Doutorado e Mestrado em Ciência da Computação

Instituto de Informática

Programação Paralela e Distribuída

Prof. Dr. Sérgio T. Carvalho

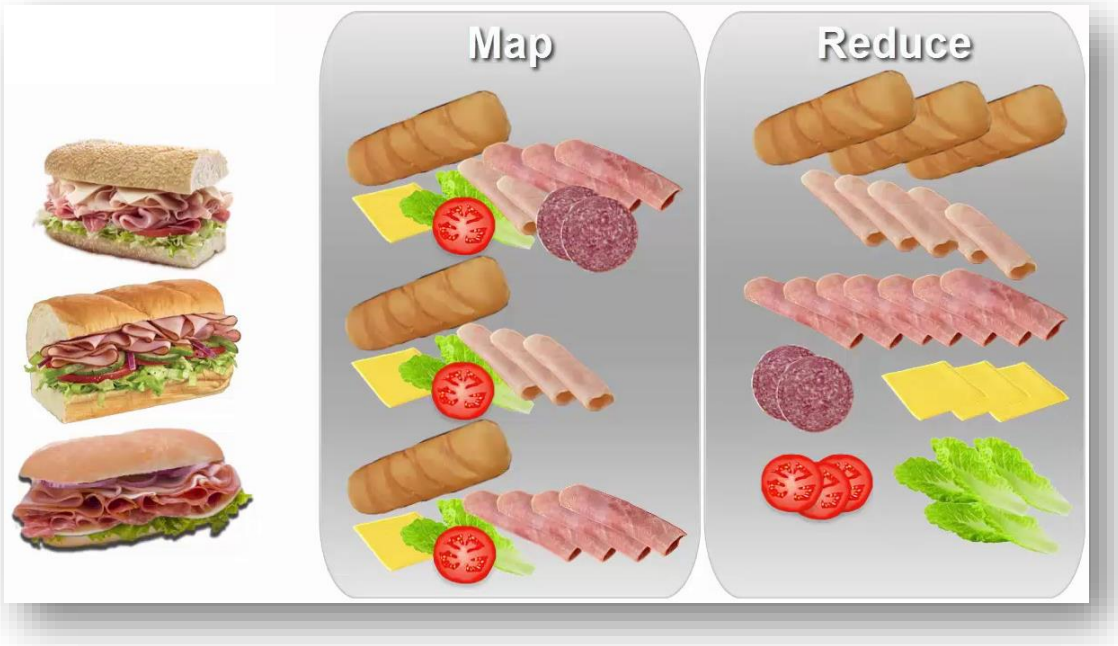
MapReduce:
Processamento simplificado de dados em grandes clusters

Marcos Alves Vieira

Goiânia, 09 de julho de 2019

Sumário

- Introdução
- MapReduce
- *Frameworks* MapReduce
- Exemplo de Implementação
- Conclusão
- Referências



Introdução

- Big Data
  - Análise de grandes quantidades de informações em **tempo viável**
    - Analogia: contar as laranjas maduras de um gigantesco cesto

- Exploração de paradigmas de programação paralela e processamento distribuído
  - Desenvolver softwares para ambientes distribuídos é uma **tarefa complexa**
    - Envolve **conceitos e problemas** que devem ser considerados
      - Concorrência
      - Tolerância a falhas
      - Distribuição de dados
      - Balanceamento de carga

MapReduce

- Modelo de **programação paralela** para **processamento distribuído** de **grandes volumes de dados**
- Proposto pela Google em 2004
- Adequado para problemas que podem ser particionados em **subproblemas**
  - O **processamento** dos dados é **distribuído** em diversas máquinas, porém **em conjuntos de dados diferentes**
  - Cada máquina é responsável por **processar completamente grupos de dados**, ao invés de processar todos os dados em uma determinada etapa

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
Sistema de Arquivos Distribuídos

- Sistema de arquivos distribuídos:
  - Principal responsável pelo sucesso do MapReduce
  - Possui as mesmas características de um sistema de arquivos tradicional
  - Permite o **armazenamento** e **compartilhamento de arquivos** entre diversos computadores
  - Permite a manipulação dos arquivos de forma transparente
  - Permite escalabilidade
- A implementação original do MapReduce da Google utilizava o sistema de arquivos distribuído GFS

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {sanjay,hgobioff,shuntak}@google.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

