NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Google Map & Reduce

lecture 10 (2021-04-19)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

Table of Contents

Google MapReduce

http://static.googleusercontent.com/media/research.google.com/pt-PT//archive/mapreduce-osdi04.pdf

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an imple-

Google's MapReduce: summary

- a programming **model** and an associated **implementation** for processing **large datasets**
- runs on a large cluster of **commodity machines** … a typical … computation processes many terabytes of data on **thousands** of machines
- a new abstraction that allows us to expresses **simple computations** we were trying to perform but **hides the messy details** of parallelization, fault-tolerance, data-distribution and load-balancing in a library

Programming model: batch processing

- Map-reduced is designed for batch processing
- Batch processing:
  - All input is known when the computation starts
  - The complete computation is executed
  - No interactions with the user

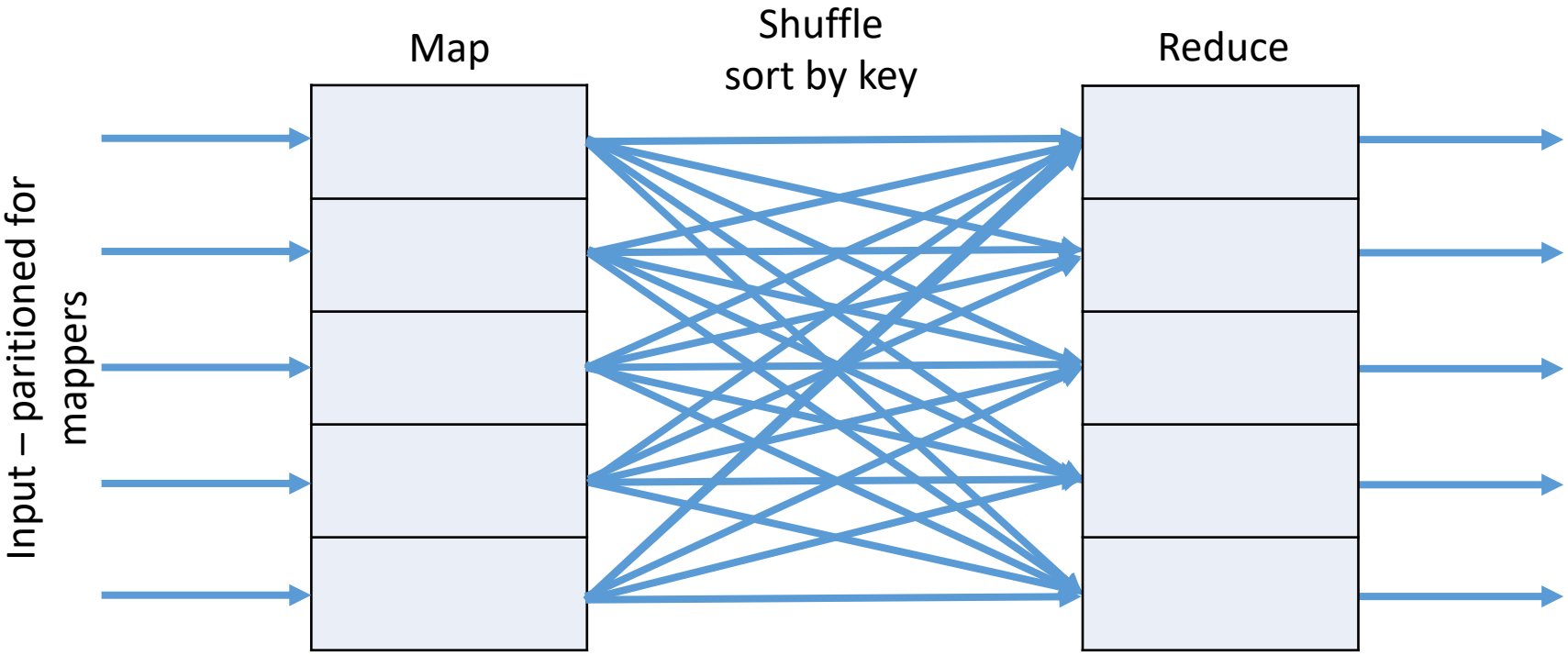
Programming model

- Sequence of map and reduce stages
- Map: processes input (files); emit tuples
- Reduce: process tuples grouped by key; Emit tuples

Programming model

- Sequence of map and reduce stages
- Map: processes input (files); emit tuples
  - emit (k_1, v_1)
  - emit (k_2, v_2)
  - emit (k_1, v_3)
- Reduce: process tuples grouped by key; Emit tuples
  - input (k_1, {v_1, v_3})
  - Input (k_2, {v_2})
  - emit (k_a, v_a)
  - emit (K_b, v_b)
  - emit (k_c, v_c)

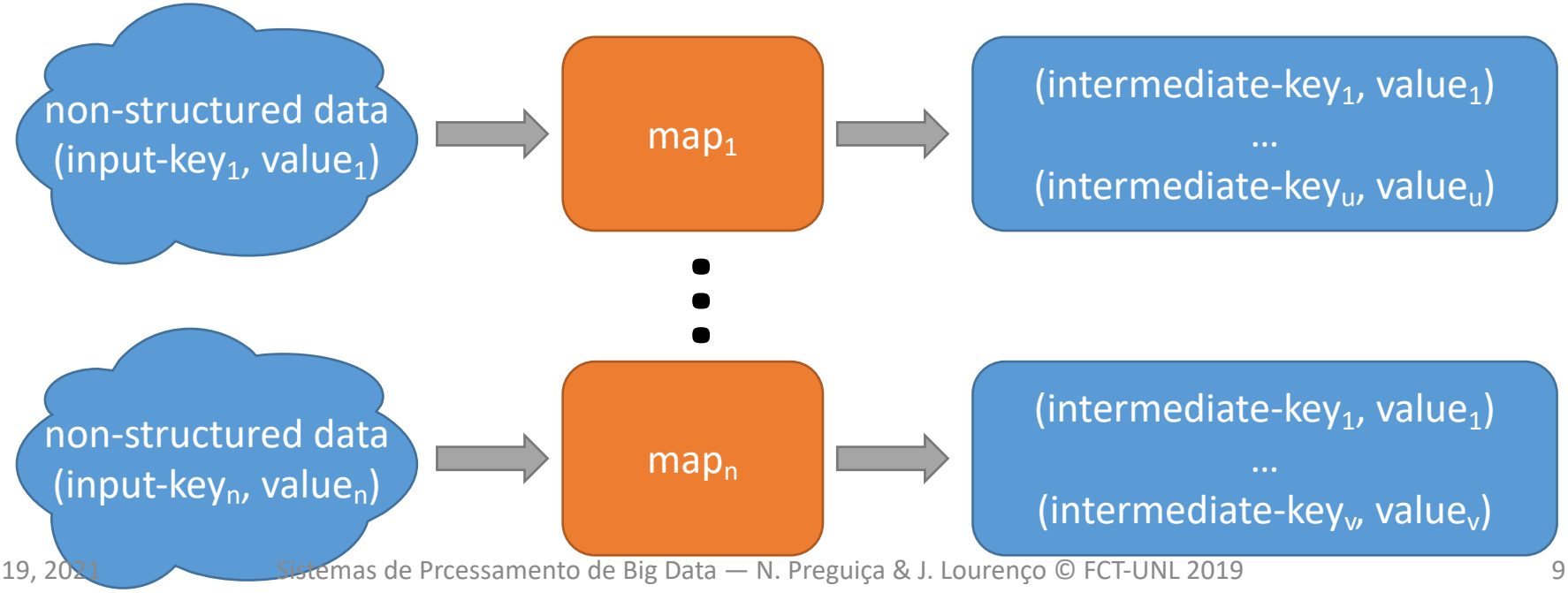
Map-reduce steps



- Input is typically (key, value) pairs
  - Partitioned for multiple mappers
- Map and Reduce are performed by a number of processer

Map function

- Each map task processes a chunk of the input
- Input: non-structured data, (input-key, value)
- Output: (intermediate-key, value)



Shuffle and Sort: between map and reduce

- The output of the map is stored on local disk
- The infrastructure
  - sorts all the data by key
  - sometimes aggregates all the key/value pairs with the same key into a single pair key/list-of-values
  - sends (distribute/shuffle) the data among the nodes running reducers
- The sorted data is the input of the reduce tasks

