

# MC970/M0644

## Programação Paralela na Nuvem usando Spark

**Hervé Yviquel, Guido Araújo**

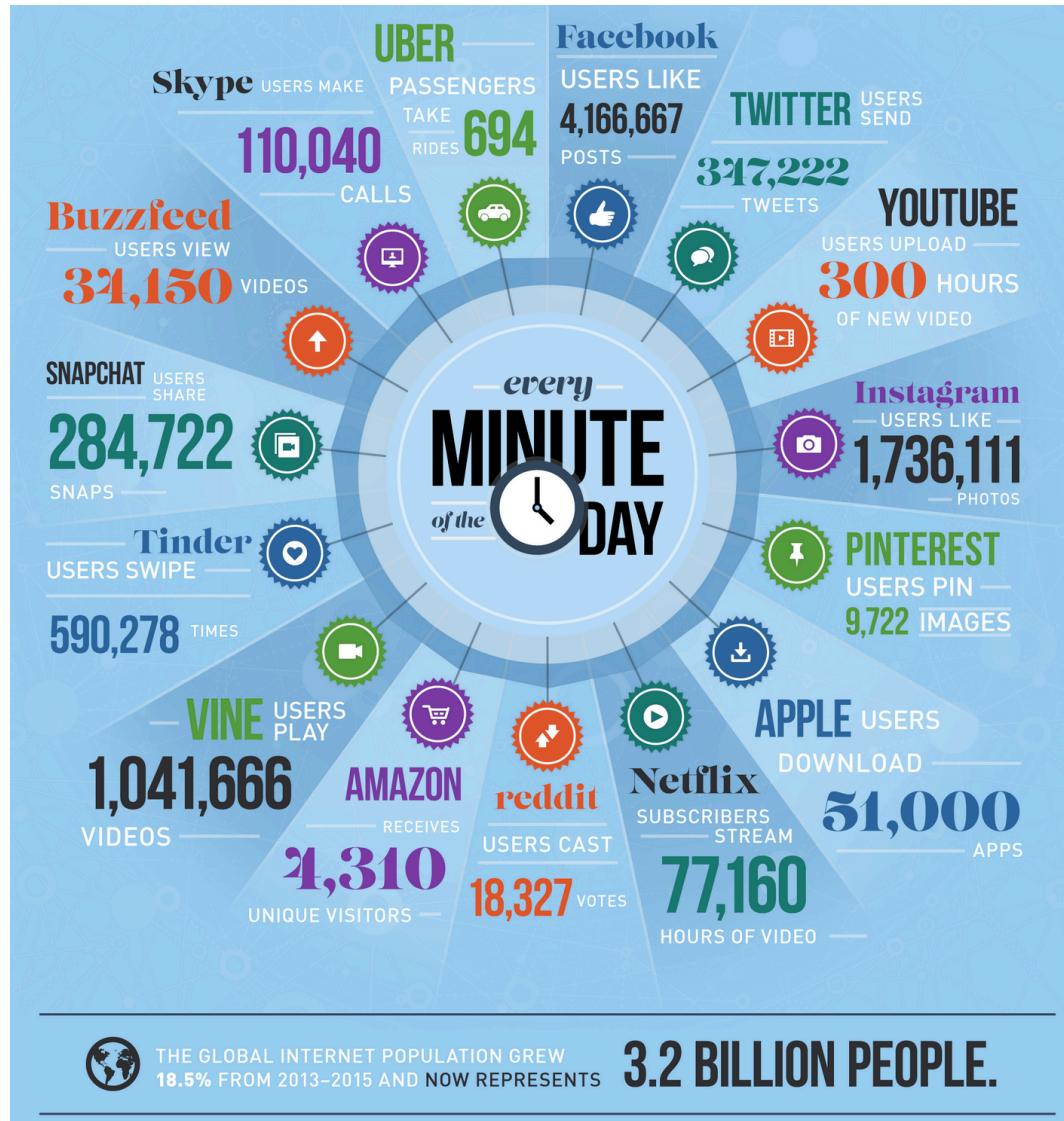
herve.yviquel@ic.unicamp.br



# Vivemos em um Mundo de Dados



# Enormes Volumes de Dados...



Fonte: domo.com

Data Never Sleeps 3.0

3.7 bilhões de pessoas em Janeiro de 2017

# Mas não é só "volume"



Fonte: [usr.uvic.cat](http://usr.uvic.cat)