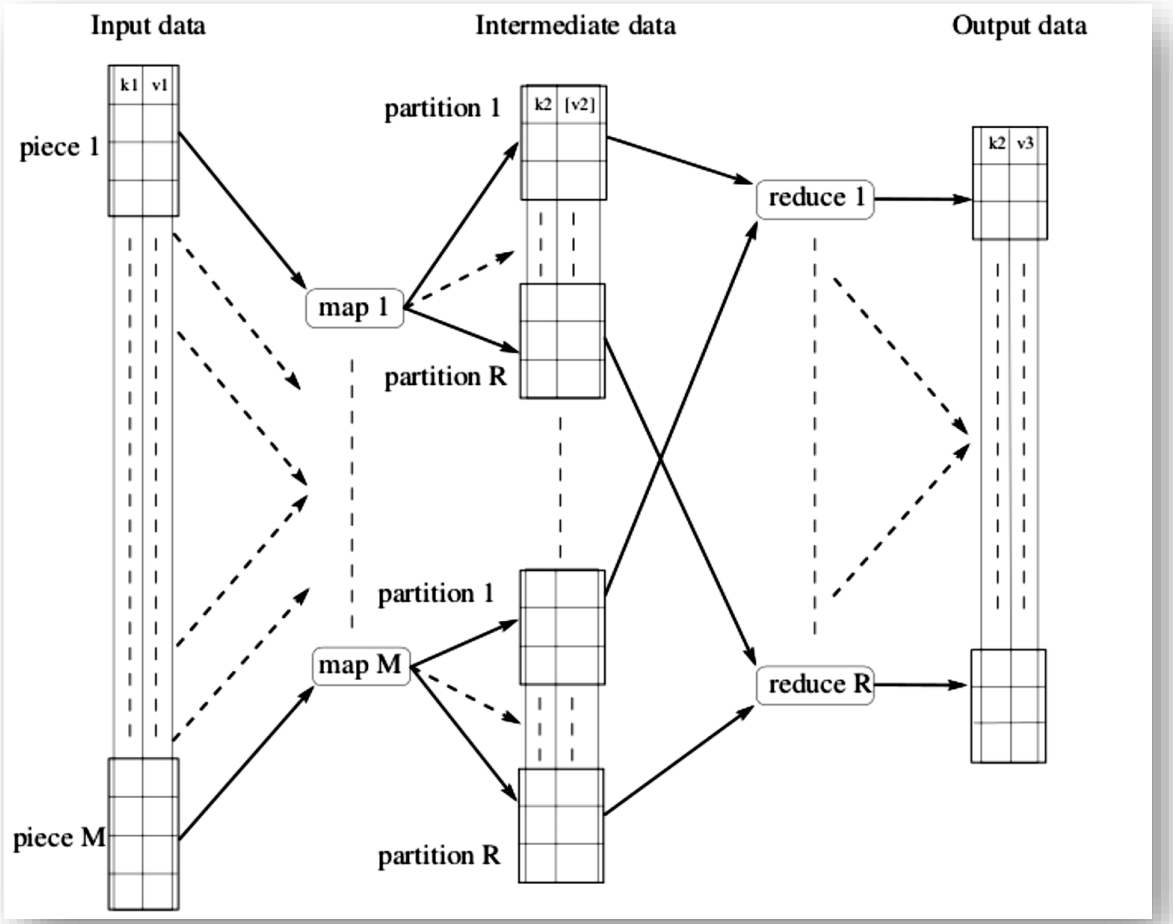
MapReduce
Funcionamento

- Busca abstrair toda a complexidade de paralelização de uma aplicação usando apenas as funções `Map` e `Reduce`

- Funcionamento do MapReduce:
  - Função `Map`
    - **Mapeia** um conjunto de dados em uma coleção de tuplas `<chave, valor>`
    - Entrada: blocos do(s) arquivo(s) armazenado(s) no sistema de arquivos distribuído
    - Saída: tuplas `<chave, valor>`
  - Função `Reduce`
    - **Reduz** as tuplas com a mesma chave, produzindo a saída final do processamento
    - Agrupa as tuplas com chaves em comum

MapReduce
Funcionamento

- Toda a computação distribuída é controlada pelo ***framework* MapReduce**
  - Sistema de arquivos distribuído
  - Protocolos de comunicação e troca de mensagens
  - O escalonador escolhe quais máquinas executarão as tarefas
  - O usuário define quais dados das entradas serão usados como chaves e valores

