FIAP GRADUAÇÃO

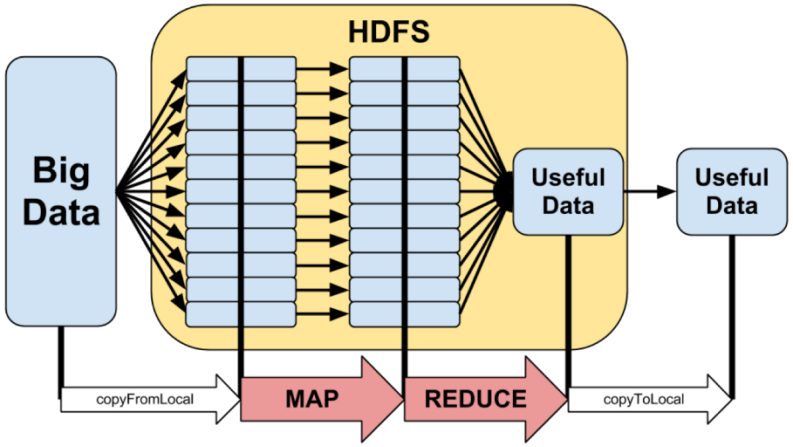
BANCO DE DADOS

ARQUITETURA DE BI E BIG DATA

PROF. MILTON

Versão 2 – <março de 23>

MAP REDUCE


HDFS
Big Data
Useful Data
Useful Data
copyFromLocal
MAP
REDUCE
copyToLocal

Map Reduce

- Em 2004 o Google publicou o artigo chamado MapReduce: Simplified Data Processing on Large Clusters.

- MapReduce é um modelo de programação para processamento de grandes volumes de dados.

- Esta abstração foi inspirada nas primitivas **map** e **reduce** do Lisp e outras linguagens funcionais.

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

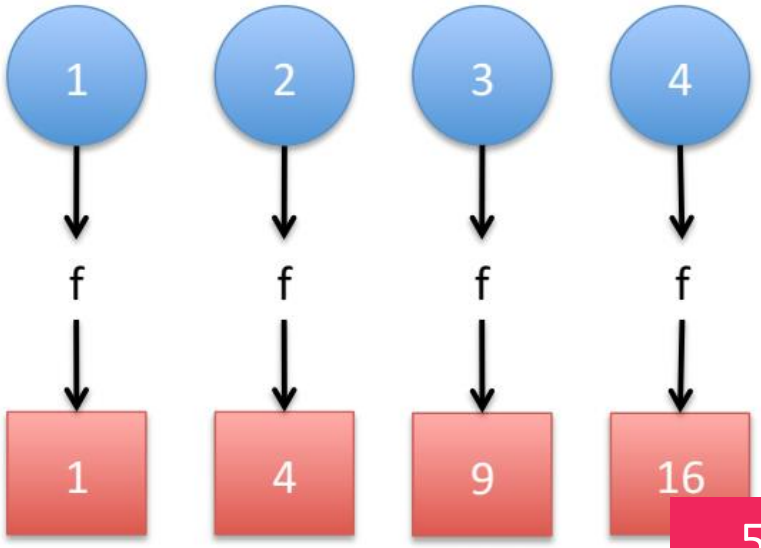
https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Map e Reduce no Lisp

- Lisp é uma linguagem funcional, projetada por John McCarthy que apareceu pela primeira vez em 1958.
- A primitiva **map** recebe como parâmetros um operador unário e uma lista, onde este operador será chamada para cada elemento da lista

```
(map) square) '(1) 2) 3) 4) ))
```

