

Processamento de dados massivos com Spark: conceitos, desafios e programação

Vinícius Dias
viniciusvdias@ufop.edu.br



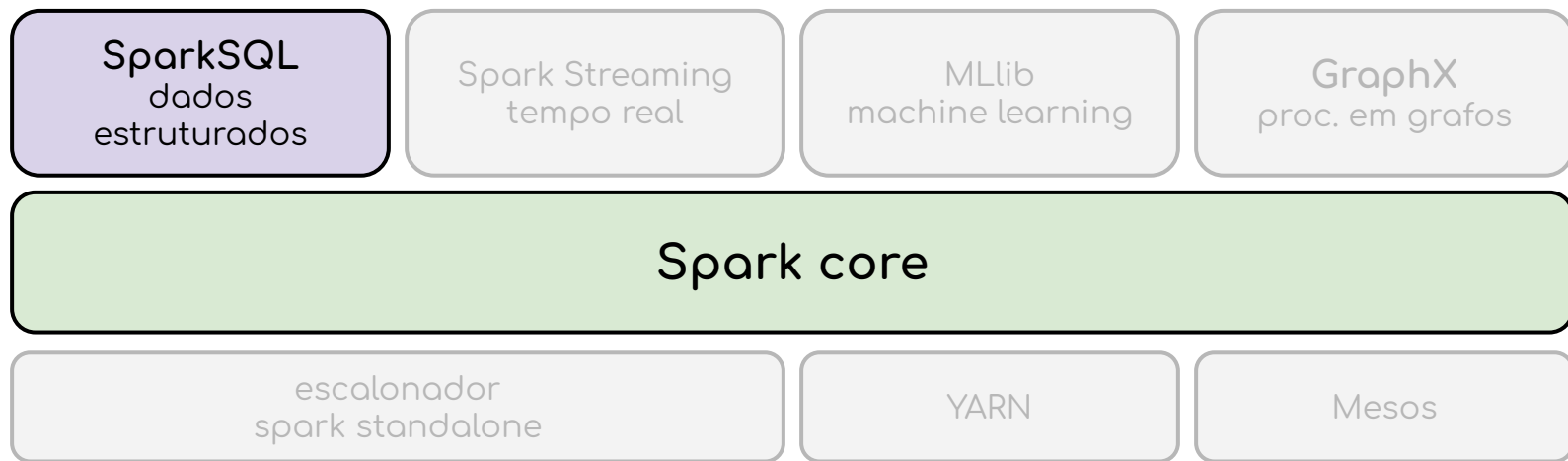
UFOP

Universidade Federal
de Ouro Preto

Roteiro

1. Motivação e contextualização
2. Spark: uma solução genérica e integrada
- 3. **SparkSQL: lidando com dados estruturados**
4. SparkStreaming: processamento contínuo

Contextualizando ...



Até agora temos visto funcionalidades de propósito geral do **Spark core**.

⇒ Este módulo trata de uma abstração sobre Spark para dados estruturados: o **SparkSQL**.

Formato de arquivos

- ⇒ O Spark provê suporte a muitos formatos de arquivos.
- ⇒ Dependendo da estrutura, a manipulação e carregamento de dados pode mudar consideravelmente.
- ⇒ Como consequência, pode ser necessário lidar de formas específicas para cada formato de entrada.
 - e.g. JSON, CSV, texto comum ou binário.
 - veremos algumas possibilidades neste módulo.
- ⇒ Além disso **a API está em constante transformação.**
 - neste curso, tomamos como base o **Spark 2.2.0**
 - é importante utilizar as documentações certas.

Arquivos JSON

(JavaScript Object Notation)

- ⇒ formato semi-estruturado.
- ⇒ baseado em **nome/valor**.
- ⇒ além disso, possui uma noção mais clara de tipos de dados.
- ⇒ Os elementos de um arquivo JSON são:
 - pares nome: valor que descrevem um campo.

`"nome": "CICERO JOSE DE SOUZA"`

- listas de valores.

`"capitais": ["Rio de Janeiro", "Belo Horizonte", "São Paulo"]`

- objetos ou listas de objetos.

```
{  "id":1003403,"nome":"CICERO JOSE DE SOUZA",
  "cpf":"***.224.457-**",
  "cargo":"GUARDA DE ENDEMIAS",
  "vinc":"CEDIDO SUS/LEI 8270",
  "ingresso":"01/03/2006"}
```

JSON em Spark

Como em JSON fica bem definido o que é cada entidade e atributo, fica fácil criar tipos específicos para o dado.

⇒ Um detalhe é que em Spark, cada registro de JSON deve estar em uma linha do arquivo:

```
{"id":1576623,"nome":"DANIEL CHEDID PEREIRA JORDAO  
RAMOS","cpf":"***.379.611-**","cargo":"ANALISTA ADMINISTRATIVO","vinc":"ATIVO  
PERMANENTE","ingresso":"03/06/2014"}  
{"id":1003403,"nome":"CICERO JOSE DE SOUZA","cpf":"***.224.457-**","cargo":"GUARDA DE  
ENDEMIAS","vinc":"CEDIDO SUS/LEI 8270","ingresso":"01/03/2006"}
```

...

Esquema de uma base de dados

- ⇒ Se refere ao formato geral dos dados.
 - quais campos compõem os registros ?
 - quais os tipos de cada campo ?
 - **i.e.** texto, inteiro, ponto flutuante, complexo, etc.
- ⇒ Em geral, dizemos que um dado está estruturado se o mesmo possui um esquema.

Exemplo com "Pessoa" e esquema

⇒ Um esquema deve caracterizar hierarquicamente cada atributo de um conceito.

- quais os **nomes** de cada campo ?
- quais os **tipos** de cada campo ?

⇒ Atributos de "Pessoa".

- **nome:** Nome, **tipo:** texto/string
- **nome:** CPF, **tipo:** inteiro/long
- **nome:** RG, **tipo:** texto/string
- **nome:** Sexo, **tipo:** texto/string
- **nome:** Naturalidade, **tipo:** texto/string
- **nome:** Email, **tipo:** texto/string
- **nome:** Salário, **tipo:** ponto flutuante/double

Um exemplo com um arquivo JSON

Considere o arquivo JSON `station-data.json`.

- ⇒ Ele é um pouco mais sofisticado do que os exemplos que vimos até agora.
- ⇒ As informações pessoais da pessoa estão agrupadas em um **sub-objeto JSON**.
- ⇒ Veja o exemplo de uma linha:

em Spark isto
deve estar em
uma linha, pois é
um registro

```
{"station_id":2,  
  "name":"San Jose Diridon Caltrain Station",  
  "lat":37.329732,  
  "long":-121.901782,  
  "dockcount":27,  
  "landmark":"San Jose",  
  "installation":"2013-08-06"}
```

Qual é o esquema?

⇒ Esquema para "Estações de bicicletas".

- **nome:** station_id, **tipo:** int
- **nome:** name, **tipo:** string
- **nome:** lat, **tipo:** double
- **nome:** long, **tipo:** double
- **nome:** dockcount, **tipo:** int
- **nome:** landmark, **tipo:** string
- **nome:** installation, **tipo:** date

Um exemplo com um arquivo JSON (cont.)