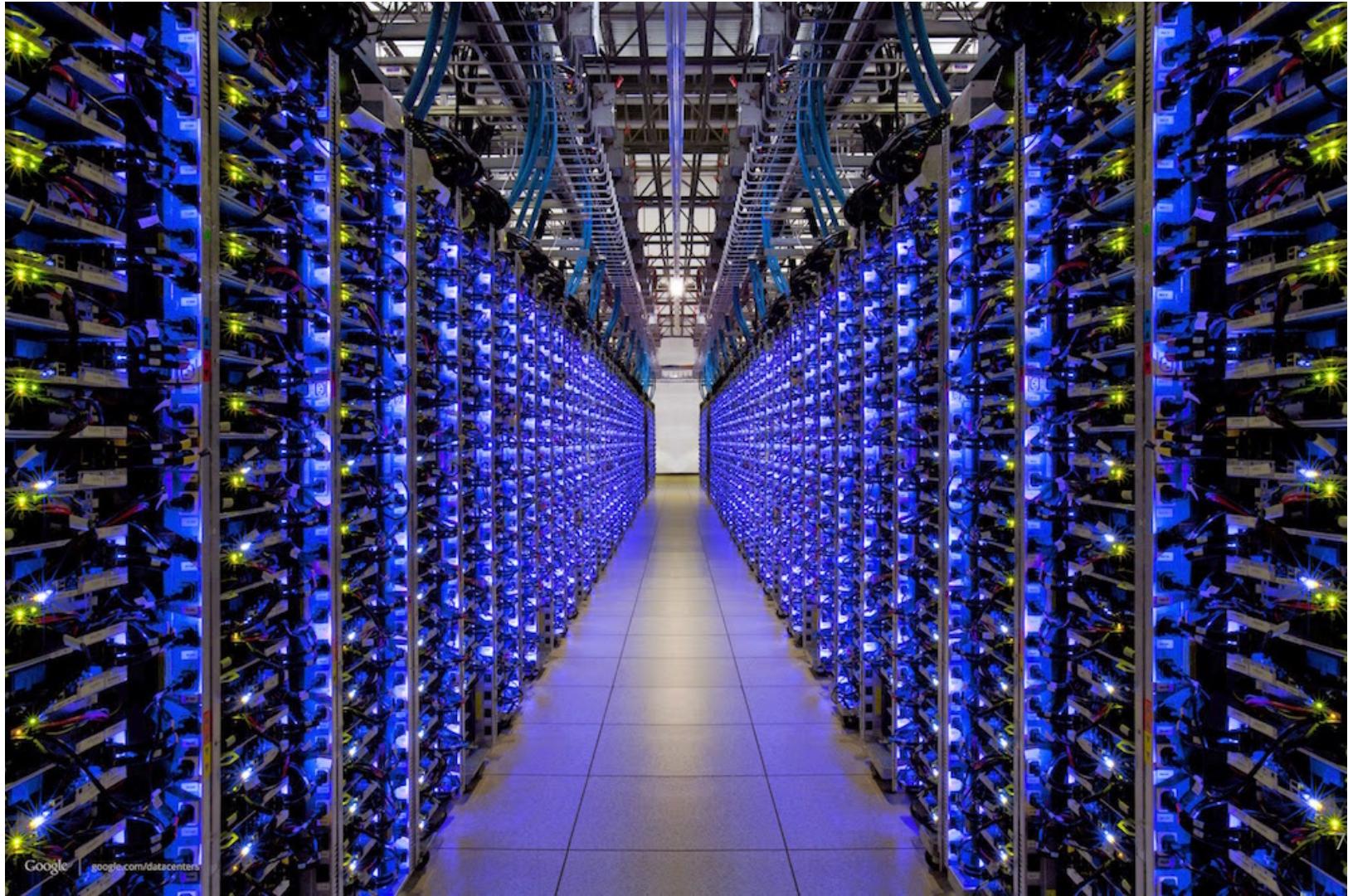
*The Importance of Big Data*

# O que é “Big Data”

« Big Data é qualquer dado que é caro para se gerenciar e do qual é difícil extrair valor »

Prof. Thomas Siebel, Diretor do AMPLab, UC-Berkeley

# O datacenter como Computador



# Computação em Nuvem

Solução para “The Rising of Big Data”

- Massivo poder de processamento
  - Datacenter (50.000 to 80.000 servidores)
  - Cluster de computadores
- Pode ser útil para outros domínios de aplicação
  - Aplicações científicas (HPC)
  - Aplicações mobile (Mobile cloud offloading)
  - Internet das Coisas (IoT)

**Mas como programar arquitetura distribuída ??**

# Message Passing Interface

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

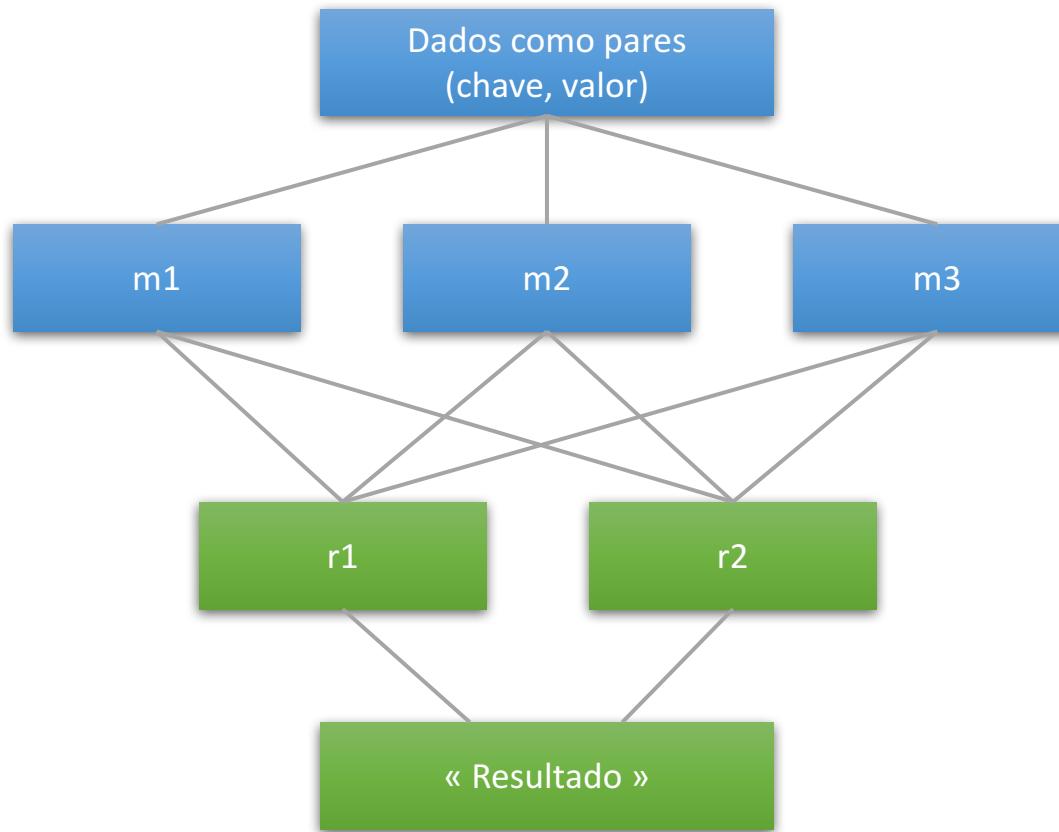
int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

## Utilizado em High-Performance Computing

- Programação de baixo nível
  - Eficiente mais complicado
- Exige redes com baixa latência
  - Nuvem são imprevisíveis (recursos compartilhados)
- Sem tolerância a falhas
  - Compartilhar recursos aumenta falhas

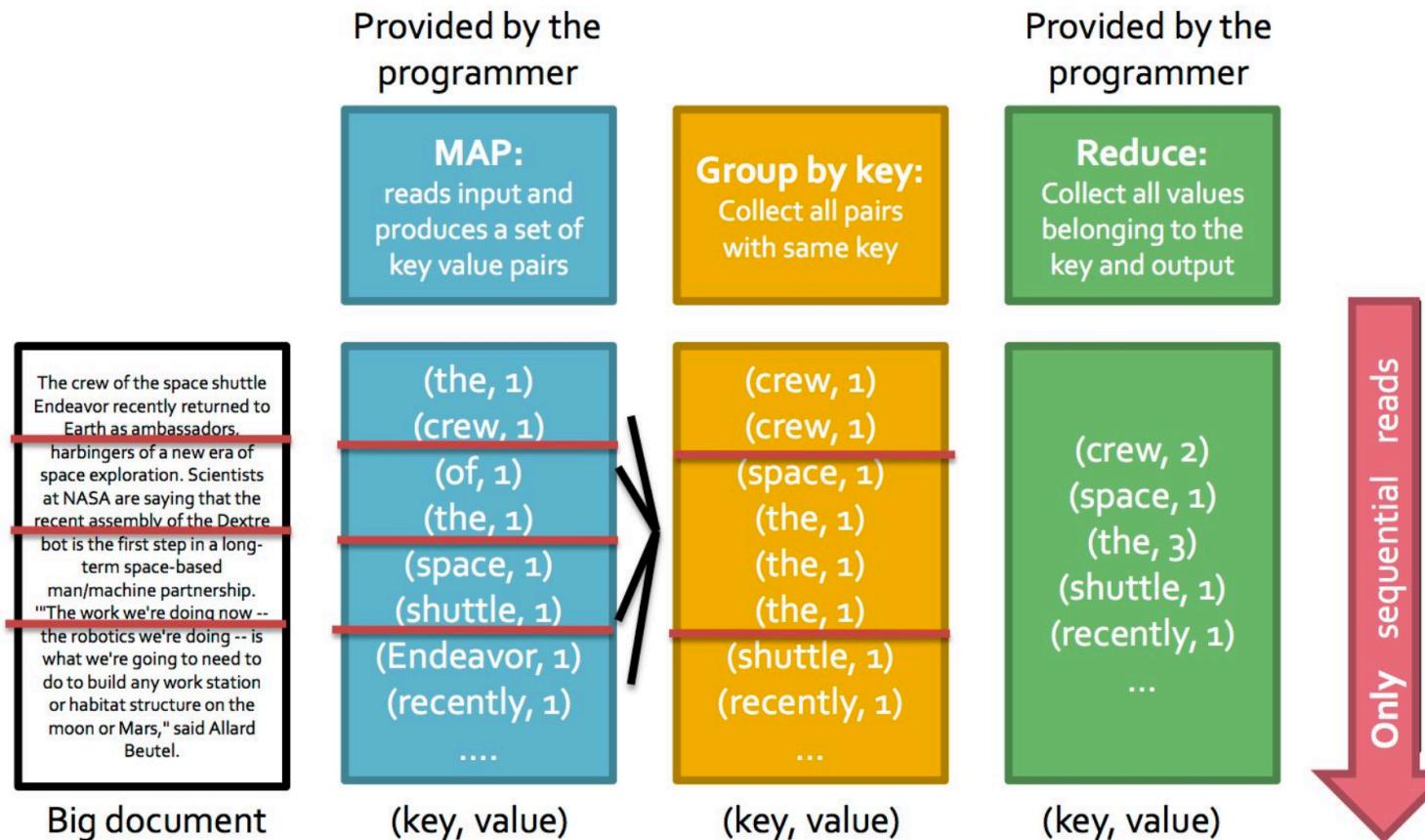
# Dividir para conquistar!

