Data Management for Data Science

Lecture 9: The MapReduce Programming Model and Algorithms in MapReduce

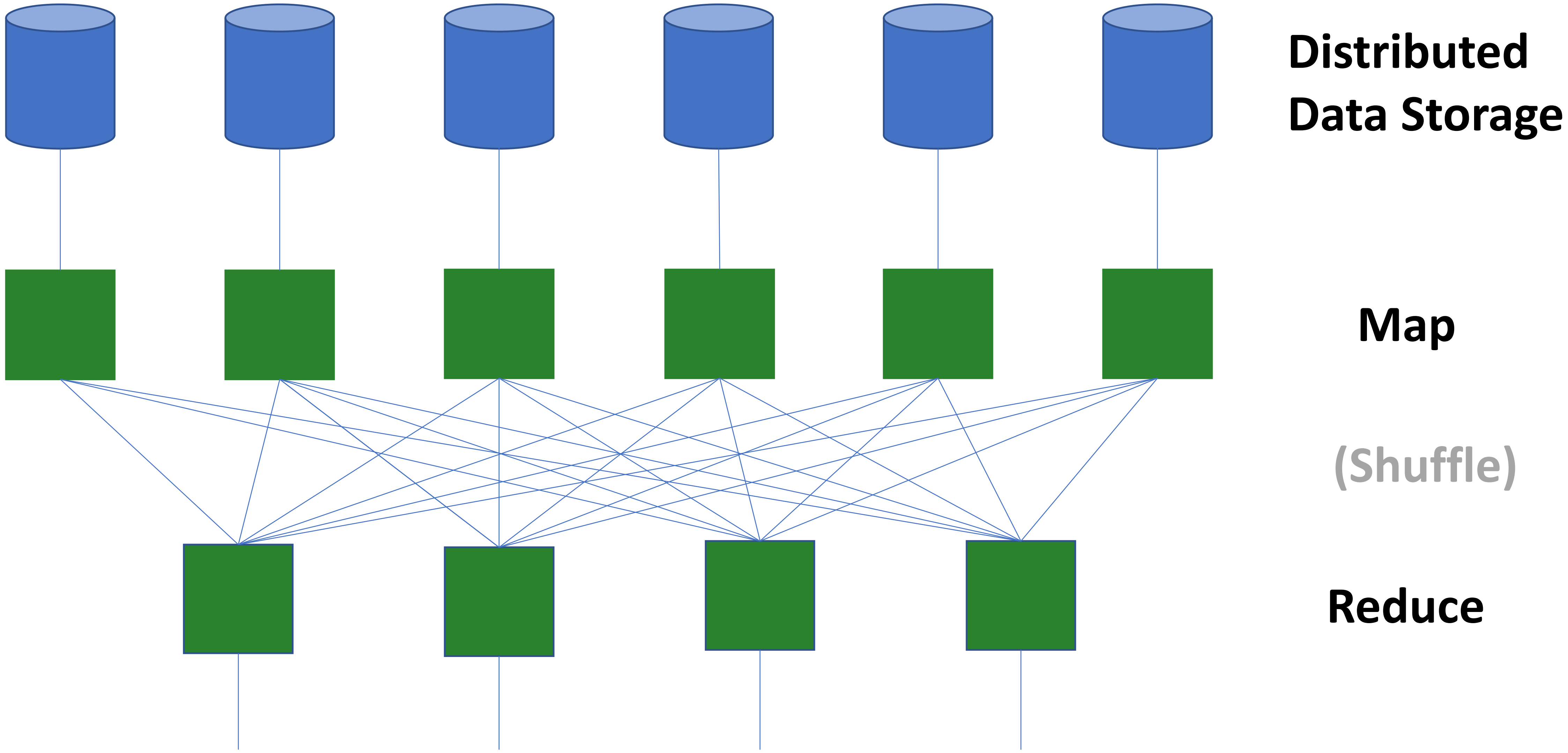