Reduce function

- Each reduce task processes a chunk of the sorted data
  - All data with the same key is processed by the same reduce task
- Input: (intermediate-key, value)
- Output: (output-key, value), or non-structured output

Programming model... working

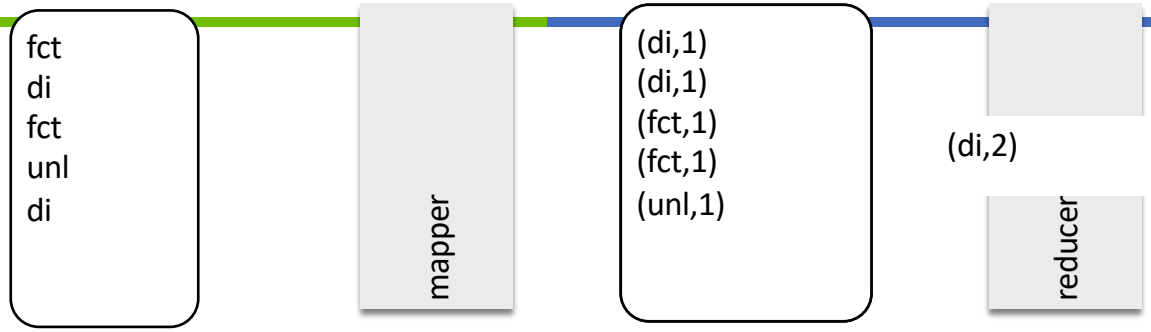
- Count the number of times each word appears in a document

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  for each v in values:
    result += ParseInt(v)
  Emit(key, AsString(result))
```

Programming model… working

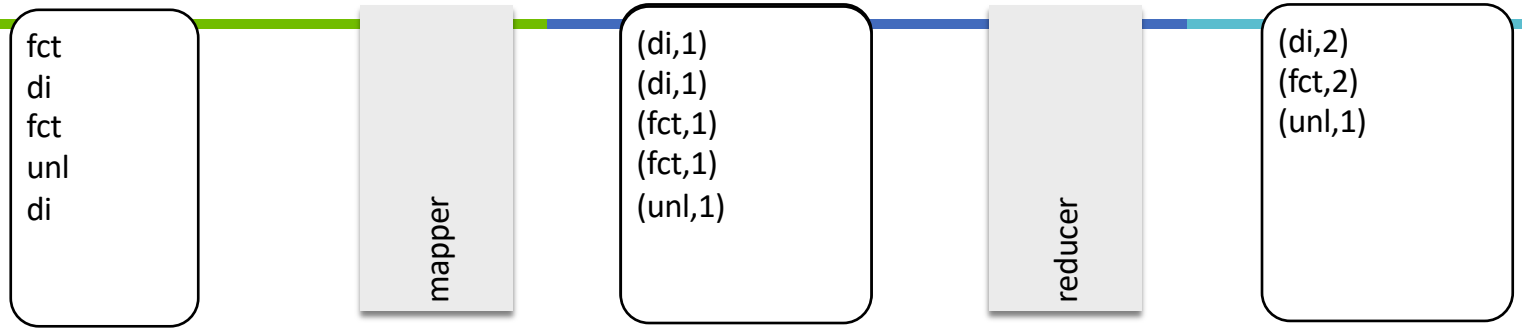

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  for each v in values:
    result += ParseInt(v)
  Emit(key, AsString(result))
```

Programming model… working




```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  for each v in values:
    result += ParseInt(v)
  Emit(key, AsString(result))
```

Programming model… working

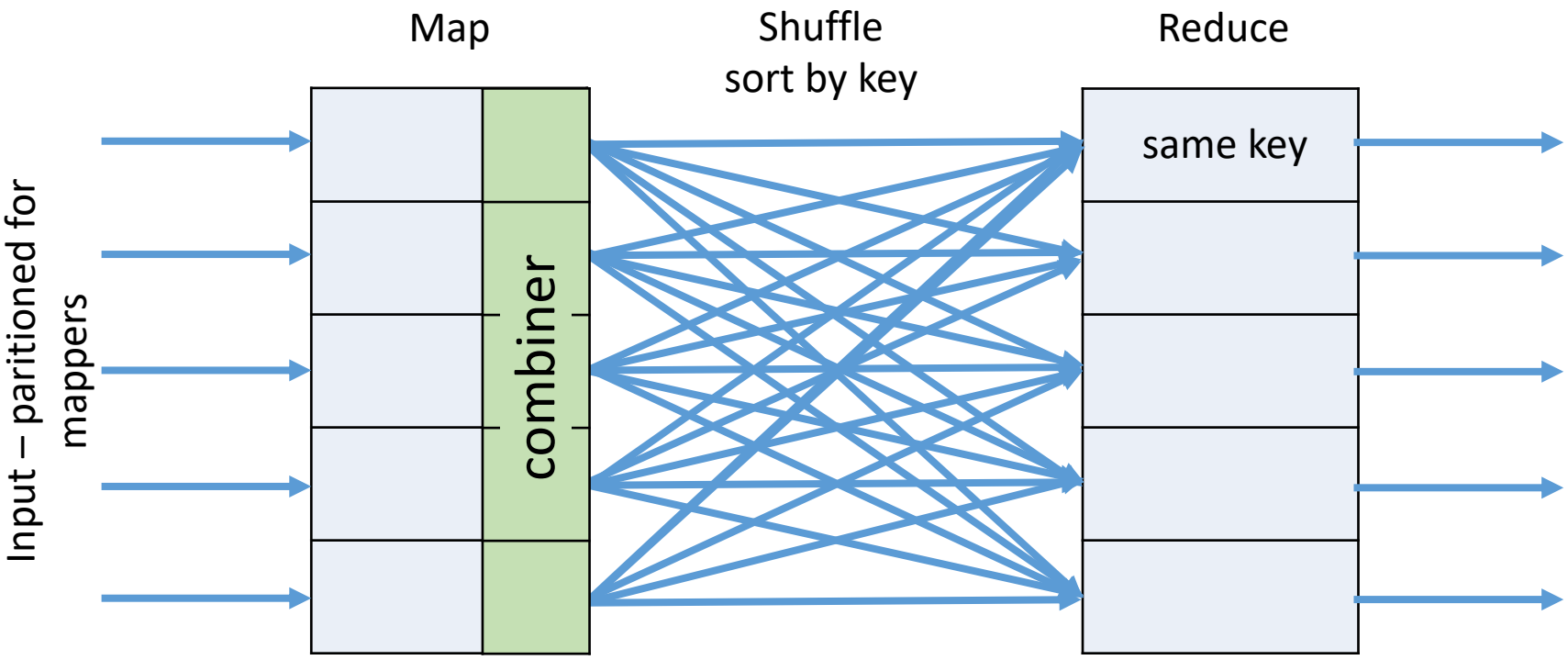



```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, “1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  for each v in values:
    result += ParseInt(v)
  Emit(key, AsString(result))
```