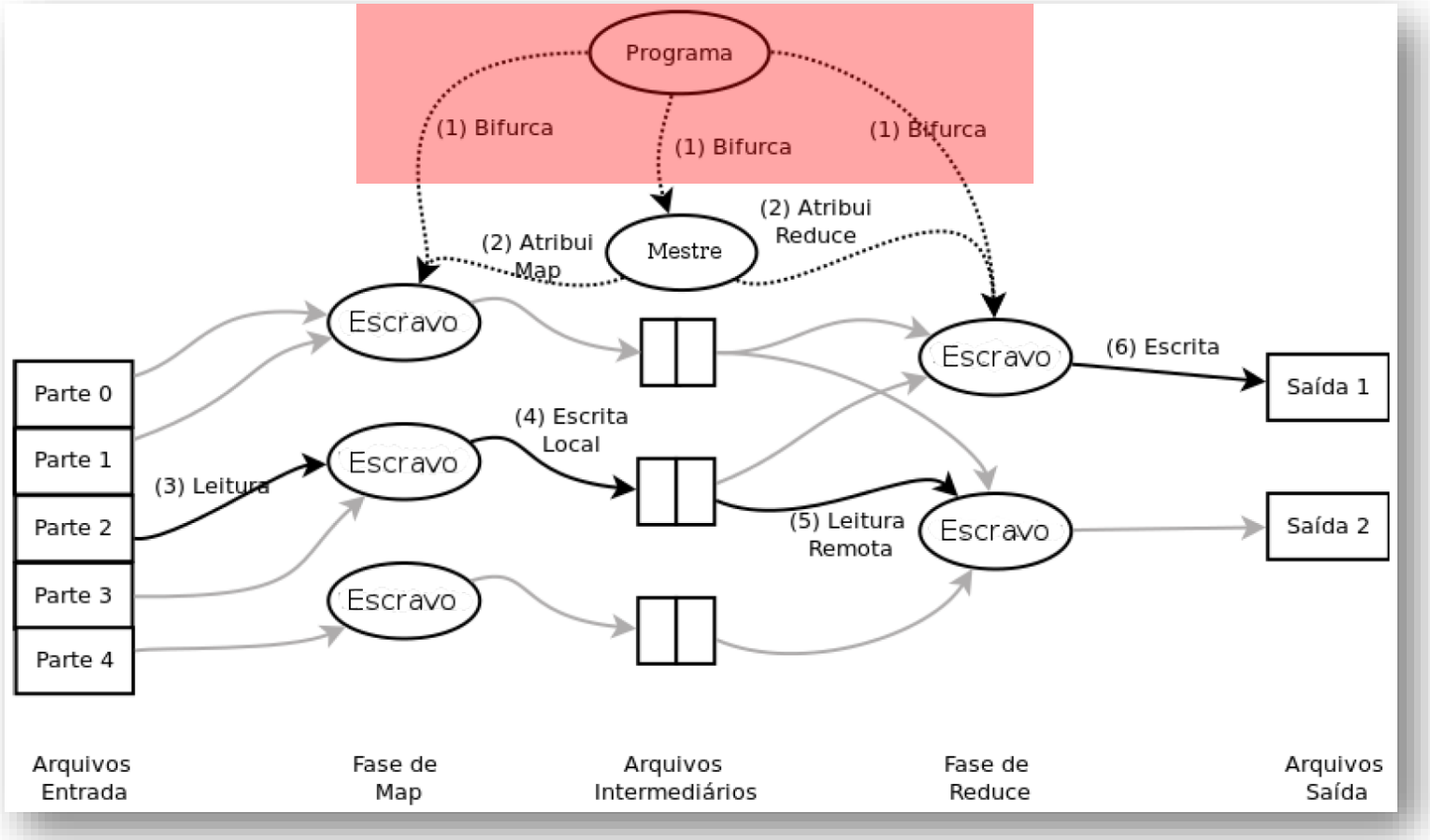


Exemplo de Implementação

- **Problema**
  - Calcular o somatório de recebimentos do Programa Bolsa Família em cada estado brasileiro em um determinado mês
- **Base de dados**
  - Obtida por meio do Portal da Transparência do Governo Federal
  - Arquivo com os pagamentos do Programa Bolsa Família referentes ao mês de Abril/2019
  - Possui 14.207.771 registros em um arquivo CSV de 1,26 GB
- **Implementação**
  - Oracle VM VirtualBox 6.0
    - Sistema operacional CentOS 7 64 bits
    - Apache Hadoop 3.1.1
    - Painel de controle Ambari 2.7.3

