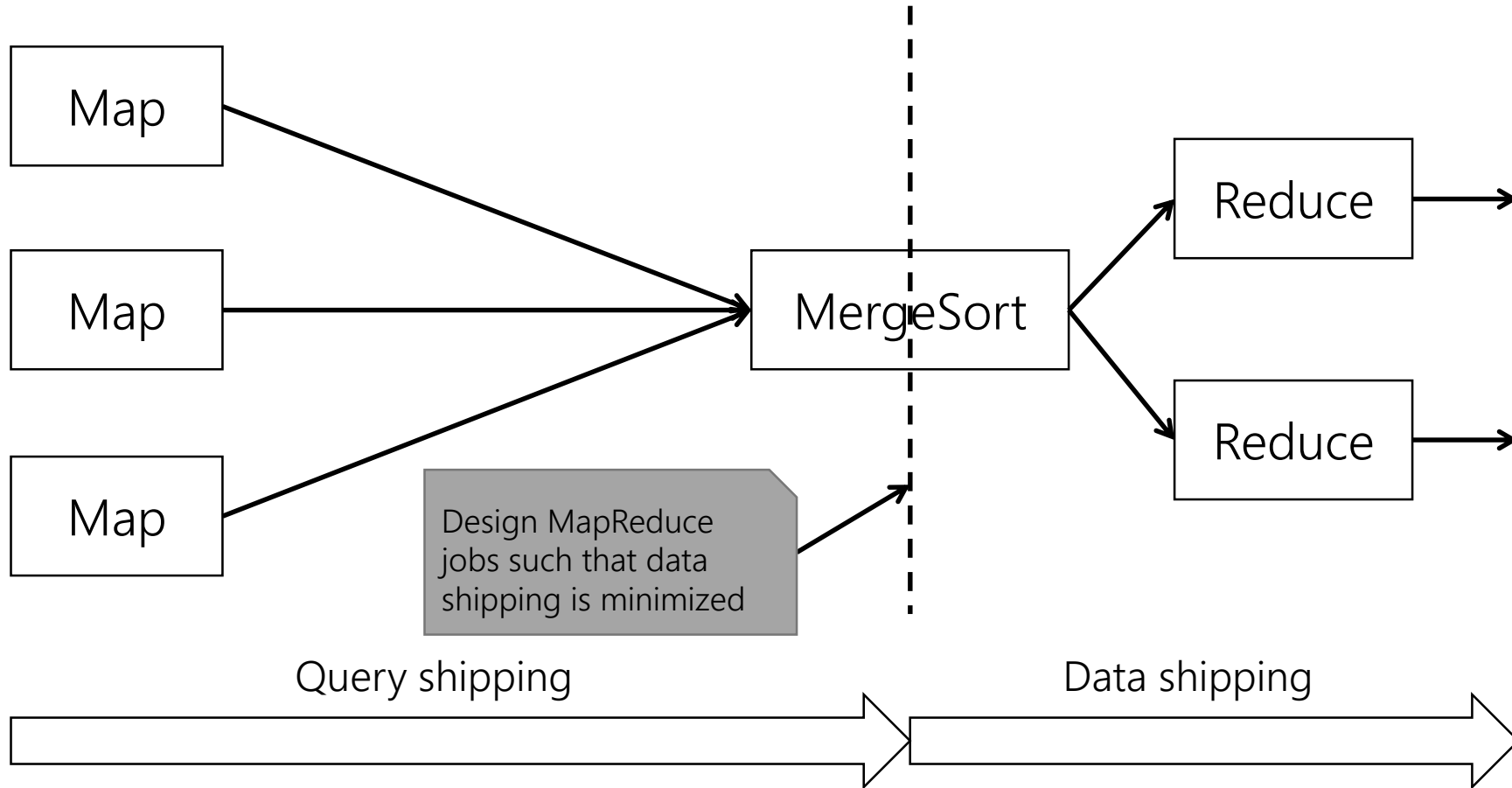
MapReduce Objects

Remember the block (sometimes also called chunk) is the atomic unit read from the distributed storage (see previous slide)



Internal algorithm

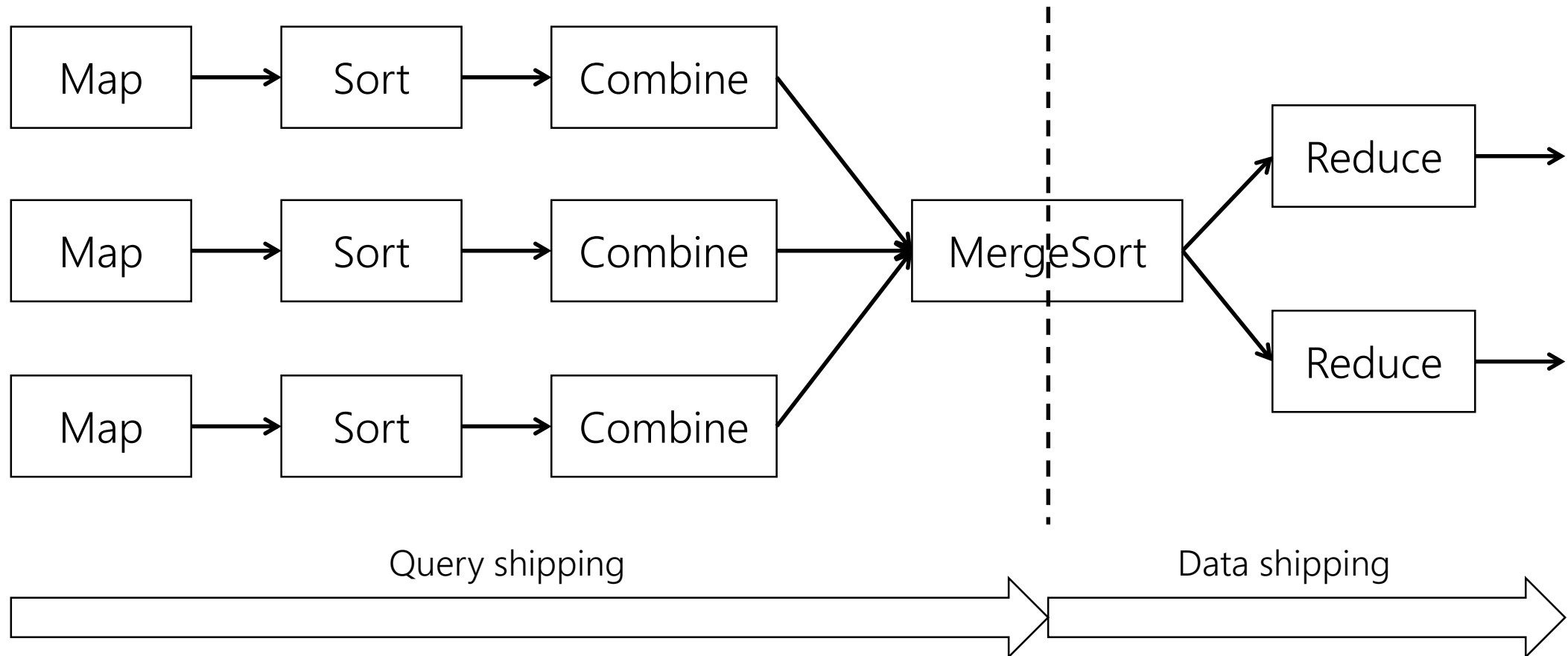
Query shipping vs. data shipping (I)