Extended programming model

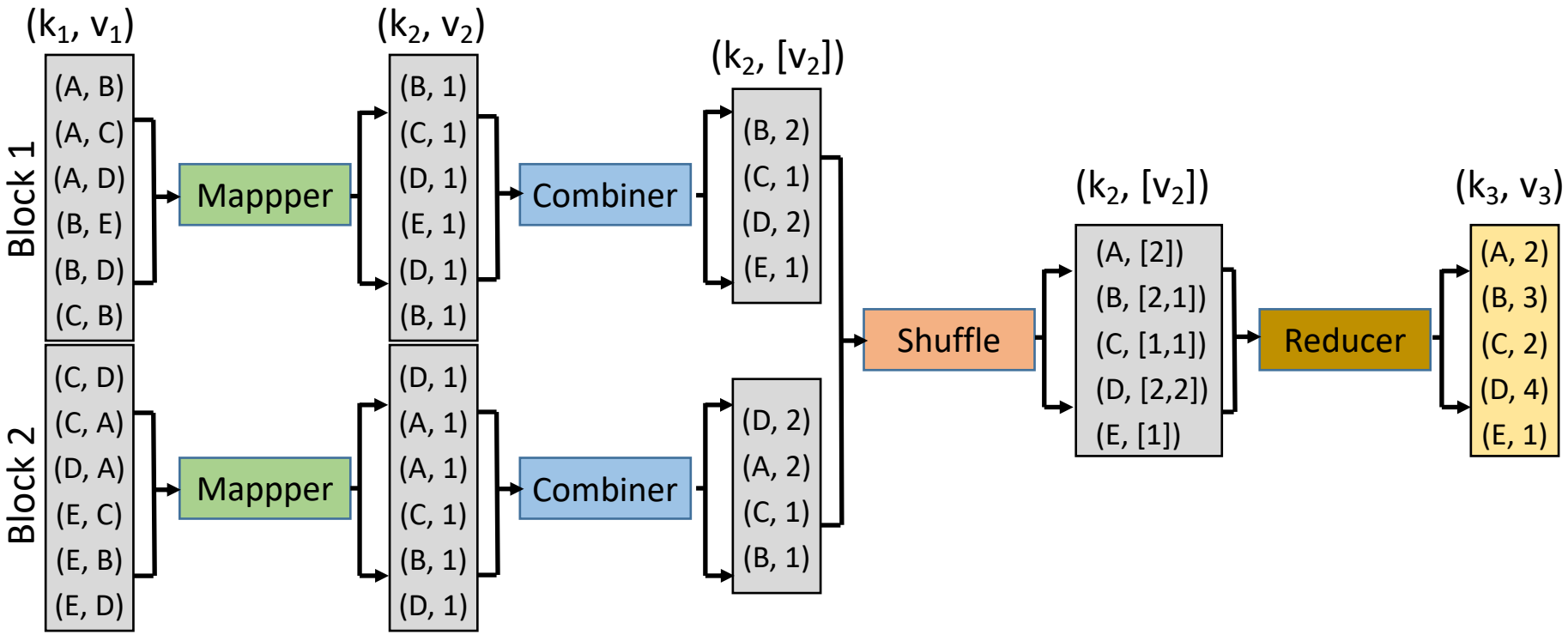
- In the word count example, what is the problem of emitting a tuple for each word?
- How could it be solved?
- Combiner phase

Extended programming model



- Combiner runs in each mapper
- Same interface as the reduce
  - (key, value) pairs generated by a mapper are sorted and fed to the combiner

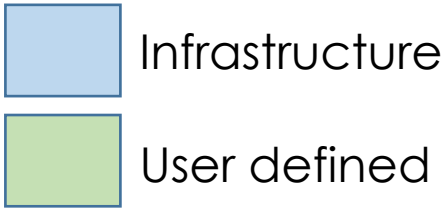
Combiner example



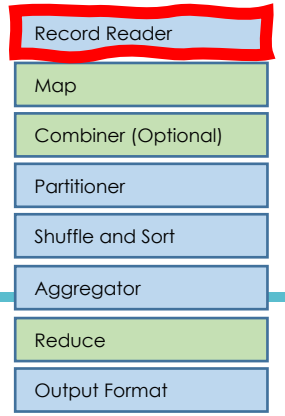
Map-reduce phases

- MapReduce is broken down into several steps:

1. Record Reader
2. Map
3. Combiner (Optional)
4. Partitioner
5. Shuffle and Sort
6. Aggregator
7. Reduce
8. Output Format

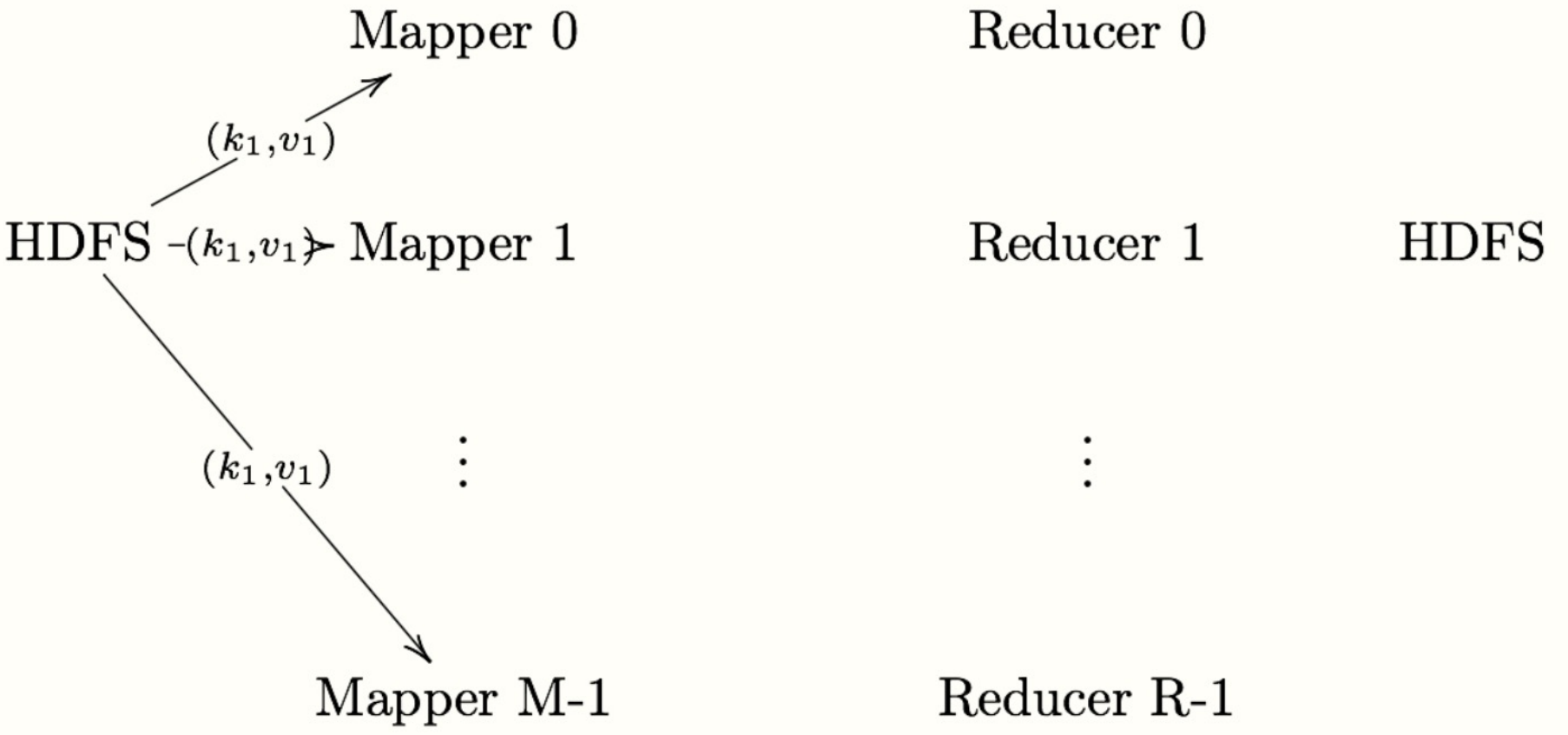


MR phases: Record Reader (1)



- **Record Reader** splits input into fixed-size pieces for each mapper
- The key is positional information (the number of bytes from start of file) and the value is the chunk of data composing a single record
- In hadoop, each map task's is an input split which is usually simply a HDFS block
- Hadoop tries scheduling map tasks on nodes where that block is stored (data locality)

MR phases: Record Reader (1)

