INDUSTRIA 4.0
Processamento de Big Data

FONTE: DELIRIUM CAFE

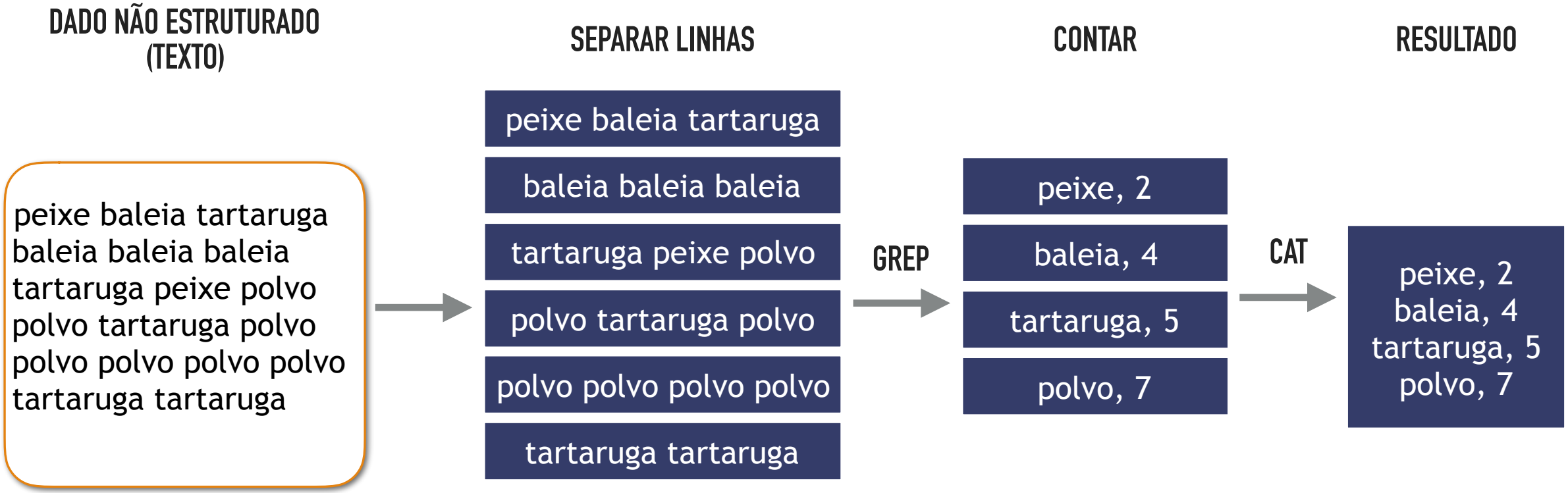
Aula #10 - Processamento distribuído de dados

EDUARDO CUNHA DE ALMEIDA

Agenda

- Divisão do problema
- MapReduce
- Hadoop
- SQL-on-Hadoop: Hive

Divisão do problema (conta palavras)



Landscape de Proc. Distribuído

FONTE: MATTTURCK.COM

MapReduce

MapReduce é um modelo de programação para simplificar o processamento de dados distribuídos.

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

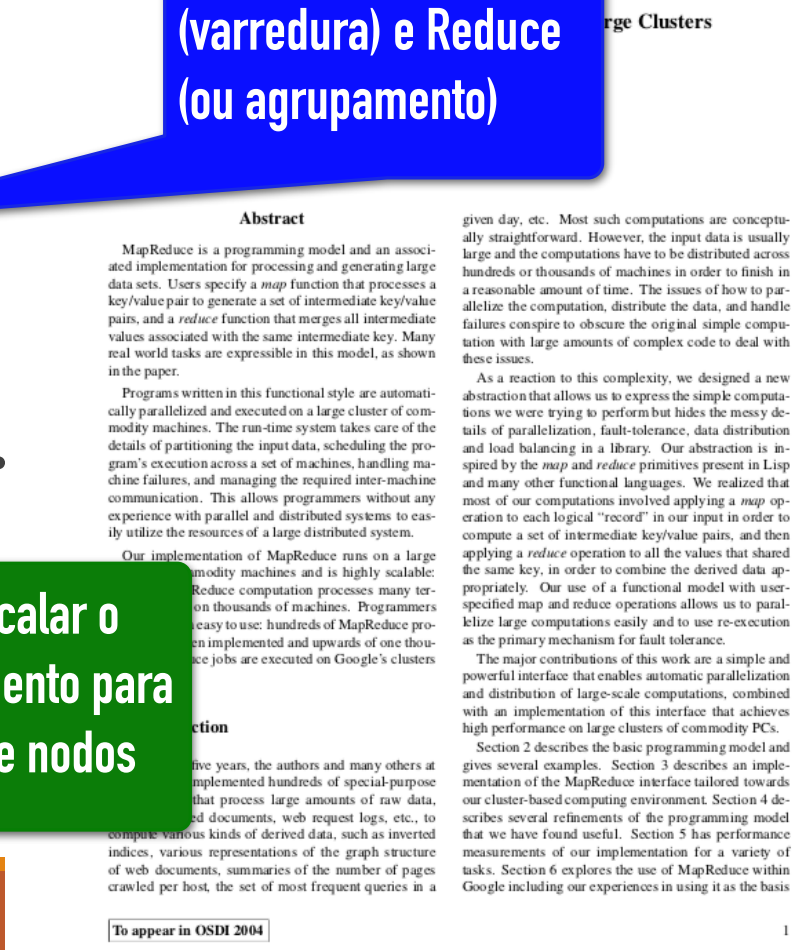
MapReduce

MapReduce é um modelo de programação para simplificar o processamento de dados distribuídos.