Vamos carregar dados de estações no formato JSON e visualizar.

```
scala> val stations = spark.read.json("data/station-data.json")
```

```
scala> stations
```

```
resX: org.apache.spark.sql.DataFrame = [dockcount: bigint, installation: string ... 5  
more fields]
```

```
scala> stations.show
```

```
scala> stations.printSchema
```

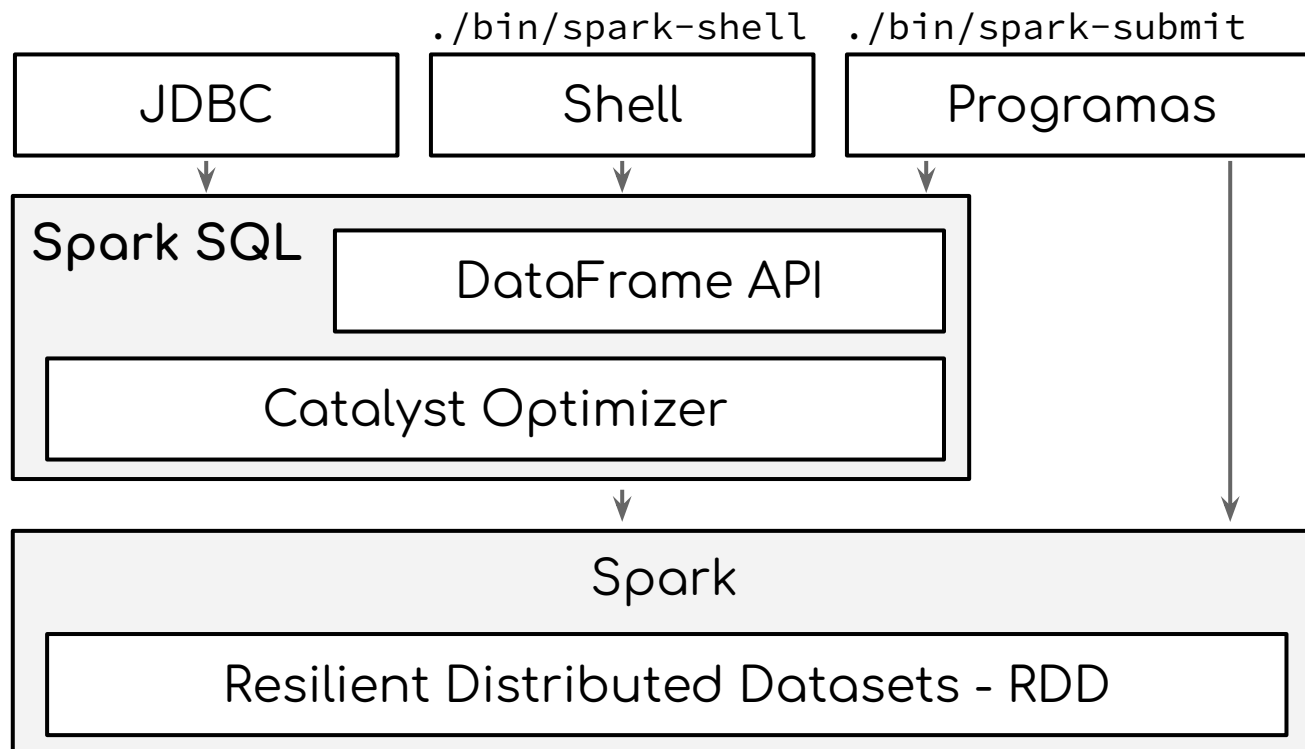
```
root
```

```
|-- dockcount: long (nullable = true)  
|-- installation: string (nullable = true)  
|-- landmark: string (nullable = true)  
|-- lat: double (nullable = true)  
|-- long: double (nullable = true)  
|-- name: string (nullable = true)  
|-- station_id: long (nullable = true)
```

SparkSQL

- ⇒ Módulo de Spark para processamento de dados estruturados.
- ⇒ Provê uma abstração de dados chamada DataFrame
- ⇒ Funciona como um sistema de consultas SQL para dados no cluster.

SparkSQL



- ⇒ SparkSQL usa a interface de RDDs para implementar a abstração de **DataFrames**.
- ⇒ Essa abstração é otimizada para tipos de dados estruturados (**Catalyst Optimizer**)
- ⇒ O acesso a esse novo recurso continua sendo através de uma shell ou por submissão de programa.

SQLContext

Assim como tínhamos um contexto toda aplicação Spark genérica, temos para SparkSQL:

- ⇒ Em uma instância do `./bin/spark-shell`
 - `Spark core` tem o `SparkContext` através da variável `sc`.
 - `SparkSQL` tem o `SQLContext` através da variável `spark.sqlContext`.
 - mas como `DataFrames` são criados a partir de `RDDs`, o `SparkContext` é uma dependência do `SQLContext`.

Conferindo os contextos na shell

Abra um `./bin/spark-shell` e confirme a existência dos dois contextos, através das variáveis `sc` e `sqlContext`.

```
$ ./bin/spark-shell
```

```
scala> sc
```

```
resX: org.apache.spark.SparkContext = org.apache.spark.SparkContext@78731200
```

```
scala> sc.appName
```

```
resX: String = Spark shell
```

```
scala> spark.sqlContext
```

```
resX: org.apache.spark.sql.SQLContext =  
    org.apache.spark.sql.SQLContext@769938d6
```

Estrutura de uma aplicação que utiliza o SparkSQL

```
// arquivo com nome SparkSQLApp.scala
import org.apache.spark.sql.SparkSession

object SparkSQLApp {
  def main(args: Array[String]): Unit = {

    val spark = SparkSession
      .builder()
      .appName("Spark SQL basic example")
      .config("spark.some.config.option", "some-value")
      .getOrCreate()

    val numsDf = spark.read.json("file:///vagrant/data/movimentacoes.json")
    println("Esquema do DataFrame:\n")
    numsDf.printSchema()

    spark.stop()
  }
}
```


DataFrames

- ⇒ São coleções de dados **distribuídas** e organizadas em colunas.
- são implementadas a partir de RDDs.
 - por isso são distribuídas !
 - mas isso é transparente para o programador.
 - para DBA's ...
 - pense no dataframe como tabelas de um banco.
 - para programadores em R/Python ...
 - equivalente aos dataframes dessas linguagens.
 - Existem APIs para manipulação de DataFrames em Scala, Java e Python.
 - constante mudança (amanhã pode existir para R).
 - vamos manter nossos exemplos em Scala.

Criando DataFrames

⇒ O ponto de partida para criação de dataframes é o **SparkSession**. Iremos abordar dois métodos neste curso:

- criando dataframes a partir de fontes de dados.
 - ou seja, arquivos (json, parquet file, etc.)
 - neste curso iremos lidar com arquivos **json**.
 - já vimos um exemplo disso
- criando dataframes a partir de **RDDs existentes**.
 - como veremos a seguir isso pode envolver a inferência ou especificação de um esquema para os dados.
 - afinal, estamos lidando com dados estruturados.

→
não iremos
abordar essa
opção neste
minicurso

Salvando no sistema de arquivos

Um Dataframe pode ter sido carregado, transformado ou criado a partir de qualquer método ou fonte que será possível salvá-lo como JSON de maneira direta.