Resultado: `(1 4 9 16)`



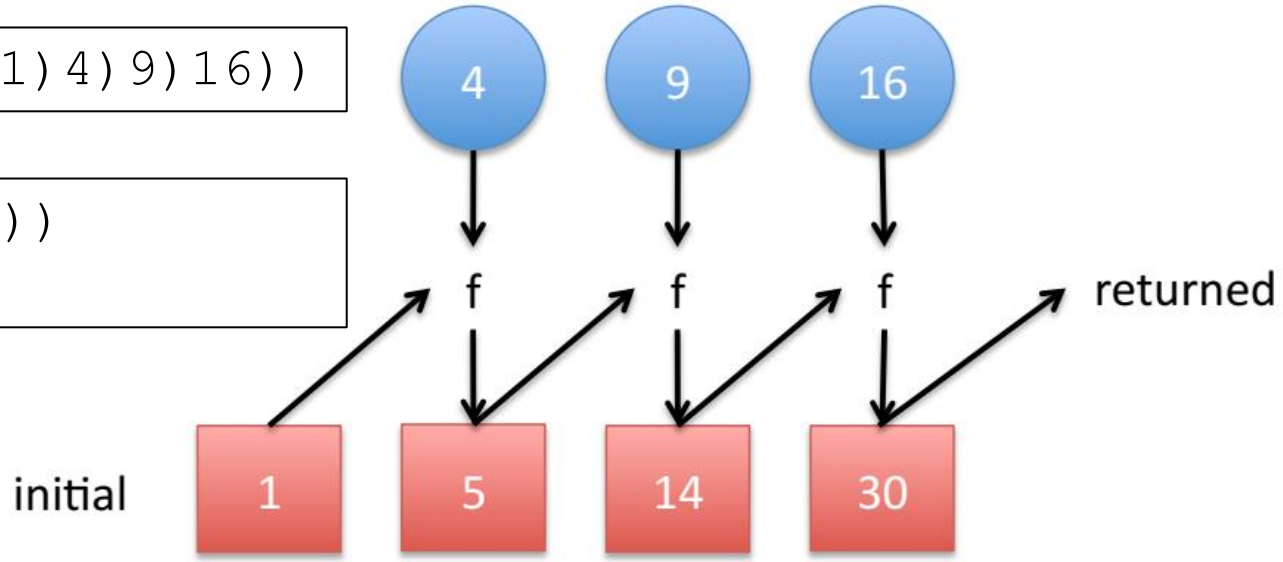
https://cse.buffalo.edu/~stevko/courses/cse704/fall10/papers/cse704-mapreduce.pdf

Map e Reduce no Lisp

- A primitiva **reduce** recebe como parâmetros um operador binário e uma lista, onde os elementos da lista serão combinados utilizando este operador

```
(reduce)+)'(1)4)9)16))
```

```
(+16(+9(+4 1)))
Resultado: 30
```



https://cse.buffalo.edu/~stevko/courses/cse704/fall10/papers/cse704-mapreduce.pdf

Map Reduce em Lisp

- Map
  - processa cada registro individualmente
- Reduce
  - Processa (combina) o conjunto de todos os registros em lote

https://cse.buffalo.edu/~stevko/courses/cse704/fall10/papers/cse704-mapreduce.pdf

O que a Google percebeu

- Pesquisa por palavra-chave
  - Encontre uma palavra-chave em cada página da web individualmente e, se ela for encontrada, retorne sua URL

Map

  - Combine todos os resultados (URLs) e devolva-os

Reduce

- Contagem do número de ocorrências de cada palavra
  - Conte o número de ocorrências em cada página da web individualmente e retorne a lista de <palavra, número>

Map

  - Para cada palavra, some (combine) a contagem

Reduce

https://renil.wordpress.com/2007/05/03/mapreduce-functional-programming-lisp/

Visão Geral do Map Reduce

Map

Os dados na fase map são divididos entre os nós task tracker onde os dados estão localizados. Cada nó na fase map emite pares chave-valor com base no registro de entrada, um registro por vez.