## Map-Reduce [Google2004]



Particione  
↓  
**Map**  
↓  
Combine  
↓  
**Reduce**  
↓  
**Armazene**

# Contar Palavras com Map-Reduce



# Sobre Map-Reduce

- Confie no sistema de arquivo distribuído
  - Divide arquivos em grandes blocos de dados (e.g. 64MB)
  - Usar a localidade de dados para acelerar o processamento distribuído
- Independente da arquitetura
  - Massivamente paralelizável
- Permite tolerância a falhas

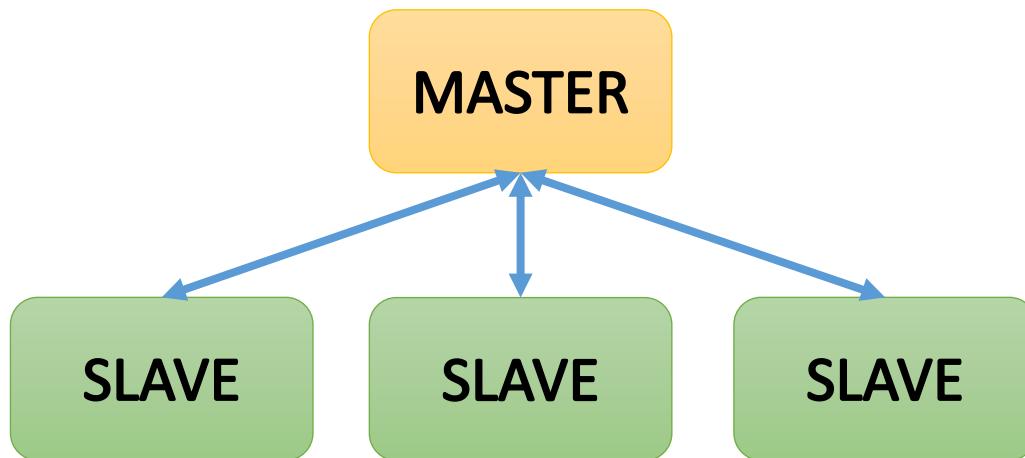
# Arquitetura *mestre e escravo*

1 processo mestre (*JobTracker*)

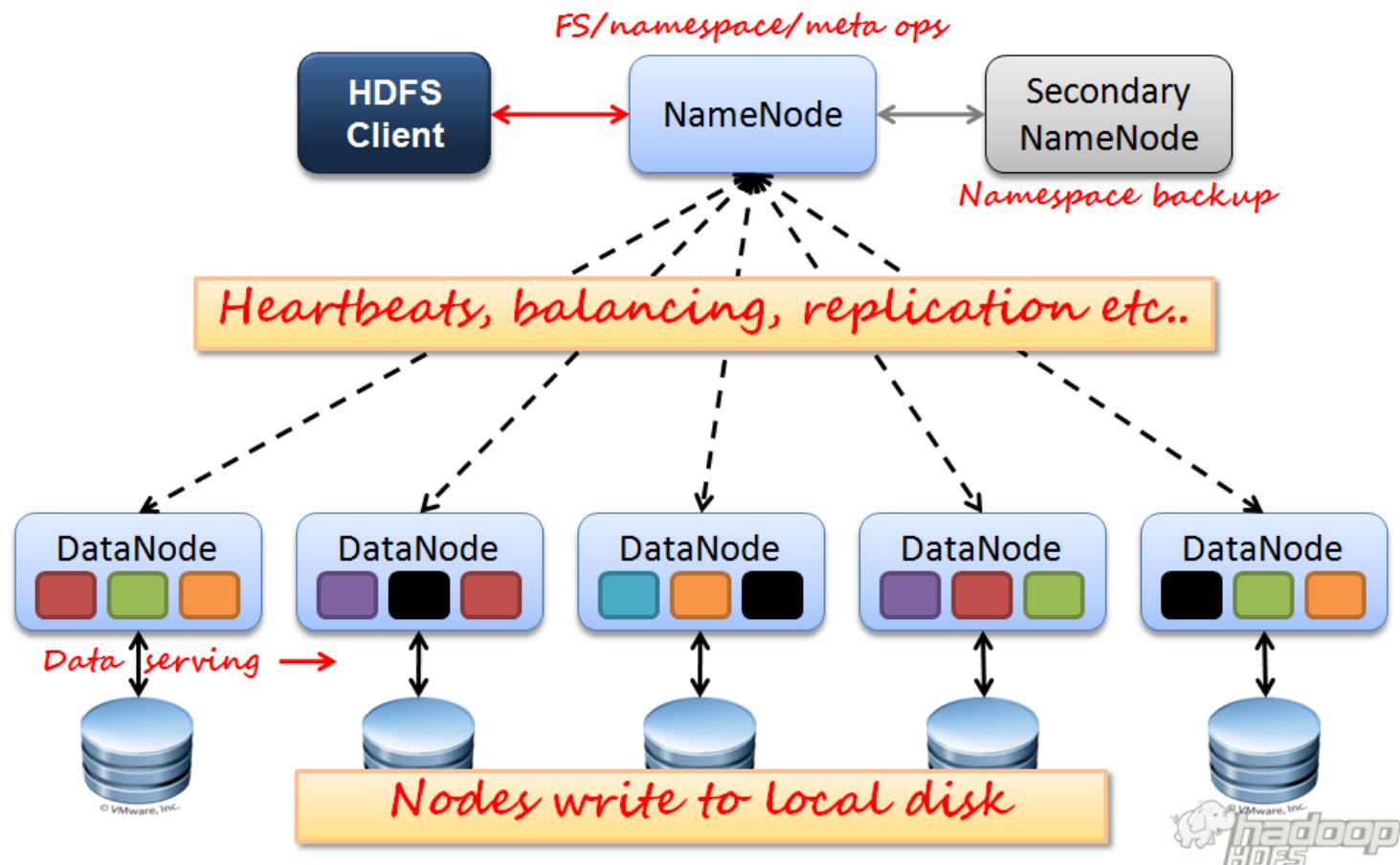
responsável por escalaronar a execução de tarefas pelos escravos, monitorá-las e re-executá-las em caso de falha

N processos escravos (*TaskTracker*) – 1 por cada nó

responsável por executar as tarefas (map ou reduce) designadas pelo mestre.



