Parte 4.4: Query DSL

Visão geral de DSL

- Método poderoso e comum de expressão de transformações
- O DSL consiste em:
  - A classe **`Column`**: Representa uma coluna
  - Métodos do `DataFrame` (e.g. `agg, groupBy, select`)
    - E alguns métodos declarados (Datasets) escritos em termos da classe Column
  - Funções auxiliares, e.g. `spark.implicits._, spark.functions`
  - Abaixo, a expressão DSL é: **`$"age">25`**

```
folksDF.filter($"age">25)
```

- Pode-se também obter transformações por meio de **Expressões SQL**
  - Suporta SQL comum
  - Abaixo, a expressão SQL é: **`"age>25”`**

```
folksDF.filter("age>25")
```

Introdução a Classe Column

- **`Column`**: Principal classe do DSL, representa uma coluna de um DataFrame
  - Geralmente é processada a partir dos dados de um DataFrame
  - pacote: **`org.apache.spark.sql`**

- Muitas operações de `DataFrame` recebem objetos `Column` como parâmetro
  - e.g. `filter()`:
  - `filter(condition: Column): Dataset[T]`
  - `DataFrame.col()` permite acessar as informações de uma coluna específica
  - **`col(colName: String):Column`**
  - `Column` suporta muitos operadores de expressão
    - e.g. < (MaiorQue), > (MenorQue), **`===`** (equalidade), **`=!=`** (inequalidade), etc. [2]
    - Expressões são passadas para transformações em `DataFrame`

Exemplo: Usando o DSL

- Acesse uma coluna de um DataFrame usando a função `col()`
  - **`col(colName: String):Column`**
  - Use operadores de classe Column para criar expressões
  - Abaixo, criamos uma instância de Column especificando ”todas as linhas cujo valor da coluna age > 25”

```
folksDF.filter(folksDF.col("age") > 25)
```

- O Scala permite outras variações do código com o mesmo resultado

```
folksDF.filter(folksDF("age") > 25)
folksDF.filter($"age" > 25)
folksDF.filter('age > 25)
```

Examinando uma Transformação

```
> folksDF.filter('age>25).groupBy('gender).avg("age").show
+------+--------+
|gender|avg(age)|
+------+--------+
|     F|    46.0|
|     M|    35.0|
+------+--------+
```

- **`filter('age>25)`** obtem as linhas com "age">25
  - `'age` é um símbolo para o Scala — convertido para um objeto `Column` por conversões implícitas
  - `filter()` retorna um `DataFrame`
- **`groupBy('gender)`**: Agrupa os valores a partir da coluna "gender"
  - `groupBy()` é usado para agregação
  - `groupBy(cols: Column*): RelationalGroupedDataset`
  - `RelationalGroupedDataset` define métodos de agregação
- **`avg("age")`**: Obtem a média do campo "age" para os grupos
  - Definido no `RelationalGroupedDataset` com assinatura
  - **`avg(colNames: String*): DataFrame`**

Trabalhando com o Query DSL

- Vamos nos aprofundas em vários exemplos para ter uma idéia de como as DSLs funcionam e quais são as suas habilidades
- Vamos ver tanto transformações tipadas quanto as não tipadas
  - O comum entre elas é o uso de DSL (e.g. `Column` instances)

Exemplo: Dataset.orderBy()

- Transformação tipada — duas versões
  - **`orderBy(sortExprs: Column*): Dataset[T]`**
  - **`orderBy(sortCol: String, sortCols: String*): Dataset[T]`**
  - Ambas são sinônimos de `sort()`

```
// Ordenar por “age” em ordem decrescente

> folksDF.orderBy("age").show // O mesmo que folksDF.sort("age")
+---+------+----+
|age|gender|name|
+---+------+----+
| 20|     M|Mike|
| 35|     M|John|

// Ordenar por “age” em ordem decrescente
// 'age é um símbolo do Scala – convertido para Column via conversões
// implícitas
// desc é um método da classe Column

> folksDF.orderBy('age.desc).show  // Resultados em ordem decrescente
```

