Spark

Inteligência na Web e Big Data

Fabricio Olivetti de França e Thiago Ferreira Covões
folivetti@ufabc.edu.br, thiago.covoes@ufabc.edu.br

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Apache Spark

Spark: Além do MapReduce

De tal forma a facilitar o desenvolvimento de aplicações e generalização de tarefas comuns no MapReduce, foi criada a plataforma Spark.

- Mais eficiente: computação distribuída + multithreading.
- Linguagem estendida.
- Compartilhamento de Dados mais rápido.
- Árvore de processamento.
- Suporte a Scala, Java, Python, R.

Resilient Distributed Dataset: RDD

Abstração principal do Spark:

- A base de dados é um *objeto*.
- Aplica operações nesses objetos, como *métodos*
- Pode estar armazenado em Hadoop, sistema de arquivo comum ou até na memória (transparente).
- Transformações (paralelas) e Ações (saída de dados).
- Refaz operações em caso de falhas.

Criando uma linguagem de processamento de dados

Criando uma linguagem de processamento de dados

Que tipos de operações precisamos?

Criando uma linguagem de processamento de dados

- Temos uma coleção de registros, precisamos aplicar certas operações neles e obter resultados.
- Esses registros são imutáveis, sempre gero nova coleção.

Map

Precisamos processar cada registro de forma independente:

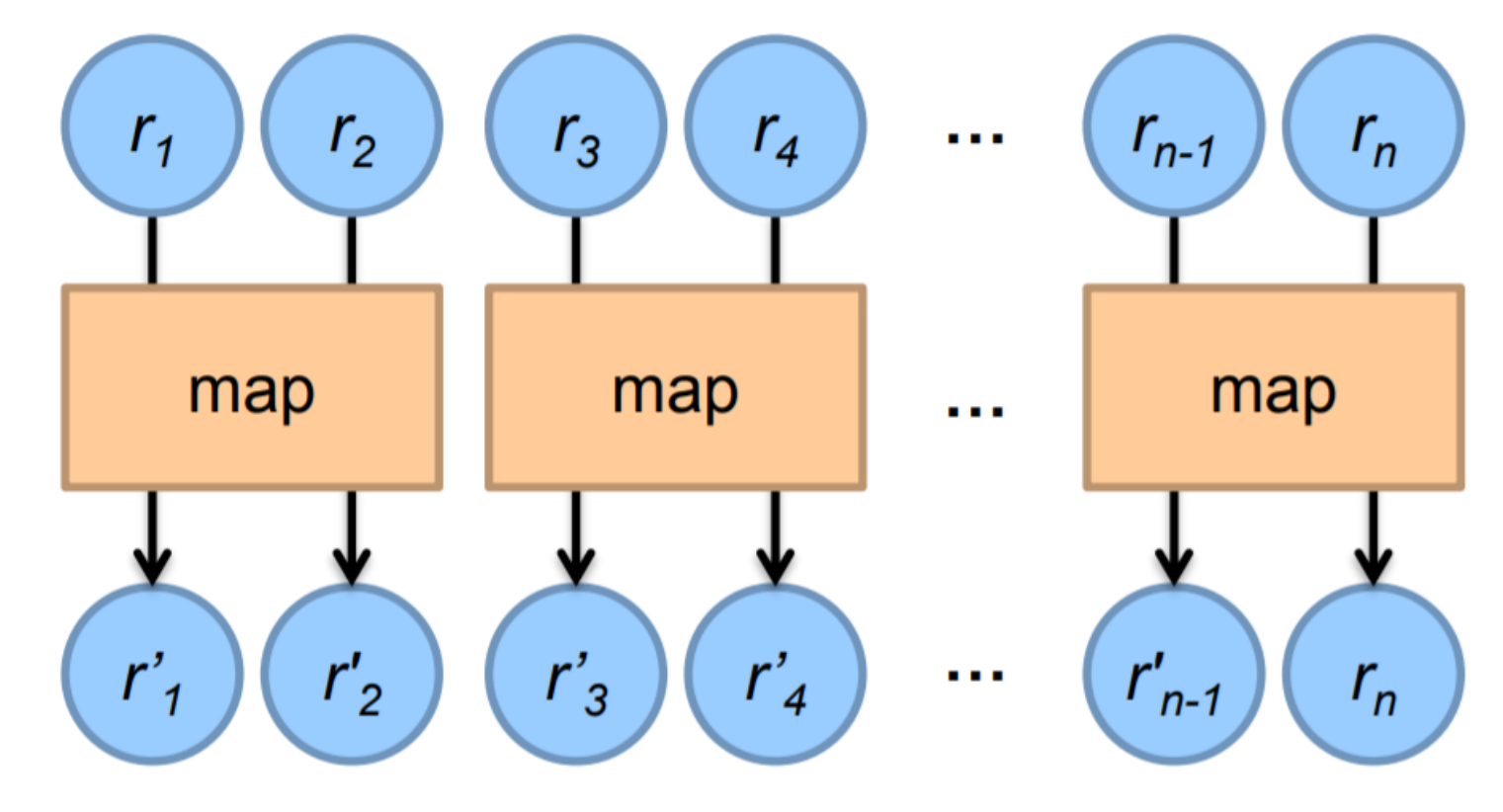

Figura 1: Fonte: https://roegiest.com/

Map

É fácil paralelizar os `mappers`, eles independem dos outros registros!