Combiner

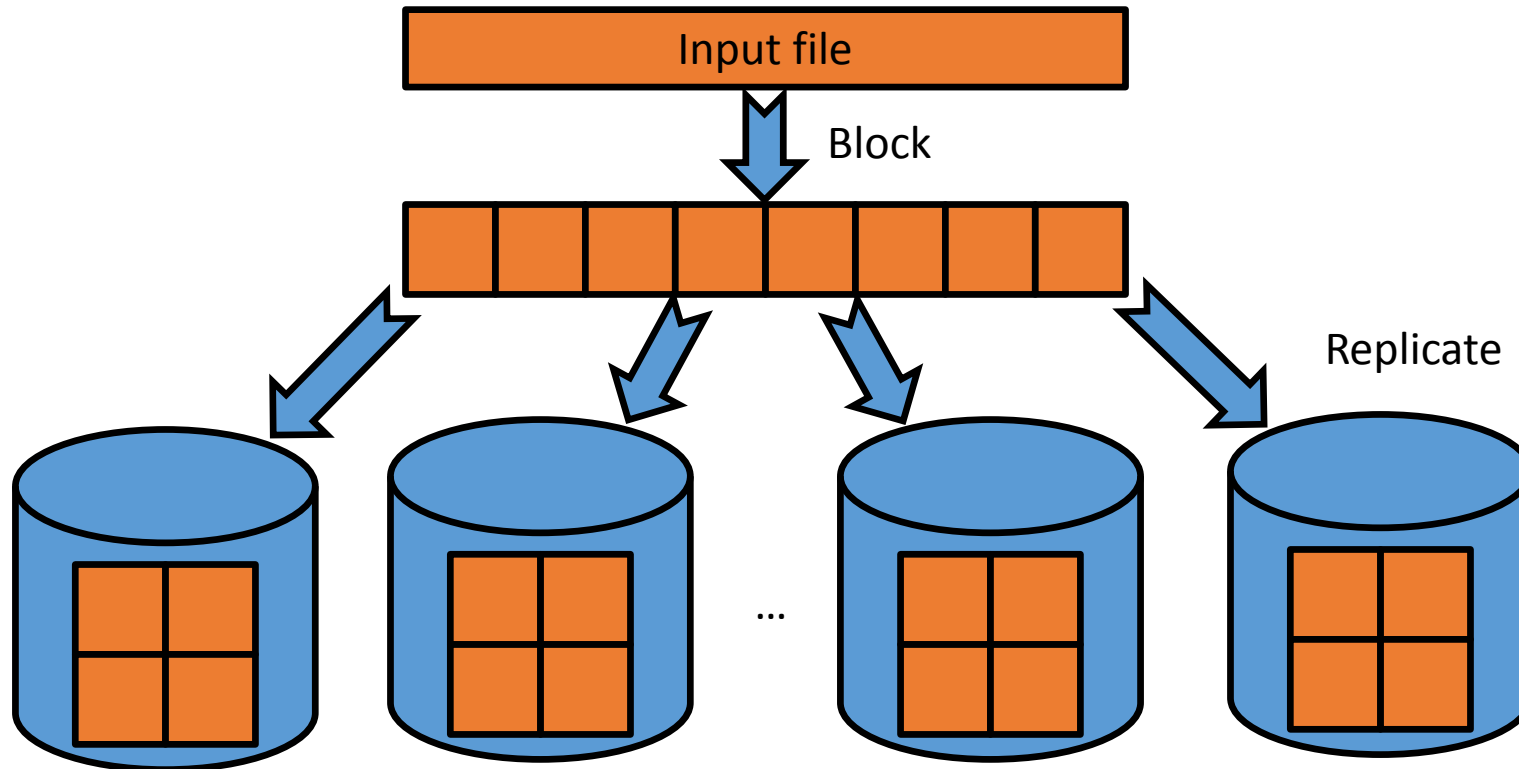
- Coincides with reducer function when it is:
 - Commutative
 - Associative
- Exploits data locality at the Mapper level
 - Data transfer diminished since Mapper outputs are reduced
 - Saving both network and storing intermediate results costs
- Only makes sense if $|I|/|O| \gg \#CPU$
 - Skewed distribution of input data improves early reduction of data

Query shipping vs. data shipping (II)



