Especialização em Desenvolvimento de Software *Full Stack*

Módulo : - BIG DATA

Prof. Me. Luiz Mário Lustosa Pascoal

luizmlpascoal@gmail.com

Um pouco sobre mim…

- **Graduado em Sistemas de Informação pela Universidade Federal da Grande Dourados (UFGD) - 2012**

- **Mestre em Ciência da Computação pela Universidade Federal de Goiás (UFG) - 2014**
  - Linha de Pesquisa: Sistemas de Recomendação, Computação Evolutiva e Otimização Combinatória

- **Doutorando em Ciência da Computação pela Universidade Federal de Goiás (UFG) – 2015 …**
  - Linha de Pesquisa: Sistemas de Recomendação, Otimização Interativa, Visualização da Informação e Processos Estocásticos

Docente no ensino superior:

- Professor substituto no INF/UFG (Universidade Federal de Goiás)
- FASAM (Faculdade Sul-Americana)
- Faculdade Delta

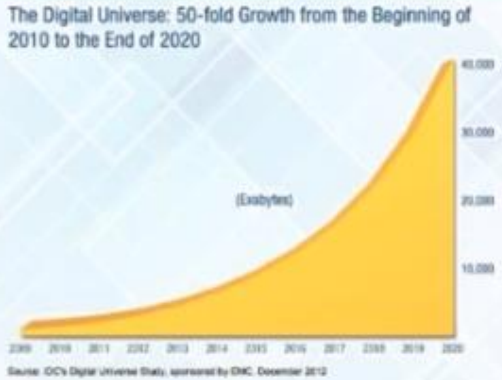
Conteúdo Programático para o Módulo

- **Big Data – entendendo o problema.**
- **Por quê Programação Paralela e Distribuída?**
- **Introdução a Programação Paralela e Distribuída.**
- **Programação Paralela**
  - Threads
  - Controle de Concorrência.
- **Programação Distribuída**
  - Modelos de sistemas distribuídos.
  - MapReduce
  - Hadoop Distributed File System
  - Spark

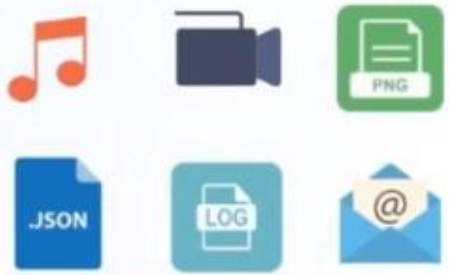
Big Data

- **O que é?**
  - Big Data é o termo que descreve o imenso volume de dados tão **grande** e **complexo**, podendo ser ***Estruturados, Semi-estruturados e Não-estruturados,*** que se torna difícil processar usando sistemas tradicionais
- **Exemplos :**
  - Facebook – 1bi de usuários por dia (4bi de *likes* e 300 milhões de fotos).
  - Netflix – 75 milhões de assinantes e 15 mil filmes em seu catálogo.
  - iOT – Internet das coisas, estimativa de 20 bilhões de dispositivos conectados a Internet em 2020.
  - *eCommerce,* Governo, *Banking, Healthcare,* Transporte.
- **Qual o objetivo?**
  - Trabalhar bem com os dados disponíveis a fim de gerar análises rápidas e eficientes.
- **Bases para o Big Data**
  - **Volume:** armazenamento de várias fontes de dados
  - **Variedade:** diferentes formatos de dados e estruturas que os dados são gerados.
  - **Velocidade:** os dados são fornecidos em alta velocidade e distinta.
  - **Valor:** capacidade de distinção do significado de dados para avaliar dados com maior valor.
  - **Veracidade:** confiabilidade nos dados de entrada.

Os 5 V's



Volume



Different kinds of data is being generated from various sources

Variety

Data is being generated at an alarming rate

Velocity

Mechanism to bring the correct meaning out of the data

Value

Min	Max	Mean	SD
4.3	?	5.84	0.83
2.0	4.4	3.05	5000000
15000	7.9	1.20	0.43
0.1	2.5	?	0.76

Uncertainty and inconsistencies in the data

Veracity

. . . . . .

V's associated with Big Data may grow with time

Big Data como oportunidade...

Retirado do canal edureka! (2017)

Big Data

- **Tecnologias envolvidas em Big Data**
  - Sistemas de arquivos Distribuídos (HDFS)
  - Processamento Paralelo
  - Computação em Nuvem (*cloud computing*)
  - Grids de Mineração de Dados
  - Redes de Alta Velocidade
  - Sistemas de Armazenamento escaláveis
  - Paradigma de programação específico.
  - *Machine Learning*

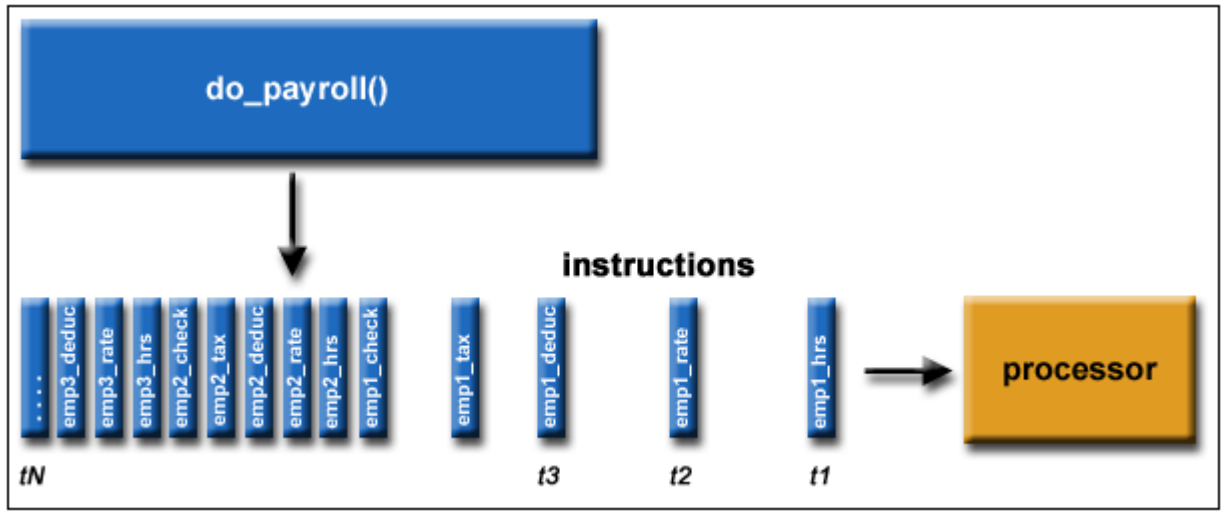
Tudo isso é incrível, porém como posso usar?

Vamos por partes...

Programação Paralela e Distribuída

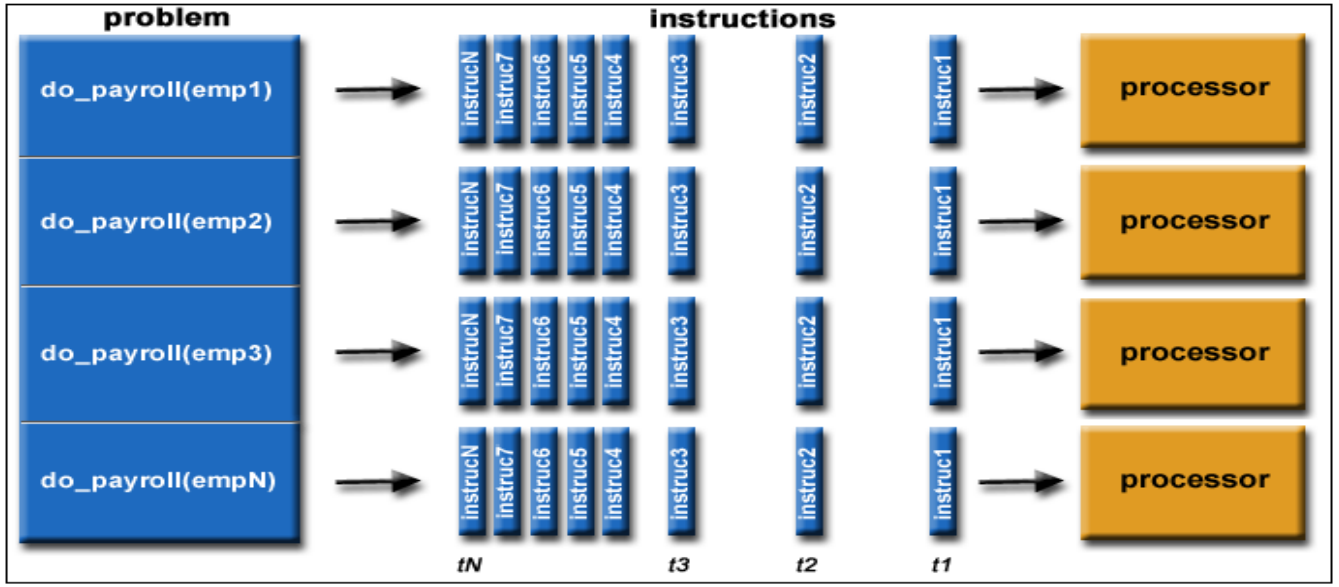
- **O que é um Programação Concorrente?**
  - Um programa concorrente é um **conjunto de programas sequenciais ordinários** os quais são executados em uma abstração de ***paralelismo***.
- **Paralelismo** é o uso simultâneo de múltiplos recursos computacionais para resolver um **problema**.
- O **problema deve ser capaz de:**
  - Ser quebrado em diversas partes para ser resolvido simultaneamente.
  - Executar múltiplas instruções computacionais a qualquer momento.
  - Ser resolvido em menos tempo com múltiplos recursos computacionais do que com um único recurso.

Execução: Serial x Paralela



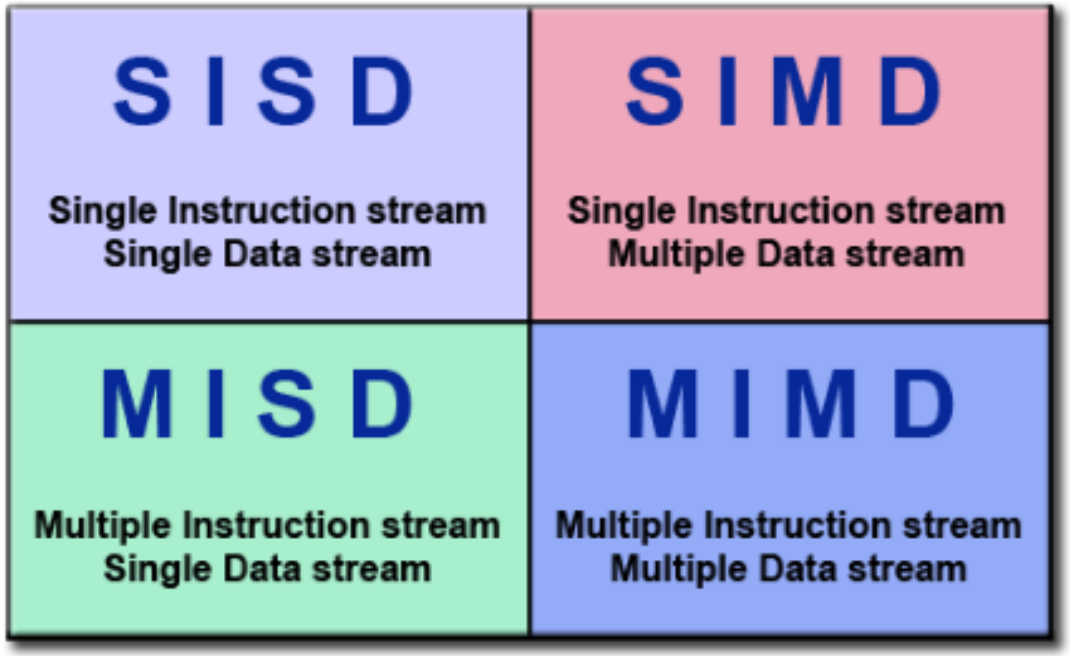
Recursos Computacionais geralmente são:

- Único computador com múltiplos núcleos de processamento (Paralelo).
- Vários computadores conectados por meio de uma rede (Distribuído).