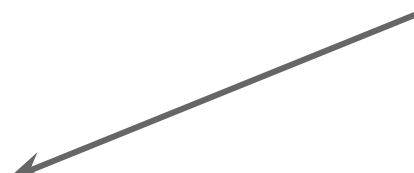
Considere que carregamos um DataFrame a partir de um arquivo CSV:

```
scala> var pessoas = spark.read.csv("data/pessoas.csv")
scala> pessoas = pessoas.toDF("cpf", "nome", "cidade")
scala> pessoas = pessoas
      .withColumn("cpf", regexp_replace(col("cpf"), "cpf::", ""))
scala> pessoas = pessoas
      .withColumn("nome", regexp_replace(col("nome"), "nome::", ""))
scala> pessoas = pessoas
      .withColumn("cidade", regexp_replace(col("cidade"), "cidade::", ""))
```

Salvando no sistema de arquivos (cont.)

Para salvar esse mesmo DataFrame como um arquivo JSON, precisamos usar a chamada

outros formatos populares de saída são possíveis (csv, json, parquet, etc.)



`df.write.json(<arq>)`, onde:

⇒ **arq:** é o nome do arquivo de saída (com o prefixo do sistema de arquivos).

```
scala> pessoas.write.json("data/pessoas.json")
```

Operações com DataFrames

(consultas método 1)

⇒ Retornam resultado imediato (implementadas com **ações**).

- show, first, take, count, etc.
- printSchema (esse nós já vimos).

⇒ Retornam novos DataFrames (implementadas com **transformações**).

- select
- filter
- join
- groupBy
- count (**esse count aqui não será uma ação !!**)

⇒ Muitas outras operações na documentação ...

Operações sobre DataFrames

Vamos fazer um conjunto de operações sobre dois DataFrames.

- ⇒ A semântica das operações continua a mesma.
- ⇒ Apenas o seu uso que é um pouco diferente.
 - em se tratando de um modelo estruturado.

Primeiro, criamos os DataFrames de estações e viagens de bicicleta, através dos arquivos JSON:

```
scala> val stations = spark.read.json("data/station-data.json")  
scala> val trips = spark.read.json("data/trip-data.json")
```

Select e filter

`df.select (coluna1, coluna2, ...):`

- ⇒ executado sobre um DataFrame e seleciona algumas colunas do Dataframe, retornando um novo.
- ⇒ `coluna1, coluna2, ...` são strings com os nomes das colunas que se deseja selecionar.
- ⇒ o número de linhas será o mesmo.

`df.filter (condicoes):`

- ⇒ executado sobre um DataFrame e potencialmente filtra linhas do DataFrame, retornando um novo.
- ⇒ as colunas permanecerão as mesmas.
- ⇒ Por exemplo:
 - `df.filter (col("idade") > 20)`
 - seleciona as linhas de `df` em que o valor da coluna `idade` é maior do que 20.

São úteis para reduzir o tamanho do problema logo no início da consulta.

Select e filter

Objetivo: selecionar o ID e o nome de estações em San Jose que possuem mais do que 12 espaços para bicicleta.

```
scala> val sanJoseStations = stations
      .filter(col("landmark") === "San Jose" && col("dockcount") > 12)
      .select("station_id", "name")
```

```
scala> sanJoseStations.show
```

note que a condição de igualdade dentro dos operadores é diferente: três caracteres "="

Isso internamente será transformado em uma função no formato que vimos para os RDDs comuns

Join, GroupBy e count

`df.join(outroDf, condicao):`

- `outroDf`: dataframe que irá ser juntado com `df`.
- `condicao`: critério para juntar os dataframes.
 - envolve a chave desejada.
 - a especificação dos campos na condição é similar ao `filter: df("coluna") == outroDf("coluna")`

`df.groupBy(coluna1, coluna2, ...):`

- agrupa um dataframe utilizando **como chave o conjunto de colunas especificado** no parâmetro.
 - pode ser uma ou mais.
- geralmente o próximo passo é aplicar uma operação em cada grupo.

`grupos.count():`

- ⇒ conta o número de elementos de cada grupo do dataframe.
- ⇒ pode ser usado logo depois do `groupBy`, por exemplo.

Join, GroupBy e count

Agora, vamos juntar os dois dataframes:

```
scala> val tripsStations = trips  
      .join(stations, trips("start_terminal") === stations("station_id"))
```

Quantas viagens por estação?

```
scala> val byStations = tripsStations  
      .groupBy(col("station_id"), col("name"))  
      .count
```

Quantas viagens por local?

```
scala> val byLandmark = tripsStations  
      .groupBy(col("landmark"))  
      .count
```

este count será aplicado a cada grupo e retornará um novo dataframe com a contagem por grupo.

Considerações sobre desempenho

- ⇒ O módulo SparkSQL tem acesso a mais conhecimento sobre o dado do que no Spark core.
 - afinal, a informação está estruturada.
 - com **campos tipados e bem definidos**.
- ⇒ Com isso, é capaz de fazer caching por colunas do DataFrame.
 - isso implica em colocar atributos do mesmo tipo perto.
 - utilizando a memória de forma mais eficiente e possibilitando consultas mais direcionadas.
 - e.g. do tipo "selecione esta coluna do DF".

Considerações sobre desempenho (cont.)

- ⇒ Com o **Spark core** tradicional, muitas das operações que envolviam coletar um subconjunto dos dados poderiam gerar computação de toda o RDD.
- ⇒ O armazenamento de DataFrames no **SparkSQL**, baseado em colunas, permite realizar esse tipo de consulta de forma mais eficiente.
 - consultas com restrições são repassadas para o gerenciador de memória.
 - ele pode conseguir atender a consulta buscando apenas o subconjunto do dado.
 - isso é possível pelo **armazenamento colunar**.

Persistência em SparkSQL (cont.)

No **primeiro caso**, criamos e persistimos o DataFrame diretamente.

```
scala> val stations = spark.read.json("data/station-data.json")
scala> stations.cache
scala> stations.count // gera computação, a computação preguiçosa
                      // também se aplica aos DataFrames.
```

[Jobs](#)[Stages](#)[Storage](#)[Environment](#)[Executors](#)

Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
PhysicalRDD [cidade#82,cnpj#83L,infoPessoal#84,movimentacao#85,pis#86L], MapPartitionsRDD[55] at map at JsonRDD.scala:41	Memory Deserialized 1x Replicated	2	100%	4.9 KB	0.0 B	0.0 B

Para eliminar o DataFrame da memória, a chamada é a mesma:

```
scala> stations.unpersist
```

Roteiro

1. Motivação e contextualização
2. Spark: uma solução genérica e integrada
3. SparkSQL: lidando com dados estruturados
- 4. **SparkStreaming: processamento contínuo**

Aplicações em lote

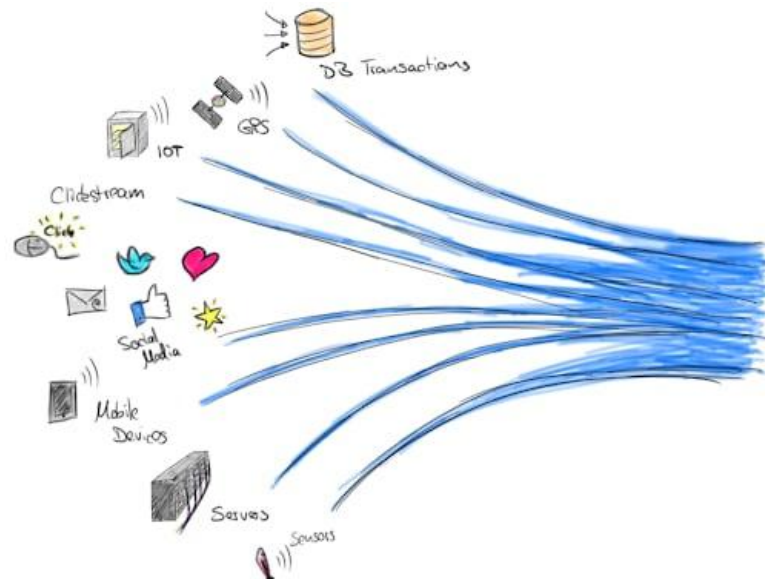
Algumas aplicações possuem um comportamento de execução *offline*.

