CLOUD COMPUTING SYSTEMS

Lectures 6-7

Nuno Preguiça

(nuno.preguica_at_fct.unl.pt)

OUTLINE

Computing services

1. First generation batch processing: Map-reduce
2. Second generation batch processing: Spark
3. Stream processing

OUTLINE

Computing services

1. **First generation batch processing: Map-reduce**
    - **Programming model**
    - Execution model
        - Handling faults
2. Second generation batch processing: Spark
3. Stream processing

MapReduce

"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"

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 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 Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a 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.

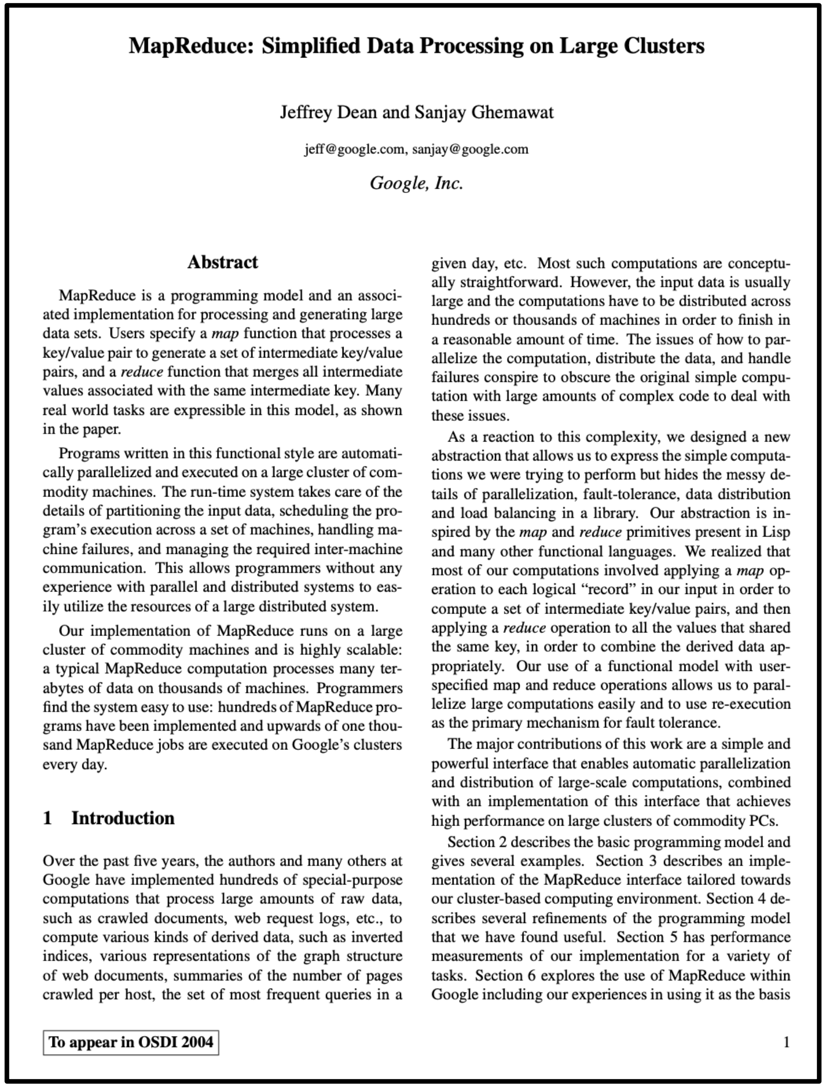
Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

MAPREDUCE (2)

"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."



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 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 Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a 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 implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

EXAMPLE APPLICATION

Consider you have a huge text.

Goal: find out the words that appear more frequently in a text.

Can be transformed into:

Goal 1: Count the number of times each word appears in the text.

Goal 2: Order the words by frequency.

Is this a useless example?

Not really… e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

MapReduce: Overview

Each computation step is composed of a map and a reduce steps.

Sequentially read a lot of data

Map phase:

- Extract the important information

Group by key: Sort and Shuffle the output of the map phase

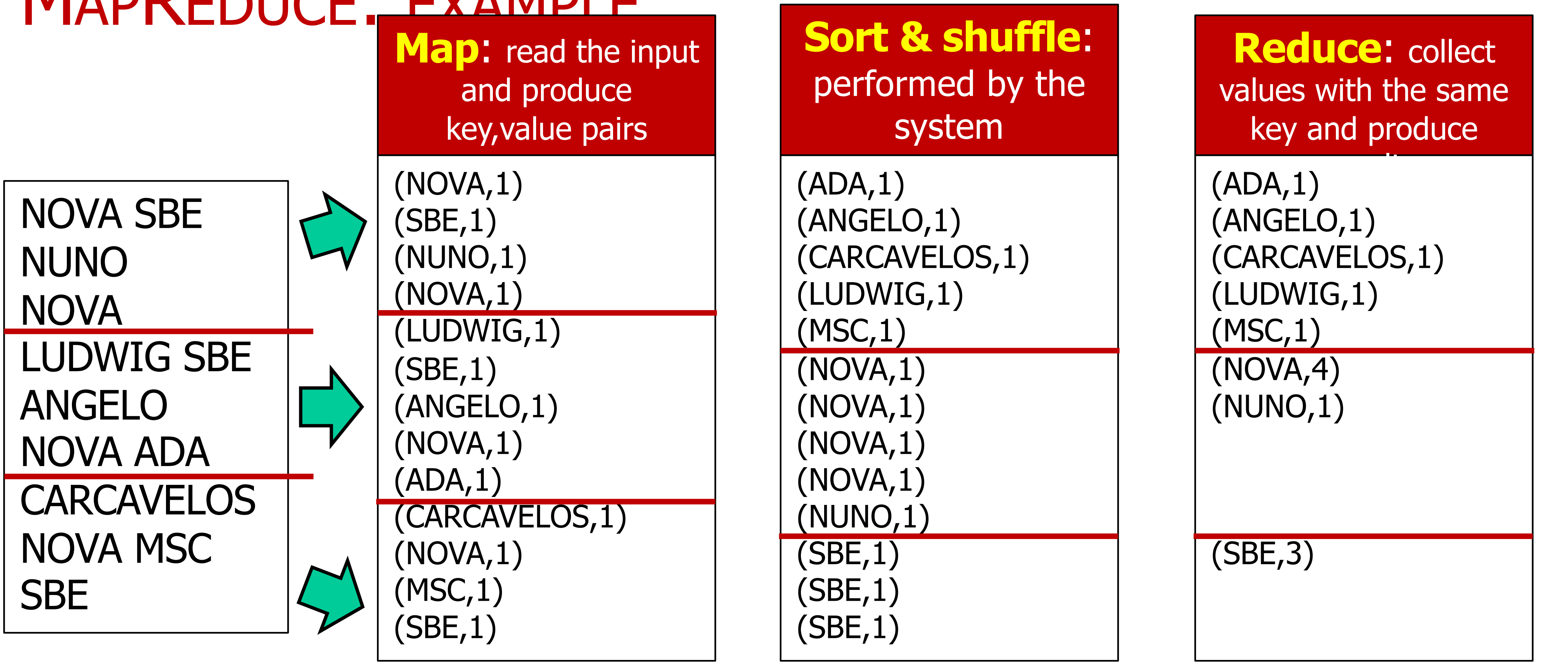
Reduce phase:

- Aggregate, summarize, filter or transform

Write the result

A computation is a sequence of map-reduce computations.