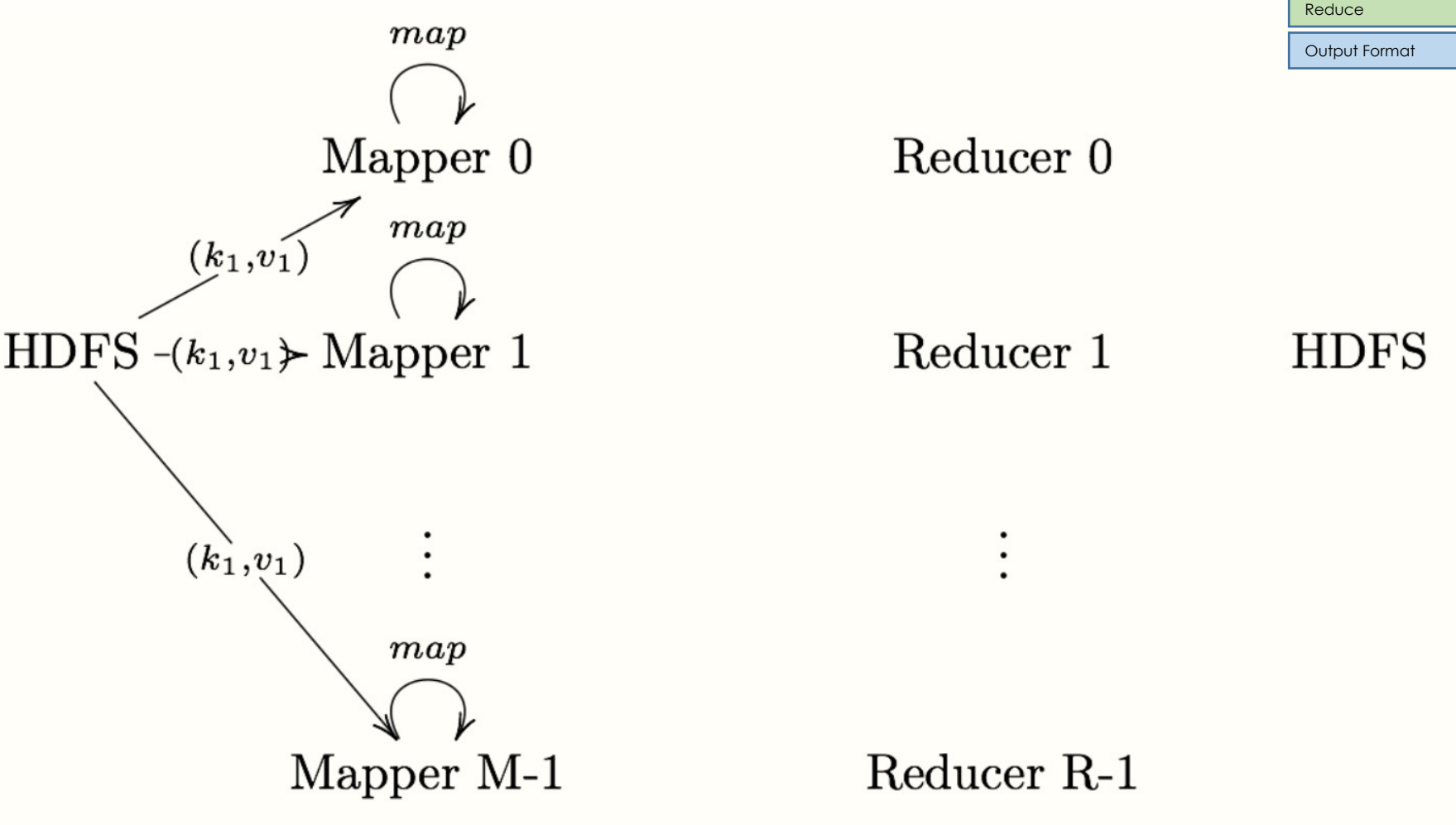


MR phases: Map (2)

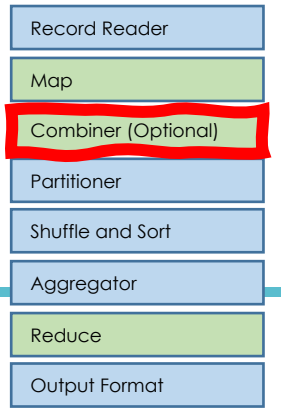
- **Map** User defined function outputting intermediate key-value pairs
- **key** (k_2): Later, the infrastructure will group and possibly aggregate data according to these keys. *Choosing the right keys is here is important for a good MapReduce job*
- **value** (v_2): The data to be grouped according to its key

MR phases: Map (2)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format



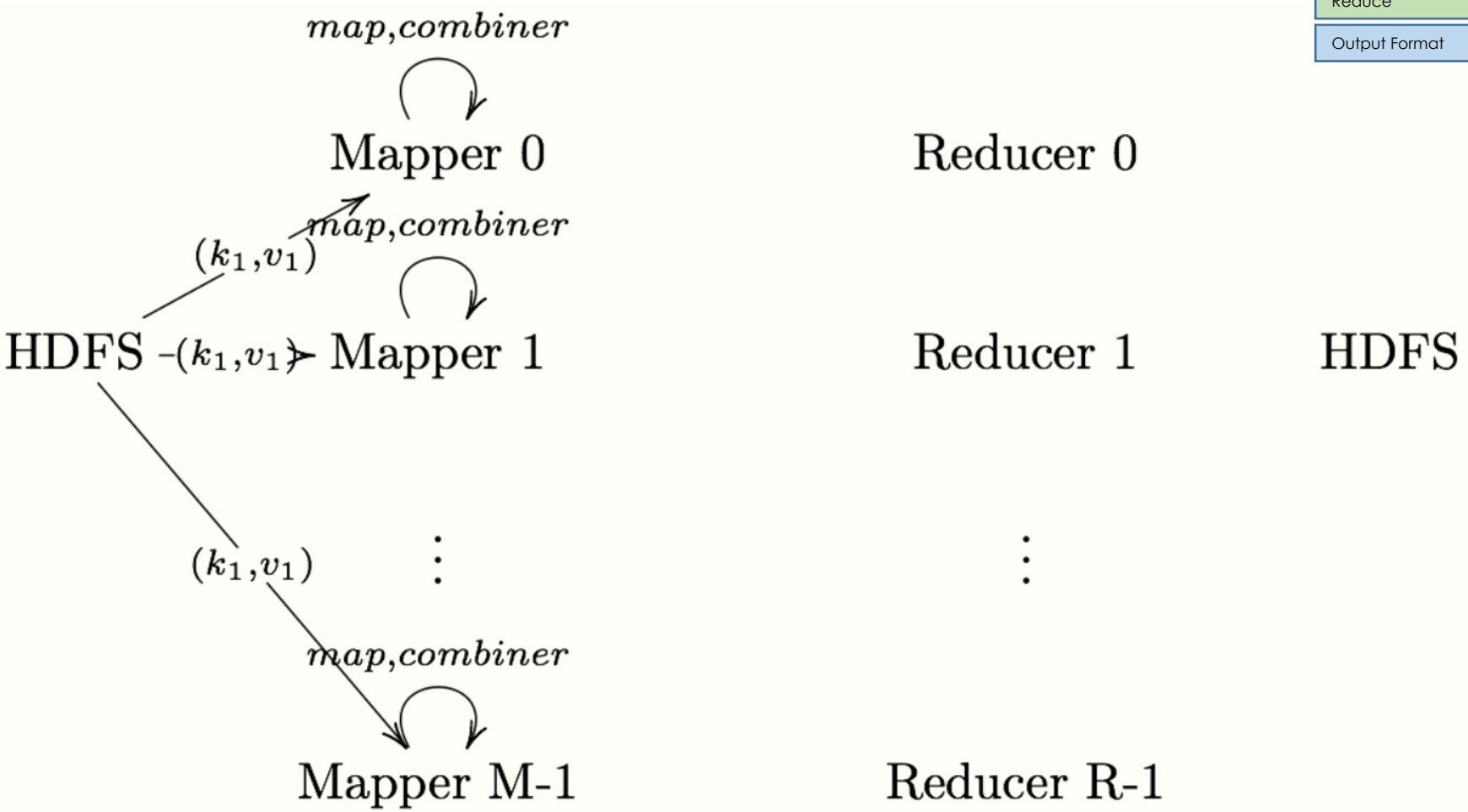
MR phases: Combiner (3)



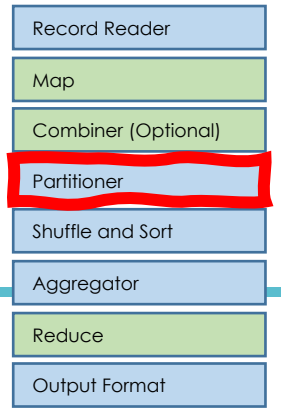
- **Combiner** (optional) UDF aggregating data according to intermediate keys on a mapper node
- This can usually reduce the amount of data to be sent over the network increasing efficiency
- Combiner should be written with the idea that it is executed over most but not all map tasks
  i.e., $(k_2,v_2) \mapsto (k_2,v_2)$
- Usually very similar or the same code as the reduce method

MR phases: Combiner (3)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format