- ⇒ o dado chega, é tratado e armazenado.
- ⇒ o dado é analisado.
- ⇒ o processo continua, mas sempre podemos voltar atrás e refazer análises e execuções.
- ⇒ dizemos que neste caso, a aplicação **não é sensível à latência**.
 - o Spark core (propósito geral) é um exemplo disso.
 - portanto, o MapReduce também se encaixa aqui.

Execuções com essas características são denominadas **aplicações em lote** (batch).

Aplicações sensíveis à latência

- ⇒ Muitas vezes o dado está disponível na forma de fluxos.
- ⇒ Não há tempo para armazenamento sistemático.
- ⇒ A validade da informação é muito menor.
- ⇒ Dados de redes sociais (twitter), monitoração, detecção de anomalias, etc.

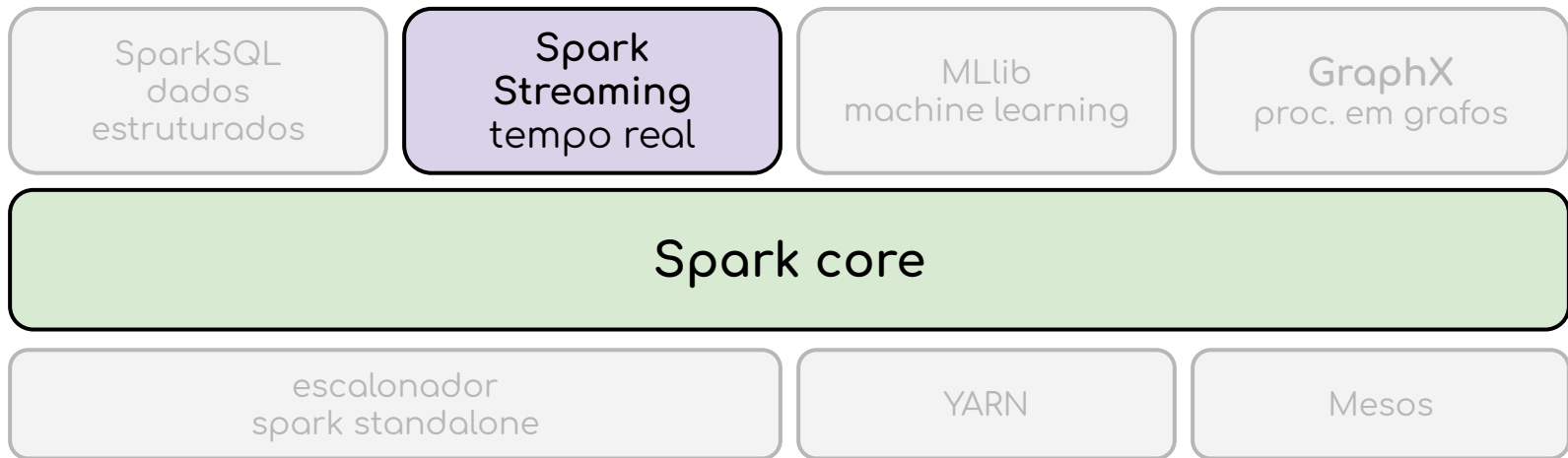


Aplicações sensíveis à latência

- ⇒ A análise desse tipo de dado precisa ser feita em **tempo real, continuamente**.
- ⇒ Dizemos que essas aplicações são sensíveis à latência.
- ⇒ Geralmente essas aplicações não possuem uma terminação definida.

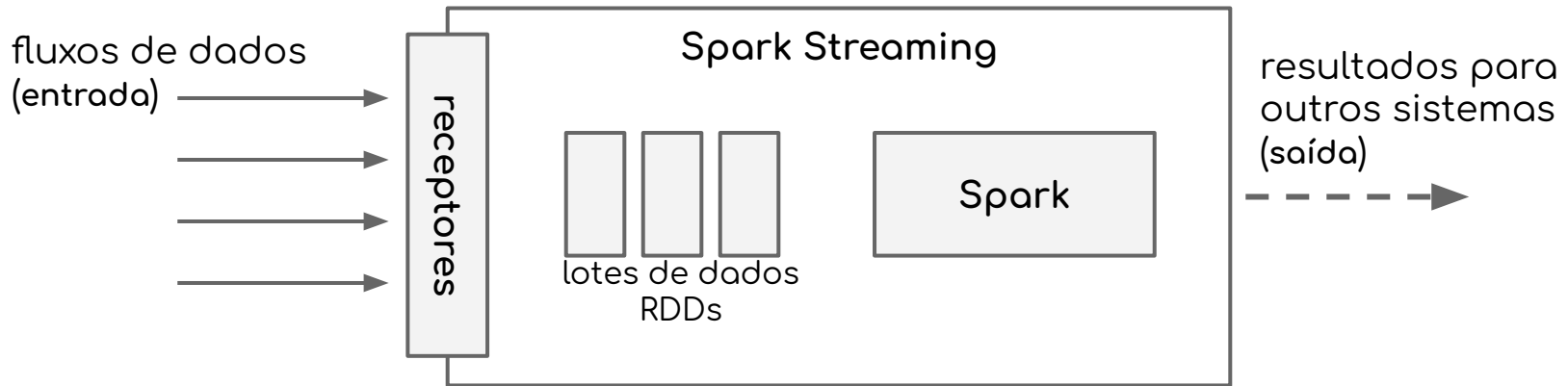
Execuções com essas características são denominadas aplicações de fluxos contínuos (streaming).

Contextualizando ...



⇒ Este módulo trata de uma abstração sobre Spark para fluxos de dados: o **SparkStreaming**.

Arquitetura



- ⇒ Os fluxos de dados (entradas) são discretizados de acordo com um intervalo de tempo (*batch interval*).
- ⇒ Cada batch equivale a um RDD e sempre está associado a um intervalo.
- ⇒ Operações aplicadas sobre o fluxo de dados criam efeito em todos os lotes (*batch*).
- ⇒ Operações especiais podem considerar mais de um lote para agregar informações ao longo do tempo.

A abstração DStream

- ⇒ Todos os módulos construídos para Spark são construídos sobre o **Spark core**.
 - isto é, sobre a abstração básica de RDDs.
- ⇒ Por definição, um RDD por si só induz um processamento em lote.
- ⇒ Para lidar com fluxos contínuos utilizando essa abstração em lote de RDDs, construiu-se a seguinte abstração:
 - **DStream**, ou *discretized streams*.

DStream é uma sequência de dados que chega ao longo do tempo.

