Processamento de Dados Massivos (Big-Data) com Spark

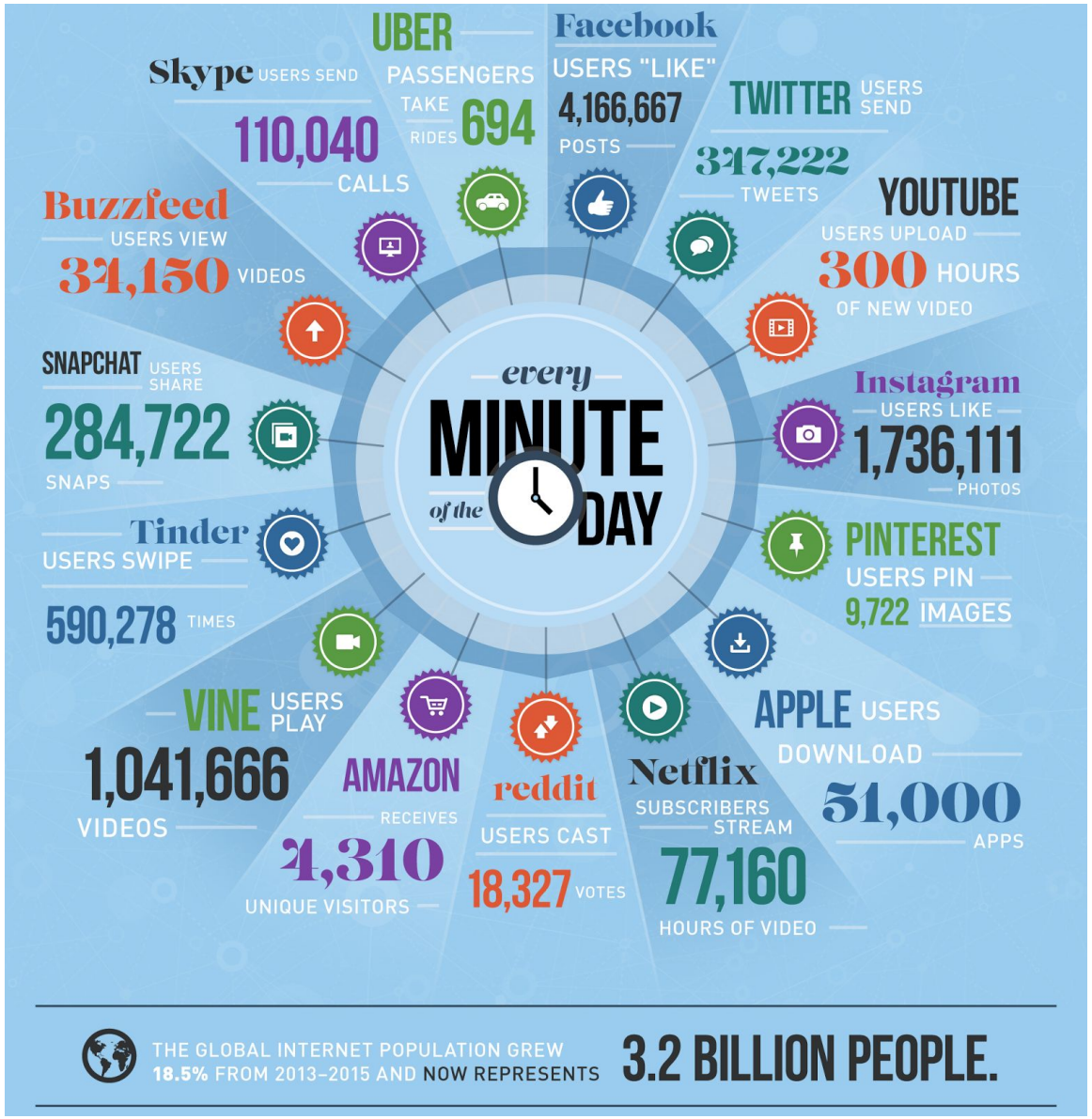
Vinícius Dias
Orientador: Dorgival Guedes

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFMG

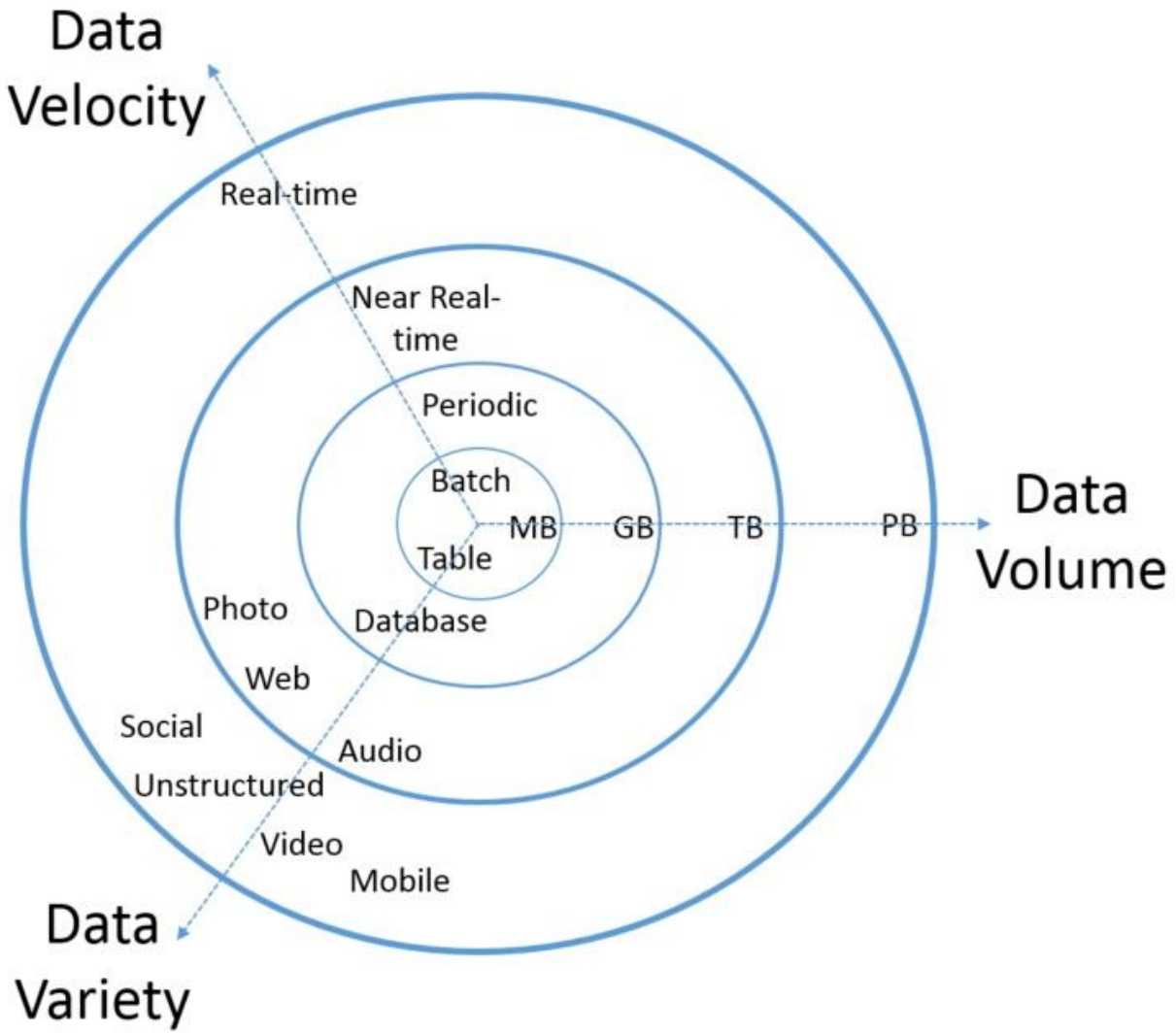
Vivemos em um mundo de dados...

De que volume de dados estamos falando?