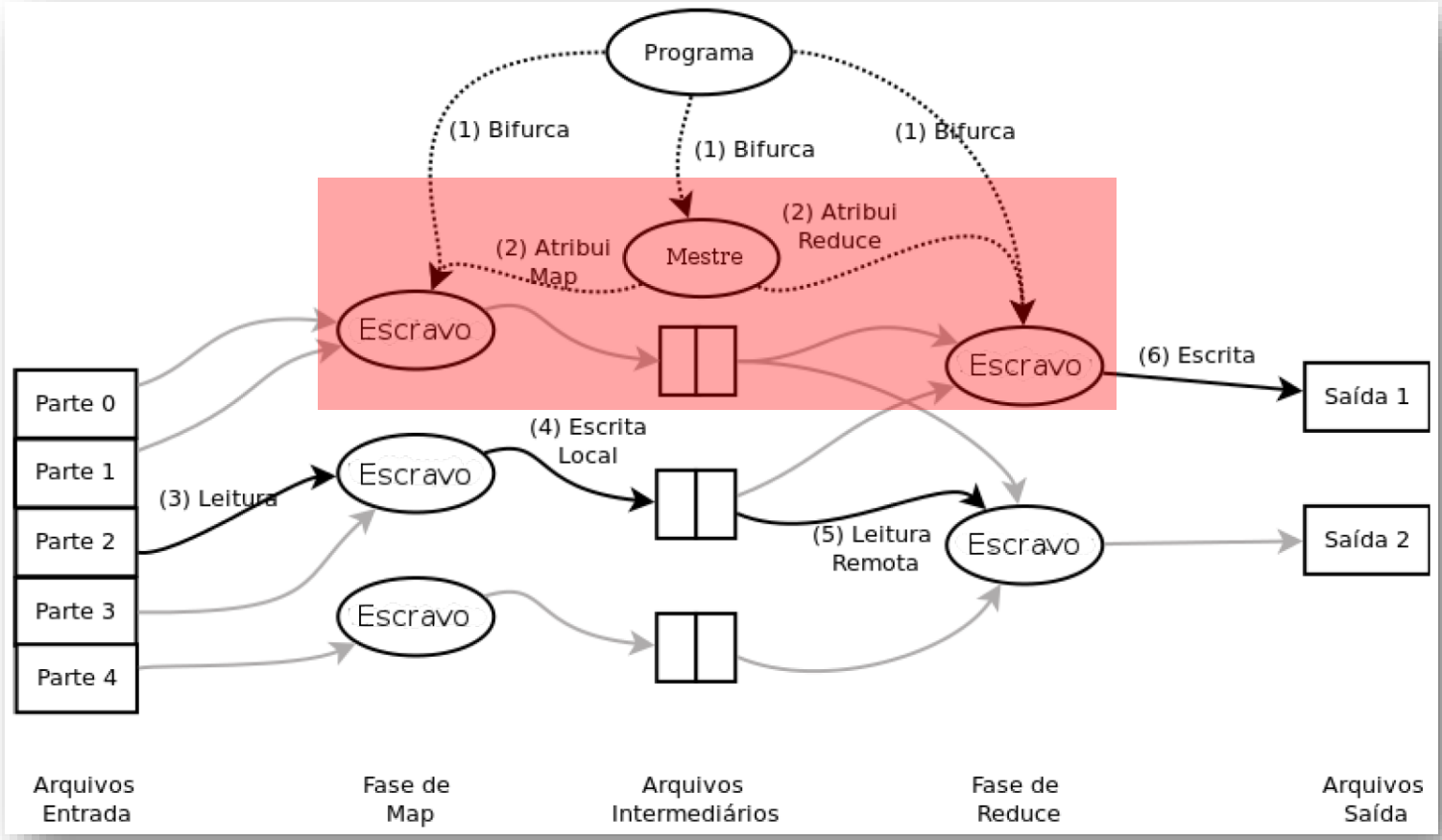
Demonstração

MapReduce
Modelo de Programação



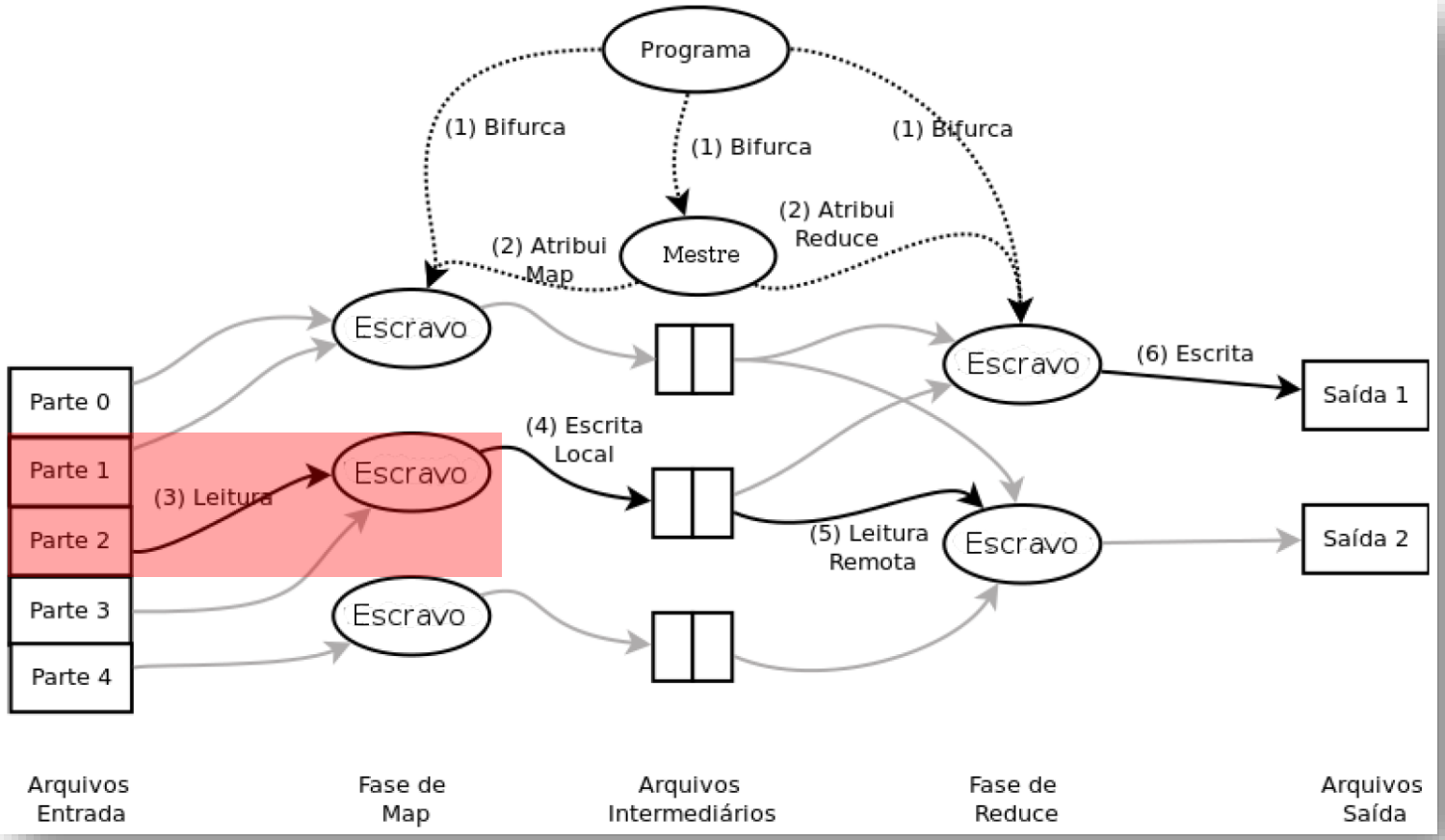
1. O *framework* divide os dados de entrada e inicia cópias do programa nas máquinas do cluster computacional.