# Sistema de Arquivo Distribuído



# Tolerância a Falhas

- Replicação dos dados (HDFS)
  - Pode reconstruir dados corrompidos
- Mensagens de *Heartbeat*
  - Informa o status das máquinas
  - Pode reexecutar tarefas expiradas

# Apache Hadoop

Framework para computação distribuída

## **Hadoop Common**

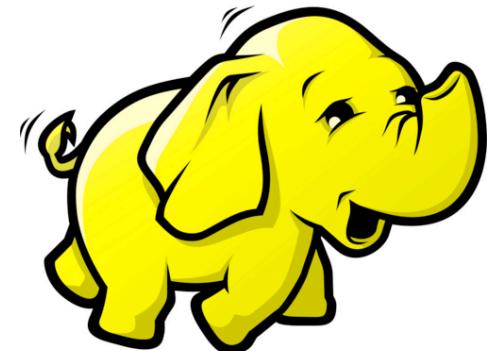
Core library

## **Hadoop MapReduce**

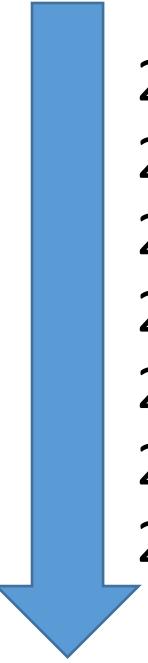
## **Hadoop Distributed File System**

## **Hadoop YARN**

Job scheduling and cluster resource management



# Breve História (1)

- 
- 2002 – Creation of **Nutch**
  - 2003 – **GFS paper** by Google
  - 2004 – **MapReduce paper** by Google
  - 2005 – Nutch added support to GFS and MapReduce
  - 2006 – **Hadoop** creation from Nutch project
  - 2008 – Apache Hadoop **top-level**
  - 2009 – **Hadoop won Graysort Daytona competition**

# Map-Reduce não resolve Todos os Problemas

## **Processamento de consultas**

Banco de Dados

## **Processamento iterativo**

Dados ficam na memoria

## **Processamento de fluxos**

Nem sempre o arquivo é a forma da entrada

## **Processamento de grafos**

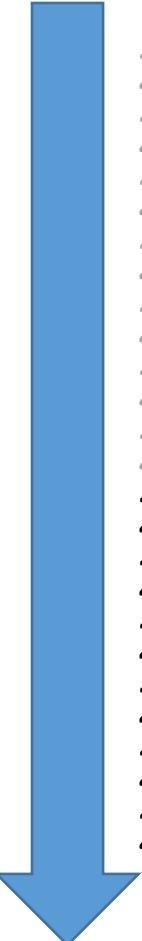
Processar estruturas irregulares

# Spark

- Plataforma de computação em clusters.
  - Criada para ser rápida e de propósito geral
  - Modelo de programação de alto-nível
- O processamento é multi-estágio
  - Representado como grafo direcionado e acíclico (DAG)
  - Suporte processamento iterativos e de fluxos
  - Processamento em memoria  
(Ate 100x mais rápido que Map-Reduce)