MapReduce

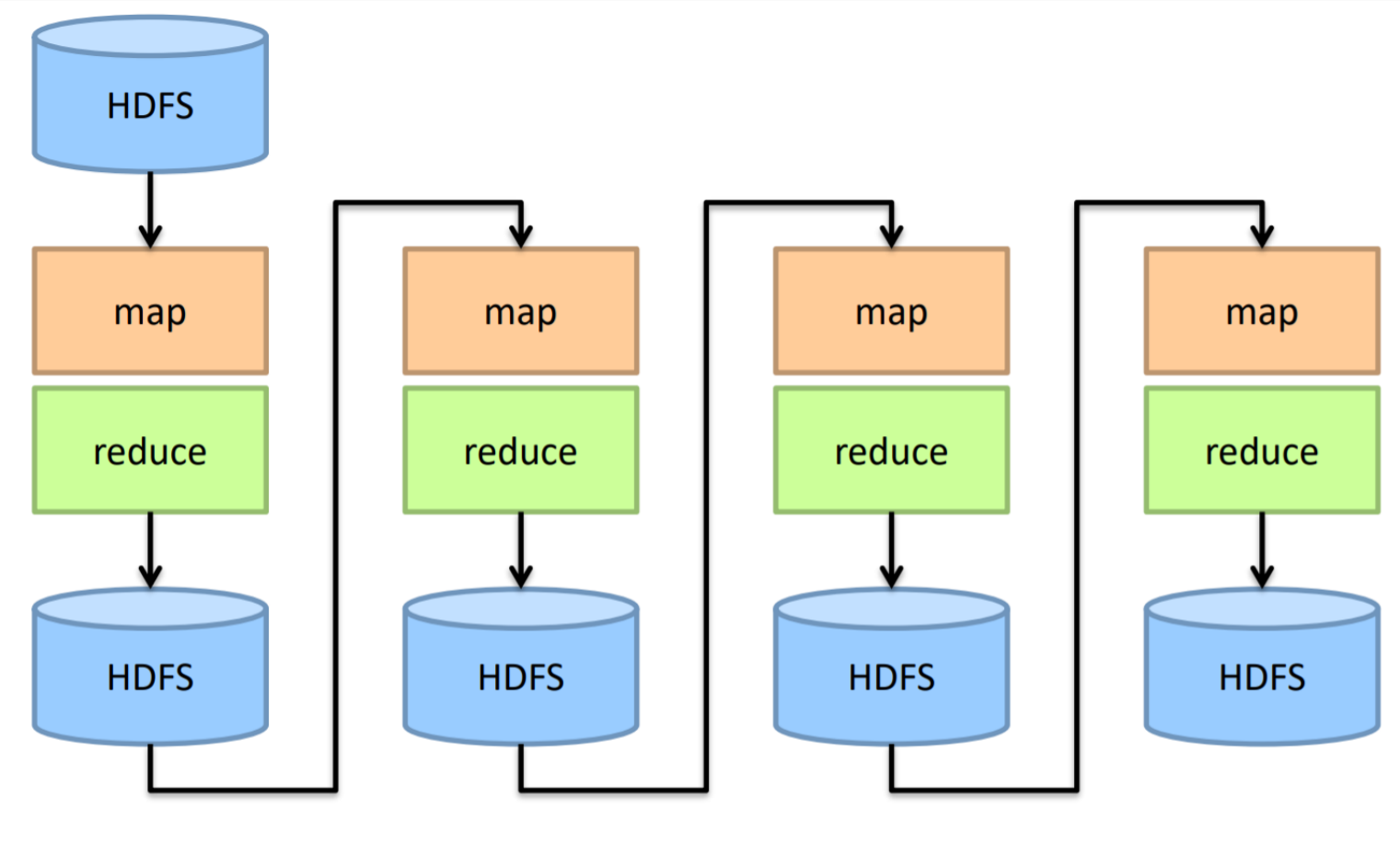

Figura 2: Fonte: https://roegiest.com/

MapReduce

```
List (k1, v1) -> map :: (k1, v1) -> List (k2, v2)
  -> reduce :: (k2, Iterable v2) -> List (k3, v3)
```

