MapReduce

A Processing Framework for Massive Parallelism

Motivation

Example: Card Counting

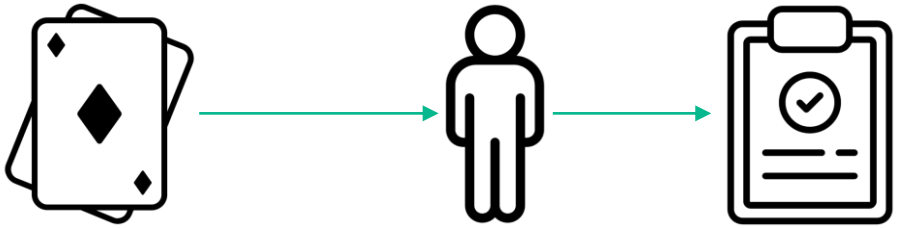
Input:

A bunch of cards

Result:

A♠×13, 2♠×15, 3♠×16, …, 3♥×12; 2♥×21; A♥×15

How would you do it?

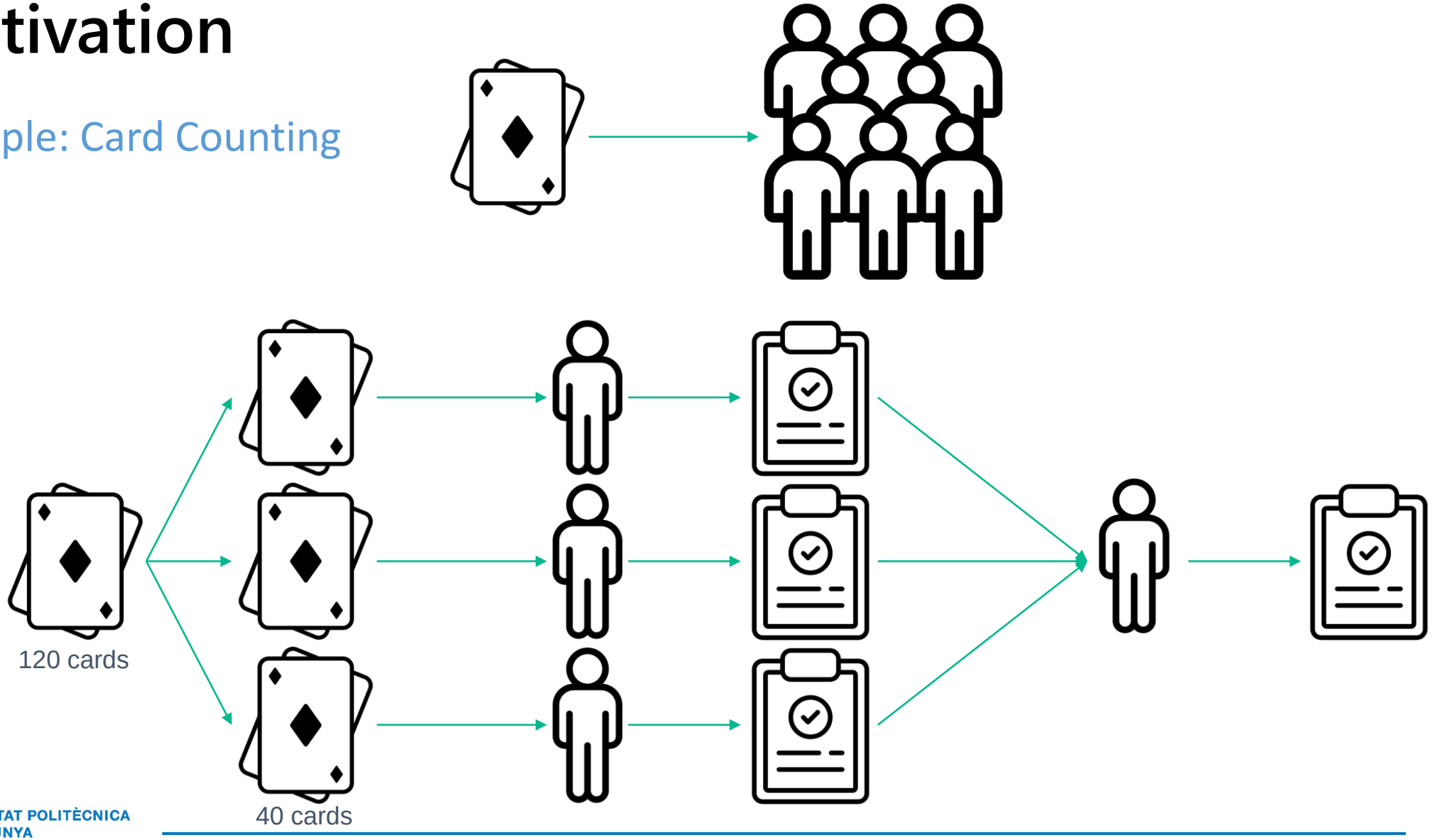


(sequential execution)