Exemplo: RelationalGroupedDataset.agg()

- `groupBy` retorna um objeto `RelationalGroupedDataset`
  - Após agrupar, use os métodos do objeto `RelationalGroupedDataset`
  - Abaixo, exemplo de utilização do método `agg()`
- `agg(exprs: Map[String, String]): DataFrame`
  - Especifique um mapa contendo nome da coluna e o método de agregação
  - Métodos de agregação disponíveis: `avg, max, min, sum, count`

```
// Usando o método RelationalGroupedDataset.agg()
// Nome da coluna/Nome do método de agregação
> folksDF.filter('age>25).groupBy('gender).agg(
     |   "age" -> "avg",
     |   "age" -> "max",
     |   "gender" -> "count").show
+------+--------+--------+-------------+
|gender|avg(age)|max(age)|count(gender)|
+------+--------+--------+-------------+
|     F|    46.0|      52|            2|
|     M|    35.0|      35|            1|
+------+--------+--------+-------------+
```

Exemplo: Dataset.agg()

- `agg(expr: Column, exprs: Column*): DataFrame`
  - Agregar em todo o DataFrame
  - Abaixo, as funções `min, max, e avg` fazem parte do objeto `org.apache.spark.sql.functions`
  - As funções estão disponíveis para todas as operações de DataFrame

```
> folksDF.agg(max('age)).show
+--------+
|max(age)|
+--------+
|      52|
+--------+

> folksDF.agg(min('age), max('age), avg('age)).show
+--------+--------+--------+
|min(age)|max(age)|avg(age)|
+--------+--------+--------+
|      20|      52|   36.75|
+--------+--------+--------+
```

Exemplo: Operadores de Column

```
// && - "and" (age entre 25, 50)
> folksDF.filter('age>25 && 'age<50).show

> folksDF.filter('age === 35) // Age é igual a 35 - Equalidade é expressada
                              // com 3 sinais de igual
> folksDF.filter('age.equalTo(35)) // Equalidade usando o equalTo

> folksDF.filter('age =!= 35) // Age não é igual a 35 - Inequalidade é
                              // expressada com =!=


> folksDF.agg(avg('age).alias("avgAge")).show  // Definindo outro nome para a
                                               // coluna
+------+
|avgAge|
+------+
| 36.75|
+------+

> folksDF.filter('name.startsWith("J")) // Filtro: Nomes que começam com J

> folksDF.filter('name.like("%e"))      // Filtro: Nomes que terminam com E
```

Exemplo: Select

- Abaixo, usamos os seletores do objeto `Column` para criar uma visualização
  - O resultado é um DataFrame com duas colunas (gender, age)

```
> val genderAgeDF = folksDF.select('gender,'age)
genderAgeDF: org.apache.spark.sql.DataFrame =
                                                [gender: string, age: bigint]

> genderAgeDF.show
+------+---+
|gender|age|
+------+---+
|     M| 35|
|     F| 40|
|     M| 20|
|     F| 52|
+------+---+
```

Exemplo: Literals

- Literals suportam operações em colunas usando operadores simples
- `lit()` transforma um valor literal em uma instancia do objeto Column
  - Isso pode também ser usado de forma simples com DSL

```
> folksDF.select('age*lit(2), 'name, 'gender).show
+---------+----+------+
|(age * 2)|name|gender|
+---------+----+------+
|       70|John|     M|
|       80|Jane|     F|
|       40|Mike|     M|
|      104| Sue|     F|
+---------+----+------+
```

Dados mais Complexos

- Suponha que os seus dados de pessoas (people.json) tenham dentro do JSON um elemento com um valor "aninhado" — e.g.

```
{"name": "John", "gender": "M", "age": 35,
 "address":{"street":"123 Aspen St.", "city":"Buffalo",
"state":"NY","zip":"14205"} }
```