MAPREDUCE: EXAMPLE



NOVA SBE
NUNO
NOVA

LUDWIG SBE
ANGELO
NOVA ADA

CARCAVELOS
NOVA MSC
SBE

Map: read the input and produce key,value pairs

(NOVA,1)
(SBE,1)
(NUNO,1)
(NOVA,1)

(LUDWIG,1)
(SBE,1)
(ANGELO,1)
(NOVA,1)
(ADA,1)

(CARCAVELOS,1)
(NOVA,1)
(MSC,1)
(SBE,1)

Sort & shuffle: performed by the system

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)

(NOVA,1)
(NOVA,1)
(NOVA,1)
(NOVA,1)
(NUNO,1)

(SBE,1)
(SBE,1)
(SBE,1)

Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)

(NOVA,4)
(NUNO,1)

(SBE,3)

Map: for each word, output its count.

Reduce: count the frequency per word.

Word Count Using MapReduce

```
map(key, value):  // key: document name; value: text of the document
    for each word w in value:
        emit(w, 1)

reduce(key, values): // key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
```

MAPREDUCE MODEL

Input: a set of key-value pairs

Programmer specifies two methods:

- **Map(k, v)** → <k', v'>*
  - Takes a key-value pair and outputs a set of key-value pairs
    - E.g., key is the filename, value is a single line in the file
  - Map is called for every *(k,v)* pair
- **Reduce(k', <v'>*)** → <k', v''>*
  - All values *v'* with same key *k'* are reduced together and processed in *v'* order
  - Reduce is called for each unique key *k'*

EXAMPLE APPLICATION

Consider you have a huge text.

Goal: find out the words that appear more frequently in a text.

Can be transformed into:

Goal 1: Count the number of times each word appears in the text.

Goal 2: Order the words by frequency.

Is this a useless example?

Not really… e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

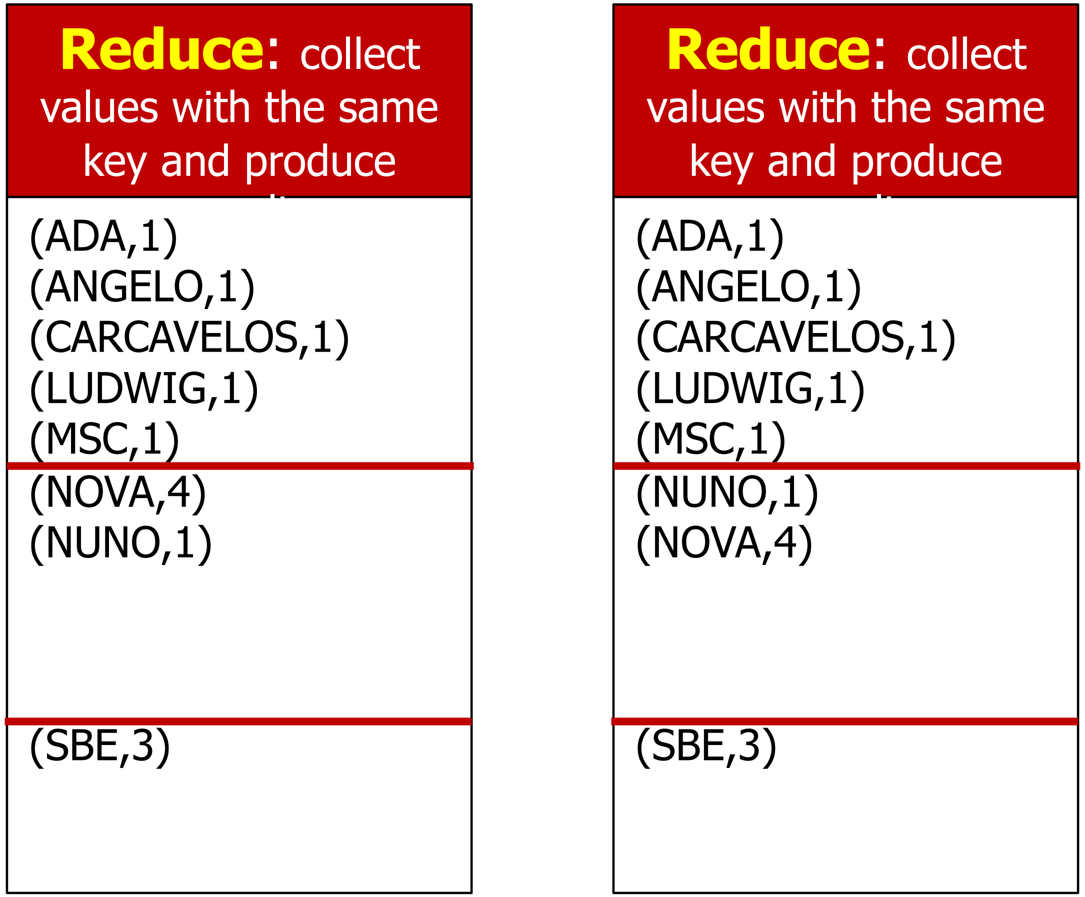
GOAL 2: ORDER THE WORDS BY FREQUENCY.

Can we sort the values of the reduce before returning them?

NO !!!

Each reduce will be processed independently (by a different machine).

Also bad idea because it requires storing potentially large amount of data.



Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)
(NOVA,4)
(NUNO,1)
(SBE,3)

Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)
(NUNO,1)
(NOVA,4)
(SBE,3)

MAPREDUCE: EXAMPLE

	Map: read the input and produce key,value pairs	**Sort & shuffle**: performed by the system	**Reduce**: collect values with the same key and produce
(ADA,1)	(1,ADA)	(1,ADA)	(ADA,1)
(ANGELO,1)	(1,ANGELO)	(1,ANGELO)	(ANGELO,1)
(CARCAVELOS,1)	(1,CARCAVELOS)	(1,CARCAVELOS)	(CARCAVELOS,1)
(LUDWIG,1)	(1,LUDWIG)	(1,LUDWIG)	(LUDWIG,1)
(MSC,1)	(1,MSC)	(1,MSC)	(MSC,1)
(NOVA,4)	(4,NOVA)	(1,NUNO)	(NUNO,1)
(NUNO,1)	(1,NUNO)	(3,SBE)	(SBE,3)
(SBE,3)	(3,SBE)	(4,NOVA)	(NOVA,4)

Map: reverse order of pair.

Reduce: reverse order of pair.