Photo by Angshu Purkait on Unsplash

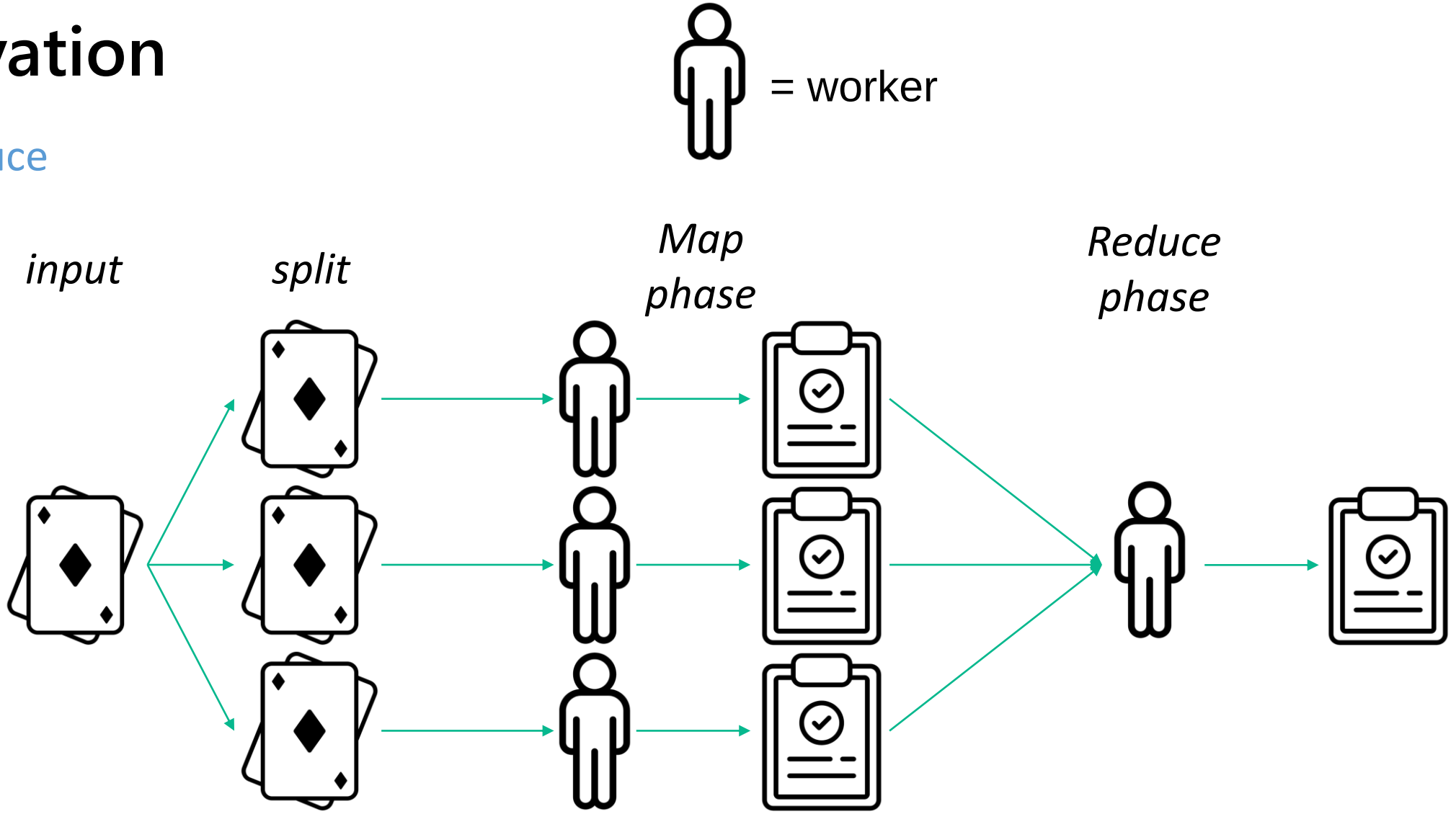
Motivation

Example: Card Counting



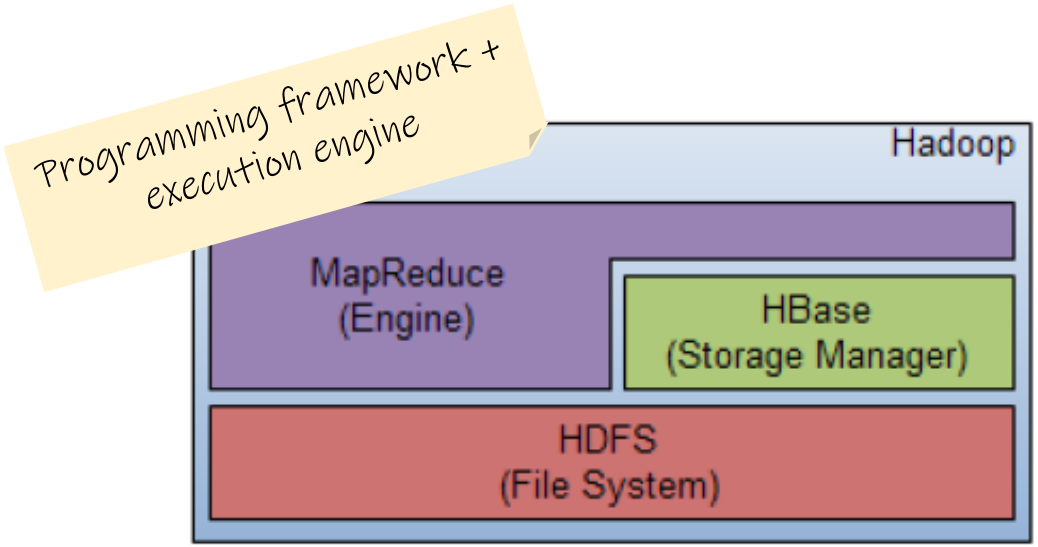
Motivation

MapReduce

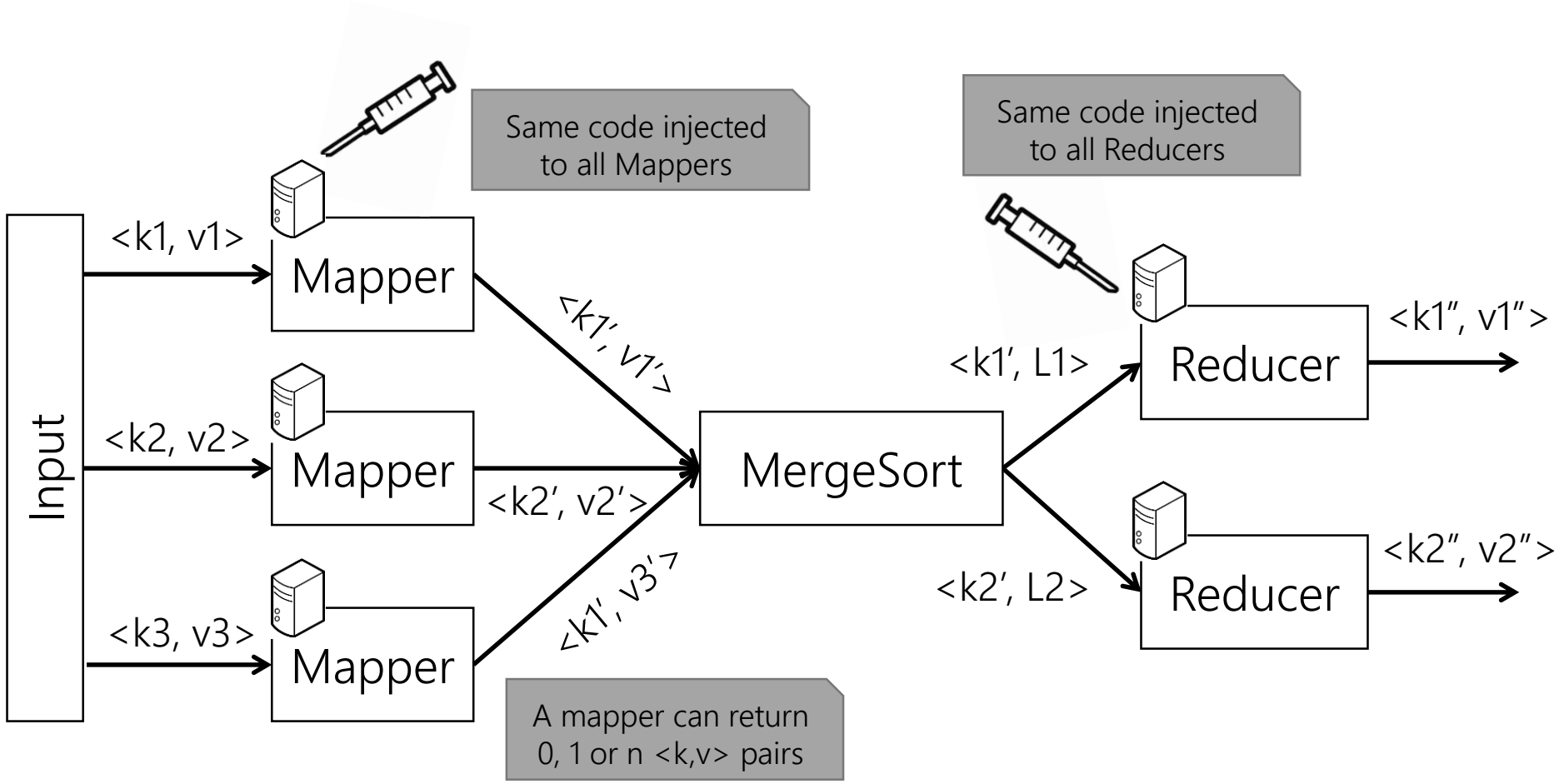


MapReduce

- 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
- 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
  - Functions, immutability



The MapReduce framework



The MapReduce framework in detail

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

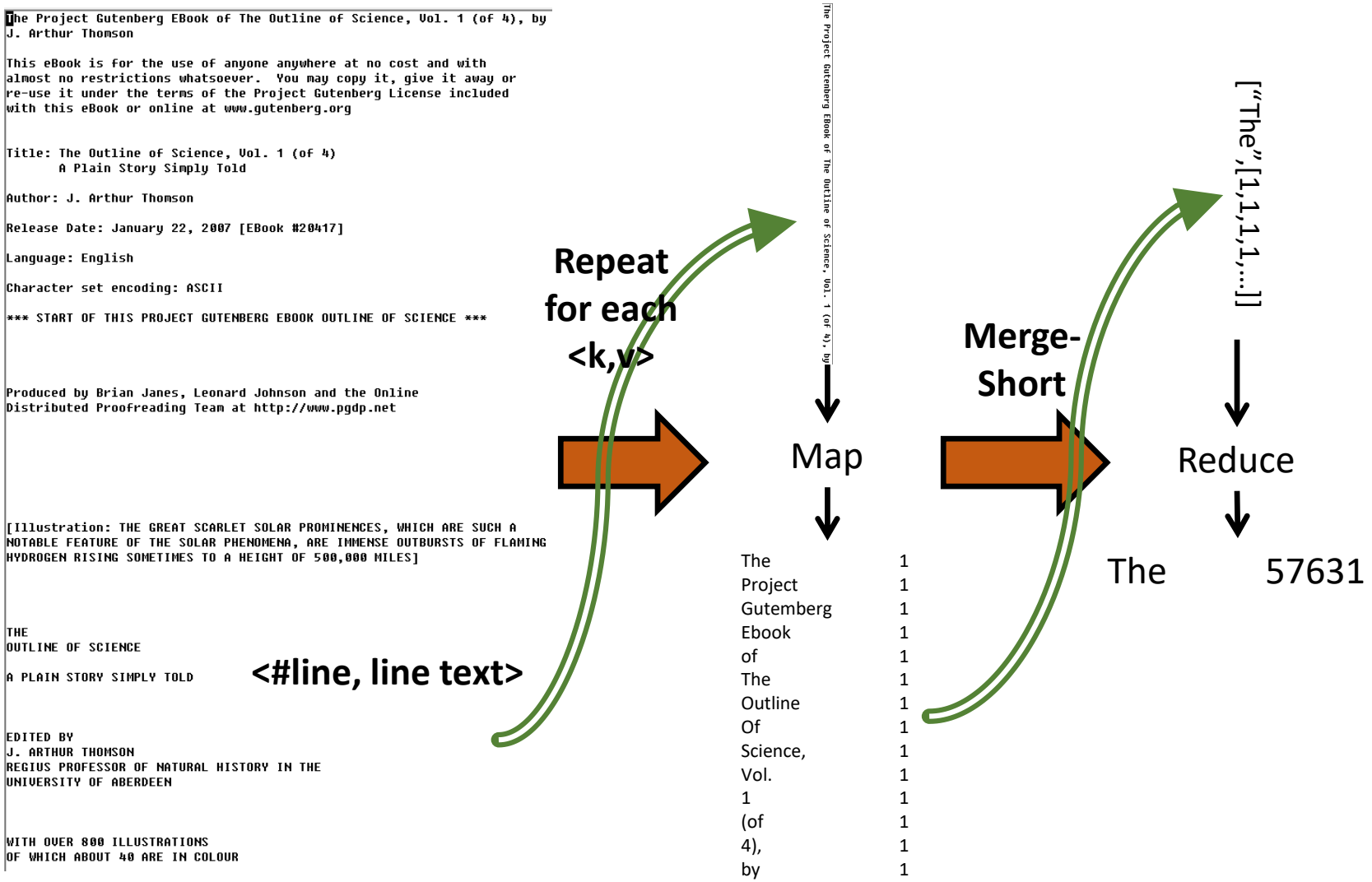
The MapReduce framework in detail

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

MapReduce example

Word count

Word count example



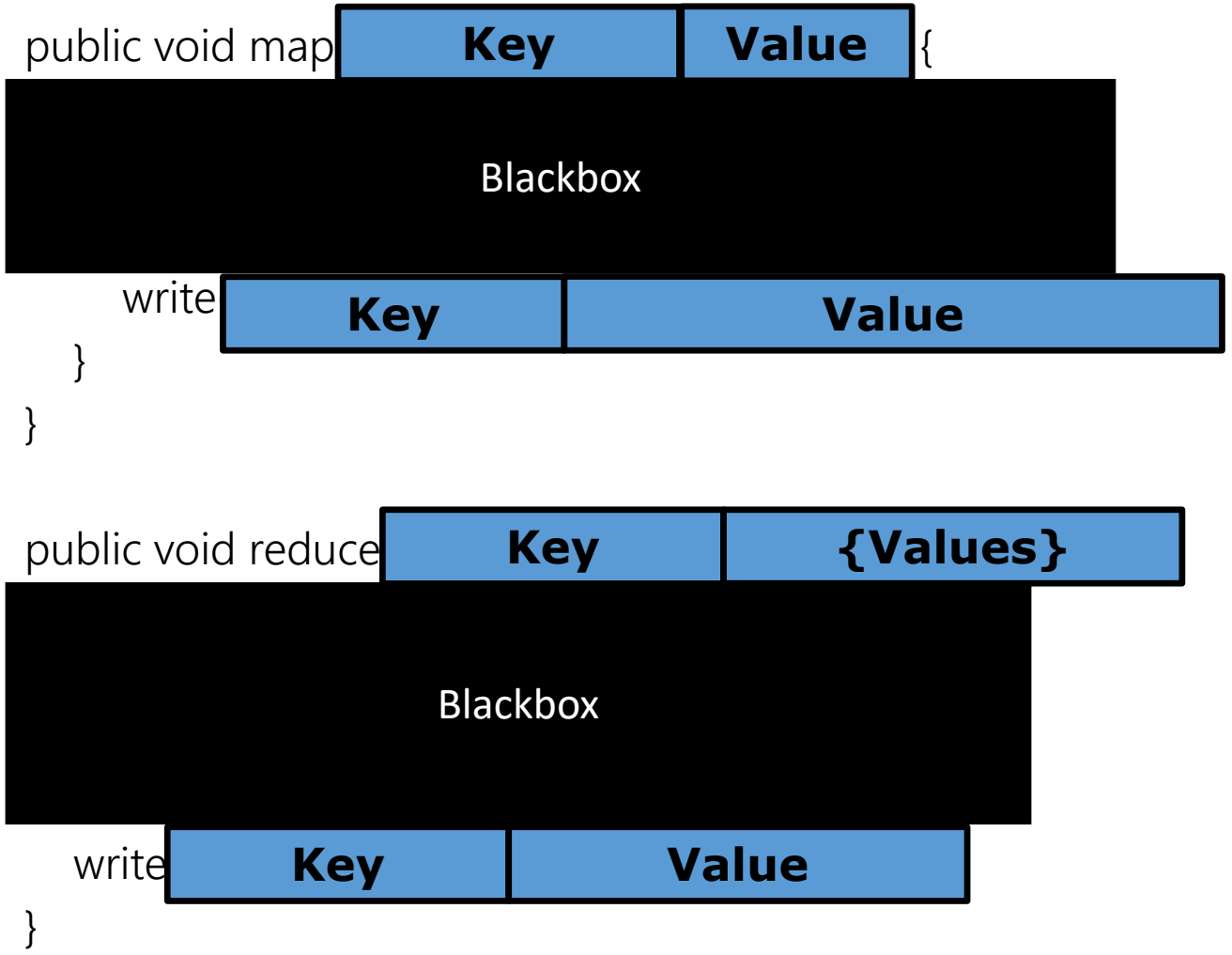
WordCount Code Example

```
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));
}
```

