Parte 4.2: SparkSession e Carregamento/Armazenamento de dados

Scala Systems

Principais tipos de API

- Todos os tipos falados até agora estão em: **`org.apache.spark.sql`**
- **`DataFrame`**: Coleção distribuída com um **schema**
  - Sinônimo para **`Dataset[Row]`**
- **Dataset:** Coleção **fortemente tipada** com schema
- **`Column`**: Uma coluna em um DataFrame
  - Usado para criar expressões no Query DSL
- **`Row`**: Representa o dado no formato tabular
- **`SparkSession`**: Ponto de entrada para o Spark SQL
  - `Substitui os antigos objetos SQLContext e HiveContext`
  - Pré-criados no Spark shell
- **`DataFrameReader/Writer`**: Para carregar/armazenar dados

SparkSession (ponto de entrada da API)

- As habilidades da classe incluem:
  - Ler arquivos de dados (via **`DataFrameReader`**)
  - Criar instâncias de DataFrame e Dataset
  - Executar consultas SQL

- Instâncias acessadas a partir de um Builder (uma Fábrica)
  - Builder é parte do **objeto** SparkSession
  - Mais detalhes em breve
  - Uma sessão é pré-criada no REPL na variável **`spark`**
    - `getOrCreate()` para criar ou retornar uma instância já criada

```
> SparkSession.builder.getOrCreate
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5

> spark
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5
```

DataFrameReader

- Interface para carregar dados de uma origem externa
  - Obtida via **`SparkSession.read()`**
- Por padrão já suporta leitura de formatos de dados comuns
  - **Infere o schema** automáticamente
  - json, parquet, csv, jdbc para databases relacionais, hive, entre outros
- Abaixo, o método **`spark.read()`** retorna um `DataFrameReader`
  - **`json("people.json")`** carrega dados do arquivo *people.json*
    - O dado precisa estar no formato de linhas JSON
    - Um objeto JSON por linha, mais algumas outras limitações

```
val folksDF=spark.read.json("people.json") // Obter o dado
```

DataFrameReader API

- Métodos para carregar arquivos em formatos específicos
  - `csv(path: String)`: Carrega um arquivo no formato CSV
  - `jdbc(…)`: Carrega dados de um database relacional
  - `json(path: String)`: Carrega um arquivo no formato JSON
  - `parquet(path: String)`: Carrega um arquivo no formato PARQUET
  - `text(path: String)`: Carrega um arquivo no formato TEXTO
  - Entre outros

- Pode-se também explicitar detalhes do formato dos dados
  - `format()`: Específicar o formato do dado
    - Recebe uma classe ou um nome de um formato (e.g json)
  - `schema()`: Específicar o schema do dado
    - Detalhes do dado (via instâncias `StructType/StructField`)

```
val folksDF=spark.read.format("json").schema(…).load("people.json")
```

DataFrameWriter

- Interface para armazenar dados em uma fonte externa
  - Obtido via **`SparkSession.write()`**
  - Habilidades similares ao `DataFrameReader`
    - Por padrão suporta escritas nos formatos csv, jdbc, json, parquet, e texto
- Abaixo, exemplificamos o armazenamento de dados no formato parquet
  - O dado foi lido no formato JSON
  - Pode-se fácilmente transformar o dado (JSON => parquet)

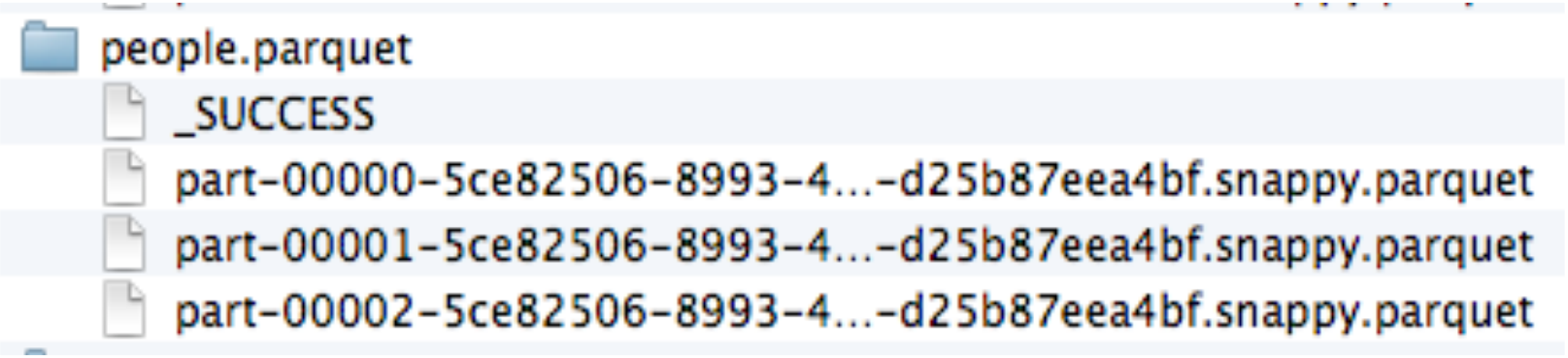
```
val folksDF=spark.read.json("people.json") // Obter o dado (JSON)

folksDF.write.parquet("people.parquet")    // Escrever o dado (parquet)
```

