

# MapReduce

MCZA020-13 - Programação Paralela

---

Emilio Francesquini

[e.francesquini@ufabc.edu.br](mailto:e.francesquini@ufabc.edu.br)

2019.Q1

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP**.

## Computação em Nuvem

É um modelo que possibilita **acesso ubíquo**, de forma **conveniente** e **sob demanda** a um conjunto de recursos de computação configuráveis (por exemplo, redes, servidores, dispositivos de armazenamento, aplicações e outros serviços) que **podem ser rapidamente provisionados e dispensados** com o mínimo esforço de gestão ou interação do prestador de serviço.

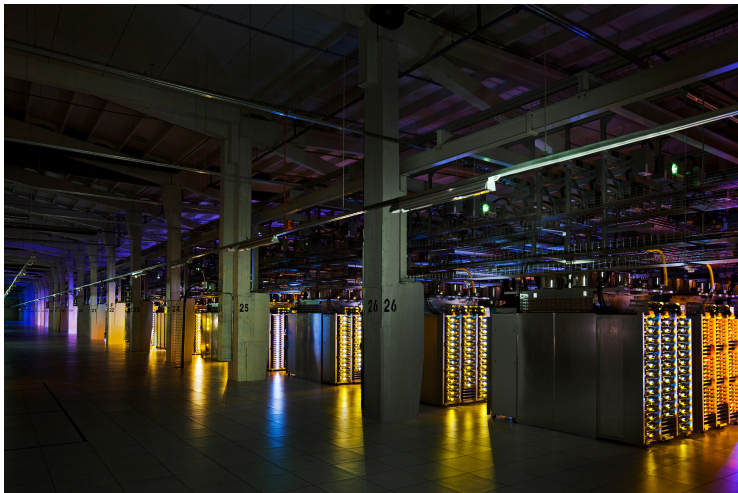
<http://csrc.nist.gov/publications/PubsSPs.html#800-145>

- Problemas “em escala da web”
- **Grandes** *data centers*
- Computação paralela e distribuída
- Aplicações web interativas

- Características
  - ▶ Definitivamente *data-intensive*
  - ▶ Mas podem também ser *processing-intensive*
- Exemplos:
  - ▶ *Crawling*, indexação, busca, mineração de dados da web
  - ▶ Pesquisa em biologia computacional na era “pós-genômica”
  - ▶ Processamento de dados científicos (física, astronomia, etc.)
  - ▶ Redes de sensores
  - ▶ Aplicações Web 2.0
  - ▶ etc.

Estratégia simples (mas de difícil execução):

- Dividir para conquistar
- Usar mais recursos computacionais a medida que mais dados aparecerem

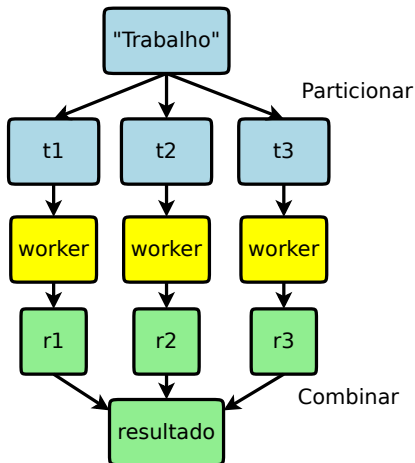


Fonte: <http://www.google.com/intl/pt-BR/about/datacenters/>

- Escalabilidade horizontal, não vertical
  - ▶ Existem limites para máquinas SMP e arquiteturas de memória compartilhada
- Mova o processamento para perto dos dados
  - ▶ a banda de rede é limitada
- Processe os dados sequencialmente, evite padrões de acesso aleatórios
  - ▶ *seeks* são custosos, mas a vazão (*throughput*) do disco é razoável



## Divisão e conquista



- Como repartir as unidades de trabalho entre os *workers*?
- O que fazer quando temos mais trabalho do que *workers*?
- E se os *workers* precisarem compartilhar resultados intermediários entre si?
- Como agregar os resultados parciais?
- O que fazer se um *worker* parar de funcionar?
- Como saber se todos os *workers* terminaram seus trabalhos?

# Hadoop e o paradigma MapReduce

---

## Word Count

Gerar uma lista de frequência das palavras em um conjunto **grande** de arquivos: ordem de *terabytes*!