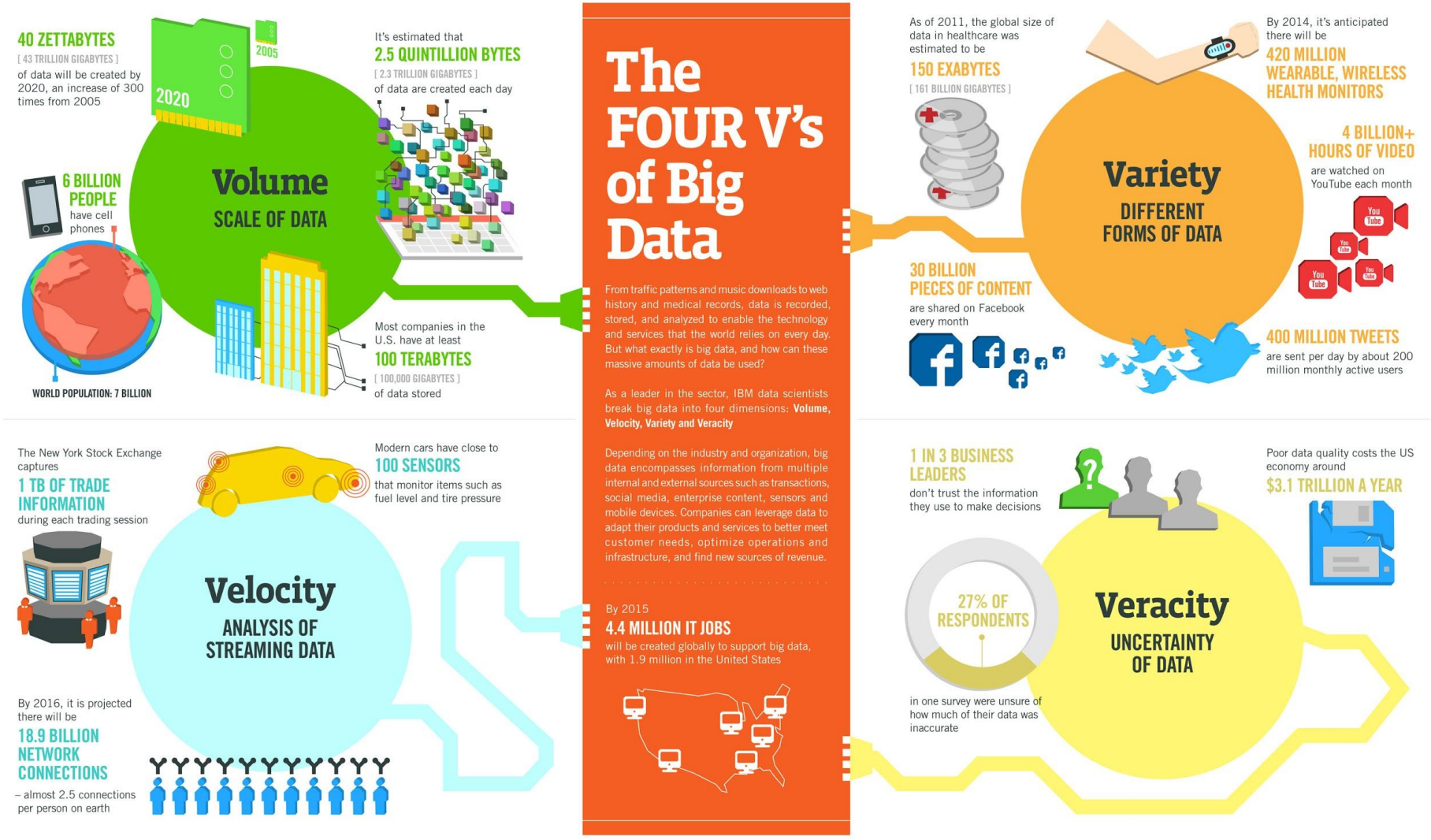
Domo Data Never Sleeps 3.0

Mas não só "volume"
3 V's de Big-Data

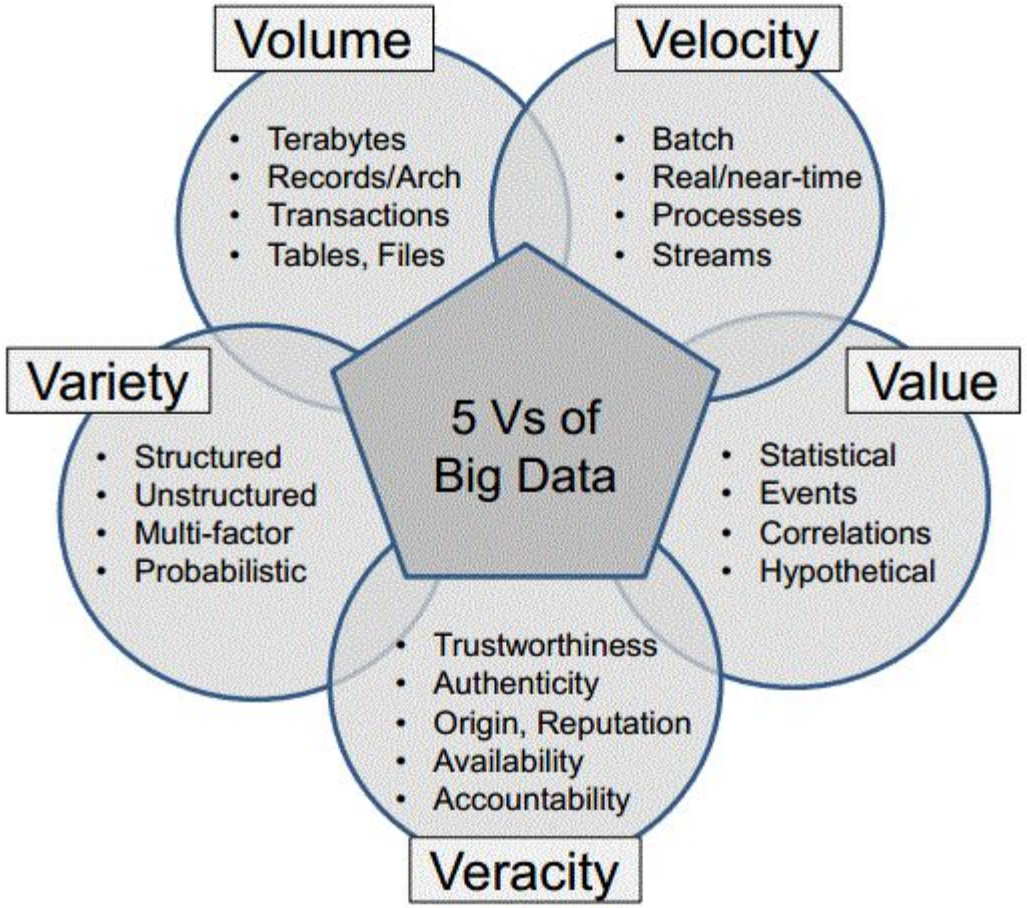


"Big Data, real world", Linguamatics

… ou 4 V's ?


40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005
2020
2005
6 BILLION PEOPLE
have cell phones
WORLD POPULATION: 7 BILLION
Volume
SCALE OF DATA
It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day
Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored
The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session
Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure
Velocity
ANALYSIS OF STREAMING DATA
By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth
The Four V's of Big Data
From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?
As a leader in the sector, IBM data scientists break big data into four dimensions: Volume, Velocity, Variety and Veracity
Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.
By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States
As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]
30 BILLION PIECES OF CONTENT
are shared on Facebook every month
Variety
DIFFERENT FORMS OF DATA
By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS
4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month
400 MILLION TWEETS
are sent per day by about 200 million monthly active users
1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions
27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate
Veracity
UNCERTAINTY OF DATA
Poor data quality costs the US economy around
$3.1 TRILLION A YEAR
Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS
IBM

… ou 5 V's ?