IntWritable, LongWritable, … are variants of data types that are optimized for serialization in the Hadoop environment

WordCount Code Example



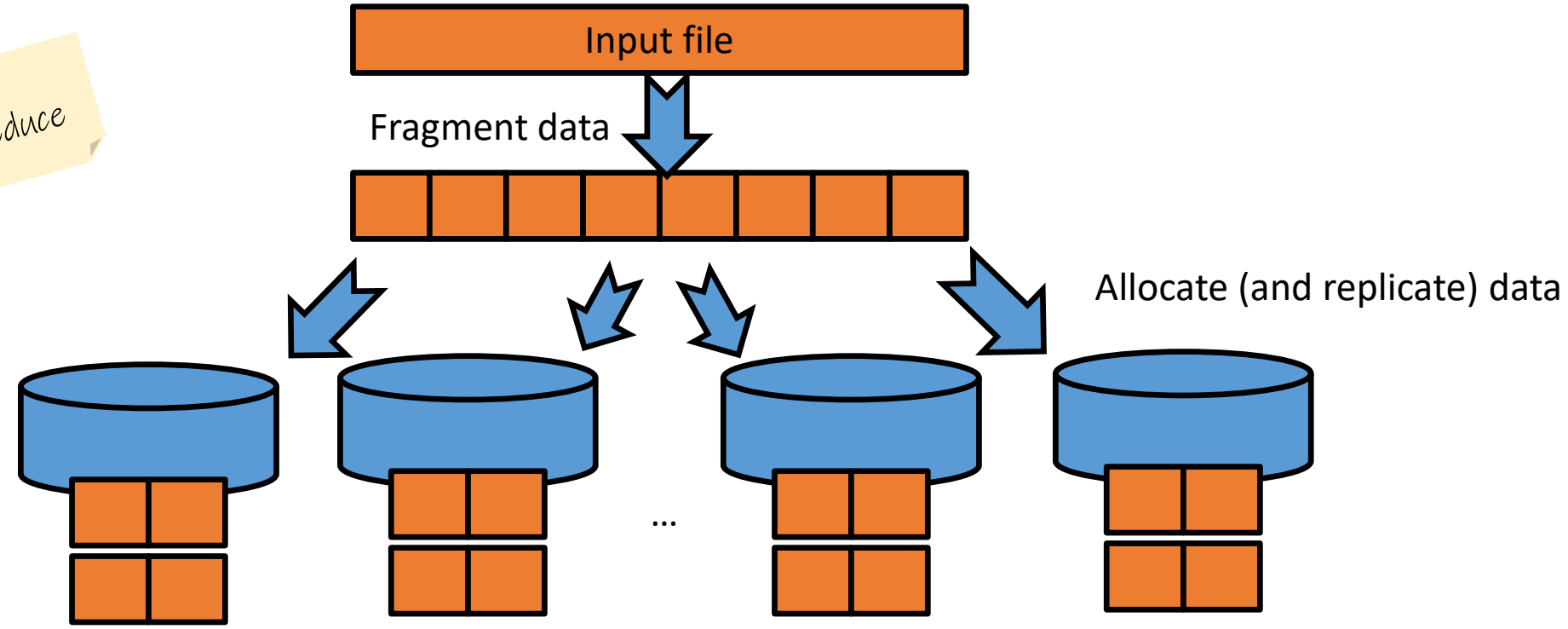
Architecture

MapReduce implementation

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, HBase, Cassandra, MongoDB…)