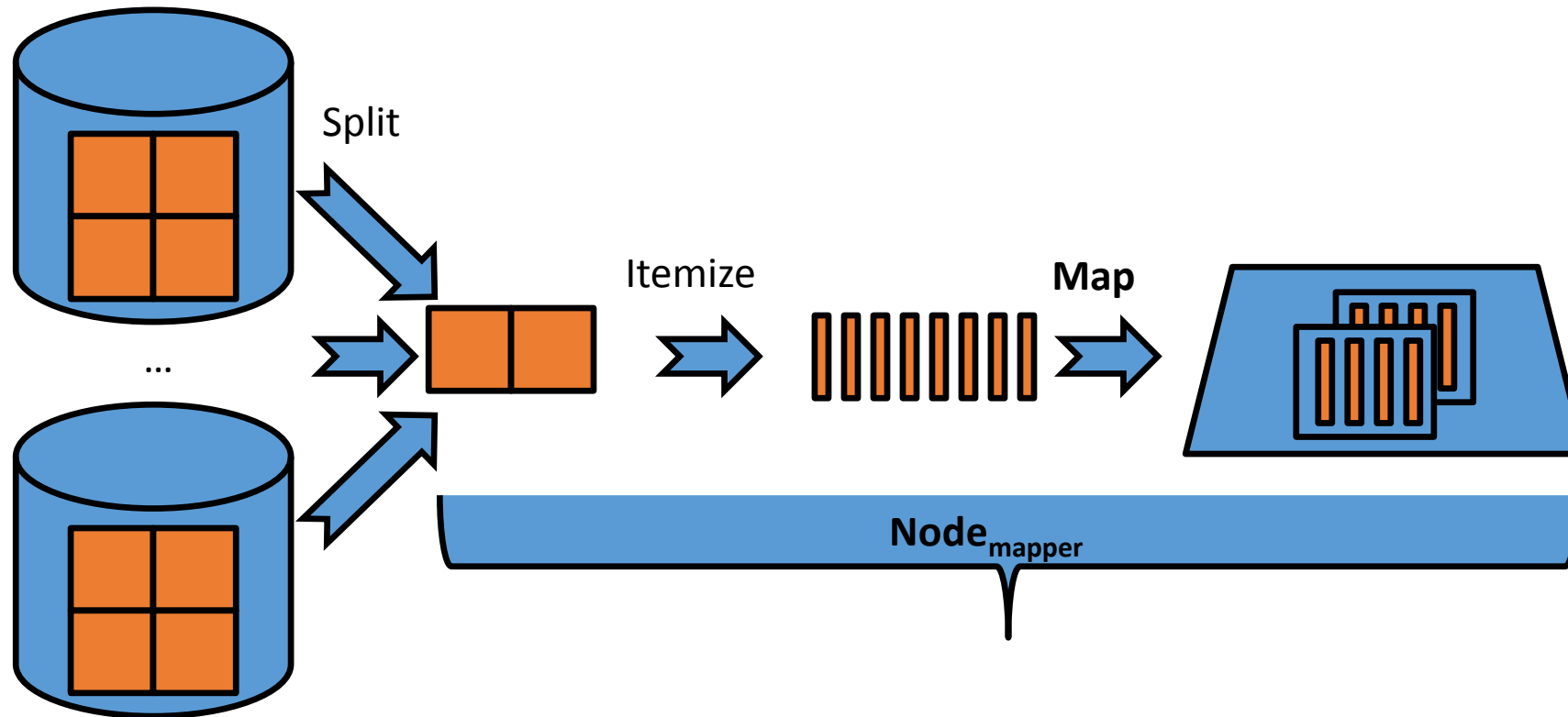
Algorithm: Data Load

1. Upload the data to the Cloud
 - Partition them into blocks
 - Using HDFS or any other storage (e.g., HBase, MongoDB, Cassandra, CouchDB, etc.)
2. Replicate them in different nodes



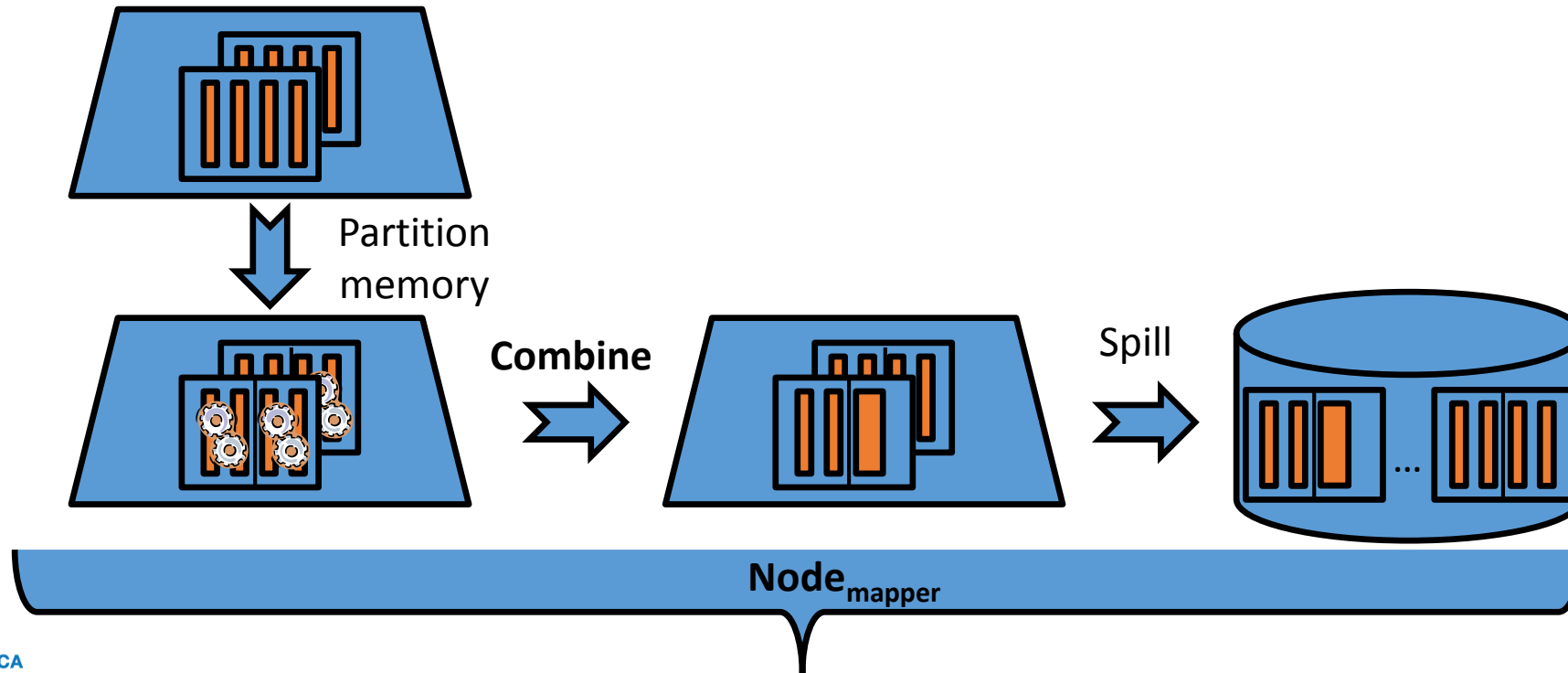
Algorithm: Map Phase (I)