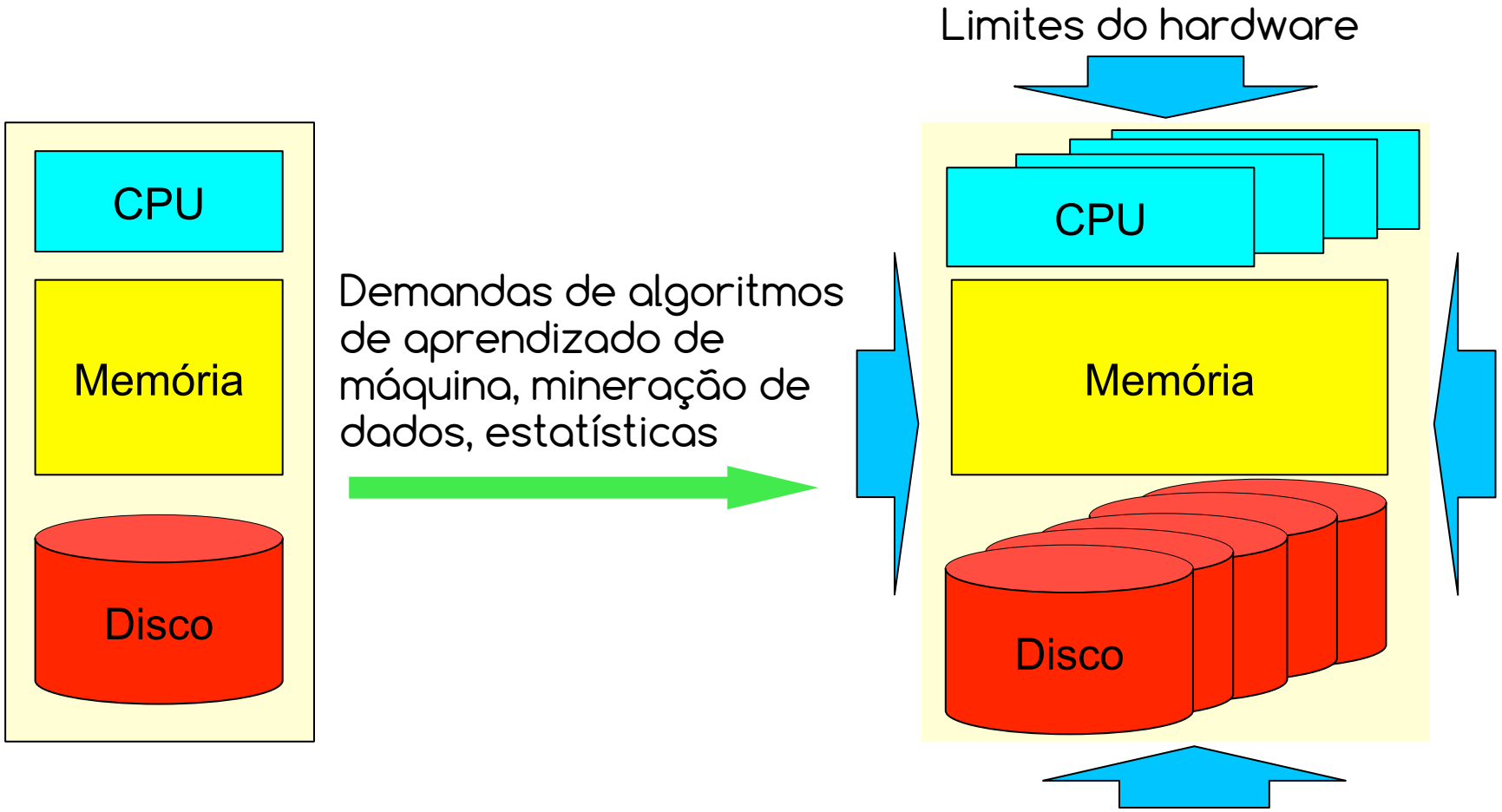
Hadoop Big data Course, *iiht.com*

Um consenso

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

Thomas Siebel, Diretor do AMPlab, Universidade de Berkeley

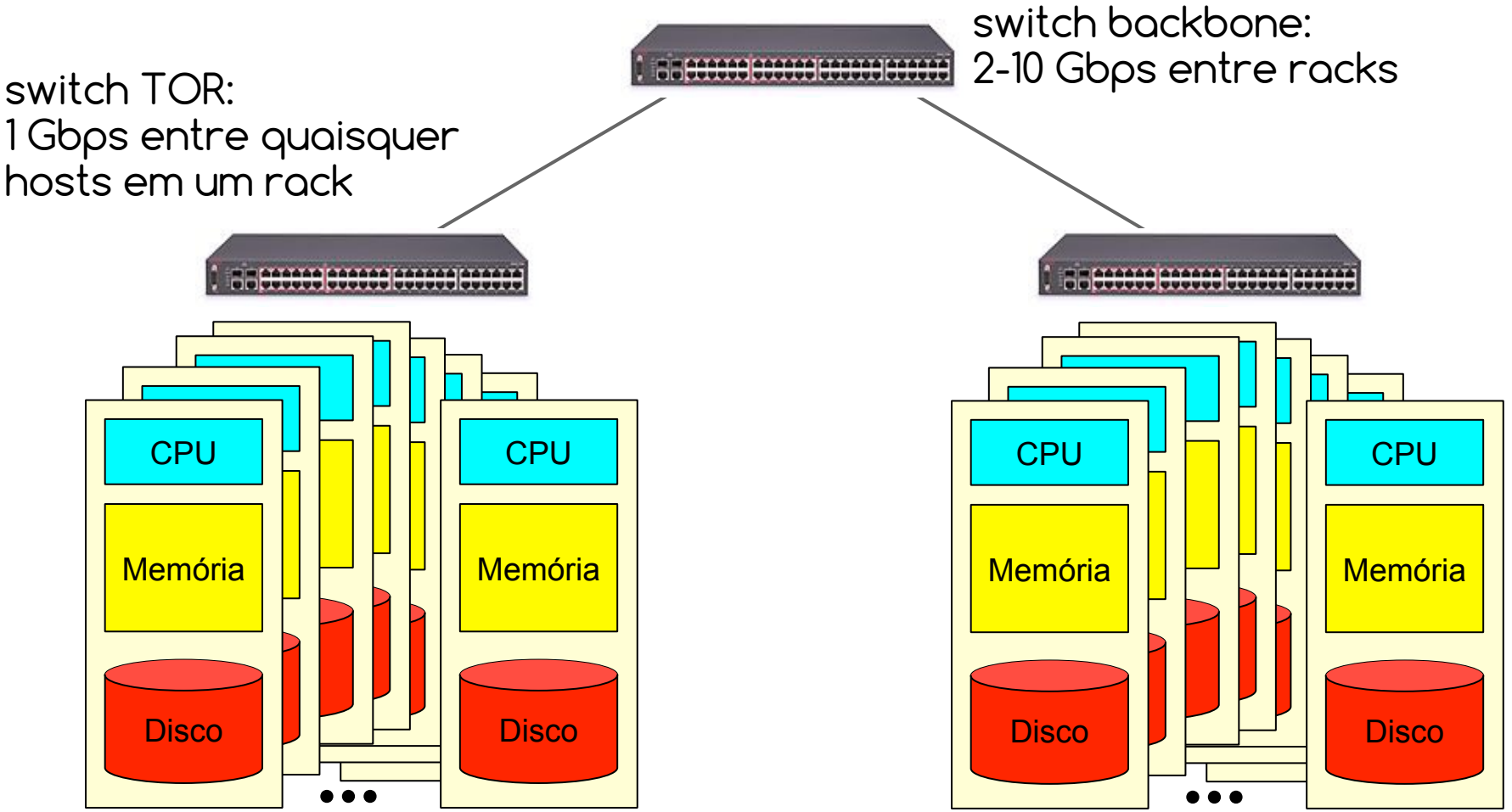
Arquiteturas tradicionais têm seus limites



Grandes ideias

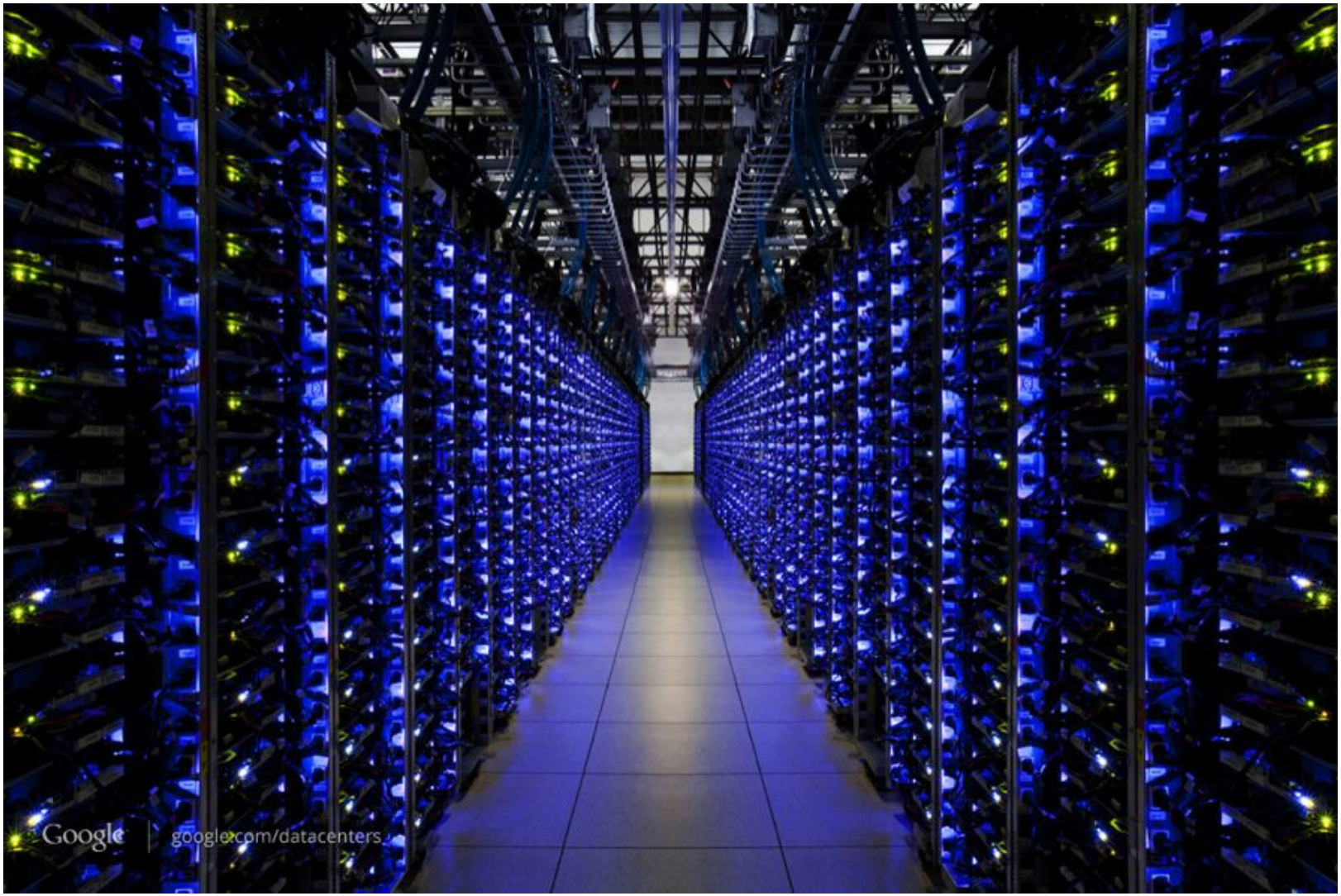
- Crescer (scale) "out", não "up"
  - Múltiplas máquinas em um cluster/datacenter
  - Solução barata
- Mover o processamento para os dados
  - A conexão entre máquinas tem banda limitada
- Processar os dados sequencialmente, se possível
  - "*Seeks*" são caros, mas a taxa de transferência é OK

Solução: agregados de computadores



cada rack contém de 16 a 64 nós

“O datacenter é o computador”