Prof. Asoc. Endri Raço

Today’s Lecture

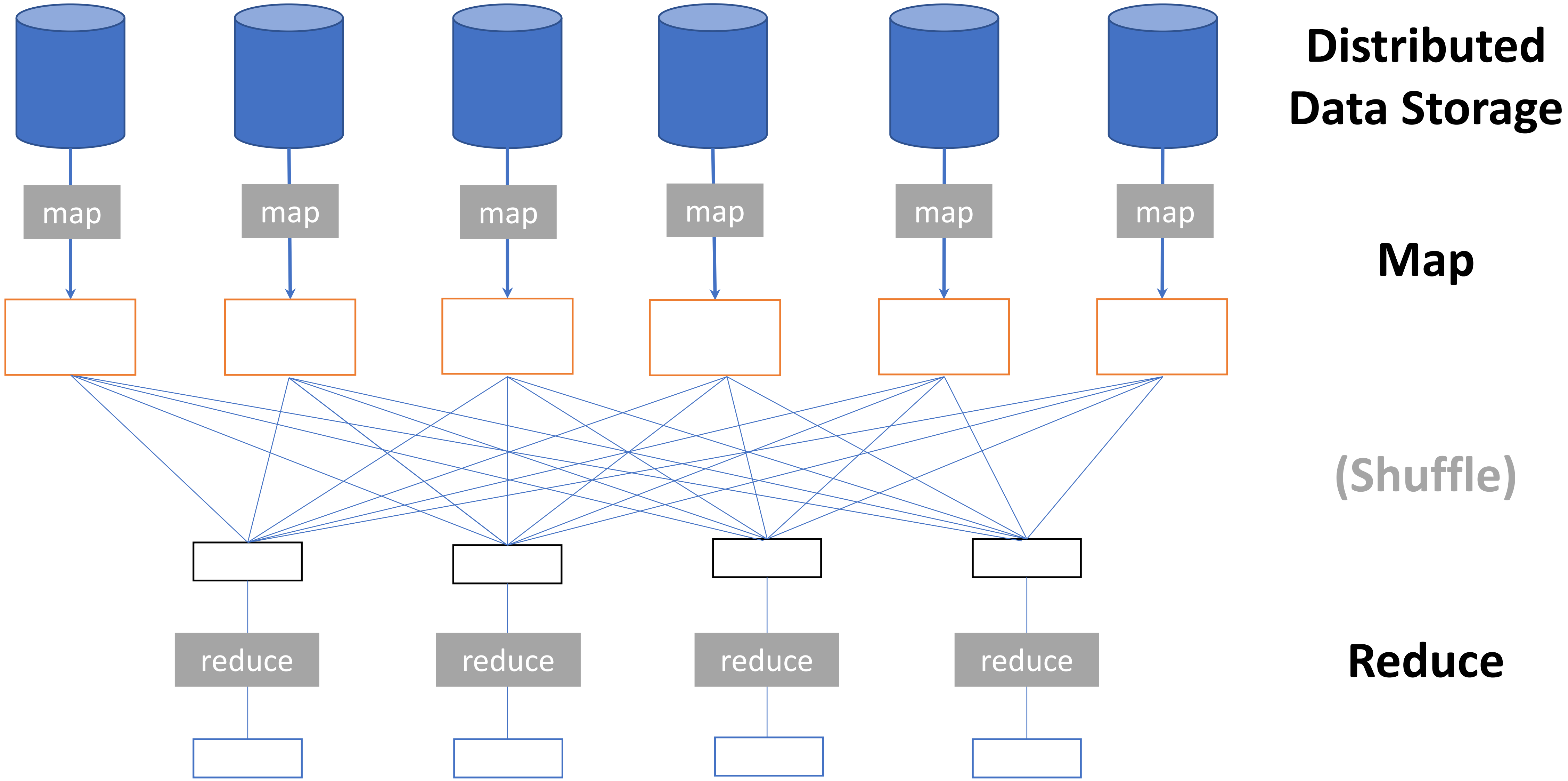
1. The MapReduce Abstraction
2. The MapReduce Programming Model
3. MapReduce Examples