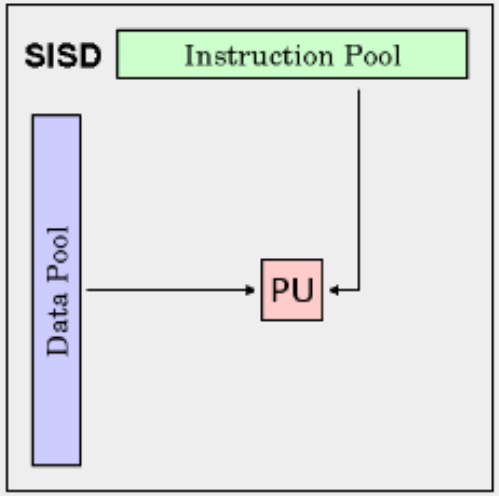


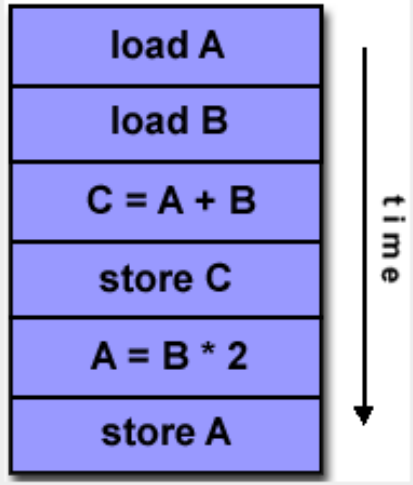
Taxonomia de Flynn

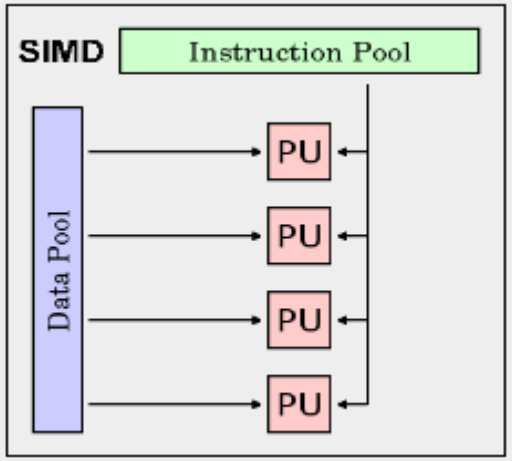
Distingue arquiteturas de computadores multi-core de acordo com o **Fluxo de Instruções** e **Fluxo de Dados.**

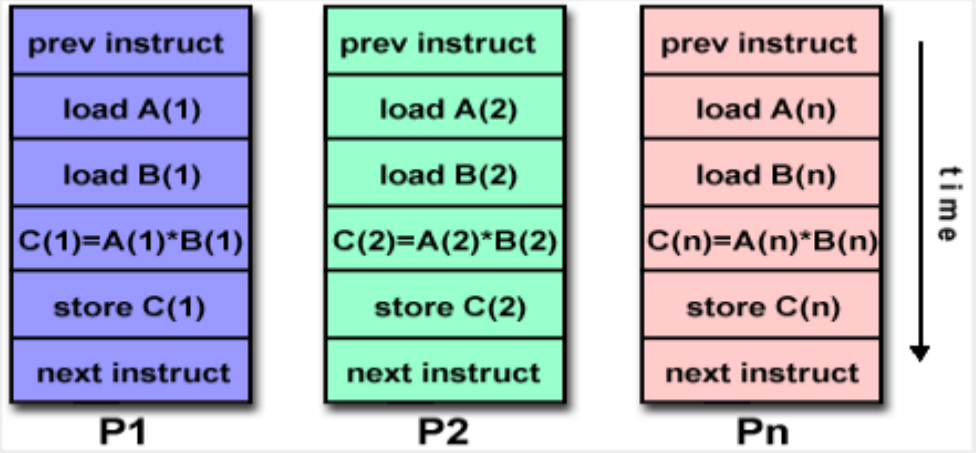


Funcionamento dos modelos

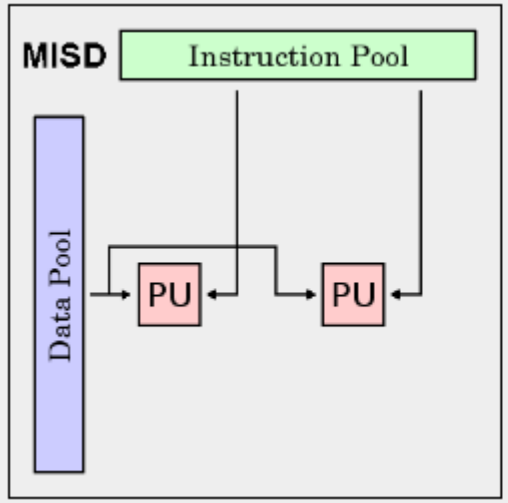


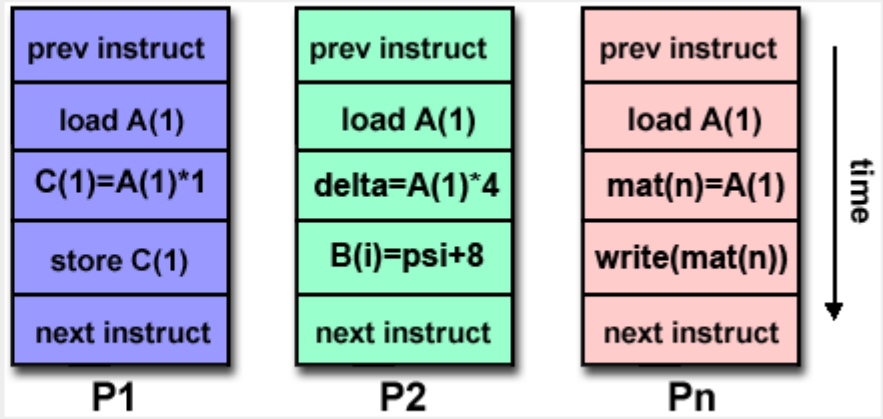


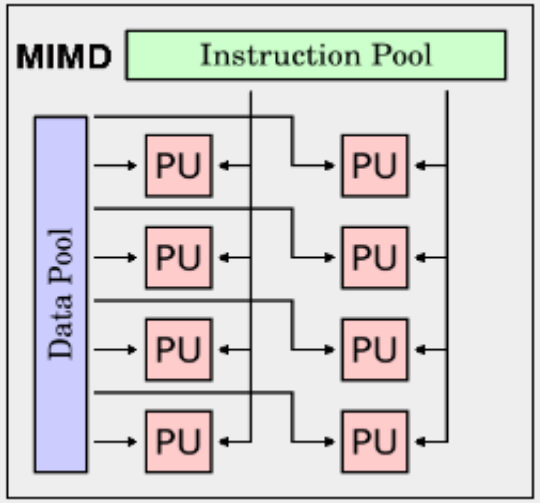


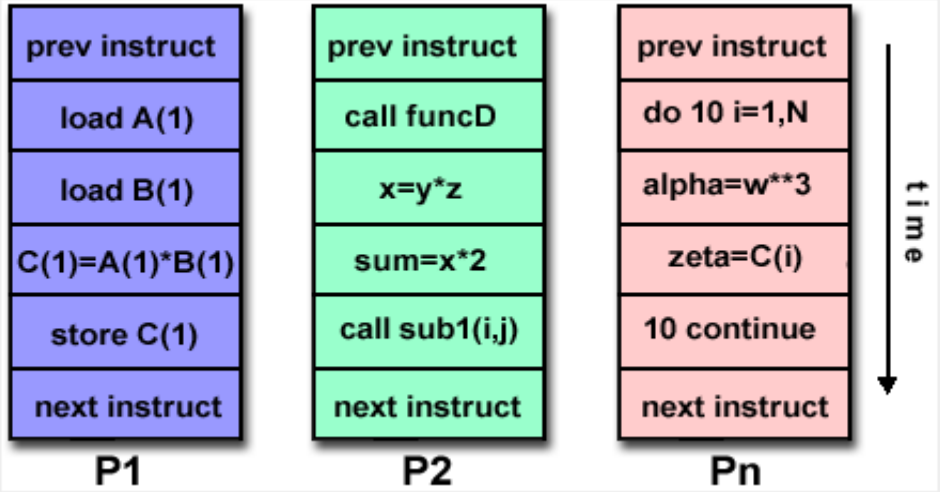


Funcionamento dos modelos [2]









Modelo de Programação Paralela

- **Threads (linhas de instruções)**
  - Programação utilizando memória compartilhada
  - Um único processo “pesado” pode ter múltiplos caminhos “leves” de execução.
- **Em Java**
  - Em Java, usamos a classe *Thread* do pacote java.lang para criarmos *linhas de execução* paralelas.
  - Existem 3 formas de se criar uma Thread:
    1) A classe *Thread* recebe como argumento um objeto com o código que desejamos rodar.
    2) Estendendo a classe Thread e instanciando um objeto desta nova classe.
    3) Implementando a interface **Runnable** e passando um objeto desta nova classe como argumento do construtor da classe Thread.
  - Nos 3 casos a tarefa a ser executada deve ser descrita pelo método **run()**
  - A criação da thread não indica sua execução, para tal use o método **start();**

Exemplo 1 – executem qualquer um dos dois códigos... Como foi a saída?

(2)

```
3  class Programa extends Thread {
4
5      private int id;
6      // colocar getter e setter pro atributo id
7
8      public Programa(int id) {
9          this.id = id;
10     }
11
12     public void run() {
13         for (int i = 0; i < 100; i++) {
14             System.out.println("Programa " + id + " valor: " + i);
15
16         }
17     }
18 }
19
20 public class TesteT {
21
22     public static void main(String[] args) {
23         // TODO Auto-generated method stub
24
25         for (int i = 1; i <= 3; i++) {
26
27             Programa p1 = new Programa(i);
28             p1.start();
29
30         }
31     }
32
33 }
```

(3)

```
3  class Programa implements Runnable {
4
5      private int id;
6      // colocar getter e setter pro atributo id
7
8      public Programa(int id) {
9          this.id = id;
10     }
11
12     public void run() {
13         for (int i = 0; i < 100; i++) {
14             System.out.println("Programa " + id + " valor: " + i);
15
16         }
17     }
18 }
19
20 public class TesteT {
21
22     public static void main(String[] args) {
23         // TODO Auto-generated method stub
24
25         for (int i = 1; i <= 3; i++) {
26
27             Programa p1 = new Programa(i);
28             Thread t1 = new Thread(p1);
29             t1.start();
30
31         }
32     }
33
34 }
```

(1)

Saída do Exemplo 1...