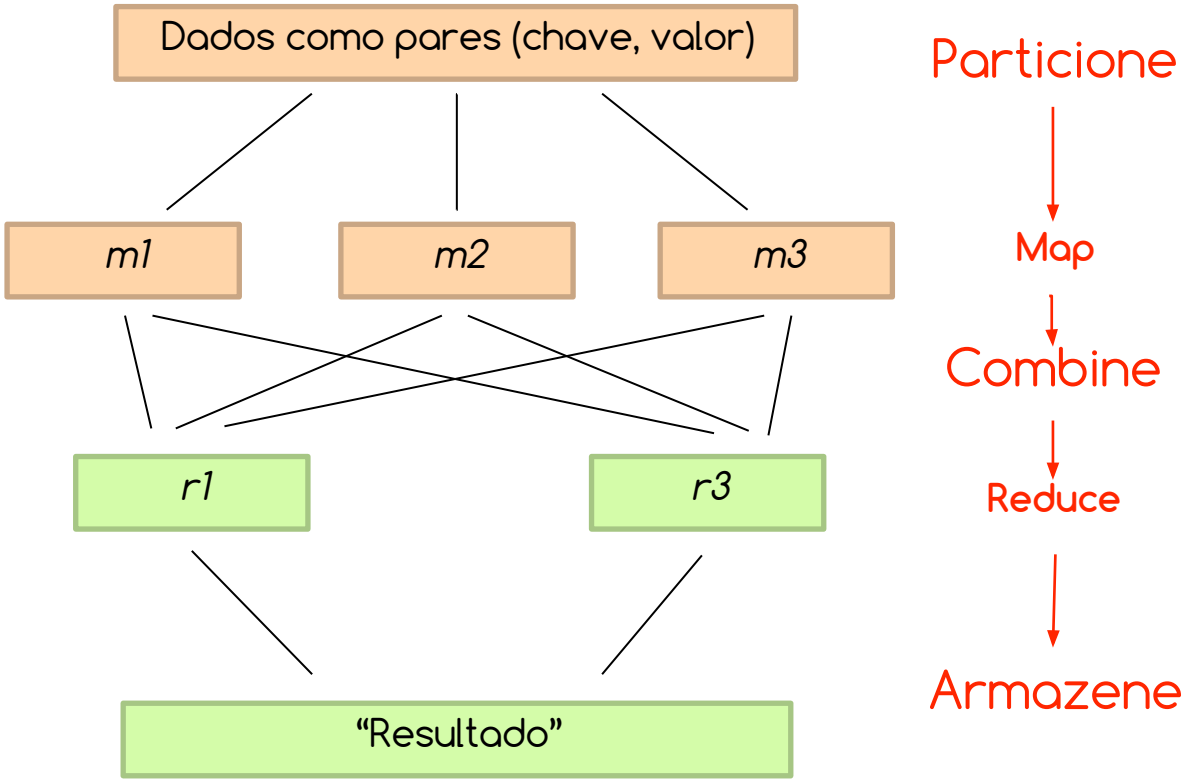


Roteiro

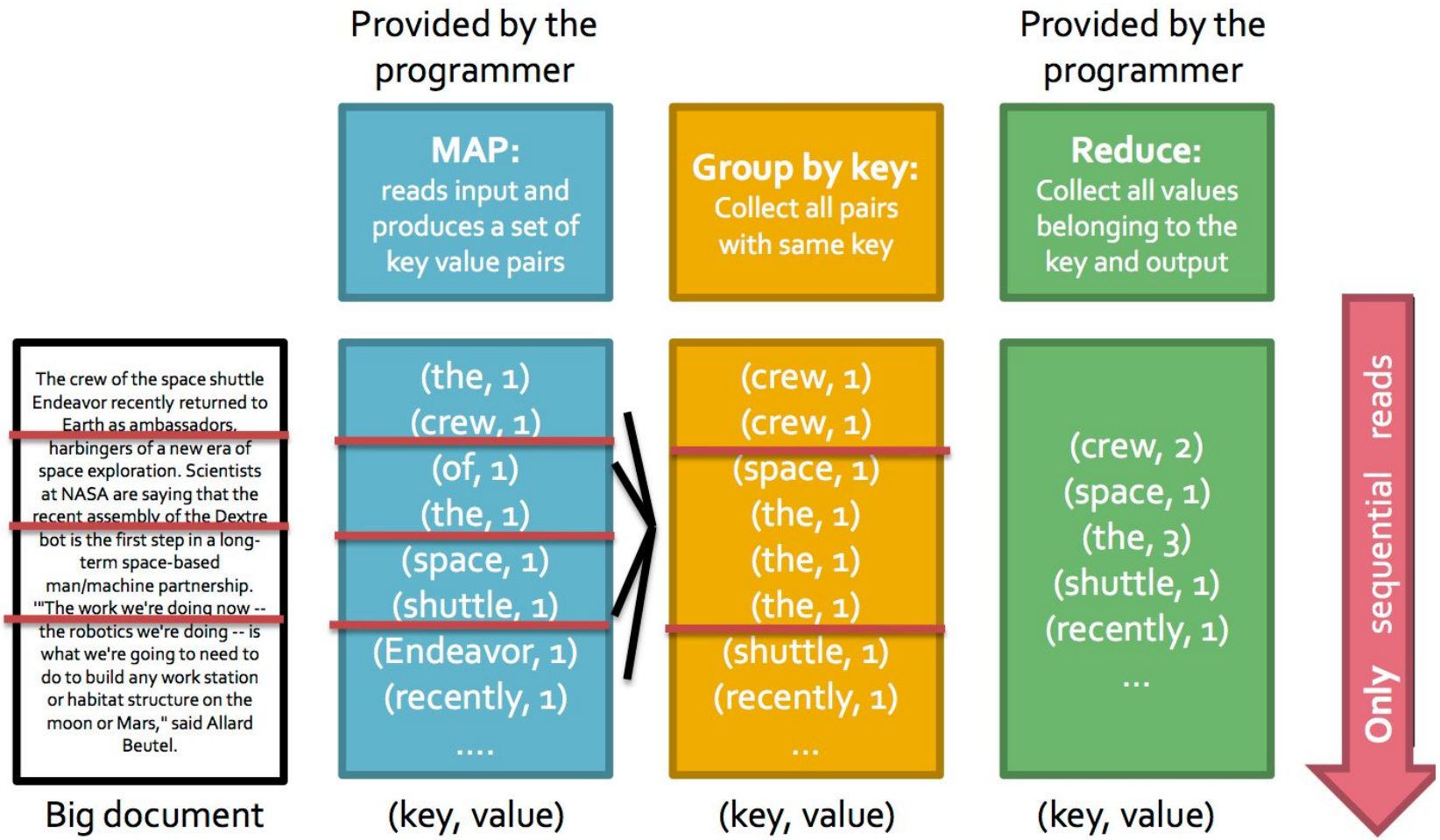
1. Motivação e problema
2. **Map/Reduce: a primeira solução**
3. Spark: uma solução genérica e integrada
4. Componentes de execução
5. RDDs e operações
6. Considerações finais

O modelo Map-Reduce (Google, OSDI 2004)

Dividir para conquistar!



Conta palavras com MapReduce




NACS
BIG DATA
Harvard Business Review
GETTING CONTROL OF BIG DATA
The Economist
The data deluge
Biomedical Computation REVIEW
Big Data Analytics in Biomedical Research
COMMUNICATIONS ACM
nature
SCIENCE IN THE PETABYTE ERA
LINUX MAGAZINE
PMKT21
Desvendando a Era do Big Data
hadoop
veja
ENTENDA O QUE É BIG DATA
FOREIGN AFFAIRS
The Rise of Big Data
Government Executive
BIG DATA

M-R não resolve todos os problemas

- Processamento de consultas
  - Às vezes, tudo que se precisa é uma consulta SQL
- Processamento iterativo
  - Algoritmos que não se resumem a um único M-R
- Processamento de *streams*
  - Nem sempre o arquivo é a forma da entrada
- Processamento de grafos (redes complexas)
  - Estruturas irregulares que afetam o fluxo dos dados e do processamento

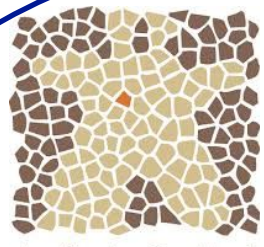
Para cada problema, a ferramenta certa!

A "família" cresceu

Algoritmos

Consultas

Grafos



Modelos de programação diferentes

Streams

(Heron)

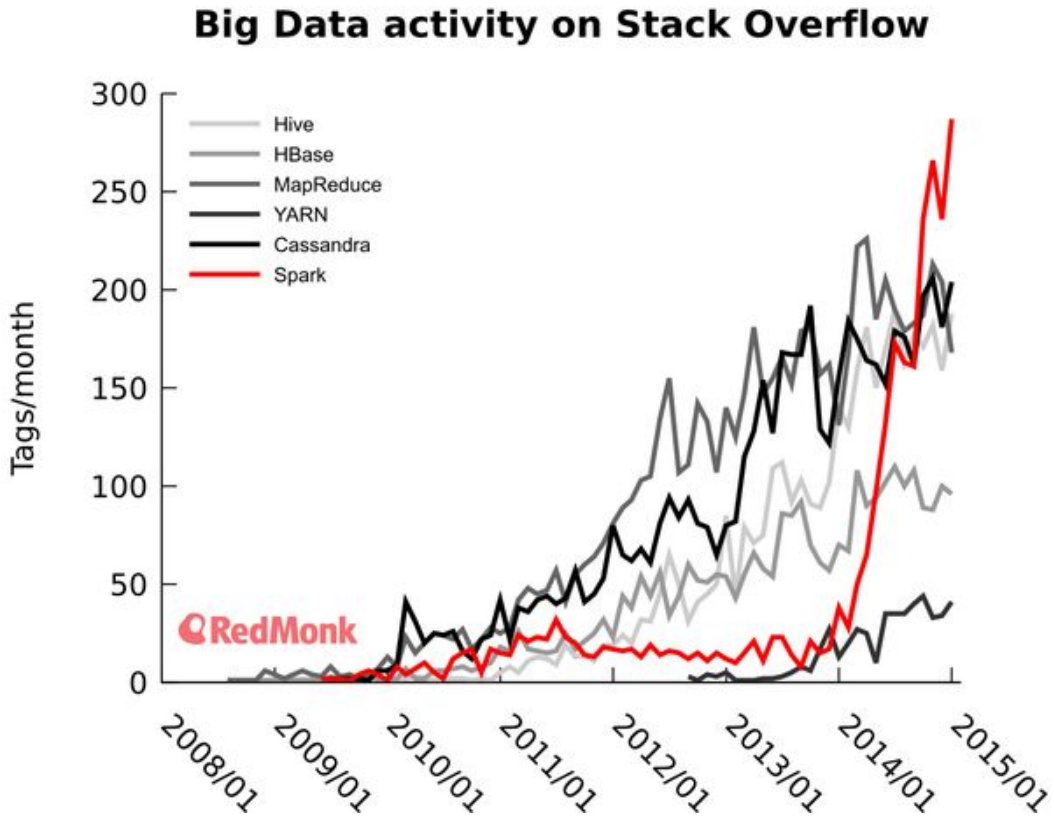
Roteiro

1. Motivação e problema
2. Map/Reduce: a primeira solução
3. **Spark: uma solução genérica e integrada**
4. Componentes de execução
5. RDDs e operações
6. Considerações finais

Spark: uma solução integrada

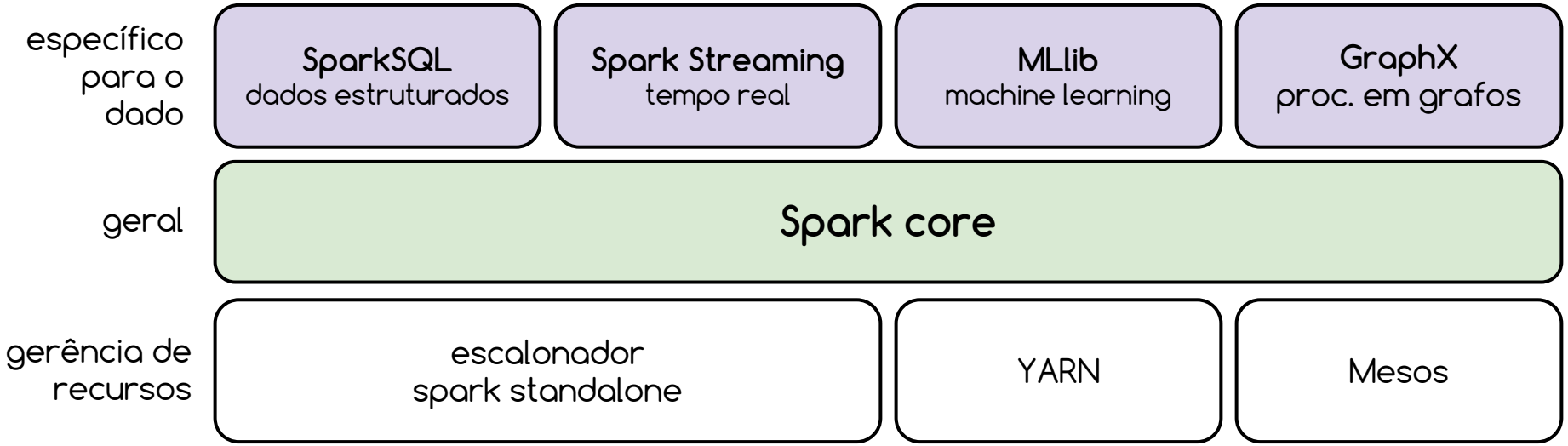
- Plataforma de computação em *clusters*.
  - criada para ser rápida e de propósito geral.
- O processamento é multiestágio:
  - Representado como grafo direcionado e acíclico (DAG).
  - Não apenas um par Map/Reduce
  - Processamento em memória (diferente do Hadoop).
- Toda a computação acontece em função de estruturas de dados denominadas RDDs (*Resilient Distributed Dataset*)

Popularidade do Spark



fonte: Donnie Berkholz, RedMonk

Arquitetura



- **geral:** processamento em DAGs, com chave/valor e/ou particionamento.
- **específico para o dado:** conhecimento sobre a estrutura do dado possibilita especializar o *Spark core* e ganhar em desempenho.
- **gerência de recursos:** é preciso orquestrar a concessão de recursos físicos das máquinas do *cluster* (memória, CPU-cores e disco).