MapReduce
Modelo de Programação



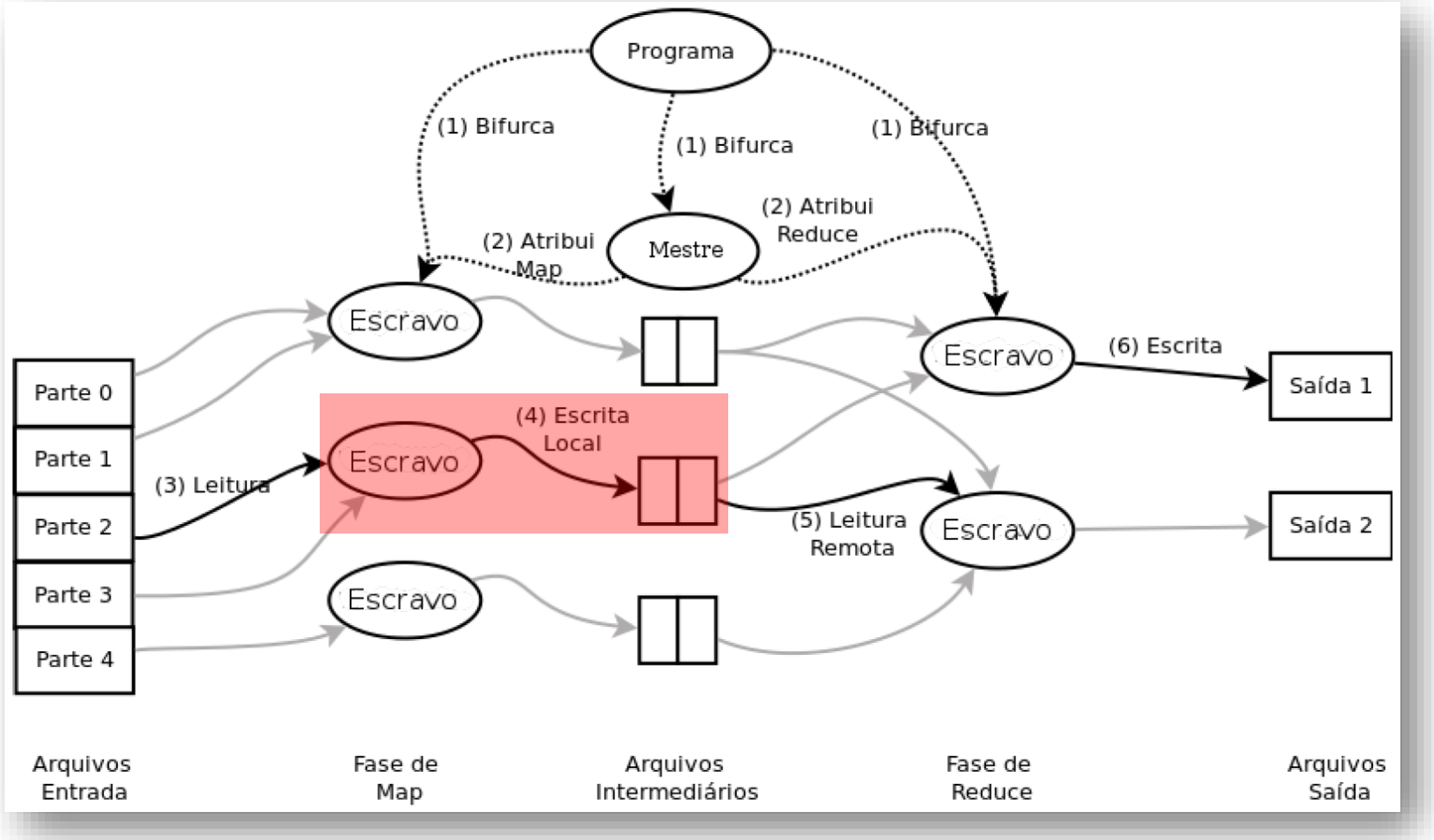
2. Uma das cópias do programa é o **Mestre** e as demais são **Escravos**. O Mestre é responsável por atribuir as tarefas de **mapeamento** e **redução** aos Escravos.

MapReduce
Modelo de Programação



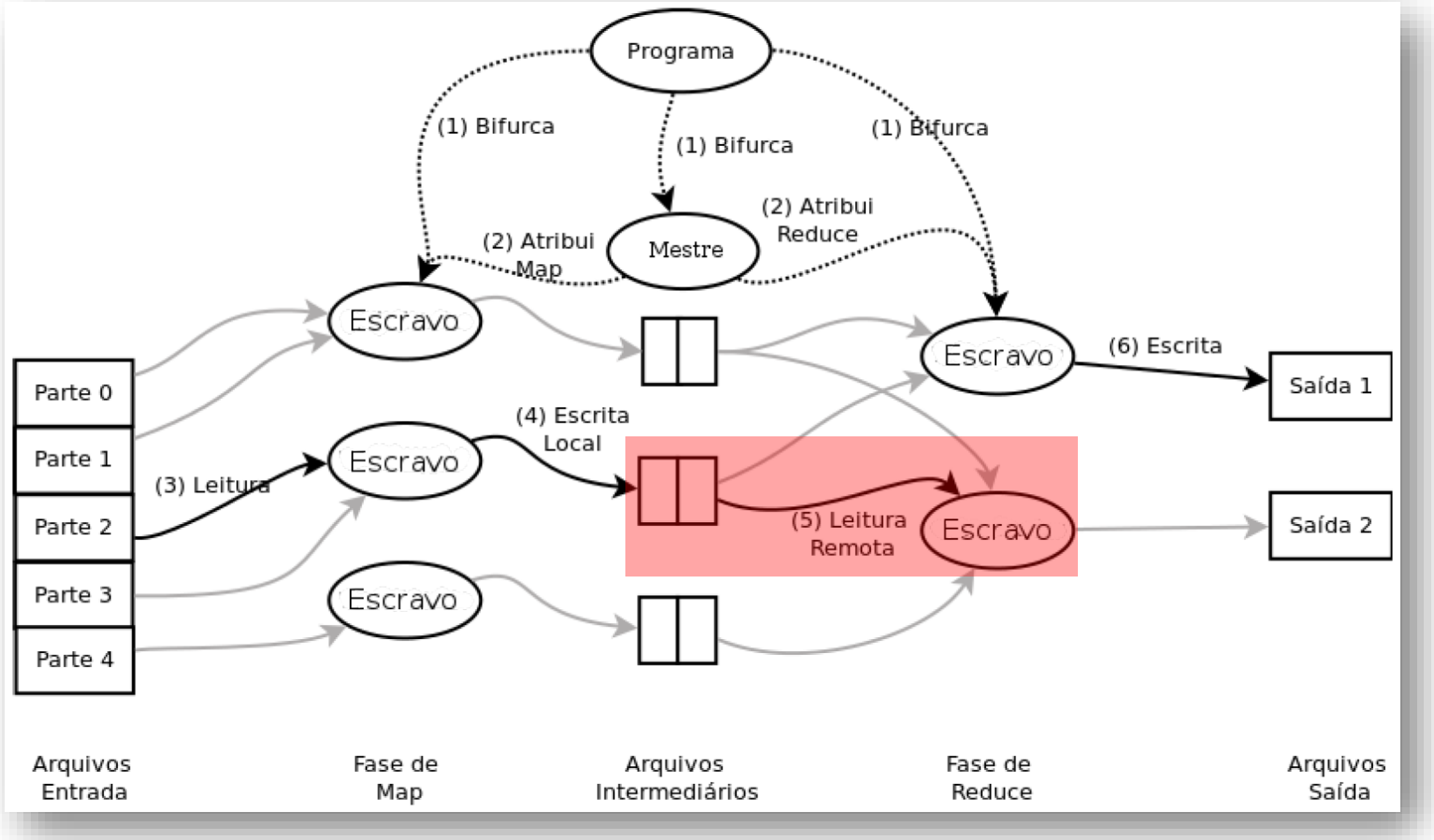
3. Na função de **mapeamento**, o Escravo lê a entrada e separa as tuplas `<chave, valor>`, armazenando-as em memória.