  - Acesse elementos aninhados usando o operador . (ponto), exemplo:

```
> val folksAddressDF = spark.read.json("data/people-with-address.json")
> folksAddressDF.select($"address.city").show
+-------------+
|         city|
+-------------+
|      Buffalo|

> folksAddressDF.select('address.city).show // Isso não vai functionar

// Acessar a coluna completa. Use show(false) para evitar truncamento
> folksAddressDF.select($"address").show(false)
+-----------------------------------+
|address                            |
+-----------------------------------+
|[Buffalo,NY,123 Aspen St.,14205]   |
```

UDF: User Defined Function

- UDFs definem novas funções baseadas no objeto `Column`

- Para definir e catalogar uma UDF:
  - Defina uma função regular
  - Use o método **`udf()`** para catalogar (**`org.apache.spark.sql.functions`**)
  - Utilize a função como DSL

```
// Definindo uma função regular para validar o campo “age"
> val young : (Int) => Boolean = (age) => (age<45)
> import org.apache.spark.sql.functions.udf
> val youngUDF = udf(young)  // Catalogando como uma UDF
youngUDF: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function1>,BooleanType,Some(List(IntegerType)))
// Usando a função
> folksDF.filter(youngUDF('age)).show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
| 20|     M|Mike|
+---+------+----+
```

Exemplo de uma UDF mais complexa

- Abaixo, criamos uma UDF que recebe dois parâmetros
  - Desempenha quase a mesma função do exemplo anterior.

```
// Definindo uma função regular para validar os campos “age" e “gender”
> val youngMale : (Int, String) => Boolean = (age, gender) =>
                                        (age<45 && gender=="M")
youngMale: (Int, String) => Boolean = <function2>

// Catalogando a função como uma UDF
> import org.apache.spark.sql.functions.udf
> val youngMaleUDF = udf(youngMale)
youngMaleUDF: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function2>,BooleanType,Some(List(IntegerType,
StringType)))

// Use it
> folksDF.filter(youngMaleUDF('age, 'gender)).show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 20|     M|Mike|
+---+------+----+
```

Introdução ao objeto Row

- **Row**: Objeto que representa uma linha de um `DataFrame`
  - Lembre-se, `DataFrame == DataSet[Row]`
  - Geralmente não lidamos com objetos Row diretamente
  - pacote: **`org.apache.spark.sql`**

- Acessando valores do objeto Row
  - **`apply(i: Int): Any`** — Acesso genérico ordinal
  - **`getXXX(i: Int): XXX`** — Acesso de tipos primitivos
    - `getInt(i: Int): Int, getString(i: Int): String, etc.`

```
> folksDF.map(r => (r.getLong(0), r.getString(1), r.getString(2))).show
+---+---+----+
| _1| _2|  _3|
+---+---+----+
| 35|  M|John|

> folksDF.rdd.map(r => (r(0), r(1), r(2))).collect
res14: Array[(Any, Any, Any)] = Array((35,M,John), ,,,)
```

Extraindo dados de Row para dentro de uma Classe

- É relativamente simples extrair dados para dentro de um classe
  - Geralmente é usado uma classe case

```
scala> case class Person (age: Long, gender: String, name: String)
defined class Person

scala> folksDF.map( r => Person(r.getLong(0), r.getString(1),
r.getString(2))).show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|

// Uma alternativa é usar o pattern match – obtem-se o mesmo resultado.
> folksDF.map{
  case Row (age:Long, gender:String, name:String) => Person(age, gender,
name)
}.show
```

Por quê usar o DSL

- Interface fluida, simples para utilizar e entender
  - E geralmente se desempenha melhor

- Observe o exemplo abaixo
  - Uma representação do mesmo exemplo usando RDD ficara assim:

```
folksRDD.filter(p=>p.age>25)
        .map(p=>(p.gender, p.age)).collect
```

  - Qual é mais simples de entender?

```
folksDF.filter('age>25).select('gender, 'age).show
+------+---+
|gender|age|
+------+---+
|     M| 35|
|     F| 40|
|     F| 52|
+------+---+
```