Ele imprimiu cada valor em sequência? Exatamente intercalado?

Não! Pois existem menos processadores para processar do que o número de *threads*, logo o **escalonador de threads** entra em ação.

O escalonador pega todas as threads e faz com o processador alterne a execução de cada uma delas por meio da **troca de contexto**.

A *troca de contexto* é justamente as operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.

Outras métodos presentes na classe *Threads*...

join(): Os demais processos esperam a thread terminar seu processamento.

sleep(): Pausa a execução fazendo com que a thread durma por um tempo específico. O tempo é em milisegundos.

yield(): Pausa a execução da thread passando para outra thread executar.

notify(): Herdado da classe Object. Este método retoma a execução de uma thread que estava pausada.

notifyAll(): This method is inherited from Object class. This method wakes up all threads that are waiting on this object's monitor to acquire lock.

wait(): This method is inherited from Object class. This method makes current thread to wait until another thread invokes the notify() or the notifyAll() for this object.

Para exemplos de códigos:

https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread

http://www.java2novice.com/java_thread_examples/

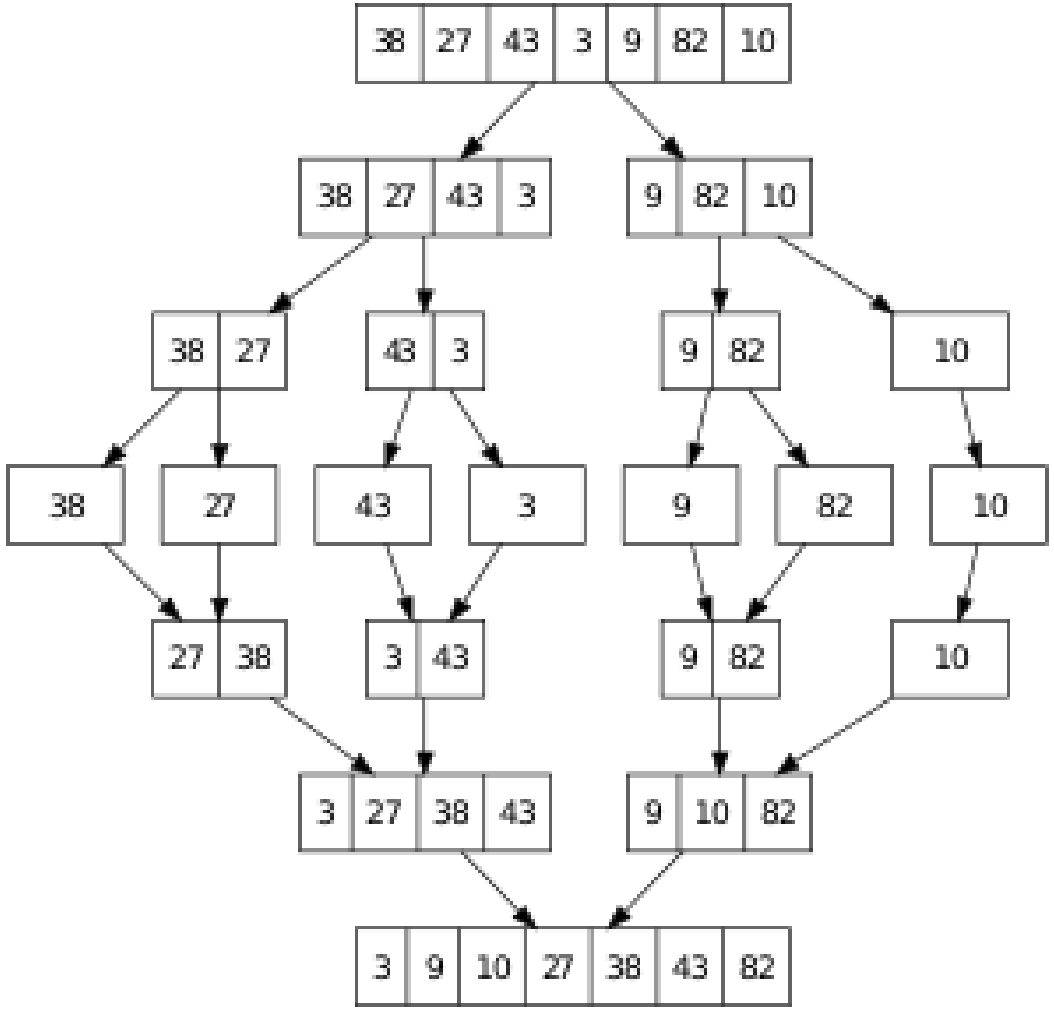
https://www.javatpoint.com/creating-thread

https://www.tutorialspoint.com/java/java_multithreading

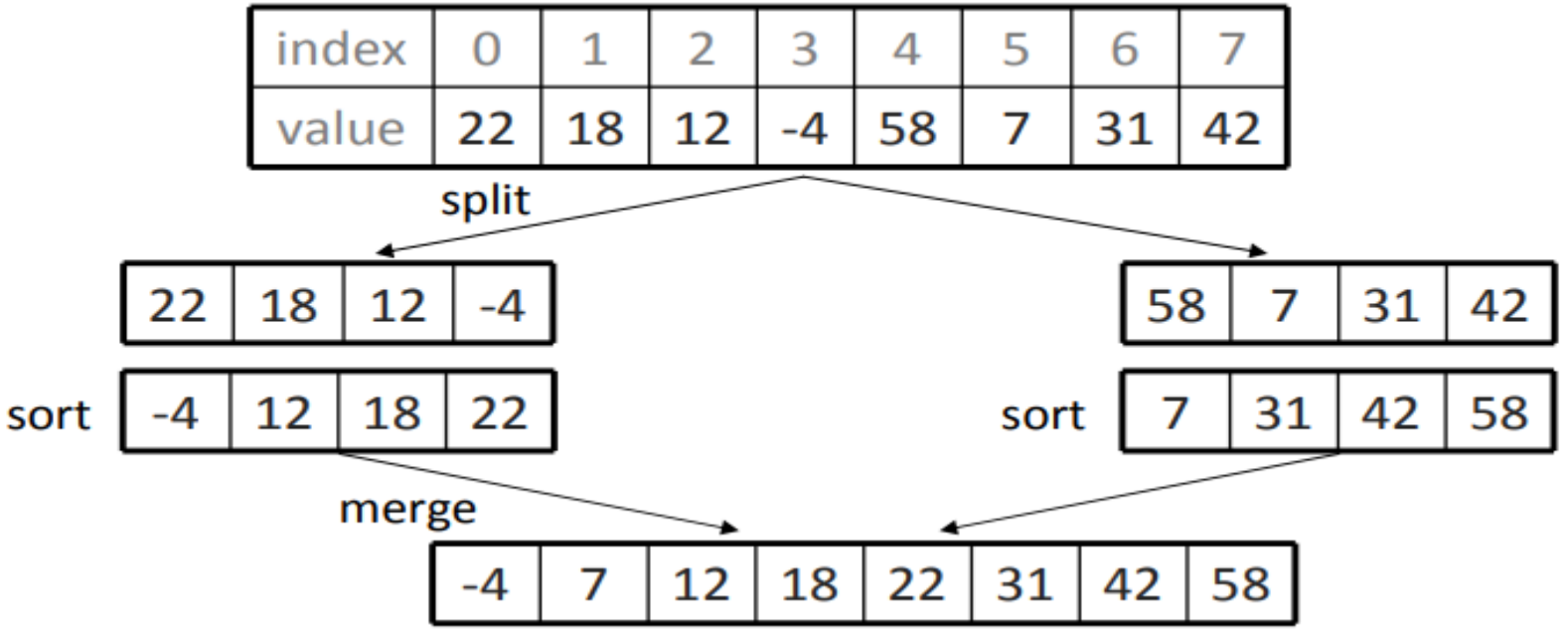
Exemplo de desempenho.

MergeSort.java

"Baseado em divisão e conquista".



MergeSort paralelo com 2 threads.



Idéia:

- **Divide o array no meio.**
- **Recursivamente ordena (sort) cada metade em sua própria thread.**
- **Une as duas metades já ordenado.**

Hands-on – MergeSort Paralelo

Acesse o site :

https://courses.cs.washington.edu/courses/cse373/13wi/lectures/03-13/

1. Baixe os arquivos **MergeSort.java** e **Sorter.java** e execute-os em sua máquina e veja o desempenho deste algoritmo alterando a linha "**int cores = 8**" para o número de cores disponível.

DICA: adicione o código do **Sorter.java** como class no arquivo **MergeSort.java**

2. Abra o arquivo "26-parallel-algorithms.pdf" e faça as modificações propostas nos slides 20, 21 e 22, para executar 2 novas threads a cada divisão do vetor. Comparar o desempenho com o método anterior.

Resultados

Houve melhorias no desempenho (speedup)?

As threads do Java são pesadas para serem utilizadas sem um controle correto.

Além disto, temos que a **Lei de Amdahl's** que diz que o speedup está limitado a porção paralelizável do programa.

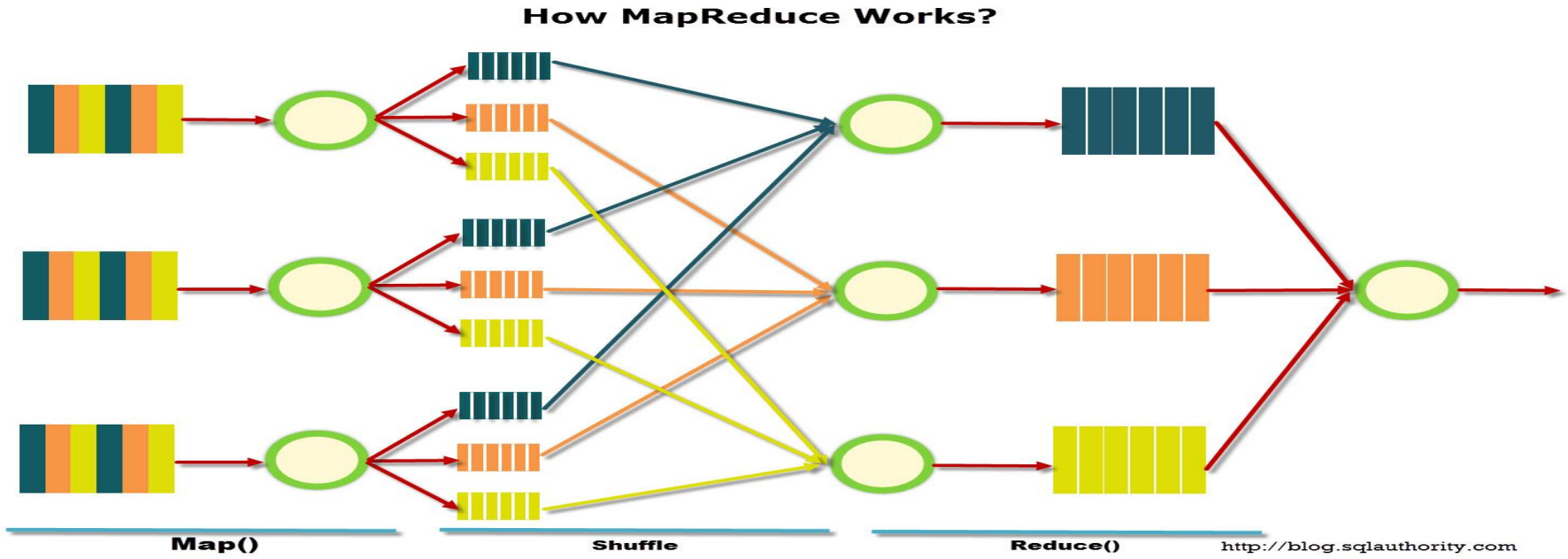
Ex. Se 33% do programa deve ser sequencial, então independente do número de processadores só terá speedup de 3x.