Curso UFABC 2019  
UFABC 2019 em SP

Word Count

Curso de Graduação  
UFABC 2019 BigData SP

2019, 3  
BigData, 1  
Curso, 2  
de, 1  
em, 1  
Graduação, 1  
UFABC, 3  
SP, 2

Assuma que a máquina tem memória suficiente ( $> 1$  TB !)

```
word-count() {  
  for each document d {  
    for each word w in d {  
      w_count[w]++  
    }  
  }  
  save w_count to persistent storage  
}
```

Fácil, mas provavelmente a execução demorará um longo tempo, pois a entrada é da ordem de *terabytes*

```
Mutex lock; // protege w_count
word-count() {
    for each document d in parallel {
        for each word w in d {
            lock.Lock();
            w_count[w]++
            lock.Unlock();
        }
    }
    save w_count to persistent storage
}
```

## Problemas:

- utiliza uma estrutura de dados única e global
- recursos compartilhados: seção crítica!

Então, como fazer programas que processam petabytes de dados?

- O modelo inicial proposto pelo Google apresentou conceitos para simplificar alguns problemas
- Paralelização da computação em um aglomerado de máquinas comuns (com centenas/milhares de CPUs)
- Paralelização e distribuição automática de computação deveria ser o mais simples possível
- O sistema de execução se encarrega de:
  - ▶ particionar e distribuir os dados de entrada
  - ▶ escalonar as execuções em um conjunto de máquinas
  - ▶ tratar as falhas
  - ▶ comunicação entre as máquinas



O modelo de programação paralela MapReduce aborda os problemas da seguinte forma:

- 1 Leia uma grande quantidade de dados
- 2 Aplique a função **MAP**: extrai alguma informação de valor!
- 3 Fase intermediária: Shuffle & Sort
- 4 Aplique a função **REDUCE**: reúne, compila, filtra, transforma, etc.
- 5 Grave os resultados

- A ideia do modelo de programação Map e Reduce não é nova
- Presente em linguagens funcionais há mais de 40 anos!
- No Hadoop é a parte do arcabouço responsável pelo processamento distribuído (paralelo) de grandes conjuntos de dados
- Usa padrões já conhecidos:

```
cat      | grep  | sort   | uniq   > arquivo  
entrada | map  | shuffle | reduce > saída
```

## Map em programação funcional

```
map({1,2,3,4}, (×2)) → {2,4,6,8}
```

Todos os elementos são processados por um método e os elementos não afetam uns aos outros.

## Reduce em programação funcional

```
reduce({1,2,3,4}, (×)) → {24}
```

- Todos os elementos da lista são processados juntos
- Tanto em Map quanto em Reduce: a entrada é fixa (imutável), e a saída é uma nova lista (em geral)

- Ótimo para trabalhar com grandes quantidades (petabytes) de dados
- Realiza computação “perto” dos dados
- Dados são compartilhados através de um *sistema de arquivos distribuído*

## Apache Hadoop

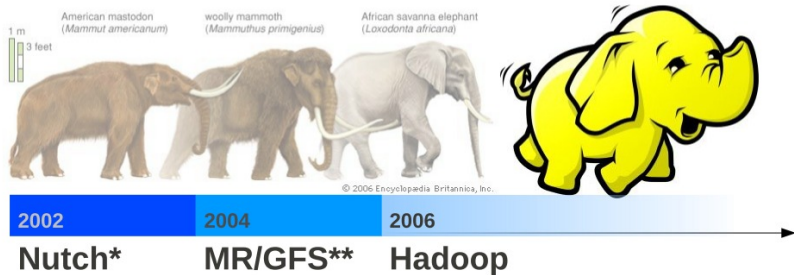
Hadoop remove a complexidade da computação de alto desempenho

## Custo eficiente

- Máquinas comuns
- Rede comum
- Tolerância a falhas automática
  - ▶ Poucos administradores
- Facilidade de uso
  - ▶ Poucos programadores

Arcabouço para processamento e armazenamento de dados em larga escala:

- Código aberto
- Implementado em Java
- Inspirado no GFS e MapReduce do Google
- Projeto *top-level* da Fundação Apache
- Tecnologia recente, porém já muito utilizada



\* <http://nutch.apache.org/>

\*\* <http://labs.google.com/papers/mapreduce.html>  
<http://labs.google.com/papers/gfs.html>



- 2003** Google publica artigo do GFS (SOSP'03)
- 2004** Google publica artigo do MapReduce (OSDI'04)
- 2005** Doug Cutting cria uma versão do MapReduce para o projeto Nutch
- 2006** Hadoop se torna um subprojeto do Apache Lucene