3. Each mapper (i.e., JVM) reads a subset of blocks/chunks (i.e., split)
4. Divide each split into records
5. Execute the map function for each record and keep its results in memory
 - JVM heap used as a circular buffer



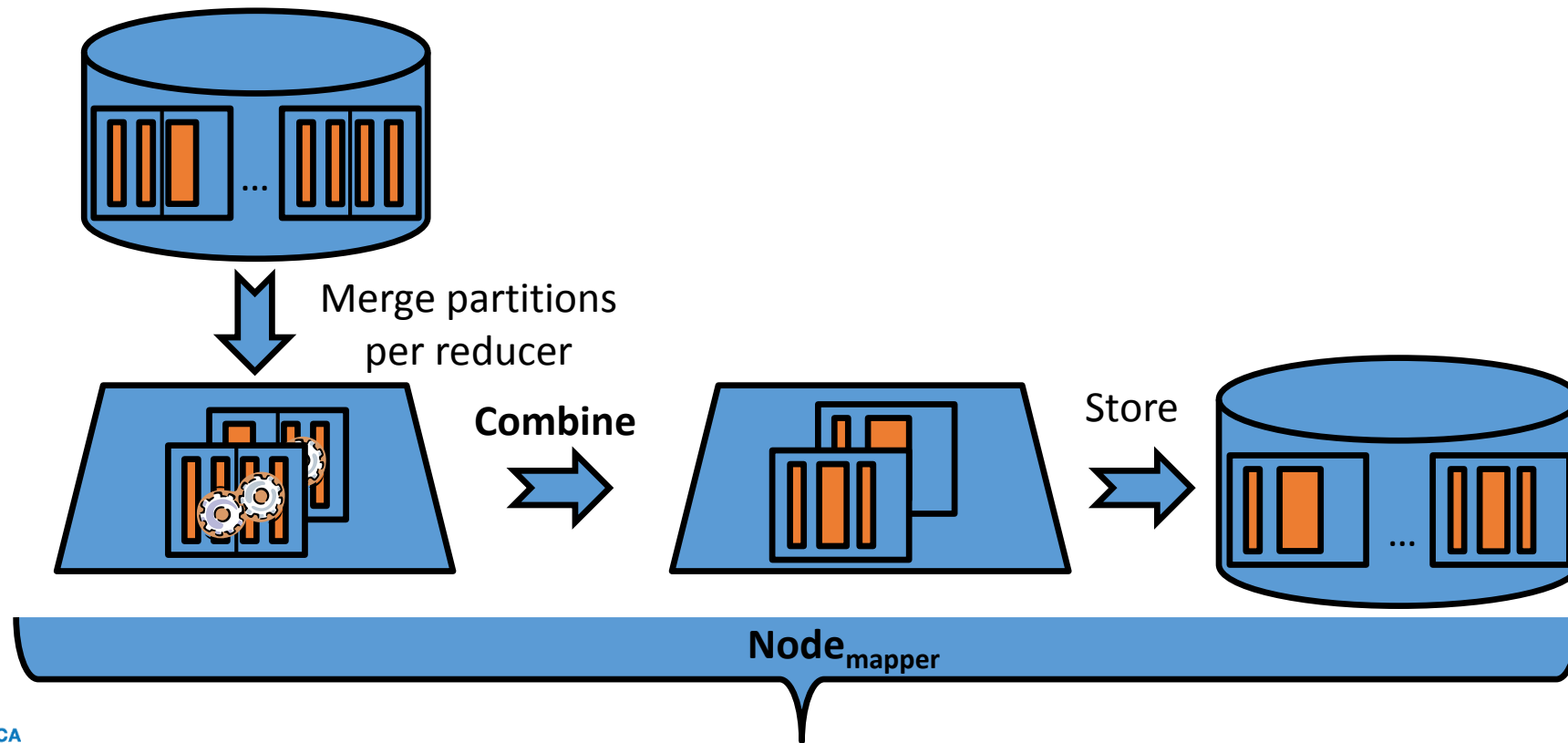
Algorithm: Map Phase (II)

6. Each time memory becomes full
 1. The memory is then partitioned per reducers
 - Using a hash function f over the new key
 2. Each memory partition is sorted independently
 - If a combine is defined, it is executed locally during sorting
 3. Spill partitions into disk (massive writing)