Composição

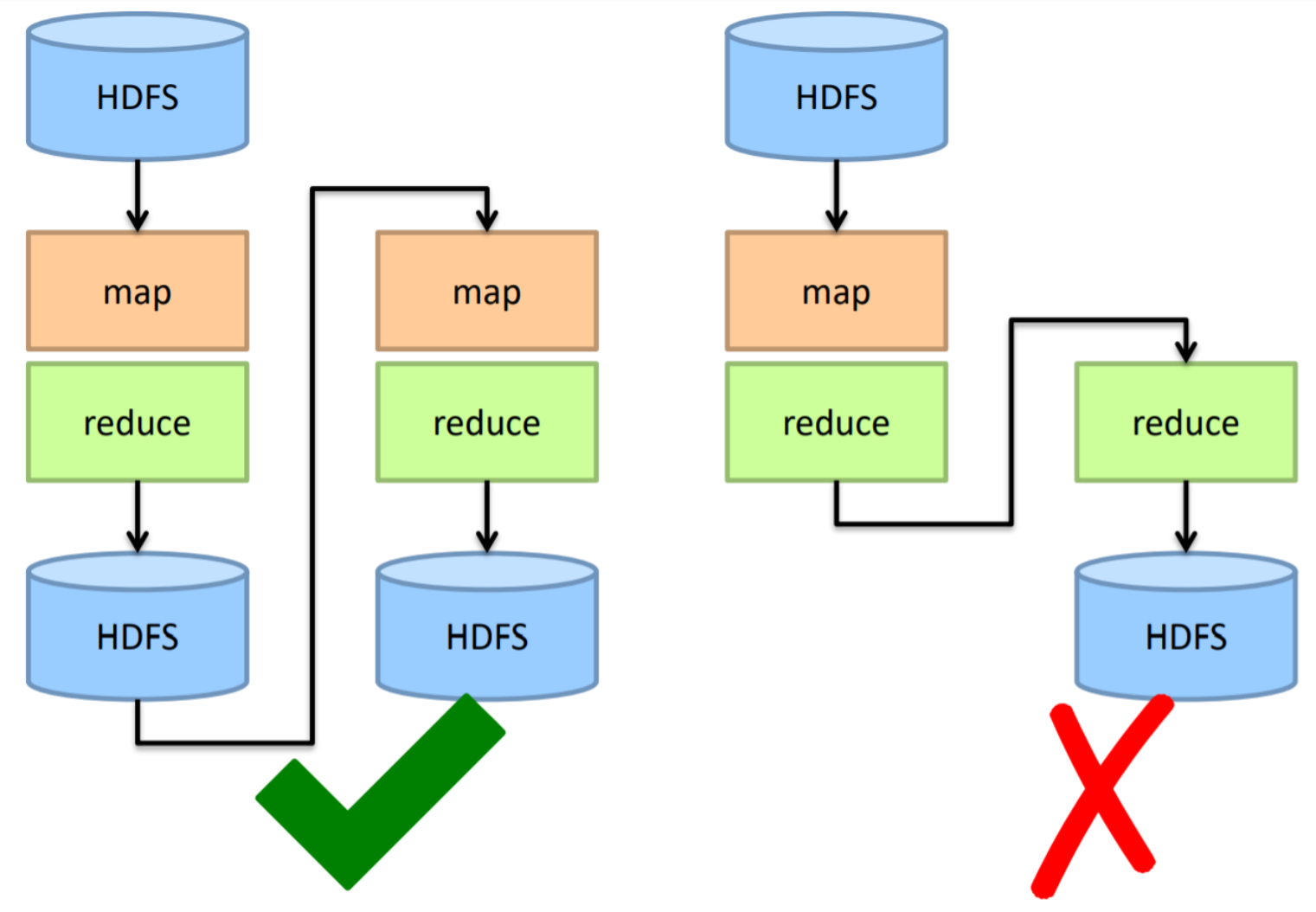

Figura 3: Fonte: https://roegiest.com/

Operações em uma RDD

Tipo Map

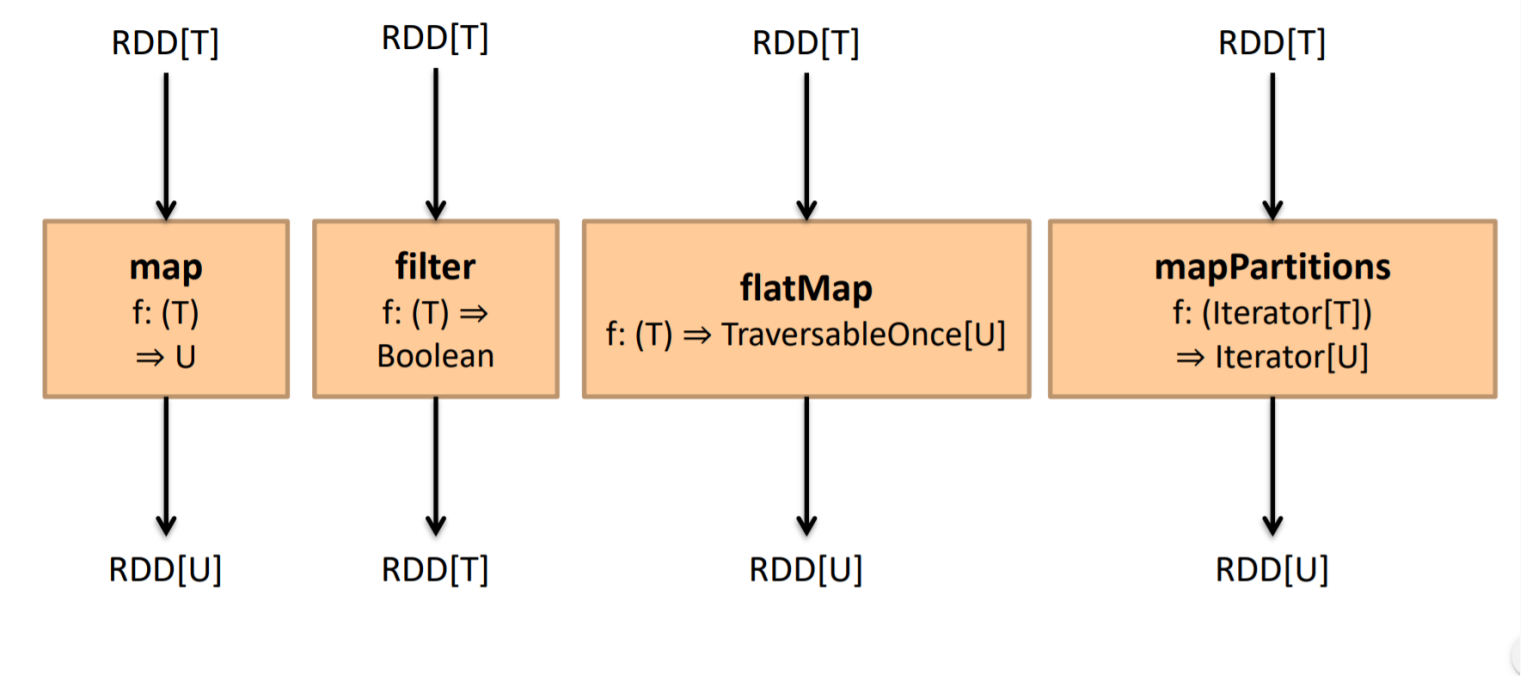

Figura 4: Fonte: https://roegiest.com/

Tipo Reduce

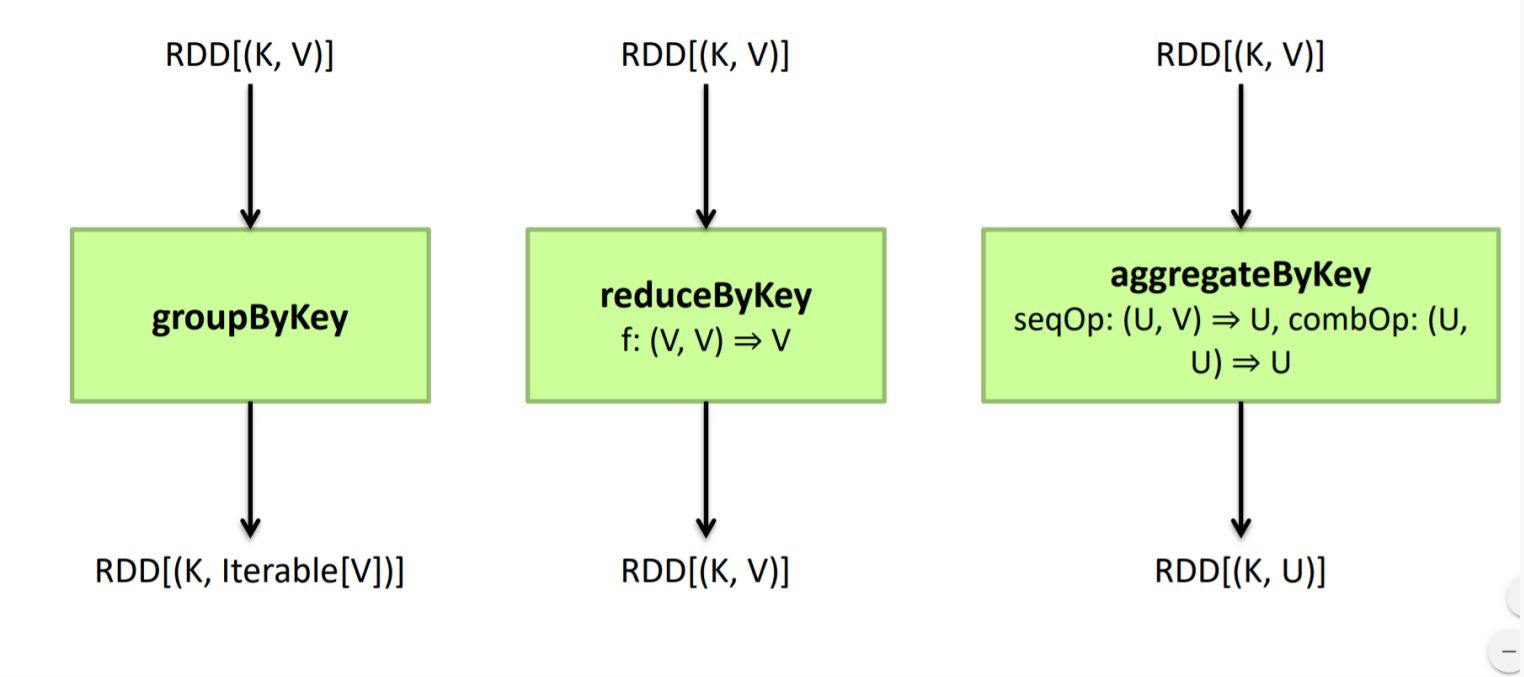

Figura 5: Fonte: https://roegiest.com/

Tipo Sort

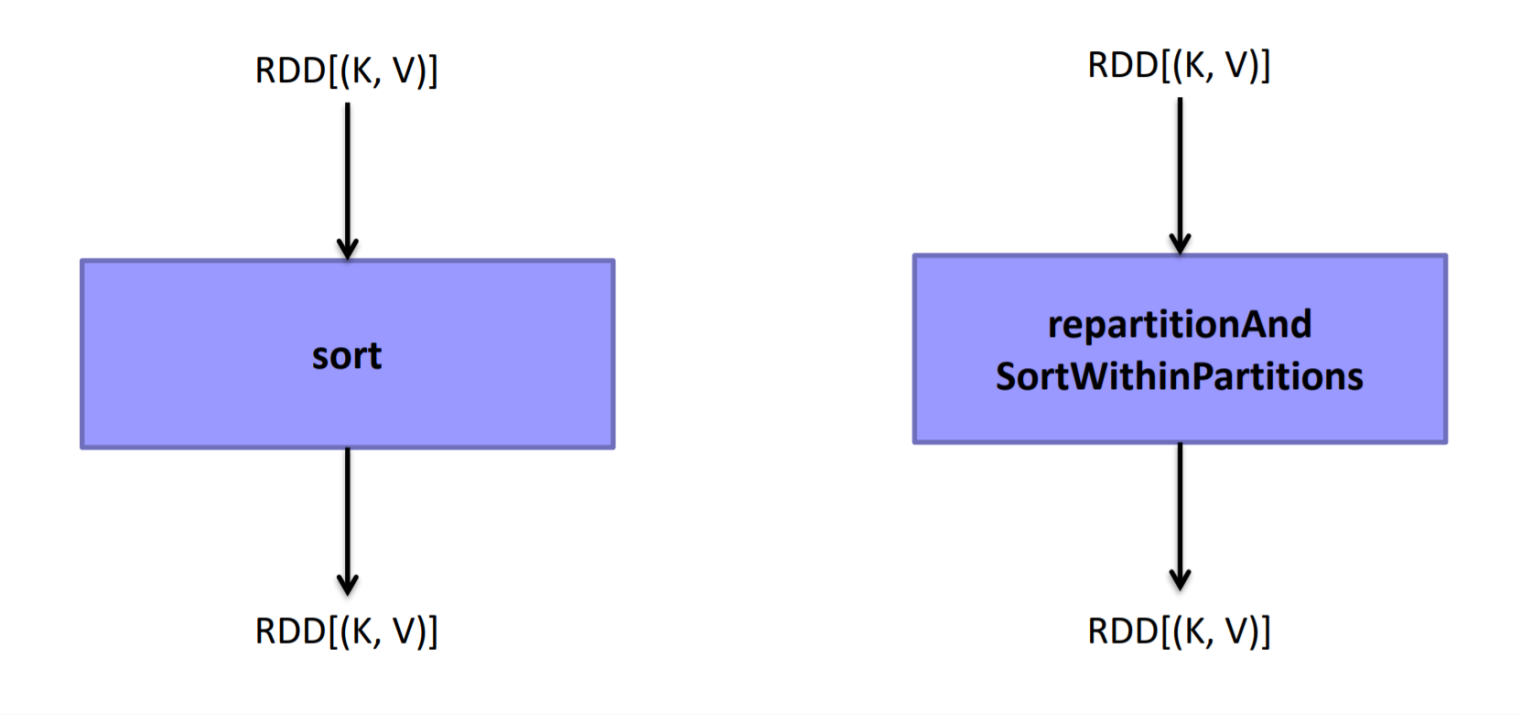

Figura 6: Fonte: https://roegiest.com/

