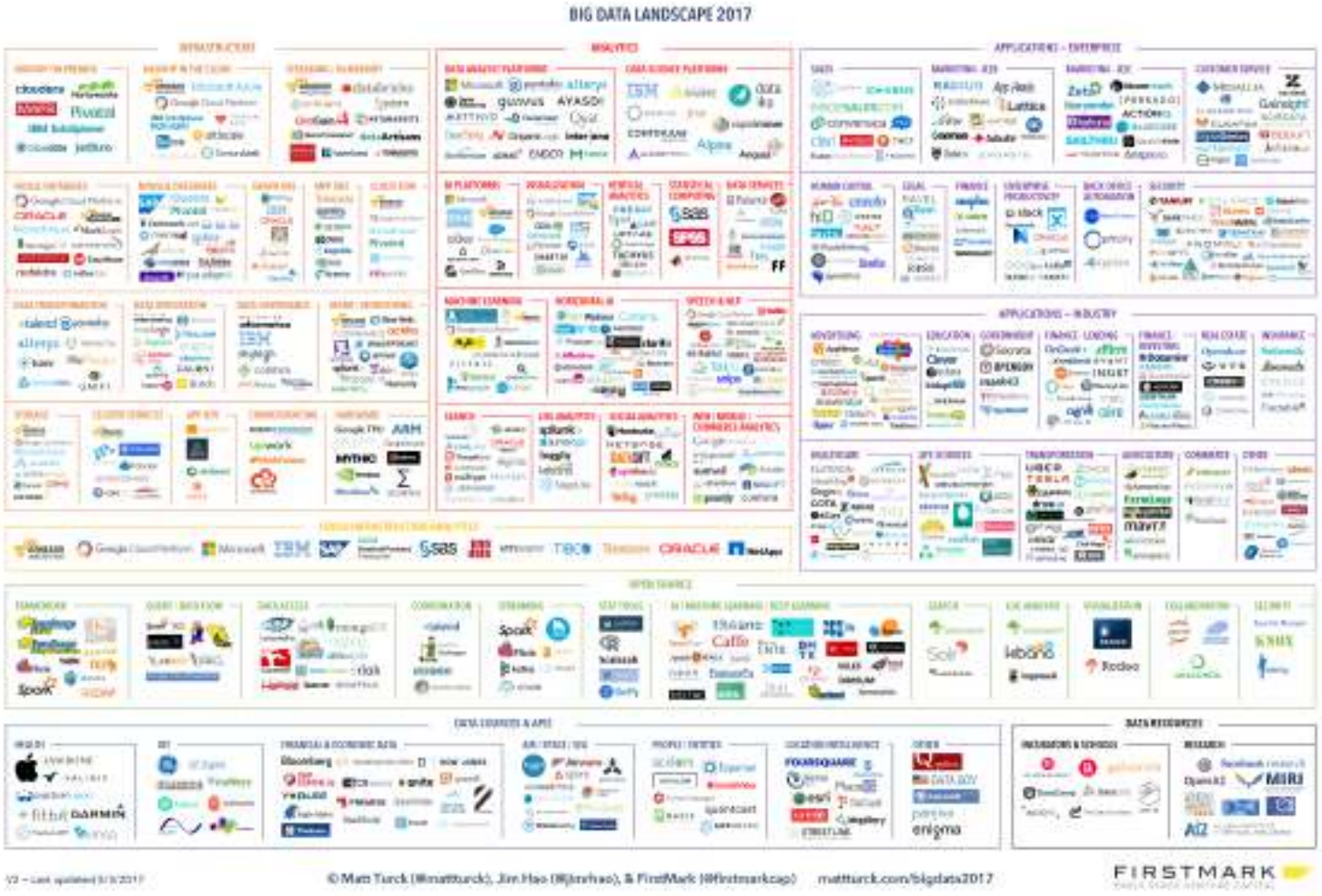
Cloud Computing et informatique distribuée
Cours : Map Reduce

Francesca Bugiotti

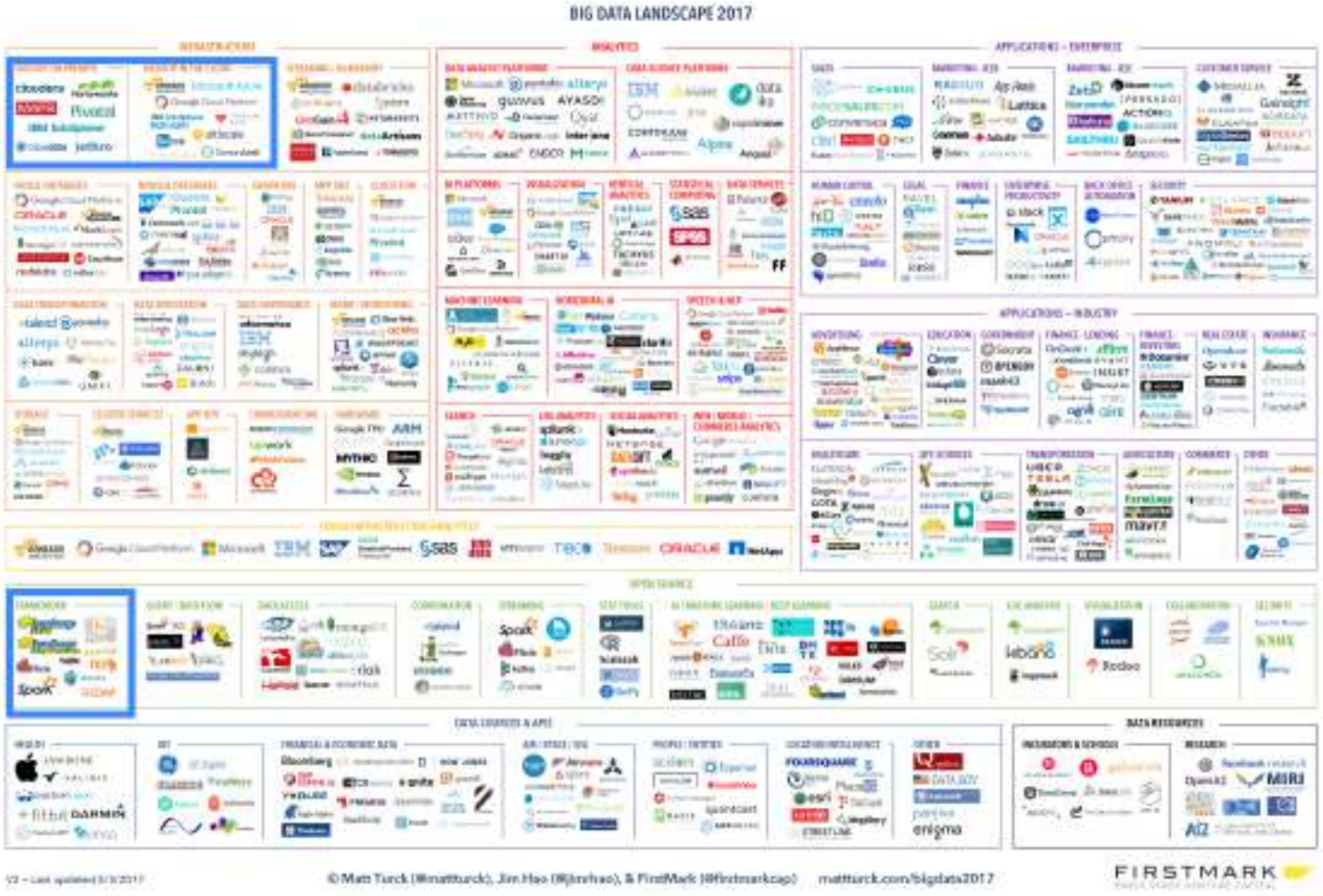
CentraleSupélec

May 7, 2020

Technologies



Technologies



Objectives

- The MapReduce programming model

File System

A File System defines structure and logic rules used to manage the groups of information and their names in a storage device

Long-term information storage

- Providing access for processes
- Storing large amount of information

File System

You have your 2TB disk:

If the space is not sufficient?

- A bigger disk

- An external hard drive

Issues: copy data, transfer data, maintain data, etc.