Desta forma, é necessário ter o melhor **controle** do que e quando se deve paralelizar.

Arquitetura MapReduce

Map/Reduce

- É uma estratégia para paralelização de algoritmos baseada na arquitetura cliente/servidor.
- É dividido em duas fases:
  - **Map:** um nó **master** recebe o problema, divide em vários subproblemas e distribui os subproblemas para os **workers.**
  - **Reduce:** o **master** coleta as subsoluções dos **workers** e as combina de forma a produzir a solução do problema inicial.



Big Data e Map/Reduce – Analogia com Restaurante [1]

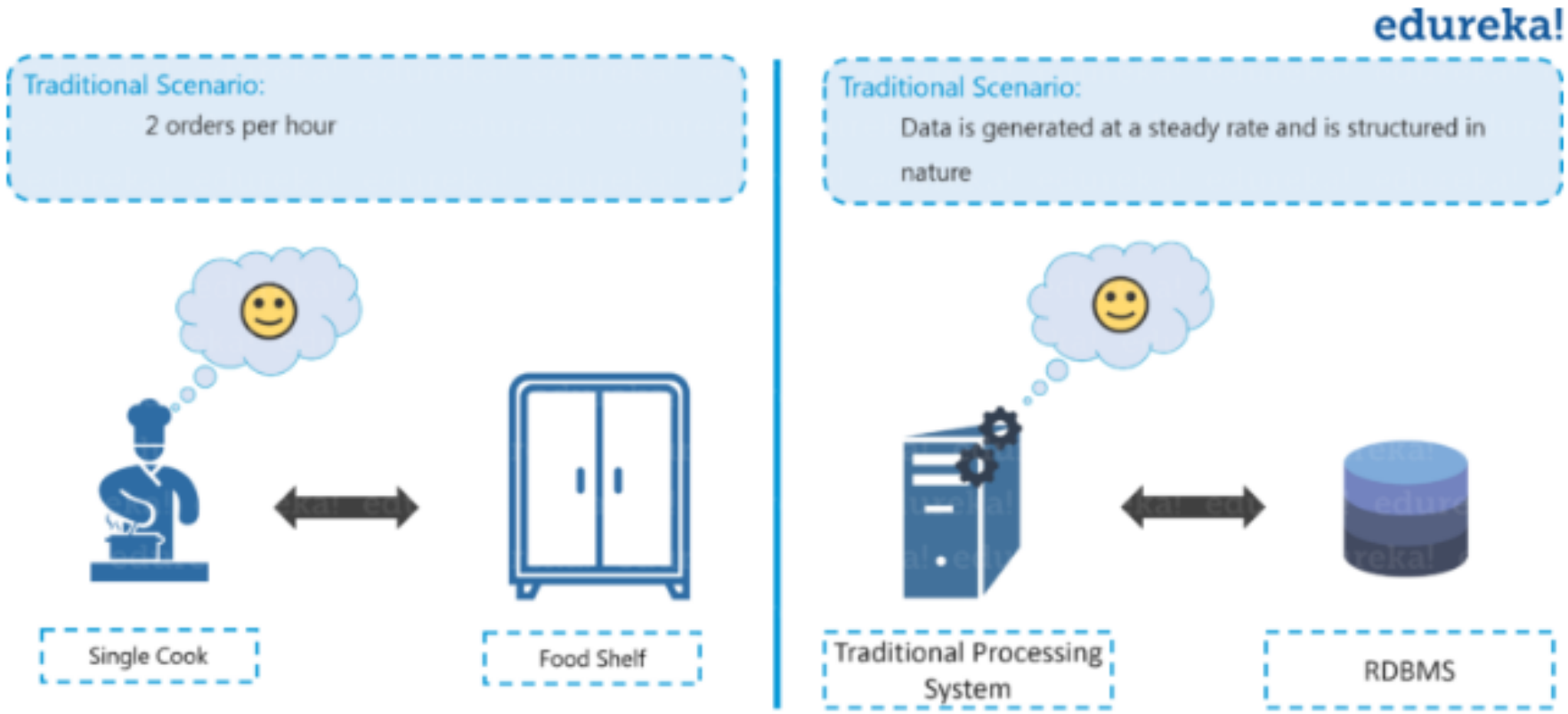


Fig: Hadoop Tutorial – Traditional Restaurant Scenario

Fonte: https://www.edureka.co/blog/hadoop-tutorial/?utm_source=blog&utm_medium=left-menu&utm_campaign=hadoop-tutorial

Big Data e Map/Reduce – Analogia com Restaurante [2]

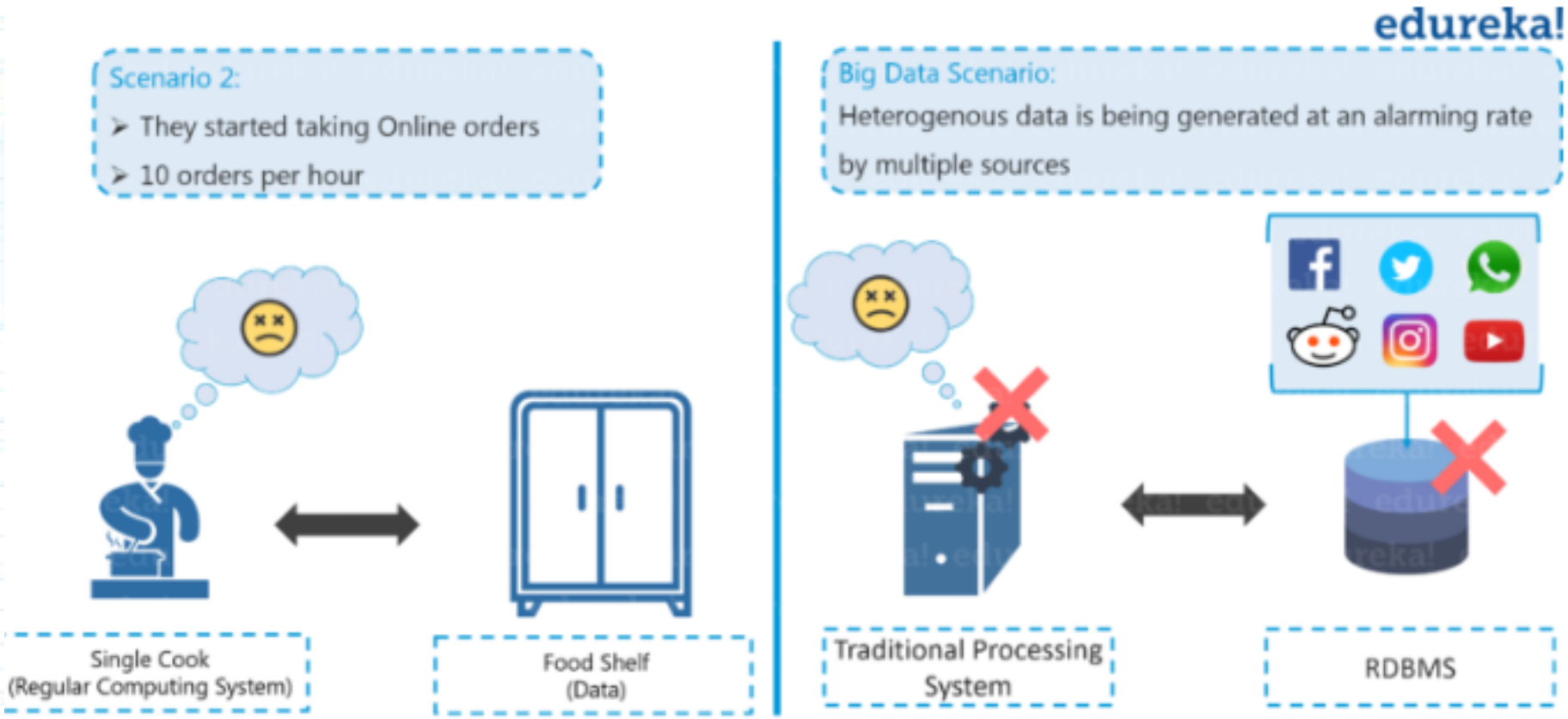


Fig: Hadoop Tutorial – Traditional Scenario

Fonte: https://www.edureka.co/blog/hadoop-tutorial/?utm_source=blog&utm_medium=left-menu&utm_campaign=hadoop-tutorial

Big Data e Map/Reduce – Analogia com Restaurante [3]