Word Count Sort Using MapReduce

```
map(key, value):  // key: word; value: word count
    emit(value, key)

reduce(key, values): // key: word count; value: word
    for each v in values:
        emit(v, key)
```

MapReduce

Programmer responsible for**:**

- **Map** function
- **Reduce** function

MapReduce system responsible for**:**

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **sort by key & shuffle** step
- Handling machine **failures**
- Managing required inter-machine **communication**

OUTLINE

Computing services

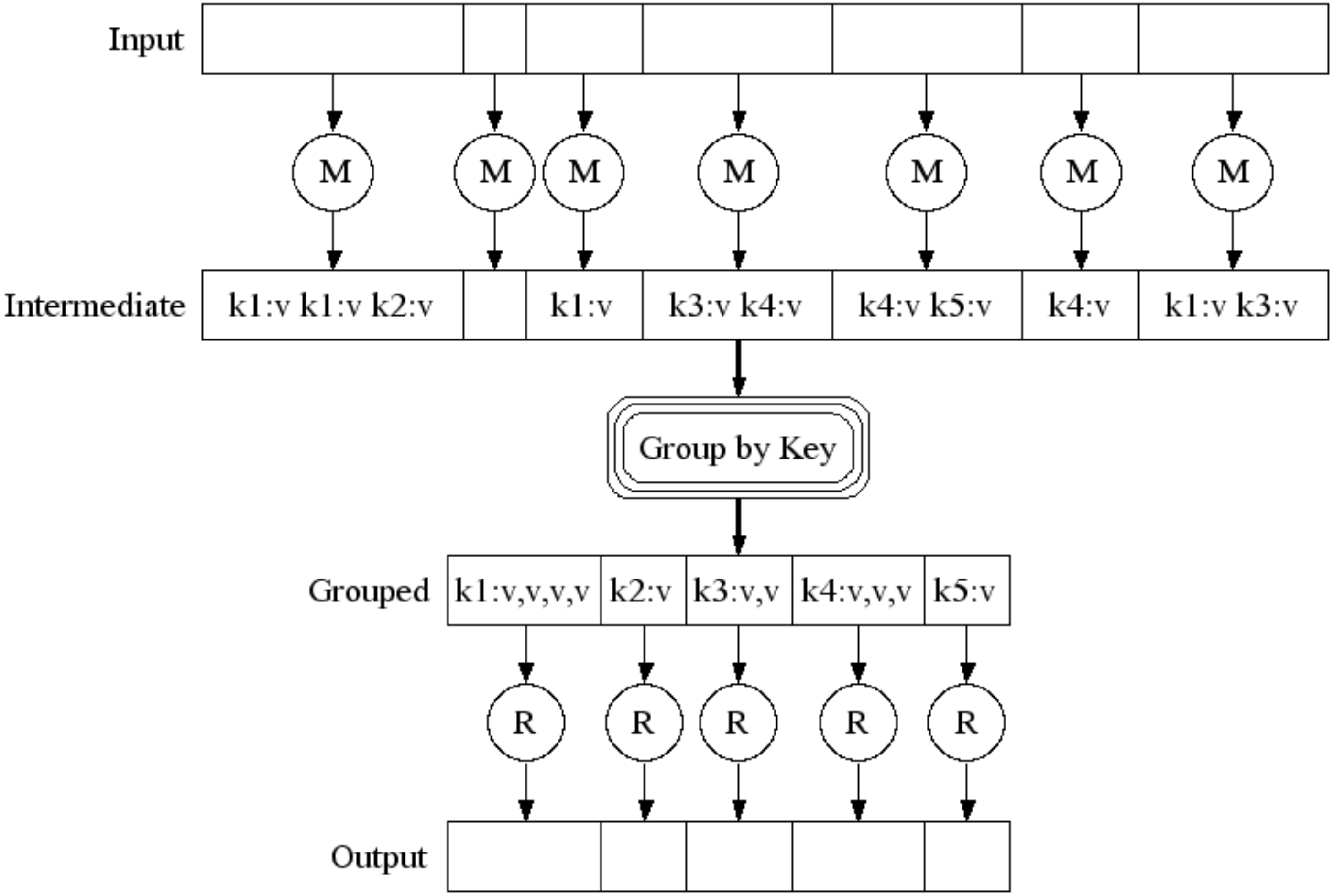
1. **First generation batch processing: Map-reduce**
   - Programming model
   - **Execution model**
     - **Handling faults**
2. Second generation batch processing: Spark
3. Stream processing