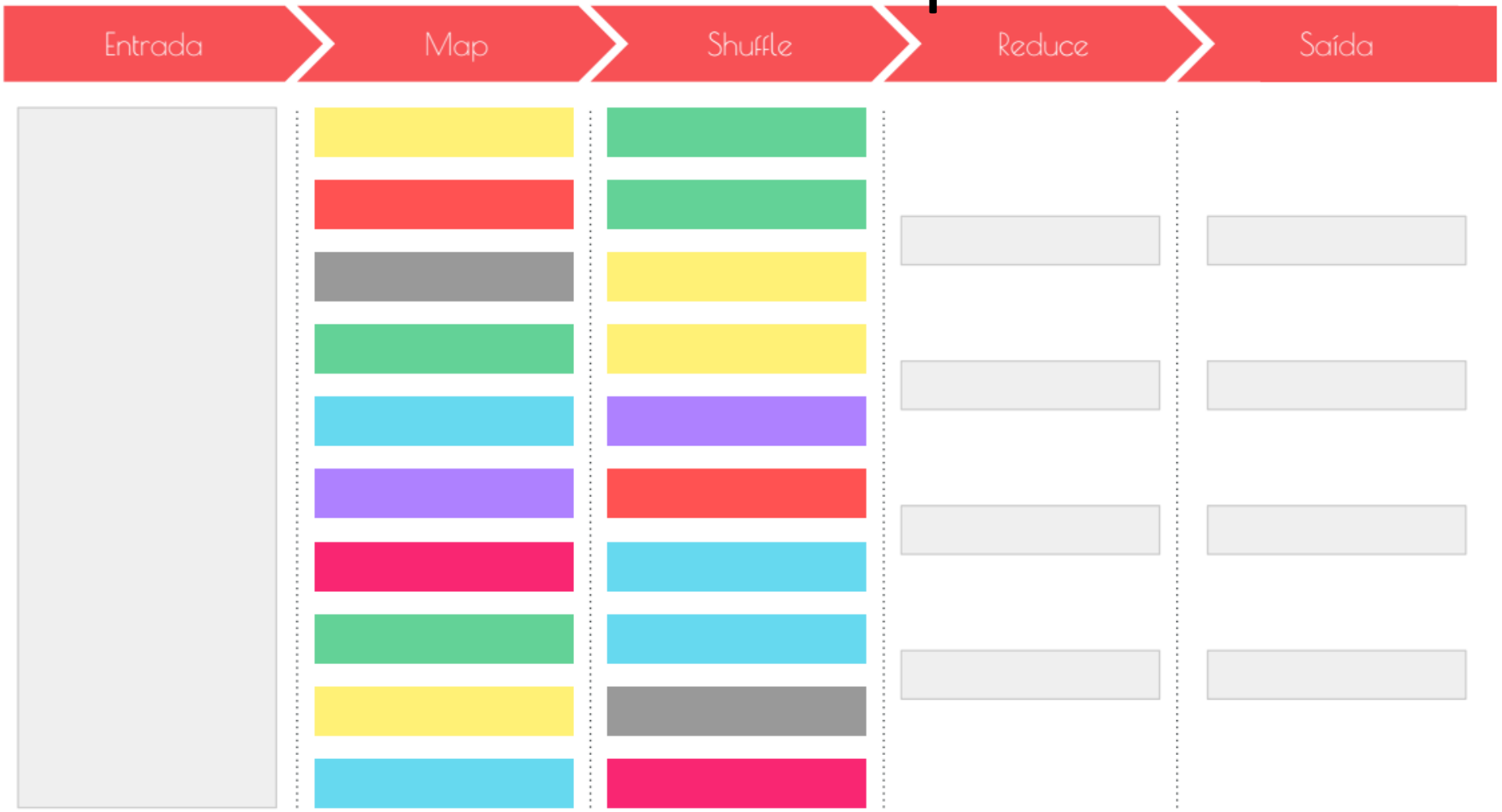
Shuffle

A fase shuffle é tratada pela estrutura Hadoop. Ela transfere os resultados dos mappers para os reducers juntando os resultados por meio chave.

Reduce

A saída dos mappers é enviado para os reducers. Os dados na fase reduce são divididos em partições onde cada reducer lê uma chave e uma lista de valores associados a essa chave. Os reducers emitem zero ou mais pares chave-valor com base na lógica utilizada

Visão Geral do Map Reduce



Visão Geral do Map Reduce FIAP

Entrada → Map → Shuffle → Reduce → Saída

"A bela jovem saiu com um belo rapaz para um belo passeio no belo parque desta bela cidade"

Chave	Valor
A	1
bela	1
jovem	1
saiu	1
com	1
um	1
belo	1
rapaz	1
para	1
um	1
belo	1
passeio	1
no	1
belo	1
parque	1
desta	1
bela	1
cidade	1

Chave	Valor
A	1
bela	1
bela	1
belo	1
belo	1
belo	1
cidade	1
com	1
desta	1
jovem	1
no	1
para	1
parque	1
passeio	1
rapaz	1
saiu	1
um	1
um	1

Chave	Valor
A	1
bela	1, 1
belo	1, 1, 1
cidade	1
com	1
desta	1
jovem	1
no	1
para	1
parque	1
passeio	1
rapaz	1
saiu	1
um	1, 1

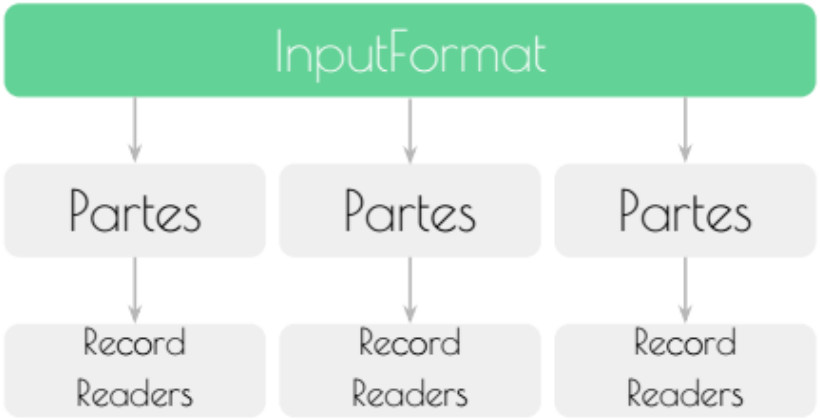
Chave	Valor
A	1
bela	2
belo	3
cidade	1
com	1
desta	1
jovem	1
no	1
para	1
parque	1
passeio	1
rapaz	1
saiu	1
um	2