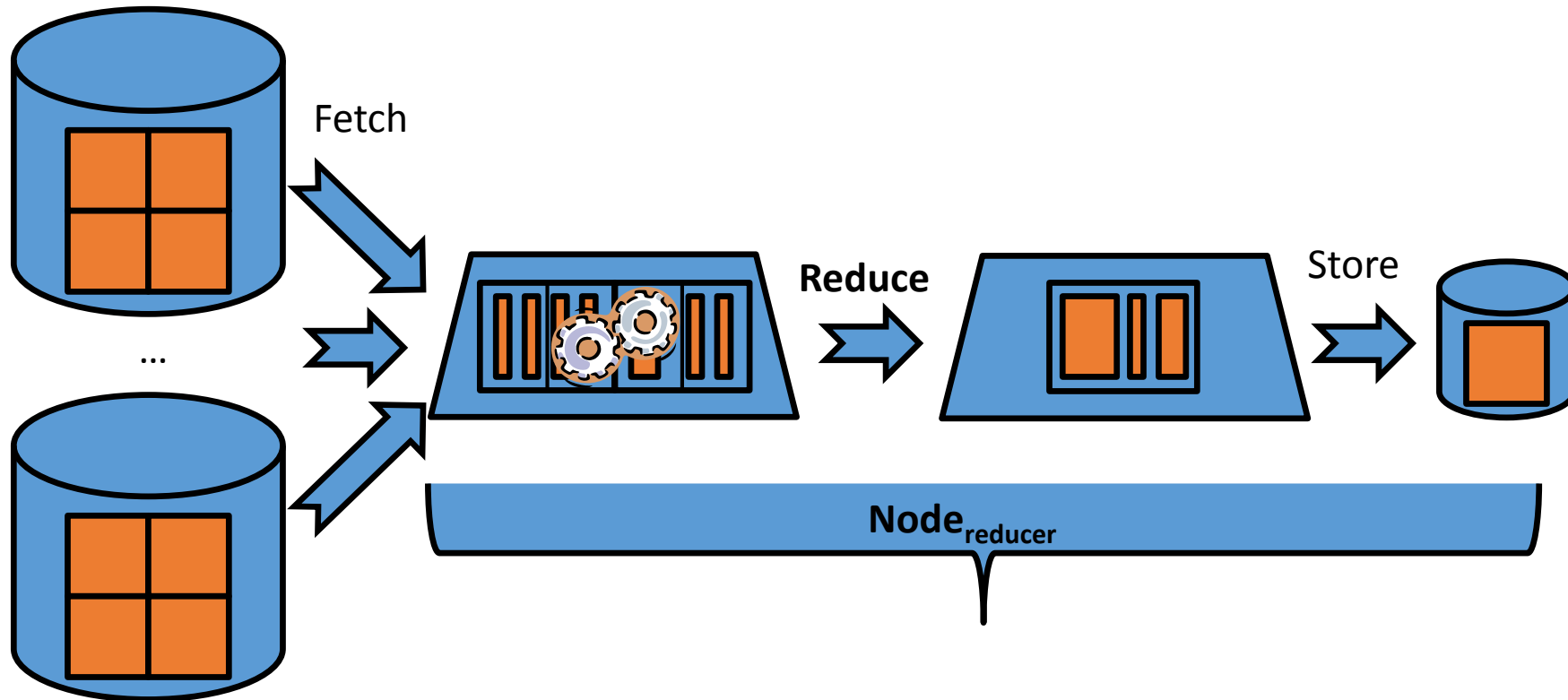
Algorithm: Map phase (III)

7. Partitions of different spills are merged
 - Each merge is sorted independently
 - Combine is applied again
8. Store the result into disk



Algorithm: Shuffle and Reduce

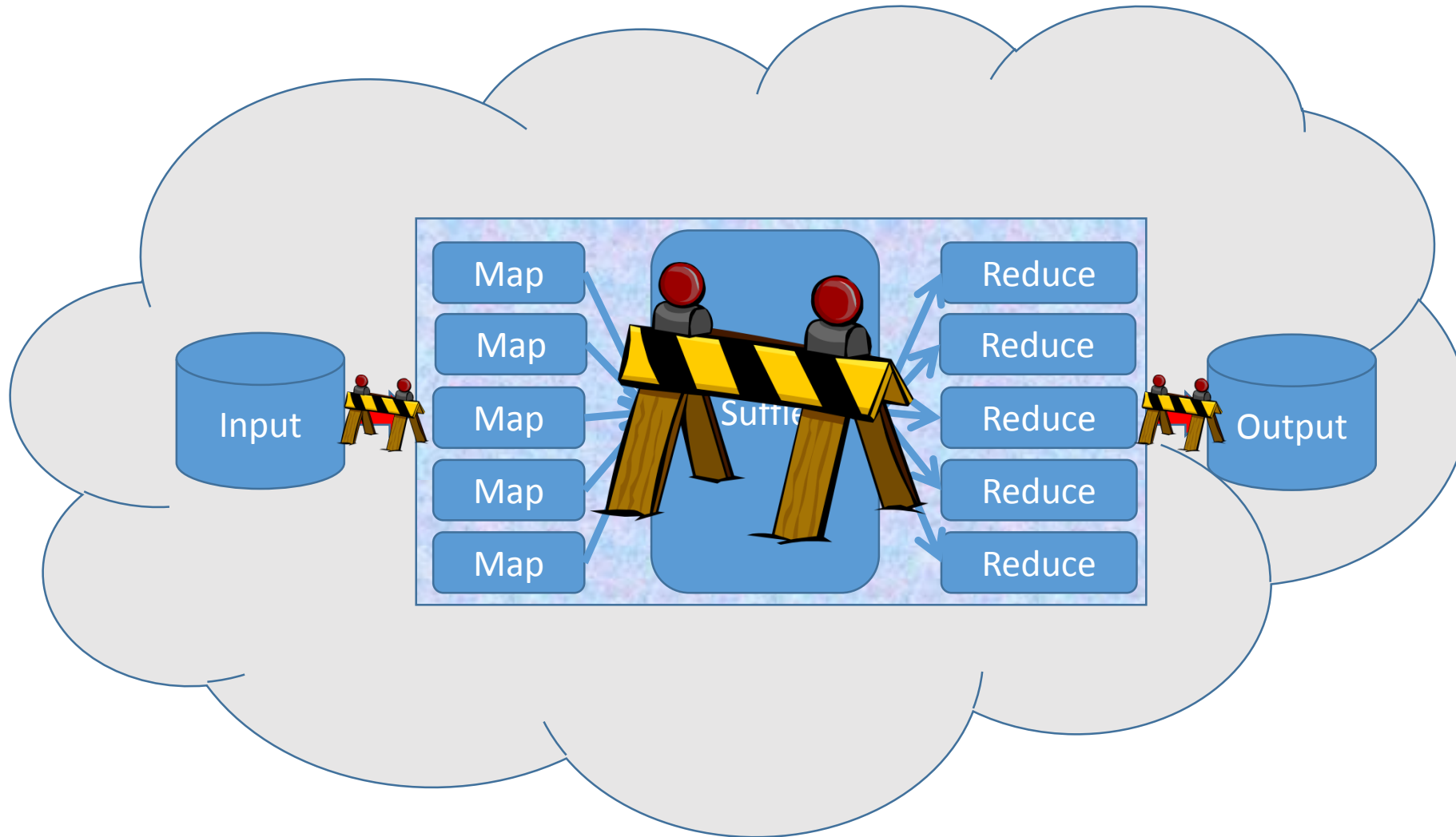
9. Reducers fetch data through the network (massive data transfer)
10. Key-Value pairs are sorted and merged
11. Reduce function is executed per key
12. Store the result into disk