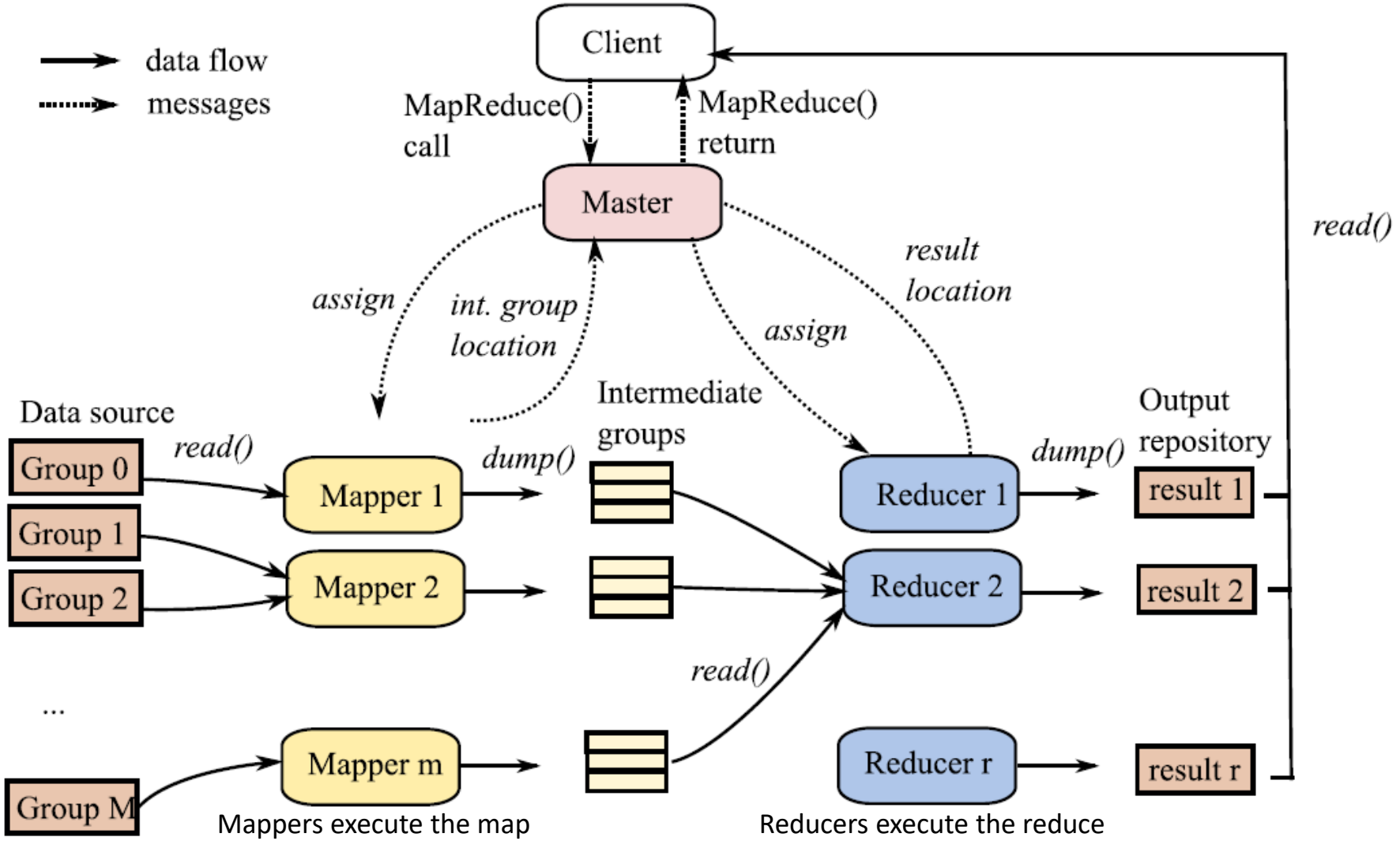
This step is NOT performed by MapReduce



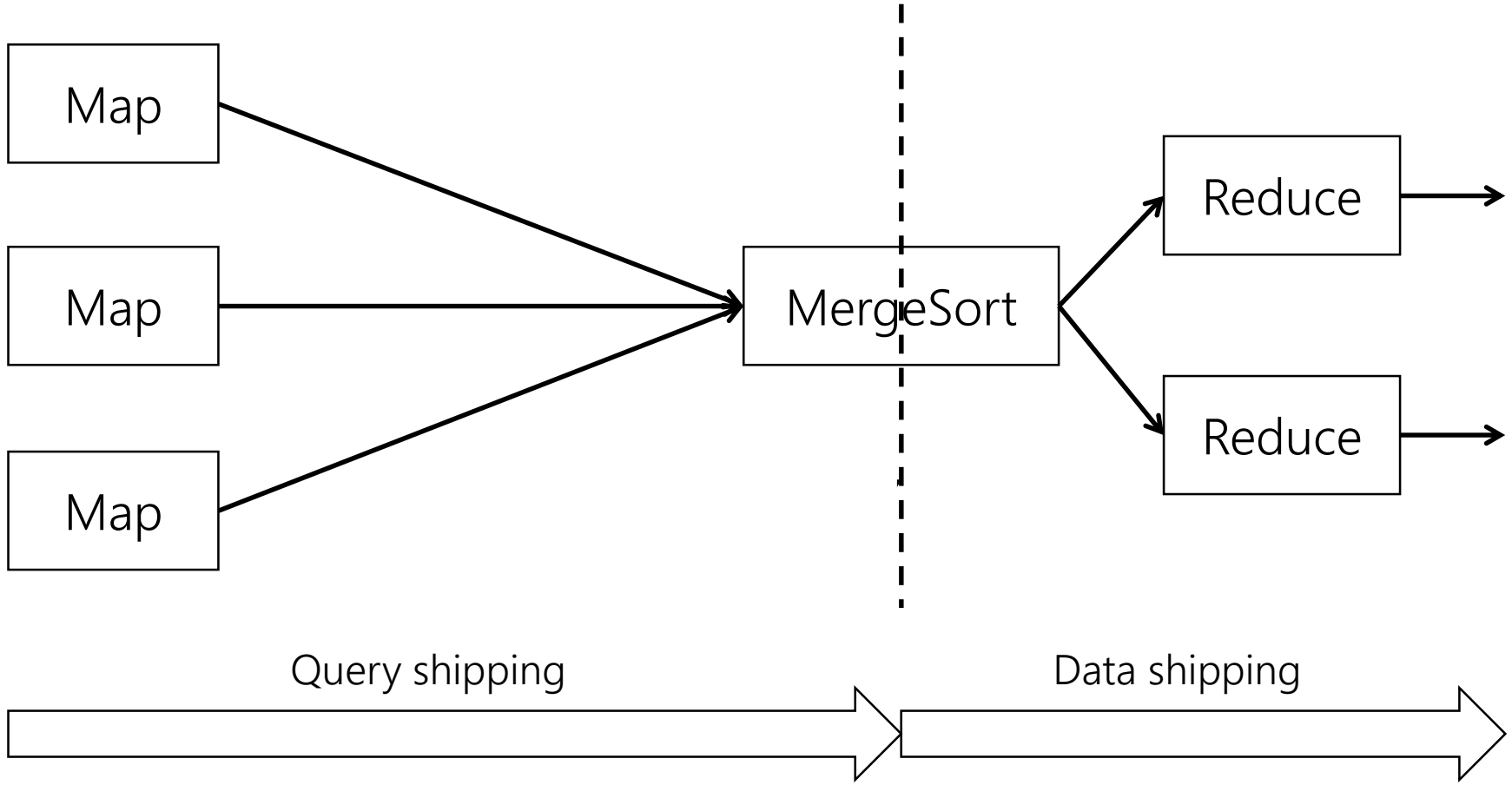
MapReduce Main Components

- 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



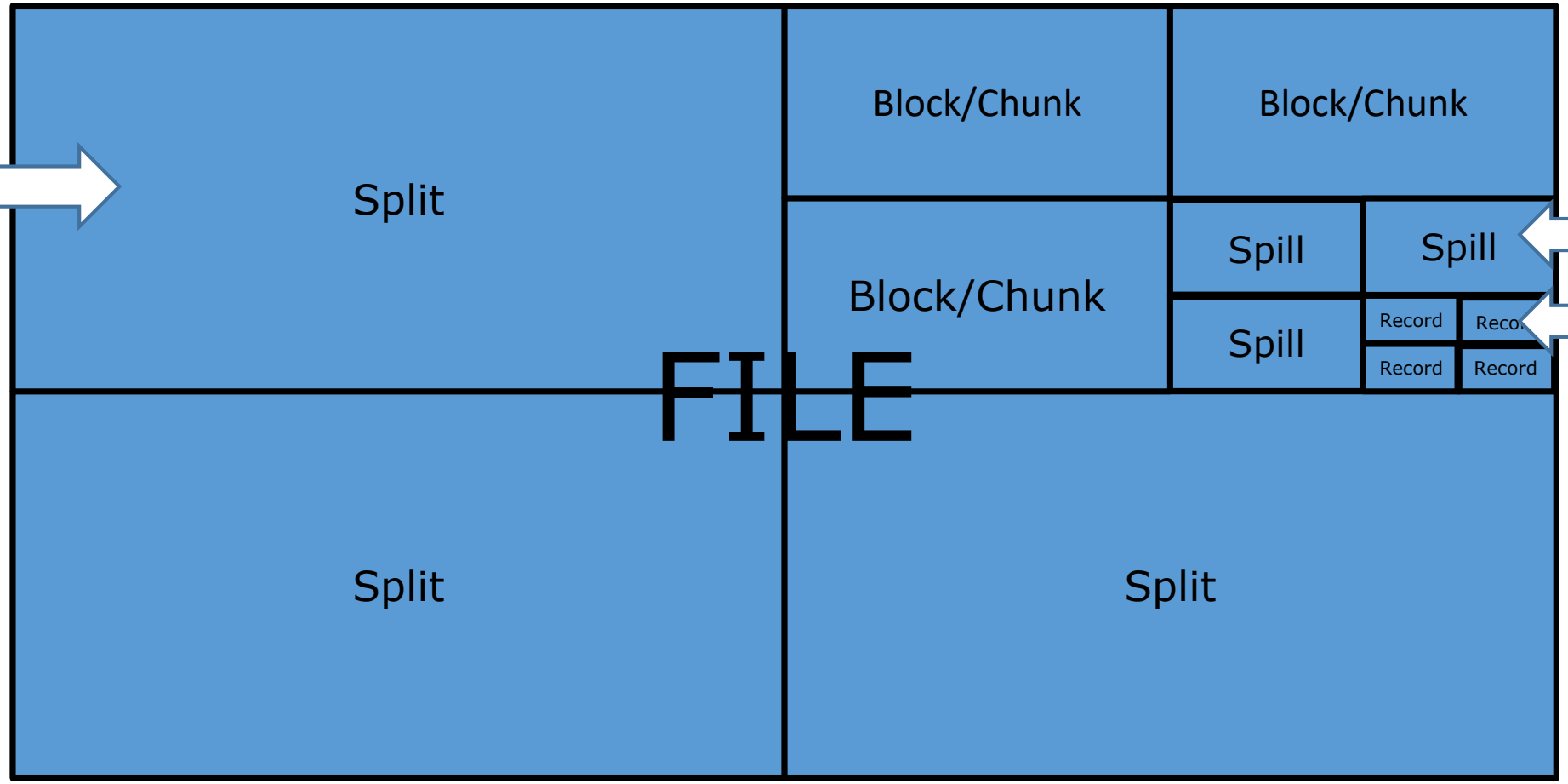
Query shipping vs. data shipping



MapReduce Objects

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

Set of blocks read by a mapper (usually 1)



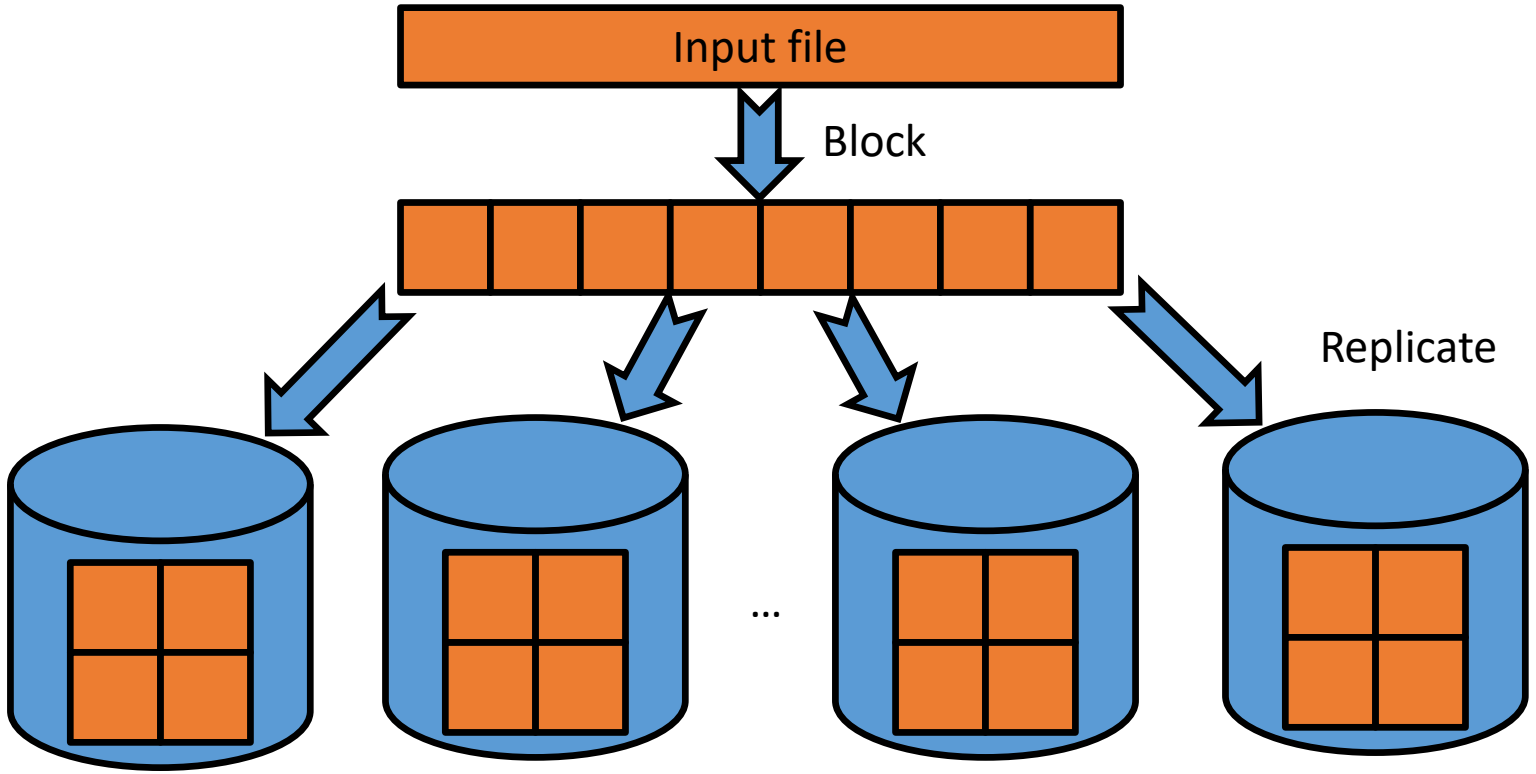
The size of the buffer used to store the results of the mappers