MAPREDUCE: LOGICAL EXECUTION…

Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

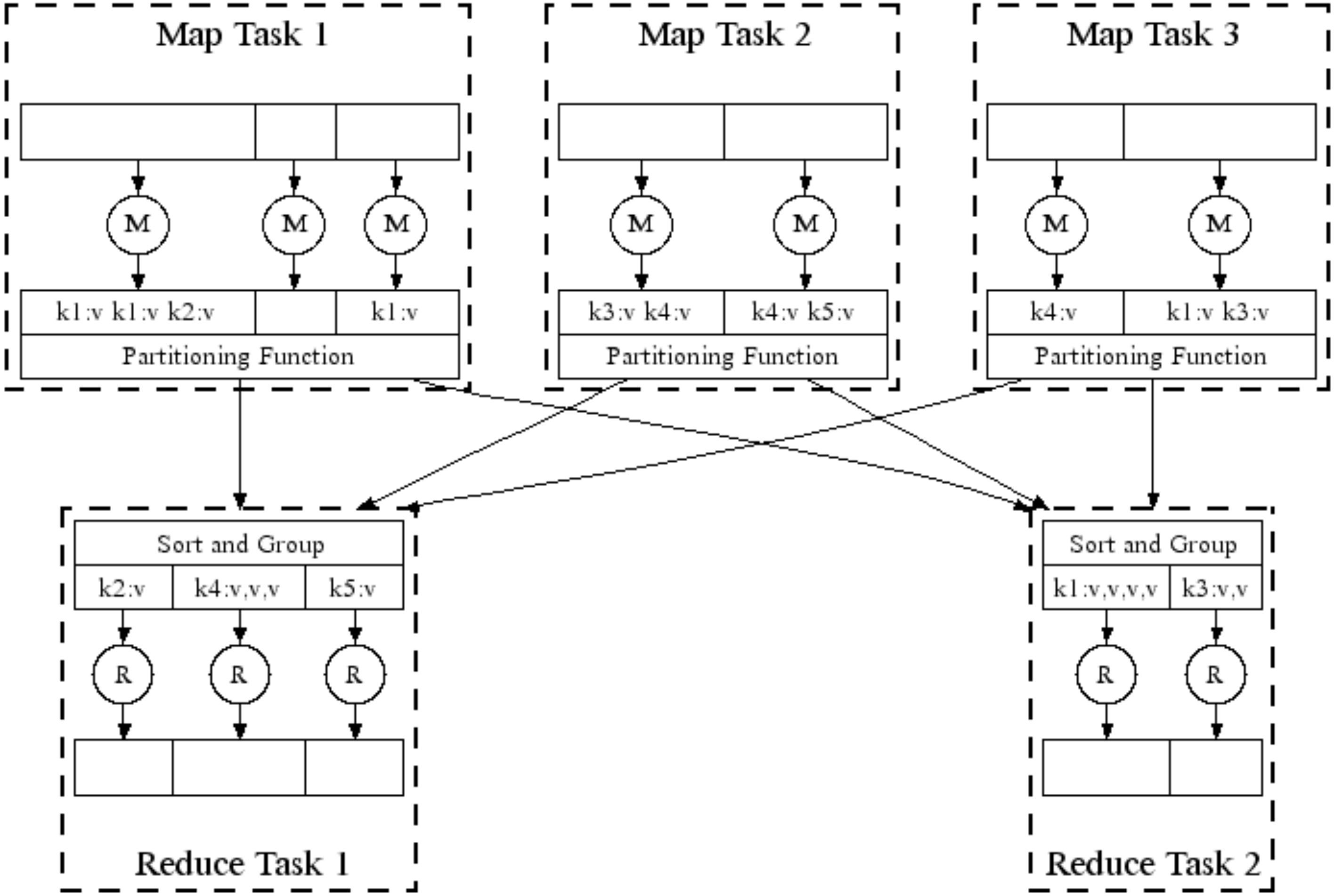


MAPREDUCE: DISTRIBUTED EXECUTION…

Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

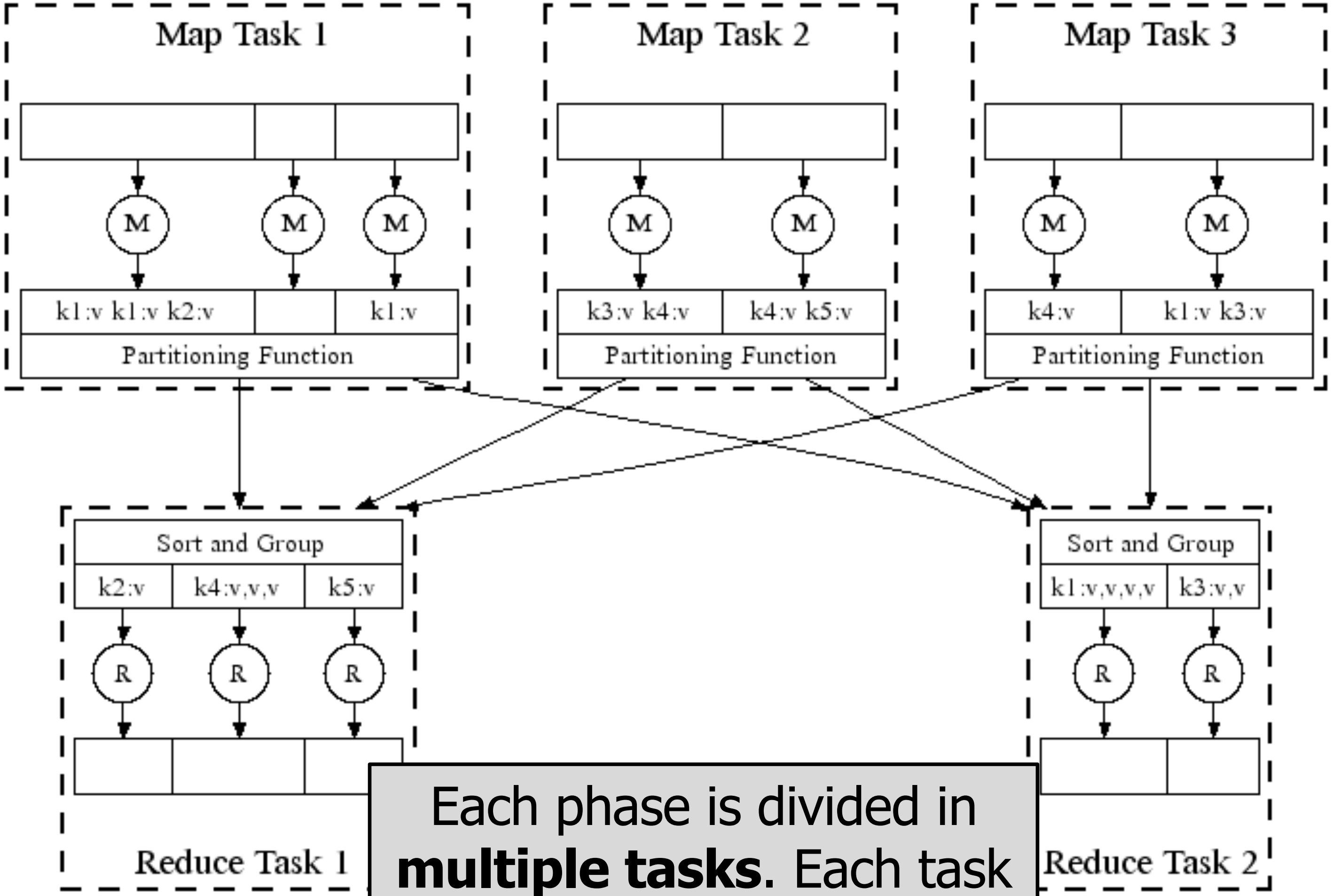


MAPREDUCE: DISTRIBUTED EXECUTION...

Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce



Each phase is divided in **multiple tasks**. Each task is executed independently on a **different nodes.**