- 2007** Yahoo! Inc. torna-se o maior contribuidor e utilizador do projeto (aglomerado com mais de 1.000 nós)
- 2008** Hadoop deixa a tutela do projeto Lucene e se transforma em um projeto *top-level* da Apache
- 2010** Facebook anuncia o maior aglomerado Hadoop do mundo (mais de 2.900 nós e 30 petabytes de dados)
- 2011** Apache disponibiliza a versão 1.0.0
- 2019** Versão atual: 3.2.0



Adobe



amazon  
web services™



facebook



last.fm



LinkedIn



YAHOO!



twitter



The New York Times

**A COMPUTER WANTED.**

WASHINGTON, May 1.—A civil service examination will be held May 18 in Washington, and, if necessary, in other cities, to secure eligibles for the position of computer in the Nautical Almanac Office, where two vacancies exist—one at \$1,000, the other at \$1,400.

The examination will include the subjects of algebra, geometry, trigonometry, and astronomy. Application blanks may be obtained of the United States Civil Service Commission.

**The New York Times**

Published: May 2, 1892

Copyright © The New York Times

<http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/>

- Em 2007, o jornal The New York Times converteu para PDF todos seus os artigos publicados entre 1851 e 1980
- Cada artigo é composto por várias imagens previamente digitalizadas que precisavam ser posicionadas e redimensionadas de forma coerente pra a criação do PDF
- 4 TB de imagens TIFF em 11 milhões de arquivos PDF
- 100 instâncias EC2 da Amazon foram utilizadas durante 24 horas para gerar 1,5 TB de arquivos PDF, a um custo de aproximadamente **US\$ 240,00**

Por que o Hadoop faz tanto sucesso?

---

Por que usar Hadoop?

- Código aberto
- Econômico
- Robusto
- Escalável
- Foco na regra de negócio

## Código aberto

- Comunidade ativa
- Apoio de grandes corporações
- Correções de erros frequentes
- Constante evolução do arcabouço



## Econômico

- Software livre
- Uso de máquinas e redes convencionais
- Aluguel de serviços disponíveis na nuvem:
  - ▶ Amazon Elastic MapReduce
  - ▶ Google App Engine MapReduce
  - ▶ etc.

## Robusto

- Se em 1 máquina há probabilidade de haver falhas...
  - ▶ Tempo médio entre falhas para 1 nó: 3 anos
  - ▶ Tempo médio entre falhas para 1.000 nós: 1 dia

## Estratégias

- Replicação dos dados
- Armazenamento de metadados

## Escalável

- Permite facilmente adicionar máquinas ao aglomerado
- Adição não implica na alteração do código-fonte
- Limitação apenas relacionada a quantidade de recursos disponíveis

## Foco na regra de negócio

- Hadoop realiza todo o “trabalho duro”
- Desenvolvedores podem focar apenas na abstração do problema

## Único nó mestre

- Ponto único de falha
- Pode impedir o escalonamento

## Dificuldade das aplicações paralelas

- Problemas não paralelizáveis
- Processamento de arquivos pequenos
- Muito processamento em um pequeno conjunto de dados

## Problemas

- Os dados que serão processados não cabem em um nó
- Cada nó é composto por hardware comum
- Falhas podem (e irão) acontecer

## Ideias e soluções do Apache Hadoop

- Sistema de arquivos distribuído
- Replicação interna
- Recuperação de falhas automática

## Problemas

- Mover dados é caro (largura de banda pequena)
- Mover computação é barato
- Programação paralela e distribuída é difícil

## Ideias e soluções do Apache Hadoop

- Mover a computação para onde estão os dados
- Escrever programas que são fáceis de se distribuir
- Paralelismo de dados utilizando conceitos de linguagem funcional

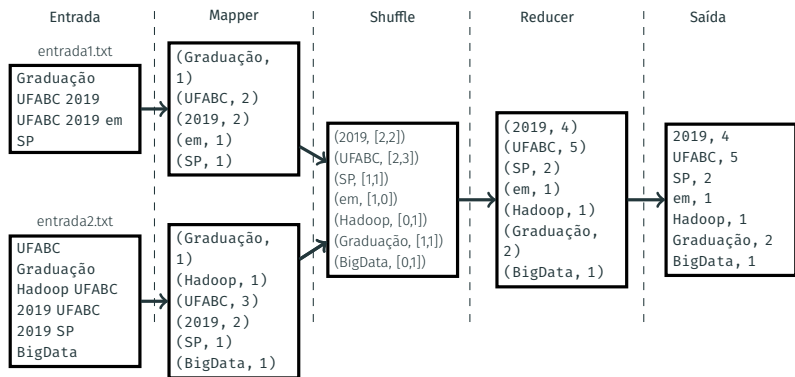
- A função Map atua sobre um conjunto de entrada com chaves e valores, produzindo uma lista de chaves e valores
- A função Reduce atua sobre os valores intermediários produzidos pelo Map para, normalmente, agrupar os valores e produzir a saída