Usando SQL

- `createOrReplaceTempView()`: Registra um nome de tabela
  - i.e. Portanto, isso fica catalogado no SQL Parser
  - Atrelada ao o SparkSession, é destruída quando a sessão é finalizada

- `SparkSession.sql()`: Executa uma consulta SQL usando o Spark
  - Suporta o SQL 2003
  - `sql(sqlText: String): DataFrame`

```
folksDF.createOrReplaceTempView("people")    // Configurando a tabela

// Obter a média por grupos da coluna “gender”
spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
+------+--------+
|gender|avg(age)|
+------+--------+
|     F|    46.0|
|     M|    27.5|
+------+--------+
```

SQL vs Consultas DSL em DataFrame

- Desempenham a mesma execução
  - Depois de analisados, são processados pela mesma interface
  - O Catalyst Analisa e otimiza a consulta de ambas as formas
  - A performance é a mesma
- Escolha o melhor para você
  - Se prefere SQL, então essa deve ser a sua escolha
  - Se prefere códigos, então consultas DSL devem te atender melhor

MINI-LAB — Reveja a Documentação

- Acesse a documentação do Spark em:
  http://spark.apache.org/docs/latest/
  - Busque por **`org.apache.spark.sqlas`**
- Rejeva a **classe `Dataset`**, e faça o seguinte:
  - Reveja os métodos que usamos e como a classe `Column` é usada
  - Observe que existem várias versões de muitos métodos (para flexibilidade)
- Reveja brevemente a **classe `Column`**
  - Visite a sessão de operadores
- Reveja brevemente a **classe `RelationalGroupedDataset`**
  - Visite a sessão de operadores
- Reveja brevemente o pacote `org.apache.spark.sql.`**`functions`**

Lab 4.3: Transformações em DataFrames

Parte 4.4: Datasets

Criando um Datasets[T]

- Geralmente criados a partir de DataFrames usando a função **`as[T]`**

  `as[U](implicit arg0: Encoder[U]): Dataset[U]`

  - Pelo fato do `SparkSession` geralmente carregar os dados em `DataFrames`

```
// Assumindo que folksDF foi criado anteriormente

case class Person (name: String, gender: String, age: Long)

val folksDS=folksDF.as[Person]
folksDS: org.apache.spark.sql.Dataset[Person] = [age: bigint, gender: string
... 1 more field]

folksDS.show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
```

A API Dataset : Transformações tipadas

- Possue muitas transformações
  - A maioria utiliza lambdas — parecido com as transformações em RDDs
- `filter(func: (T) ⇒ Boolean): Dataset[T]:`
  - Filtro baseado em uma função
- `flatMap[U](func: (T) ⇒ TraversableOnce[U]): Dataset[U]`
  - Parecido com a versão para RDDs
- `groupByKey[K](func:(T) ⇒ K): KeyValueGroupedDataset[K, T]`
  - Agrupar por uma chave (do tipo K) retornada por uma função
- `select[U1](c1: TypedColumn[T, U1]): Dataset[U1]`
  - Retorna um novo Dataset composto pelas colunas indicadas

Transformações com Datasets

- Algumas transformações retornam valores do mesmo tipo
  - e.g. **`filter()`** que possue algumas variações, como por exemplo:

```
filter(func: (T) ⇒ Boolean): Dataset[T]
filter(conditionExpr: String): Dataset[T]
filter(condition: Column): Dataset[T]
```

```
// Assumindo que folksDS foi criado anteriormente
// As funções de filtro geram o mesmo resultado
// Elas expressam a mesma condição de formas diferentes

folksDS.filter(p => p.age>25)    // lambda
folksDS.filter("age > 25").show  // SQL
folksDS.filter($"age" > 25)      // DSL
folksDS.filter('age > 25)         // DSL (com um símbolo do Scala)
```

Exemplo: Transformações com mudança de tipo

- Algumas transformações retornam tipos diferentes do inicial
  - Abaixo, usamos o método `groupByKey()` para agrupar valores pela coluna "gender"

```
groupByKey[K](func: (T) ⇒ K)(implicit arg0: Encoder[K]):
KeyValueGroupedDataset[K, T]
```