Implementação e API's

- Spark é implementado em Scala:
  - executa sobre a JVM
  - funcional + orientada a objetos
  - também permite programação procedimental

- Linguagens com API para Spark:
  - Scala (nativo)
  - Python
  - Java
  - R (DataFrames/SparkSQL, principalmente)

Utilizaremos a API de Scala para os exemplos, principalmente.

Ferramentas de trabalho

Investigação interativa: spark shell

Desenvolvimento de aplicação: spark submit

Exemplo: selecionando itens

Quantas linhas contêm a palavra Python?

```
$ ./bin/spark-shell
…
scala> val lines =
         sc.textFile(“file:///home/pdm/README.md”)
lines: org.apache.spark.rdd.RDD[String] = ...

scala> val pythonLines =
                lines.filter(line => line.contains("Python"))
pythonLines: org.apache.spark.rdd.RDD[String] = ...

scala> pythonLines.first()
res1: String = high-level APIs in Scala, Java, and Python, and ...
```

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.

Roteiro

1. Motivação e problema
2. Map/Reduce: a primeira solução
3. Spark: uma solução genérica e integrada
4. **Componentes de execução**
5. Modelo de programação: RDDs e operações
6. Considerações finais

SparkContext

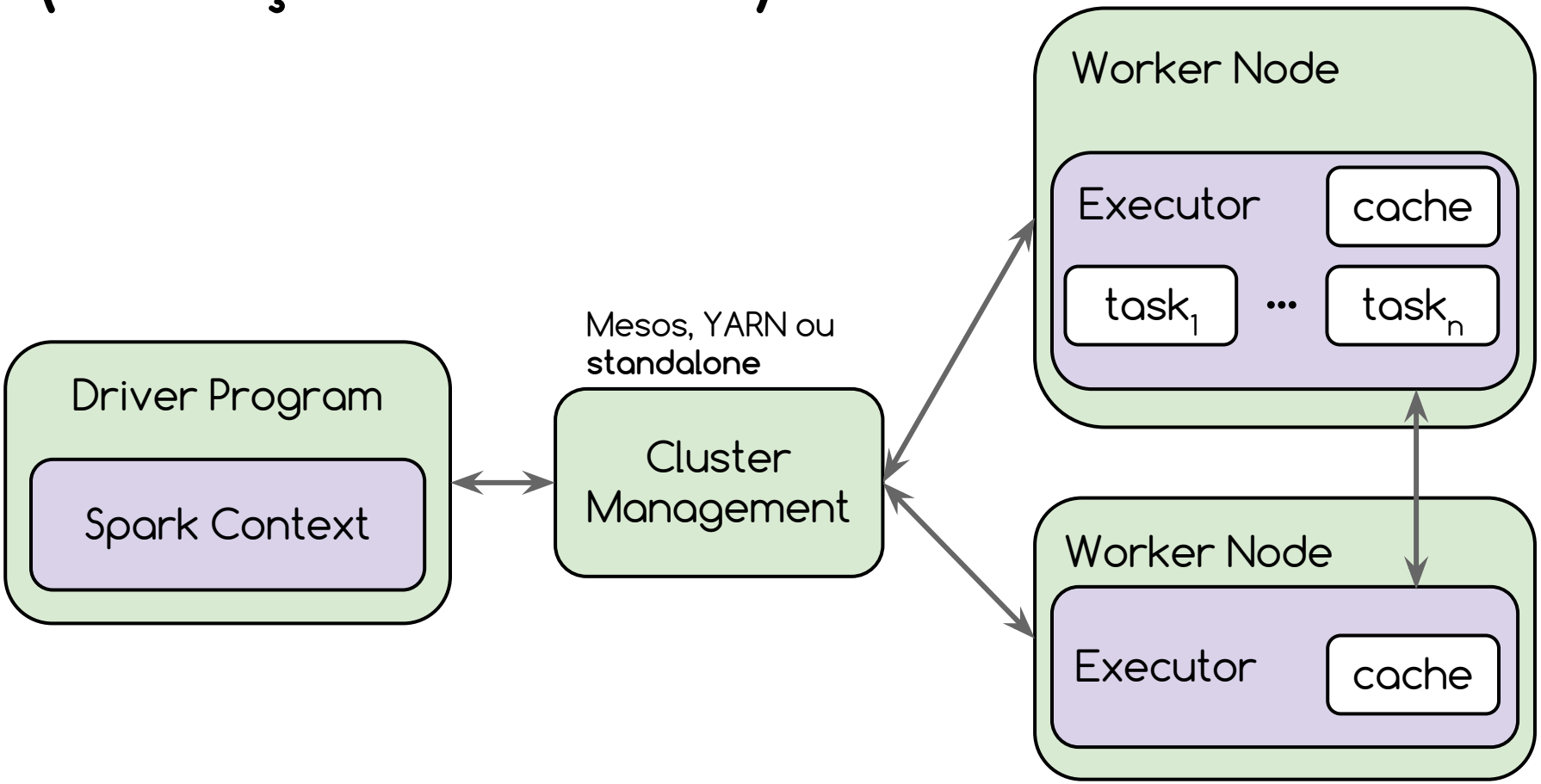
- É a interface entre o *driver* e recursos.
- *spark-shell:* o contexto é automaticamente instanciado como 'sc':

```
scala> sc
res0:org.apache.spark.SparkContext=...
```

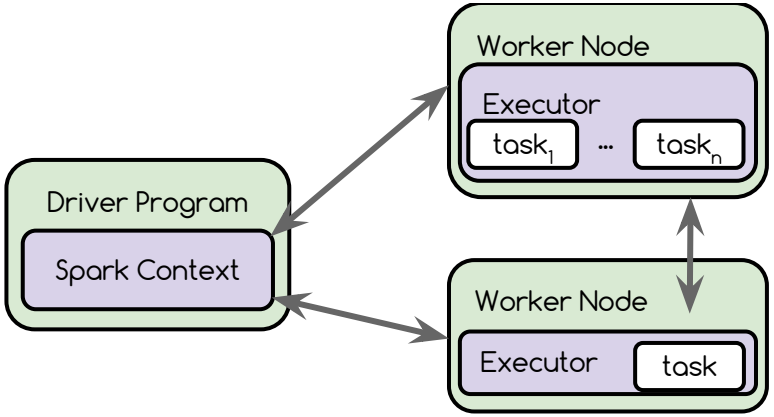
- *spark-submit:* o contexto precisa ser instanciado manualmente:

```
object MySparkDriver {
    def main(args: Array[String]) {
        val sc = new SparkContext (conf)
        …
        sc.stop()
    }
}
```

Conceitos básicos
(execução em *cluster*)



Os *executors*



- Realizam computação paralela para o *driver*.
  - seu tempo de vida está associado a uma aplicação.
- A computação paralela é no nível de tarefas.
  - tarefas computam pedaços de uma coleção.
  - é OK associar uma tarefa a um core virtual.
- A memória do *executor* é particionada em:
  - área de *caching:* dados em memória.
  - área necessária para executar tarefas.
- Na execução do tipo local existe apenas um executor acoplado ao driver (*default*).

Comandos

spark-submit **ou** spark-shell

```
$ ./bin/spark-submit \
    --class <classe_da_aplicação> \
    --master <url_do_master> \
    --executor-memory <mem_por_executor> \
    --executor-cores <num_vcores> \
    <jar_da_aplicação> \
    <parâmetros_da_aplicação>

$ ./bin/spark-shell \
    --master <url_do_master> \
    --executor-memory <mem_por_executor> \
    --executor-cores <num_vcores> \
```

Execução de aplicações (URLs)

URL em *--master URL*	**descrição**
local	**sem paralelismo**
local[K]	**K *worker threads***
local[*]	uma *worker thread* por *vcore*
local-cluster[E,C,M]	E executors, C cores cada, M MB de memória
spark://host:port	standalone (port padrão é 7077)
mesos://host:port	sobre o mesos
yarn-client	sobre o yarn (apenas executors)
yarn-cluster	sobre o yarn (inclusive driver)

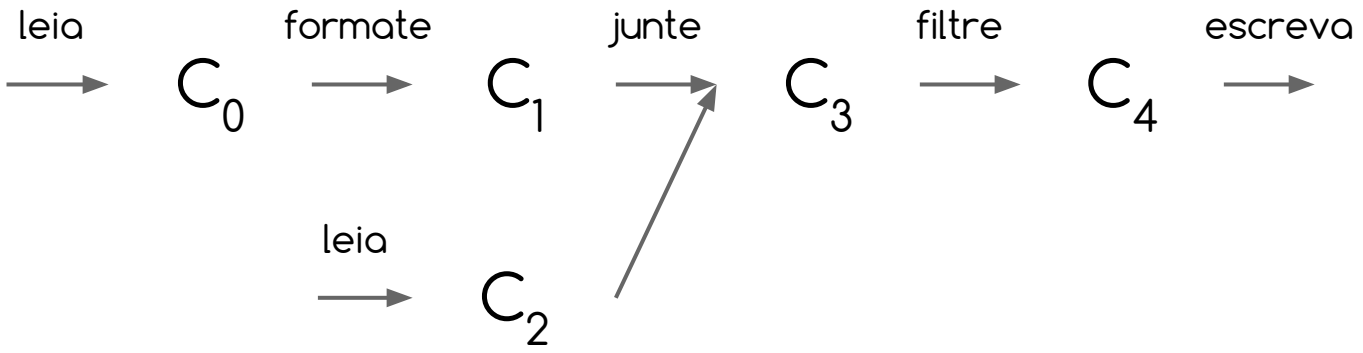
Roteiro

1. Motivação e problema
2. Map/Reduce: a primeira solução
3. Spark: uma solução genérica e integrada
4. Componentes de execução
5. **Modelo de programação: RDDs e operações**
6. Considerações finais

Modelo de programação

A aplicação é uma DAG de coleções e operações

1. Leia uma base de dados (resultado: C_0).
2. Formate C_0 (resultado: C_1).
3. Leia outra base de dados (resultado: C_2).
4. Faça uma junção das bases C_1 e C_2 (resultado: C_3).
5. Então, filtre o C_3 (resultado: C_4) e escreva no disco.