MapReduce
Modelo de Programação



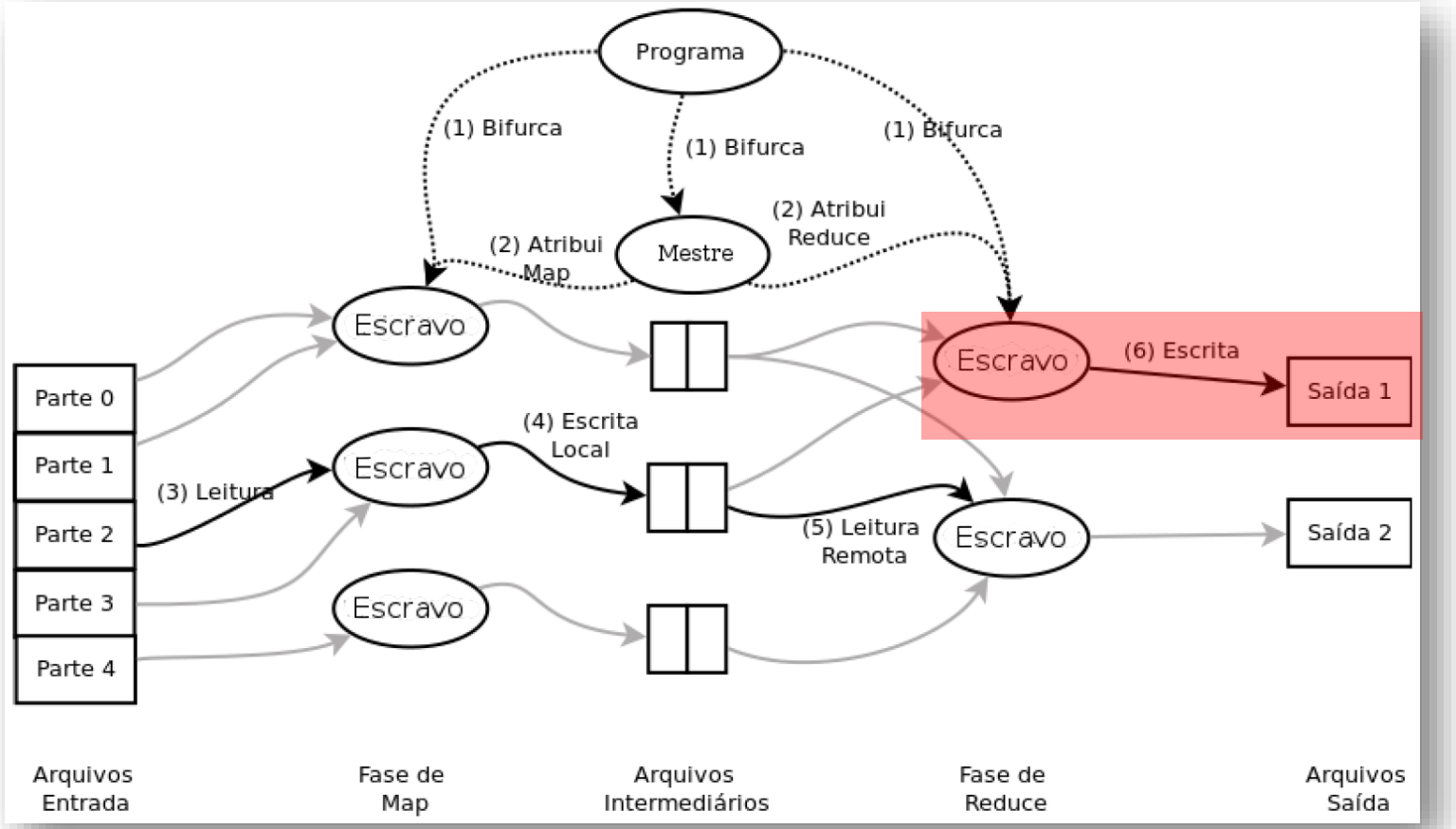
4. Periodicamente as tuplas são escritas em disco. A localização dos blocos é passada ao Mestre, que repassa aos Escravos, os quais realizarão a **redução**.