    - Os dados são agrupados a partir da função (que converte T para K)

```
// Assumindo que folksDS foi criado anteriormente

> val folksDSGrouped = folksDS.groupByKey(p => p.gender)
folksDSGrouped: org.apache.spark.sql.KeyValueGroupedDataset[String,Person] =
…

> folksDS.groupByKey(T => T.gender).count.show  // Usando o agrupamento
+-----+--------+
|value|count(1)|
+-----+--------+
|    F|       2|
|    M|       2|
+-----+--------+
```

Exemplo: Média dos valores da coluna “Age”

- Abaixo, calculamos a média do campo “age” dos grupos de “gender”
  - Usando o método KeyValueGroupedDataset.agg, que representa:

```
agg[U1](col1: TypedColumn[V, U1]): Dataset[(K, U1)]
```

  - Observe o argumento TypedColumn
    - A forte tipificação do método
  - Criamos o objeto TypedColumn via **Column.as[Double]**

```
as[U](…): TypedColumn[Any, U]
```

```
folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double])
res5: org.apache.spark.sql.Dataset[(String, Double)] = [value: string,
avg(age): double]

folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double]).show
+-----+--------+
|value|avg(age)|
+-----+--------+
|    F|    46.0|
|    M|    27.5|
+-----+--------+
```

Exemplo: Select

- O primeiro exemplo abaixo seleciona colunas (com um tipo explicito)
  - Observe o uso do método **`as[]`** em cada seletor de coluna
    - Isso as converte em um objeto do tipo `TypedColumn`
    - Isso faz com que a versão tipada do objeto seja chamada
    - O resultado é um Dataset de pares (`String`,`Long`)
- O segundo exemplo abaixo, usa seletores comuns do objeto `Column`
  - Isso faz com que um `Select` de colunas sem tipos seja chamado
  - O resultado é um DataFrame

```
folksDS.select('gender.as[String], 'age.as[Long])
res6: org.apache.spark.sql.Dataset[(String, Long)] = [gender: string, age:
bigint]

folksDS.select('gender,'age)
res130: org.apache.spark.sql.DataFrame = [gender: string, age: bigint]
```

De Datasets para DataFrames

- É simples migrar um `Dataset` para um `DataFrame`
  - Apenas utilize o método `toDF()`, ou use qualquer método não tipado
    - Métodos não tipados retornam `DataFrames`

```
// groupBy('gender) é uma operação não tipada (DataFrame)
// Realizando esta operação, os dados passam a fazer parte de um DataFrame

val results = folksDS.filter('age > 25).groupBy('gender).avg("age")
results: org.apache.spark.sql.DataFrame = [gender: string, avg(age): double]

results.show
+------+--------+
|gender|avg(age)|
+------+--------+
|     F|    46.0|
|     M|    35.0|
+------+--------+
```

Confuso?

- Sim, a API é complexa e pode parecer confuso
  - Diferentes formas de fazer uma transformação
    - DSL, SQL, lambdas
  - `DataFrames` e `Datasets`
    - Com um limite bastante fluido entre eles
  - Qual devemos usar?

- Para nos ajudar com isso, vamos compara-los nos seguintes tópicos
  - **Erros**: Quando erros acontecen
  - **Simplicidade de uso**: Quão simples é usar a API
  - **Maturidade e Estabilidade**: Quão completa e estável é a API
  - **Performance**: Escolhas que afetam a performance

Erros

- As APIs capturam erros em **momentos diferentes**
  - A tabela abaixo mostra a diferença
  - **Erro de sintaxe**: e.g. erro ortográfico: "**SLECT**" ou **`df.filler()`**
  - **Erro de análise**: e.g. propriedade desconhecida, e.g
    - SQL: **`"GROUP BY gnder"`**
    - DataFrame: **`df.groupBy("gnder")`**
    - Dataset: **`ds.groupByKey(p=>p.gndr)`**
- `Datasets` capturam mais erros em **tempo de compilação**
  - Isso é bom