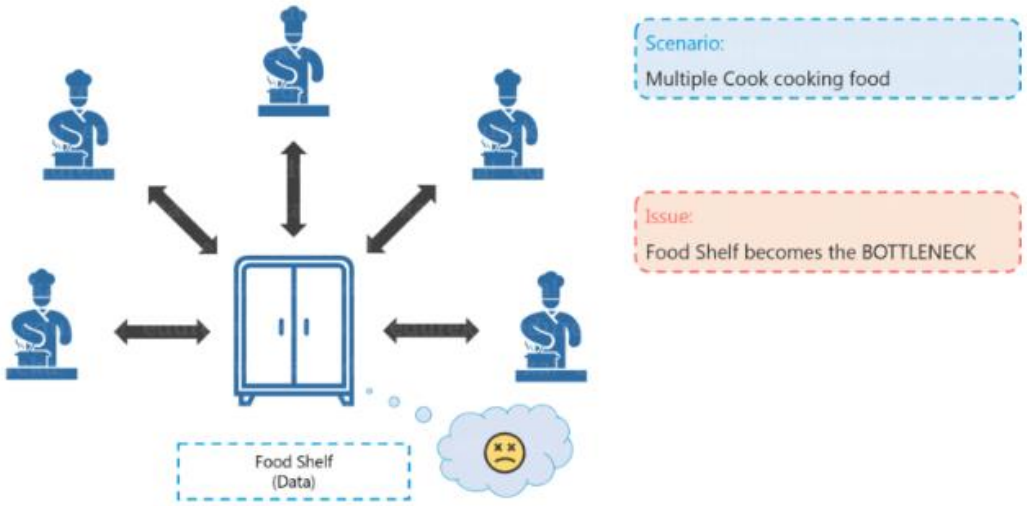


Fig: Hadoop Tutorial – Distributed Processing Scenario

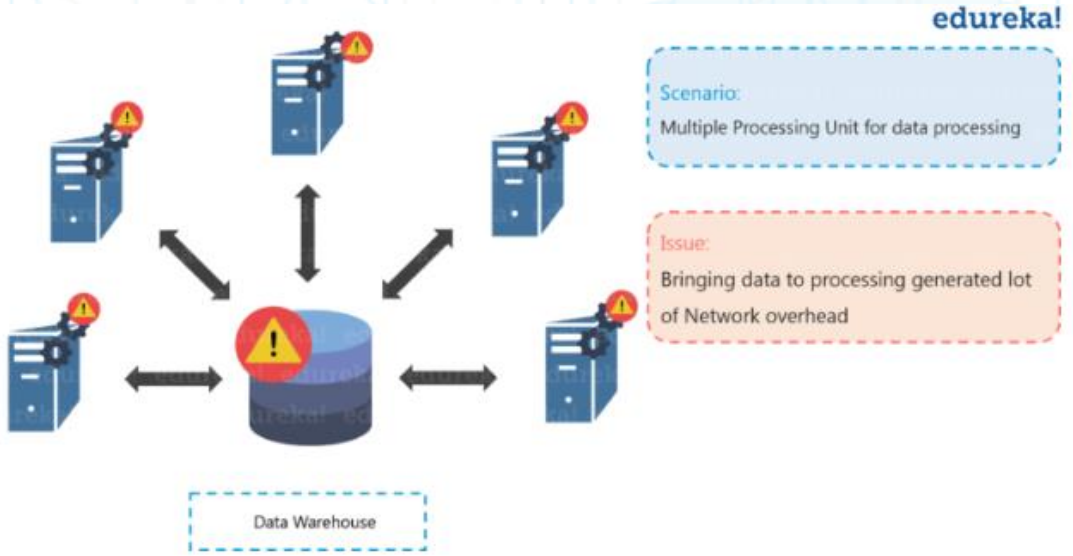


Fig: Hadoop Tutorial – Distributed Processing Scenario Failure

Fonte: https://www.edureka.co/blog/hadoop-tutorial/?utm_source=blog&utm_medium=left-menu&utm_campaign=hadoop-tutorial

Big Data e Map/Reduce – Analogia com Restaurante [4]

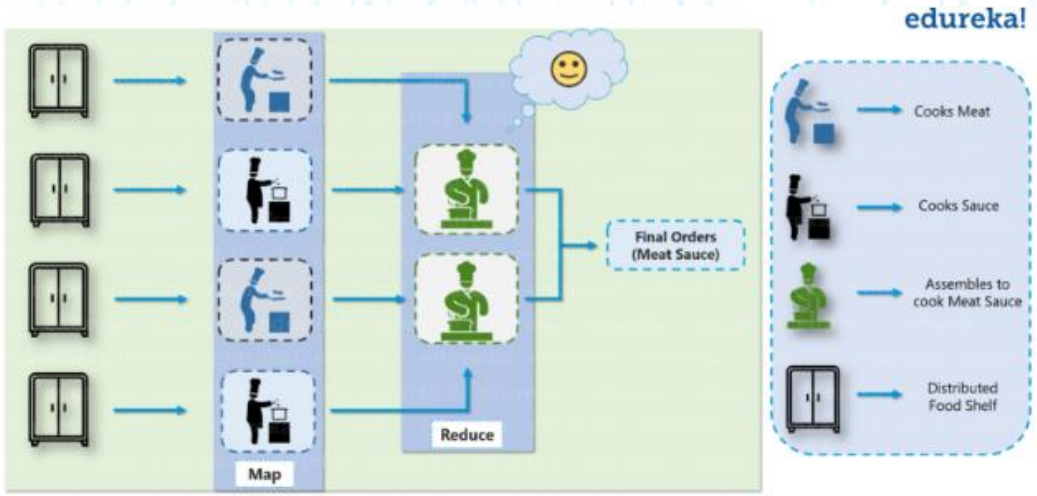


Fig: Hadoop Tutorial – Solution to Restaurant Problem

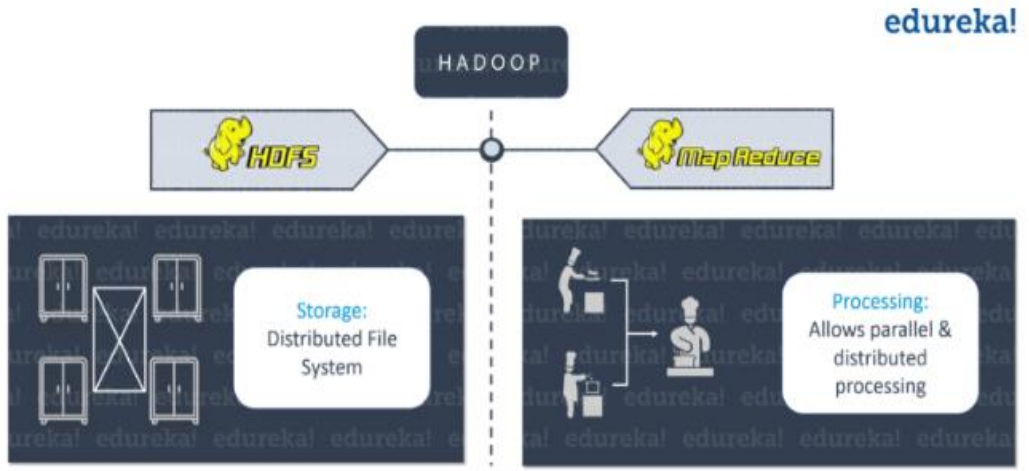
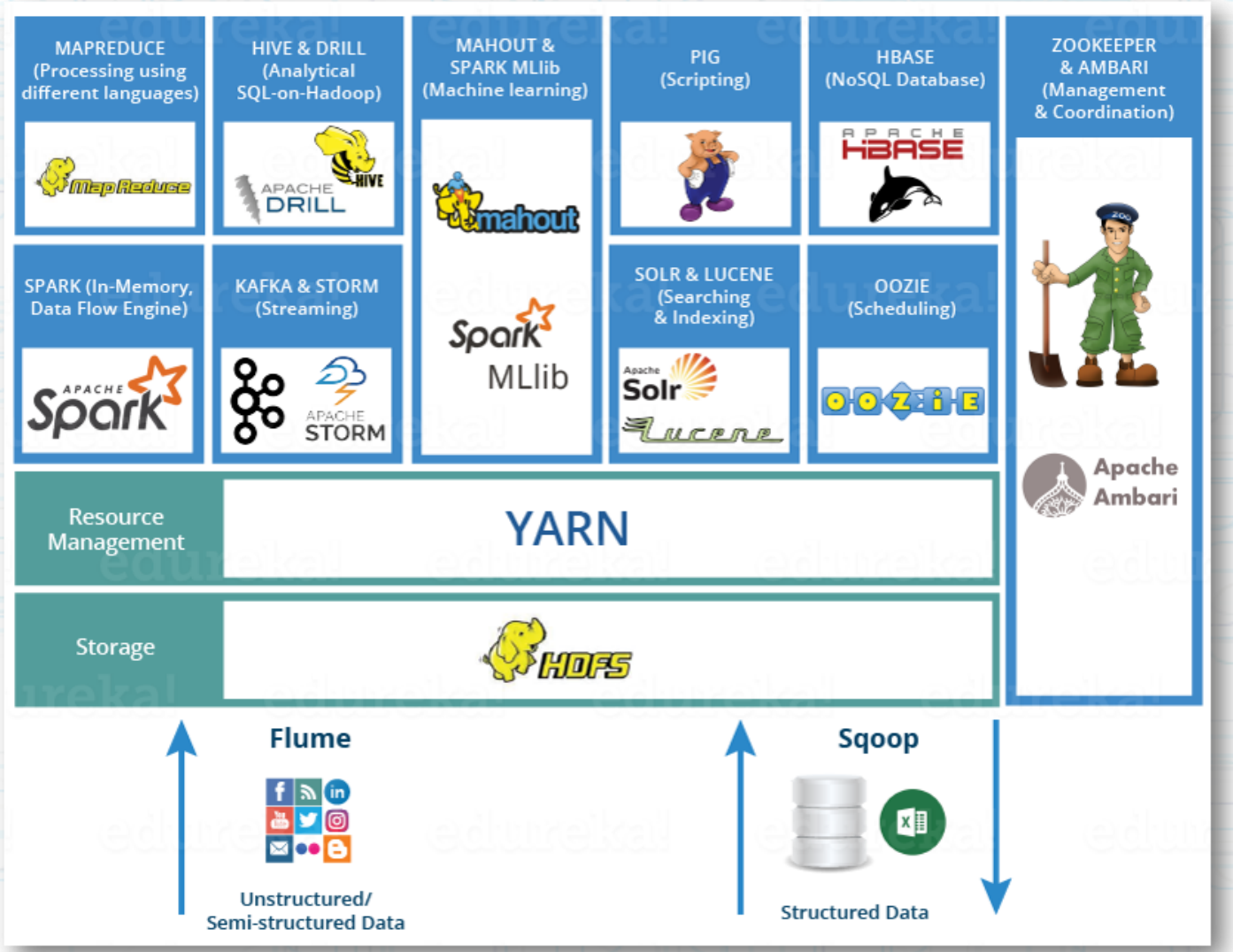


Fig: Hadoop Tutorial – Hadoop in Restaurant Analogy

Fonte: https://www.edureka.co/blog/hadoop-tutorial/?utm_source=blog&utm_medium=left-menu&utm_campaign=hadoop-tutorial