Baseado em 2 primitivas Map (varredura) e Reduce (ou agrupamento)

Esconde problemas de distribuição da computação

Permite escalar o processamento para milhares de nodos



rge Clusters

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
modity machines and is highly scalable:
Reduce computation processes many ter-
on thousands of machines. Programmers
easy to use: hundreds of MapReduce pro-
n implemented and upwards of one thou-
ce jobs are executed on Google's clusters

ction

ive years, the authors and many others at
mplemented hundreds of special-purpose
that process large amounts of raw data,
d 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

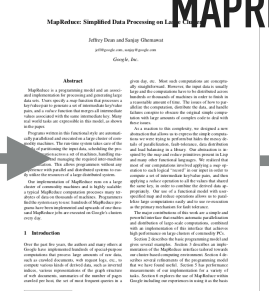
1

Histórico do MapReduce

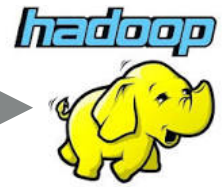
ARTIGO GOOGLE FILE SYSTEM

2003

ARTIGO MAPREDUCE



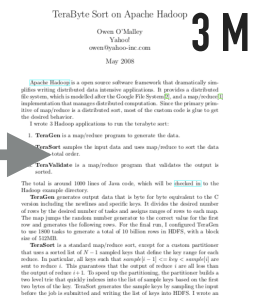
2004



YAHOO! 600 NODOS

2006

TERASORT BENCHMARK 3 MIN 48 SEC



2008

2009-11

2011

YAHOO! 42K NODOS HADOOP

2014

FINANCIAMENTO 110$ MI

2018

VERSÃO 3.1

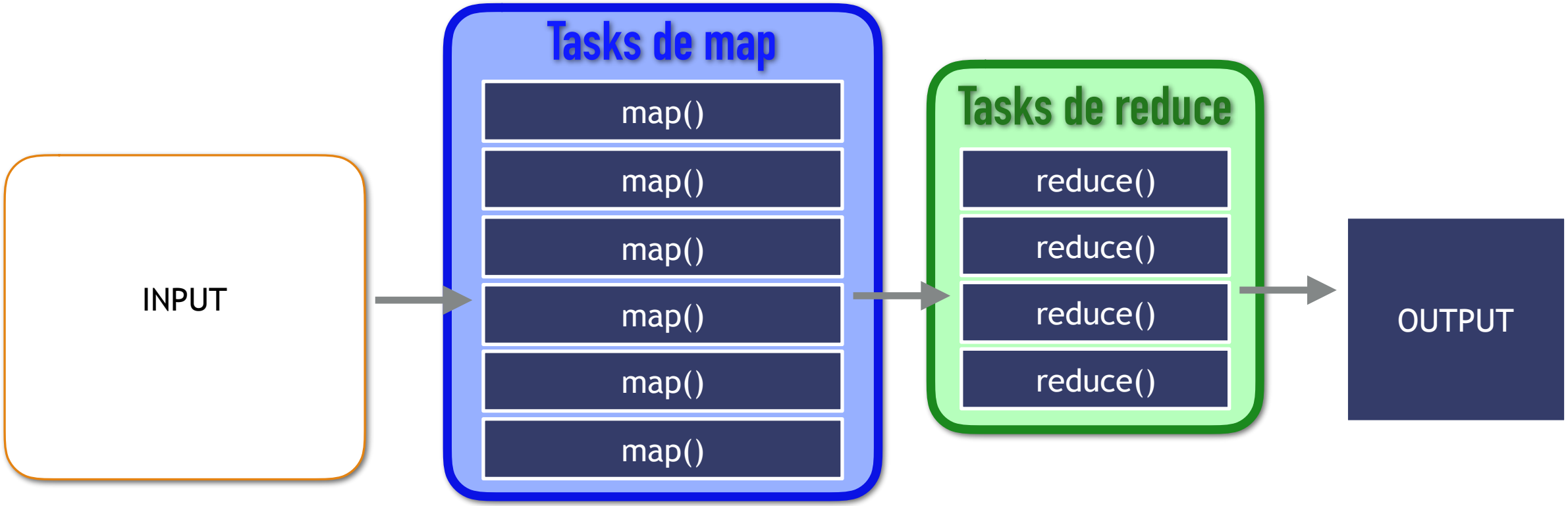
Cluster Hadoop na Yahoo

FONTE: HTTPS://WIKI.APACHE.ORG/HADOOP/POWEREDBY#Y

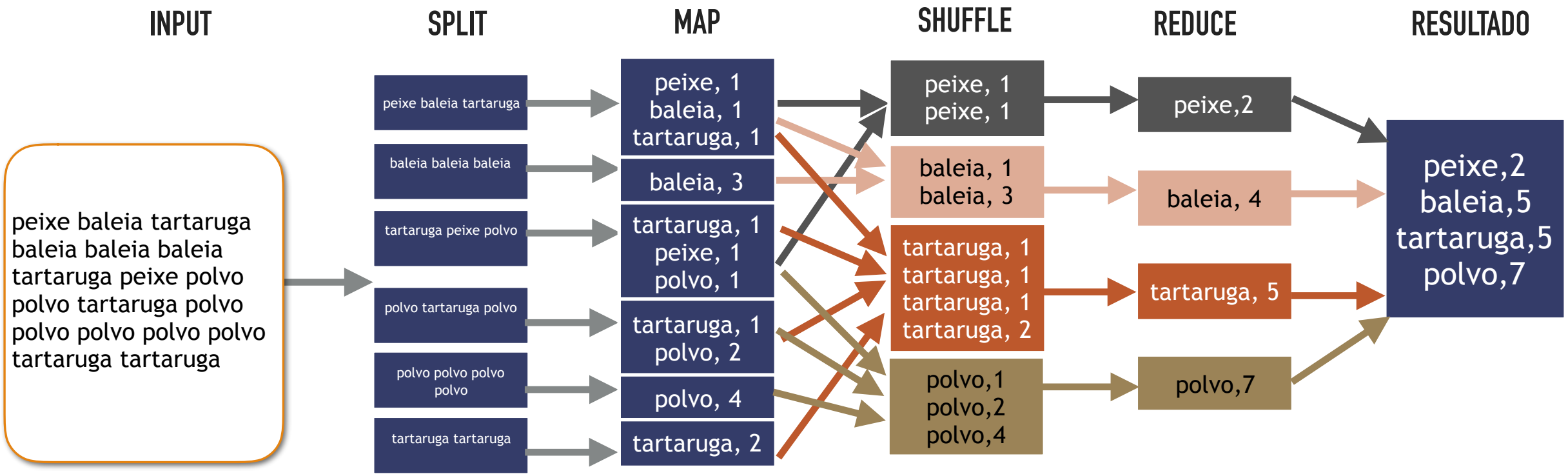
Visão geral do MapReduce

- Recebe entrada de dados (geralmente grande)
- **Map**: extrai “algo” linha por linha
- Embaralha e ordena
- **Reduce**: agrega, sumariza, filtra
- Escreve o resultado