Many devices implies many disks

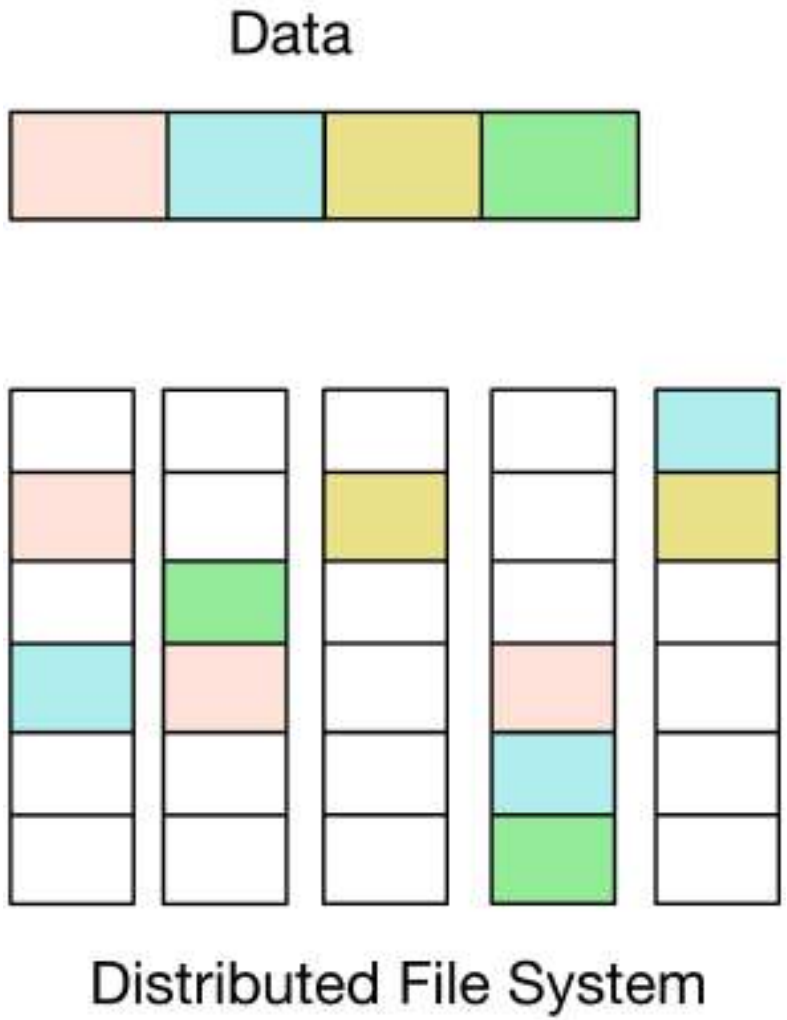
- From two PC

- To many heterogeneous devices

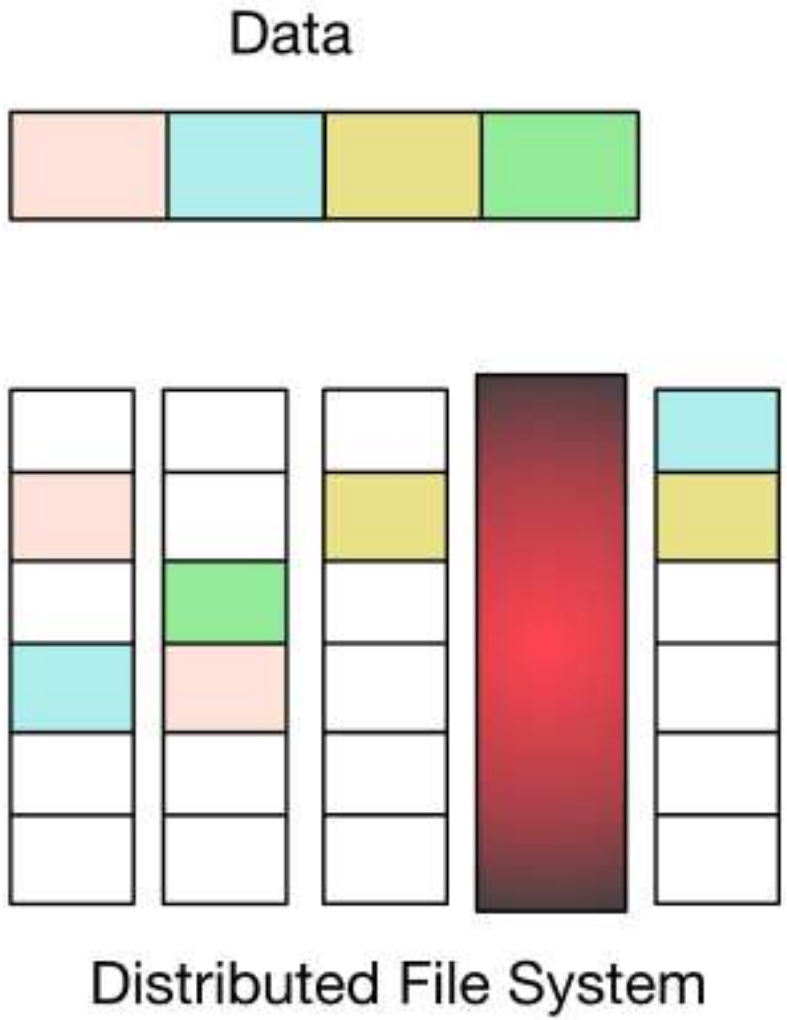
Distributed File System

- Data distributed across the cluster
- Data possibly replicated for fault tolerance

Data Distribution



Data Distribution



Fault tolerance

Fault tolerance: the ability to restart the computation given a failure

- Redundant data storage
- Restart of failed individual parallel jobs

Restart failed jobs

Suppose that we have a set of jobs that collaborate in order to solve a problem and belong to a main process

What can be done if one (or more) fails?

- Start all the process from the beginning
- Record checkpoint information so that you can restart just some jobs if they fail
- Make the jobs independent and restart just the failed jobs

Big Data

Big Data processing includes the features

- Multiple data types
- Large datasets

Big Data

Large volumes of data: splitting data

- Partitioning and placement of data in and out of computer memory
- Model to synchronize the datasets
- Replications and recovery of files

Large volumes of data: access data

- Fast data access
- Move the computation to data
- Scheduling of many parallel tasks
- Support data variety

Programming model

A programming model for Big Data should handle all these features

- Splitting large volumes of data
- Abstraction over a distributed file system
  - Handle replications
  - Synchronize datasets
- Schedule parallel tasks
- Recover from failures
- Handle different data types
- Move the computation near the data

Starting from Pasta

You come back from work on a Friday evening

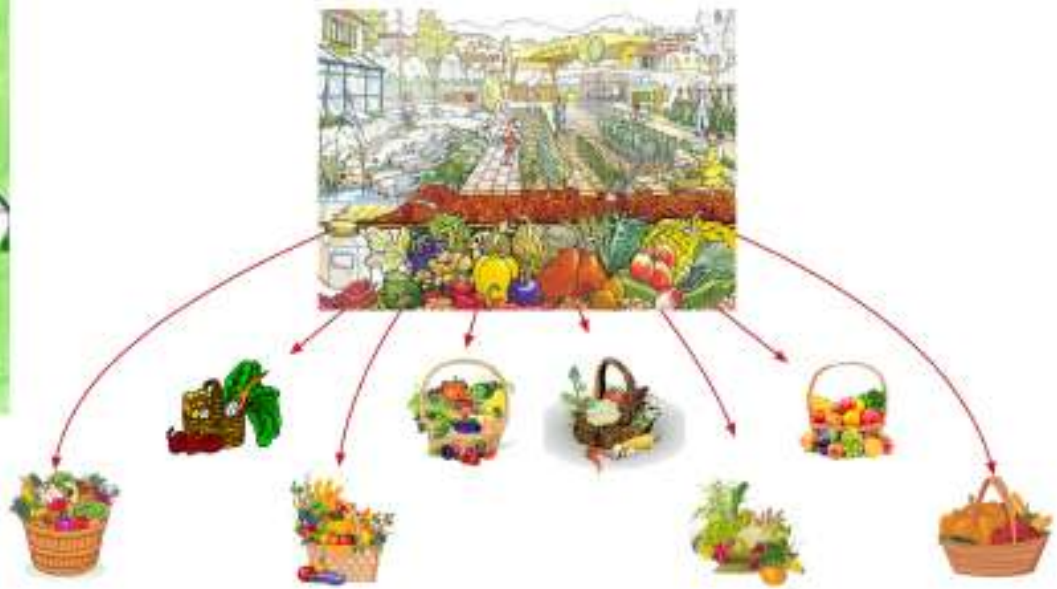
You realize that you have forgotten that you have organized a party at your place that evening

What can be done?