Ecossistema Hadoop para Big Data

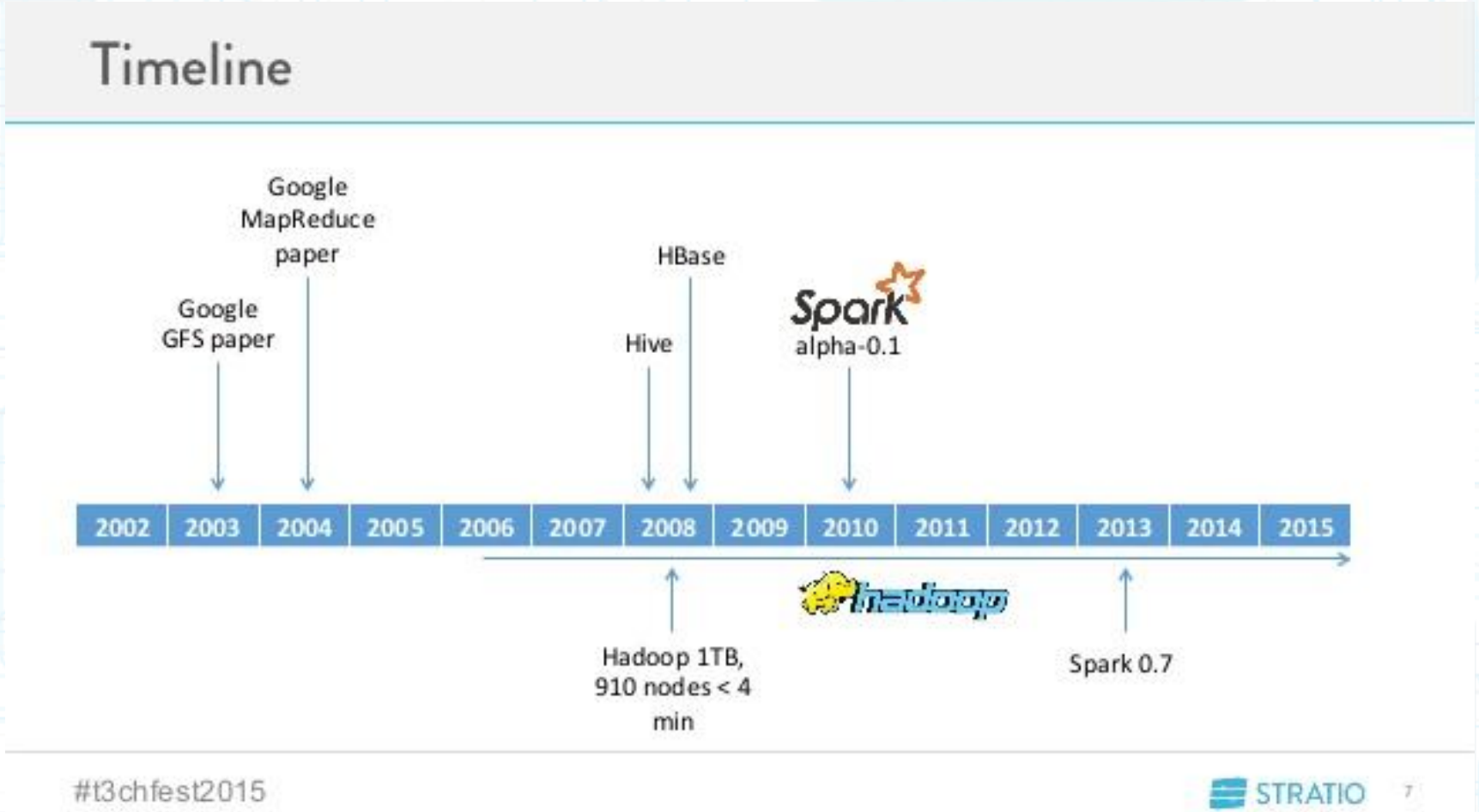


Arquitetura e Princípios

- História
  - Apache MapReduce (Hadoop)
    - Só contemplava duas operações
    - Performance ruim para processamento
  - A cada operação havia necessidade de utilizar o disco
- Algumas abstrações:
  - Hive/Pig
  - Ainda apresentava problemas de performance com acesso ao disco
- Surgimento do Apache Spark

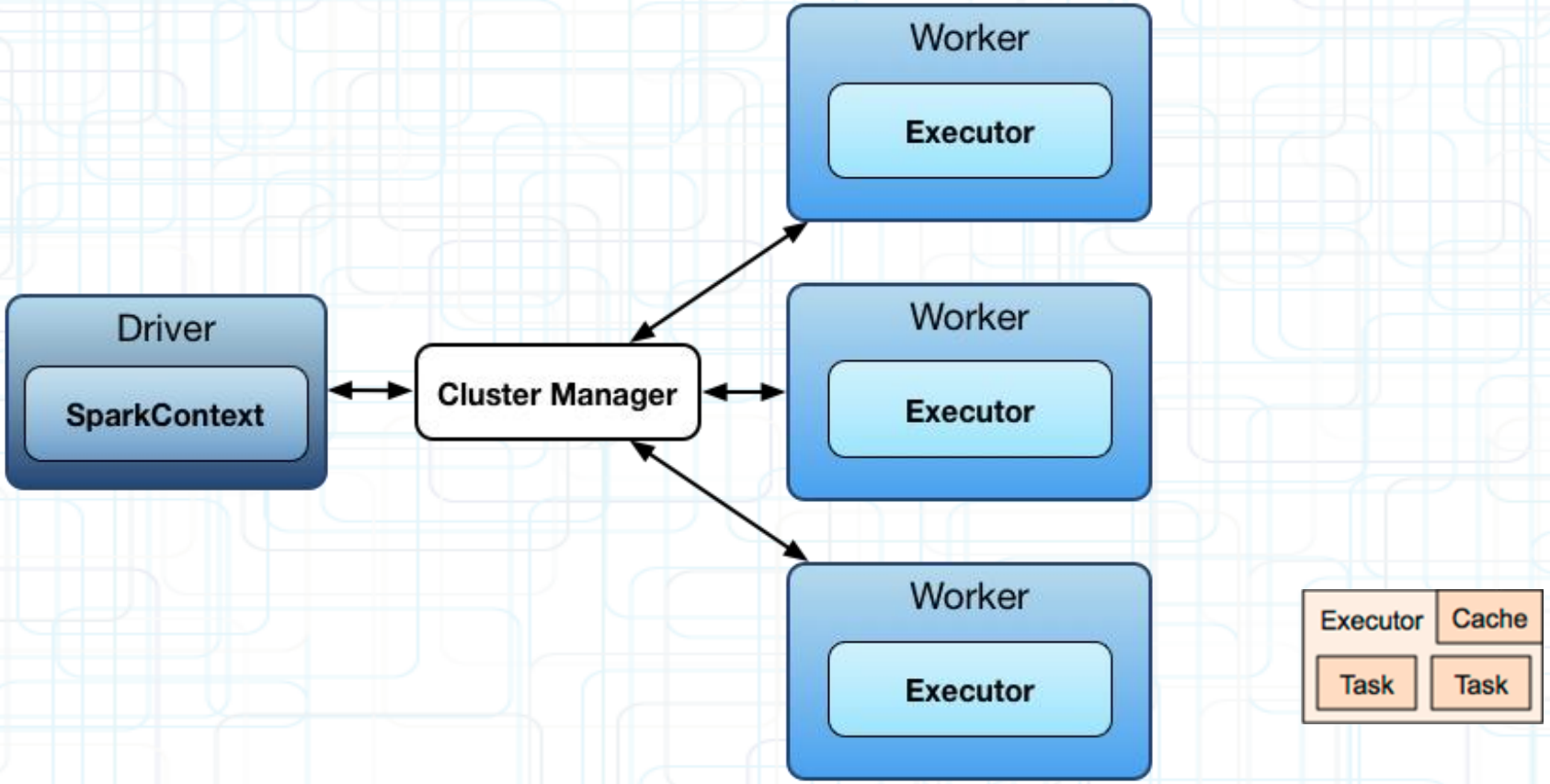
Arquitetura e Princípios

- História



Arquitetura e Princípios

- Arquitetura Cliente/Servidor (Escravo/Mestre)

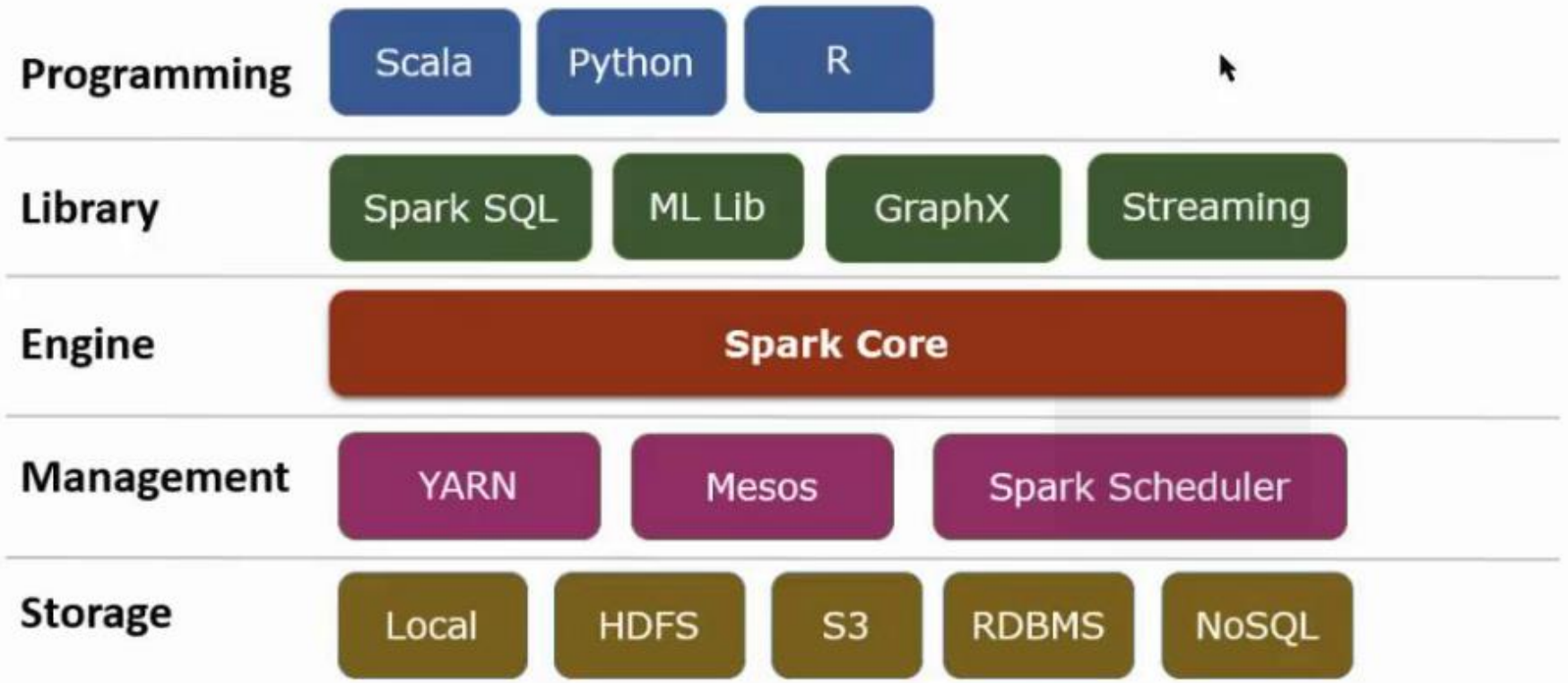


Arquitetura e Princípios

- Princípios
  - Simples para usar
  - Performático
  - Escalável
  - Tolerante a falhas
  - De propósito geral

Arquitetura e Princípios

- Ecossistema



Executando o Spark

- *Spark Shell*
  - **cd $SPARK_HOME (diretório aonde está o spark)**
  - **./bin/spark-shell**

```
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
15/02/28 10:26:21 WARN Utils: Your hostname, ubuntu resolves to a loopback address: 127.0.1.1; using 192.168.127.150 instead (on interface eth0)
15/02/28 10:26:22 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
15/02/28 10:26:30 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context available as sc.

scala>
```

- *Scala IDE*
  - http://scala-ide.org/download/sdk.html

Arquitetura e Princípios

- **O que é Scala??**
  - É uma **linguagem de programação** de propósito geral, diga-se multiparadigma, projetada para expressar padrões de programação comuns de uma forma concisa, elegante e *type-safe*
  - ***Tipos de Dados:***

Variable Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Declaração de variáveis:

```
var x = 10
x = 20

val y: Int = 10;
val y = 10
```