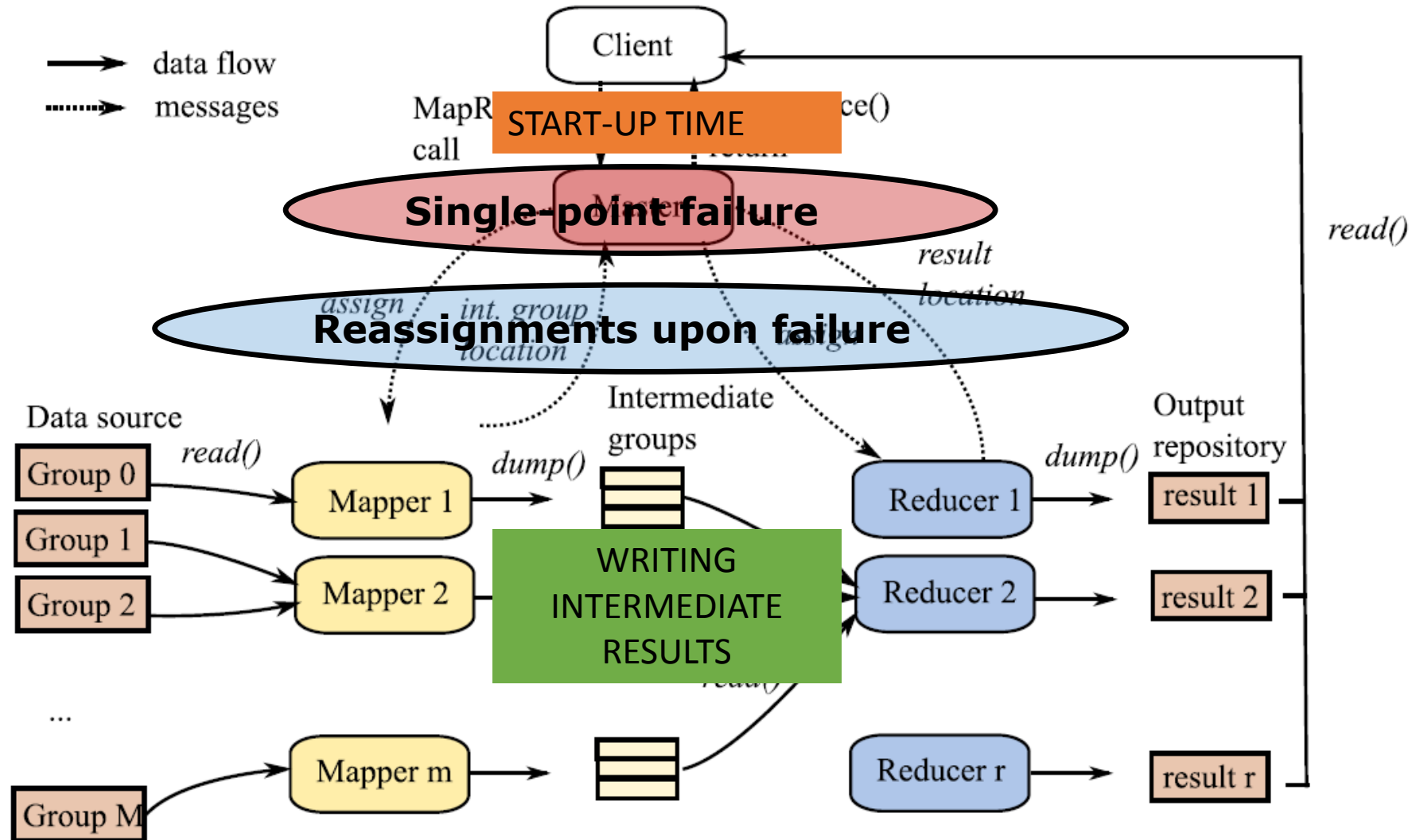
Drawbacks

Bottlenecks of the algorithm

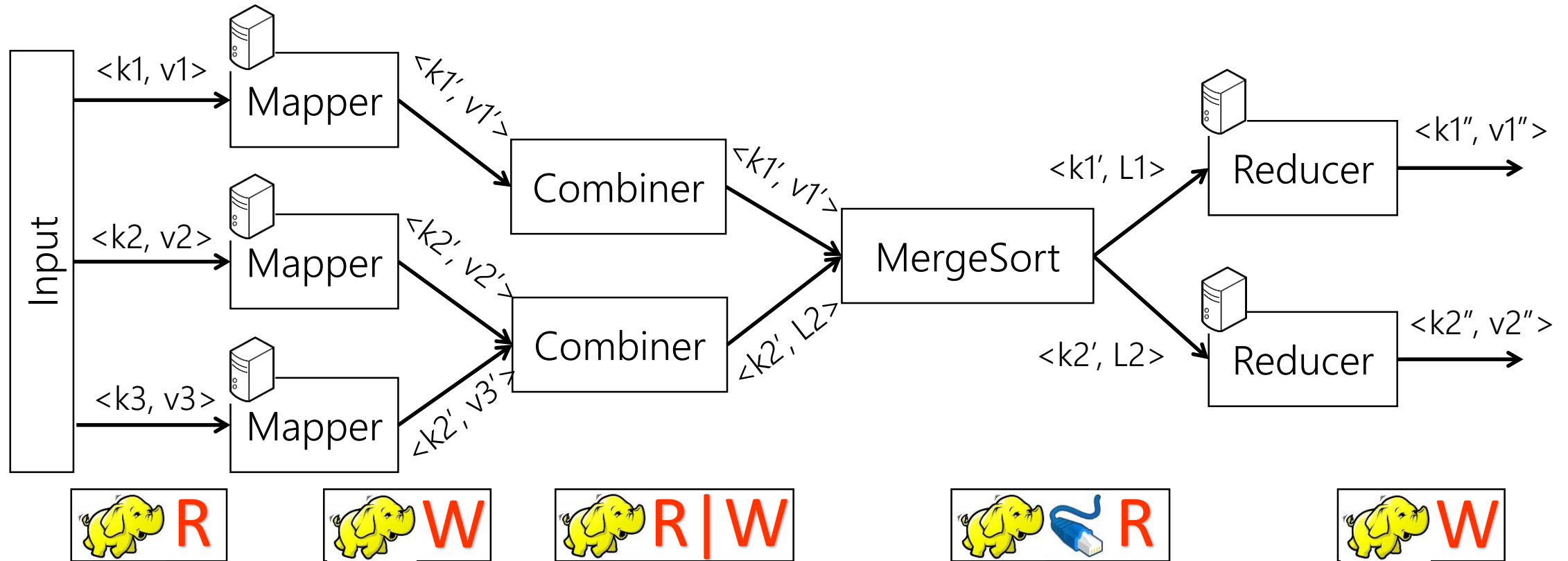
Synchronization Barriers



MapReduce: Tasks and Data Flows



MapReduce intra-job coordination