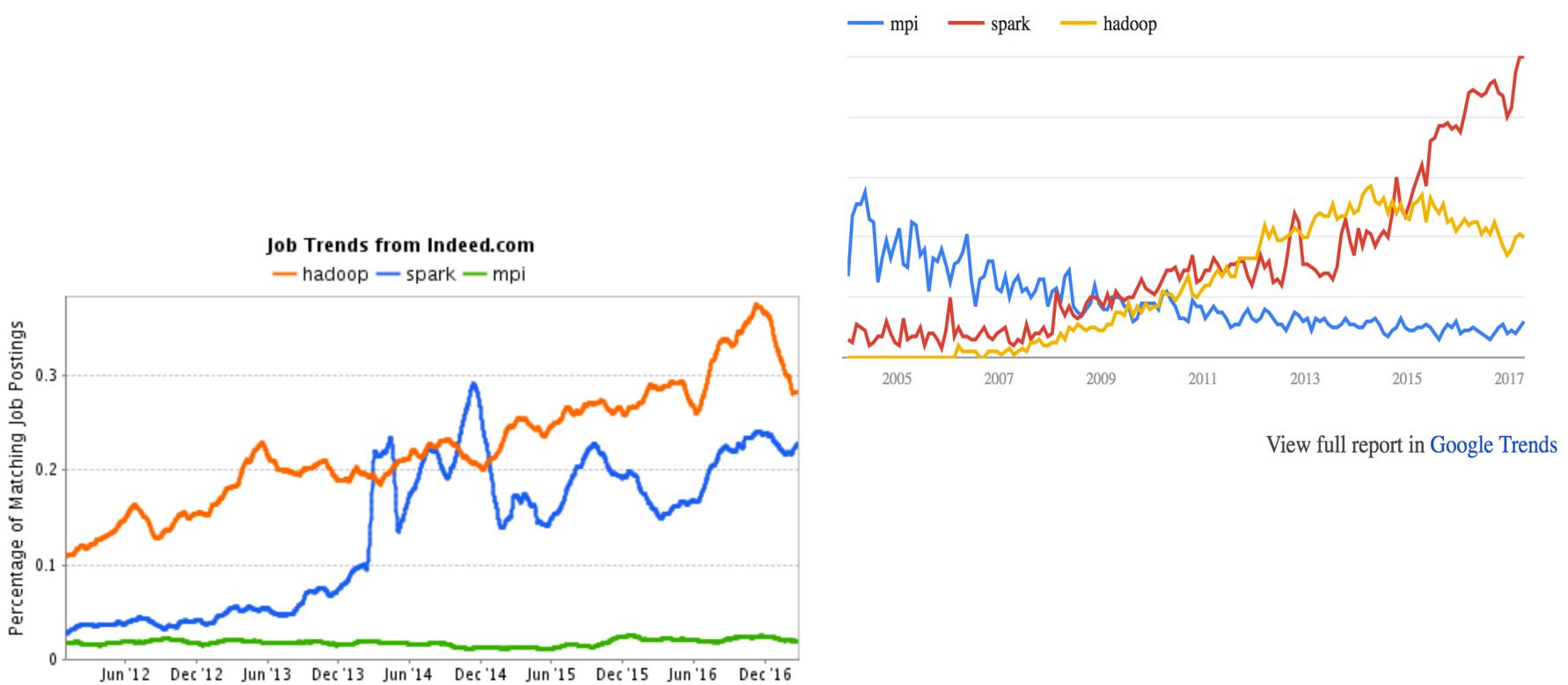


# Breve História (2)

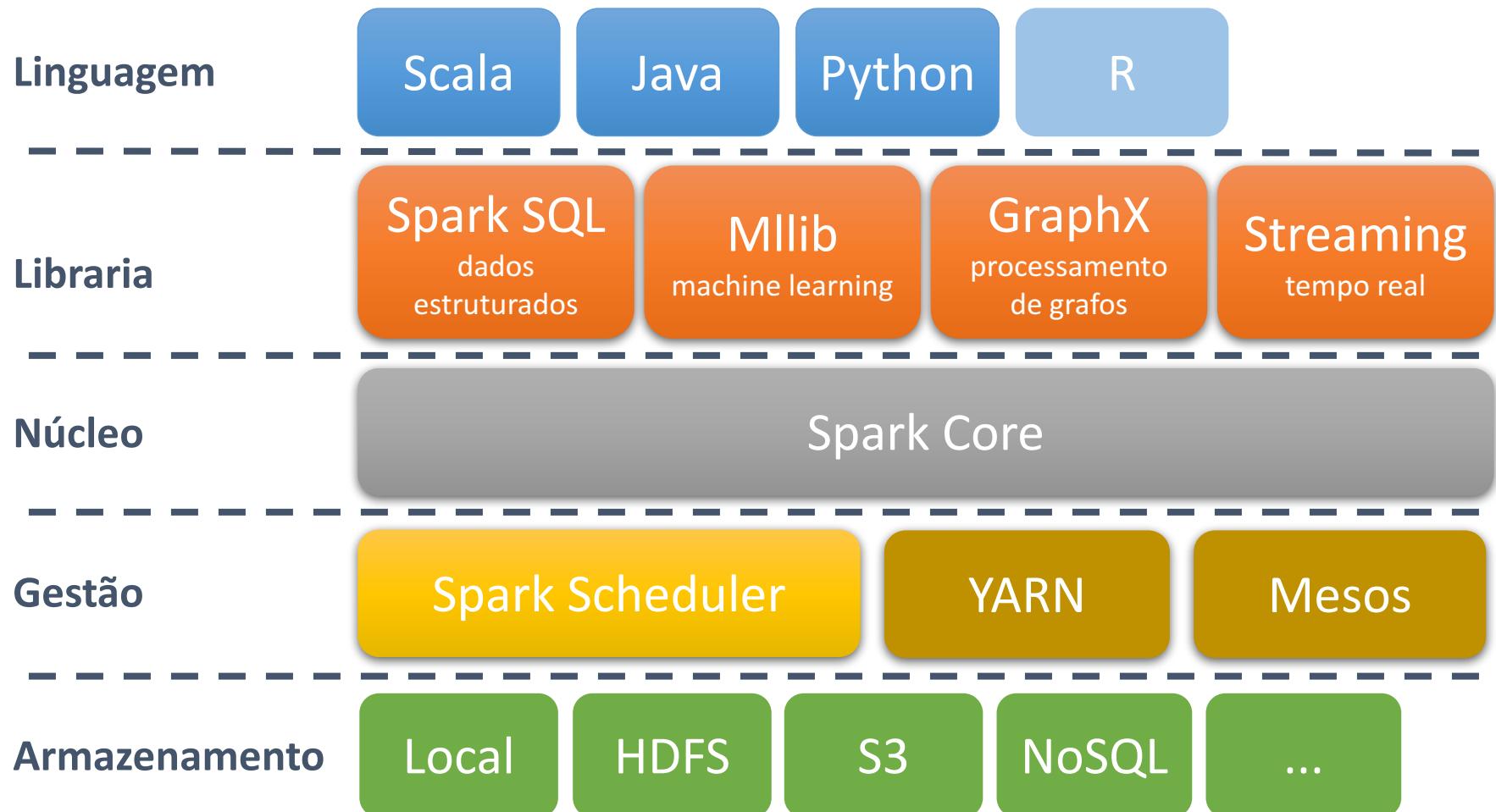
- 
- 2002 – Creation of Nutch
  - 2003 – GFS paper by Google
  - 2004 – MapReduce paper by Google
  - 2005 – Nutch added support to GFS and MapReduce
  - 2006 – Hadoop creation from Nutch project
  - 2008 – Apache Hadoop top-level
  - 2009 – Hadoop won *Graysort Daytona* competition
  - 2009 – **Development of Spark** at UC-Berkeley
  - 2010 – **Spark paper** and open-source
  - 2013 – Spark transferred to Apache
  - 2014 – Apache Spark top-level
  - 2014 – **Spark won Graysort Daytona** competition
  - 2016 – Spark **version 2.0**

# Tendências para MPI, Hadoop, e Spark

Interest over time. Web Search. Worldwide, 2004 - present, Programming.



# Pilha de Software do Spark



# Scala vs Java vs Python

- Spark foi escrito originalmente em Scala, que permite sintaxe de função concisa e uso interativo
- A API da Java adicionada para aplicativos autônomos
- A API do Python foi adicionada mais recentemente com um shell interativo

# Sobre Scala

Linguagem de alto nível para a JVM

- Programação orientada a objetos e funcional

Estaticamente tipada

- Comparável em velocidade com Java
- Inferência de tipo (não precisa escrever tipos explícitos em geral)

Interoperabilidade com Java

- Pode usar qualquer classe Java (herança de, etc.)
- Pode ser chamado a partir do código Java

# Visita Rápida de Scala

## Declarar variáveis:

```
var x: Int = 7  
var x = 7 // type inferred  
val y = "hi" // read-only
```

## Equivalente em Java:

```
int x = 7;  
  
final String y = "hi";
```

## Funções:

```
def square(x: Int): Int = x*x  
def square(x: Int): Int = {  
    x*x  
}  
  
def announce(text: String) =  
{  
    println(text)  
}
```

## Equivalente em Java:

```
int square(int x) {  
    return x*x;  
}  
  
void announce(String text) {  
    System.out.println(text);  
}
```

# Funções em Scala (Clausuras)

```
(x: Int) => x + 2 // versão completa  
x => x + 2 // type inferred  
_ + 2 // argumento implícito  
x => { // corpo é um bloco de código  
    val numberToAdd = 2  
    x + numberToAdd  
}  
// função regular  
def addTwo(x: Int): Int = x + 2
```

# Visita Rápida de Scala (2)

## Processar coleções com programação funcional

```
val list = List(1, 2, 3)
```

```
list.foreach(x => println(x)) // prints 1, 2, 3
```

```
list.foreach(println) // mesmo
```

```
list.map(x => x + 2) // retorna nova List(3,4,5)
```

```
list.map(_ + 2) // mesmo
```

```
list.filter(x => x % 2 == 1) // retorna nova List(1, 3)
```

```
list.filter(_ % 2 == 1) // mesmo
```

```
list.reduce((x, y) => x + y) // => 6
```

```
list.reduce(_ + _) // mesmo
```