FIAP

InputFormat

- O primeiro passo para executar um programa MapReduce é localizar e ler o arquivo de entrada contendo os dados brutos.
- O formato do arquivo é completamente arbitrário, mas os dados devem ser convertidos para algo que o programa pode processar.
- Isso é feito pelo InputFormat e RecordReader.
- InputFormat decide como o arquivo será dividido em peças menores para processamento usando uma função chamada InputSplit.

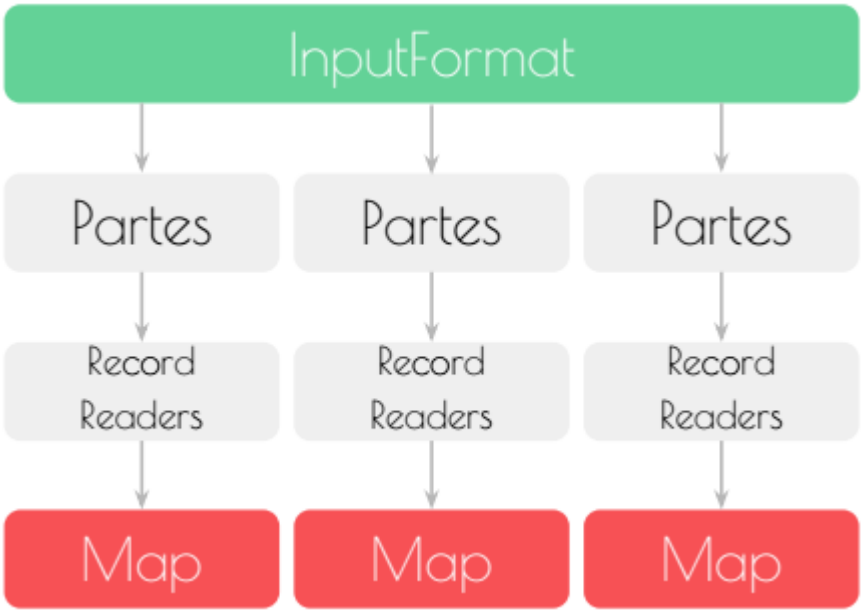
InputFormat



- O objeto **InputFormat** é responsável por validar a entrada, dividindo os arquivos entre os **mappers** e instanciando os objetos **RecordReaders**.
- Por padrão, o tamanho de uma parte é igual ao tamanho de um bloco que no Hadoop o padrão é 64 Mb.
- As partes possuem um conjunto de registros onde cada um deles será quebrado em pares chave-valor para o **map**. A separação dos registros é feita antes mesmo da instanciação do processo map.
- O **job** que está executando a tarefa **MapReduce** tentará colocar a tarefa **map** o mais próximo dos dados possível, ou seja, executar o processo **map** no mesmo nó do cluster onde o dados está armazenado.

Método MAP

- O método map recebe como argumento 3 parâmetros:
  - Writable chave,
  - Writable valor e
  - context.
- Por padrão, o **RecordReader** define a chave para o método **map** como sendo o byte offset do registro no arquivo de entrada e o valor é basicamente a linha inteira.