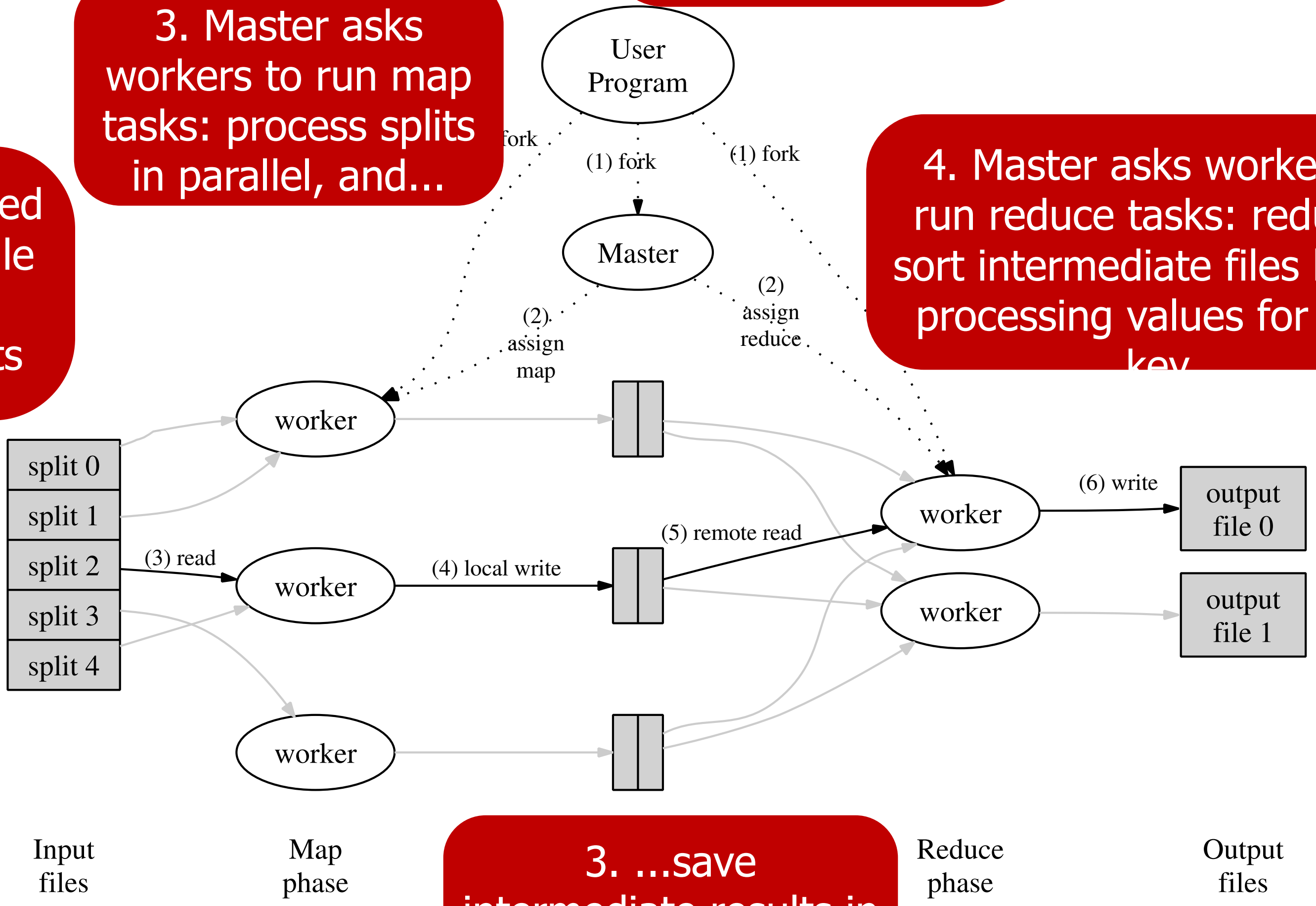
MapReduce architectu

2. Clients send programs to the master.

3. Master asks workers to run map tasks: process splits in parallel, and...

1. Input files stored in a distributed file system, and divided into splits

4. Master asks workers to run reduce tasks: reducers sort intermediate files before processing values for each key



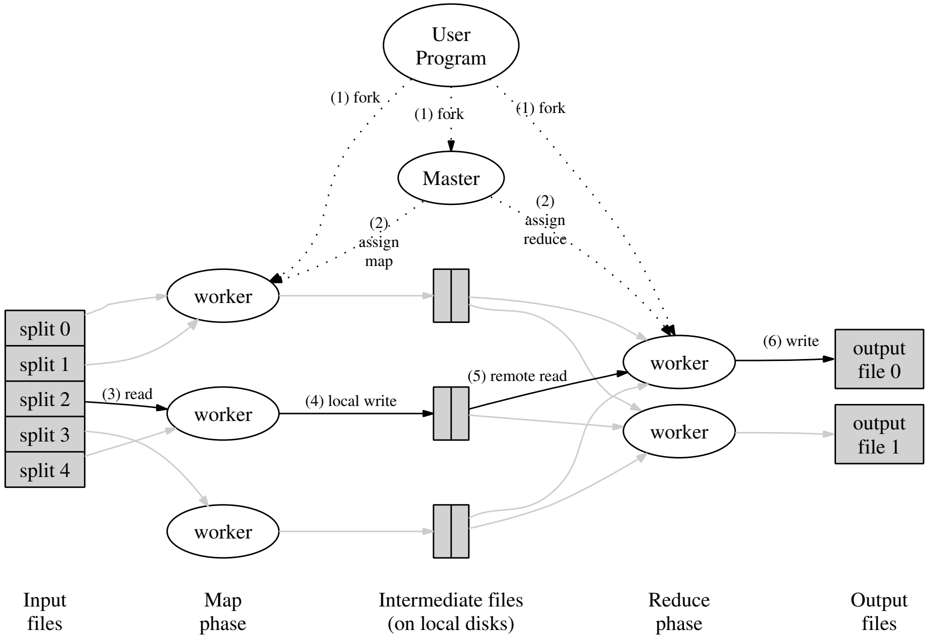
3. ...save intermediate results in multiple files by key

MapReduce system: master node

Master node coordinates the execution:

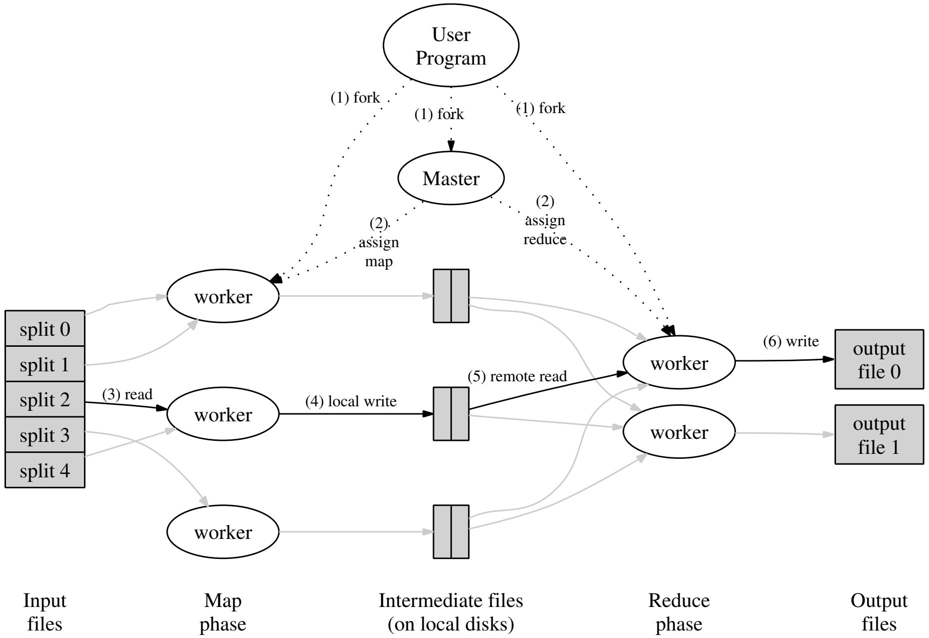
- **Task status:** (idle, in-progress, completed)
- **Idle tasks** get scheduled as workers become available
- When a map task completes, it sends the master the location and sizes of its intermediate files, one for each reducer
- Master pushes this info to reducers

Master pings workers periodically to detect failures



MAPREDUCE SYSTEM: WORKER

Worker node performs map or reduce tasks, as requested by the coordinator.



MapReduce system: handling faults

Map worker failure

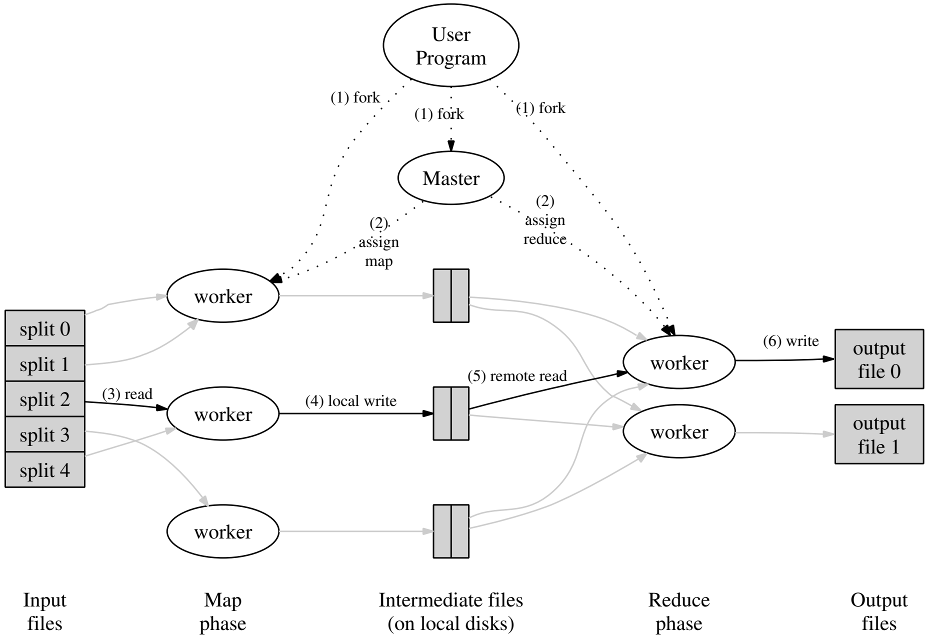
Upon detection of the failure of a worker, map tasks restarted in different worker