MapReduce
Modelo de Programação



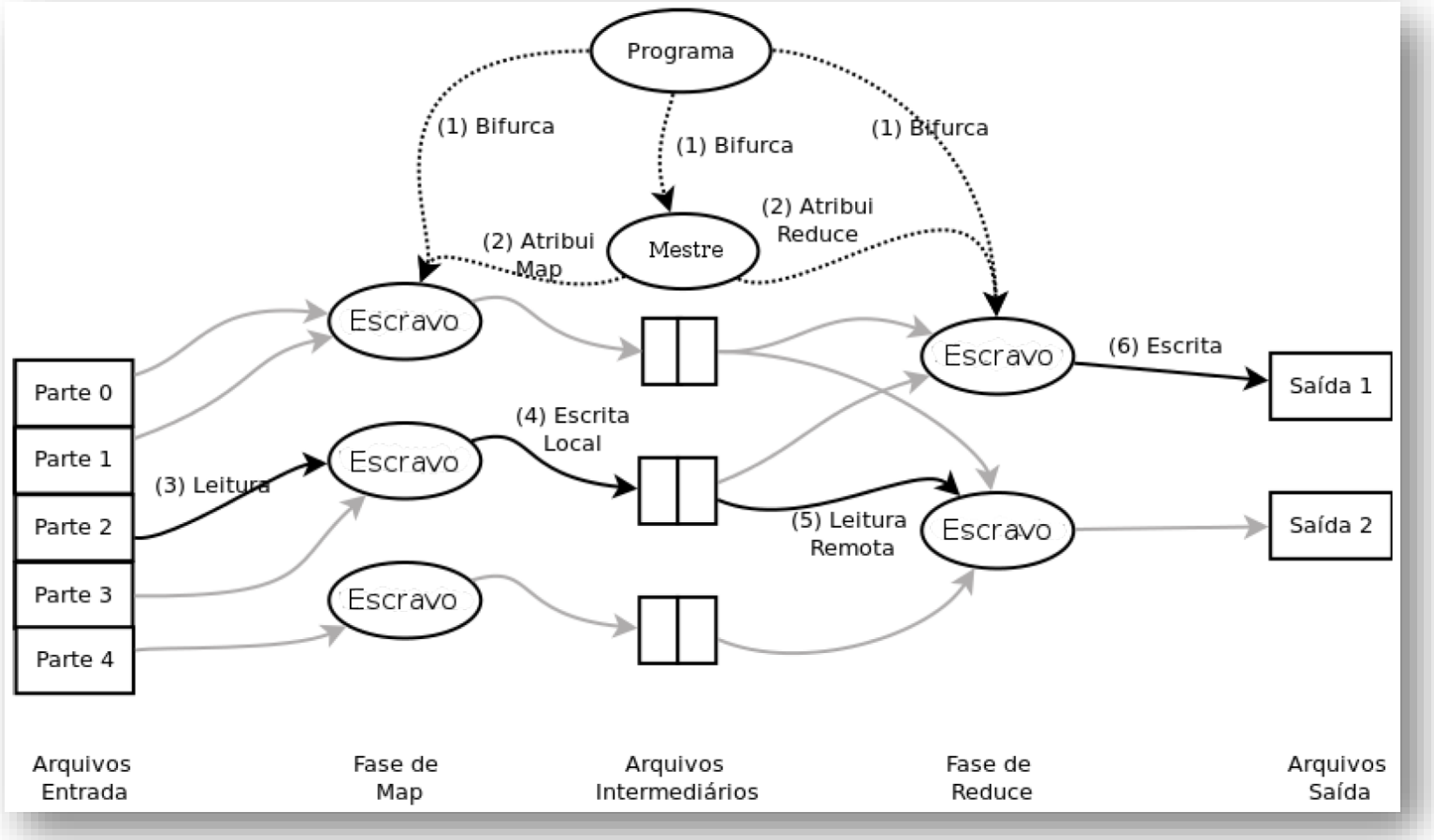
5. O Escravo responsável pela **redução** faz a leitura remota dos dados gravados na fase de **mapeamento** e agrupa todas as ocorrências das chaves e as ordena.

MapReduce
Modelo de Programação



6. O Escravo responsável pela **redução** obtém todos os valores de uma determinada chave e a envia para a função de **redução**.

MapReduce
Modelo de Programação



7. Ao término das tarefas de **mapeamento** e **redução**, o **Mestre** acorda o programa do usuário, retornando o controle para ele.

MapReduce
Tolerância a falhas

- Falhas nos nós **Escravos**
  - O Mestre testa periodicamente a comunicação com os nós Escravos.
  - Caso não receba resposta em um determinado intervalo de tempo, o Mestre marca o nó Escravo como **falho**.
  - Todas as tarefas de **mapeamento** escalonadas para o nó Escravo falho são escalonadas para outros nós, **mesmo as já finalizadas**.
    - Dados do **mapeamento** são salvos **localmente**.

MapReduce
Tolerância a falhas

- Falhas no nó **Mestre**
  - O nó Mestre executa ***checkpoints*** periódicos de suas estruturas de dados.
  - Em caso de falha, um novo nó Mestre é iniciado **a partir do último estado salvo**.
  - O MapReduce assume que sempre existirá somente um nó Mestre.
- Quando o MapReduce está perto de terminar, cópias das tarefas restantes são iniciadas como *backup*.
  - A tarefa será dada como concluída quando a tarefa principal ou o *backup* terminar.