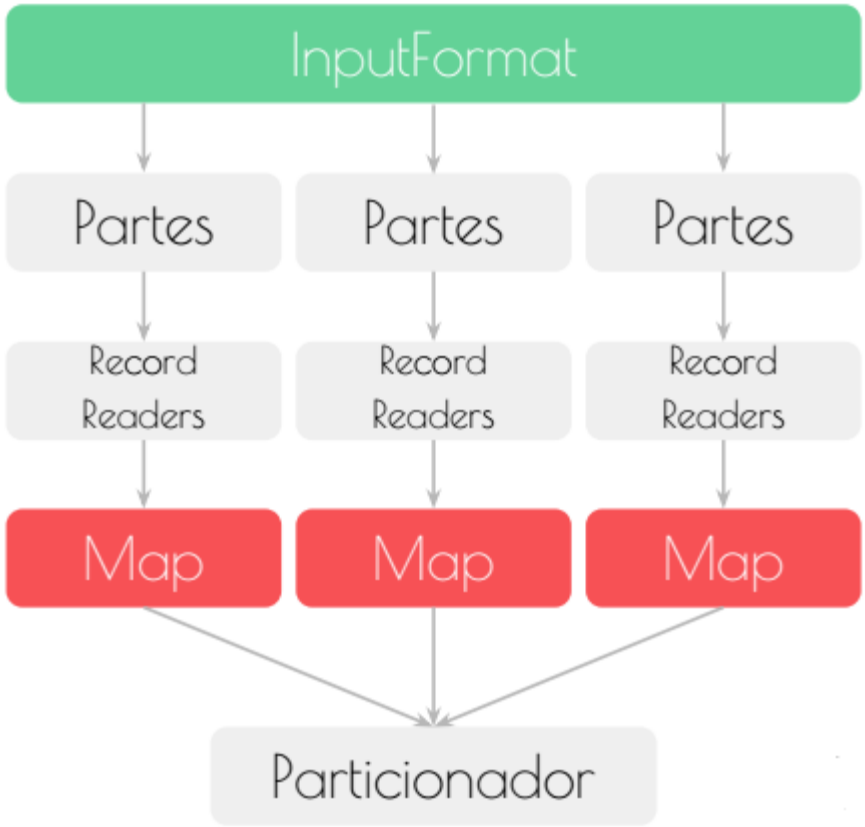


Computer-science World
Quantum Computing

<0,Computer Science World>
<23,Quantum Computing>

Particionador

- O particionador recebe a saída gerada pelo método **map** e faz um **hash** da chave e cria uma partição com base no **hash** da chave.
- Cada partição se destina a um **reducer**, assim, todos os registros com a mesma chave serão enviados para a mesmo partição (e, portanto, enviados para o mesmo **reducer**).



Reduce

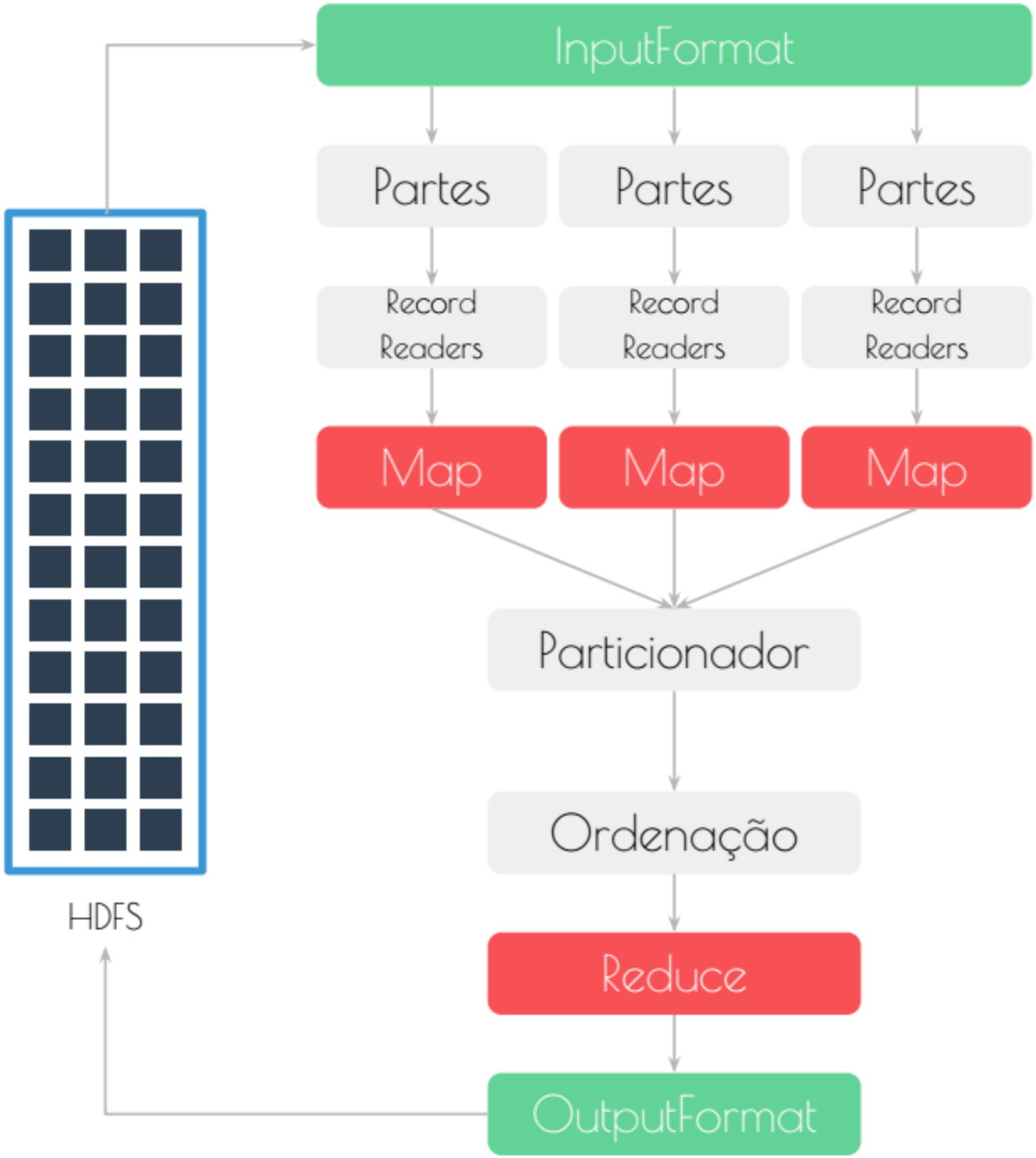
- O método **reduce** é chamado para cada chave e a lista de valores associados a essa chave.
- O método **reduce** processa cada valor e grava o envia o resultado para o contexto.
- O **OutputCommitter** cria um arquivo de saída para cada reduce executado.