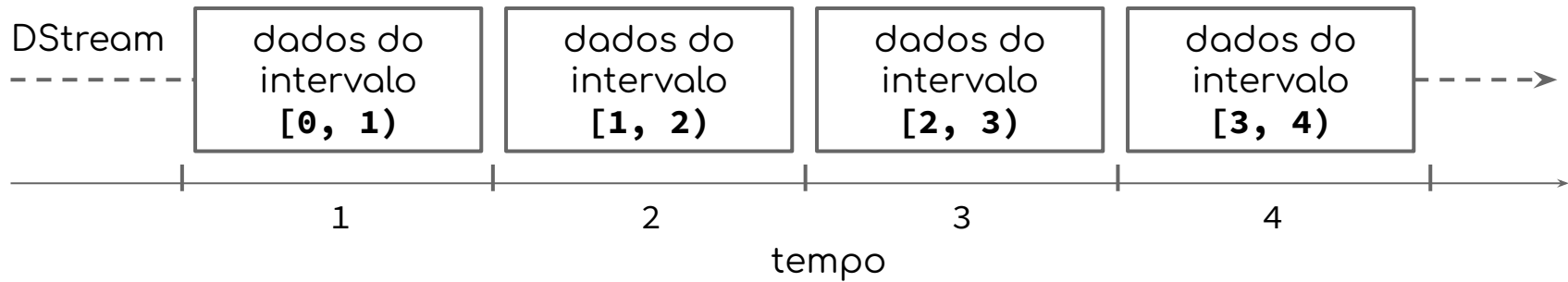
Cada DStream é representado como uma **sequência de RDDs**.

Cada RDD faz referência aos dados obtidos em um intervalo de tempo (*batch interval*).

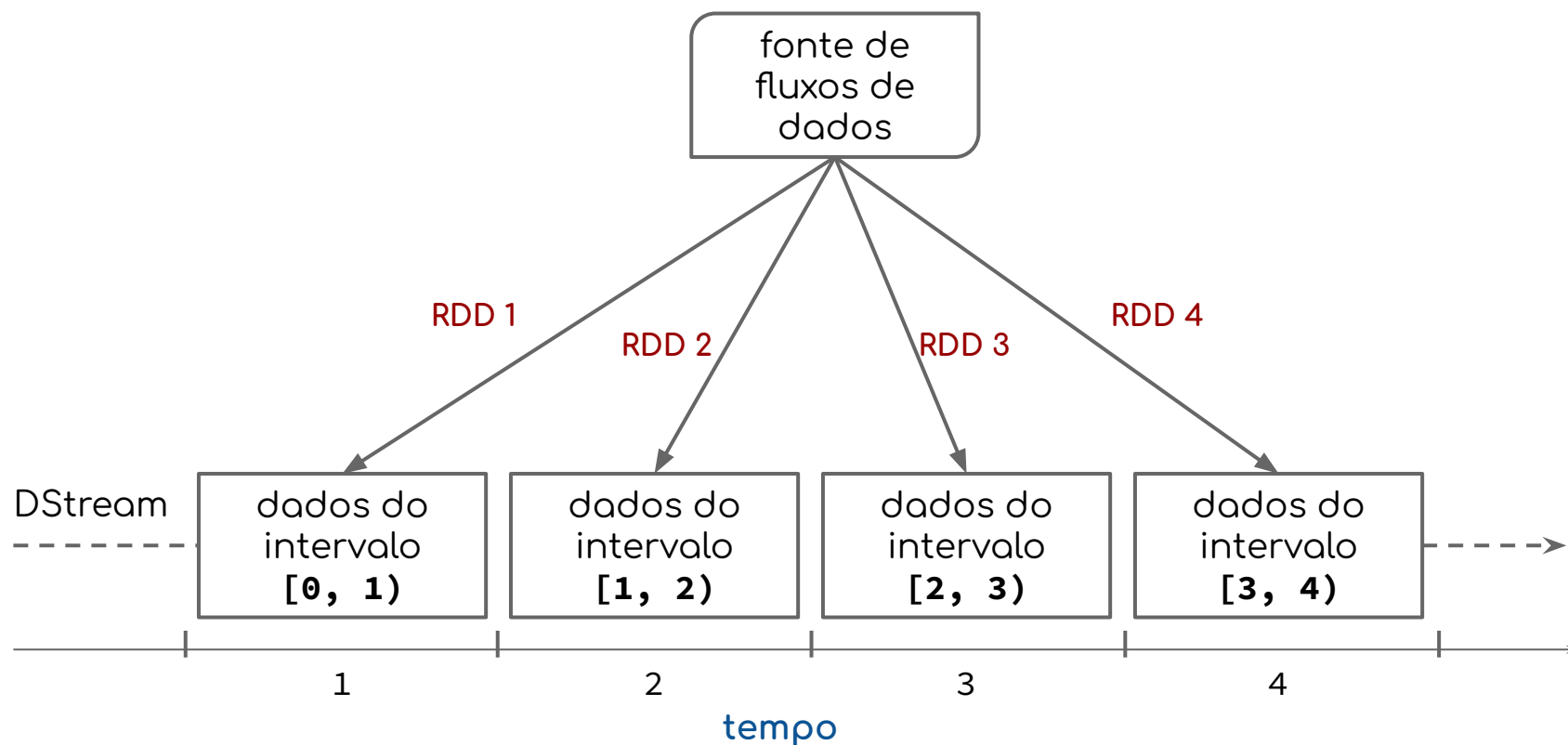
- ⇒ Essa estratégia é denominada *micro-batch*.

Micro-batch

Do ponto de vista visual, fica fácil imaginar um DStream sendo construído na linha do tempo:



Considerando a fonte de dados e RDDs ...



Cada RDD representa um *micro-batch* na linha do tempo.

- ⇒ Agora, operações sobre os fluxos de dados podem ser feitas com operações comuns sobre RDDs !
- ⇒ E isso já conhecemos bem ...

Tipos de transformações

Existem dois tipos de transformações que podem ser aplicadas sobre DStreams:

- ⇒ **sem estado**: são transformações aplicadas a cada lote de dados (RDD do intervalo).
- ⇒ **com estado**: são transformações que agregam de alguma forma informações de diferentes lotes de dados.
 - isto é, diferentes RDDs de intervalos diferentes.

Transformações que ignoram estado são equivalentes ao que já vínhamos fazendo com RDDs comuns.

Transformações que mantêm estado utilizam um conceito novo em fluxos de dados: janela de processamento.

- ⇒ assim, podemos atualizar estado considerando vários intervalos de dados (vários RDDs)

StreamingContext

Aplicações que lidam com fluxos de dados não tem terminação determinada:

- ⇒ isso quer dizer que não existe o "fim" da execução.
 - a terminação é feita explicitamente pelo administrador do sistema.
- ⇒ outra consequência disso é que não faz sentido criarmos uma aplicação de SparkStreaming pelo `spark-shell`.

- ⇒ O SparkStreaming também possui um contexto a ser inicializado, assim como:
 - Spark core possui o `SparkContext`.
 - SparkSQL possui o `SQLContext`.

O contexto do SparkStreaming é o **StreamingContext**.

No início de cada aplicação, ele deve ser configurado.

StreamingContext

Para criar um StreamingContext, basta passar para o seu construtor o nome de uma configuração

ou seja, onde configuramos parâmetros específicos
nome da aplicação, master, recursos (módulo IV).

e o tamanho de um *micro-batch*.

```
val conf = new SparkConf().setAppName ("Streaming App")  
val ssc = new StreamingContext (conf, Seconds (1))  
// existe Minutes e Milliseconds também
```

neste caso, **a cada um segundo** os dados do fluxo de dados serão agregados e transformados em um RDD do *DStream*.

Então, inicializamos a aplicação ...

```
ssc.start
```

e dizemos para a aplicação esperar pela terminação explícita (ctrl-c)

```
ssc.awaitTermination
```

isso é necessário para o sistema terminar da forma adequada e liberar os recursos alocados.