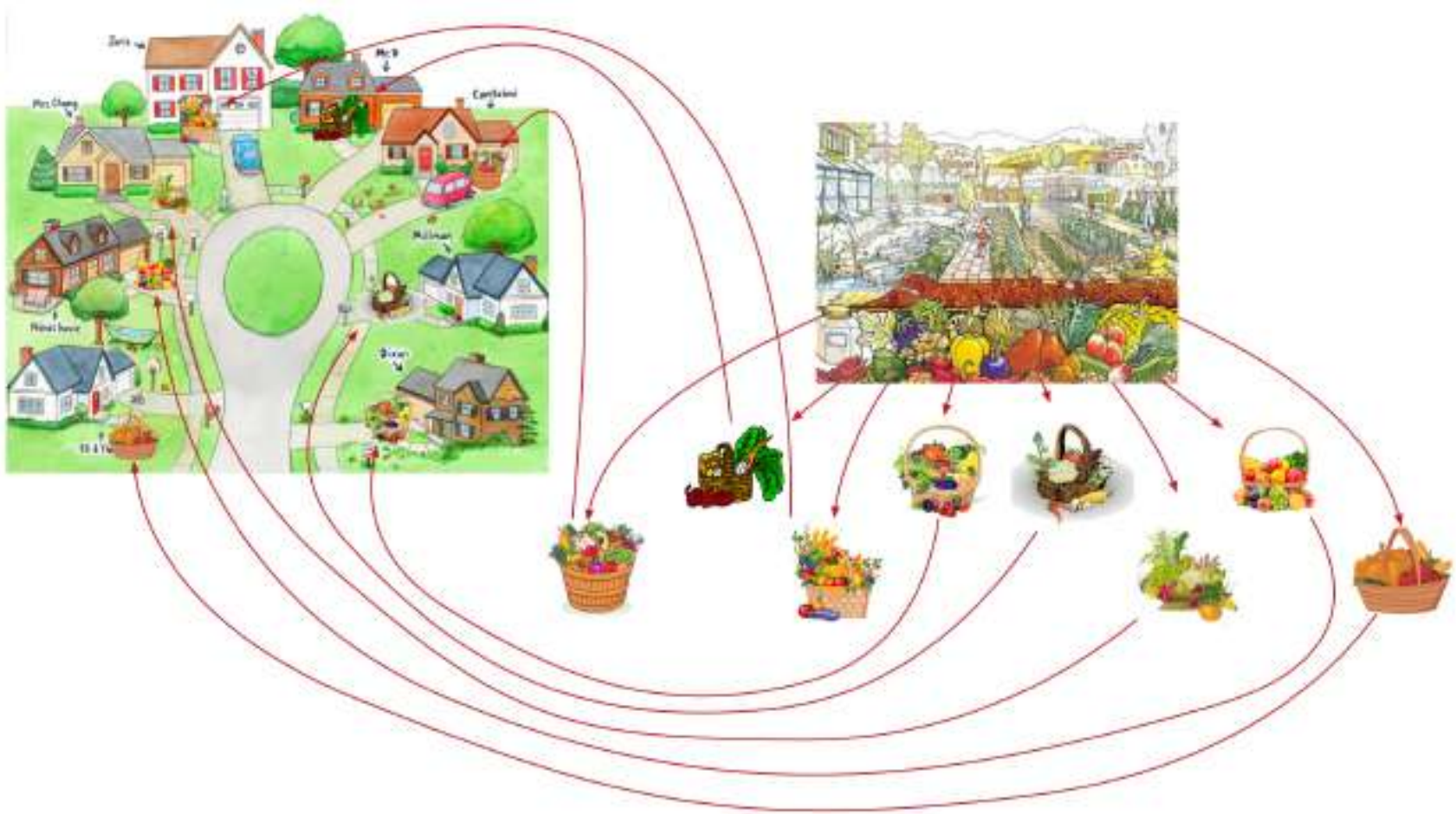
Italian solution: prepare a pasta party!

Pasta Party

Pasta Party



Pasta Party



Pasta Party

Pasta Party

Pasta Party

Pasta Party

Pasta Party

Pasta Party

A Programming model: MapReduce

- Splitting large volumes of data
- Abstraction over a distributed file system
  - Handle replications
  - Synchronize datasets
- Schedule parallel tasks
- Recover from failures
- Handle different data types
- Move the computation near the data

Implemented by a large number of frameworks including Hadoop

Cloud Computing et informatique distribuée
Cours : Map Reduce

Francesca Bugiotti

CentraleSupélec

May 6, 2020

Objectives

- The MapReduce programming model

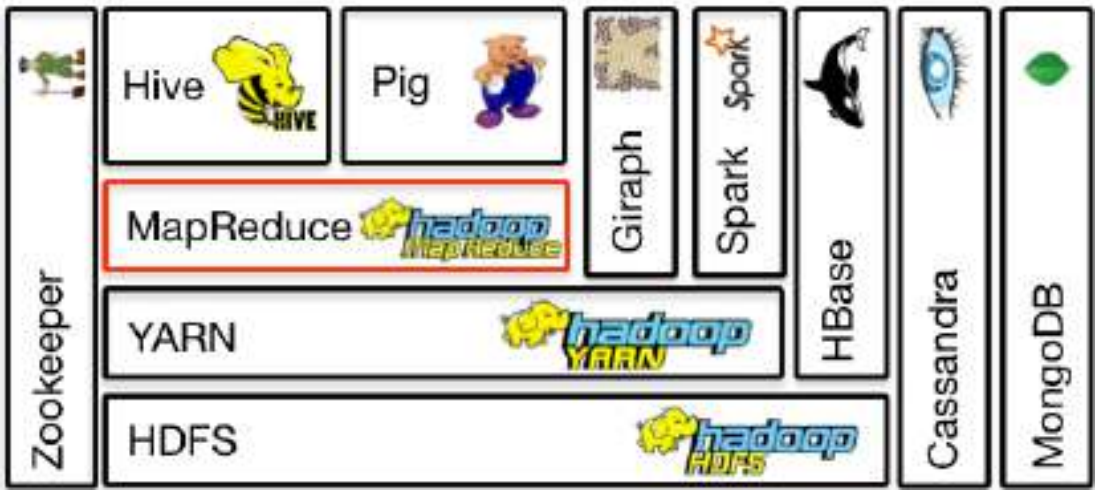
Pasta Party

A Programming model: MapReduce

- Splitting large volumes of data
- Abstraction over a distributed file system
  - Handle replications
  - Synchronize datasets
- Schedule parallel tasks
- Recover from failures
- Handle different data types
- Move the computation near the data

Implemented by a large number of frameworks including Hadoop

MapReduce



Hadoop MapReduce [1]

With MapReduce you must create: map tasks, reduce tasks and run them

Past

Traditional parallel programming

- Locks
- Semaphores
- Monitors

An incorrect usage:

- Incorrect programs
- Performance impact

Impact

Map

takes as input an object with a key (k,v) and returns a bunch of key-value pairs:

$$(k_1,v1),(k_2,v_2), \ldots ,(k_n,v_n)$$

Shuffle

The framework collects all the pairs with the same key k and associates with k all the values for k:

$$(k,[v_1, \ldots ,vn])$$

Reduce

takes as input a key and a list of values $(k,[v_1, \ldots ,v_n])$ and combine them somehow

MapReduce programming

Programmers specify two functions:

- map $(k_1, v_1) \rightarrow [(k_2, v_2)]$
- reduce $(k_1, [v_1]) \rightarrow [(k_2, v_2)]$

where (k, v) denotes a (key, value) pair and [...] denotes a list

MapReduce program

A MapReduce program, referred to as a job, consists of:

- Code for Map and a code for Reduce packaged together
- Configuration parameters (where the input lies, where the output should be stored)
- The input, stored on the underlying distributed file system
- Each MapReduce job is divided by the system into smaller units called tasks
  - Map tasks
  - Reduce tasks
- Input and output of MapReduce jobs are stored on the underlying distributed file system