OutputFormat

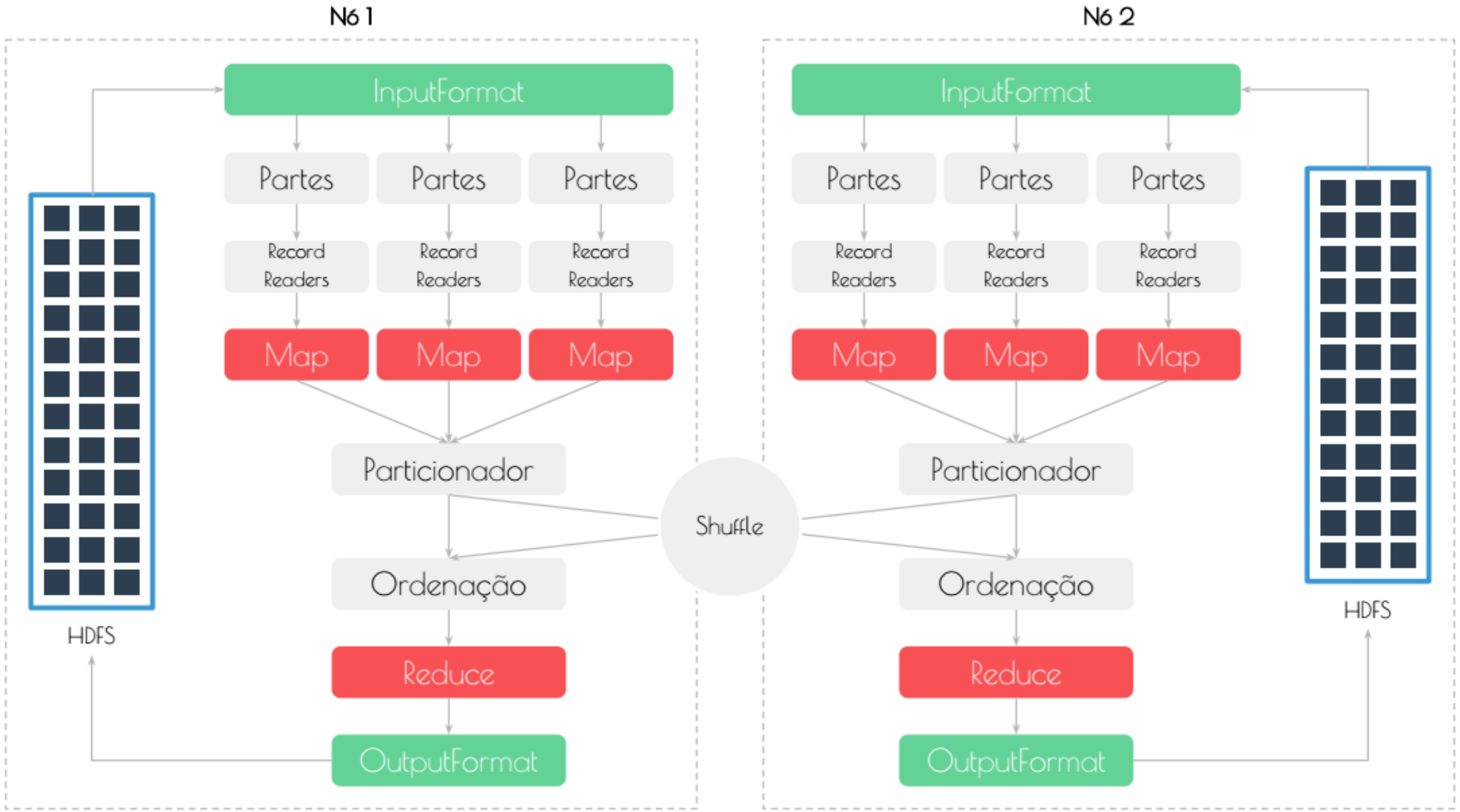
- Os resultados de um job MapReduce são armazenados no diretório especificado pelo usuário.
- Um arquivo vazio chamado _SUCCESS é criado para indicar que o job foi concluído com sucesso (mas não necessariamente sem erros).