Tipo Join

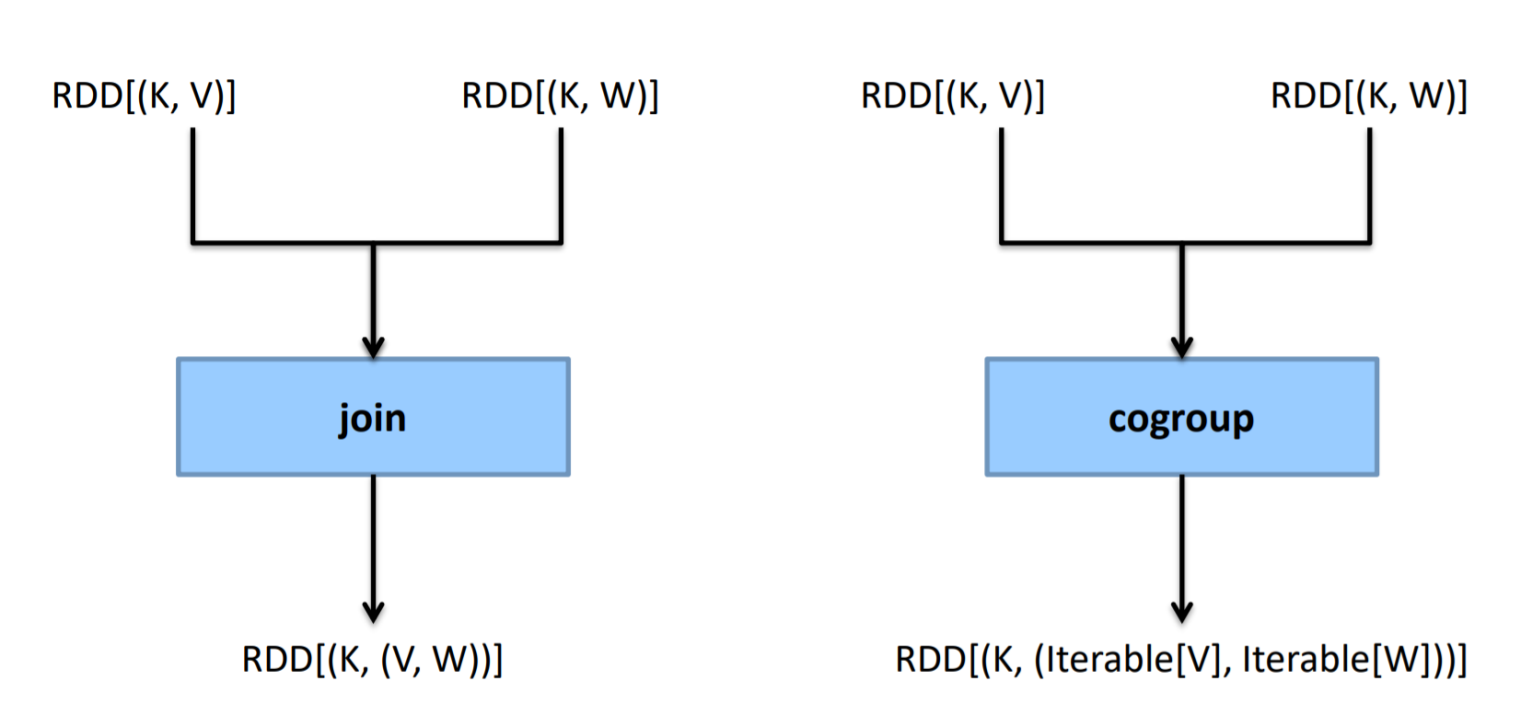

Figura 7: Fonte: https://roegiest.com/

Tipo Join

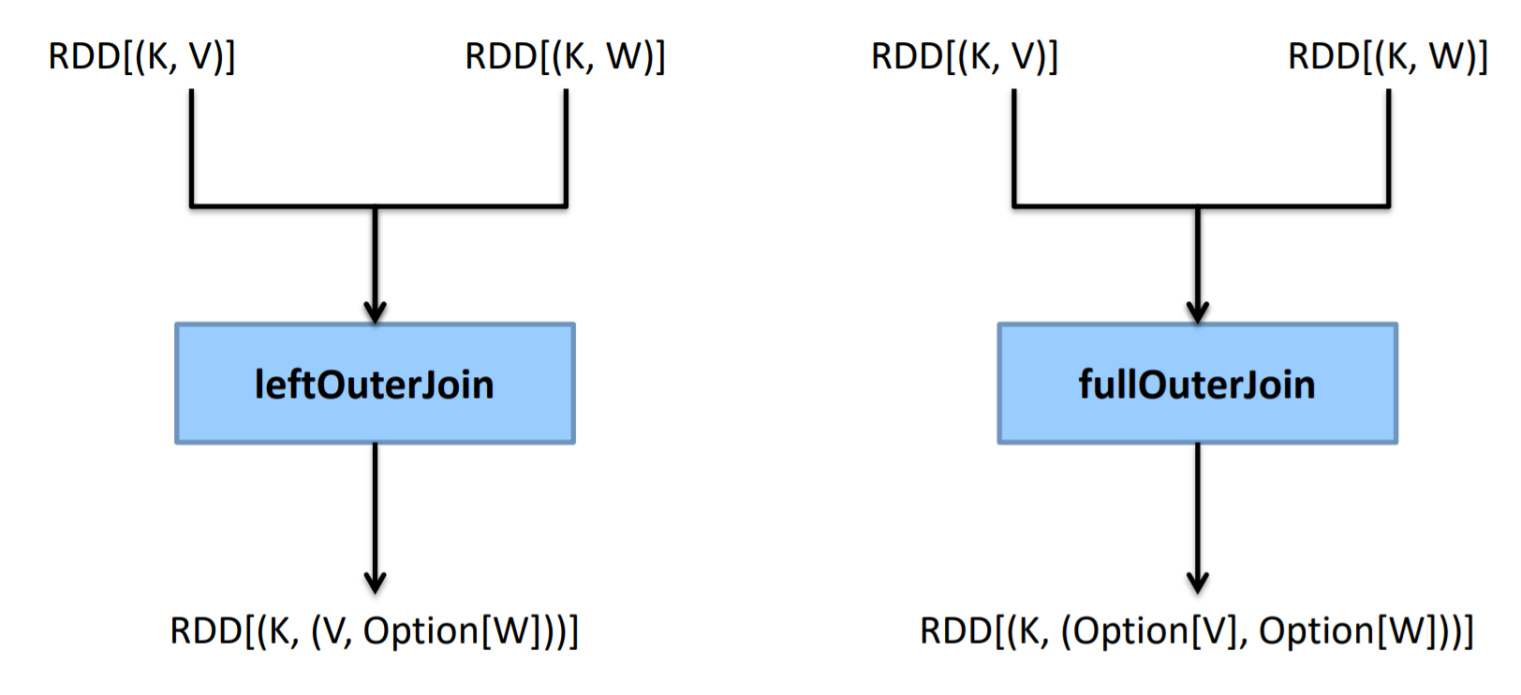

Figura 8: Fonte: https://roegiest.com/

Tipo Set

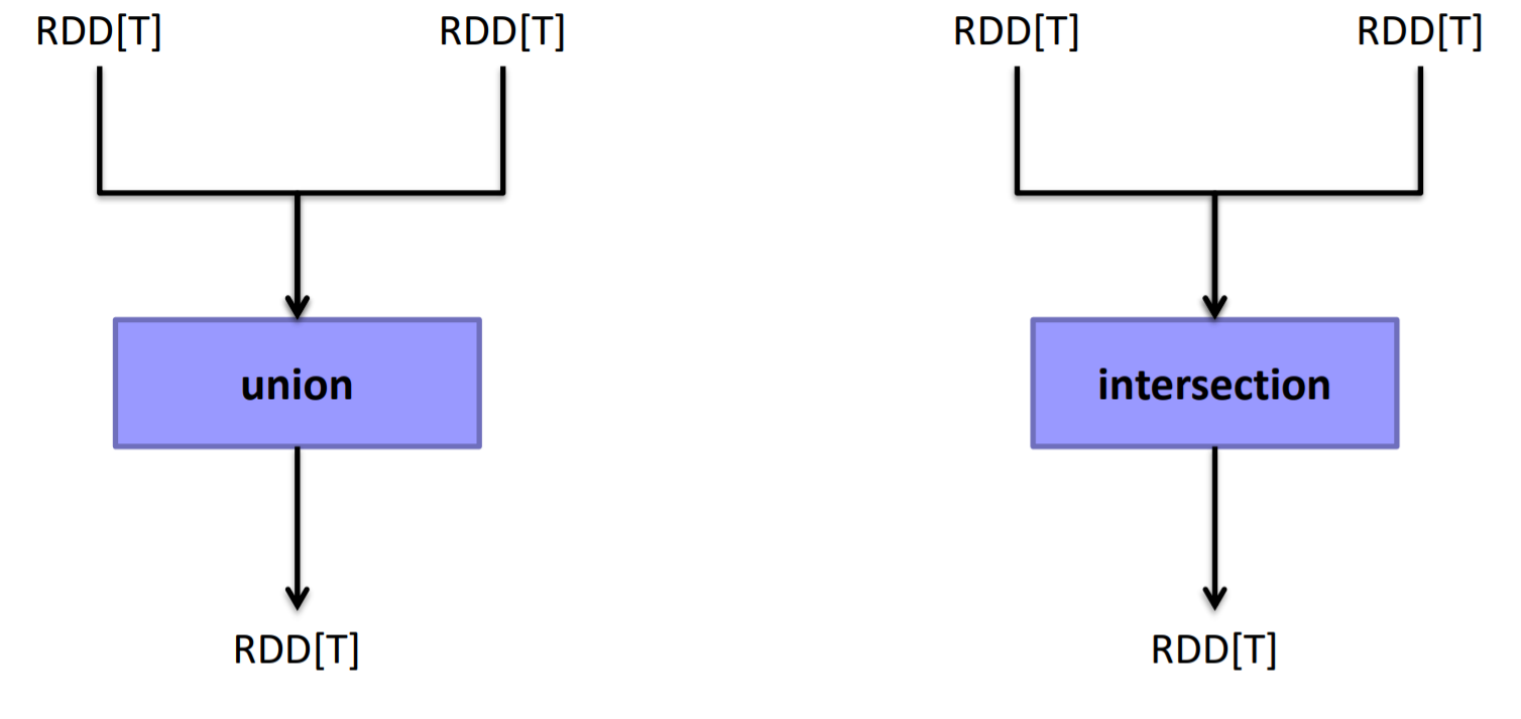

Figura 9: Fonte: https://roegiest.com/

Tipo Set

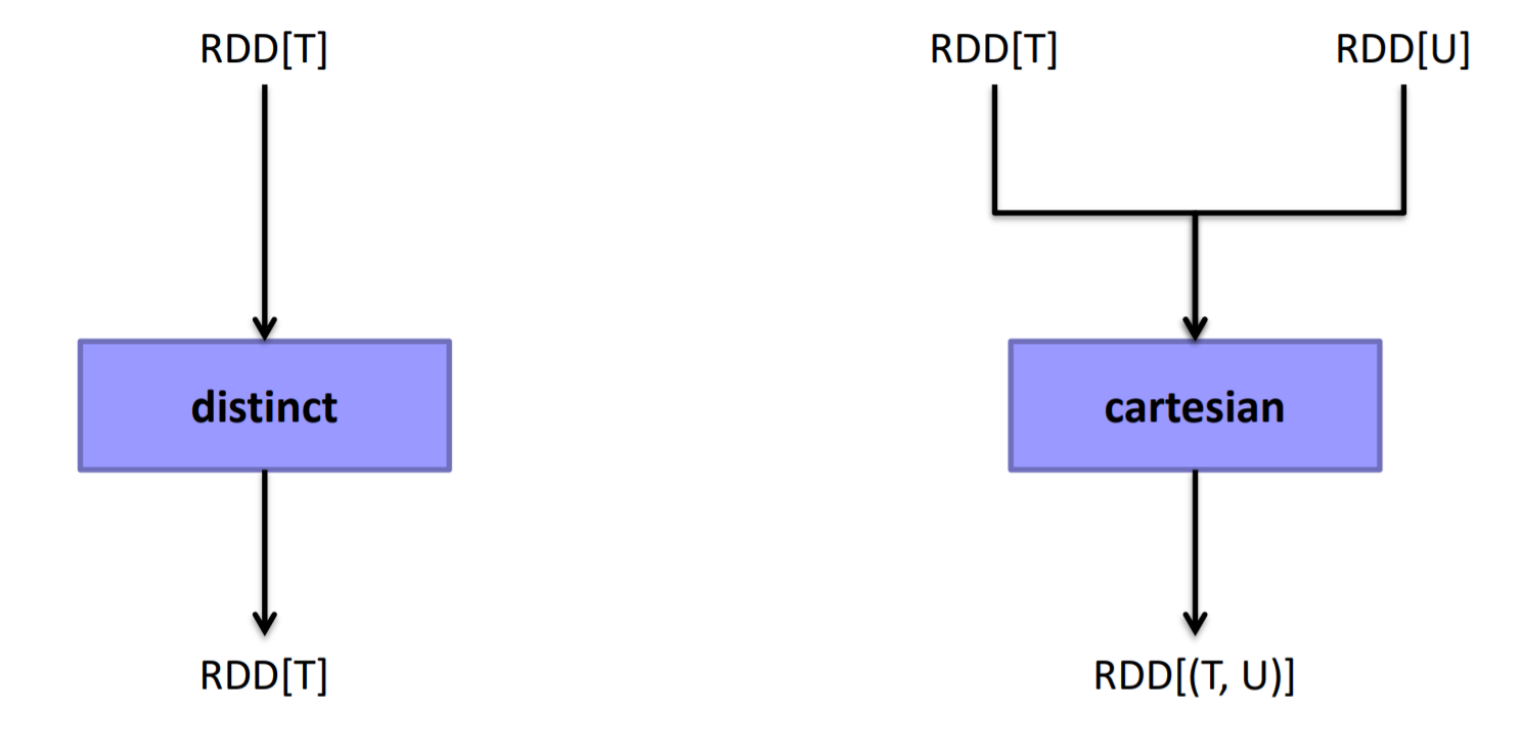

Figura 10: Fonte: https://roegiest.com/