MapReduce execution process I

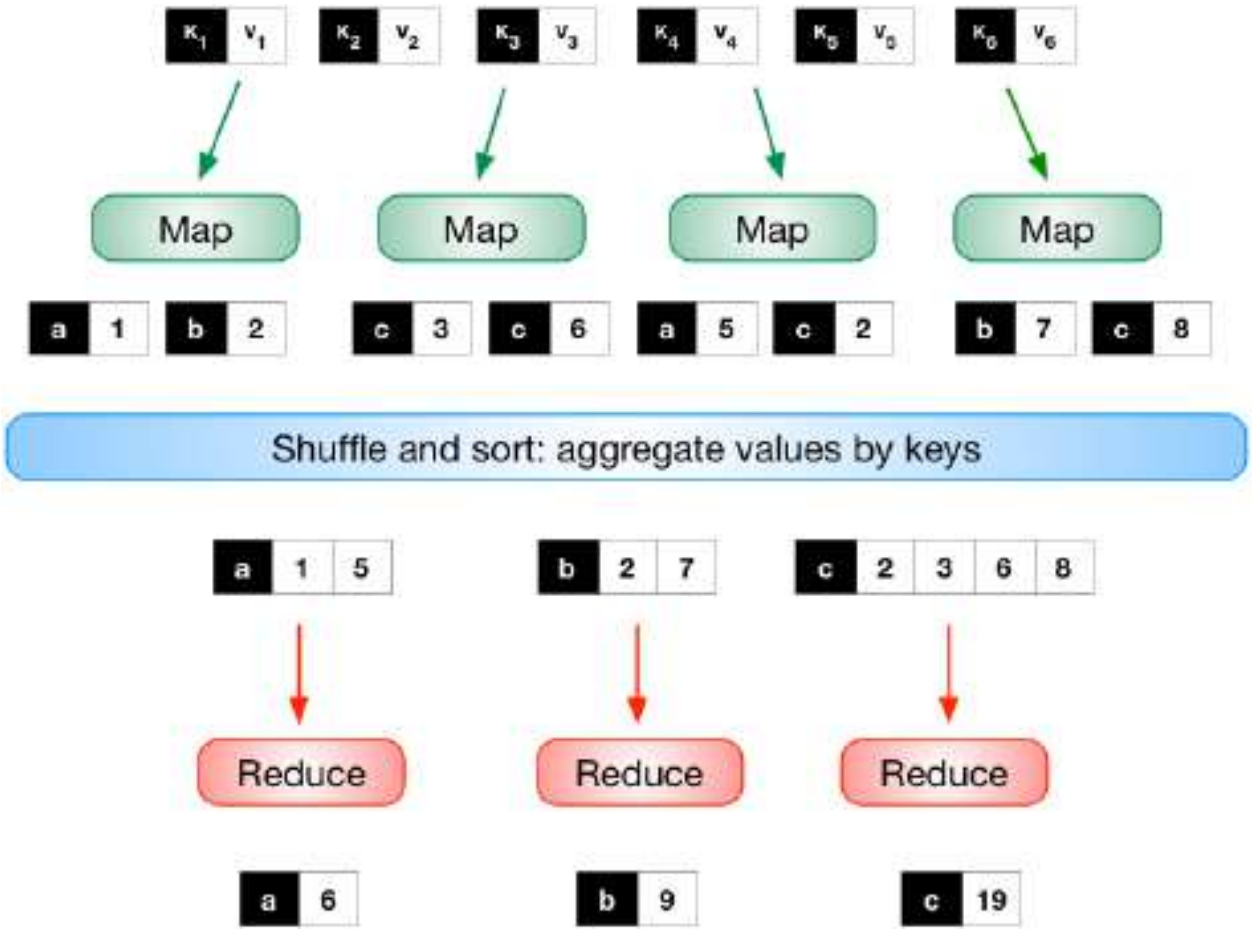
1. Some Map tasks are given each one or more chunks of data
2. Each Map task turns the chunk into a sequence of key-value pairs
    - The way key-value pairs are produced is determined by the code written by the user for the Map function
3. The key-value pairs from each Map task are collected by a master controller and sorted and grouped by key (Shuffle and Sort)
4. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
5. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way

MapReduce execution process II

- The way values are combined is determined by the code written by the user for the Reduce function

6. Output: key-value pairs from each reducer are written persistently back onto the distributed file system
7. The output ends up in r files, where r is the number of reducers
   - The r files often serve as input to yet another MapReduce job

MapReduce


Map
Map
Map
Map
Shuffle and sort: aggregate values by keys
Reduce
Reduce
Reduce

Counting words I

Problem:

counting the number of occurrences of each word in a collection of documents

- Input: a repository of documents, each document is an element
- Map: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

$$(w_1, 1), (w_2, 1), \ldots , (w_n, 1)$$

- Shuffle: groups by key and generates pairs of the form

$$(w_1, [1, 1, \ldots , 1]) , \ldots , (w_n, [1, 1, \ldots , 1])$$

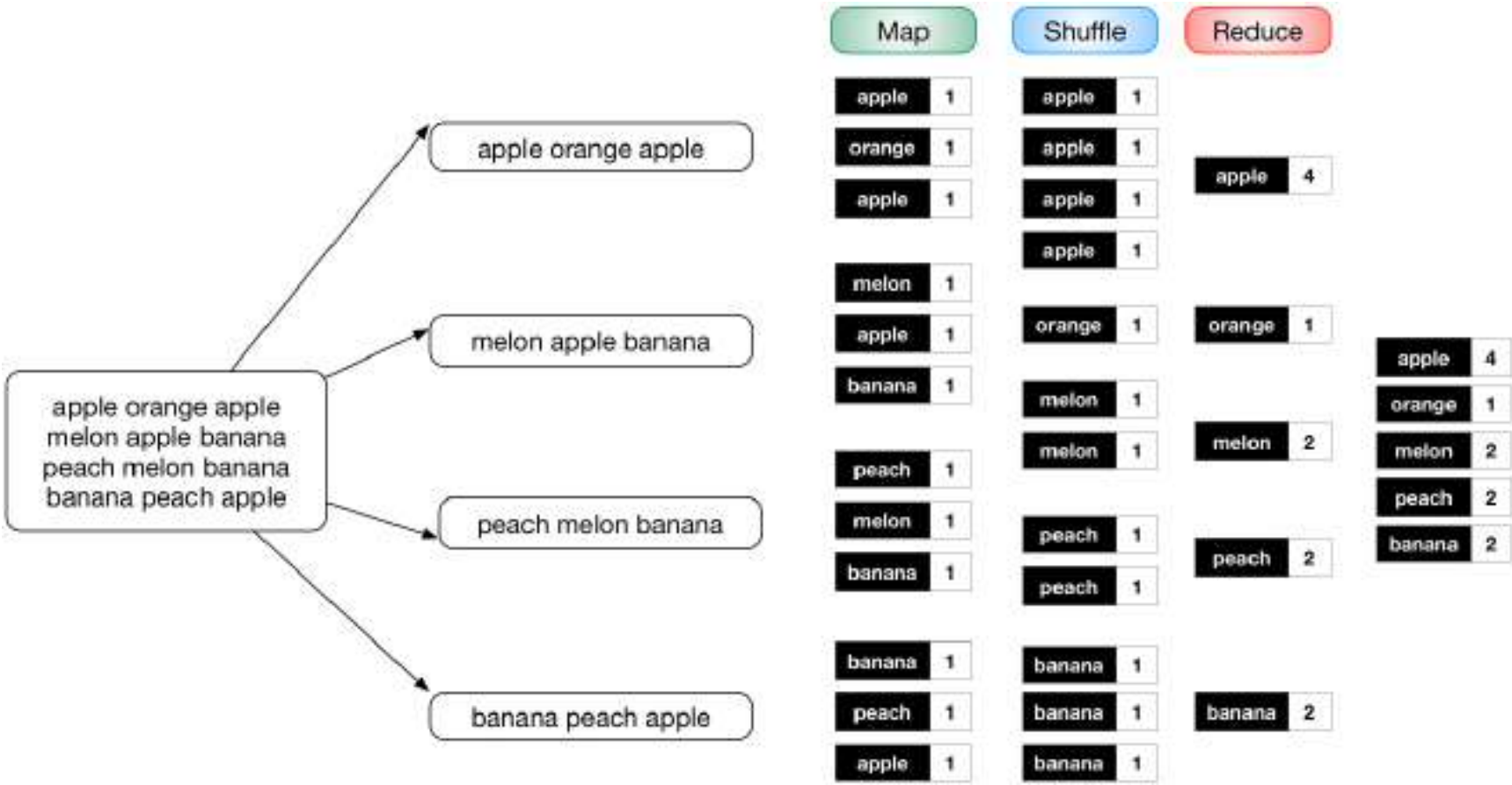
Counting words II

- Reduce: adds up all the values and emits:

$$(w_1, k) , \ldots , (w_n, l)$$

- Output: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

MapReduce


Map
Shuffle
Reduce
apple orange apple
melon apple banana
peach melon banana
banana peach apple
apple orange apple
melon apple banana
peach melon banana
banana peach apple
apple 1
orange 1
apple 1
melon 1
apple 1
banana 1
peach 1
melon 1
banana 1
banana 1
peach 1
apple 1
apple 1
apple 1
apple 1
apple 1
orange 1
melon 1
melon 1
peach 1
peach 1
banana 1
banana 1
banana 1
apple 4
orange 1
melon 2
peach 2
banana 2
apple 4
orange 1
melon 2
peach 2
banana 2