Reduce worker failure

Reduce task is restarted in other worker

Stragglers (slow workers)

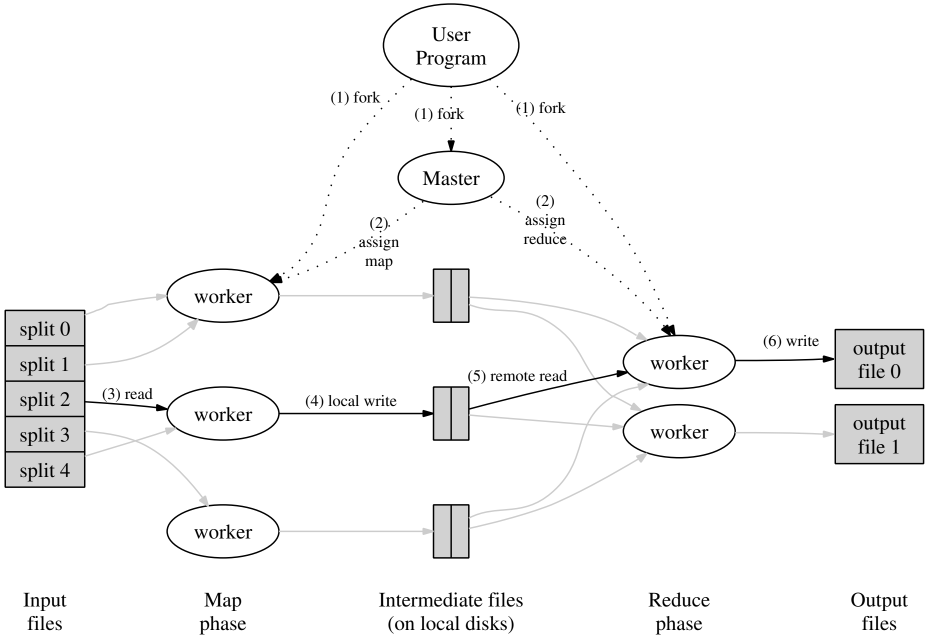
If a task is taking too long to complete, it is launched in other worker. First result used.



MAPREDUCE SYSTEM: HANDLING FAULTS (2)

Master failure

MapReduce task is aborted and client is notified

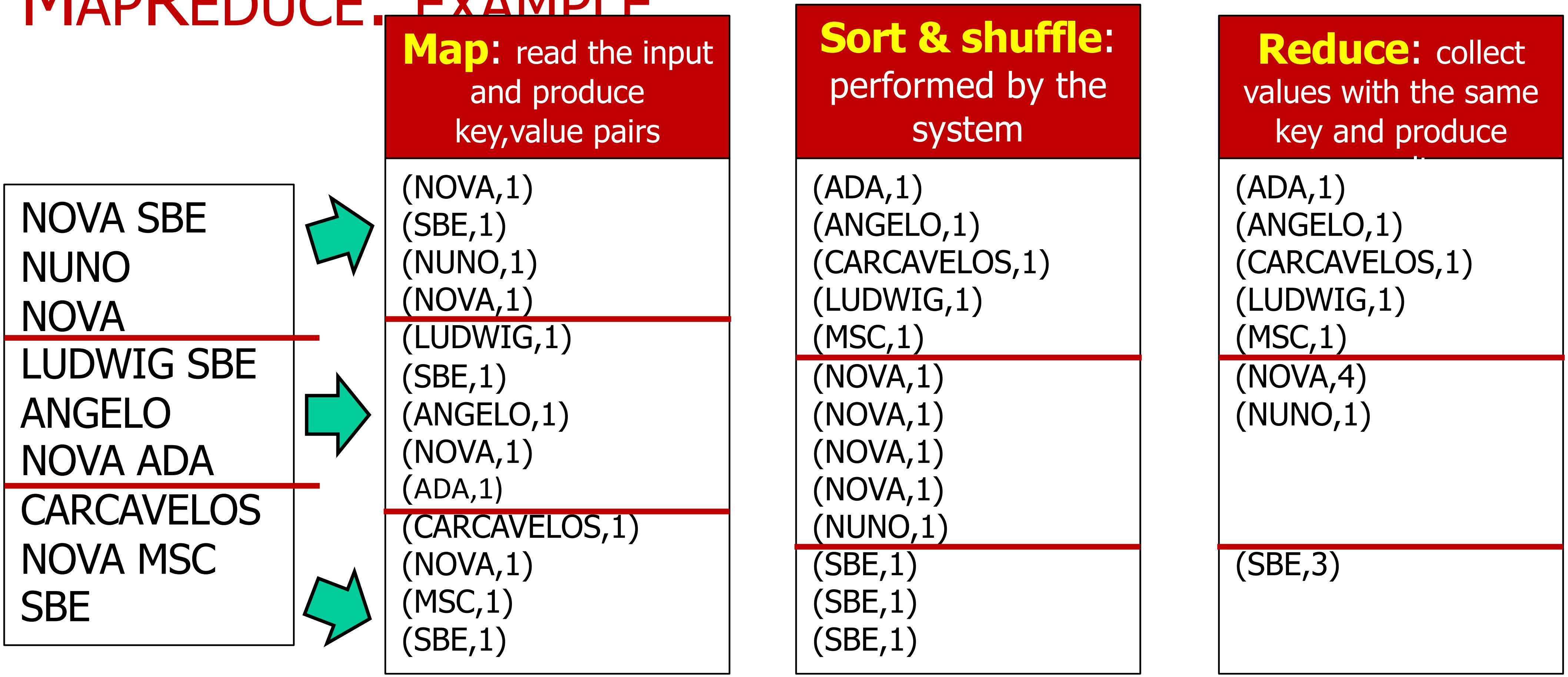


OUTLINE

Computing services

1. **First generation batch processing: Map-reduce**
    - **Programming model**
    - Execution model
        - Handling faults
2. Second generation batch processing: Spark
3. Stream processing

MAPREDUCE: EXAMPLE



NOVA SBE
NUNO
NOVA
LUDWIG SBE
ANGELO
NOVA ADA
CARCAVELOS
NOVA MSC
SBE

Map: read the input and produce key,value pairs

(NOVA,1)
(SBE,1)
(NUNO,1)
(NOVA,1)
(LUDWIG,1)
(SBE,1)
(ANGELO,1)
(NOVA,1)
(ADA,1)
(CARCAVELOS,1)
(NOVA,1)
(MSC,1)
(SBE,1)

Sort & shuffle: performed by the system

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)
(NOVA,1)
(NOVA,1)
(NOVA,1)
(NOVA,1)
(NUNO,1)
(SBE,1)
(SBE,1)
(SBE,1)