Each record is a key-value pair

Internal algorithm

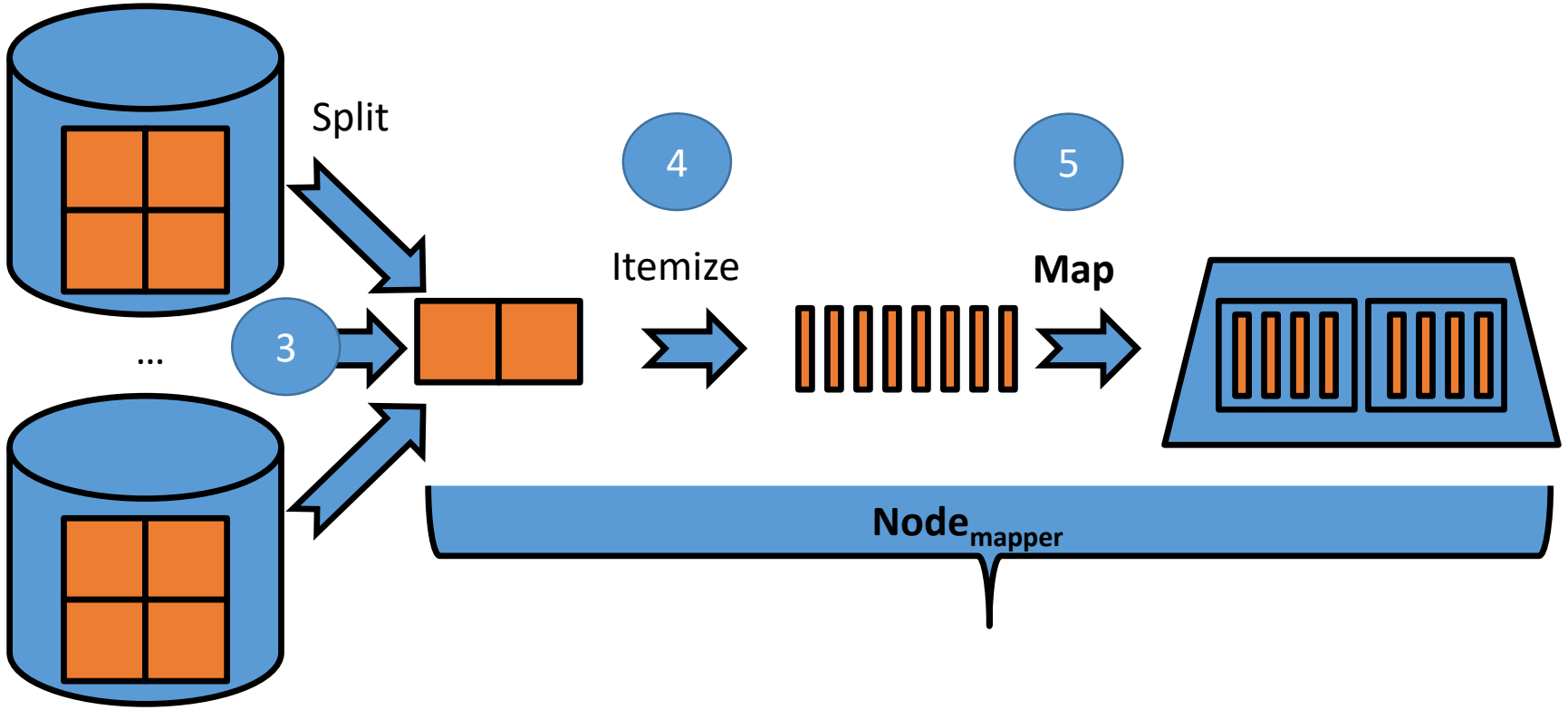
Algorithm: Data Load

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



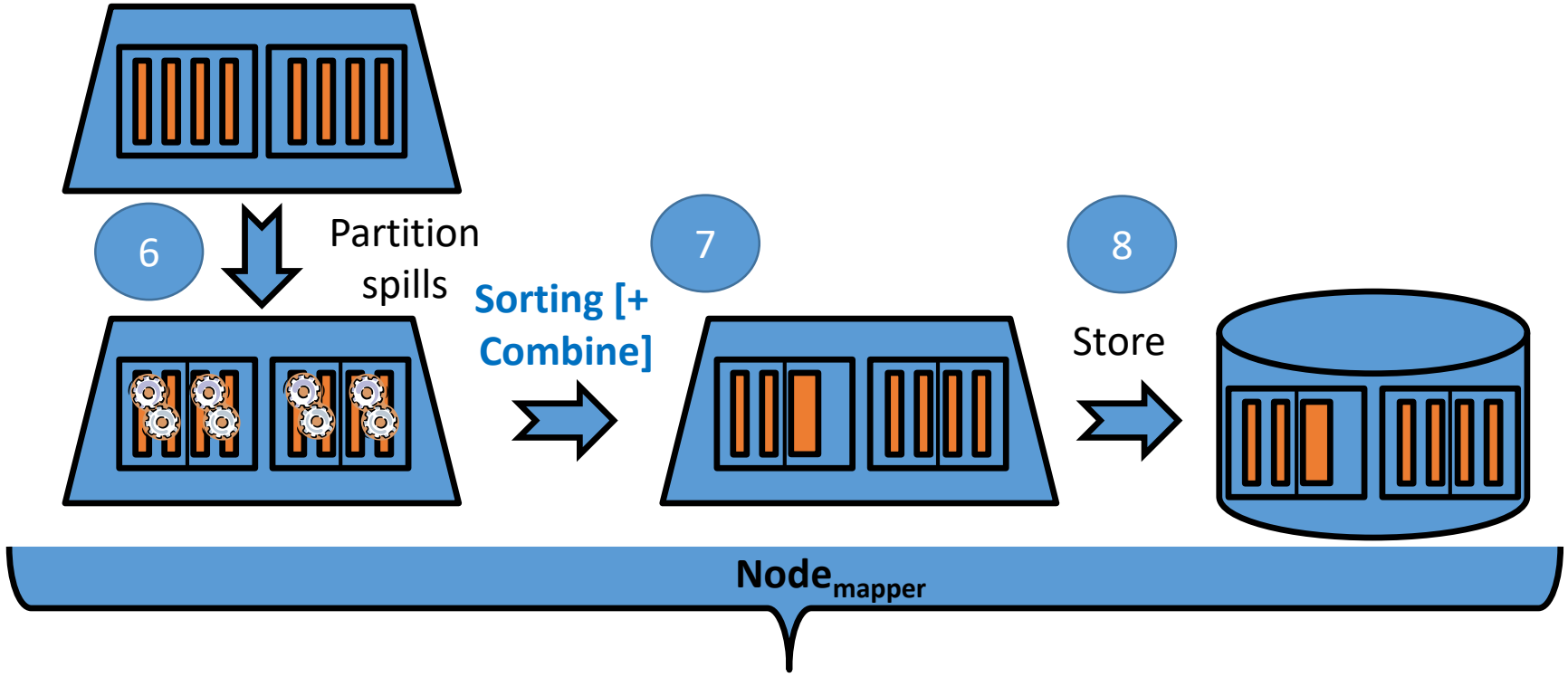
MapReduce Algorithm: Map Phase (I)

3. Each mapper reads a split (logical chunk assigned to a map task, usually 1 block)
4. Divides the split into records, i.e. converts bytes onto key-value pairs (*itemization*)
5. Executes the map function for each record and leaves them in memory divided into spills



MapReduce Algorithm: Map Phase (II)

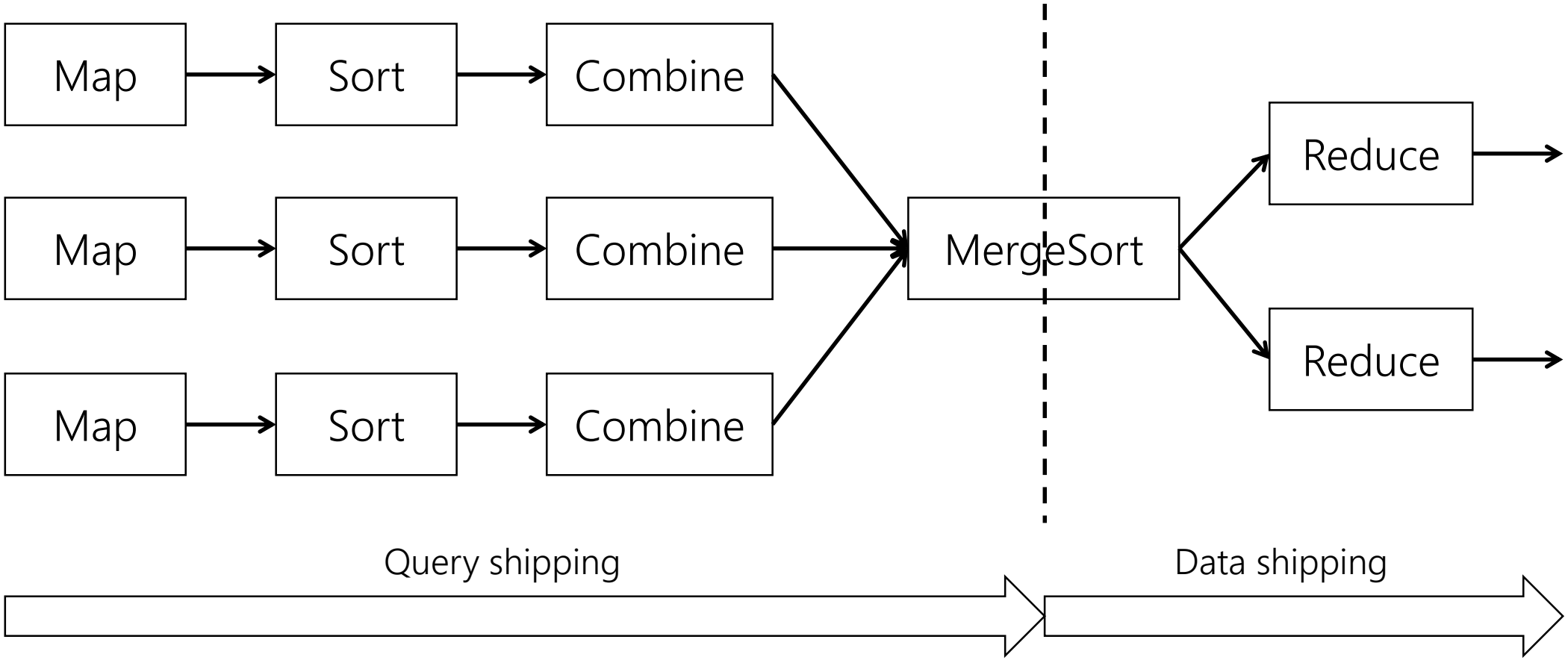
6. Each spill is then partitioned per reducers
   - Using a hash function f over the new key, according to the number of reducers R; i.e., f(x) = x%R
7. Each spill partition is sorted independently
   - If a combine function is defined, it is executed locally after sorting
8. Store the spill partitions into disk (massive writing)



The Combine Function (I)

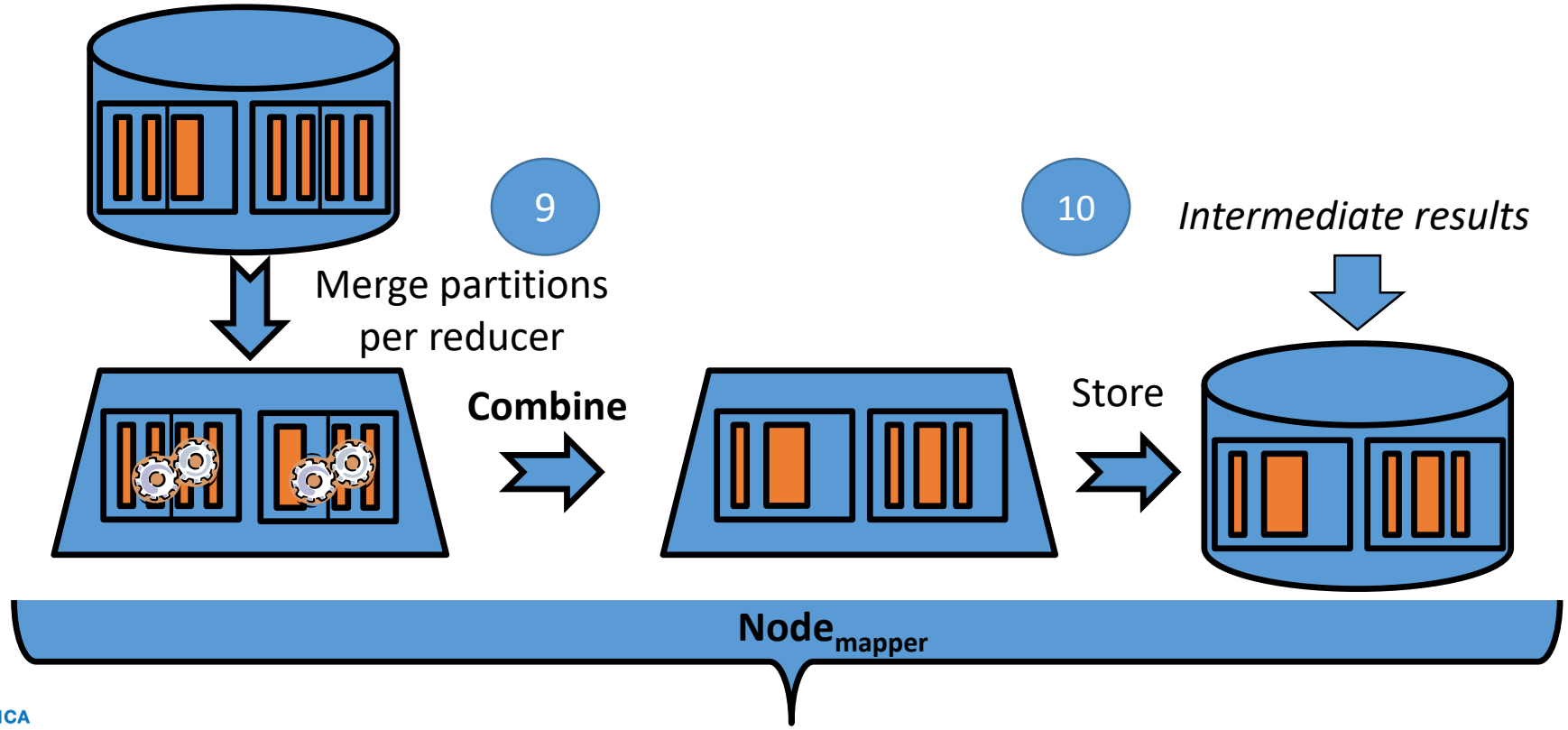
- Combine is executed locally in the Mapper nodes
  - Reduces the number of tuples sent to reducers
- Only possible when the combine function is
  - Commutative
  - Associative
- Typically, the combine and reduce functions are the same