Scala - tuplas

- Pode-se criar tuplas sem a necessidade de declarar tipos de variáveis em tuplas.

```
scala> val tup1 = (0, "zero")
scala> tup1.getClass
res1: Class[_ <: (Int, String)] = class scala.Tuple2

scala> val tup2 = (1, "um", 1.0)
scala> tup2.getClass
res2: Class[_ <: (Int, String, Double)] = class scala.Tuple3

scala> val tup2 = (1, "um", 1.0)
scala> tup2.getClass
res2: Class[_ <: (Int, String, Double)] = class scala.Tuple3
```

- Pode se acessar os valores de cada tupla usando o sublinhado (“_”)

```
scala> println(tup1._1)
0

scala> println(tup1._2)
zero

scala> println(tup2._2)
um

scala> println(tup2._3)
1.0
```

Scala - funções

- Para definir uma **função** é preciso dizer o tipo dos parâmetros e o tipo de retorno.
- Segue abaixo as funções **add** e **sum** que realizam a soma entre dois inteiros.

```
def add(firstInput: Int, secondInput: Int): Int = {
  val sum = firstInput + secondInput
  return sum
}
```

```
scala> def sum (x: Int, y: Int) : Int = { return x + y }
sum: (x: Int, y: Int)Int

scala> sum (1,2)
res0: Int = 3
```

- Funções não precisam ter nome, sendo dita anônima ou função **lambda**.

```
scala> (x: Int, y: Int) => x + y
res1: (Int, Int) => Int = <function2<

scala> res1(1,2)
res2: Int = 3
```

Scala - funções

- Para que serve uma função **lambda** ou como invocar uma**?**
- Muitas funções do Spark tem como parâmetro outras funções, portanto o uso do lambda acaba otimizando o código pois não há a necessidade de declarar formalmente a função.
- Segue abaixo a função **max,** que recebe dois inteiros e retorna o maior valor entre os dois.

```
scala> val max: (Int, Int) => Int = (m: Int, n: Int) => if(m > n) m else n
max: (Int, Int) => Int = <function2<

scala> max(3,4)
res8: Int = 4
```

- A função **max** pode ser simplificada da seguinte forma:

```
scala> def max (x: Int, y: Int) : Int = { if (x > y) x else y }
```

Scala - funções

- Funções com funções internas

```
def encodeWithSeed(num: Int, seed: Int): Long = {

  def encode(x: Int, func: (Int) => Int): Long = {
    val y = x + 1000
    func(y)
   }

   val result = encode(num, (n: Int) => (n * seed))
   result
}
```

- Funções com verificações

```
def f(x: Int, y: Int, operator: String): Double = {
  operator match {
    case "+" => x + y
    case "-" => x - y
    case "*" => x * y
    case "/" => x / y.toDouble
  }
}

val sum = f(10,20, "+")
val product = f(10, 20, "*")
```

Scala - classes

- Por possuir integração com a OO, o Scala permite a criação de **classes** com seus devidos **atributos** e **métodos**

```
class Car(mk: String, ml: String, cr: String) {
  val make = mk
  val model = ml
  var color = cr

      def repaint(newColor: String) = {
        color = newColor
      }
    }
```

Scala - coleções

- O Scala também trabalha com o pacote **Collections** do Java.
  - **Array / List / Vector / Set / Map**

```
val arr = Array(10, 20, 30, 40)
arr(0) = 50
val first = arr(0)
```

```
val xs = List(10,20,30,40)
val ys = (1 to 100).toList
val zs = someArray.toList
```

```
val v1 = Vector(0, 10, 20, 30, 40)
val v2 = v1 :+ 50
val v3 = v2 :+ 60
val v4 = v3(4)
val v5 = v3(5)
```

```
val fruits = Set("apple", "orange", "pear", "banana")
```

```
val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
val indiaCapital = capitals("India")
```

Scala - coleções

- Transformações comumente utilizados com o **Collections**
  - **Map / FlatMap / Filter / Foreach / Reduce**

```
val xs = List(1, 2, 3, 4)
val ys = xs map {x => x * 10.0}
val ys = xs map {_ * 10.0}
val ys = xs.map((x: Int) => x * 10.0)
```

```
val line = "Scala is fun"
val SingleSpace = " "
val words = line.split(SingleSpace)
val arrayOfChars = words flatMap {_.toList}
```

```
val xs = (1 to 100).toList
val even = xs filter {_ %2 == 0}
```

```
val words = "Scala is fun".split(" ")
words.foreach(println)
```

```
Val words = "Scala is fun" split(" ")
val longestWord = words reduce {(w1, w2) => if(w1.length > w2.length) w1 else w2}
```

Fonte de Dados.

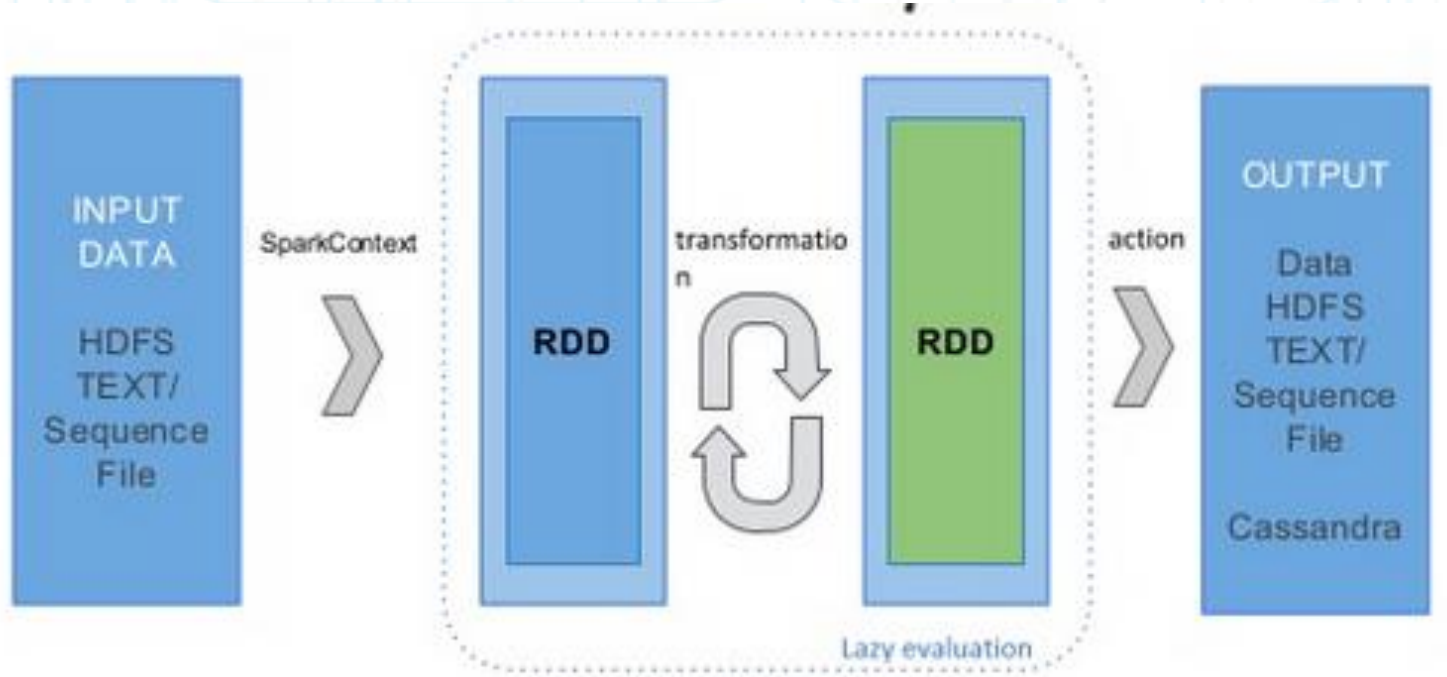
- O Spark integra com quase qualquer tipo de fonte de dados:
  - HDFS
  - Hbase
  - Cassandra
  - Amazon S3
  - Stream (ex. sockets)
  - Local

Arquitetura e Princípios

- **API básica de programação – Abstração de dados**
  - ***RDD (Resilient Distributed Datasets)***
    - Imutável
    - Particionado
    - Tolerante a falhas
    - Interface – Possui Polimorfismo de OO
      - HadoopRDD
      - ParallelCollectionRDD
      - JdbcRDD
    - Fortemente tipado
  - Em memória

Arquitetura e Princípios

- **API básica de programação**
  - ***RDD (Resilient Distributed Datasets)***



Arquitetura e Princípios

- **API básica de programação – Abstração dos dados**
  - ***RDD (Resilient Distributed Datasets)***
  - Existem duas formas de criar um RDD:
    1. Paralelizando uma coleção já existente dentro do programa

```
val config = new SparkConf().setMaster("local").setAppName(“Init Parallelize")
val sc = new SparkContext(config)

val xs = (1 to 10000).toList
val rddParallelize = sc.parallelize(xs)
```

    2. Referenciando dados em um sistema externo ao programa (HDFS, Hbase, ou qualquer fonte de dado com um Hadoop InputFormat)

```
val config = new SparkConf().setMaster("local").setAppName(“Init Text File")
val sc = new SparkContext(config)

val rddTextFile = sc.textFile("/path/to/file-or-directory")
```

Arquitetura e Princípios

- **API básica de programação – Abstração de dados**
  - ***RDD (Resilient Distributed Datasets)***
    - Observações:
      - Se for realizar a leitura de arquivos em uma máquina local, todos os *workers* devem ter o arquivo em suas máquinas no mesmo caminho.
      - Todos os métodos para leitura de arquivos suportam ler diretórios, arquivos comprimidos e expressões regulares.
        - textFile(“/path/to/dir”)
        - textFile(“path/to/dir/*.txt”)
        - textFile(“path/to/dir/*.gz”)
      - Além de arquivos de texto, o Spark suporta outros formatos:
        - O método **wholeTextFiles(“path/to/dir”)** lê um diretório contendo vários arquivos de texto e retorna um vetor com o par [nome-do-arquivo, conteúdo-do-arquivo]

Arquitetura e Princípios

- **API básica de programação – Abstração de dados**
  - ***RDD (Resilient Distributed Datasets)***
    - Salvando arquivos:
      - Para salvar um RDD em arquivo pode-se usar o método **saveAsObjectFile**, que salva o arquivo usando serialização Java, ou **saveAsTextFile** que salva o arquivo em texto puro.

```
scala> val file = sc.textFile("README.md")
file: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[10] at textFile at :27

scala> file.count
res1: Long = 95

scala> file.saveAsObjectFile("/tmp/readme_object")
scala> file.saveAsTextFile("/tmp/readme_text")
```

Spark Core

- Algumas das principais funcionalidades do Spark são suas **transformações** que são métodos para realizar operações como filtros e mapeamentos.