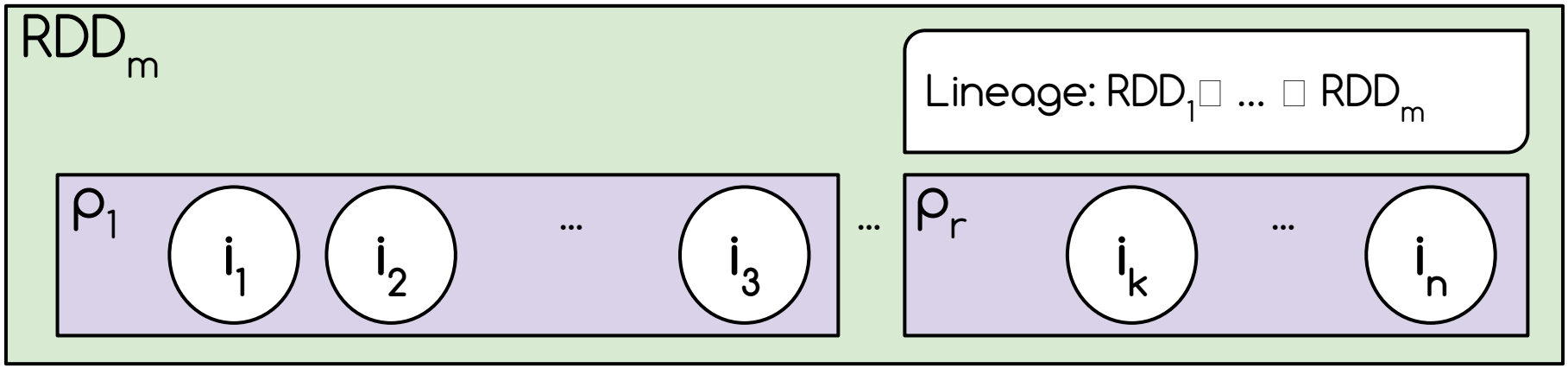


Coleção de dados em Spark

RDD: *Resilient Distributed Dataset*

- RDD é:
  - Uma abstração para trabalhar com grandes conjuntos de dados (dataset)
  - Um tipo de dados que pode ser manipulado pela API Spark nas diversas linguagens

Conceitos básicos



- Imutável
- Tolerante a falhas
- Partição: unidade de persistência e computação.
- Persistência (memória, disco, serializado, etc.).
- O RDD deste exemplo tem:
  - **n** itens, **r** partições e dependência de profundidade **m**.

Criação de RDDs: arquivos

- Essa alternativa já foi vista:

```
val linesRDD =
            sc.textFile(“file:///caminho/para/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*.

```
// ...
// essa coleção é local, sem Spark por aqui.
val bigRange = (1 to 1000000)

// aqui existe Spark (RDD)
val bigRangeRDD = sc.parallelize (bigRange)
// ...
```

Operações

- Qualquer operação sobre um RDD se enquadra em uma das categorias:
  - Transformação.
    - criam um novo RDD a partir de outro.
    - avaliação é preguiçosa (lazy).
  - Ação.
    - retornam resultado para o *driver*.
    - avaliação é imediata.

Esclarecimentos e exemplo

- Transformação já vista: `filter`.
- Ações já vistas: `count` e `first`.

```
val linesRDD = sc.textFile(“file:///caminho/para/README.md”)
    // criação
    // sem computação, apenas o lineage do RDD foi registrado.

val xlinesRDD = linesRDD.filter(line => line.contains("x"))
    // transformação
    // sem computação, apenas o lineage do RDD foi registrado.

val nxLines = xlinesRDD.count()
    // ação
    // ocorre o disparo de uma computação
    // em especial, do xlinesRDD
```

Mais operações

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

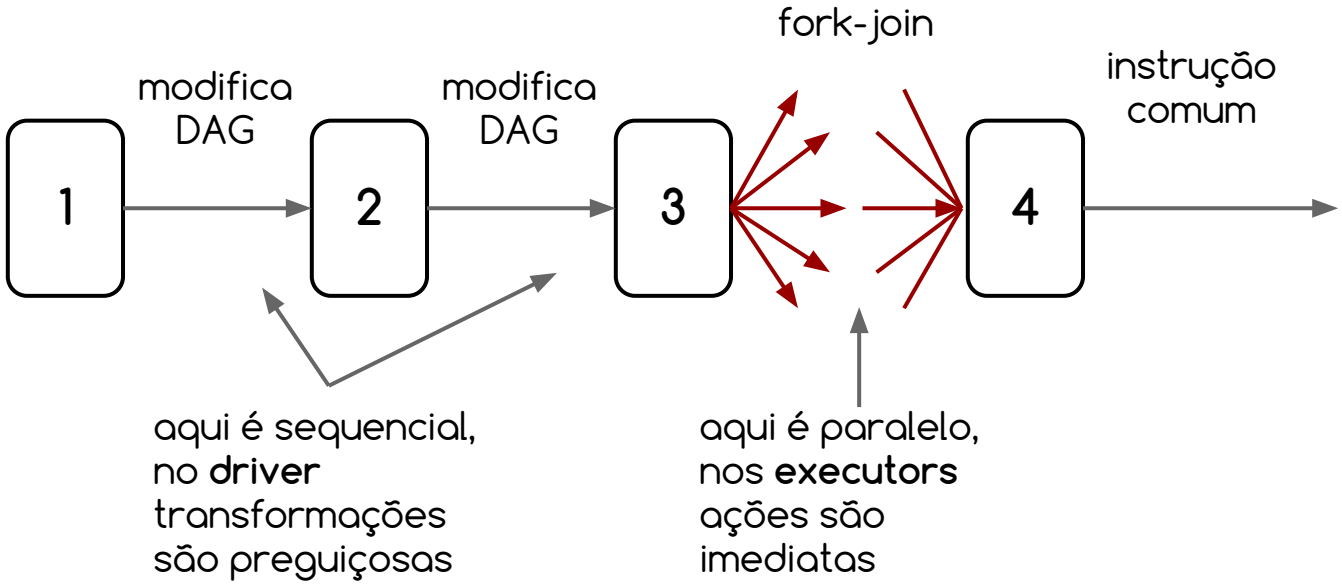
- reduce(f) - ação
  - Aplica a função `f()` a todos os elementos do RDD "de uma vez".
  - Por exemplo, (_ + _) significa "some todos os elem"
  - A função tem que ser associativa.

Atenção: estes **não são** os mesmos do Map-Reduce/Hadoop

Avaliação preguiçosa

```
1.  val exRDD = sc.parallelize (1 to 10)
2.  val incExRDD = exRDD.map (n => n+1)
    // ação reduce dispara a computação
3.  val sum = incExRDD.reduce (_ + _)
4.  println ("soma = " + sum)
```

Atenção: estes **não são** os mesmos Map/Reduce de hadoop



Mais operações

transformações: `flatMap` e `ReduceByKey`
ação: `take`

- `rdd.flatMap(func)`: mapeamento um-para-muitos
  - `func`: função que recebe um elemento e mapeia para vários, potencialmente.

- `pairRdd.reduceByKey(func)`: combina todos os valores de mesma chave.
  - `func`: recebe dois valores e retorna um terceiro valor representando a combinação dos dois primeiros.

- `rdd.take(n)`: coleta n itens do `rdd` para o *driver*.

WordCount em Spark

```
val lines = sc.textFile (inputFile)
    // cada item do RDD é uma linha do arquivo (String)
```

WordCount em Spark

```
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
```

WordCount em Spark

```
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)
```

WordCount em Spark

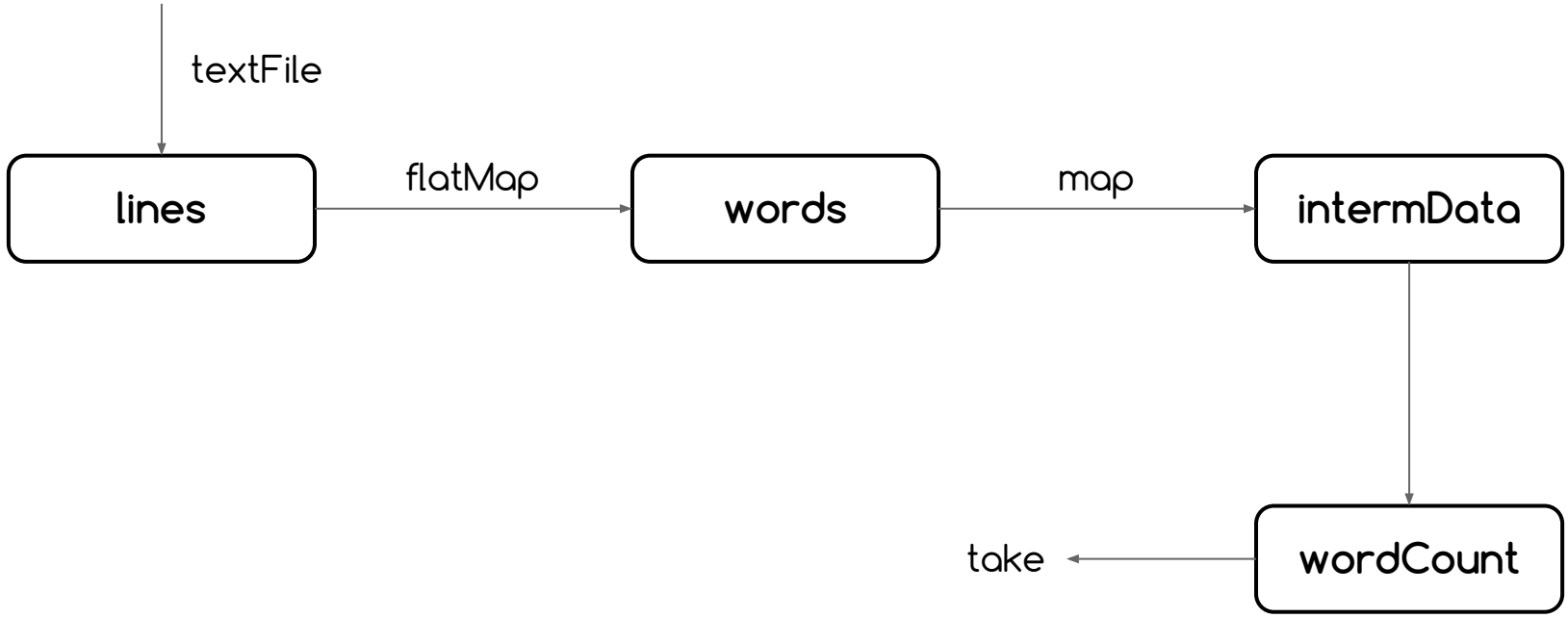
```
val lines = sc.textFile (inputFile)
    // cada item do RDD é uma linha do arquivo (String)
val words = lines.flatMap (line => line.split (" "))
    // cada item do RDD é uma palavra do arquivo
val intermData = wordsRDD.map (word => (word,1))
    // cada item do arquivo é um par (palavra,1)
val wordCount = intermData.reduceByKey (_ + _)
    // cada item do RDD contém a ocorrência final de cada
    // palavra.
```

WordCount em Spark