MapReduce (Job)



MapReduce: Conta palavras completo



MapReduce: Conta palavras completo

MAP

peixe, 1
baleia, 1
tartaruga, 1

baleia, 3

tartaruga, 1
peixe, 1
polvo, 1

tartaruga, 1
polvo, 2

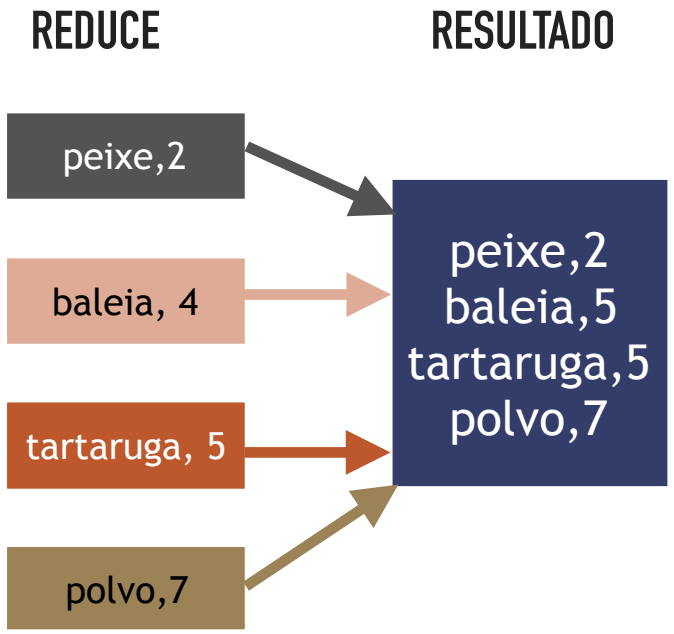
polvo, 4

tartaruga, 2

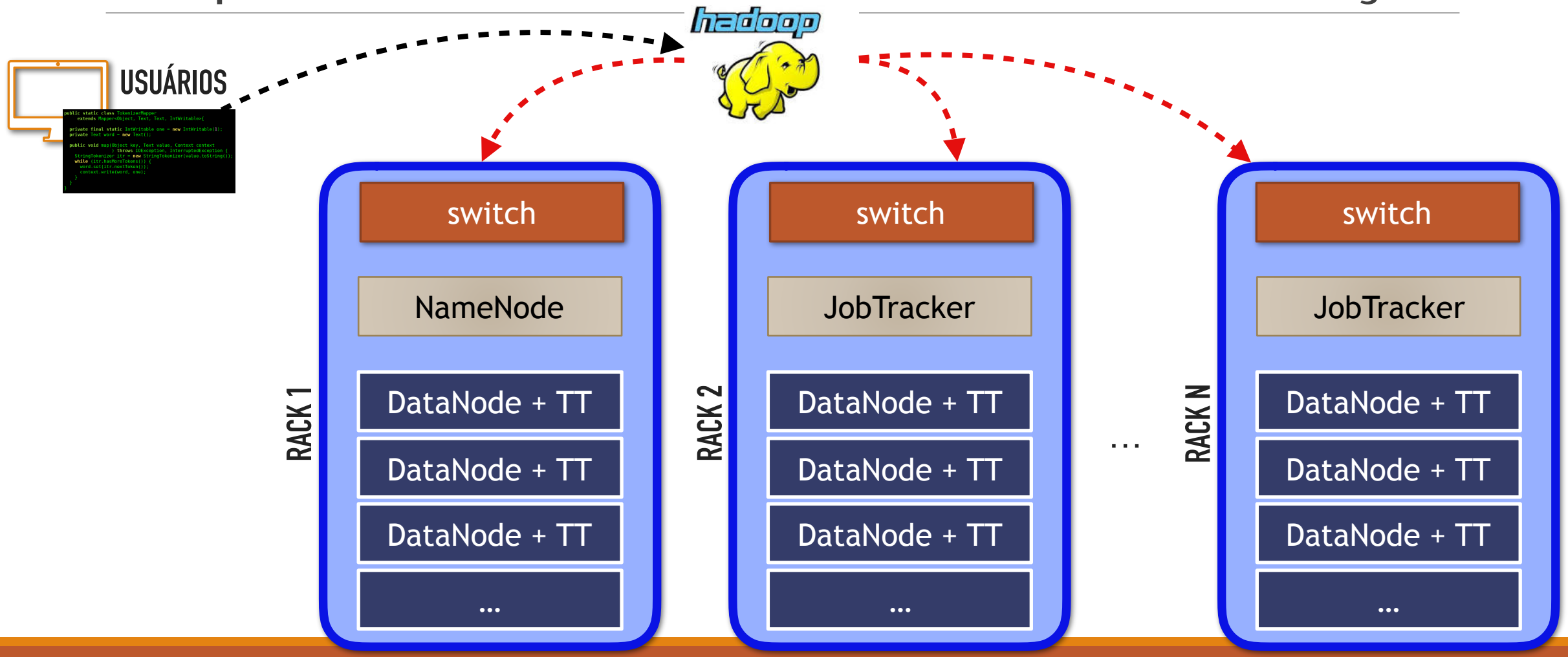