Tipo de Erro	SQL	DataFrame	Dataset
Erro de sintaxe	Tempo de execução	Tempo de compilação	Tempo de compilação
Erro de análise	Tempo de execução	Tempo de execução	Tempo de compilação

Simplicidade de uso

- Abaixo, mostramos três formas de transformação
  - A forma usando Dataset é a mais complexa
  - Pode ser ainda mais difícil de compreender em transformações mais complexas

- A segurança de tipo do `Dataset` **leva mais complexidade** para o código

```
// DataFrame
folksDF.filter('age>25).groupBy('gender).avg("age")

// Dataset
folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double])

// SQL
spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
```

Maturidade e Estabilidade

- `DataFrames` e SQL estão em versões bem antigas
  - 1.0/1.3
- `Datasets` foram introduzidos na versão 1.6, e refinados na versão 2.0
  - Muitos dos modelos tipados de API (`Dataset`) estão marcados como **experimentais**
- Isso significa que `Datasets` vão continuar a evoluir
  - Se você o utiliza, então vai ter que evoluir seu código também
  - Também é provável que você encontre métodos que não atendam às suas necessidades

Performance

- `DataFrames` e SQL basicamente performam igual
  - São otimizados e transformados para passarem pelo mesmo processo de execução
- `Datasets` são baseados em lambdas (para lidarem bem com a API tipada)
  - O uso de lambdas pode **afetar a performance**
    - O otimizador Catalyst não pode inferi-las
    - Podem trazer um impacto negativo de performance

Resumo

- Você possue muitas opções
- Consultas com operadores DSL, API tipada do Dataset, e consultas SQL são melhores que RDDs
  - A não ser que você precise fazer algo que é suportado apenas por RDDs
- Consultas com operadores DSL e SQL são equivalentes
  - Use o que te agrada mais
- A API tipada do Dataset permite ter segurança de tipo em tempo de compilação
  - Captura boa parte dos erros em tempo de compilação
  - Mas existem outros atalhos

Lab 4.4: A API Tipada do Dataset

Parte 4.5: Métodos flatMap, explode, split

Divisão de dados — flatMap()

- **Divisão** das linhas recebidas é uma função muito útil
- Existem muitas formas de dividir os dados, com resultados parecidos
  - Vamos ver algumas técnicas comuns.
- `flatMap()` é uma escolha popular

  ```
  flatMap[U](func: (T) ⇒ TraversableOnce[U]): Dataset[U]
  ```

  - `func` cria uma coleção de cada elemento
  - As coleções são combinadas em um resultado
  - Parte da API **Tipada** do `Dataset`

Exemplo do uso do flatMap() em um DataFrame

- Abaixo, vamos dividir as linhas usando o `flatMap`
  - `flatMap` recebe uma função **lambda**
  - Cada elemento representa uma instância do objeto Row
    - Nós acessamos via `Row.getString()`

```
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF("line")

// Dividindo as linhas usando a expressão regular que representa espaços
// Cada linha cria uma coleção de palavras, que são combinadas
> val flatMappedWords =
   linesDF.flatMap(_.getString(0).toLowerCase().split("\\s+"))

// O uso do flatMap abaixo utiliza um atalho para essa mesma lambda
// flatMap(line => line.getString(0).toLowerCase().split("\\s+"))
```

Exemplo do uso de explode/split em DataFrame

- Abaixo, nós dividimos as linhas utilizando DSL
  - `split()` divide os dados em colunas baseando-se em uma expressão regular

```
split(str: Column, pattern: String): Column
```

  - `explode()` cria uma instância de Row para cada elemento em um Array/Map

```
explode(e: Column): Column
```

```
// Dividir linhas pelo método split() usando a mesma expressão regular
// split retorna instâncias do objeto Row contendo um array<string>
// explode retorna instâncias do objeto Row contendo strings

> val splitWordsDF = linesDF.select(explode (split('line, "\\s+")).as("word")
).select(lower('word).as("word"))
```