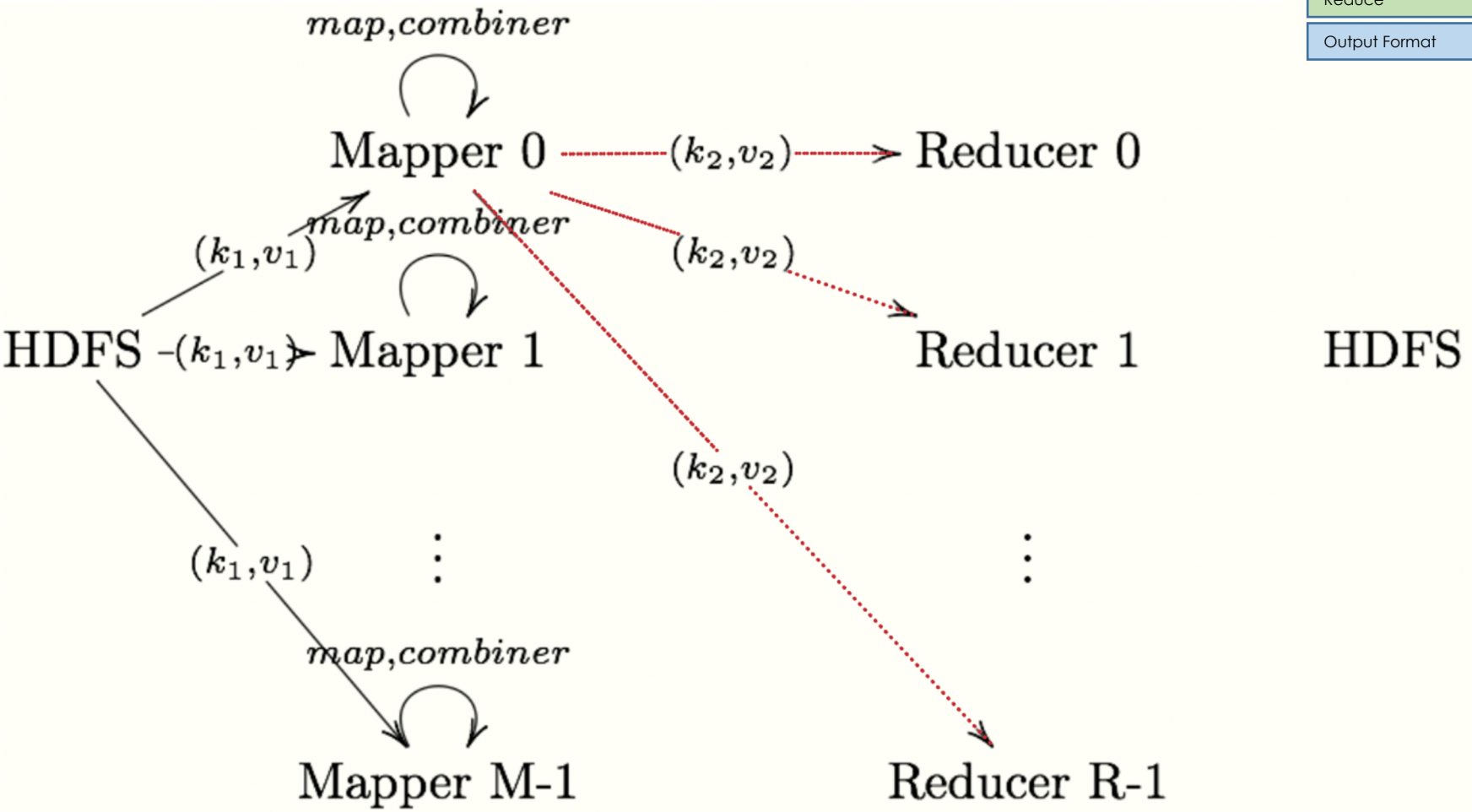
MR phases: Partitioner (4)



- **Partitioner** Where to send the key-value pairs (k_2,v_2) produced in the map phase?
- *Reducer# = hash(k) mod R*
- Will usually result in a roughly balanced load across the reducers
- A balancer system is in place for the cases when the key-values are too unevenly distributed
- In hadoop, the intermediate pairs (k_2,v_2) are written to the local harddrive and grouped by *reducer#* and k_2

MR phases: Partitioner (4)

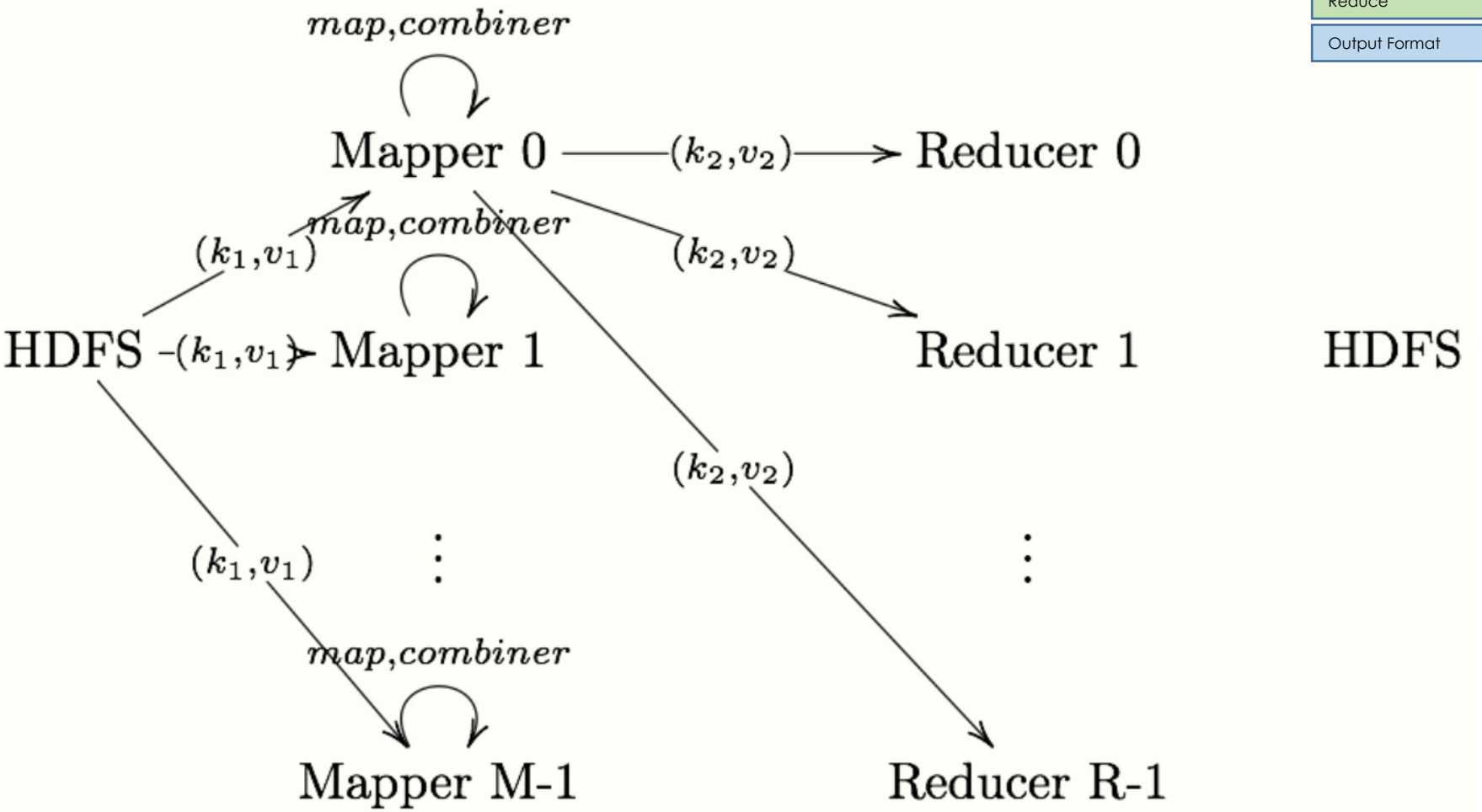
Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format



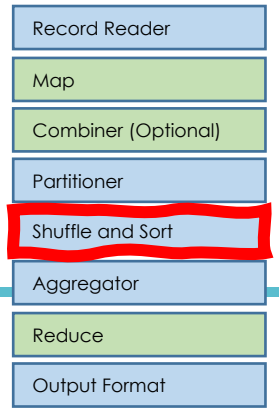
MR phases: Shuffle (5)