The Combine function (II)



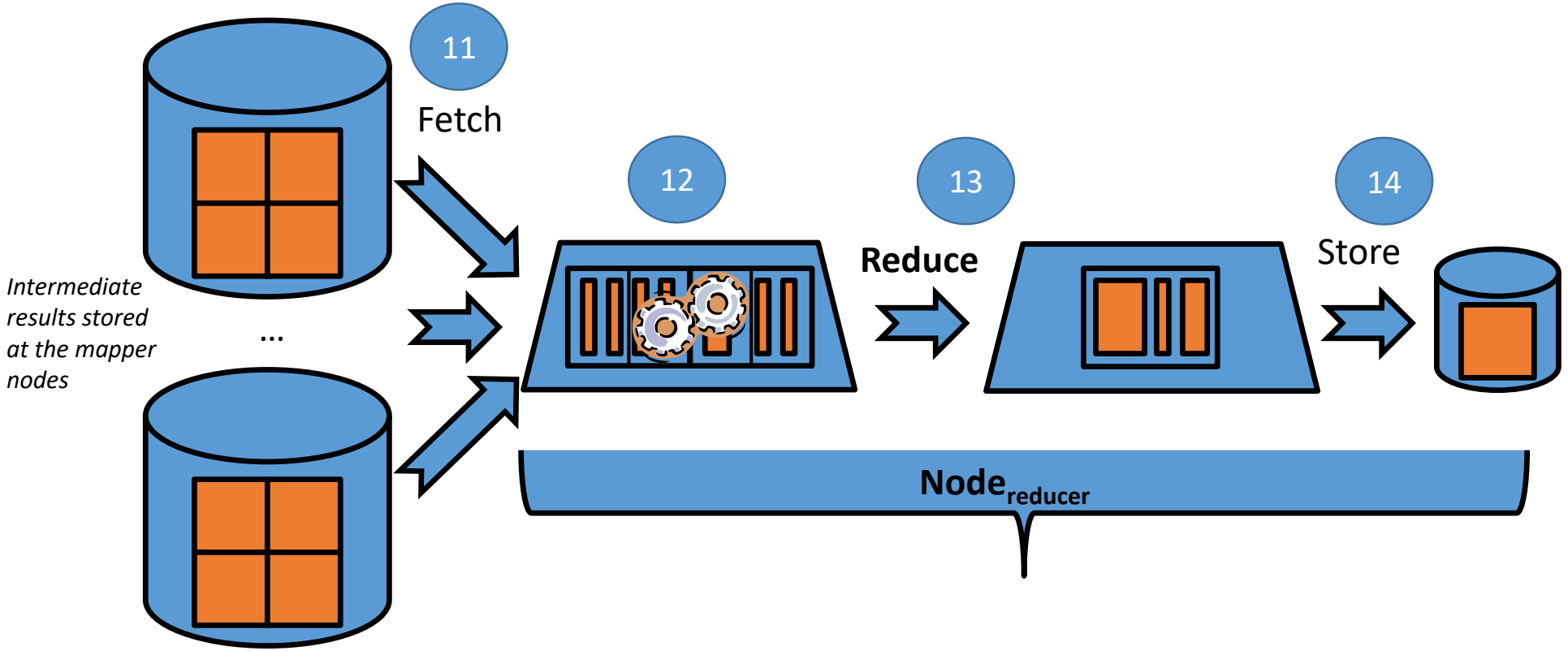
MapReduce Algorithm: Map Phase (III)

9. Spill partitions are merged per reducer
   - Each merge is sorted independently
   - Combine is applied again
10. Store the result into disk into a single map output file



MapReduce Algorithm: Shuffle and Reduce

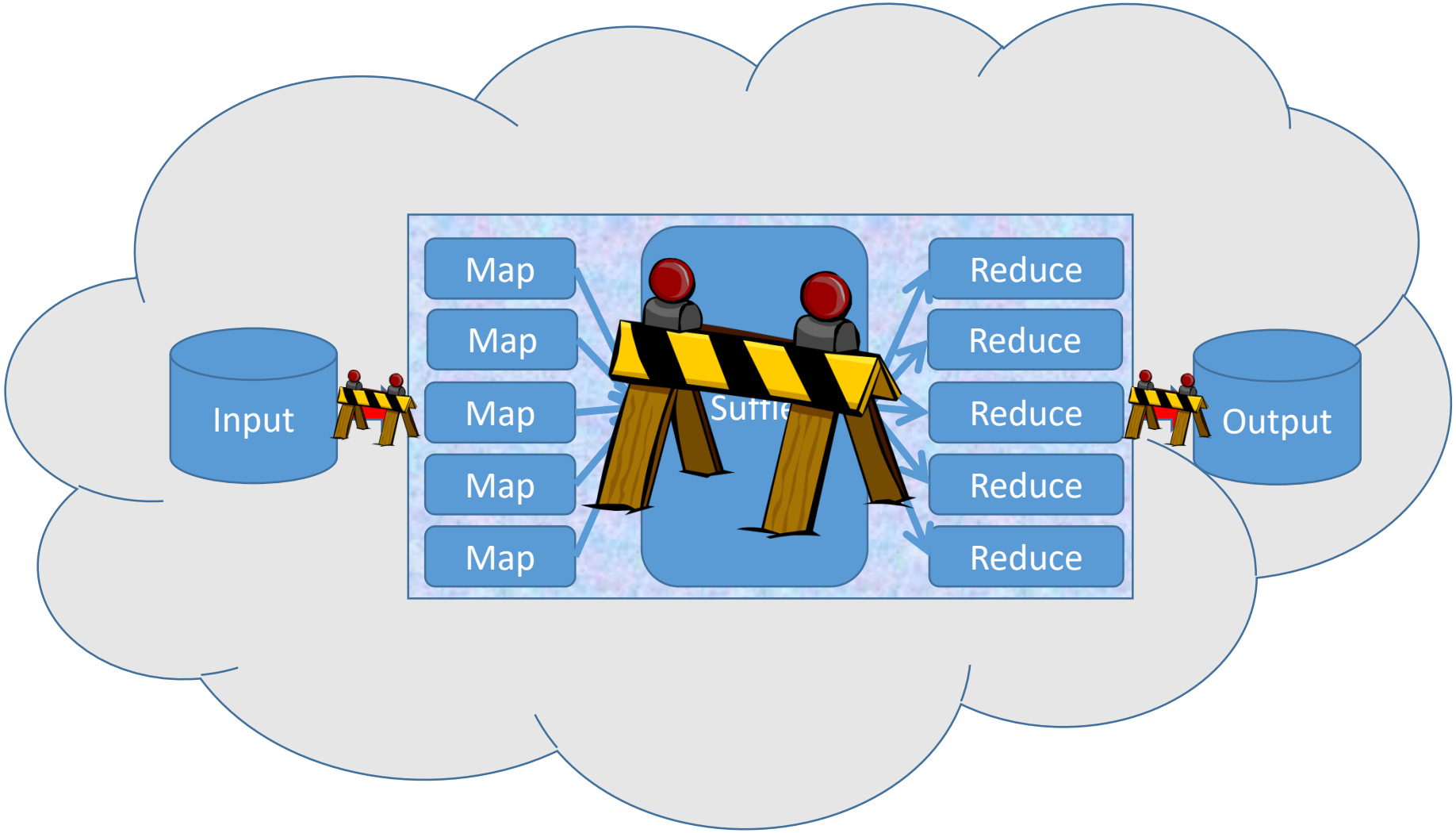
11. Reducers fetch the intermediate results from mappers (massive data transfer)
12. Intermediate results fetched are merged and sorted
    - When merging files, for those <k,v> with the same k a list of values is generated: k, {v_1, ..., v_n}>
13. Reduce function is executed per key
14. Store the result into disk