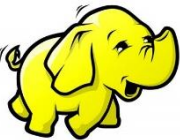
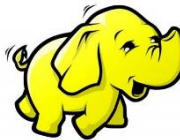
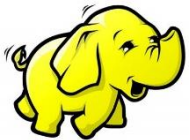
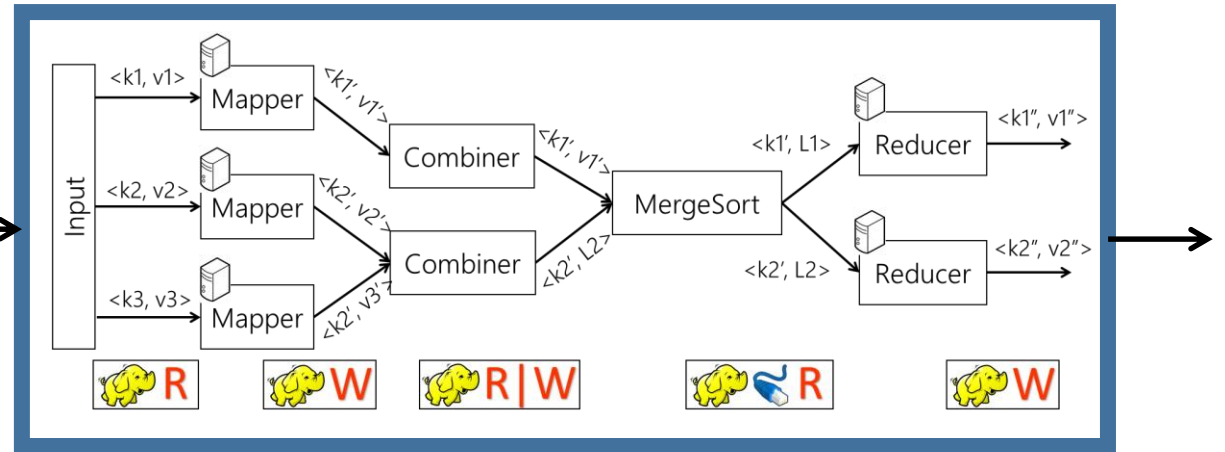
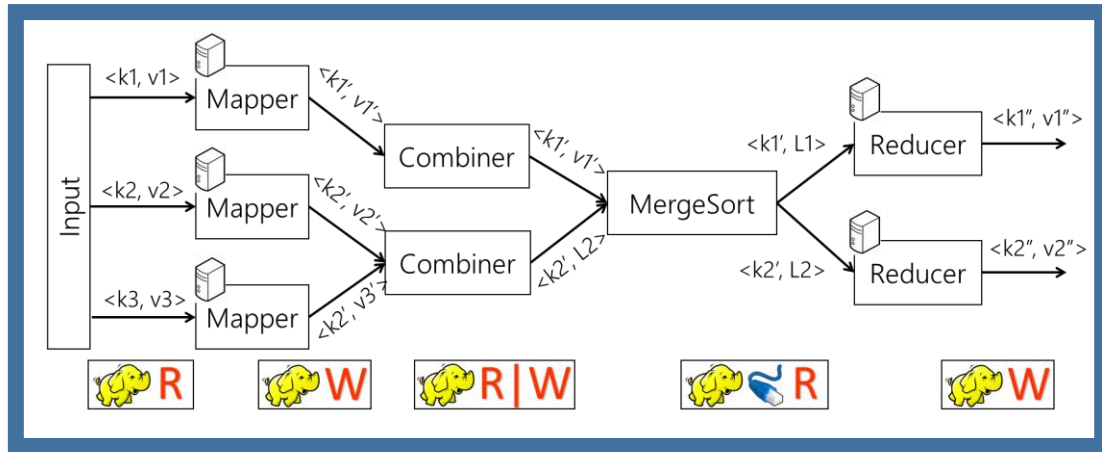


MapReduce inter-job coordination

Count

Rank

...