- **Shuffle** Effectively sends the intermediate pairs (k_2,v_2) to the reducer nodes

MR phases: Shuffle (5)



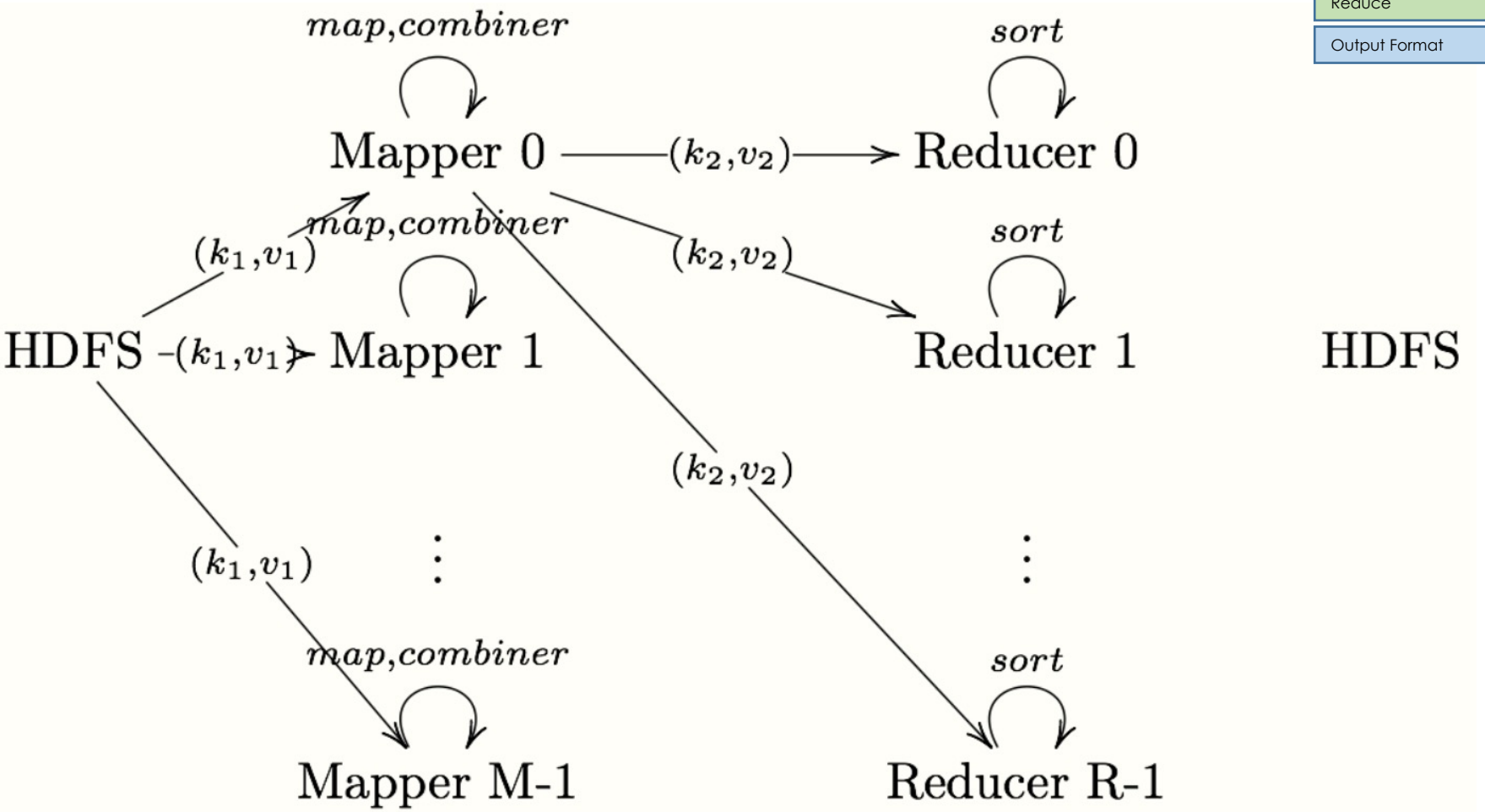
MR phases: Sort (5)



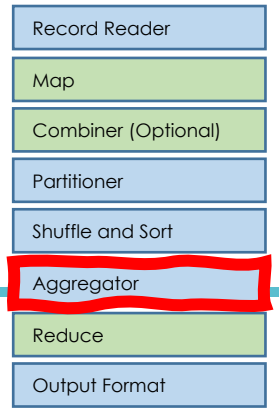
- **Shuffle** Effectively sends the intermediate pairs (k_2,v_2) to the reducer nodes
- **Sort** On reducer node, sorts by key to help group equivalent keys
- Ensures that all key-value pairs are grouped by their key on a single reducer
- A balancer system is in place for the cases when the key-values are too unevenly distributed
- In hadoop, the intermediate keys (k_2,v_2) are written to the local harddrive and grouped by which reduce they will be sent to and their key.

MR phases: Sort (5)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format

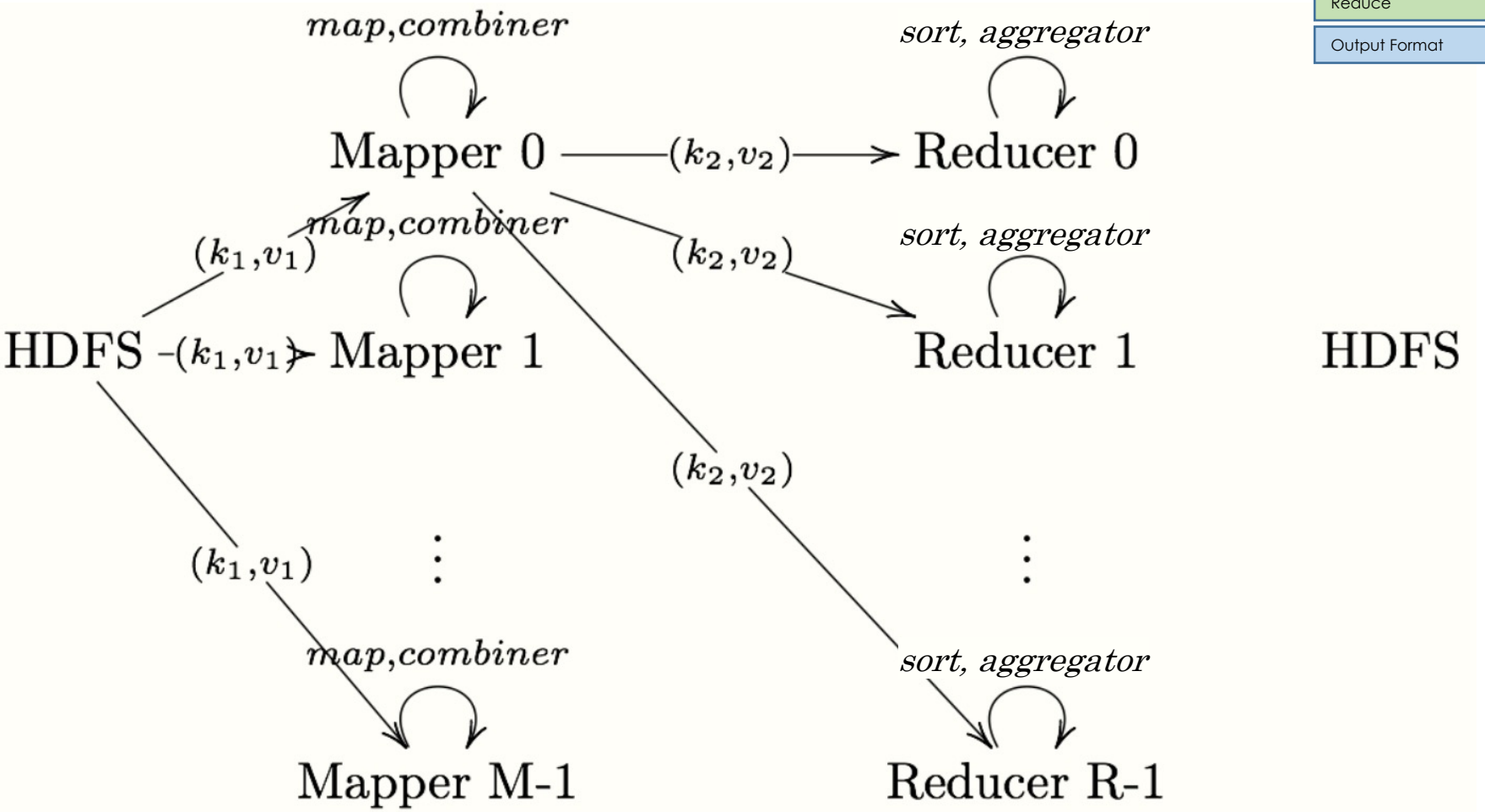


MR phases: Aggregator (6)