```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

MapReduce: Conta palavras completo

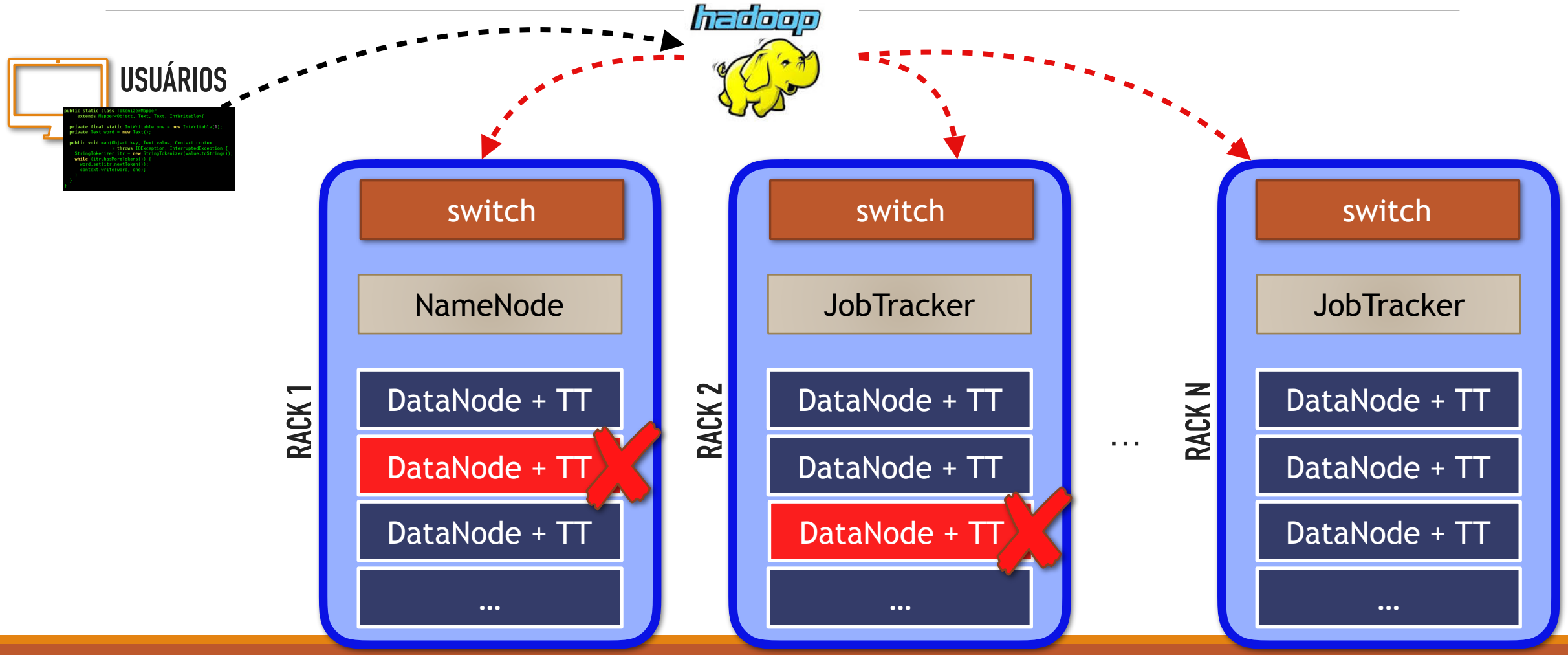
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```



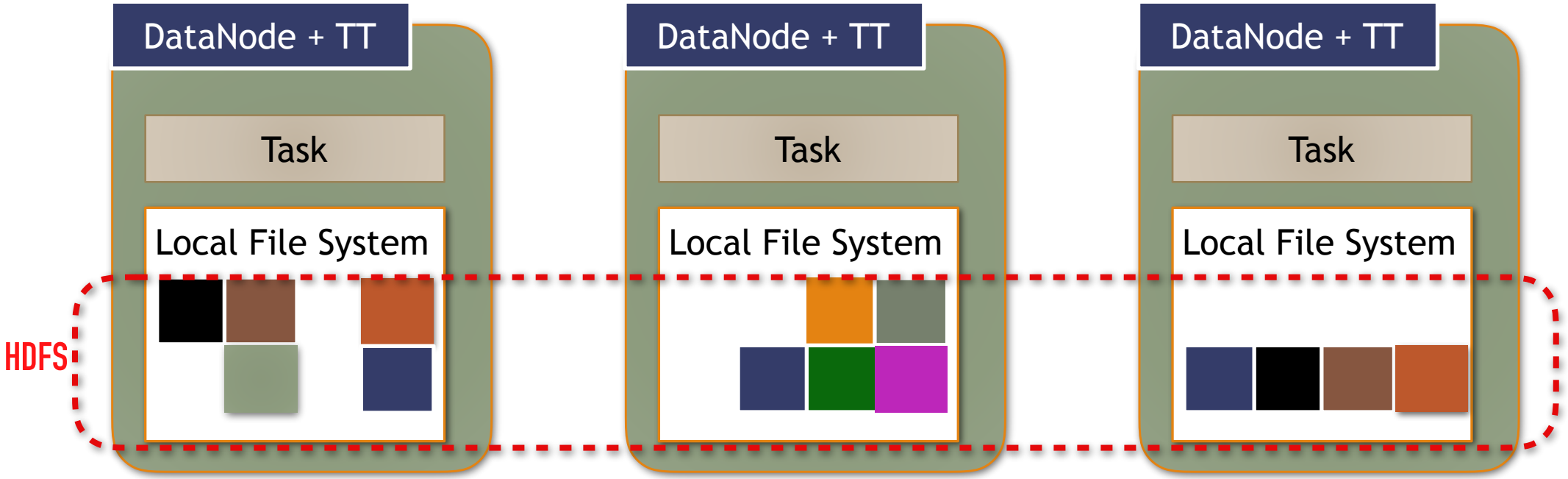
MapReduce: mecanismo de execução



… funciona mesmo com falhas!!