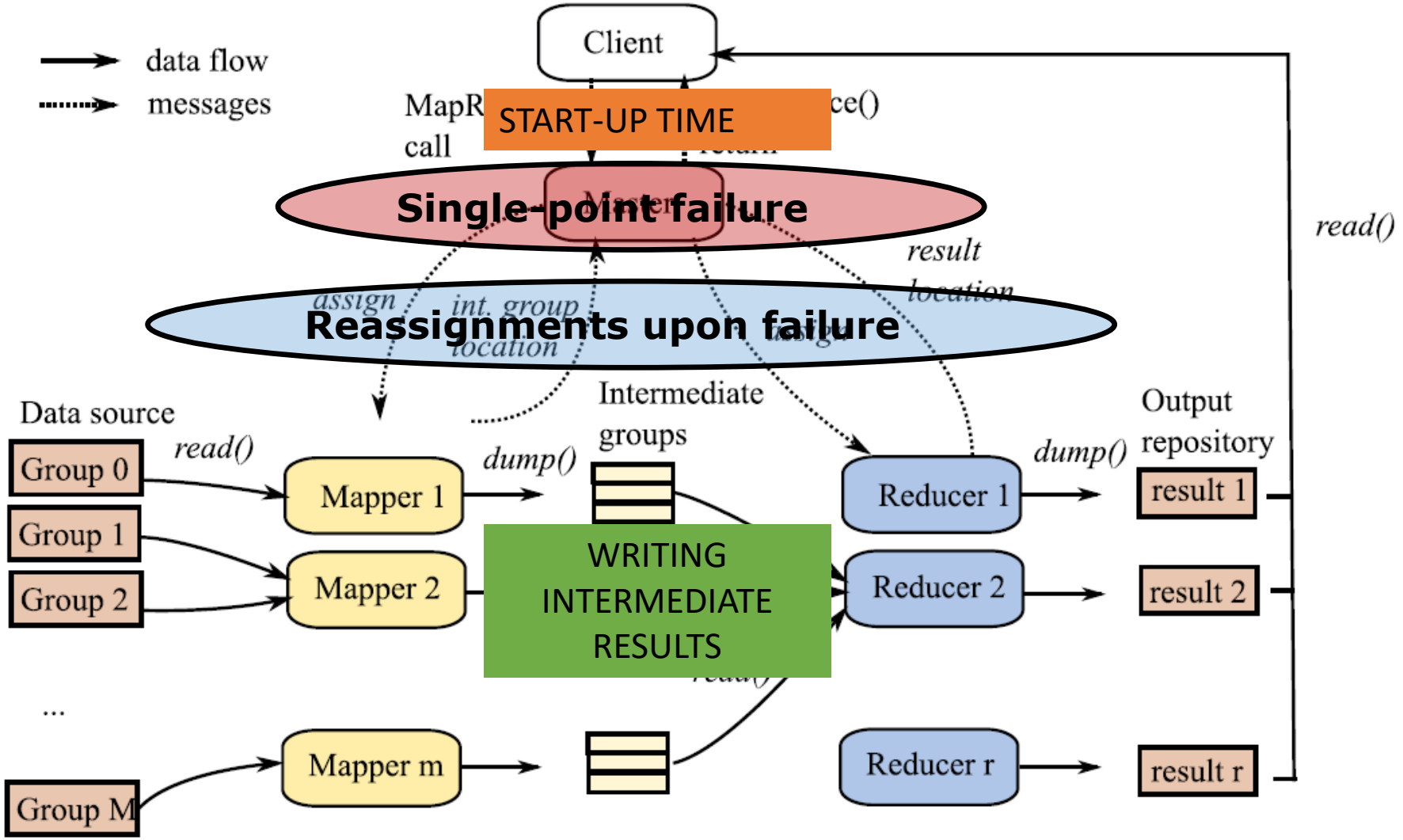
Drawbacks

Bottlenecks of the algorithm

Synchronization Barriers



MapReduce: Tasks and Data Flows



Implementing Operations with MapReduce

- The MapReduce paradigm program is computationally **complete** and **ANY** program can be adapted to it
- Furthermore, MapReduce's signature **is closed**
  - Thus, iterations can be achieved by nesting a MapReduce job
- However, some tasks better adapt to it than others
  - Easily adaptable:
    - Aggregations
    - Selections
    - Projections
    - Set operators
    - Sorting
  - Difficult for:
    - Joins
    - Any other operation needing to access data in another fragment to be executed (i.e., against the independence principle)

All these operators can be executed in each fragment independently (independence principle of their execution)

Beyond MapReduce: 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

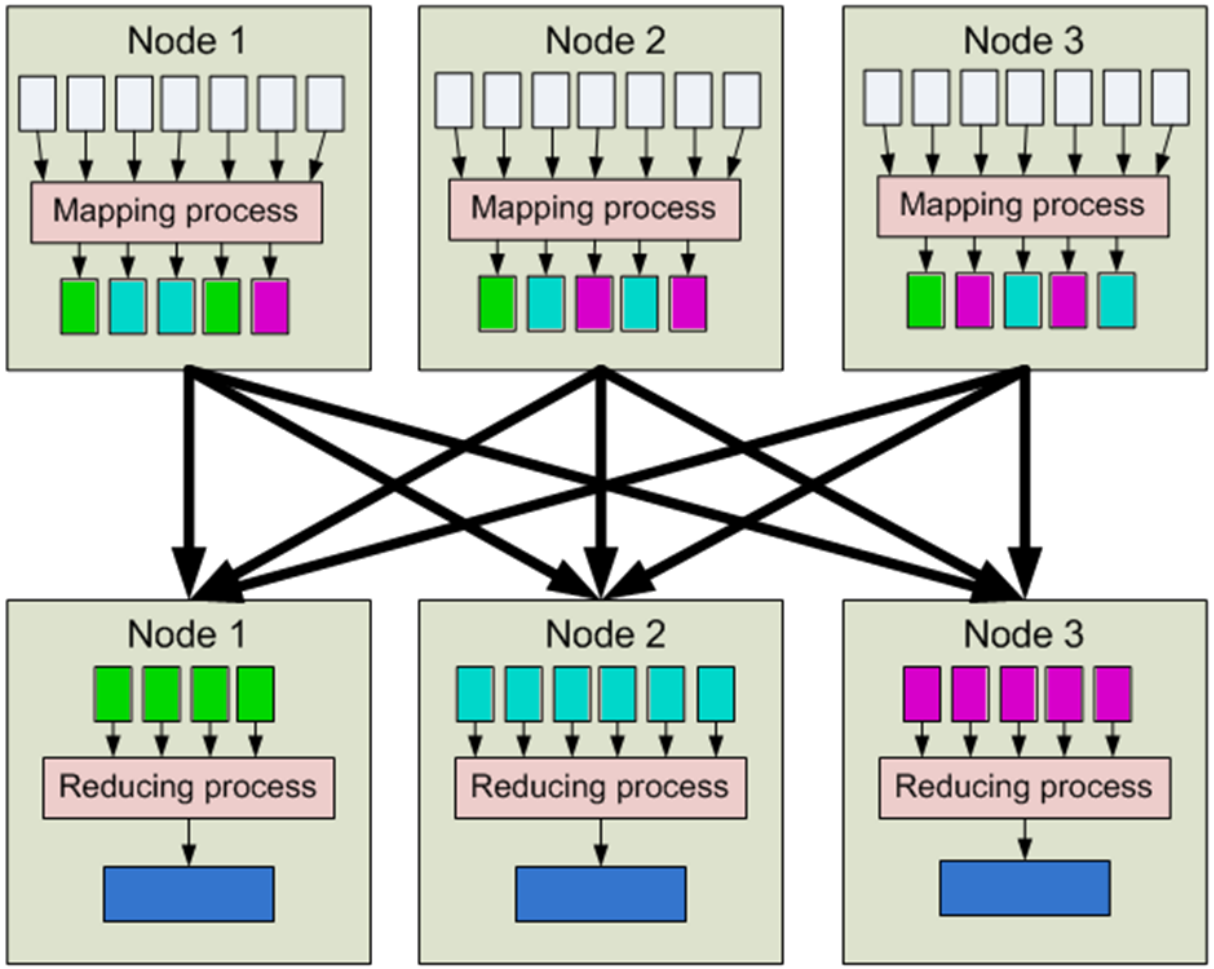
- Based on a richer data structure (dataframe)
- Instead of two operations: map and reduce, it provides more than 20 operators
  - Projection, Selection, Join, Group By… welcome back!
  - Reduces the black box problem in MapReduce
- Models the Spark job execution as a graph defined by the user programmatically
  - 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)