- **Aggregator** Ensures that all key-value pairs are grouped by their key on a single reducer
- A balancer system is in place for the cases when the key-values are too unevenly distributed

MR phases: Aggregator (6)



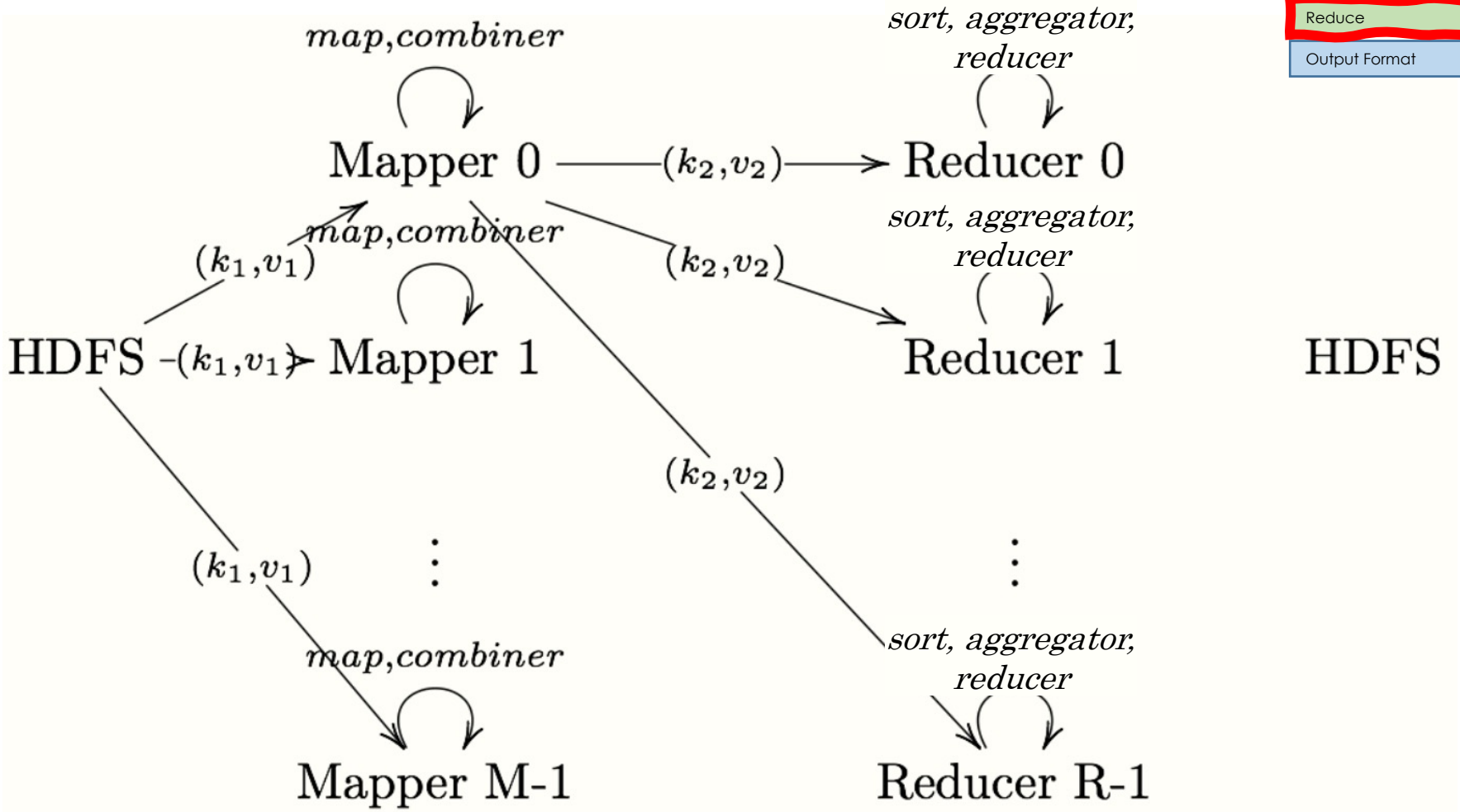
MR phases: Reduce (7)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format

- **Reduce** *User Defined Function* that aggregates data (v) according to keys (k), and emits key-value pairs to output

MR phases: Reduce (7)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format

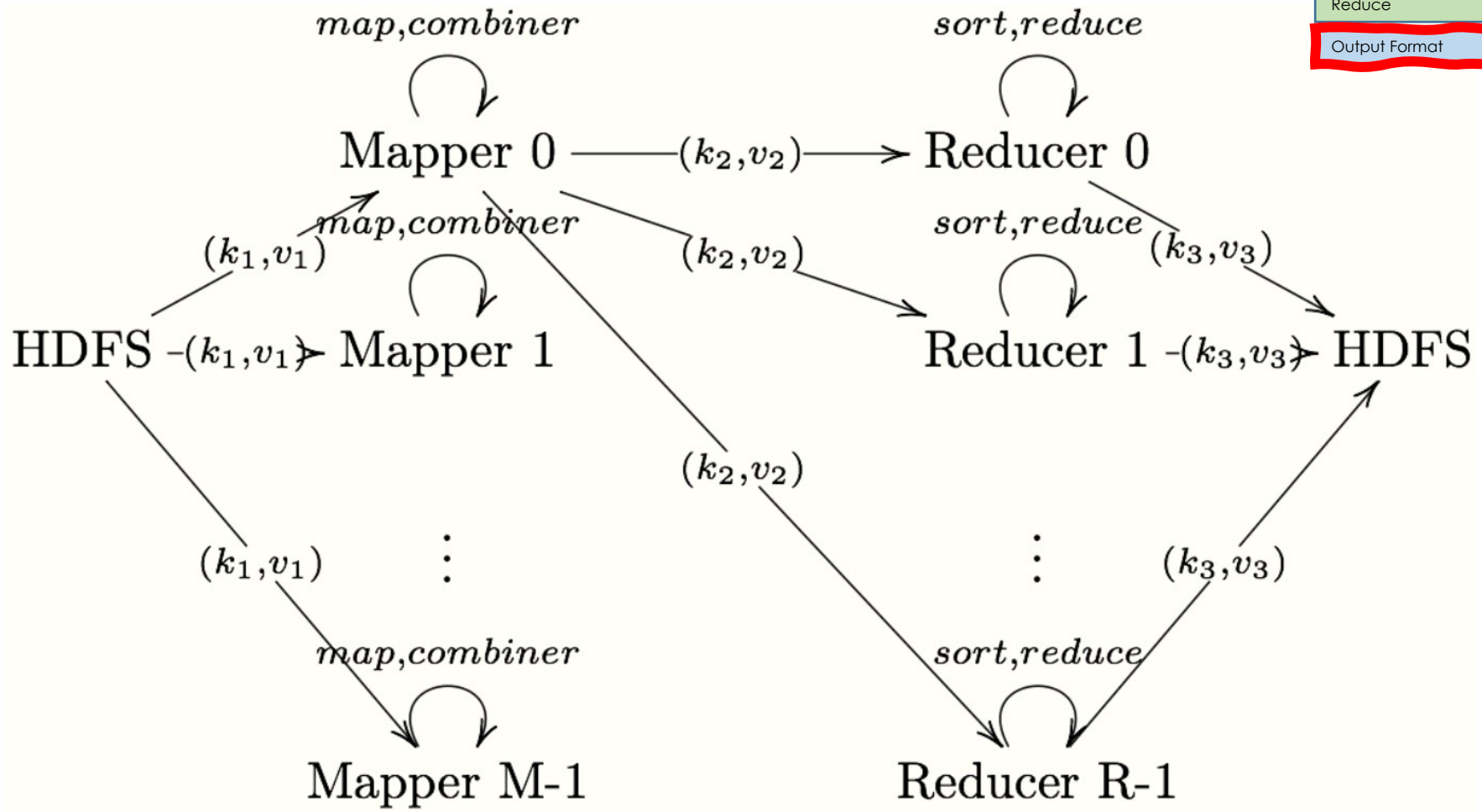


MR phases: Output Format (8)