Template de uma aplicação

```
/* StreamingApp.scala */
import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds

object StreamingApp {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Streaming App")
    val ssc = new StreamingContext (conf, Seconds (1))

    // lógica da aplicação:
    // 1. configurar uma fonte de dados (fluxos)
    // 2. transformar esses fluxos
    // 3. exportar os resultados para ambientes externos
    // (arquivo, outras apps, etc)

    // inicializa a aplicação e espera ela terminar
    // lembre-se que a terminação é explícita: alguém termina ela (Ctrl+C)

    ssc.start
    ssc.awaitTermination
  }
}
```

Um cenário possível para a saída

```
$ ./spark/bin/spark-submit  
  --class my.spark.app.WordCountStreaming  
  my-spark-app/app/build/libs/app.jar
```

```
...  
Time: 1436370446000 ms  
  
Time: 1436370447000 ms  
(linha,1) (esta,1) (uma,1) (é,1)
```

```
Time: 1436370448000 ms  
...
```

```
Time: 1436370466000 ms  
(intervalo,1) (outra,1) (linha,,1)  
(outro,1) ((batch),1)  
...
```

```
Time: 1436370489000 ms  
(terminar,1) (para,1) (agora,,1)  
(nos,1) (dois,1) (Ctrl+c,1)  
...
```

```
Time: 1436370490000 ms  
...  
^C
```

```
$ nc -l -p 5432
```

esta é uma linha
outra linha, outro intervalo (batch)
agora, para terminar Ctrl+c nos dois
^C

Observações:

- a cada segundo, todo o processamento sobre as linhas acumuladas foi feito.
- O item acima só acontece se existir linhas para serem consumidas, naturalmente.
- a terminação foi explícita (**Ctrl + C**).

Fluxo de dados com arquivos

Depois de instanciar o `StreamingContext`, o próximo passo é criar `DStreams` e configurar as transformações sobre as mesmas.

⇒ Só então iniciaremos o processamento através da chamada `StreamingContext.start`

Uma forma de criar um `DStream` é a partir da leitura de arquivos locais, no formato texto e linha a linha.

⇒ Em especial, processamento de logs segue essa lógica:

- diferentes aplicações geram logs e os depositam em um repositório comum.
- é interessante inclusive para associar eventos em diferentes sistemas.
- esse é um problema conhecido de agregação de logs e associação de eventos.

O **SparkStreaming** suporta esse tipo de processamento. A ideia principal é:

⇒ o ambiente irá monitorar um diretório (local ou remoto).
⇒ novos arquivos adicionados nesse diretório serão utilizados como fonte de dados A criação de um fluxo de dados com essas características é:

```
val fStream = ssc.textFileStream (<diretório>)
```

onde <diretório> pode ser um caminho local (`file://`) ou remoto (`hdfs://`)

Transformações sem estado

As mesmas de RDDs comuns:

- ⇒ `map`
- ⇒ `flatMap`
- ⇒ `filter`
- ⇒ `reduceByKey`

Propriedade: não mantêm nenhum tipo de estado entre diferentes intervalos.

Contando as categorias de logs por intervalo

Problema: encontrar a categoria de log mais frequente a cada intervalo de 5 segundos.

Criamos o DStream a partir do StreamingContext e contabilizamos um para as linhas que contém uma das três categorias dos logs.

(warn, info, error)

```
val logData = ssc.textFileStream(inputDir)
    .flatMap(line => {
        if (line.contains("INFO")) List(("info", 1))
        else if (line.contains("WARN")) List(("warn", 1))
        else if (line.contains("ERROR")) List(("error", 1))
        else List.empty
    })
```

Reduzimos por chave e imprimimos o resultado no fluxo de saída com a função print.

```
val counts = logData.reduceByKey (_ + _)
counts.print
```

Aplicação completa

```
package my.spark.app
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
object LogPorIntervalo {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Log por janela")
    val ssc = new StreamingContext (conf, Seconds (1))
    val inputDir = args(0)
    val logData = ssc.textFileStream(inputDir)
    .flatMap(line => {
      if (line.contains("INFO")) List(("info", 1))
      else if (line.contains("WARN")) List(("warn", 1))
      else if (line.contains("ERROR")) List(("error", 1))
      else List.empty
    })

    val counts = logData.reduceByKey(_ + _)
    counts.print()
    ssc.start
    ssc.awaitTermination
  }
}
```

Submetendo a aplicação

Para submeter a aplicação, iremos usar o script que simula um fluxo de arquivos em um diretório.

⇒ Note que aplicação informa exatamente isso:

- ler de um fluxo do tipo arquivo de texto (`textFileStream`)
- e o diretório fonte foi informado (`data-stream/`)

Considerando o código em um arquivo de nome `LogPorIntervalo.scala` no projeto offline (recompilação é necessária):

```
$ cd ~/spark
$ $ ./spark/bin/spark-submit --class my.spark.app.LogPorIntervalo
  my-spark-app/app/build/libs/app.jar logfiles
```

Em paralelo (em outro bash) inicie o script que simula o fluxo de arquivos no diretório `data-stream/`

```
$ ./bin/fake-stream.sh data/exemplo-menor-shuf.log 1000 5 logfiles
```


Resultado

Depois de um tempo, você deve observar um resultado como esse:

...

```
-----  
Time: 1436414850000 ms  
-----
```

```
(warn,8)  
(info,468)  
(error,17)
```

...

Note que o resultado só aparece na saída porque no final chamamos uma **operação de saída sobre o fluxo**: `print`.

Transformações com estado

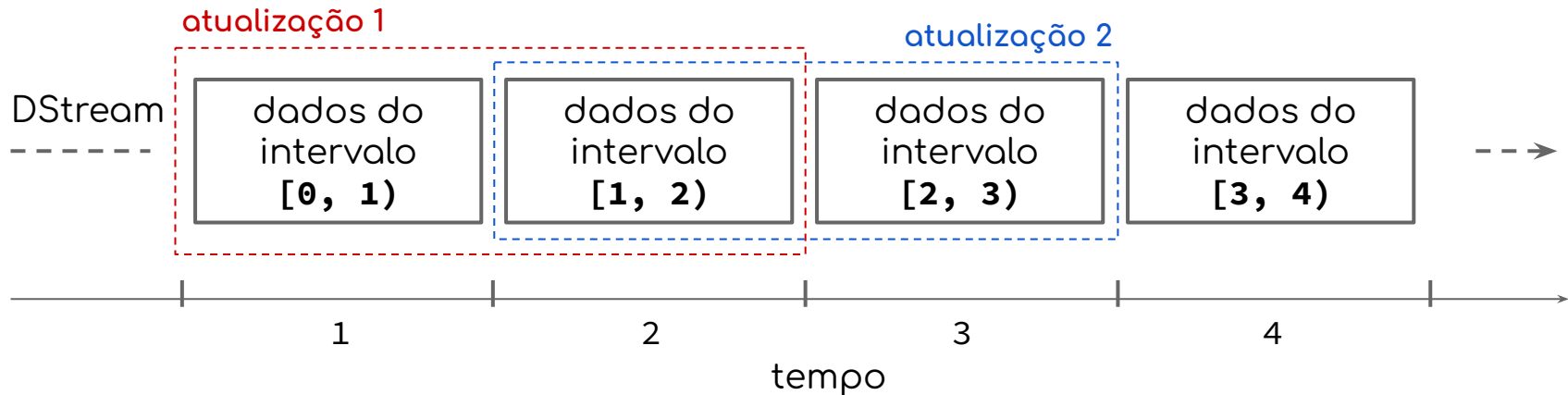
Até então, as operações estavam sendo aplicada a cada intervalo de tempo isoladamente.