Implementation - pseudocode

```
Map(String docID, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, counts[]):
   int sum = 0;
   for each c in counts:
       sum += c;
   Emit(term, sum);
```

Implementation Python

```
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

Implementation Python

```
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (currentword, currentcount)
        currentcount = count
        currentword = word
if currentword == word:
    print '%s\t%s' % (currentword, currentcount)
```

References I


Hadoop Map Reduce.
http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.
Accessed 2016.

Cloud Computing et informatique distribuée
Cours : Map Reduce

Francesca Bugiotti

CentraleSupélec

May 6, 2020

Counting words I

Problem:

counting the number of occurrences of each word in a collection of documents

- Input: a repository of documents, each document is an element
- Map: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

$$(w_1, 1), (w_2, 1), \ldots , (w_n, 1)$$

- Shuffle: groups by key and generates pairs of the form

$$(w_1, [1, 1, \ldots , 1]) , \ldots , (w_n, [1, 1, \ldots , 1])$$

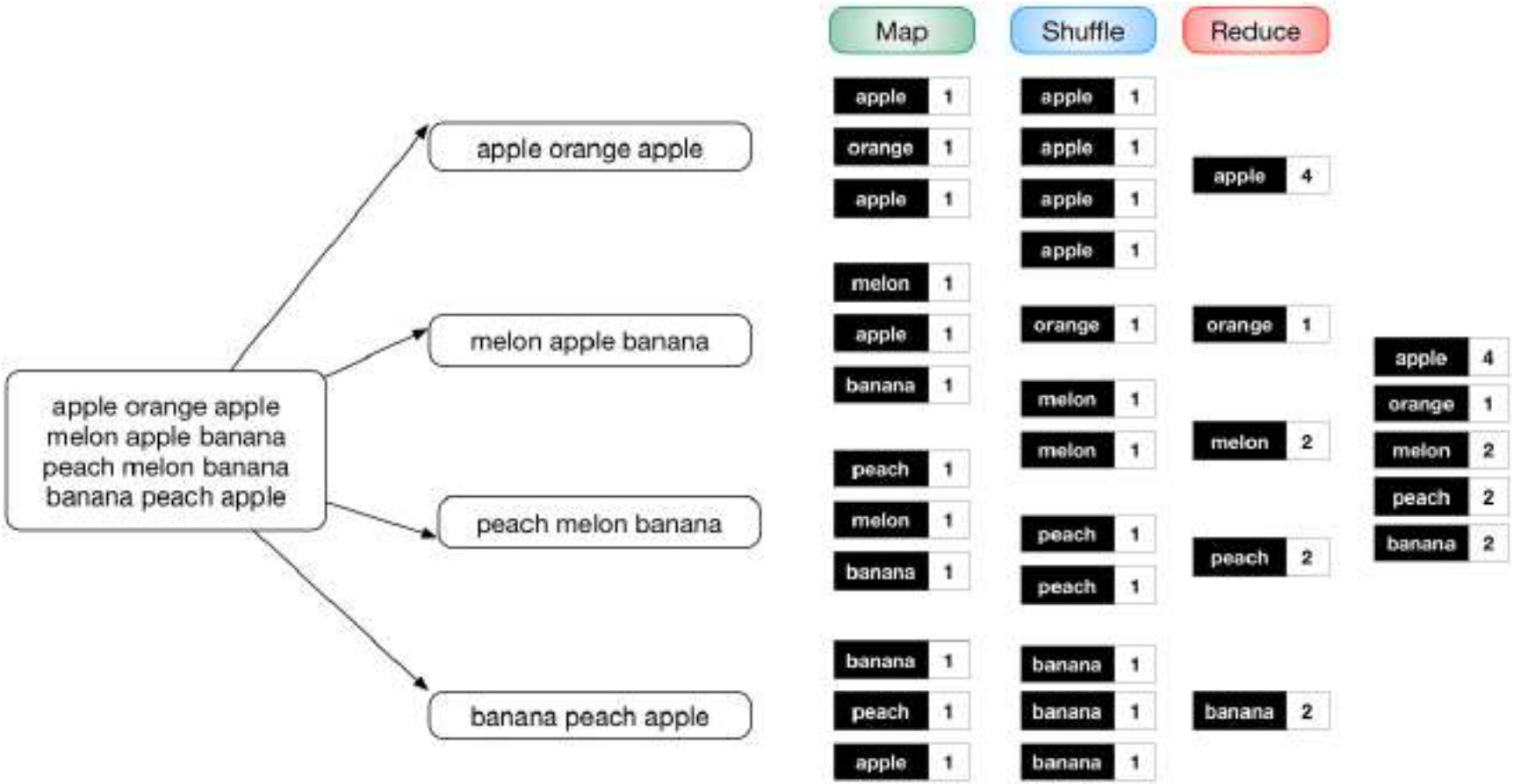
Counting words II

- Reduce: adds up all the values and emits:

$$(w_1, k) , \ldots , (w_n, l)$$

- Output: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

MapReduce


Map
Shuffle
Reduce
apple orange apple
melon apple banana
peach melon banana
banana peach apple
apple orange apple melon apple banana peach melon banana banana peach apple
apple 1
orange 1
apple 1
melon 1
apple 1
banana 1
peach 1
melon 1
banana 1
banana 1
peach 1
apple 1
apple 1
apple 1
apple 1
apple 1
orange 1
melon 1
melon 1
peach 1
peach 1
banana 1
banana 1
banana 1
apple 4
orange 1
melon 2
peach 2
banana 2
apple 4
orange 1
melon 2
peach 2
banana 2

Implementation - pseudocode

```
Map(String docID, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, counts[]):
   int sum = 0;
   for each c in counts:
       sum += c;
   Emit(term, sum);
```

Implementation Python

```
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

Implementation Python

```
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (currentword, currentcount)
        currentcount = count
        currentword = word
if currentword == word:
    print '%s\t%s' % (currentword, currentcount)
```

Counting words I

Problem:

Counting how many times a word appears in the documents

- q1: is it necessary to emit a key-value?
- q2: what about the reduce?
- q3: the reduce operation runs on how many nodes of a distributed environment?