```
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 a ocorrência final de cada
    // palavra.
val 5contagens = wordCount.take (5)
    // 5 resultados no programa driver
```

WordCount em Spark

```
val lines = sc.textFile (inputFile)
val words = lines.flatMap (line => line.split (" "))
val intermData = words.map (word => (word,1))
val wordCount = intermData.reduceByKey (_ + _)
val 5contagens = wordCount.take (5)
```



WordCount em Hadoop

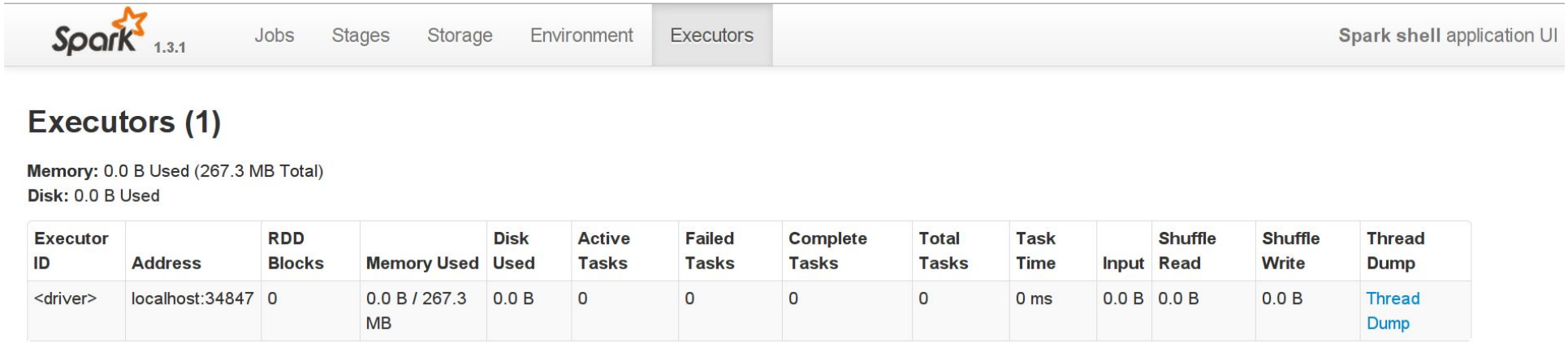
- Para quem já teve contato com Hadoop, deve ter percebido o ganho em simplicidade. Compare:

```
public class WordCount {

 public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }
```

O Código em Hadoop

```
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable>{
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
           sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
   Job job = new Job(conf, "wordcount");

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   job.setMapperClass(Map.class);
   job.setReducerClass(Reduce.class);
   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));
   job.waitForCompletion(true);
}
}
```

Spark application UI



Persistência

- RDDs são avaliados preguiçosamente.
- isso significa que haverá recomputação toda vez que uma ação sobre esse RDD for solicitada.
- **Exemplo:** 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(_+_))
```

Persistência (`localhost:4040`)

Spark 1.3.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Spark Jobs (?)

Total Duration: 55 s
Scheduling Mode: FIFO
Completed Jobs: 10

jobs com tempos semelhantes

Completed Jobs (10)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	reduce at <console>:24	2015/06/11 22:41:16	0.1 s	1/1	2/2
8	reduce at <console>:24	2015/06/11 22:41:16	0.1 s	1/1	2/2
7	reduce at <console>:24	2015/06/11 22:41:16	0.1 s	1/1	2/2
6	reduce at <console>:24	2015/06/11 22:41:16	0.1 s	1/1	2/2
5	reduce at <console>:24	2015/06/11 22:41:16	0.1 s	1/1	2/2
4	reduce at <console>:24	2015/06/11 22:41:15	0.1 s	1/1	2/2
3	reduce at <console>:24	2015/06/11 22:41:15	0.1 s	1/1	2/2
2	reduce at <console>:24	2015/06/11 22:41:15	0.1 s	1/1	2/2
1	reduce at <console>:24	2015/06/11 22:41:15	0.1 s	1/1	2/2
0	reduce at <console>:24	2015/06/11 22:41:15	0.5 s	1/1	2/2

API de persistência

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

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

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

⇒ **cache()** considera o nível padrão, isto é, MEMORY_ONLY.

- o mesmo que `persist(StorageLevel.MEMORY_ONLY)`

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

Persistência (depois)

- RDDs são avaliados preguiçosamente.
- isso significa que haverá recomputação toda vez que uma ação sobre esse RDD for solicitada.

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

Persistência (`localhost:4040`)

Spark 1.3.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Spark Jobs (?)

Total Duration: 32 s
Scheduling Mode: FIFO
Completed Jobs: 10

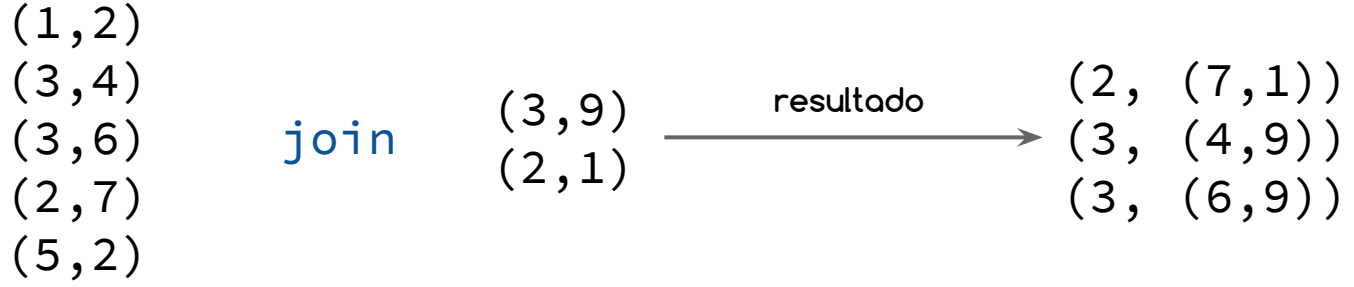
só o primeiro demandou mais tempo (1 s)

Completed Jobs (10)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	reduce at <console>:24	2015/06/11 22:42:36	37 ms	1/1	2/2
8	reduce at <console>:24	2015/06/11 22:42:36	39 ms	1/1	2/2
7	reduce at <console>:24	2015/06/11 22:42:36	39 ms	1/1	2/2
6	reduce at <console>:24	2015/06/11 22:42:36	43 ms	1/1	2/2
5	reduce at <console>:24	2015/06/11 22:42:36	89 ms	1/1	2/2
4	reduce at <console>:24	2015/06/11 22:42:36	38 ms	1/1	2/2
3	reduce at <console>:24	2015/06/11 22:42:36	38 ms	1/1	2/2
2	reduce at <console>:24	2015/06/11 22:42:36	39 ms	1/1	2/2
1	reduce at <console>:24	2015/06/11 22:42:36	0.2 s	1/1	2/2
0	reduce at <console>:24	2015/06/11 22:42:35	1 s	1/1	2/2

Mais operações
transformações: `join`

- `pairRdd.join(otherPairRdd)`:
  faz um *inner-join* entre os dois RDDs.
  - esta operação combina itens de dois RDDs de pares pela chave.
    - o resultado é um RDD contendo todas as combinações de pares de valores que compartilham uma mesma chave nos RDDs de entrada.



Estudo de caso: *pagerank*

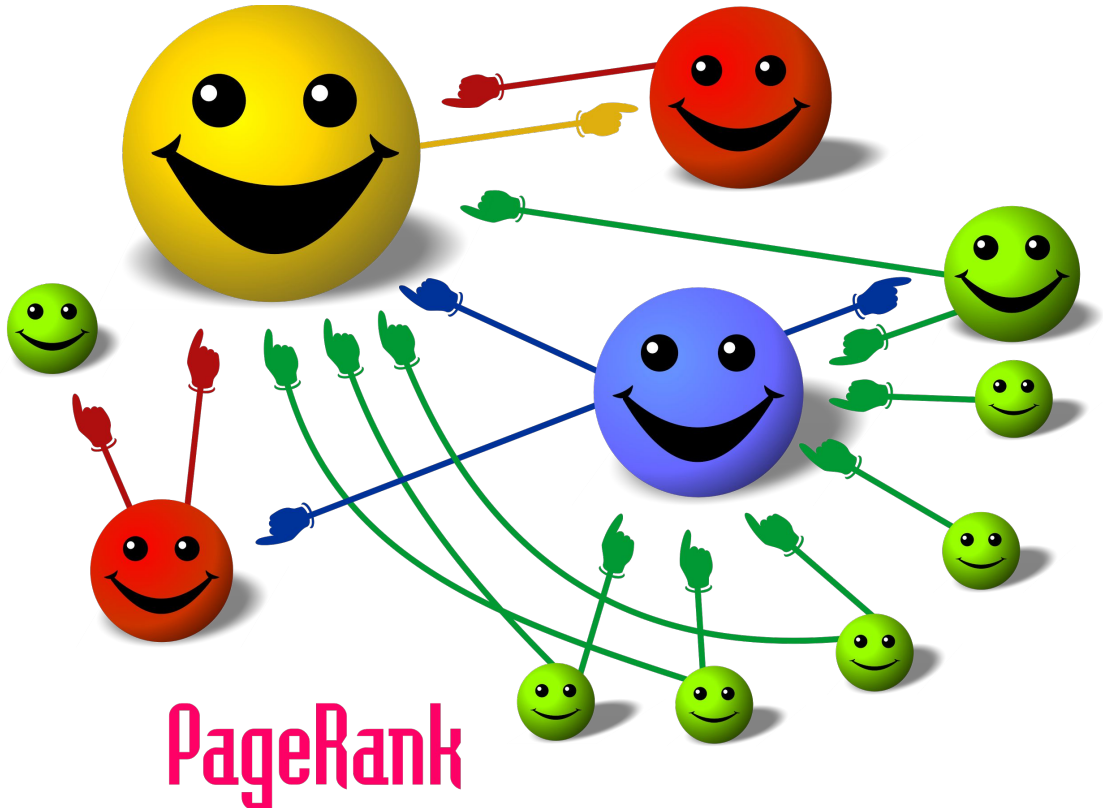
- É um exemplo clássico que mostra dois pontos fortes de Spark: *caching* e *computação iterativa*.

- **Propósito:** criar um ranqueamento de importância de nós em um grafo.
- Onde é usado?
  - o **Google search** utiliza o PageRank.
  - ele foi proposto por Larry Page, cofundador da Google.