⇒ Essas novas transformações agregam informações de vários intervalos de tempo. Por isso, são transformações que mantêm estado. Veremos a principal:

➤ `reduceByKeyAndWindow`

Antes de estudarmos alguns exemplos, vamos entender um conceito importante na agregação de dados entre vários intervalos: ***janela de processamento***.

Janela de processamento



- ⇒ Então, cada RDD representa um lote no intervalo.
- ⇒ Operações que não mantêm estado são aplicadas a cada lote de forma isolada.
- ⇒ Uma janela de processamento é definida por:
 - **tamanho**: número de lotes a serem considerados conjuntamente.
 - na API isso é chamado **windowLength**
 - **passo de avanço**: representa quantos lotes a janela avança a cada atualização.
 - na API isso é chamado **slideInterval**
 - No exemplo acima, a janela possui **tamanho 2** e **passo de avanço 1**.

Contando as categorias de logs recentes

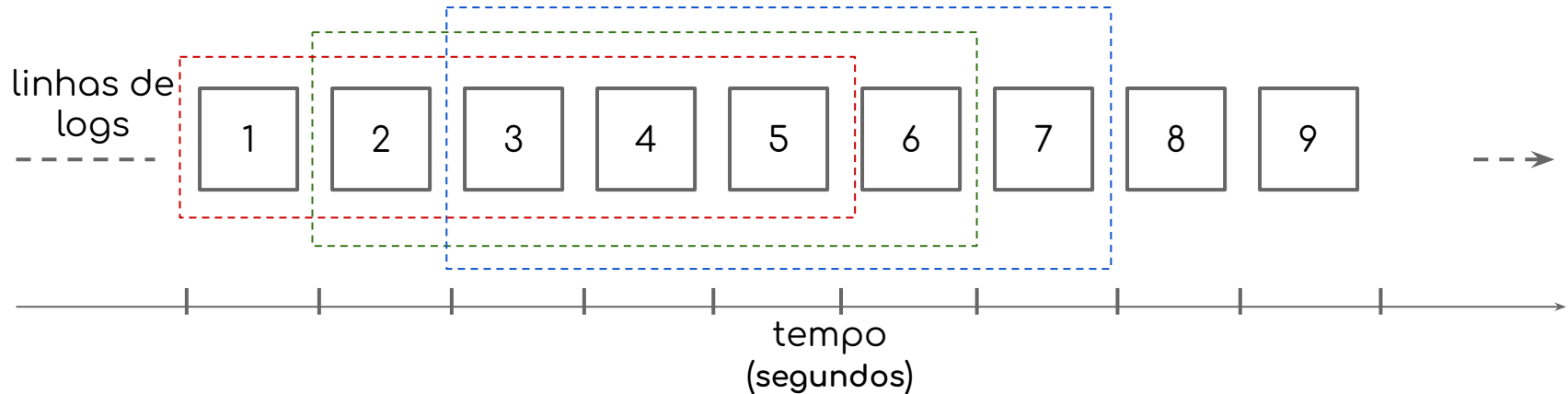
Agora temos o seguinte problema: gostaríamos de monitorar os logs das aplicações, porém mantendo apenas as contagens de categorias de logs recentes.

- ⇒ Ou seja, a partir de certo ponto, podemos começar a ignorar contagens de logs consideradas "antigas".
- ⇒ A definição de um intervalo adequado fica a cargo do analista.

Neste exemplo, para efeito de demonstração, iremos:

- processar logs a cada 1 segundo:
 - `batchInterval = Seconds(1)`
- desconsiderar logs que chegaram há mais de 5 segundos:
 - `windowLength = Seconds(5)`
 - `slideInterval = Seconds(1)`

O que faremos, visualmente ...



- ⇒ Vermelho para a janela da quinta atualização.
- ⇒ Verde para a janela da sexta atualização.
- ⇒ Azul para a janela da sétima atualização.

Implementação (parte 1)

A lógica da implementação é praticamente a mesma do problema último problema:

- ⇒ antes, estávamos considerando as contagens isoladas para cada intervalo de tempo (`batch interval`).
- ⇒ não tínhamos nenhuma relação entre intervalos.
- ⇒ essa relação será introduzida a partir de uma versão do `reduceByKey` para janelas:

```
val ssc = new StreamingContext (conf, Seconds (1))
```

```
val logData = ssc.textFileStream (inputDir)
    .flatMap(line => {
        if (line.contains("INFO")) List(("info", 1))
        else if (line.contains("WARN")) List(("warn", 1))
        else if (line.contains("ERROR")) List(("error", 1))
        else List.empty
    })
```

```
val counts = logData.reduceByKeyAndWindow((a:Int, b:Int) => a+b,
    Seconds(5), Seconds(1))
```

Isto é, as mensagens de log serão processadas em janelas de 5 segundos (`windowLength`).

A cada 1 segundo, a janela será deslizada e um novo processamento realizado (`slideInterval`).

Implementação (parte 2)

```
package my.spark.app
import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds
object LogPorJanela {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Log por janela")
    val ssc = new StreamingContext(conf, Seconds (1))
    val inputDir = args(0)
    val logData = ssc.textFileStream (inputDir)
    .flatMap(line => {
      if (line.contains("INFO")) List(("info", 1))
      else if (line.contains("WARN")) List(("warn", 1))
      else if (line.contains("ERROR")) List(("error", 1))
      else List.empty
    })
    val counts = logData.reduceByKeyAndWindow((a:Int,b:Int) => a+b, Seconds(5), Seconds(1))
    counts.print
    ssc.start
    ssc.awaitTermination
  }
}
```

Executando a aplicação

A execução segue o mesmo processo do anterior:

1. Adicionar a implementação no arquivo `LogPorJanela.scala` no diretório de códigos no projeto **my-spark-app**.
2. Recompilar as aplicações com `./bin/build package`
3. Submeter a aplicação pelo script `./bin/spark-submit`.

```
$ $ ./spark/bin/spark-submit --class my.spark.app.LogPorJanela  
my-spark-app/app/build/libs/app.jar logfiles/
```

Em paralelo, geramos o fluxo de arquivos com o script
`fake-file-stream.sh`

```
$ ./fake-file-stream.sh data/exemplo-menor-shuf.log 1000 5 logfiles
```

... para gerar arquivos de pedaços de 1000 linhas de 1 em 1 segundo.

Observando o resultado

Você deve observar um resultado similar a isso:

...