Exemplo do uso do método flatMap() em Dataset/RDD

- Em Datasets é parecido com DataFrame
  - O Dataset possue Strings simples, não instâncias do objeto Row
- Em RDD é parecido com Dataset (mas o processamento vai ser em um nível mais baixo)
  - explode/split não estão disponíveis para RDD ou Dataset

```
// Dataset: Use nosso DataFrame para fazer isso
> val linesDS = linesDF.as[String]
linesDS: org.apache.spark.sql.Dataset[String] = [line: string]

// Divida a linha usando o flatMap – os elementos são do tipo String agora
> val splitWordsDS = linesDS.flatMap(_.toLowerCase().split("\\s"))
```

```
// RDD: Crie a partir do método parallelize
> val linesRDD = sc.parallelize(Seq("Twinkle twinkle …", "…", "…" ))
res12: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[602] at flatMap at
<console>:27

// Divida a usando o flatMap – os elementos são do tipo String agora
> val splitWordsRDD = linesRDD.flatMap(_.toLowerCase().split("\\s"))
```

Resumo

- Tem muitas formas de uma simples divisão de dados ser feita
  - DataFrames vs. Datasets vs. RDDs
  - `flatMap()` vs. `explode/split`
  - A API é grande, com muitas opções
- Geralmente, escolha entre Dataset ou DataFrame em vez de RDD
  - Simplistas, e tem um desempenho melhor
- Iremos explorar os benefícios do Catalyst e do Tungsten em breve

Lab 4.5: Divisão de dados

Questões de revisão

- O que é Spark SQL, e por que temos que usá-los?
- Como você lê dados utilizando o `SparkSession`? Quais tipos de dados podem ser lidos?
- O que é `DataFrame`? E `Dataset`?
- O que é a consulta DSL em `DataFrame`?

Resumo

- Spark SQL é um componente para processamento de dados estruturados
  - Ele adiciona um schema ao dado que estamos trabalhando
  - Ele prove uma API simples com processamento mais eficiente se usarmos o Catalyst e o Tungsten
- Use **`SparkSession`**, **`DataFrameReader`**, e **`DataFrameWriter`** para ler e armazenar dados
  - Muitos tipos de dados comuns são suportados, entre eles:
  - JSON, CSV, database, parquet, text
- O DataFrame adiciona **informação de schema** nos dados
  - Em alguns casos (e.g. JSON) o Spark pode inferir o schema
- O `Dataset` adiciona **segurança de tipo em tempo de compilação**

Resumo

- Você pode consultar dados de um DataFrames usando:
  - DSL
  - SQL
  - Transformações tipadas

Parte 5: “Shuffling” de Transformações e Performance

Agrupando e Reduzindo

“Shuffling”

O otimizador Catalyst

O otimizador Tungsten

Resumo

Parte 4.1: Agrupando e Reduzindo (“Grouping and Reducing”)

Visão geral de agrupamento

- O agrupamento organiza os dados em grupos
  - Baseando-se no valor escolhido de cada coluna
  - Geralmente usado para **agregação** (e.g. count, max, sum, etc.)
  - Em exemplos anteriores, fizemos o agrupamento pela coluna “gender”
  - Um grupo para 'M’, outro para ‘F’
- O agrupamento envolve o **embaralhamento** (“shuffling”) dos dados
  - Representa a distribuição dos dados entre as partições
  - Implicações importantes de performance — mais detalhes em breve
- Os métodos de agrupamento de `Dataset/DataFrame` são simples
  - Os obetos **`[KeyValue/Relational]GroupedDataset`** lidam com a agregação
- O método `groupByKey` do RDD é mais complexo
  - Baixo nível, você lida com o peso da agregação

Métodos de agrupamento

- Métodos para **`DataFrame`**:

  `groupBy``(cols: Column*): `**`RelationalGroupedDataset`**
  - Agrupamento usando uma coluna específica
  - `groupBy(col1: String, cols: String*): RelationalGroupedDataset`
    - Agrupamento usando mais de uma coluna específica