Putting everything together

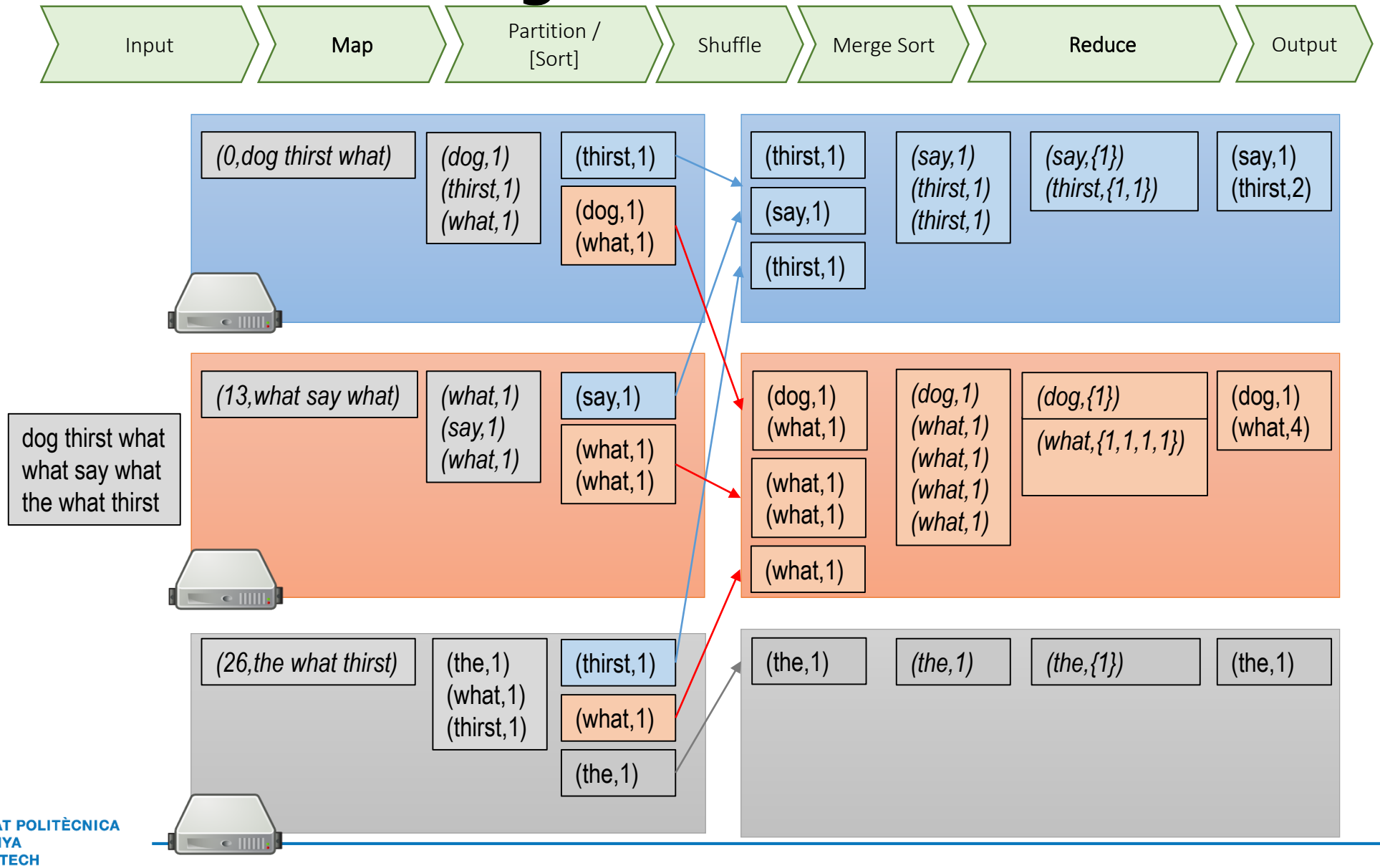
Executing a MapReduce job step by step

MapReduce

1. Input
2. Map
3. Partition [Sort]
4. Shuffle
5. Merge Sort
6. Reduce
7. Output

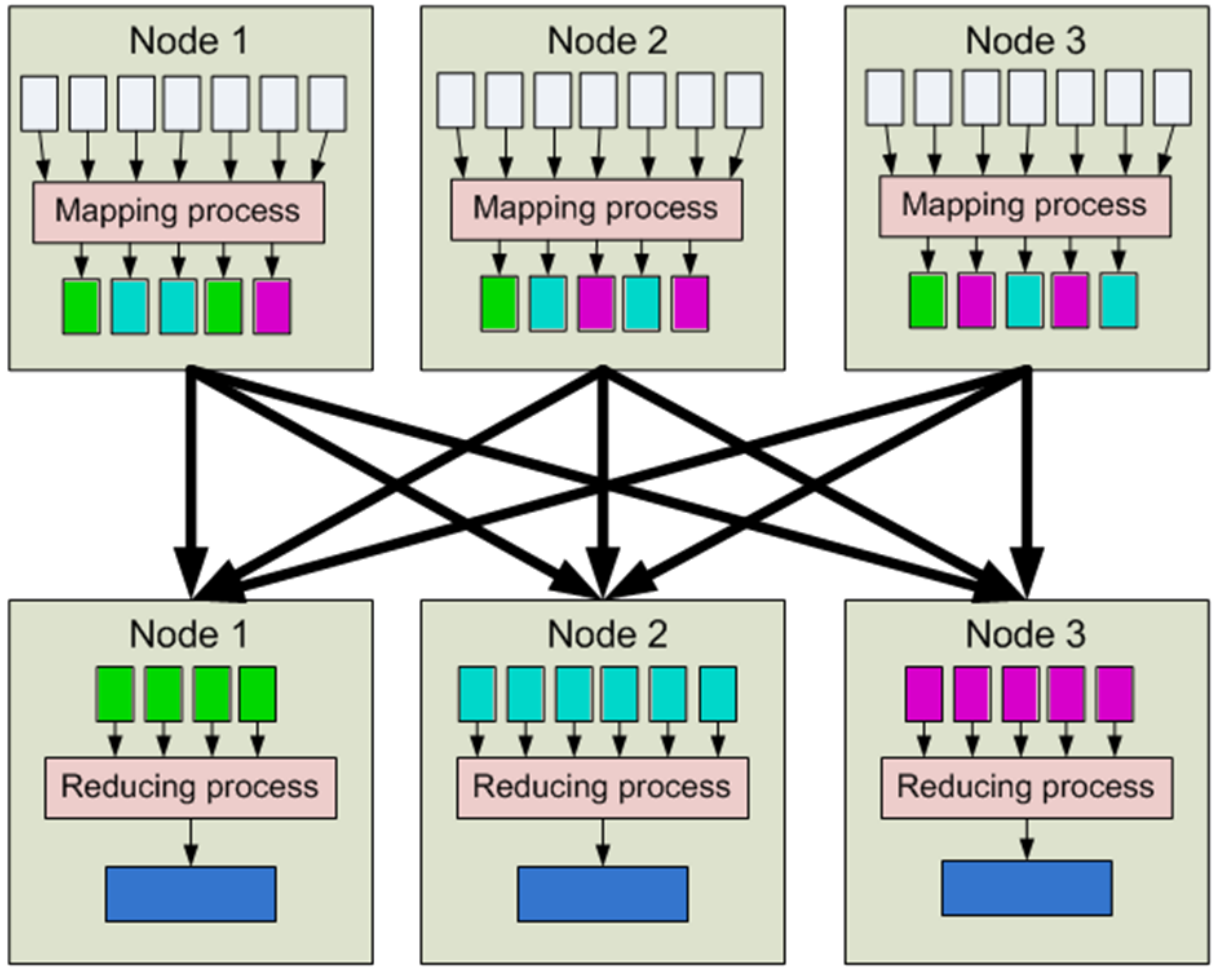


MapReduce: Counting Words

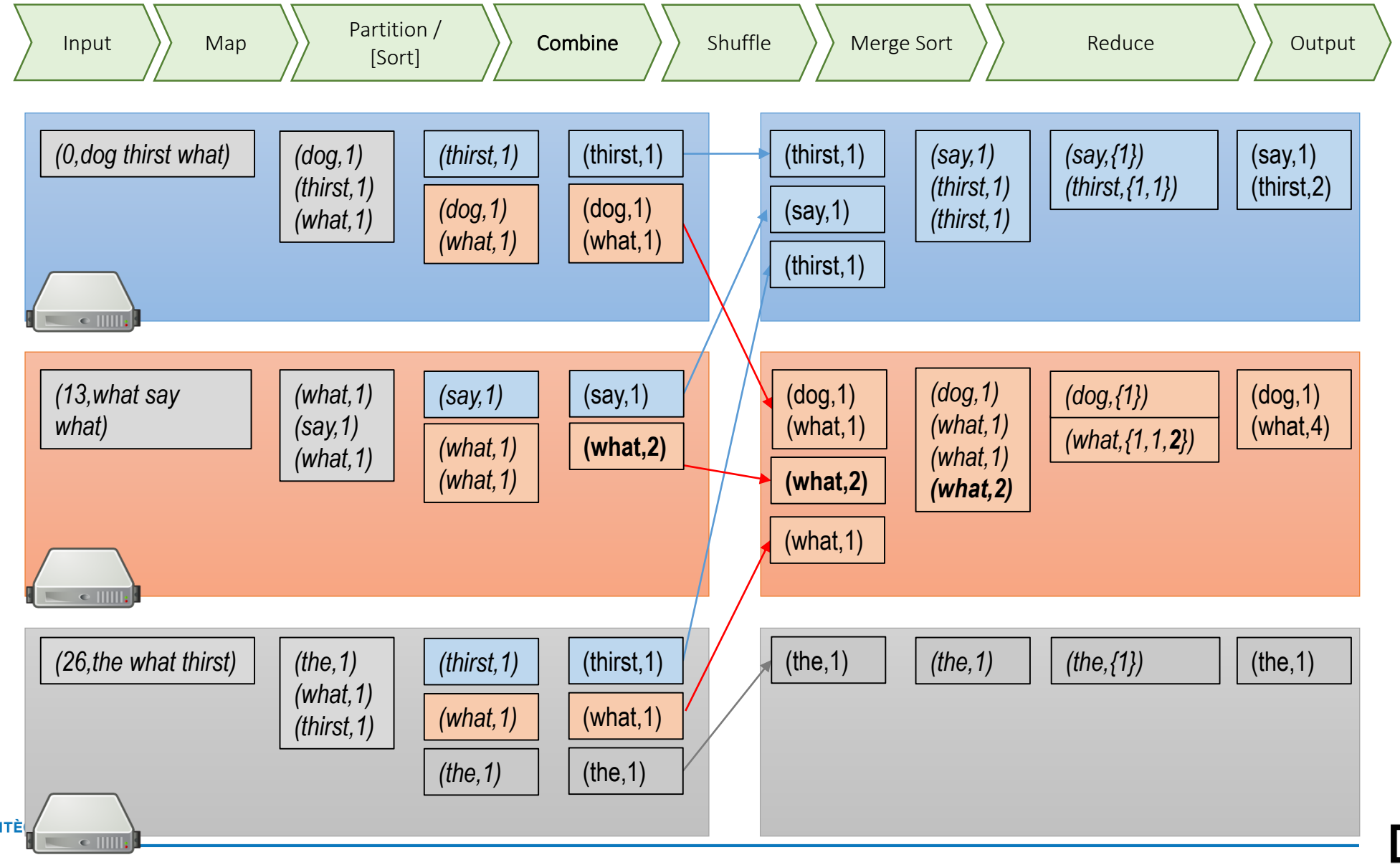


MapReduce: Combiner

1. Input
2. Map
3. Partition [Sort]
   (*"Combine"*)
4. Shuffle
5. Merge Sort
6. Reduce
7. Output

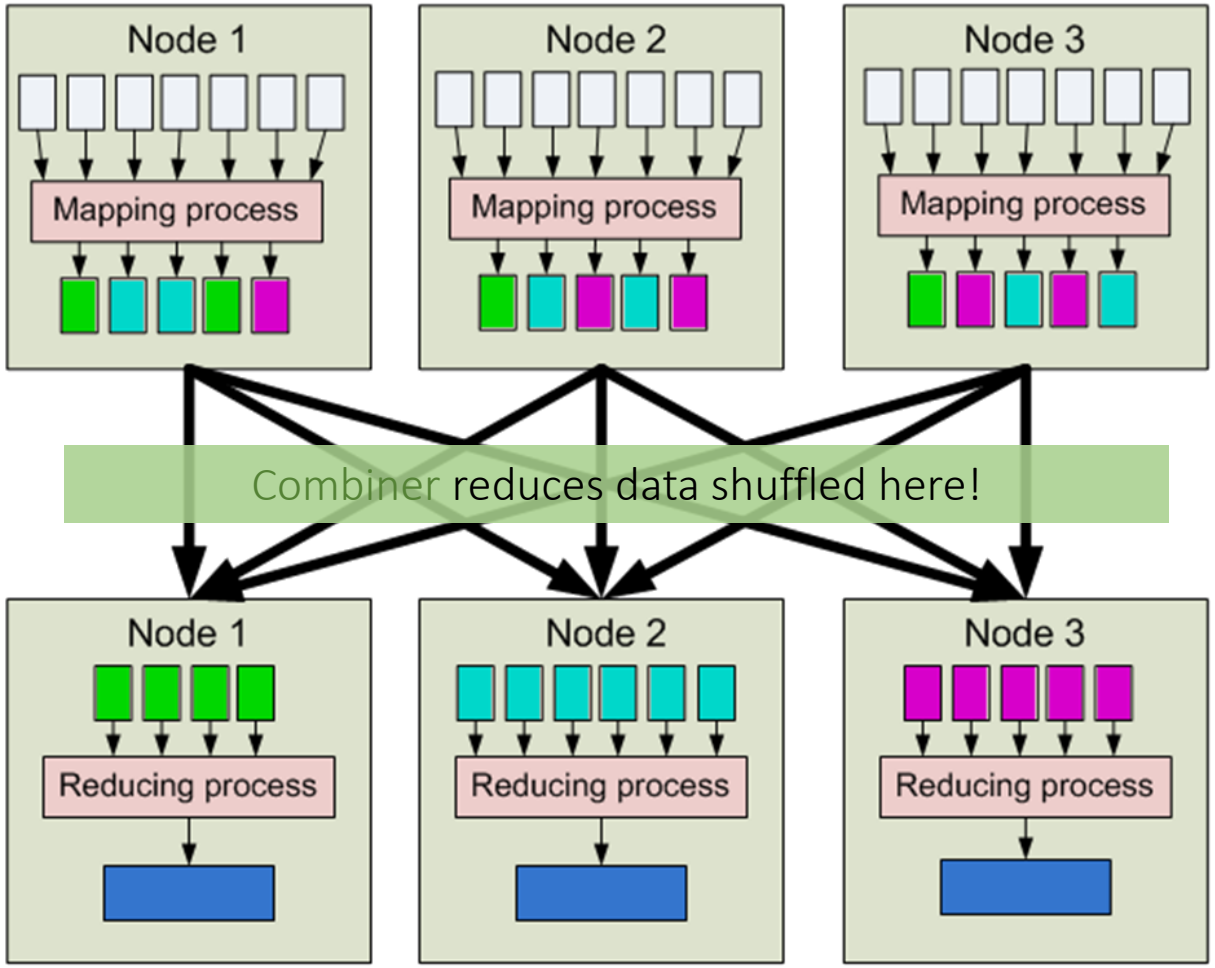


MapReduce: Combiner



MapReduce: Combiner

1. Input
2. **Map**
3. Partition [Sort]
   (“*Combine*”)
4. Shuffle
5. Merge Sort
6. **Reduce**
7. Output



Summary

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

Bibliography

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

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

S. Abiteboul et al. Web Data Management, 2012

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