# Ferramentas de Trabalho com Spark

- Investigação interativa: *spark shell*

- Desenvolvimento de aplicação: *spark submit*  
./bin/spark-submit --class SimpleApp --master local SimpleApp.jar

# Conceitos Básicos

Uma aplicação consiste de um programa chamado driver

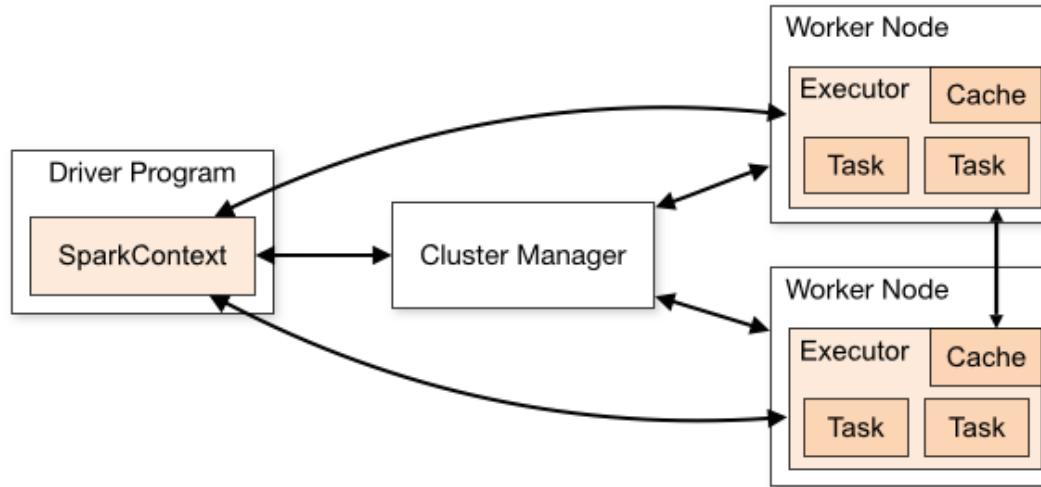
- o *driver* dispara trabalho (local ou no cluster)
- o *driver* toma controle do recurso do cluster através de um objeto de contexto (SparkContext).
- o *driver* descreve o fluxo (DAG) de uma aplicação, composto por coleções de dados distribuídas (RDDs) e seus relacionamentos (operações).
- no modo interativo, o *driver* é o próprio shell em execução.

# Escreve um Spark Job

## Simple WordCount Scala example

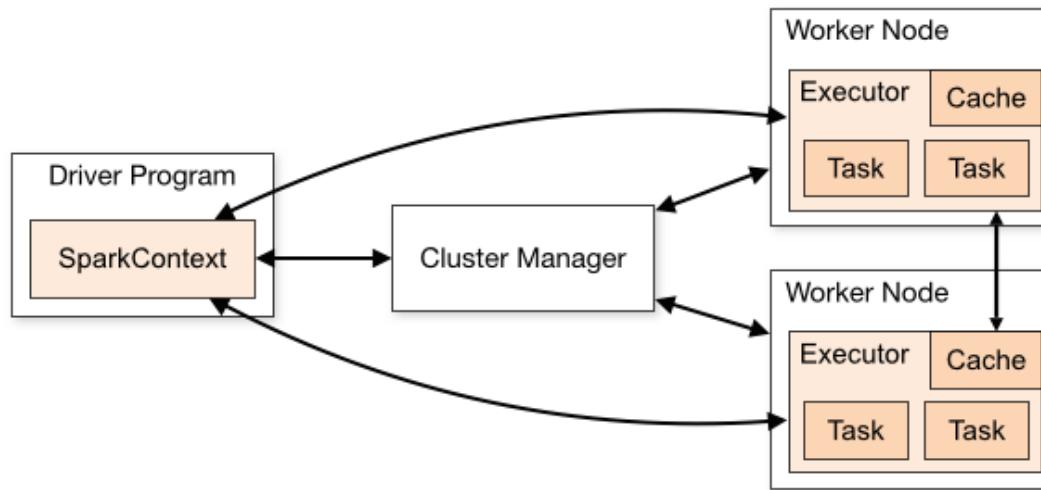
```
// Create a Scala Spark Context.  
val conf = new SparkConf().setAppName("wordCount")  
val sc = new SparkContext(conf)  
  
// Load our input data.  
val input = sc.textFile(inputFile)  
  
// Split it up into words.  
val words = input.flatMap(line => line.split(" "))  
  
// Transform into word and count.  
val counts = words.map(word => (word, 1))  
    .reduceByKey{case (x, y) => x + y}  
  
// Save the word count back out to a text file.  
counts.saveAsTextFile(outputFile)
```

# Aplicação Spark



Conjuntos de processos independentes (“executor”) em nós (“worker”) do cluster coordenados pelo programa principal (“driver”)

# Aplicação Spark



O objeto “SparkContext”

1. Conecta com o gerenciador do cluster
2. Adquire executores dos nós no cluster
3. Envia o código do aplicativo para os executores
4. Envia tarefas aos executores para serem executados

# Aplicação Auto-Contida

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object SimpleApp {
    def main(args: Array[String]) {
        val conf = new SparkConf()
        conf.setMaster("local").setAppName("WordCount")
        val sc = new SparkContext(conf)
        ...
    }
}
```

# Compilar sua Aplicação Spark

## Usando **Scala Build Tool**

- Escrever sua Aplicação Spark em Scala
- Escrever o “build.sbt”
- Executar “sbt” na pasta

```
// build.sbt
name := "WordCount"
version := "0.0.1"
scalaVersion := "2.11.8"

// Additional Libraries
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "2.1.0" % "provided"
)
```

# Coleção de dados em Spark

## Resilient Distributed Dataset (RDD)

- Uma abstração para trabalhar com grandes conjuntos de dados (dataset)
- Coleção distribuída de elementos
  - Divide em partições
  - Pode conter qualquer objeto



# Criação de RDDs: arquivos

```
val lines = sc.textFile("file:///path/to/README.md")
```

- Neste exemplo, a fonte de dados externa é um arquivo.
- o prefixo file:// indica o sistema de arquivos local.
- hdfs:// é outra opção comum, se o arquivo estiver no sistema de arquivos do Hadoop (HDFS).

# Criação de RDDs: paralelizar

O SparkContext « sc » é capaz de paralelizar/distribuir coleções locais ao programa driver