1. The MapReduce Abstraction

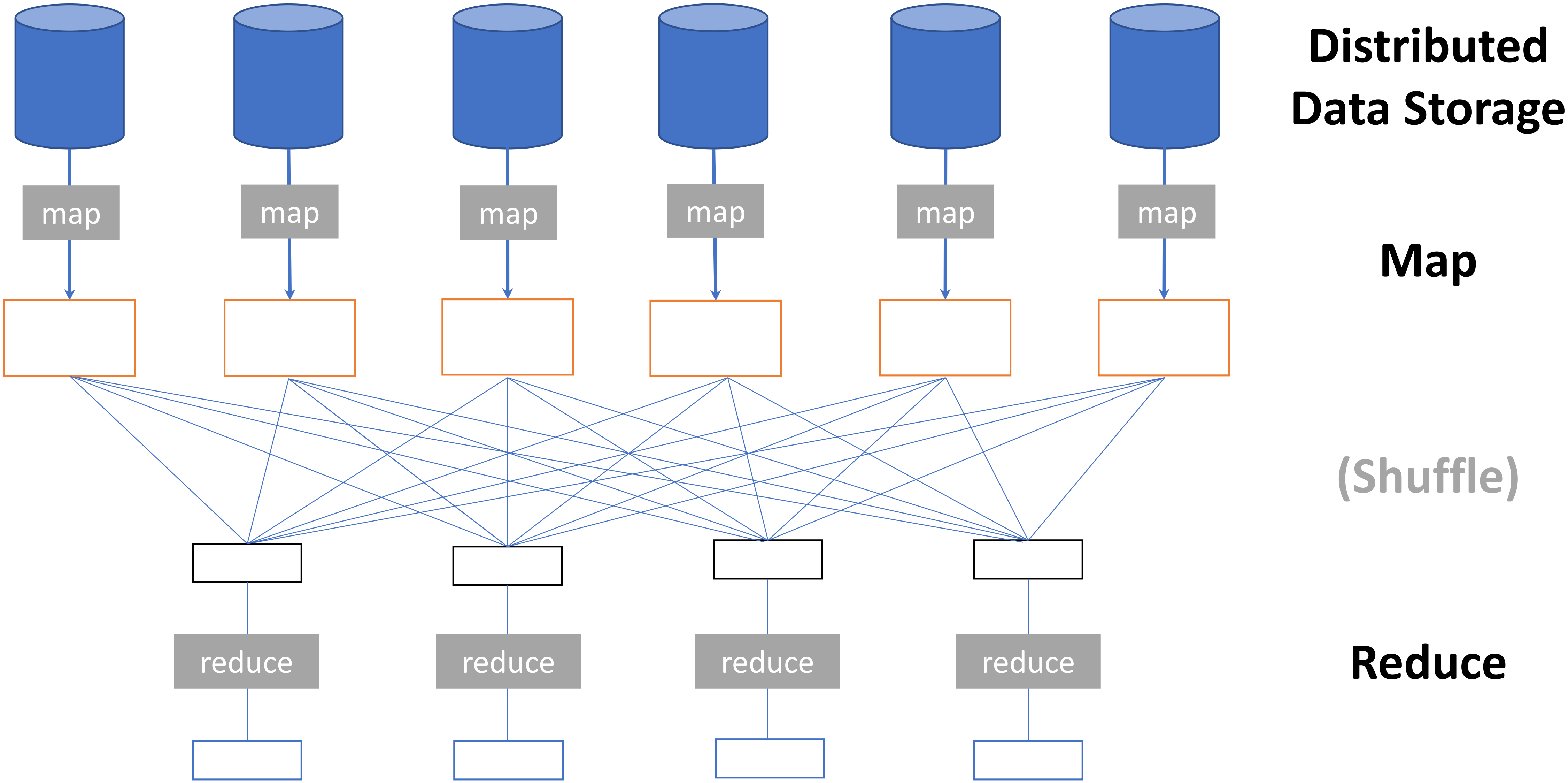
The Map Reduce Abstraction for Distributed Algorithms