Directed Acyclic Graph: DAG

No MapReduce do Hadoop, cada operação Map-Reduce é independente uma da outra, mesmo quando as tarefas estão interligadas.

Considere o problema de retornar a contagem de palavras ordenadas pela frequência:

```
1 mapper1  :: Linha -> [(Palavra, Integer)]
2 reducer1 :: [(Palavra, [Integer])] -> [(Palavra,
   ↪  Integer)]
3
4 mapper2  :: (Palavra, Integer) -> [(Integer, Palavra)]
5 reducer2 :: [(Integer, [Palavra])] -> [(Integer,
   ↪  [Palavra])]
```

Qual a definição das funções *mapper2* e *reducer2*?

Directed Acyclic Graph: DAG

Primeiro, o Hadoop irá completar TODA a tarefa MapReduce1. Somente após a conclusão e escrita do arquivo de saída (escrita custa caro!), a tarefa MapReduce2 iniciará, para então gerar a resposta.

Directed Acyclic Graph: DAG

No Spark, é feito um grafo direcionado e acíclico (árvore) de processamento que interliga as tarefas de MapReduce (ou outras transformações) para gerar o menor número de escritas possíveis.

Directed Acyclic Graph: DAG

Uma vez que a RDD é imutável e as transformações devem ser puras, é possível fazer a composição dessas funções sem introduzir efeitos inesperados.