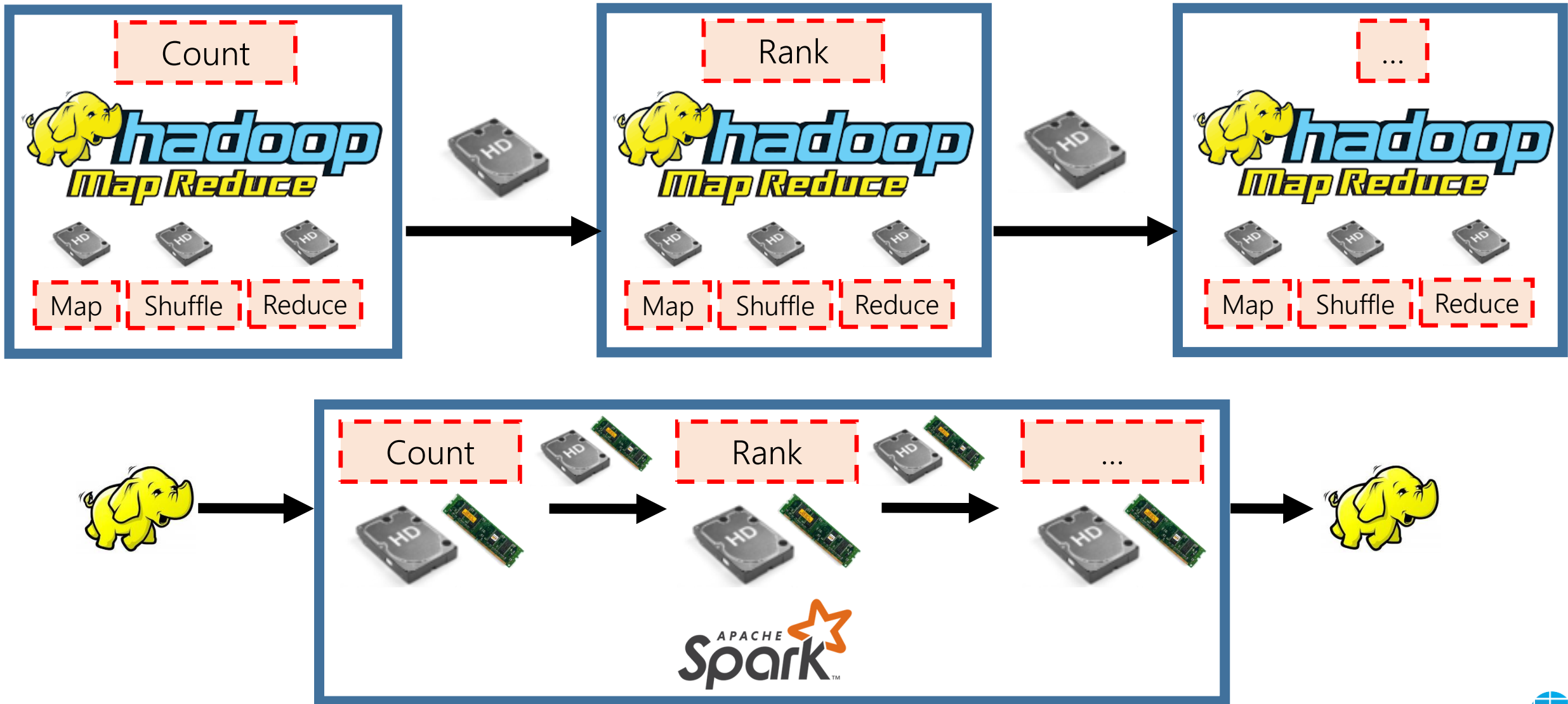
Spark

*"Unified **abstraction** for cluster computing, consisting in a **read-only**, partitioned collection of records. Can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs."*

It is an evolution of MapReduce

- Based on a richer data structure (dataframe)
- Instead of two operations: map and reduce, it provides more than 20 operators
 - Projection, Selection, Join, Group By... welcome back!
 - Reduces the black box problem in MapReduce
- Models the Spark job execution as a graph defined by the user programmatically
 - MapReduce always follows the same execution scheme: Map, Shuffle, Reduce
 - Spark is more flexible and the user can adapt and configure it
- Avoids writing intermediate results in disk (intensive in-memory processing)

Main memory coordination in Spark



Exercise

Executing a MapReduce job step by step

Activity: MapReduce

- *Objective: Understand the algorithm underneath MapReduce*
- *Tasks:*
 1. *(40') Reproduce step by step the MapReduce execution*
 - Consider the following data set:
 - Block0: "a b b a c | c d c a e"
 - Block1: "a b d d a | b b c c f"
 - Simulate the execution of the MapReduce code given the following configuration:
 - The map and reduce functions are those of the wordcount
 - The combine function shares the implementation of the reduce
 - There is one block per split
 - The "|" divides the records inside each block
 - We have two records per block
 - We can keep four pairs [key,value] per spill
 - We have two mappers and two reducers
 - Machine0, contains block0, runs mapper0 and reducer0
 - Machine1, contains block1, runs mapper1 and reducer1
 - The hash function used to shuffle data to the reducers uses the correspondence:
 - {b,d,f}->0
 - {a,c,e}->1

Summary

- The Algorithm
 - Map, Shuffle, Reduce
 - Combine
- Drawbacks
- Spark

Bibliography

S. Abiteboul et al. Web Data Management, 2012

D. Jiang et al. *The performance of MapReduce: An In-depth Study*. VLDB'10

Jeffrey Dean et al. *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04

Apache Spark: <https://spark.apache.org/>