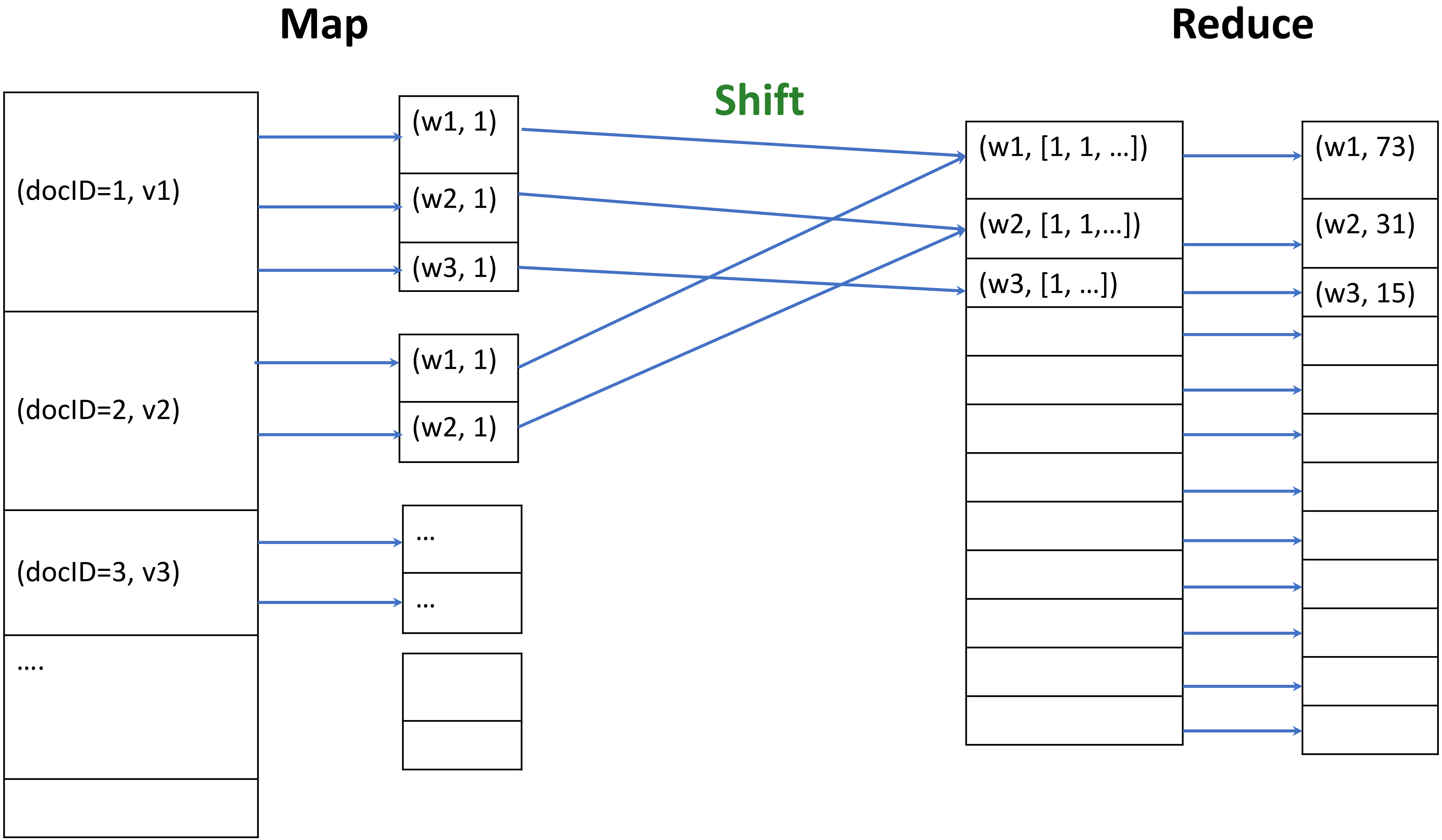


The Map Reduce Abstraction for Distributed Algorithms



The Map Reduce Abstraction for Distributed Algorithms




Map
Reduce
Shift
(docID=1, v1)
(docID=2, v2)
(docID=3, v3)
….
(w1, 1)
(w2, 1)
(w3, 1)
(w1, 1)
(w2, 1)
…
…
(w1, [1, 1, …])
(w2, [1, 1,…])
(w3, [1, …])
(w1, 73)
(w2, 31)
(w3, 15)