```
val frases = List("pandas", "i like pandas"))
val lines = sc.parallelize(frases)
```

# Tarefa em Spark

Toda a computação acontece em função de RDDs

1. Criar novos RDDs
2. Transformar RDDs existentes
3. Chamar operação sobre RDDs para calcular resultados

# Operações de RDD

Qualquer operação sobre um RDD se enquadra em uma das categorias

- **Transformação**

- Cria um novo RDD a partir de outro
- Avaliação é preguiçosa (lazy)

- **Ação**

- Retorna resultado para o driver.
- Avaliação é imediata.

# Operações de RDD

- **map(f) - transformação**

- Aplica a função  $f()$  a cada elemento  $x$  do RDD,
- gerando um RDD contendo os valores de  $f(x)$

- **reduce(f) - ação**

- Aplica a função  $f()$  a todos os elementos do RDD "de uma vez".
- Por exemplo,  $(\_ + \_)$  significa "some todos os elem"
- A função tem que ser associativa.

# Transformação

**RDD** = {1, 2, 3, 4}

- **RDD.map( x => x\*x )**  
{1, 4, 9, 16}
- **RDD.filter( x => x!=1 )**  
{2, 3, 4}

# Pseudo-set Operation (1)

**RDD1** = {coffee, coffee, panda, monkey, tea}

**RDD2** = {coffee, monkey, kitty}

- **RDD1.distinct()**  
{coffee, panda, monkey, tea}
- **RDD1.union( RDD2 )**  
{coffee, coffee, coffee, panda, monkey, tea, kitty}
- **RDD1.intersection( RDD2 )**  
{coffee, monkey}
- **RDD1.subtract( RDD2 )**  
{panda, tea}

# Pseudo-set Operation (2)

**RDD1** = {User(1), User(2), User(3)}

**RDD2** = {Venue("BarDoZe"), Venue("Bardana"),  
Venue ("Bronco")}

- **RDD1.cartesian( RDD2 )**

{(User(1), Venue("BarDoZe")), (User(1), Venue("Bardana")),  
(User(1), Venue ("Bronco")),  
(User(2), Venue("BarDoZe")), (User(2), Venue("Bardana")),  
(User(2), Venue ("Bronco")),  
(User(3), Venue("BarDoZe")), (User(3), Venue("Bardana")),  
(User(3), Venue ("Bronco"))}

# Mais Ações

RDD = {1, 2, 3, 3}

- **RDD.reduce( `_+_` )**  
9
- **RDD.collect()**  
{1, 2, 3, 3}
- **RDD.count()**  
4
- **RDD.first()**  
1
- **RDD.take(2)**  
{1, 2}
- **foreach(*func*)**  
Nothing
- **saveAsTextFile(*path*), saveAsSequenceFile(*path*), saveAsObjectFile(*path*)**

# WordCount em Spark

```
object SimpleApp {  
    def main(args: Array[String]) {  
        // Create a Scala Spark Context.  
        val conf = new SparkConf().setAppName("wordCount")  
        val sc = new SparkContext(conf)  
  
        val lines = sc.textFile(inputFile)  
        // cada item do RDD é uma linha do arquivo (String)  
        val words = lines.flatMap(line => line.split (" " ))  
        // cada item do RDD é uma palavra do arquivo  
        val intermData = words.map(word => (word,1))  
        // cada item do arquivo é um par (palavra,1)  
        val wordCount = intermData.reduceByKey(_ + _)  
        // cada item do RDD contém ocorrência final de cada palavra  
        val 5contagens = wordCount.take(5)  
        // 5 resultados no programa driver  
    }  
}
```

# WordCount em Map-Reduce (1)

```
public class WordCount {  
  
    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);  
                }  
            }  
        }  
}
```

# WordCount em Map-Reduce (2)

```
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);
    }
}
```

# WordCount em Map-Reduce (3)

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = Job.getInstance(conf, "word count");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setCombinerClass(IntSumReducer.class);  
    job.setReducerClass(IntSumReducer.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);  
}  
}
```

# Estimando Pi (de novo)

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
val pi =
```



```
println("Pi is roughly " + pi)
```

# Estimando Pi (de novo)

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
val pi = 4 *  
    sc.parallelize(0 to N)  
    .map{ i =>  
        (if(i%2==0) 1 else 1).toDouble  
        /(2*i+1)  
    }.reduce( _+_ )  
println("Pi is roughly " + pi)
```

# Mecanismo de Persistência

RDDs são avaliados preguiçosamente

```
val cachedRDD = anyRDD.persist(<nível>)
```

- <nível> indica se o caching deve ser feito em memória, disco, serializado ou misturas

```
val cachedRDD = anyRDD.cache()
```

- nível padrão
- cache() = persist(StorageLevel.MEMORY\_ONLY)

# Exemplo de Persistência

Gerar um RDD de números aleatórios (pseudo)

```
val recompRDD = sc.parallelize(1 to 1000000).map(_ =>  
Math.random())  
  
for(i <- 1 to 10) println(recompRDD.reduce(_+_))
```

**Vamos testar no spark-shell !!**

# Persistência (níveis)

nível	consumo espaço	consumo CPU	em memória	em disco
MEMORY_ONLY	muito	pouco	tudo	nada
MEMORY_ONLY_SER	pouco	muito	tudo	nada
MEMORY_AND_DISK	muito	médio	parte	parte
MEMORY_AND_DISK_SER	pouco	muito	parte	parte
DISK_ONLY	pouco	muito	nada	tudo

```
val cachedRDD = anyRDD.persist(<nível>)
```

# Transformação de Pares

**rdd = {(1, 2), (3, 4), (3, 6)}**

- **rdd.groupByKey()**  
{(1, [2]), (3, [4, 6])}
- **rdd.reduceByKey( \_+\_ )**  
{(1, 2), (3, 10)}
- **rdd.mapValues(x => x+1)**  
{(1, 3), (3, 5), (3, 7)}
- **rdd.flatMapValues(x => x.to(5))**  
{(1,2), (1,3), (1,4), (1,5), (3, 4), (3,5)}
- **rdd.keys()**  
{1, 3 , 3}
- **rdd.values()**  
{2, 4, 6}
- **rdd.sortByKey()**  
{(1, 2), (3, 4), (3, 6)}

# Transformação de Pares

**rdd = {(1, 2), (3, 4), (3, 6)}**

**other = {(3, 9)}**

- **rdd.groupByKey(other)**  
{(1, 2)}
- **rdd.join(other)**  
{(3, (4, 9)), (3, (6, 9))}
- **rdd.cogroup(other)**  
{(1, ([2], [])), (3, ([4, 6], [9])))}