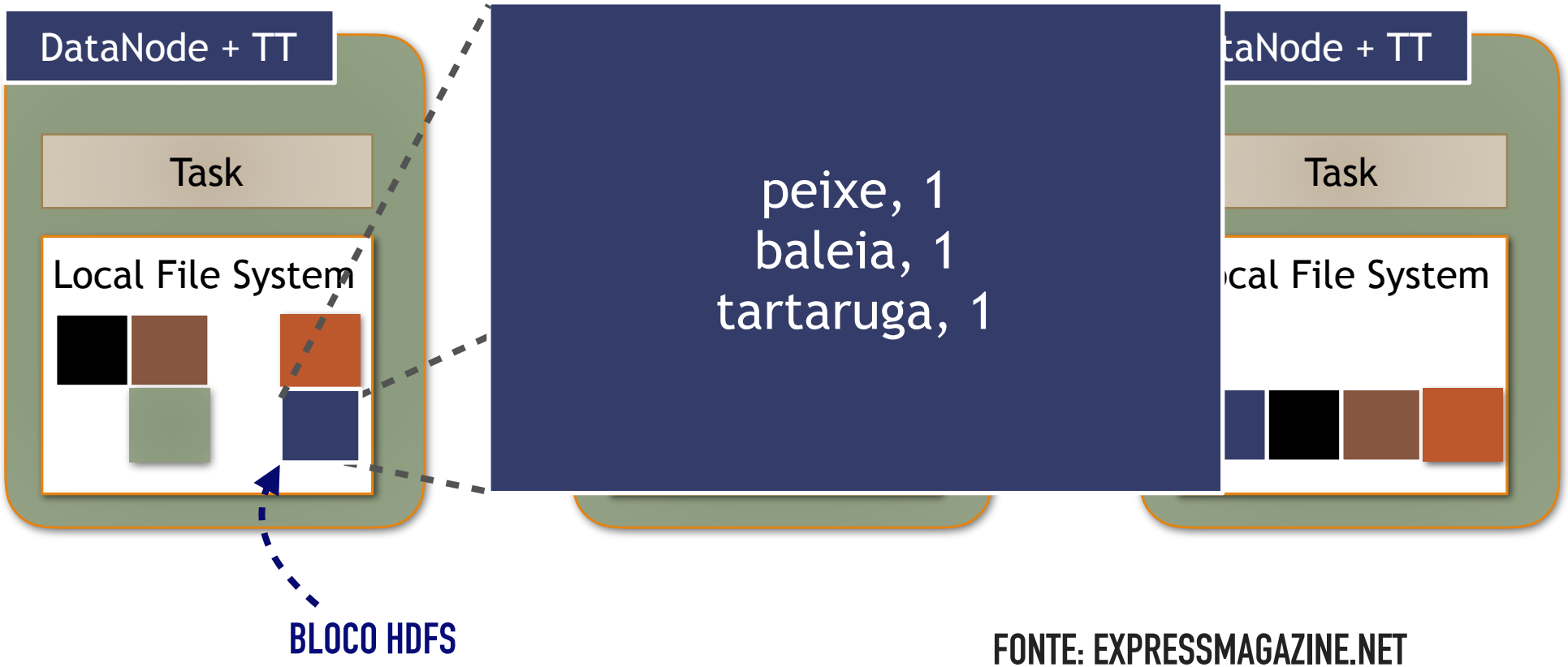


MapReduce: mecanismo de execução



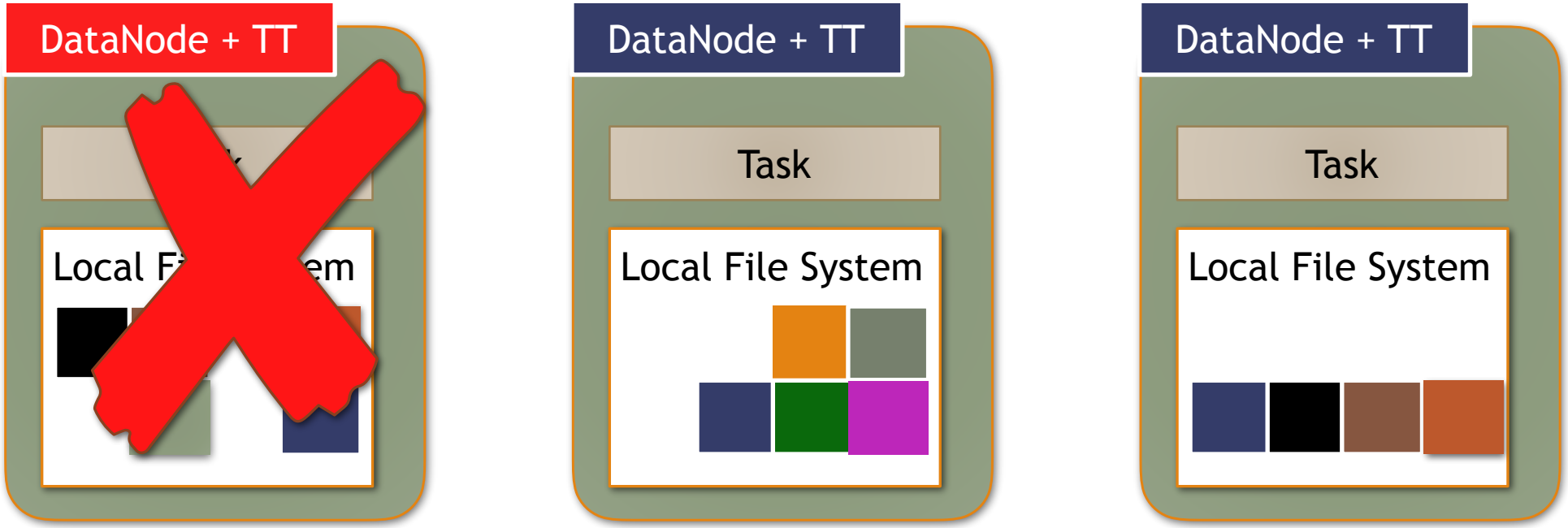
FONTE: EXPRESSMAGAZINE.NET

MapReduce: mecanismo de execução



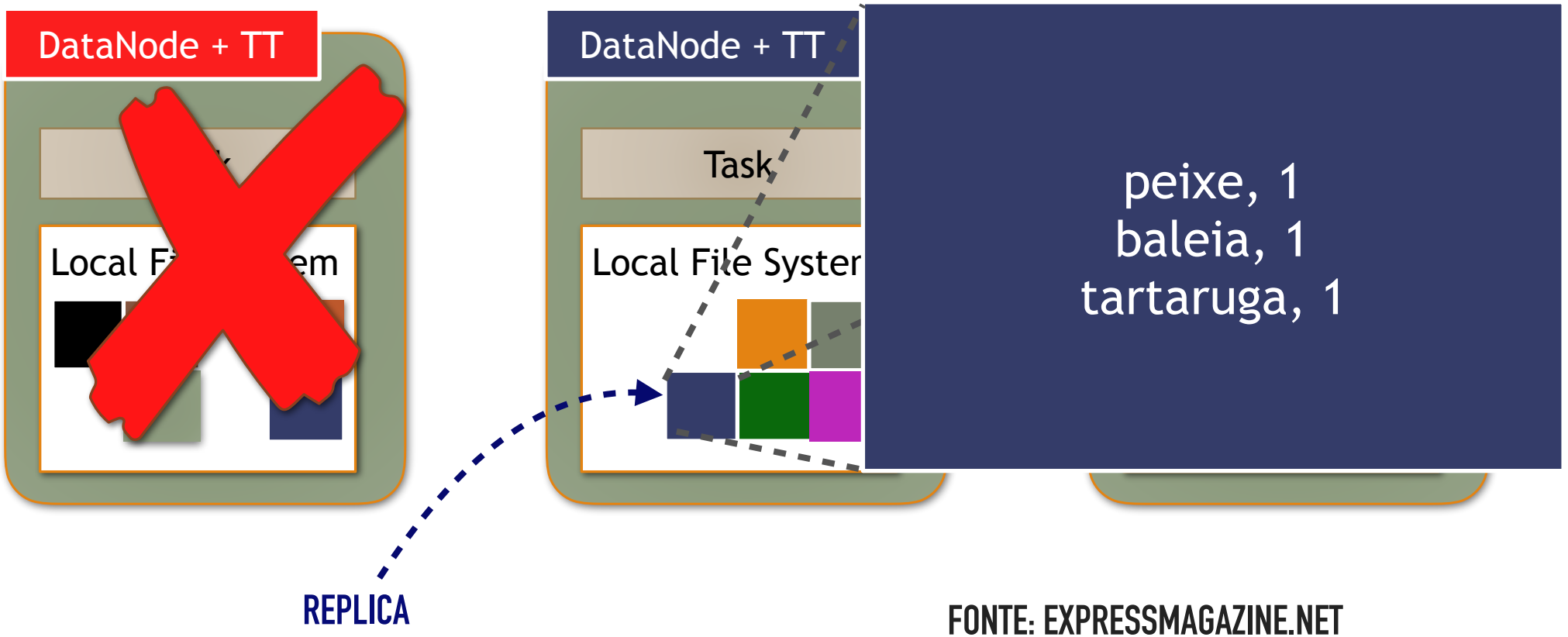
FONTE: EXPRESSMAGAZINE.NET

MapReduce: mecanismo de execução



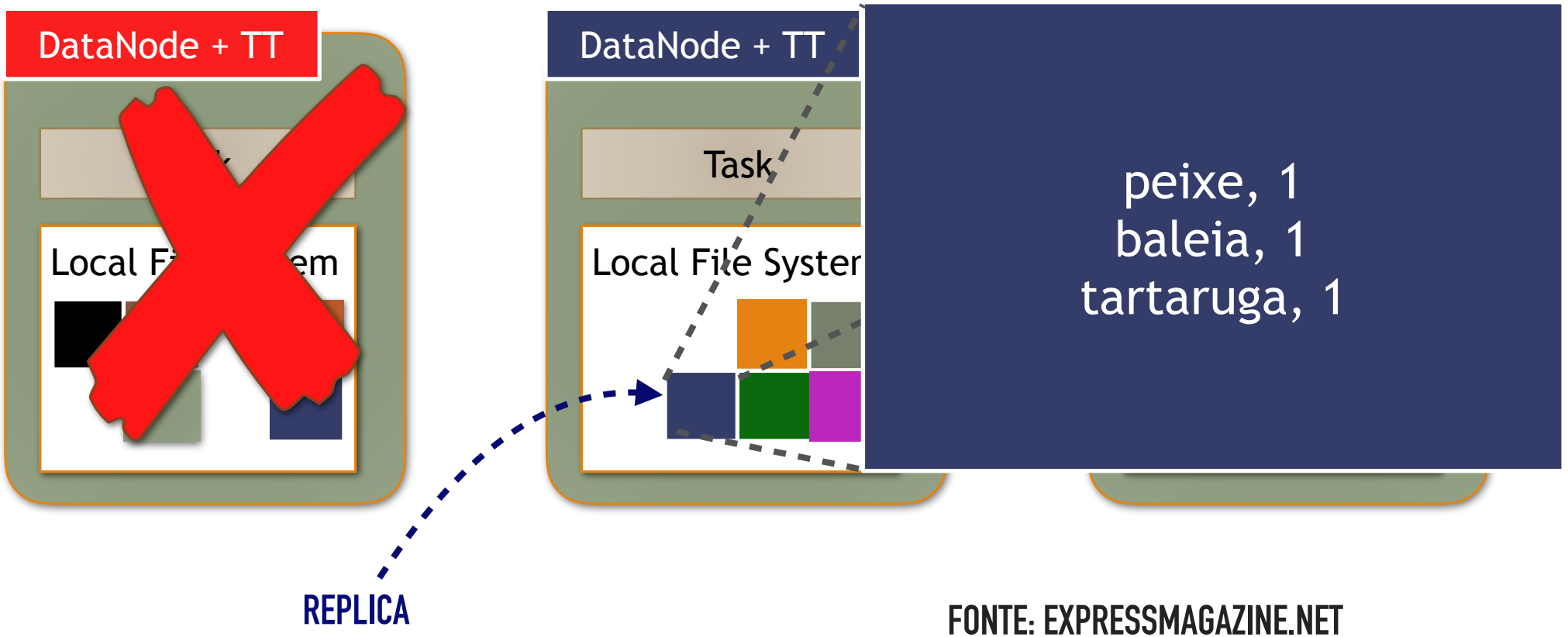
FONTE: EXPRESSMAGAZINE.NET

MapReduce: mecanismo de execução



FONTE: EXPRESSMAGAZINE.NET

MapReduce: mecanismo de execução



FONTE: EXPRESSMAGAZINE.NET

Qual linguagem programar em MR?

JAVA-LIKE

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr
      context.writ
    }
  }
}
```