Combiners

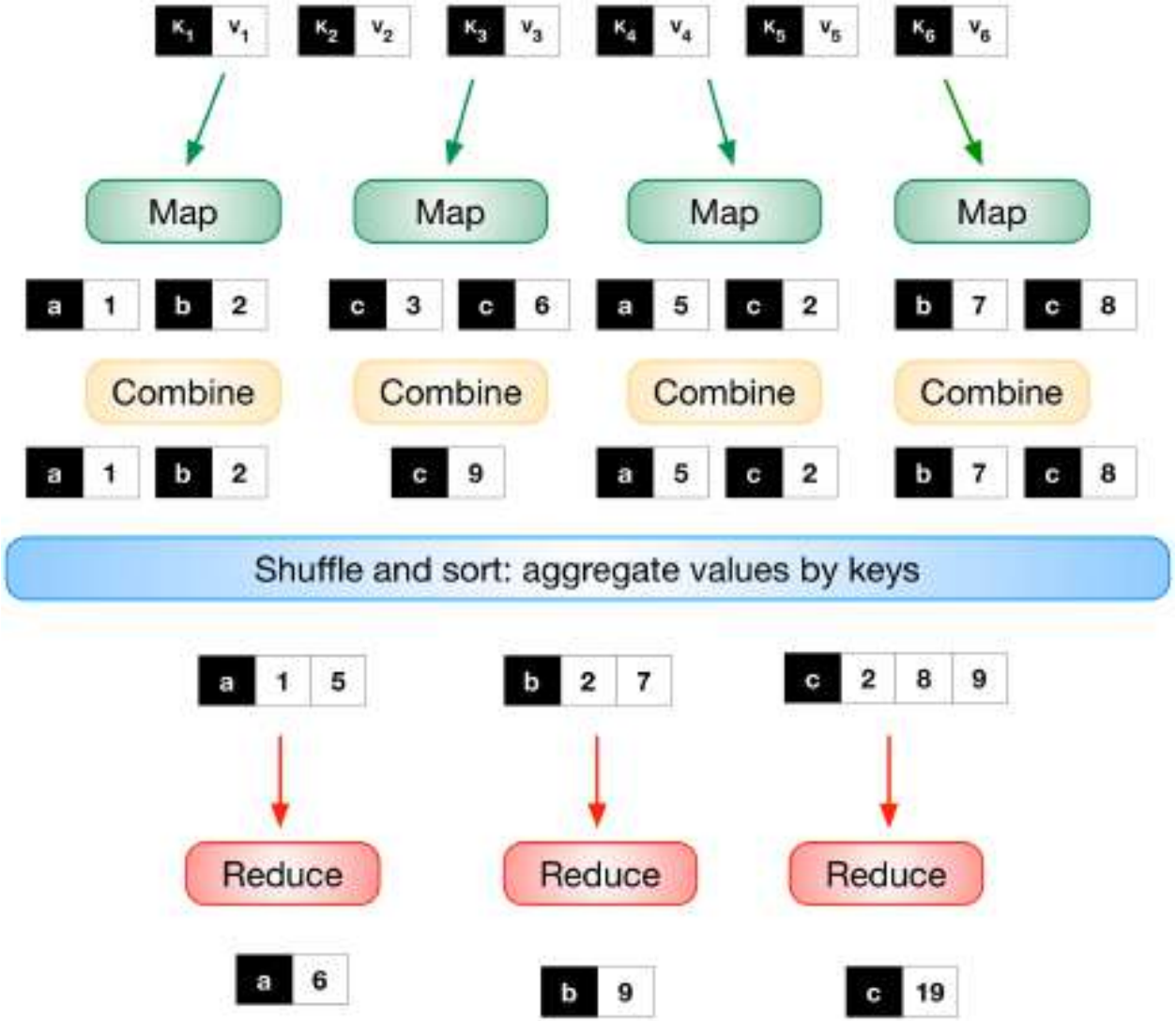
When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks

- In this case we also apply a combiner to the Map function
- In many cases the same function can be used for combining as the final reduction
- Shuffle and sort is still necessary!

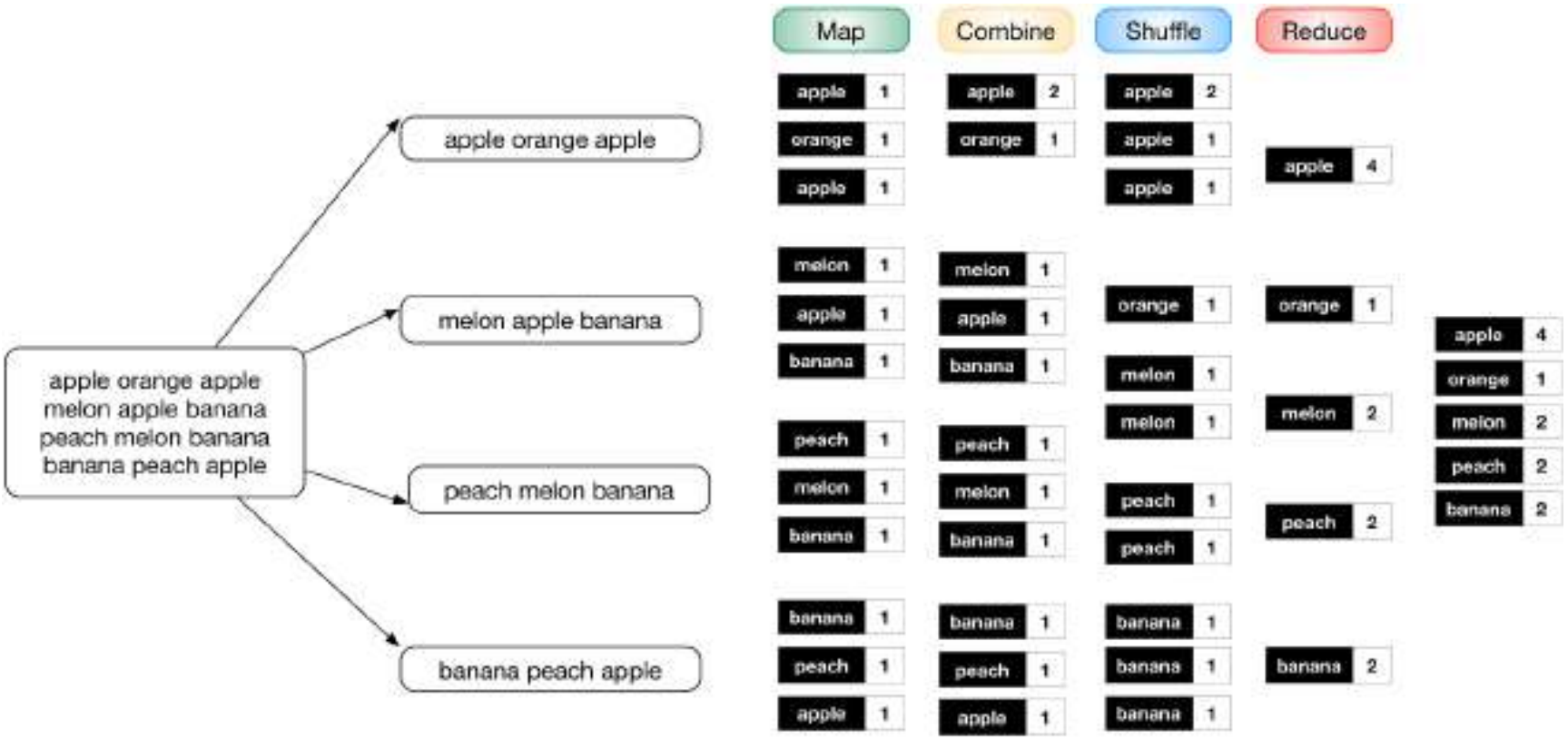
Advantages:

- It reduces the amount of intermediate data
- It reduces the network traffic

MapReduce


K1 V1
K2 V2
K3 V3
K4 V4
K5 V5
K6 V6
Map
Map
Map
Map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Combine
Combine
Combine
Combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
Shuffle and sort: aggregate values by keys
a 1 5
b 2 7
c 2 8 9
Reduce
Reduce
Reduce
a 6
b 9
c 19

MapReduce


Map
Combine
Shuffle
Reduce
apple orange apple
melon apple banana
peach melon banana
banana peach apple
apple orange apple
melon apple banana
peach melon banana
banana peach apple
apple 1
orange 1
apple 1
melon 1
apple 1
banana 1
peach 1
melon 1
banana 1
banana 1
peach 1
apple 1
apple 2
orange 1
melon 1
apple 1
banana 1
peach 1
melon 1
banana 1
banana 1
peach 1
apple 1
apple 2
apple 1
apple 1
orange 1
melon 1
melon 1
peach 1
peach 1
banana 1
banana 1
banana 1
apple 4
orange 1
melon 2
peach 2
banana 2
apple 4
orange 1
melon 2
peach 2
banana 2