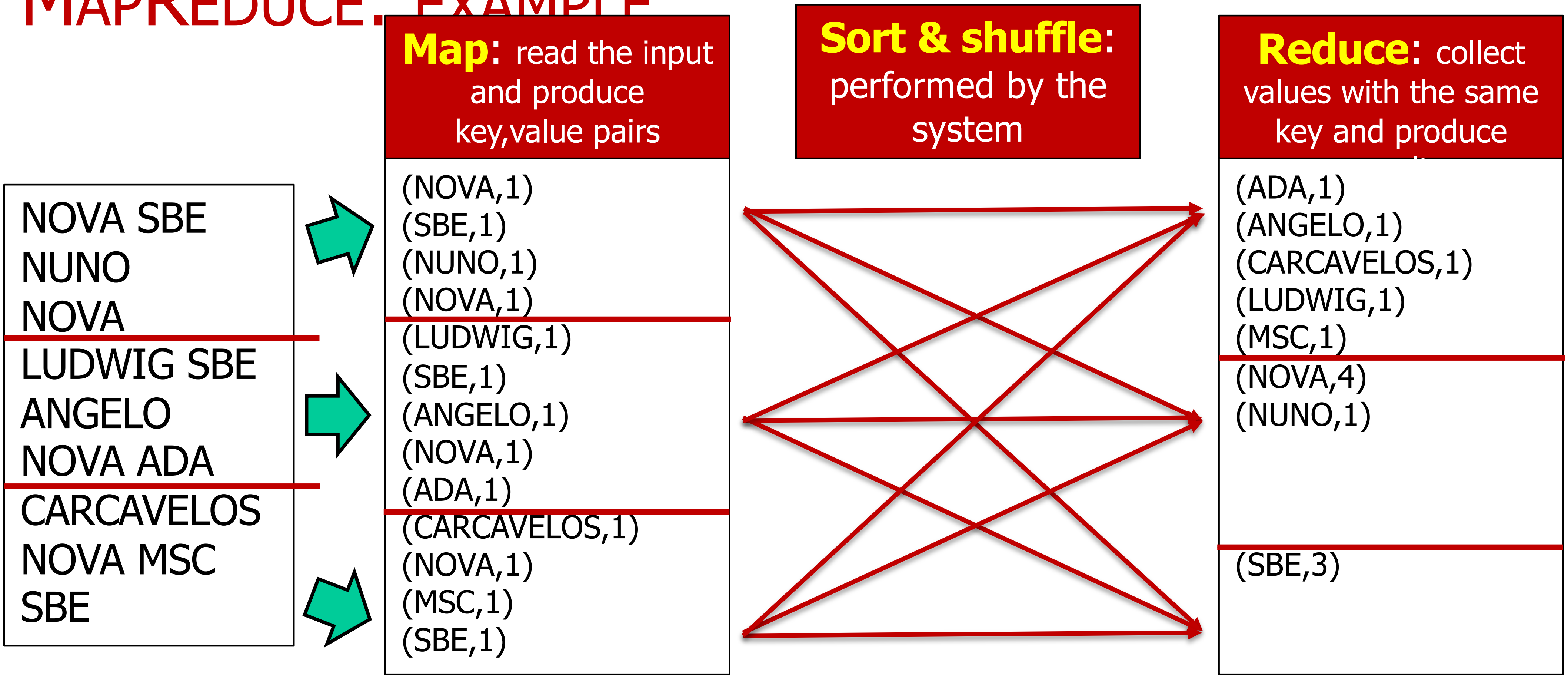
Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)
(NOVA,4)
(NUNO,1)
(SBE,3)

Map: for each word, output its count.

Reduce: count the frequency per word.

MAPREDUCE: EXAMPLE



NOVA SBE
NUNO
NOVA
LUDWIG SBE
ANGELO
NOVA ADA
CARCAVELOS
NOVA MSC
SBE

Map: read the input and produce key,value pairs

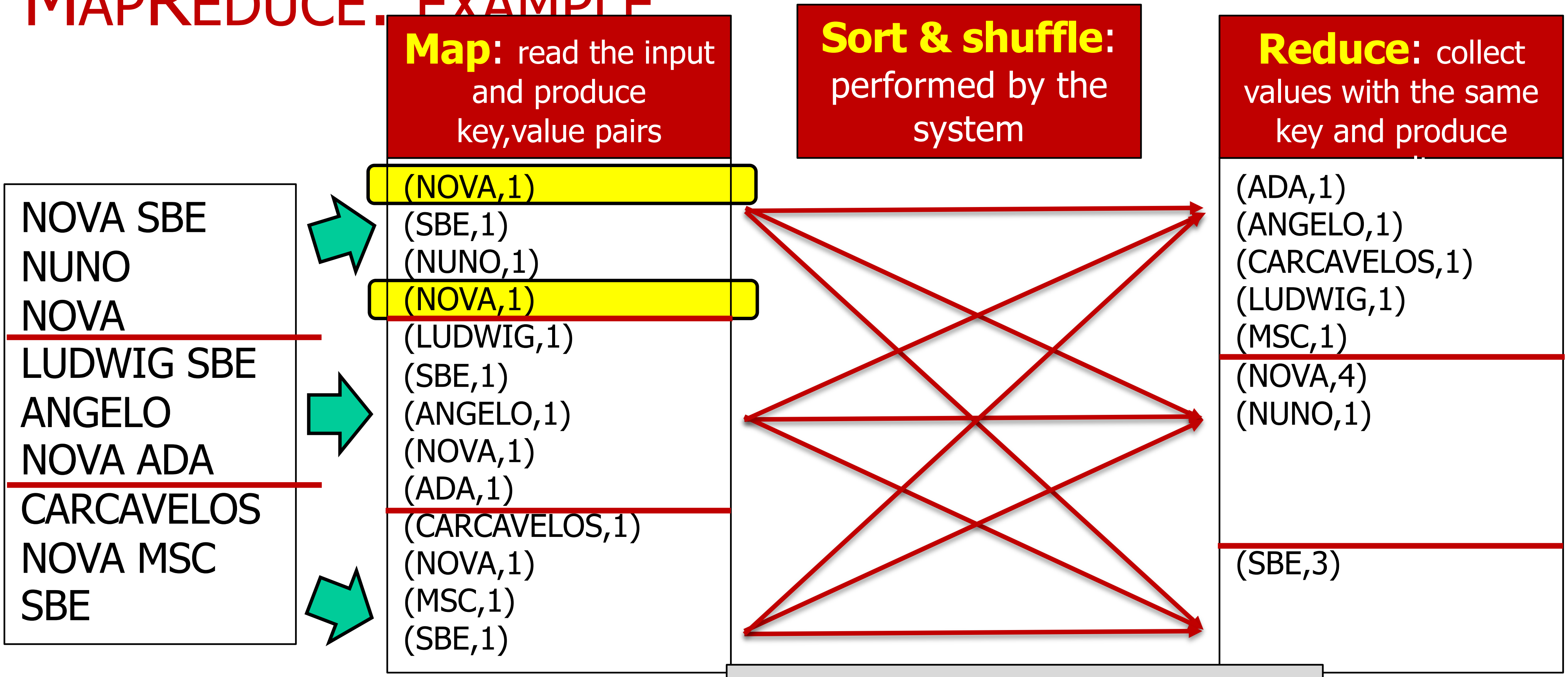
(NOVA,1)
(SBE,1)
(NUNO,1)
(NOVA,1)
(LUDWIG,1)
(SBE,1)
(ANGELO,1)
(NOVA,1)
(ADA,1)
(CARCAVELOS,1)
(NOVA,1)
(MSC,1)
(SBE,1)