OutputFormat

- O histórico do job é armazenado em _logs/history*.
- A saída do método reduce é salvo em arquivos denominados part-r-00000, part-r-00001, ... (um para cada reduce).
- Caso seja executada uma tarefa apenas de mapeamento, implementação apenas do map, os nomes dos arquivos de saída serão part-m-00000, part-m-00001, ... etc.

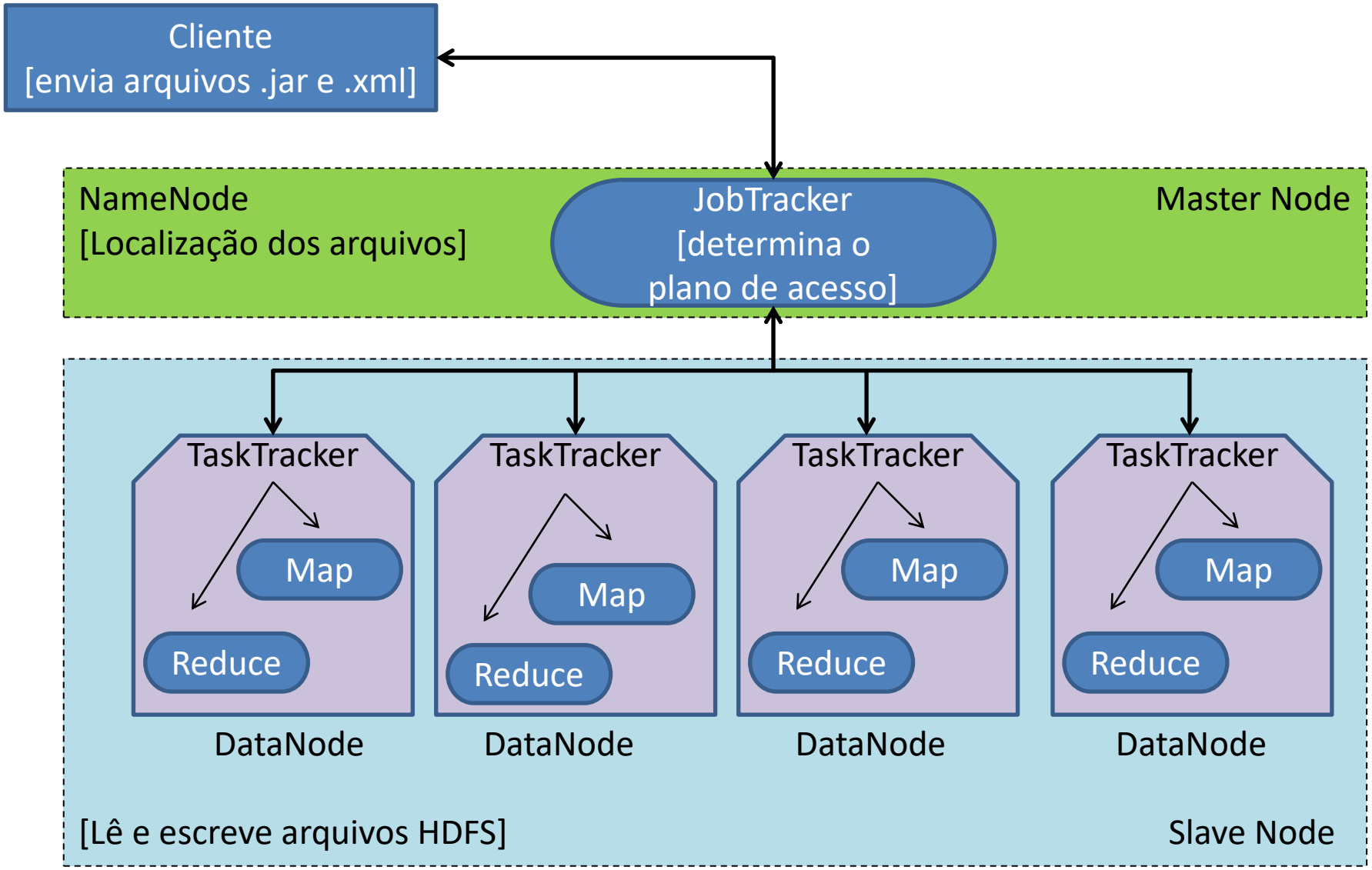

InputFormat
Partes
Partes
Partes
Record Readers
Record Readers
Record Readers
Map
Map
Map
Particionador
Ordenação
Reduce
OutputFormat
HDFS

FIAP


Nó 1
Nó 2
InputFormat
Partes
Partes
Partes
Record Readers
Record Readers
Record Readers
Map
Map
Map
Particionador
Shuffle
Ordenação
Reduce
OutputFormat
HDFS
InputFormat
Partes
Partes
Partes
Record Readers
Record Readers
Record Readers
Map
Map
Map
Particionador
Ordenação
Reduce
OutputFormat
HDFS

MapReduce Engine

FIAP



Exemplo: Contador de Palavras

- Incluindo as palavras dog, cat, mouse e hippo em um banco orientado a documentos.

```
db.items.insert({tags: ['dog', 'cat']})
db.items.insert({tags: ['dog']})
db.items.insert({tags: ['dog', 'mouse']})
db.items.insert({tags: ['dog', 'mouse', 'hippo']})
db.items.insert({tags: ['dog', 'mouse', 'hippo']})
db.items.insert({tags: ['dog', 'hippo']})
```

Fonte: https://docs.mongodb.org/manual/tutorial/map-reduce-Exemplos/

Exemplo: Contador de Palavras

- Criando o mapeamento

```
var map = function() {
   this.tags.forEach(function(t) {
      emit(t, {count: 1});
   });
}
```

Fonte: https://docs.mongodb.org/manual/tutorial/map-reduce-Exemplos/

Exemplo: Contador de Palavras

- Criando o reduce

```
var reduce = function(key, values) {
  var count = 0;
  for(var i=0, len=values.length; i<len; i++) {