Counting words with combiners I

Problem:

counting the number of occurrences for each word in a collection of documents

- Input: a repository of documents, each document is an element
- Map: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

$$(w_1, 1), \ldots, (w_n, 1)$$

- Combiner: groups by key, adds up all the values and emits:

$$(w_1, i), \ldots, (w_n, j)$$

- Shuffle: groups by key and generates pairs of the form

Counting words with combiners II

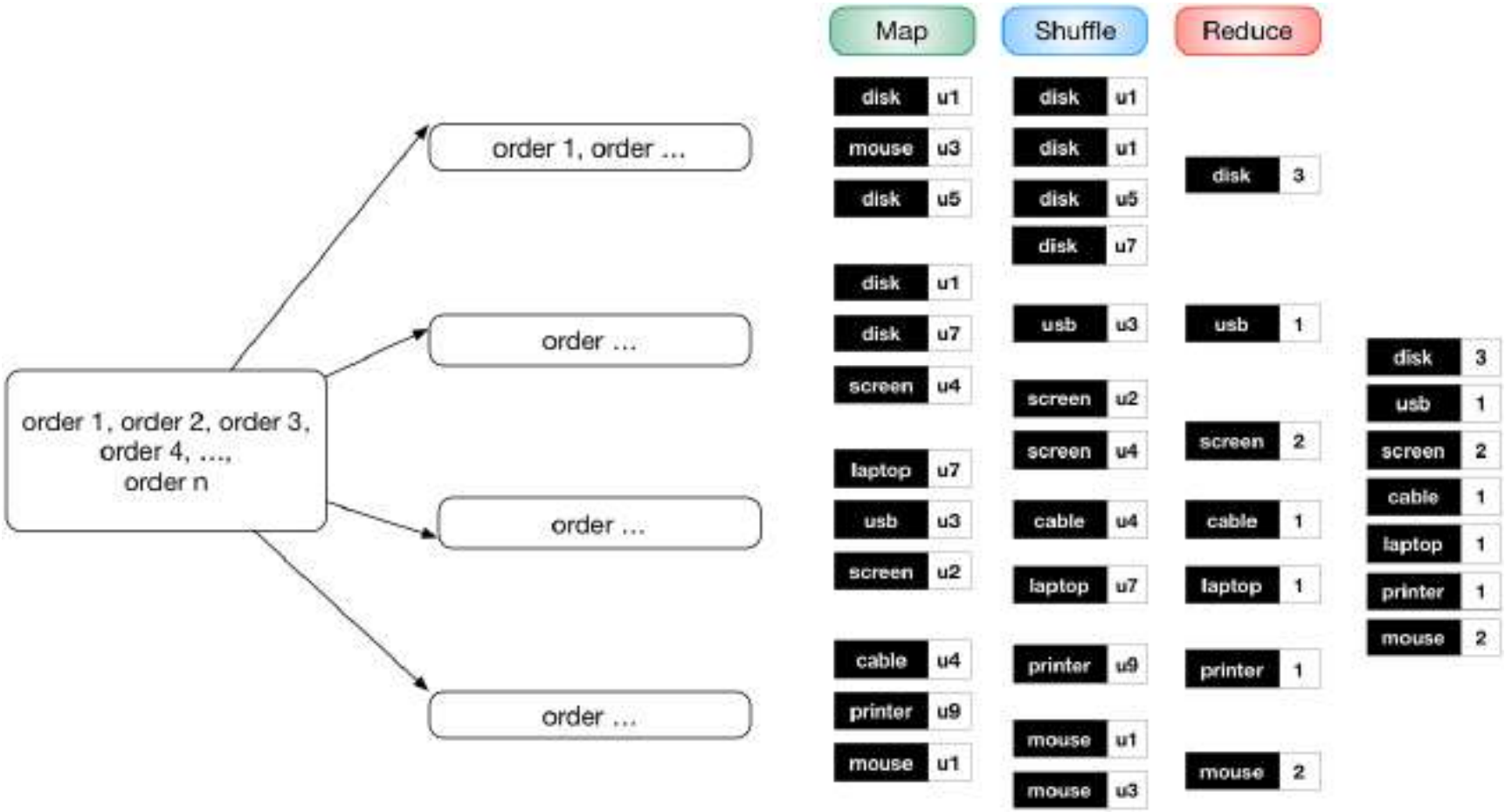
$$(w_1, [1, 1, \ldots, 1]) , \ldots , (w_n, [1, 1, \ldots, 1])$$

- Reduce: adds up all the values and emits:

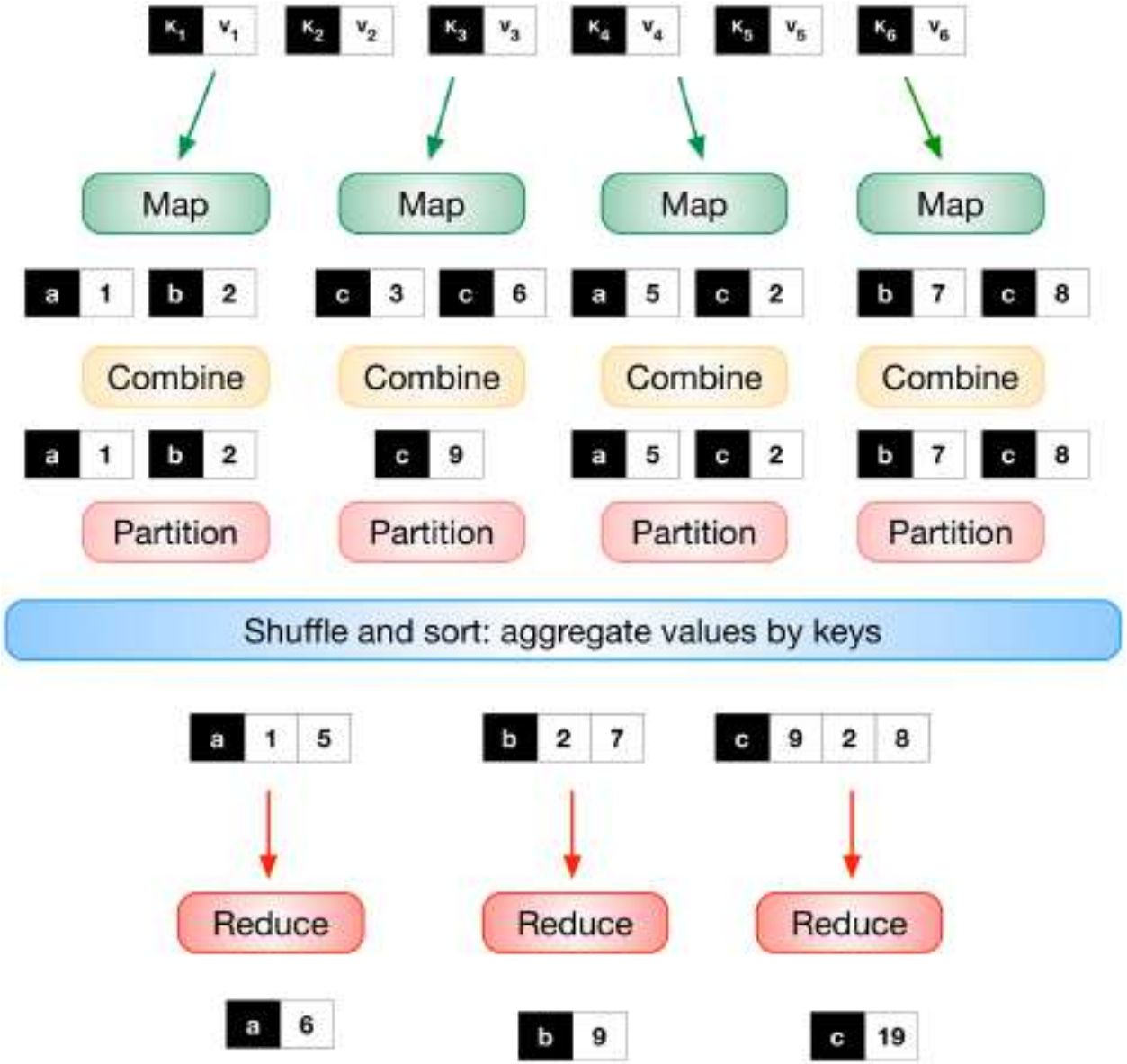
$$(w_1, k) , \ldots , (w_n, l)$$

- Output: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

Different combiners and reducers



Different combiners and reducers



Partitioner

- divides up the intermediate key space
- assigns intermediate key-value pairs to reducers
- n partitions and n reducers