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)
(NOVA,4)
(NUNO,1)
(SBE,3)

MapReduce: example



Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

NOVA SBE
NUNO
NOVA

LUDWIG SBE
ANGELO
NOVA ADA

CARCAVELOS
NOVA MSC
SBE

(NOVA,1)
(SBE,1)
(NUNO,1)
(NOVA,1)

(LUDWIG,1)
(SBE,1)
(ANGELO,1)
(NOVA,1)
(ADA,1)

(CARCAVELOS,1)
(NOVA,1)
(MSC,1)
(SBE,1)

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)

(NOVA,4)
(NUNO,1)

(SBE,3)

Mapper 1 sends two tuples for NOVA !!

How to improve this?

Improving MapReduce: Combiner

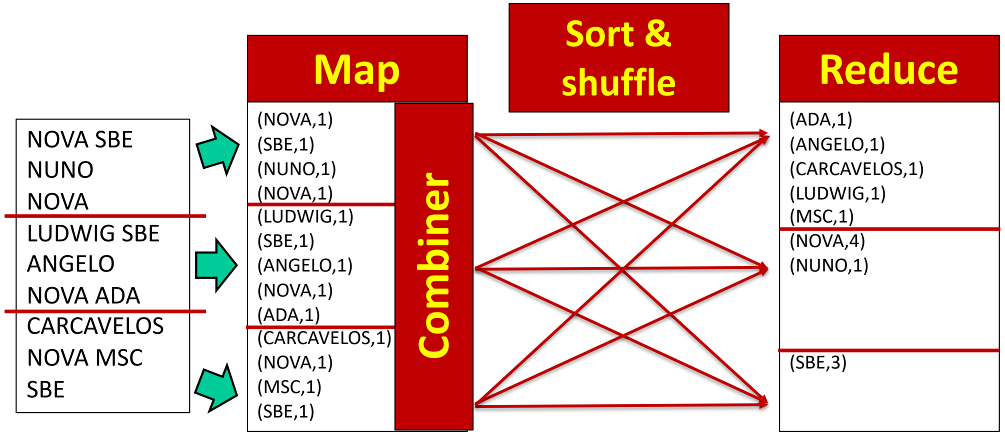
Combiner allows to pre-aggregate values in the mapper.

Combine(k, <v>*) → <k, v’>

All values *v* with same key *k* are combined and processed in *v* order

Combine is called at each mapper for each unique key *k*

Combiner function is usually the same as the reduce function.



WHY WAS MAP-REDUCE SO POPULAR?

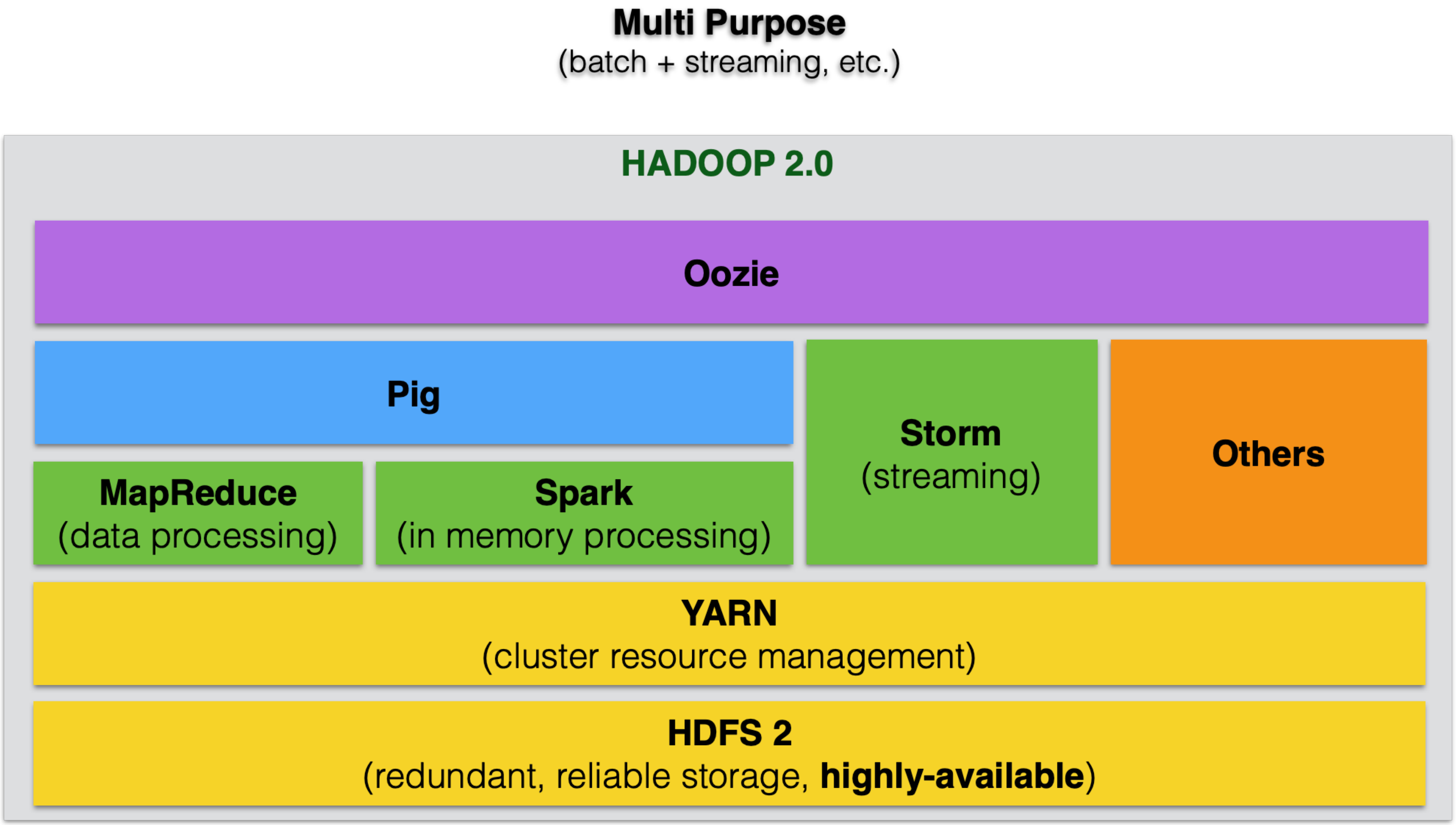
Distributed computation before MapReduce:

- 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?
- systems to efficiently execute map-reduce jobs: Hadoop.

Apache Hadoop 2.0


Multi Purpose
(batch + streaming, etc.)
HADOOP 2.0
Oozie
Pig
MapReduce
(data processing)
Spark
(in memory processing)
Storm
(streaming)
Others
YARN
(cluster resource management)
HDFS 2
(redundant, reliable storage, highly-available)