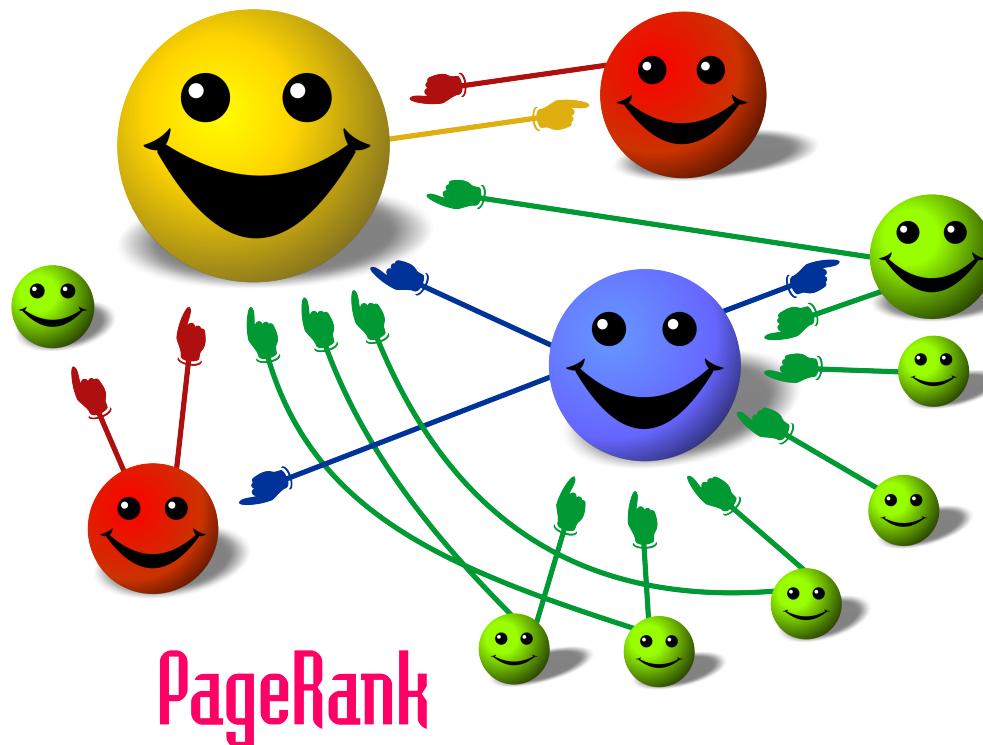
# Estudo de caso: *pagerank*

Exemplo clássico que mostra 2 pontos fortes de Spark: caching e computação iterativa

- Propósito: criar um ranqueamento de importância de nós em um grafo.
- Onde é usado?
  - Google utiliza o PageRank (proposto por Larry Page)
- Sabe a ordem de links que aparecem em uma busca que você faz no Google?

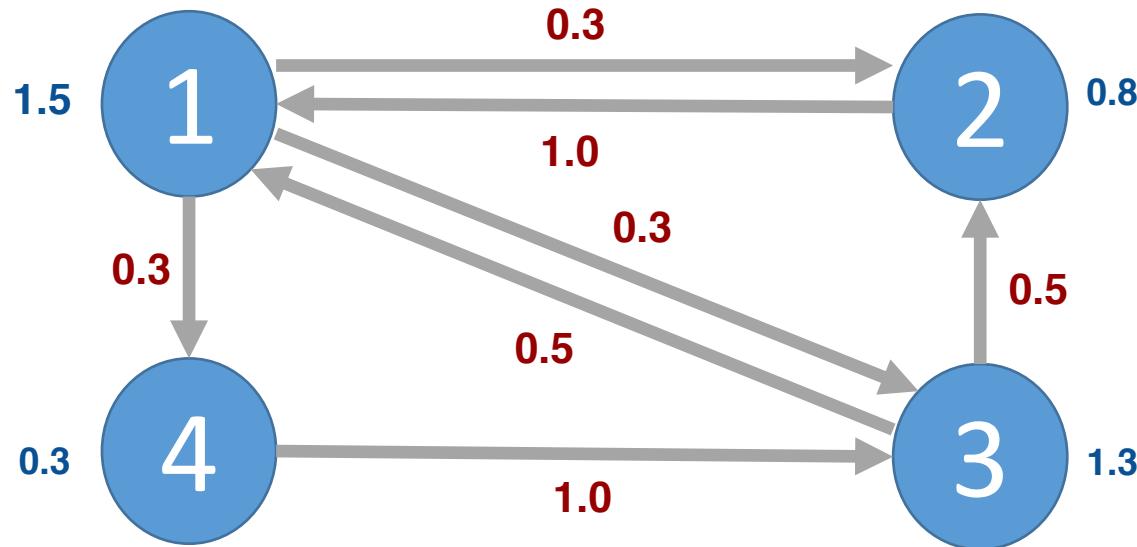
# Premissa do PageRank

A importância de uma página é determinada pela importância das páginas que apontam para ela



# Descrição do Algoritmo

- Todas iniciam com importância 1.0, ou seja, 100%.
- A cada iteração toda página distribui sua importância igualmente para os vizinhos.
- Agora cada página tem uma nova importância, que é a soma dos valores recebidos.



# Implementação do *PageRank*

```
object PageRank {  
    def main(args: Array[String]) {  
        val links = // RDD de pares (página, lista de adjacência)  
        links.cache()  
        var rankings = // RDD de pares (página, 1.0)  
        for (i <- 1 to ITERATIONS) {  
            val contribs = links.join(rankings).flatMap {  
                case (url, (adjList, rank)) =>  
                    adjList.map(dest => (dest, rank / adjList.size))  
            }  
            rankings = contribs.reduceByKey(_ + _)  
        }  
    }  
}
```

- **Parte iterativa**
  - o for principal, que ocorre no driver
- **Parte paralela**
  - as transformações a cada iteração

# Entender Clausura

```
var counter = 0
var rdd = sc.parallelize(data)
// Wrong: Don't do this!!
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

## Comportamento indefinido

- Local versus Cluster

# Melhorar Desempenho

## Particionamento inteligente

- Distribuição das chaves
- Balanceamento da carga de trabalho

## Uso consciente de memória

- Tamanho de resultados retornados para o driver através de ações
- Capacidade versus demanda para caching

## Variáveis compartilhadas

- Broadcast
- Acumuladores

# Variáveis Compartilhadas (1)

## Variáveis de transmissão (*broadcast*)

- Única cópia em cada nó
- Transmissão eficiente (BitTorrent)
- Para grande conjunto de dados de entrada

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar:
org.apache.spark.broadcast.Broadcast[Array[Int]] =
Broadcast(0)
scala> broadcastVar.value
```

# Variáveis Compartilhadas (2)

## Acumuladores

- “adicionado” a através de uma operação associativo, eficientemente suportados em paralelo
- Por exemplo para implementar contador

```
scala> val accum = sc.accumulator(0)
scala> sc.parallelize(Array(1, 2, 3, 4))
           .foreach(x => accum += x)
scala> accum.value
res2: Int = 10
```

# Bibliotecas Específicas

- **MLlib**
  - Estatística: testes de hipóteses, amostragem
  - Classificação / Regressão / Agrupamento
  - Extração de características
  - Mineração de padrões frequentes, etc.
- **SparkStreaming**
  - Processamento de dados em tempo (quase)real
- **SparkSQL**
  - DataFrames para dados estruturados.
- **GraphX**
  - Abstrações para grafos e troca de mensagens

**Tudo isso implementado sobre os mesmos conceitos!**

# Conclusão

- Modelo Map-Reduce adaptado para nuvem
  - Grande escalabilidade
  - Tolerância a falhas
- Spark é uma generalização do Map-Reduce
  - “Better, Stronger, Faster”
  - Programação alto-nível e mais flexível
  - Desenvolvimento muito ativo

# Bibliografia

- “**Learning Spark - Lightning-Fast Big Data Analysis**”  
de H. Karau, A. Konwinski, P. Wendell, e M. Zaharia
- Documentação do Apache Spark ([latest](#))
- [Palestra do Vinicius Dias \(UFMG\)](#)
- Scala Crash Course (Databricks)