- Sabe a ordem de links que aparecem em uma busca que você faz no Google search?
  - sim, o PageRank que foi usado para ranqueá-los.

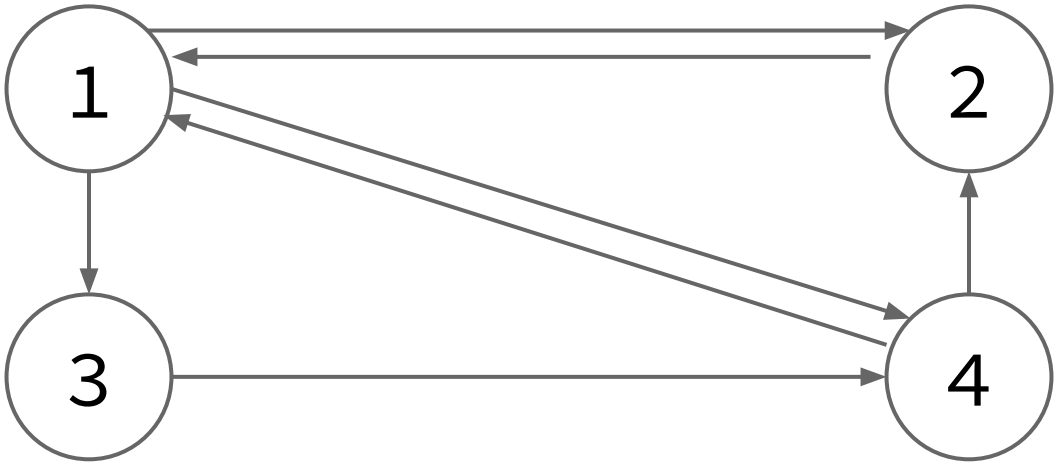
Premissa do *PageRank*

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



Descrição do algoritmo (parte 1)

- Temos uma estrutura representando páginas e para quem elas apontam.

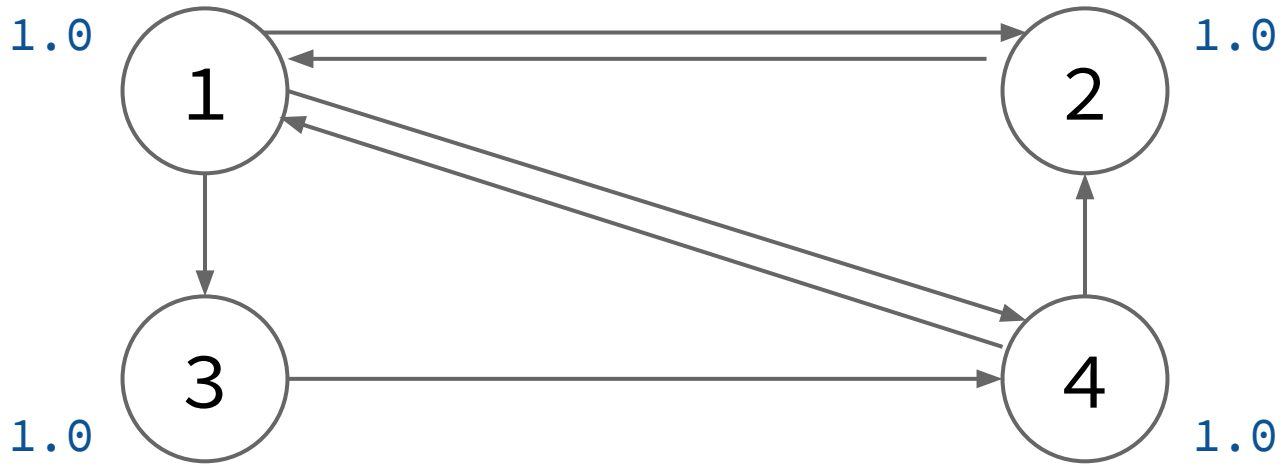


Os números poderiam representar (hipoteticamente):

1. portal.dataprev.gov.br
2. dcc.ufmg.br
3. spark.apache.org
4. vagrantup.com

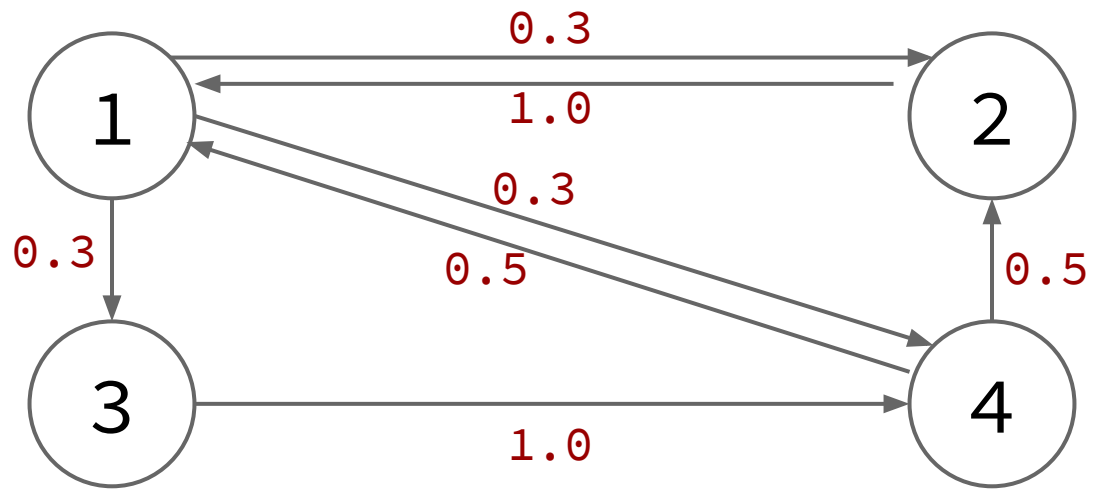
Descrição do algoritmo (parte 2)

- Todas iniciam com importância 1.0, ou seja, 100%.



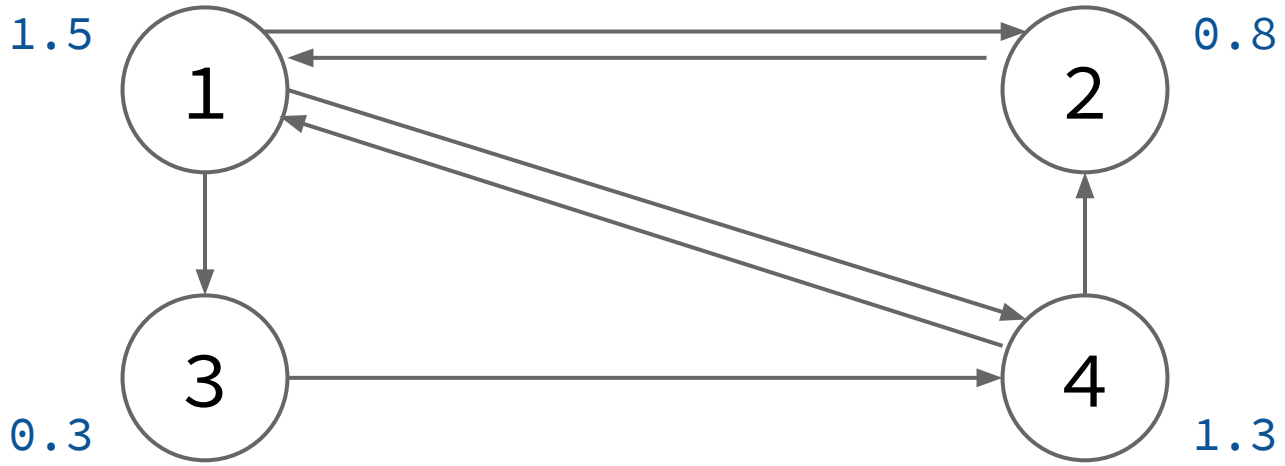
Descrição do algoritmo (parte 3)

- A cada iteração toda página distribui sua importância igualmente para os vizinhos.



Descrição do algoritmo (parte 4)

- Agora cada página tem uma nova importância, que é a soma dos valores recebidos.



- O processo se repete, por um número determinado de iterações.
- **Páginas com números maiores são mais importantes.**

PageRank: implementação

```
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.
  - é o for principal, que ocorre no driver.
- Parte paralela.
  - são as transformações a cada iteração.

PageRank: submissão

```
$ ./bin/spark-submit \
    --class PageRank \
    --master yarn-client \
    --executor-memory 2g \
    --executor-cores 4 \
    simple-spark-app_2.10-1.0.jar \
    hdfs://graphSample.txt 10
```

Roteiro

1. Motivação e problema
2. Map/Reduce: a primeira solução
3. Spark: uma solução genérica e integrada
4. Componentes de execução
5. Modelo de programação: RDDs e operações
6. **Considerações finais**

Demonstração

WordCount no spark-shell

WordCount em Spark

```
val lines = sc.textFile (inputFile)
val words = lines.flatMap (line => line.split (" "))
val intermData = words.map (word => (word,1))
val wordCount = intermData.reduceByKey (_ + _)
val 5contagens = wordCount.take (5)
```

O que mais?

Mais algumas operações

- ⇒ distinct
- ⇒ union
- ⇒ intersection
- ⇒ subtract
- ⇒ cartesian
- ⇒ sample

- ⇒ fold**ByKey**
- ⇒ combine**ByKey**
- ⇒ sort**ByKey**
- ⇒ keys
- ⇒ values
- ⇒ subtract**ByKey**
- ⇒ rightOuterJoin
- ⇒ leftOuterJoin
- ⇒ cogroup

- ⇒ take - ação
- ⇒ takeOrdered - ação

O que mais?
Desempenho

- Particionamento inteligente:
  - distribuição das chaves
  - balancemento da carga de trabalho
- Funcionalidades avançadas:
  - variáveis de brodcast
  - acumuladores
- Uso consciente de memória:
  - tamanho de resultados retornados para o *driver* através de ações.
  - capacidade *vs.* demanda para *caching*

O que mais?
Bibliotecas específicas

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

Mas tudo isso é implementado sobre os conceitos vistos nesta palestra!

Roteiro

1. Motivação e problema
2. Map/Reduce: a primeira solução
3. Spark: uma solução genérica e integrada
4. Componentes de execução
5. Modelo de programação: RDDs e operações
6. Considerações finais

Referências

- Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. (Zaharia M. et al.)
- Learning Spark. (Karau H.; Konwinski A.; Wendell P.; Zaharia M.)
- Spark Docs. (versão mais recente)
- Advanced Spark Features. (Spark Summit 2012)
- Advanced Spark. (Databricks 2014)
- Spark API. (classe RDD como ponto de partida)