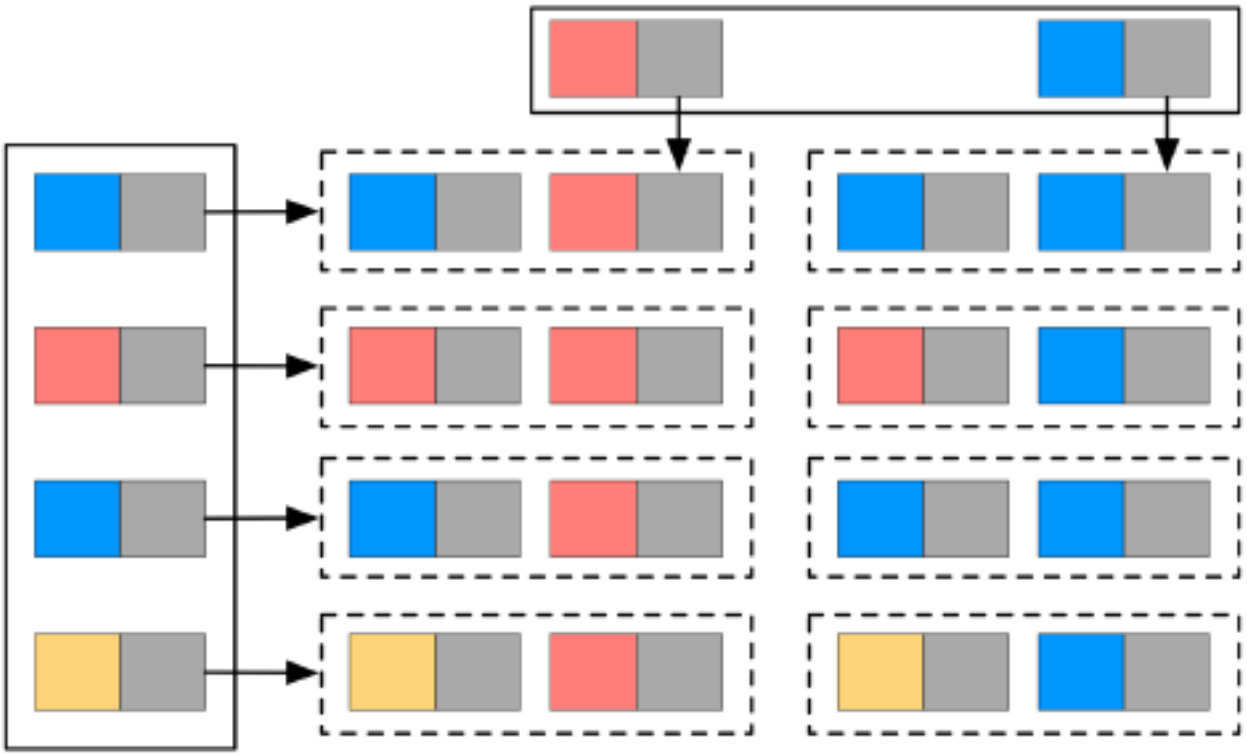
- Assigns approximately the same number of keys to each reducer
- Considers the key and ignores the value
  - Word frequency in inverted indexes

More operations

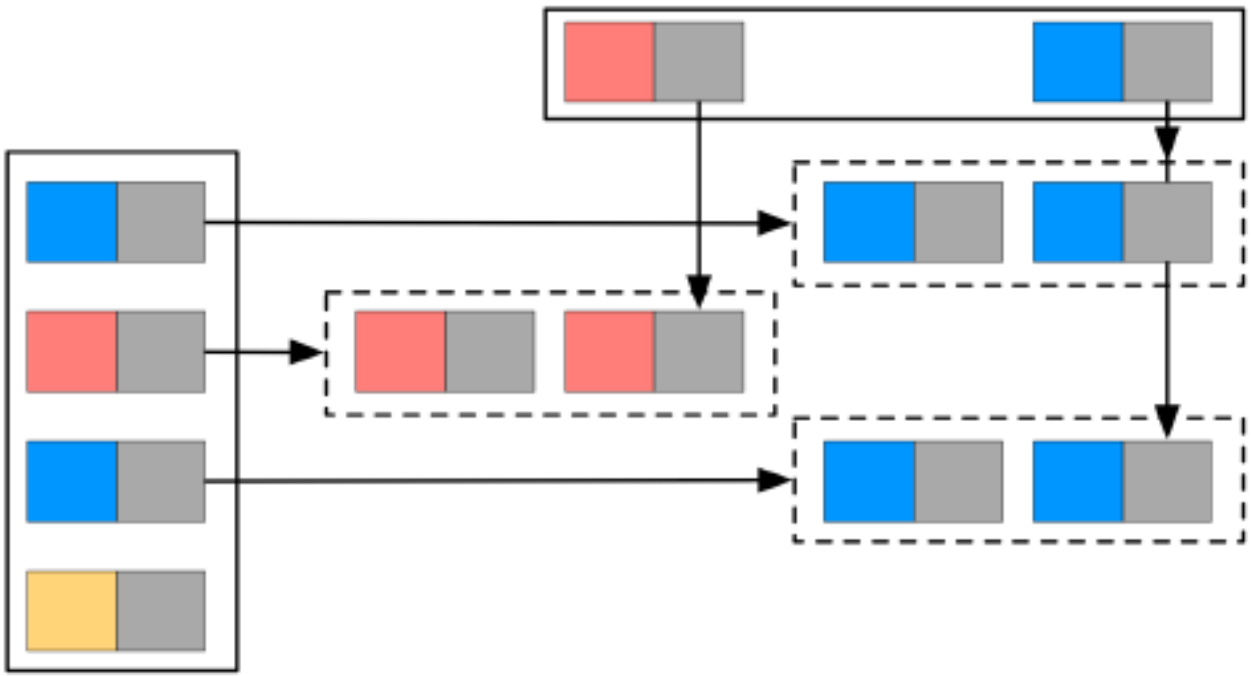
Map and reduce are transformations that work on a single list of keys. We can play with collections sharing the same keys:

- cross product, or a cartesian product
- match or join
- co-group

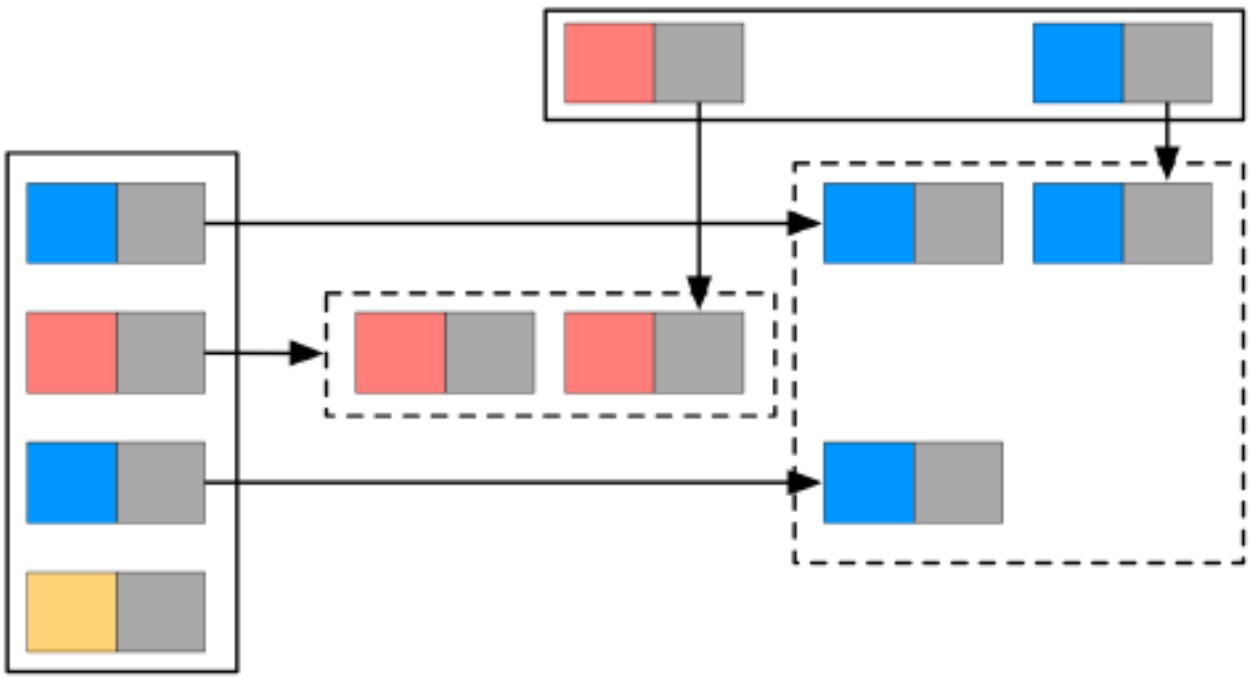
Cartesian Product



Match



Co-group



Summary

MapReduce hides complexities of parallel programming and greatly simplifies building parallel applications

- Data gets reduced to a smaller set at each step

Not good if:

- Data is frequently changing
- Map and reduce are dependent
- You need intermediate results

References I