Time: 1436440016000 ms

(warn,180)
(info,3820)

Time: 1436440017000 ms

(warn,265)
(info,3734)

Time: 1436440018000 ms

(warn,265)
(info,3734)

Time: 1436440019000 ms

(warn,357)
(info,3642)

...

note que como se
tratavam de lotes em
uma mesma janela, o
processamento foi
acumulado ao invés de
zerado.

aqui o resultado continuou o
mesmo pois nenhuma nova
informação chegou

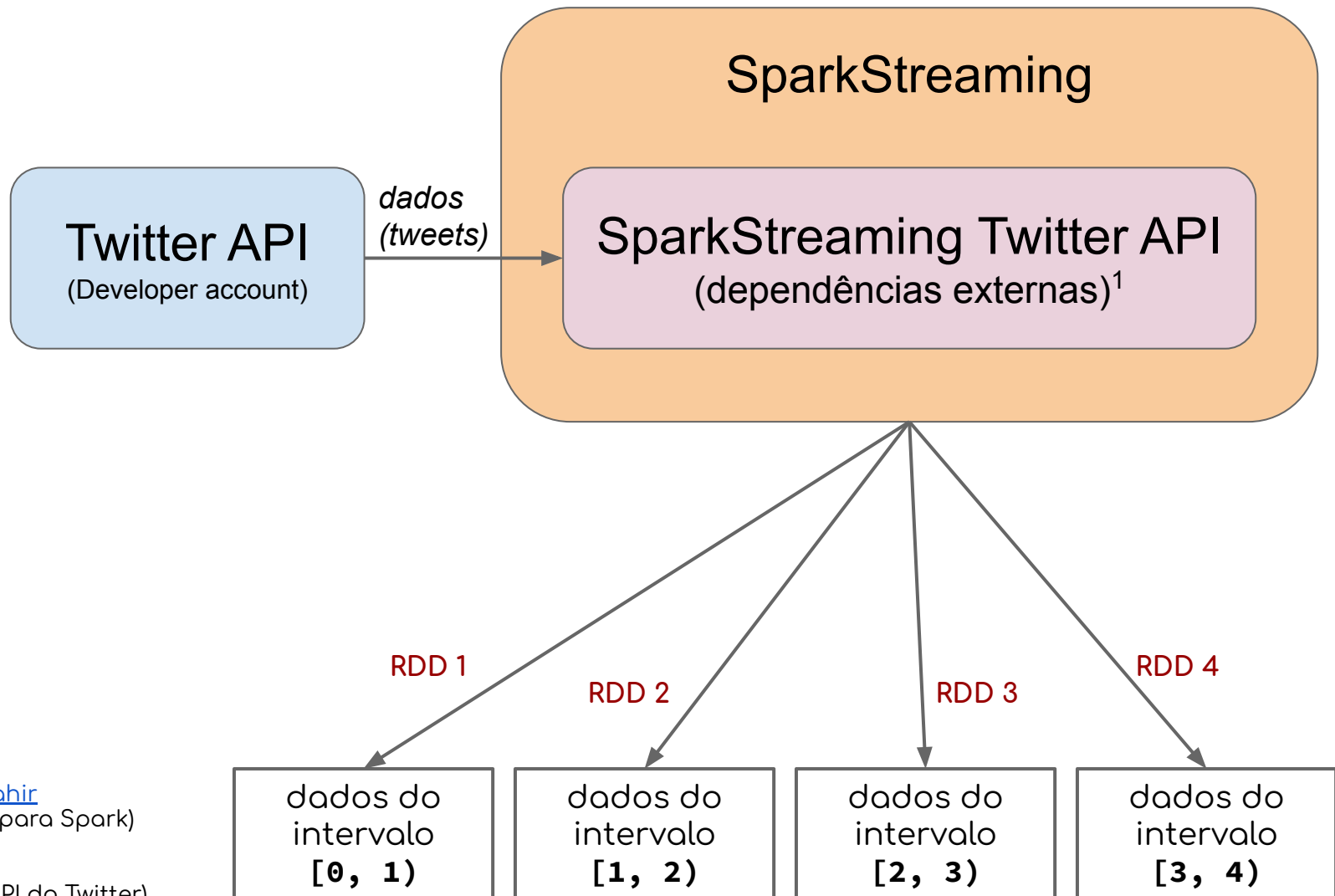
aqui chegou uma nova
informação na mesma janela
O resultado foi acumulado

Consumindo fluxo de dados de redes sociais (Twitter)

- ⇒ Fluxos contínuos de dados também podem vir de redes sociais
 - Faz sentido, já que são volumes massivos de dados sendo produzidos continuamente e que carecem de tratamento imediato.
- ⇒ Vamos ver um exemplo típico de integração do processamento de fluxos usando SparkStreaming com dados provenientes do Twitter



O que vamos fazer ...



¹ [Apache Bahir](#)
(extensões para Spark)

¹ [Twitter4j](#)
(acesso à API do Twitter)

Implementação

⇒ HashTags mais populares em tweets de contas verificadas

```
val ssc = new StreamingContext(sparkConf, Seconds(1))  
val stream = TwitterUtils.createFilteredStream(ssc, None, Some(locationQuery))
```


```
val hashTagsMaisPopulares = stream  
    .filter(status => status.getUser.isVerified)  
    .flatMap(status =>  
        status.getText.split(" ").filter(_.startsWith("#")))  
    .map(hashtag => (hashtag,1))  
    .reduceByKeyAndWindow(_ + _, Seconds(60 * 60))  
    .map(_.swap)  
    .transform(_.sortByKey(false))
```

```
hashTagsMaisPopulares.print(3)
```

Execução

```
$ ./spark/bin/spark-submit \  
  --class my.spark.app.TwitterStreamingApp \  
  my-spark-app/app/build/libs/app.jar \  
  <consumer-key> <consumer-secret> \  
  <access-token> <access-token-secret>
```

fornecido
como forma de
autenticação
junto à API do
Twitter



```
-----  
Time: 1661375358000 ms  
-----
```

```
(2,#hashtag1)  
(1,#hashtag2)  
(1,#hashtag3)  
...
```

```
-----  
Time: 1661375359000 ms  
...
```

Considerações sobre desempenho

Todas as considerações de desempenho discutidas no Spark core continuam valendo nesse contexto:

- ⇒ paralelismo por partições.
- ⇒ escolha da quantidade ideal de recursos do cluster para cada tarefa.
- ⇒ escolha das transformações mais eficientes para cada tarefa.

O que há de novo?

- ⇒ tenha em mente que persistência de DStreams significa persistir cada um dos RDDs dos lotes do fluxo de dados.
- ⇒ além disso é importante a escolha de parâmetros de duração

`batchInterval`, `windowLength` e `slidingInterval`

de acordo com os requisitos do fluxo sendo lido. Ou seja:

- ⇒ a taxa que novos dados são gerados.
- ⇒ a taxa que eles devem ser processados.

O que mais?

⇒ Spark é uma plataforma em constante atualização e muitas funcionalidades não foram abordadas aqui. Por exemplo:

- Abstrações que possibilitam a combinação de diferentes módulos do ambiente
- Conectores e extensões para integração de Spark com diversos outros sistemas: HDFS, Redis, PostgreSQL, Cassandra, MongoDB, etc.

Para os cientistas de dados ...

ML algorithms include:

- Classification: logistic regression, naive Bayes,...
- Regression: generalized linear regression, survival regression,...
- Decision trees, random forests, and gradient-boosted trees
- Recommendation: alternating least squares (ALS)
- Clustering: K-means, Gaussian mixtures (GMMs),...
- Topic modeling: latent Dirichlet allocation (LDA)
- Frequent itemsets, association rules, and sequential pattern mining



ML workflow utilities include:

- Feature transformations: standardization, normalization, hashing,...
- ML Pipeline construction
- Model evaluation and hyper-parameter tuning
- ML persistence: saving and loading models and Pipelines

Other utilities include:

- Distributed linear algebra: SVD, PCA,...
- Statistics: summary statistics, hypothesis testing,...

Processamento de dados massivos com Spark: conceitos, desafios e programação

Vinícius Dias

viniciusvdias@ufop.edu.br

Departamento de Computação e Sistemas

DECSI / UFOP

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)
- ⇒ [Spark By Examples](#) (exemplos práticos de configuração e uso da ferramenta)