	Entrada	Saída
map	$\langle k1, v1 \rangle$	lista( $\langle k2, v2 \rangle$ )
reduce	$\langle k2, \text{lista}(v2) \rangle$	lista( $\langle k3, v3 \rangle$ )



- Lê arquivos texto e conta a frequência das palavras
  - ▶ **Entrada:** arquivos texto
  - ▶ **Saída:** arquivo texto
  - ▶ **Cada linha:** palavra, separador (tab), quantidade
- **Map:** gera pares (palavra, quantidade)
- **Reduce:** para cada palavra, soma as quantidades

```
map(String key, String value):  
  // key: nome do documento  
  // value: conteúdo do documento  
  for each word w in value:  
    EmitIntermediate(w, "1");  
  
reduce(String key, Iterator values):  
  // key: uma palavra  
  // value: uma lista de contadores  
  int result = 0;  
  for each v in values:  
    result += ParseInt(v);  
  Emit(key, AsString(result));
```



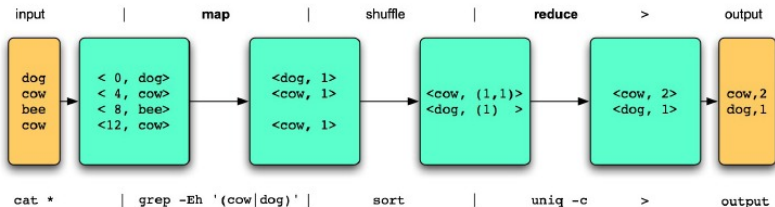
1

<sup>1</sup>Errata: os pares após shuffle devem aparecer ordenados

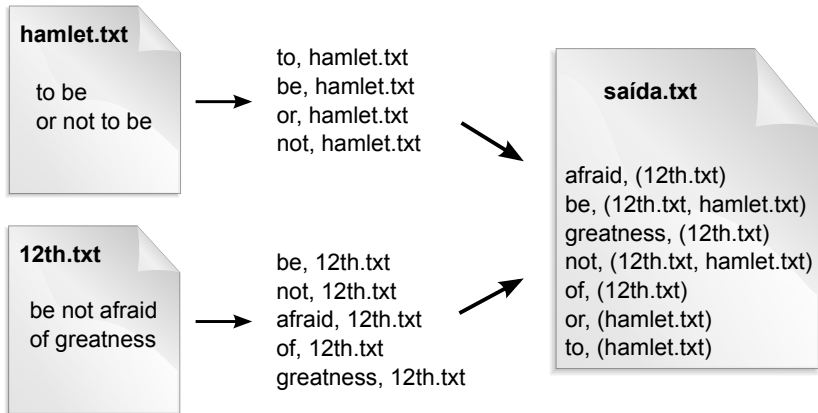
- Procura nos arquivos de entrada por um dado padrão
- **Map**: emite uma linha se um padrão é encontrado
- **Reduce**: copia os resultados para a saída

```
cat | grep | sort | uniq > arquivo
```

```
cat      | grep  | sort   | uniq  > arquivo
entrada | map | shuffle | reduce > saída
```



- Gerar o índice invertido das palavras de um conjunto de arquivos dado
- **Map:** faz a análise dos documentos e gera pares de (palavra, docId)
- **Reduce:** recebe todos os pares de uma palavra, organiza os valores docId, e gera um par (palavra, lista(docId))

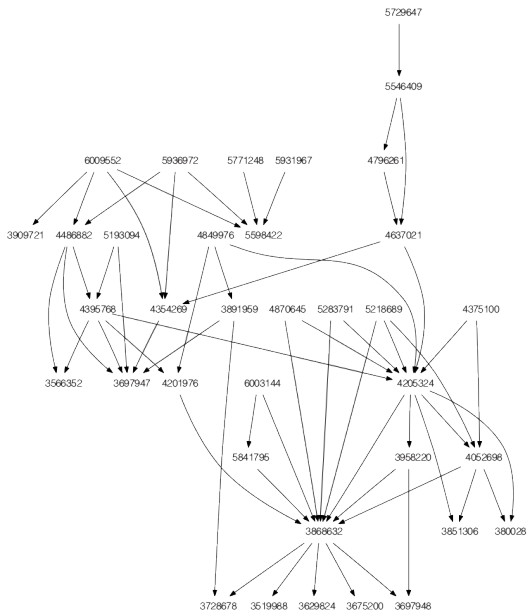




## Alguns exemplos de MapReduce

---

# Exemplo 1: Citações em patentes



Entrada: pares (A,B) que indicam que a patente A cita a patente B no seu texto.