Contador de palavras - SPARK

```
1  val textFile = sc.textfile(args.input())
2
3  textFile
4    .flatMap(line => tokenize(line))
5    .map(word => (word, 1))
6    .reduceByKey(_+_)
7    .saveAsTextFile(args.output())
```

Transformações Básicas

Criando uma RDD

Uma RDD pode ser criada a partir de uma coleção em memória ou um arquivo existente no sistema de arquivos (distribuídos ou não)~[1].

```
1 listaComidas = ['bife', 'alface', 'arroz']
2 comidasRDD   = sc.parallelize(listaComidas, 2)
3 textoRDD     = sc.textFile(nomeArquivo, 2)
```

O segundo parâmetro é a quantidade de partições por nó local. O ideal é criar cerca de 2 partições por *core*

[1] os comandos seguem a sintaxe Pyspark

MapReduce

```
1  # [Linha] -> [Int] -> [Int] -> Int
2  textoRDD.map(lambda linha: length(linha))
3          .reduce(lambda (l1, l2): l1 + l2)
```

Nota: reduce é uma ação, e portanto força o processamento.

reduceByKey

Parecido com o procedimento que fizemos até então no Haskell: aplica uma função no resultado de *groupByKey* . *sortByKey*:

```
1 # [Linha] -> [(Palavra, Int)] -> [(Palavra,[Int])]
2 #                               -> [(Palavra, Int)]
3 textoRDD.flatMap(lambda linha: [(w,1) for w in
   ↪  linha.split()])
4         .reduceByKey(lambda f1,f2: f1+f2)
```

groupByKey

Retorna o resultado de *groupByKey* . *sortByKey* após um map:

```
1 # [Linha] -> [(Palavra, Int)] -> [(Palavra,[Int])]
2 textoRDD.flatMap(lambda linha: [(w,1) for w in
  ↪  linha.split()])
3         .groupByKey()
```

filter

Similar ao Haskell, remove da RDD todos os elementos que retorna falso para a função passada como parâmetro:

```
1 # [Linha] -> [(Palavra, Int)] -> [(Palavra,[Int])]
2 textoRDD.flatMap(lambda linha: [(w,1) for w in
   ↪  linha.split()])
3         .reduceByKey(lambda f1,f2: f1+f2)
4         .filter(lambda (w, f): f > 3)
```

join

Une a RDD de tuplas com outra RDD de tuplas dado que as chaves são do mesmo tipo:

```
# [(Palavra, Int)] -> [(Palavra, Int)]
#                  -> [(Palavra, (Int, Int))]
freqPalavraRDD.join(idfPalavraRDD)
```

cartesian

Une duas RDDs através de um produto cartesiano de seus elementos:

```
# [(Palavra, Int)] -> [(Palavra, Int)]
#      -> [(Palavra, Int), (Palavra, Int)]
freqPalavraRDD.cartesian(freqPalavraRDD)
```

Pipelines de processamento

Conforme mencionado anteriormente, o retorno das transformações do Spark são intenções de computação, a sequência de transformações formam um pipeline de processamento a ser computado quando necessário.

```
pipeline1 = RDD.transforma1()
                .transforma2()
                .transforma3()

pipeline2 = pipeline1.transforma1()
                     .transforma2()

pipeline3 = pipeline1.transforma1(pipeline2)
                .transforma2()
```

Pipelines de processamento

Dessa forma, o Spark consegue otimizar o processamento e evitar redundâncias.

Ações Básicas

- .collect(): retorna a RDD em memória.
- .first(): retorna o primeiro elemento da RDD.
- .take(n): retorna os n primeiros elementos da RDD.

Associatividade e Comutatividade

Por que queremos operadores associativos?

Com a associatividade podemos colocar os parênteses onde for mais conveniente:

$$(x_1 \otimes x_2) \otimes (x_3 \otimes x_4 \otimes x_5) \otimes (x_6 \otimes x_7) \quad (1)$$

$$(x_1 \otimes x_2 \otimes x_3 \otimes x_4) \otimes (x_5 \otimes x_6 \otimes x_7) \quad (2)$$

$$(x_1 \otimes x_2) \otimes (x_3 \otimes)(x_4 \otimes x_5 \otimes x_6 \otimes x_7) \quad (3)$$

$$(4)$$

Por que queremos operadores associativos?

Dessa forma, não importa como os dados estão agrupados, o resultado será o mesmo!

Por que queremos comutatividade?

Podemos trocar a ordem dos operandos!

$$(x_1 \otimes x_2) \otimes (x_3 \otimes x_4 \otimes x_5) \otimes (x_6 \otimes x_7) \tag{5}$$

$$(x_5 \otimes x_4 \otimes x_3 \otimes x_1) \otimes (x_2 \otimes x_6 \otimes x_7) \tag{6}$$

$$(x_3 \otimes x_2) \otimes (x_1) \otimes (x_7 \otimes x_5 \otimes x_6 \otimes x_4) \tag{7}$$

$$\tag{8}$$

Por que queremos comutatividade?

Dessa forma, não importa como distribuímos os dados, o resultado final será o mesmo!

Implicações

- Não sabemos quando a tarefa começa
- Não sabemos quando a tarefa termina
- Não sabemos quando as tarefas se interrompem
- Não sabemos quando dados intermediários chegam

Sem problemas!

Pergunta

Uma sequência de `flatMaps` pode ser composta e otimizada?

```
1  val textFile = sc.textfile(args.input())
2
3  textFile
4    .flatMap(line => tokenize(line))
5    .flatMap(word => tokenize(word))
6    .map(char => (char, 1))
7    .reduceByKey(_+_)
8    .saveAsTextFile(args.output())
```

Resposta

Sim!

```
1  val textFile = sc.textfile(args.input())
2
3  textFile
4    .flatMap(line => tokenize(line)
5                       .myFlatMap(word => tokenize(word))
6            )
7    .map(char => (char, 1))
8    .reduceByKey(_+_)
9    .saveAsTextFile(args.output())
```

Comentários Finais

Comentários Finais

- Spark apresenta um conjunto de funções além do tradicional `MapReduce`.
- Esse framework otimiza sequências de `Map` e `Reduces` de tal forma a não precisar passar várias vezes pelo RDD.