- Outra importante funcionalidade são as **ações** que são operações para a realização de contagens e somatórios sobre os dados transformados

Spark – principais Transformações

Transformação	Descrição	Exemplo	Resultado
map(*func*)	retorna um RDD depois de processar cada elemento com a função *func*	sc.parallelize(Array(1, 2, 3, 4, 5)).map (x => x + 10)	{ 11, 12, 13, 14, 15 }
filter(*func*)	Retorna um RDD depois de selecionar os enementos onde *func* é verdadeira	sc.parallelize(Array(1, 2, 3, 4, 5)).filter(x => x == 3)	3
flatMap(*func*)	Similar ao map, mas cada item pode ser mapeado para 0 ou mais items (*func* deve retornar uma sequência, ao invés de um item).	sc.parallelize(List("this is line 1", "this is the second line"), 2).flatMap(_.split(" ")).foreach {println _}	{this, is, the, second, line, this, is, line, 1}
mapPartitions(*func*)	Similar ao map, mas executa *func* separadamente em cada partição (bloco). A função *func* deve ser do tipo *Iterator<T> => Iterator<U>* quando invocada sobre um RDD do tipo T.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8)), 3).mapPartitions (x => x.map((t: (Int, Int)) => t._1 + t._2)).foreach {println }	{3, 3, 3, 7, 11, 15, 15}
mapPartitionsWithIndex(*func*)	Similar a mapPartitions, mas provê *func* com um valor inteiro representando o índice da partição. A função *func* deve ser do tipo *Iterator<T> => Iterator<U>* quando invocada sobre um RDD do tipo T.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8)), 3).mapPartitionsWithIndex ((index: Int , x) => x.map((t: (Int, Int)) => (index, t._1 + t._2))).foreach {println }	(0,3) (0,3) (1,3) (1,7) (2,11) (2,15) (2,15)

Spark – principais Transformações

sample(*withReplacement*, *fraction*,*seed*)	Amostragem de uma fração de um dado, com ou sem substituição, usando um número aleatório.	sc.parallelize(Array(1, 2, 3, 4, 5)).sample(false, 0.4).foreach { println _ }	{3,4,2}
union(*otherDataset*)	Retorna uma coleção que contém a união de todos os elementos da fonte e do argumento.	sc.parallelize(Array(1, 2, 3, 4, 5)).union(sc.parallelize(Array(6,7,8,9,10))).foreach(println _)	{1,2,3,4,5,6,7,8,9,10}
intersection(*otherDataset*)	Retorna um RDD que contém uma interseção dos elementos da fonte e do argumento.	sc.parallelize(Array(1, 2, 3, 4, 5)).intersection(sc.parallelize(Array(4,5,6,7,8))).foreach(println _)	{5,4}
distinct([*numTasks*]))	Retorna uma coleção que contém os elementos distintos da fonte.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8))).distinct().foreach { println _ }	{(1,2), (5,6), (7,8), (3,4)}
groupByKey([*numTasks*])	Quando invocado sobre uma coleção de pares (k, v), retorna um par (k, Iterable<v>).	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8))).groupByKey().foreach { println _ }	(1,CompactBuffer(2, 2, 2)) (3,CompactBuffer(4)) (7,CompactBuffer(8, 8)) (5,CompactBuffer(6))
reduceByKey(*func*, [*numTasks*])	Quando invocado sobre uma coleção de pares (k, v), retorna uma coleção de pares (k, v) onde os valores de cada chave são agregados usando pela função *func*, que deve ser do tipo (v, v).	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8))).reduceByKey(_ + _).foreach { println _ }	(1,6) (3,4) (7,16) (5,6)

Spark – principais Transformações

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	Quando invocado sobre uma coleção de pares (k, v) a função *seqOp* é aplicada nos valores *v* de cada chave. Depois, função *combOp* é invocada para agregar os valores resultantes de *seqOp*. O *zeroValue* representa o valor zero (ponto inicial) para calcular o valor de retorno da função *combOp*. Esta função permite que o tipo do valor de retorno seja diferente do tipo valor de entrada. O tipo do valor de retorno é definido por *zeroValue*.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8)), 3).aggregateByKey(List[Int]())((x,y) => x ++ List(y) , _ ++ _).foreach {println}	(1,List(2, 2, 2)) (3,List(4)) (7,List(8, 8)) (5,List(6))
sortByKey([*ascending*], [*numTasks*])	Quando invocado sobre uma coleção de pares (k, v), onde k implementa Ordered, retorna uma coleção de pares (k, v) ordenados pela chaves.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8)), 1).sortByKey().foreach {println _}	(1,2) (1,2) (1,2) (3,4) (5,6) (7,8) (7,8)
join(*otherDataset*, [*numTasks*])	Quando invocado sobre uma coleção de pares do tipo (k, v) e (k, w), retorna uma coleção de pares (k, (v, w)).	sc.parallelize(Array ((1, "um"), (2, "dois"), (3, "tres"))) .join(sc.parallelize(Array ((1, "one"), (2, "two"), (3, "three")))).foreach { println _ }	(1,(um,one)) (3,(tres,three)) (2,(dois,two))
cogroup(*otherDataset*, [*numTasks*])	Mesmo que *groupWith* Quando invocado sobre uma coleção de pares (k, v) e (k, w), retorna uma coleção do tipo (k, (Iterable<v>,Iterable<w>)).	sc.parallelize(Array ((1, "um"), (2, "dois"), (3, "tres"))) .cogroup(sc.parallelize(Array ((1, "one"), (2, "two"), (3, "three")))).foreach { println _ }	(2, (CompactBuffer(dois) ,CompactBuffer(two))) (1, (CompactBuffer(um), CompactBuffer(one))) (3, (CompactBuffer(tres), CompactBuffer(three)))

Spark – principais Ações

Ações	Descrição	Exemplo	Resultado
reduce(*func*)	Agrega os elementos de uma coleção usando a função *func* (que precisa de dois argumentos, retornando um). A função deve ser acumulativa ou associativa, para ser computada corretamente em paralelo.	sc.parallelize(Array(1, 2, 3, 4, 5)).reduce((a, b) => a+b)	15
collect()	Retorna todos os elementos de uma coleção como um vetor dentro do programa *driver*.	sc.parallelize(Array(1, 2, 3, 4, 5)).collect	Array(1, 2, 3, 4, 5)
count()	Retorna o número de elementos dentro da coleção de dados.	sc.parallelize(Array(1, 2, 3, 4, 5)).count	5
first()	Retorna o primeiro elemento da coleção de dados.	sc.parallelize(Array(1, 2, 3, 4, 5)).first	1
take(*n*)	Return an array with the first *n* elements of the dataset.	sc.parallelize(Array(1, 2, 3, 4, 5)).take(2)	Array(1, 2)
takeSample(*withReplacement*, *num*, [*seed*])	Retorna um vetor com uma amostra aleatória de *num* selecionando opcionalmente uma semente para os números aleatóreos. O parâmetro *withReplacemente* diz respeito a repetir os elementos na amostra ou não.	sc.parallelize(Array(1, 2)).takeSample(true,4)	Array(1, 1, 2, 1)
takeOrdered(*n*, *[ordering]*)	Retorna os primeiros *n* elementos de um RDD usando ou a ordem natural ou um comparador customizado.	sc.parallelize(Array(1, 2, 3, 4, 5)).takeOrdered(3)	Array(1, 2, 3)

Spark – principais Ações

saveAsTextFile(*path*)	Escreve todos os elementos da coleção de dados em um arquivo texto no sistema de arquivos local ou em qualquer sistema de arquivos suportado pelo Hadoop.	sc.textFile("README.md").saveAsTextFile("README.md2")	README.md2/_SUCCESS README.md2/part-00000 README.md2/part-00001
saveAsSequenceFile(*path*) (Java and Scala)	Escreve todos os elementos da coleção de dados como um *SequenceFile* do Hadoop no sistem de arquivos local, ou em qualquer sistema de arquivos suportado pelo Hadoop.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8))).saveAsSequenceFile("SEQ")	SEQ/_SUCCESS SEQ/part-00000 SEQ/part-00001
saveAsObjectFile(*path*) (Java and Scala)	Escreve todos os elementos da coleção de dados em um formato simples usando serialização Java, que pode ser deserializada posteriormente usando *SparkContext.objectFile()*.	sc.textFile("README.md").saveAsObjectFile("README.obj")	README.obj/_SUCCESS README.obj/part-00000 README.obj/part-00001
countByKey()	Somente disponível em RDDs do tipo (k, v) hashmap de pares (K, Int) com o número de ocorrência de cada chave.	sc.parallelize(Array((1,2), (1,2), (1,2), (3,4), (5,6), (7,8), (7,8))).countByKey	Map(1 -> 3, 3 -> 1, 7 -> 2, 5 -> 1)
foreach(*func*)	Executa uma função *func* em cada elemento da coleção de dados. Geralmente utilizada para atualizar contadores ou interagir com sistemas de arquivos externos. **Observação**: modificar *accumulators* fora de um *foreach* pode um comportamento indefinido.	sc.parallelize(Array(1, 2, 3, 4, 5)).foreach { println _ }	{1,2,3,4,5}

Spark – teste: Linhas de Ônibus em Goiânia

Crie um arquivo .txt com o seguinte conteúdo referente as posições de ônibus em Goiânia:

002,PARQUE.ATHENEU,T.ISIDORIA

006,T.VEIGA.JARDIM,CENTRO

008,T.VEIGA.JARDIM,RODOVIARIA

003,T.MARANATA,RODOVIARIA

013,RODOVIARIA,CENTRO

009,T.ISIDORIA,AV.CASTELO.BRANCO

021,PARQUE.ATHENEU,T.BIBLIA

017,T.CRUZEIRO,RODOVIARIA

014,PARQUE.ATHENEU,CAMPINAS

Primeiramente vamos carregar o arquivo .txt no Spark.

val linhasOnibusRDD = sc.textFile(“path/to/arquivo.txt”)

Depois, vamos mostrar a quantidade de linhas de ônibus existem neste arquivo.

val qntLinhas = linhasOnibusRDD.count

Por fim, vamos filtrar para mostrar apenas as linhas que passam na RODOVIARIA.

val linhaRodoviaria = linhasOnibusRDD.filter(s => s.contains(“RODOVIARIA”))

Hands-on: Wordcount nas obras de Shakespeare

Entre no site http://www.gutenberg.org/cache/epub/100/pg100.txt e salve o documento em formato .txt.

Em seguida, realize a contagem de palavras (*wordcount*) no documento baixado, apresentando a seguinte informação:

- Quais são as 20 palavras mais usadas por Shakespeare e quantas vezes elas ocorreram?

Hands-on: Atividade Avaliativa - entregar

Ainda utilizando a base de dados das obras de Shakespeare, realize as seguintes operações:

1. Retire todos os caracteres especiais e todas as ***stopwords*** (palavras que podem ser consideradas irrelevantes para o conjunto de resultados. Ex. veja link https://www.ranks.nl/stopwords) usando o próprio Spark e apresente novamente as 20 palavras mais utilizadas por Shakespeare após a filtragem.
2. Salve em um arquivo de texto as palavras mais utilizadas por Shakespeare em ordem alfabética.

Referências Bibliográficas

Guller, Mohammed. **Big Data Analytics with Spark**. Apress, 2015.

Duvvuri, Srinivas, and Bikramaditya Singhal. **Spark for Data Science**. Packt Publishing Ltd, 2016.

https://computing.llnl.gov/tutorials/parallel_comp/

https://courses.cs.washington.edu/courses/cse373/13wi/lectures/03-13/

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-standalone.html

https://spark.apache.org/

https://github.com/deanwampler/spark-scala-tutorial

https://www.edureka.co/blog/hadoop-tutorial/

http://www.inf.ufpr.br/erlfilho/tutorials/spark/