cite75\_99.txt

```
"CITING", "CITED"  
3858241, 956203  
3858241, 1324234  
3858241, 3398406  
3858241, 3557384  
3858241, 3634889  
3858242, 1515701  
3858242, 3319261  
3858242, 3668705  
3858242, 3707004  
...
```

- **Entrada:** todas as citações à patentes americanas feitas entre 1975 e 1999 e suas informações
- **Saída:** para cada patente, a lista de todas as patentes que a citam

O primeiro passo é definir o *data flow* do programa!

- Qual o formato da entrada e qual deveria ser o formato da saída?
- O que a função **Map** deve fazer?
- O que a função **Reduce** deveria fazer?

- Qual o formato da entrada e qual deveria ser o formato da saída?
  
  
  
  
  
  
  
  
  
  
- O que a função **Map** deve fazer?
  
  
  
  
  
  
  
  
  
  
- O que a função **Reduce** deveria fazer?

- Qual o formato da entrada e qual deveria ser o formato da saída?
  - ▶ A entrada são pares  $(A, B)$  – **quem citou** e quem foi citado  
– ex: **3858241**, 956203
  - ▶ A saída deveria ser algo como  $B$  foi citada em  $A, C, D, \dots$  ex:  
956203    **3858241**, 5312208, 4944640, ...
- O que a função **Map** deve fazer?
  
- O que a função **Reduce** deveria fazer?

- Qual o formato da entrada e qual deveria ser o formato da saída?
  - ▶ A entrada são pares  $(A, B)$  – quem citou e quem foi citado – ex: **3858241**, 956203
  - ▶ A saída deveria ser algo como  $B$  foi citada em  $A, C, D, \dots$  ex: 956203    **3858241**, 5312208, 4944640, ...
- O que a função **Map** deve fazer?
  - ▶ Quem foi citado é o dado principal. Nesse caso, ele deve ser a *chave* e quem o citou o seu *valor*; ou seja, o `map()` **deve produzir o par  $(B, A)$** .
- O que a função **Reduce** deveria fazer?



- Qual o formato da entrada e qual deveria ser o formato da saída?
  - ▶ A entrada são pares  $(A, B)$  – **quem citou** e quem foi citado  
– ex: **3858241**, 956203
  - ▶ A saída deveria ser algo como  $B$  foi citada em  $A, C, D, \dots$  ex:  
956203    **3858241**, 5312208, 4944640, ...
- O que a função **Map** deve fazer?
  - ▶ Quem foi citado é o dado principal. Nesse caso, ele deve ser a *chave* e quem o citou o seu *valor*; ou seja, o `map()` **deve produzir o par  $(B, A)$** .
- O que a função **Reduce** deveria fazer?
  - ▶ **Juntar** todos os valores  $B$  que citam  $A$

## Método map()

```
public void map(Text key, Text value,  
                OutputCollector<Text, Text> output,  
                Reporter reporter) throws IOException  
{  
    output.collect(value, key);  
}
```

Método `reduce()`

```
public void reduce(Text key, Iterator<Text> values,
                  OutputCollector<Text, Text> output,
                  Reporter reporter) throws IOException
{
    String csv = "";
    while (values.hasNext()) {
        if (csv.length() > 0) csv += ",";
        csv += values.next().toString();
    }

    output.collect(key, new Text(csv));
}
```

## Note que

O mesmo programa do exemplo poderia ser utilizado em outros contextos

Relações entre:

- “comprador” e “vendedor”
- “funcionário” e “departamento”
- “fármaco” e “princípio ativo”
- etc.

E se eu quisesse contar o número de patentes que citam uma patente?