The Map Reduce Abstraction for Distributed Algorithms

- MapReduce is a high-level programming model and implementation for large-scale parallel data processing

- Like RDBMS adopt the the relational data model, MapReduce has a data model as well

MapReduce's Data Model

- Files!
- A File is a bag of **(key, value)** pairs
    - A bag is a **multiset**
- A map-reduce program:
    - Input: a bag of **(inputkey, value)** pairs
    - Output: a bag of **(outputkey, value)** pairs

2. The MapReduce Programming Model

User input

- All the user needs to define are the MAP and REDUCE functions

- Execute proceeds in multiple MAP – REDUCE rounds
  - MAP – REDUCE = MAP phase followed REDUCE

MAP Phase

Step 1: the MAP phase

- User provides a MAP-function:
  - Input: **(input key, value)**
  - Output: bag of **(intermediate key, value)**

- System applies the map function in parallel to all (input key, value) pairs in the input file

REDUCE Phase

Step 2: the REDUCE phase

- User provides a REDUCE-function:
  - Input: **(intermediate key, bag of values)**
  - Output: **(intermediate key, values)**

- The system will group all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

MapReduce Programming Model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

map (in_key, in_value) -> list(out_key, intermediate_value)

- Processes input key/value pair
- Produces set of intermediate pairs

reduce (out_key, list(intermediate_value)) -> (out_key, list(out_values))

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usually just one)

Example: what does the next program do?

```
map(String input_key, String input_value):
  //input_key: document id
  //input_value: document bag of words
  for each word w in input_value:
      EmitIntermediate(w, 1);

reduce(String intermediate_key, Iterator intermediate_values):
  //intermediate_key: word
  //intermediate_values: ????
  result = 0;
  for each v in intermediate_values:
    result += v;
  EmitFinal(intermediate_key, result);
```

Example: what does the next program do?

```
map(String input_key, String input_value):
  //input_key: document id
  //input_value: document bag of words
  word_count = {}
  for each word w in input_value:
      increase word_count[w] by one
  for each word w in word_count:
      EmitIntermediate(w, word_count[w]);

reduce(String intermediate_key, Iterator intermediate_values):
  //intermediate_key: word
  //intermediate_values: ????
  result = 0;
  for each v in intermediate_values:
    result += v;
  EmitFinal(intermediate_key, result);
```

3. MapReduce Examples

Word length histogram

How many big, medium, small, and tiny words are in a document?

Big = 10+ letters
Medium = 5..9 letters
Small = 2..4 letters
Tiny = 1 letter

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

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

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

Word length histogram

Split the document into chunks and process each chunk on a different computer

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Chunk 2

Abstract

Chunk 1

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

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

Word length histogram

Map Chunk 1
(204words)

Output
(Big , 17)
(Medium, 77)
(Small, 107)
(Tiny, 3)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Chunk 1