- Métodos para **`Dataset`**:

  `groupByKey``[K](func:(T) ⇒ K):`**`KeyValueGroupedDataset`**`[K, T]`
  - Agrupar por uma chave (do tipo K) retornada pela função

- Métodos para RDD (em `org.apache.spark.rdd.`**`PairRDDFunctions`**)

  `groupByKey``(): RDD[(K, Iterable[V])]`
  - Trabalha com pares (`chave, valor`)
  - Retorna outro RDD

Contagem de palavras usando DataFrame

- Simples — apenas agrupar e contar
  - Os métodos de agregação são simples de usar
    - Instâncias do objeto `RelationalGroupedDataset`
  - Este exemplo **usa o DSL**

```
// Use splitWordsDF visto em casos anteriores
// Agrupe por "word", e então faça a contagem
> val countDF = splitWordsDF.groupBy('word).count

> countDF.show
+-------+-----+
|   word|count|
+-------+-----+
|    you|    1|
|    how|    1|
```

Contagem de palavras usando Dataset

- Simples também — `groupByKey` e faça a contagem
  - Também é simples utilizar os métodos de agregação
    - Instâncias do objeto `KeyValueGroupedDataset`
  - Este exemplo utiliza lambda

```
// Use splitWordsDF visto em casos anteriores
// Agrupe por "word", e então faça a contagem
> val countDS = splitWordsDS.groupByKey(s => s).count

countDS.show
+-------+--------+
|  value|count(1)|
+-------+--------+
|    you|       1|
|    how|       1|
```

Agrupamento em RDD

- São necessários **pares de RDD** para realizar o agrupamento
  - RDD contendo pares de chave/valor **`(chave, valor)`**
  - e.g. `(M, 35) ou (twinkle, 1)`
  - Muitos outrso métodos necessitam de pares de RDD
  - **`PairRDDFunctions`** define funções extras para pares de RDDs

- Nossos pares de RDD possuem a estrutura:
  - **`chave`**: Campo “age”, **`valor`**: Contagem (1 nesse caso)
  - A agregação é codificada por nós

- O agrupamento para RDD é mais difícil de compreender que para DataFrame/Dataset
  - O agrupamente retorna uma chave (o grupo) em par com um objeto do tipo Iterable (os elementos do grupo) — codificação complexa e de baixo nível

```
groupByKey(): RDD[(K, Iterable[V])]
```

Contagem de palavras usando agrupamento em RDD

- Nós criamos pares de RDDs (obrigatório para agrupar)
  - Nossos pares tem o formato `(twinkle, 1)`
- O agrupamento é ineficiente para este exemplo
  - Existem outros métodos melhores (e.g. `reduceByKey`)
  - Mas isso serve de um exemplo de agrupamento

```
// Use splitWordsDF visto em casos anteriores

splitWordsRDD.map(x => (x, 1))  // Cria pares (word, 1)
  .groupByKey()                 // Agrupa por chave
  .map( x => (x._1, x._2.size)) // O resultado do valor representa a contagem
  .collect


res15: Array[(String, Int)] = Array((how,1), (i,1), (star,2), (wonder,1),
(are,1), (twinkle,4), (what,1), (little,2), (you,1))
```

Contagem de palavras em RDD com reduceByKey

- Pode ser feito de forma mais eficiciente usando o **reduceByKey()**

  reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]

  - Mescla os valores para a chave que é especificada, utiliza uma função de redução associativa e comutativa
  - **Mescla de forma local** em cada mapeador antes de enviar os resultados para um redutor

```
// Use splitWordsDF visto em casos anteriores

splitWordsRDD.map(x => (x, 1))    // Criando os pares de (word, 1)
  .reduceByKey(_+_)               // Reduz pela adição (contagem)
  .collect


res18: Array[(String, Int)] = Array((how,1), (i,1), (star,2), (wonder,1),
(are,1), (twinkle,4), (what,1), (little,2), (you,1))
```