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

VS.

SQL-LIKE

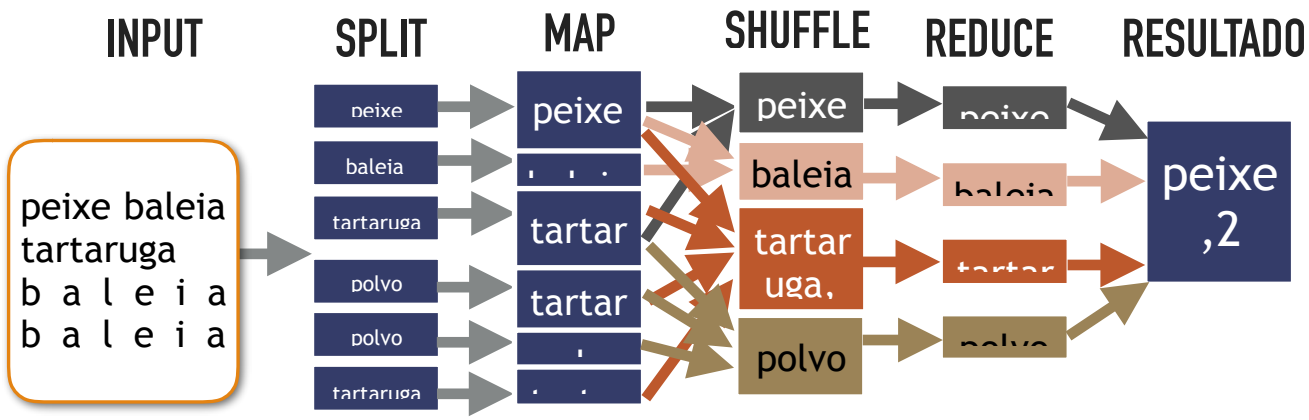
```
SELECT
        T2.UF, AVG(T1.IDADE)
FROM T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```

SQL-on-Hadoop

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text

  public void map(Object key,
                  ) throws IOE
    StringTokenizer itr = new
    while (itr.hasMoreTokens()
      word.set(itr.nextToken()
      context.write(word, one)
    }
  }
}
```

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values
                     Context context
                     ) throws IOException, InterruptedExcep
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

```
SELECT
        T2.UF, AVG(T1.IDADE)
FROM T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```



SQL

SQL-on-Hadoop


```
SELECT
        T2.UF,  AVG(T1.IDADE)
FROM  T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```