Frameworks MapReduce
Google MapReduce

- Criado por pesquisadores da Google em 2004
- Foi o primeiro **modelo de programação paralela largamente distribuído** para **grandes volumes de dados**
- Utiliza o sistema de arquivos distribuído GFS (Google File System)
- Além das etapas de mapeamento e redução, existem ainda as funções:
  - **Split**
    - Ocorre antes da etapa de mapeamento
    - Leitura e divisão dos arquivos de entrada
  - **Shuffle**
    - Particionamento: produz tuplas intermediárias
    - Ordenamento: ordena as tuplas por suas chaves
  - **Combine**
    - Não é uma etapa obrigatória
    - Ocorre antes da etapa de redução
    - Definida por uma função do usuário (como as funções `Map` e `Reduce`)
    - Pré-processa as tuplas visando diminuir a transmissão pela rede

Frameworks MapReduce
Apache Hadoop

- Criado por pesquisadores da Yahoo em 2006
- Implementado em linguagem Java
- Teve seu código-fonte aberto para a comunidade em junho de 2009
- O *framework* básico é formado pelos módulos:
  - **Hadoop Common**: contém bibliotecas e utilitários necessários para os outros módulos
  - **Hadoop Distributed File System (HDFS)**: sistema de arquivos distribuídos
  - **Hadoop YARN**: responsável por gerenciar o cluster computacional
  - **Hadoop MapReduce**: implementação do modelo de programação MapReduce
- É formado por diversas ferramentas que, juntas, compõem o "ecossistema Hadoop"

Exemplo de Implementação

Trecho da base de dados (entrada)

```
"MÊS REFERÊNCIA";"MÊS COMPETÊNCIA";"UF";"CÓDIGO MUNICÍPIO SIAFI";"NOME MUNICÍPIO";"NIS
FAVORECIDO";"NOME FAVORECIDO";"VALOR PARCELA"
"201904";"201804";"AC";"0139";"RIO BRANCO";"23702404259";"MAURILENE DE OLIVEIRA DAMASCENO";"130,00"
"201904";"201804";"AL";"2765";"INHAPI";"12581359015";"ALOISIO GALDINO DO NASCIMENTO";"253,00"
"201904";"201804";"AL";"2875";"SAO SEBASTIAO";"20666147439";"FATIMA CORREIA DOS SANTOS";"171,00"
"201904";"201804";"AL";"0971";"TEOTONIO VILELA";"23715093281";"MARIA LUZIA DOS SANTOS";"212,00"
"201904";"201804";"AM";"0215";"BOCA DO ACRE";"23754273686";"JARDEANE TEIXEIRA DE OLIVEIRA";"212,00"
"201904";"201804";"AM";"0215";"BOCA DO ACRE";"23732923688";"SIRLETE MOREIRA DA SILVA";"212,00"
```

Exemplo de Implementação

Visão geral da implementação em Java

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class BolsaFamilia {

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

  public static class BolsaFamiliaReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {

  public static void main(String[] args) throws Exception {
}
```

Exemplo de Implementação

Método Mapper

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

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {

    String line = value.toString();

    if (line != null && !line.isEmpty() && !line.contains("VALOR PARCELA")){
      String estado = line.split(";")[2].replace("\"", "");
      Double valorRecebidoDouble = Double.parseDouble(line.split(";")[7].replace("\"", "").
      replace(",", "."));
      int valorRecebido = (int) Math.round(valorRecebidoDouble);
      value.set(estado);
      context.write(value, new IntWritable(valorRecebido));
    }
  }
}
```

Exemplo de Implementação

Método Reducer

```
public static class BolsaFamiliaReducer
    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);
  }
}
```

Exemplo de Implementação

Método Main

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "bolsa familia");
  job.setJarByClass(BolsaFamilia.class);
  job.setMapperClass(BolsaFamiliaMapper.class);
  job.setCombinerClass(BolsaFamiliaReducer.class);
  job.setReducerClass(BolsaFamiliaReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Exemplo de Implementação

Saída dos dados

```
[hdfs@hdp ~]$ hdfs dfs -cat /saida/part-r-00000
AC      23909429
AL      78859015
AM      92401191
AP      17192268
BA      342855374
CE      199922783
DF      12428435
ES      31776466
GO      47907482
MA      207233768
MG      188824473
MS      22783936
MT      26539306
PA      193932600
PB      104535430
PE      215112750
PI      95109555
PR      61035684
RJ      160619176
RN      66326177
RO      12300288
RR      10132893
RS      61964565
SC      21156688
SE      49508137
SP      265293089
TO      22617578
```

Conclusão

- MapReduce é **fácil de utilizar** mesmo por programadores sem experiência em computação distribuída.
- Permite **focar no problema** e não nos detalhes de implementação de sistemas distribuídos.
- Uma **grande variedade de problemas** podem ser tratados com MapReduce.
- **Simplificou a computação** em larga escala em **grandes volumes de dados**.

Dúvidas?

Obrigado!

Referências

ANDRADE, Tiago Pedroso da Cruz. **MapReduce** - Conceitos e Aplicações. Relatório técnico. Campinas: Laboratório de Redes de Computadores Instituto de Computação Universidade Estadual de Campinas, 2014.

CHEVREUIL, Wellington Ramos et al. **MapReduce Detalhado**: Funcionamento e Recursos. Revista Mundo J, número 53, p. 56-64, 2012.

DEAN, Jeffrey; GHEMAWAT, Sanjay. **MapReduce**: simplified data processing on large clusters. Communications of the ACM, v. 51, n. 1, p. 107-113, 2008.

GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. **The Google File System**. Proceedings of the 19th ACM Symposium on Operating Systems Principles., p. 20-43, 2003.

GOLDMAN, Alfredo et al. **Apache Hadoop**: conceitos teóricos e práticos, evolução e novas possibilidades. XXXI Jornadas de atualizações em informática, p. 88-136, 2012.

SHVACHKO, Konstantin et al. **The Hadoop Distributed File System**. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1-10, 2010.