- Devo mudar o `map()`? Por quê?
- Devo mudar o `reduce()`? Por quê?
- O que cada uma dessas operações deve fazer?

```
public void reduce(Text key, Iterator<Text> values,
                  OutputCollector<Text, IntWritable>
                    output,
                  Reporter reporter)
{
    int count = 0;
    while (values.hasNext()) {
        values.next();
        count++;
    }
    output.collect(key, new IntWritable(count));
}
```

Dado um arquivo com dados meteorológicos, calcular a temperatura média **por mês**.

```
Data      Precipitacao;Temperatura; Umidade Relativa;  
                                     Velocidade do Vento;  
01/01/1990  15.5;22.24;88;1.766667;  
02/01/1990  35.9;21.2;89.75;2.333333;  
...  
30/09/2013  0;18.34;91.25;1.54332;  
01/10/2013  6.6;19.94;80.25;2.74368
```

- O que a função **Map** deve fazer?
- O que a função **Reduce** deve fazer?

```
public static class MonthTempMapper
    extends Mapper<Text, Text, IntWritable, FloatWritable> {

    private IntWritable mes          = new IntWritable();
    private FloatWritable temperatura = new FloatWritable();

    public void map(Text key, Text value, Context context)
        throws IOException, InterruptedException
    {
        // key contém a data (dd/mm/aaaa)
        String chave = key.toString();
        String[] valor = value.toString().split(";");

        if(chave.charAt(0) == '#' || valor.length != 4 || valor[1].isEmpty())
            return; // linha comentada ou com valor faltando

        // mês; as datas seguem o formato dd/mm/aaaa
        int mes = Integer.parseInt(chave.substring(3,5));

        // value contém os dados meteorológicos separados por ";"
        float temperatura = Float.parseFloat(valor[1]);

        context.write(new IntWritable(mes), new FloatWritable(temperatura));
    }
}
```



```
public static class AverageReducer
    extends Reducer<IntWritable,FloatWritable,Text,FloatWritable> {
    private FloatWritable media = new FloatWritable();

    public void reduce(IntWritable key, Iterable<FloatWritable> values,
        Context context
        ) throws IOException, InterruptedException {
        float sum = 0.0f;
        int length = 0;
        for (FloatWritable val : values) {
            sum += val.get();
            length += 1;
        }

        media.set(sum/length);

        String[] nomeDoMes = {"Jan", "Fev", "Mar", "Abr", "Mai", "Jun",
            "Jul", "Ago", "Set", "Out", "Nov", "Dez"};
        Text mes = new Text(nomeDoMes[key.get()-1]);

        context.write(mes, media);
    }
}
```

Um **anagrama** é uma palavra ou frase feita com as letras de outra (ex.: as palavras asco, caos, cosa, saco, soca são anagramas de caso).

### Exemplo:

Dada uma lista de palavras, descobrir quais dentre elas são anagramas.

### Como sempre...

Qual o *data flow* do programa?

- Qual o formato da entrada e qual deveria ser o formato da saída?
- O que a função **Map** deve fazer?
- O que a função **Reduce** deveria fazer?

```
public void map(LongWritable key, Text value,
    OutputCollector<Text, Text> outputCollector,
    Reporter reporter) throws IOException {

    String word = value.toString();
    char[] wordChars = word.toCharArray();
    Arrays.sort(wordChars);
    String sortedWord = new String(wordChars);
    sortedText.set(sortedWord);
    originalText.set(word);
    outputCollector.collect(sortedText, originalText);
}
```

```
public void reduce(Text anagramKey, Iterator<Text> anagramValues,
    OutputCollector<Text, Text> results, Reporter reporter)
    throws IOException {
    String output = "";
    while(anagramValues.hasNext())
    {
        Text anagam = anagramValues.next();
        output = output + anagam.toString() + "~";
    }
    StringTokenizer outputTokenizer = new StringTokenizer(output, "~");
    if(outputTokenizer.countTokens()>=2)
    {
        output = output.replace("~", ",");
        outputKey.set(anagramKey.toString());
        outputValue.set(output);
        results.collect(outputKey, outputValue);
    }
}
```

MapReduce não é um modelo de programação adequado para:

- processamento em tempo real
- aplicações que precisam realizar comunicação entre tarefas
- processamento de fluxo contínuo de dados
- aplicações que necessitam de garantias transacionais (OLTP)
- problemas difíceis de serem expressados com a abstração proporcionada pelo modelo MapReduce

- Tom White. **Hadoop: The Definitive Guide**. Yahoo Press.  
ISBN: 9781449311520
- Chuck Lam. **Hadoop in Action**. Manning Publications.  
ISBN: 9781935182191
- Alfredo Goldman *et al.* Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades. Em: XXXI Jornadas de Atualização em Informática. Sociedade Brasileira de Computação, 2012.  
<http://www.lbd.dcc.ufmg.br/colecoes/jai/2012/003.pdf>