- **Output Format** Translates final key-value pairs to file format (tab-separated by default)

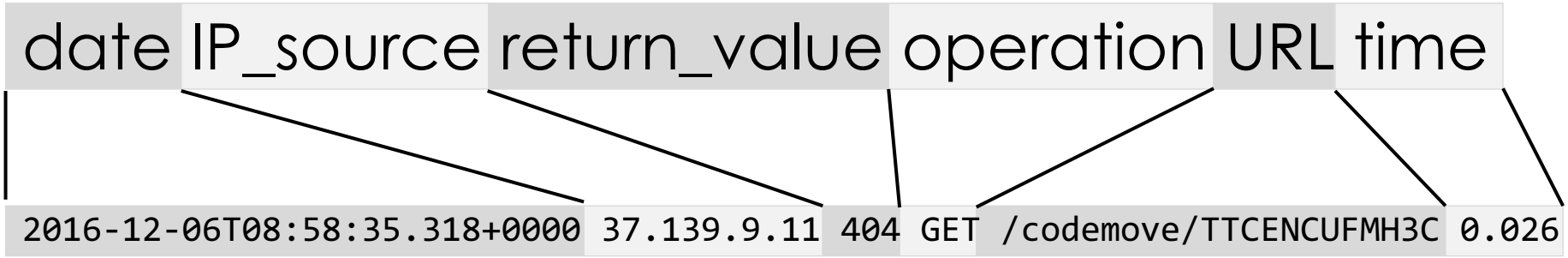
MR phases: Output Format (8)

Record Reader
Map
Combiner (Optional)
Partitioner
Shuffle and Sort
Aggregator
Reduce
Output Format



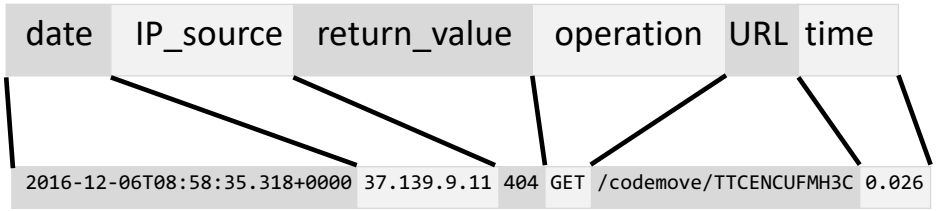
Programming map-reduce (0)

- For exercising map-reduce programming, consider a set of log files containing information for web accesses



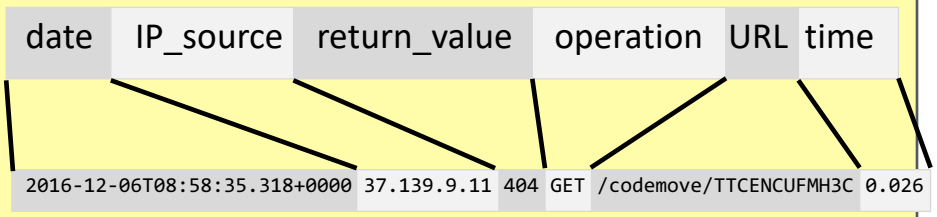
Programming map-reduce (1.1)

- Returns the list of source IP addresses
- **Input**: log files, containing information for web accesses
- **Output**: list of IP addresses



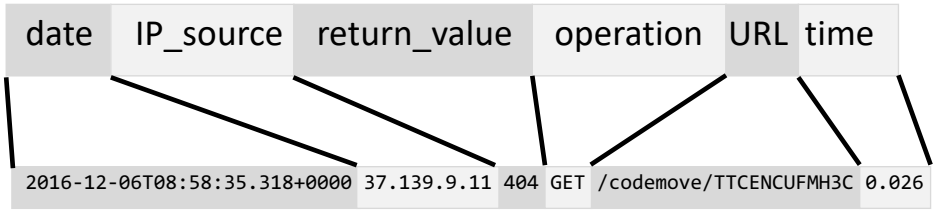
Programming map-reduce (1.2)

```
map(String key, String value):
  // key: log filename
  // value: log contents
  for each line l in value:
     words = line.split()
     EmitIntermediate(words[1], [words[1]]);

combiner(String key, Iterator values):
reduce(String key, Iterator values):
  // key: IP address
  // values: IP address
  Emit(key, key)
```



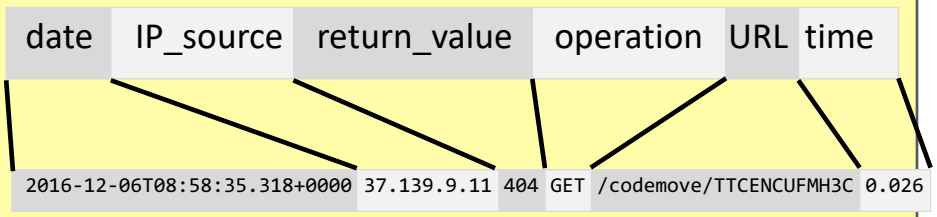
Programming map-reduce (2.1)

- Find log entries that report accesses to a given URL
- **Input**: log files
- **Output**: list of log entries (lines)



Programming map-reduce (2.2)

```
map(String key, String value):
  // key: log filename
  // value: log contents
  for each line l in value:
     words = line.split()
     if words[4] == URL:
        EmitIntermediate(words[0], [line]);

combiner(String key, Iterator values):
reduce(String key, Iterator values):
  // key: date
  // values: log line
  for each v in values:
     Emit(key, v)
```

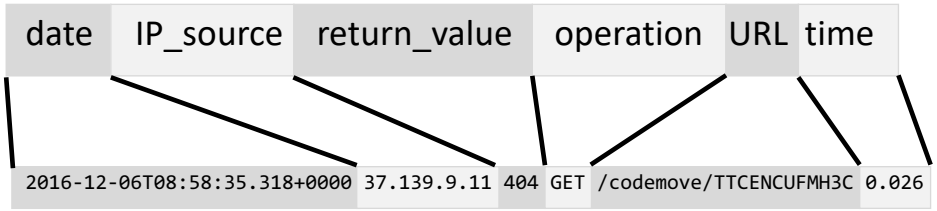


Programming map-reduce (3.1)

- Create an inverted index URL → list of unique source IP address

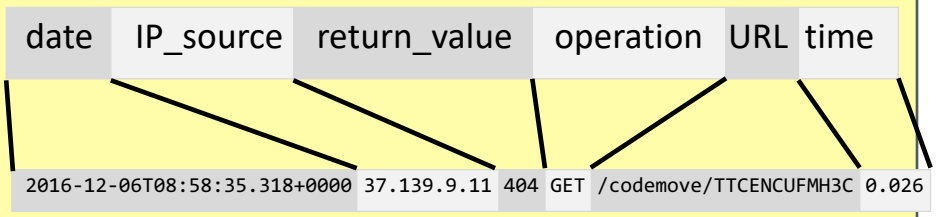
- **Input**: log files

- **Output**: list of: URL, list of unique source IP addresses



Programming map-reduce (3.2)

```
map(String key, String value):
  // key: log filename
  // value: log contents
  for each line l in value:
     words = line.split()
     EmitIntermediate(words[4], [words[1]]);

combiner(String key, Iterator values):
reduce(String key, Iterator values):
  // key: URL
  // values: list of IP addresses
  L = []
  for each v in values:
     if v not in L:
       L.append(v)
   Emit(key, L)
```



Why is map-reduce popular?

- Distributed computation before MapReduce:
  - how to divide the workload among multiple machines?
  - how to distribute data and program to other machines?
  - how to schedule tasks?
  - what happens if a task fails while running?
  - … and … and …
- Distributed computation after MapReduce
  - how to write Map function?
  - how to write Reduce function?

Ack: Slide by Junghoon Kang

Bibliography

- Jeffrey Dean and Sanjay Ghemawat. 2004. **MapReduce: simplified data processing on large clusters**. In Proc. OSDI'04. https://ai.google/research/pubs/pub62
  (*Available in CLIP as "mapreduce-osdi04.pdf"*)