Chunk 2

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

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

Word length histogram

Map task 1

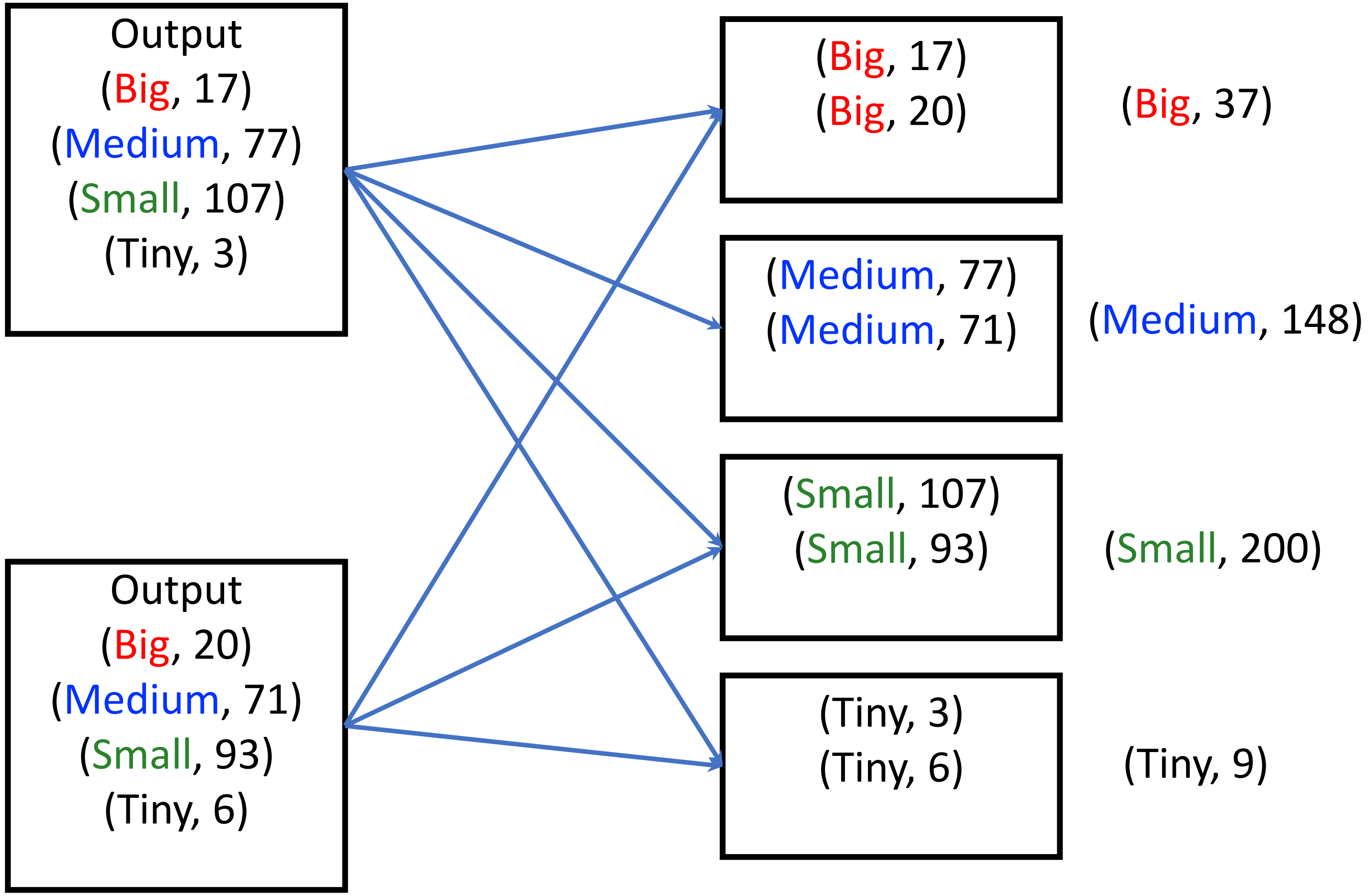
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with 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

Map task 2

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



Build an Inverted Index

Input:	Output:
doc1, (“I love medium roast coffee”)	“roast”, (doc1)
doc2, (“I do not like coffee”)	“coffee”, (doc1, doc2)
doc3, (“This medium well steak is great”)	“medium”, (doc1, doc3)
doc4, (“I love steak”)	“steak”, (doc3, do4)

Let’s design the solution!

Input:	Output:
doc1, (“I love medium roast coffee”)	“roast”, (doc1)
doc2, (“I do not like coffee”)	“coffee”, (doc1, doc2)
doc3, (“This medium well steak is great”)	“medium”, (doc1, doc3)
doc4, (“I love steak”)	“steak”, (doc3, do4)