M = MAP

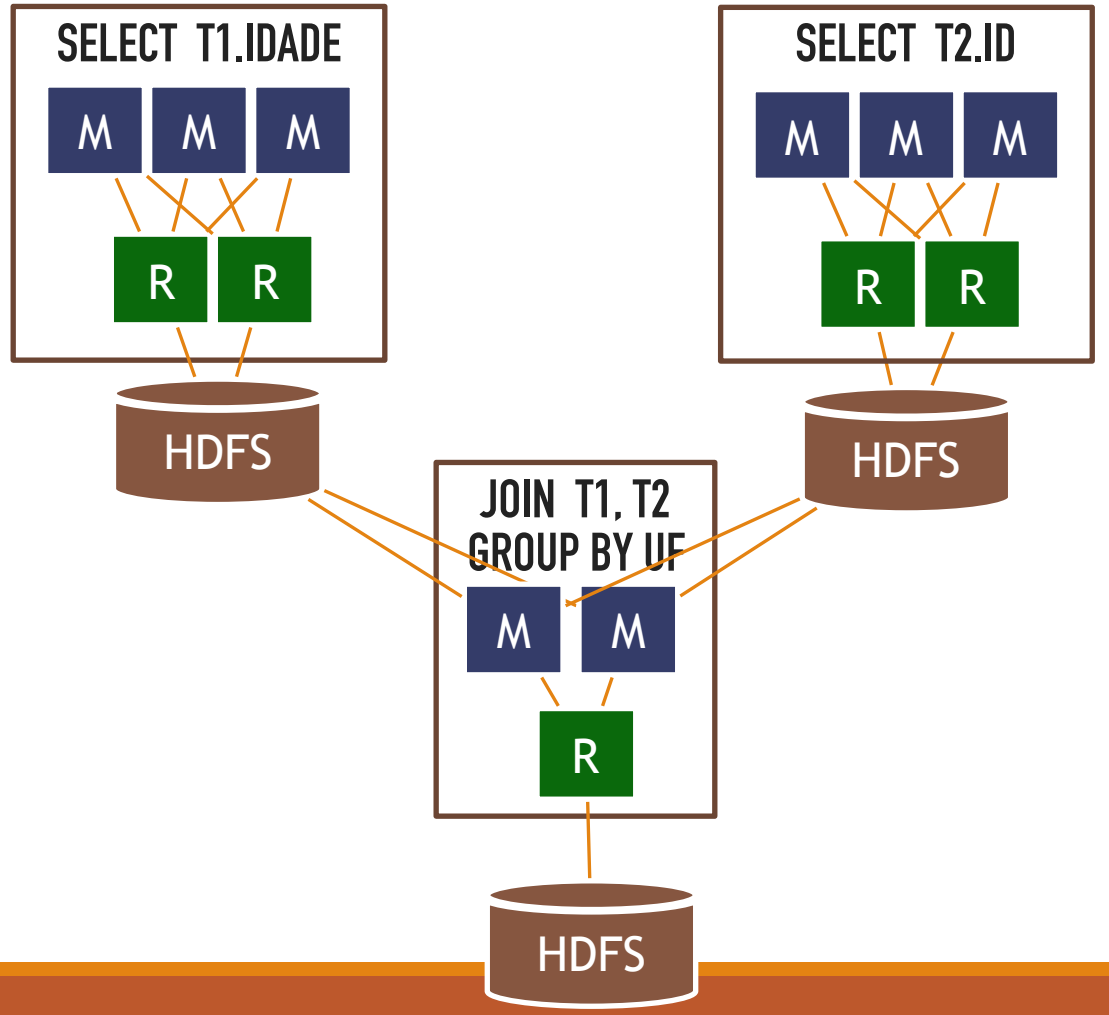
R = REDUCE

SQL-on-Hadoop

```
SELECT
        T2.UF,  AVG(T1.IDADE)
FROM  T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```

M = MAP

R = REDUCE

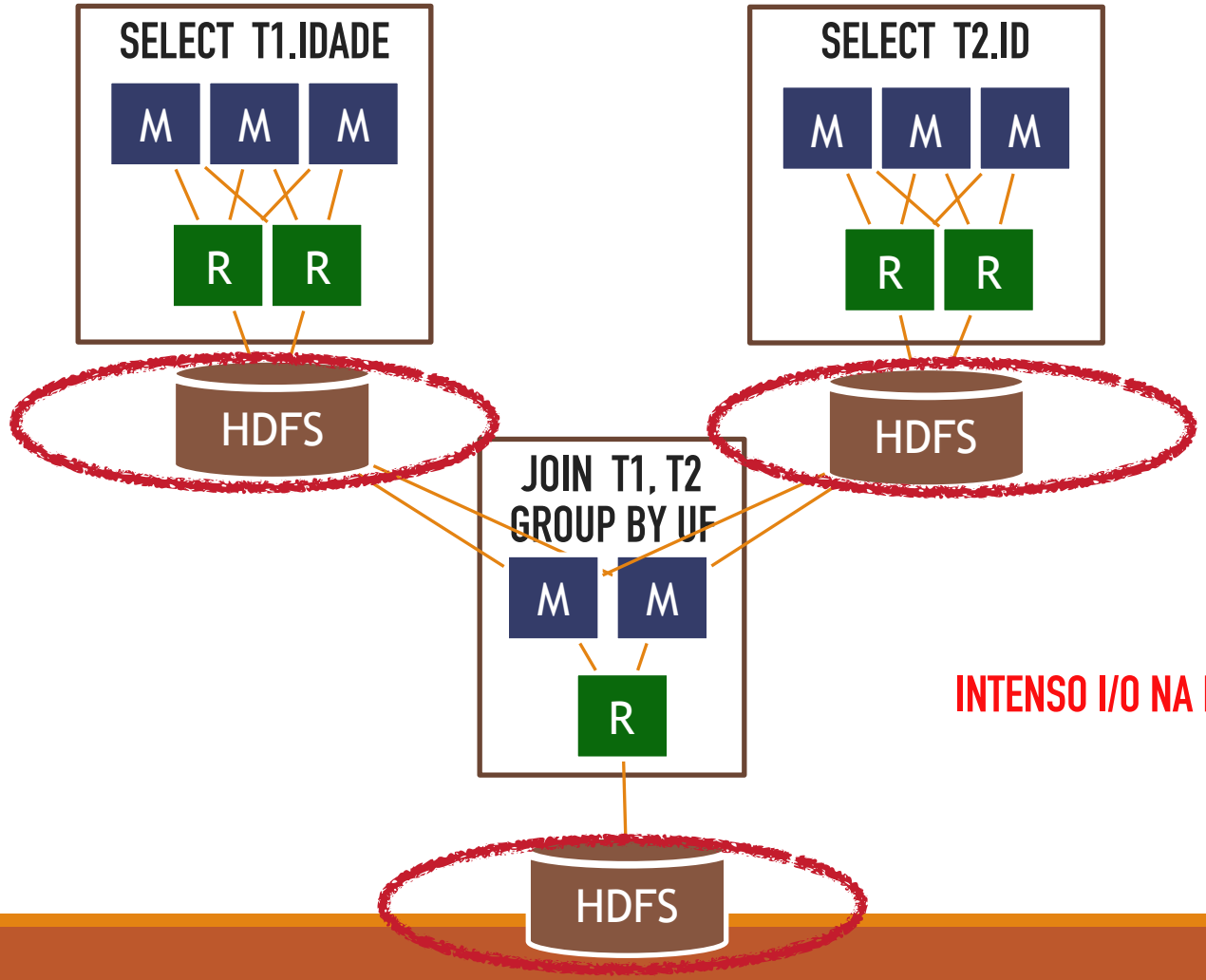


SQL-on-Hadoop

```
SELECT
        T2.UF,  AVG(T1.IDADE)
FROM  T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```

M = MAP

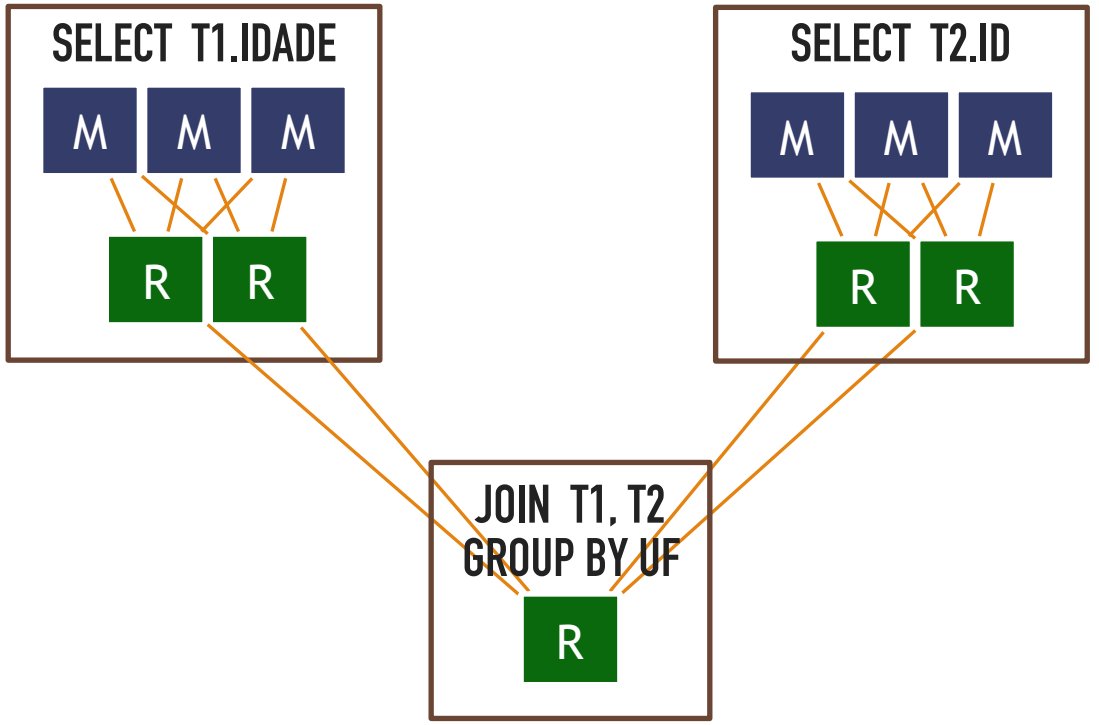
R = REDUCE



INTENSO I/O NA HDFS

SQL-on-Hadoop

```
SELECT
        T2.UF,  AVG(T1.IDADE)
FROM  T1 JOIN T2 ON (T1.ID = T2.ID)
GROUP BY T2.UF;
```



M = MAP

R = REDUCE

Várias linguagens são também ofertadas:

- Python
- Scala
- Java
- C
- PIG Latin
- …

INDUSTRIA 4.0
Processamento de Big Data

FONTE: DELIRIUM CAFE

Aula #10 - Processamento distribuído de dados

EDUARDO CUNHA DE ALMEIDA