    count += values[i].count;
  }
  return {count: count};
}
```

Fonte: https://docs.mongodb.org/manual/tutorial/map-reduce-Exemplos/

Exemplo: Contador de Palavras

- Executando o MapReduce

```
var result = db.items.mapReduce(map, reduce);
result
{
  "ok"          : 1,
  "timeMillis" : 86,
  "result"      : "tmp.mr.mapreduce_1273861517_683",
  "counts"      : {
    "input"     : 6,
    "emit"      : 13,
    "output"    : 4
  }
}
```

Fonte: https://docs.mongodb.org/manual/tutorial/map-reduce-Exemplos/

Exemplo: Contador de Palavras

- Executando o MapReduce

```
> db[result.result].find()
{ "_id" : "cat",   "value" : { "count" : 1 } }
{ "_id" : "dog",   "value" : { "count" : 6 } }
{ "_id" : "hippo", "value" : { "count" : 3 } }
{ "_id" : "mouse", "value" : { "count" : 3 } }
```

Fonte: https://docs.mongodb.org/manual/tutorial/map-reduce-Exemplos/

Executando um Mapper em Python

```
echo "foo foo quux labs foo bar quux" | python mapper.py
```

```
foo  1
foo  1
quux 1
labs 1
foo  1
bar  1
quux 1
```

Executando um Reducer em Python

```
echo "foo foo quux labs foo bar quux" |
  python mapper.py |
  sort -k1,1 |
  python reducer.py
```

```
bar     1
foo     3
labs    1
quux    2
```

Executando um Reducer em Python

```
cat armas.txt |
python mapper.py |
sort -k1,1 |
python reducer.py
```

```
Uns     3
urdia   1
urdido  1
usadas  1
```

Submetendo um job MapReduce

- WordCount.java
  - Classe de drive do MapReduce para executar o job
- WordMapper.java
  - Uma classe mapper para emitir para emitir palavras
- SumReducer.java
  - Uma classe reducer para contar palavras

https://www.dropbox.com/sh/bsopakcxg7wctfu/AAA-r0_LQkT0ze54KnzYzhDMa?dl=0