Múltiplos arquivos de escritas

- **O formato de armazenamento** para a escrita de um DataFrame é:
  - **Pasta para armazenamento** com o nome escolhido
  - **Múltiplos arquivos** com os dados dento da pasta
    - Um arquivo para cada partição, que geralmente vai ser escrito para um File System distribuído (e.g. HFS)
    - É possível escrever tudo em um único arquivo

- Abaixo, um exemplo de escrita para o *people.parquet*
  - 3 partições



Scala Systems

Interfaces fluentes

- **Interfaces fluentes** foram criadas para tornar o código mais legível e fluido
  - Torna simples a leitura e a escrita
  - Geralmente permite fazer cadeias de chamadas e as vezes usam Builders
- A API do Spark SQL utiliza Interfaces Fluentes
  - Abaixo, um exemplo de Interface Fluente na API do Spark SQL e outro exemplo do não uso de uma Interface Fluente para o mesmo caso

```
val folksDF=spark.read.format("json").schema(…).load("people.json")
```

```
// Exemplo não fluente
val reader = spark.read
reader.setFormat("json")
reader.setSchema(…)
val folksDF=reader.load("people.json")
```

LAB

MINI-LAB — Reveja a Documentação

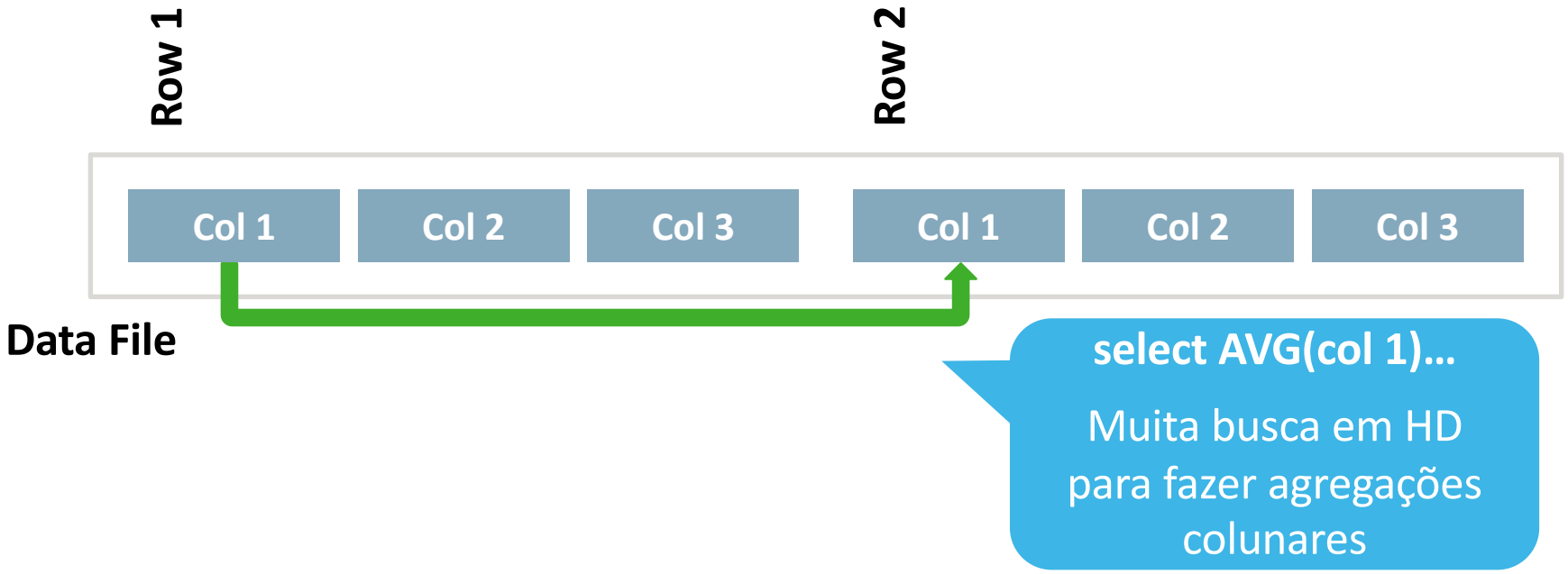
- Acesse a documentação do Spark em: http://spark.apache.org/docs/latest/
  - Busque por `org.apache.spark.sql` no paínel esquerdo
- Reveja:
  - `Classe SparkSession` (clique em `C` perto dele na lista)
    - Reveja os métodos `read()` e `createDataFrame()`
  - `Objeto SparkSession` (clique em `O` perto dele na lista)
    - Veja os métodos
    - Siga o link para o `Builder` da classe e reveja-o
  - `DataFrameReader` e `DataFrameWriter`
    - Reveja os métodos que aprendemos nessa aula

Formatos de dados

- O Spark suporta muitos formatos de dados
- Vamos passar por uma breve visão geral dos formatos suportados
- Os formatos suportados:
  - Formatos baseados em linhas e colunas
  - Formatos baseados em Texto (e.g. JSON and CSV)
  - Formatos Binários

Visão geral: Armazenamento baseado em linhas

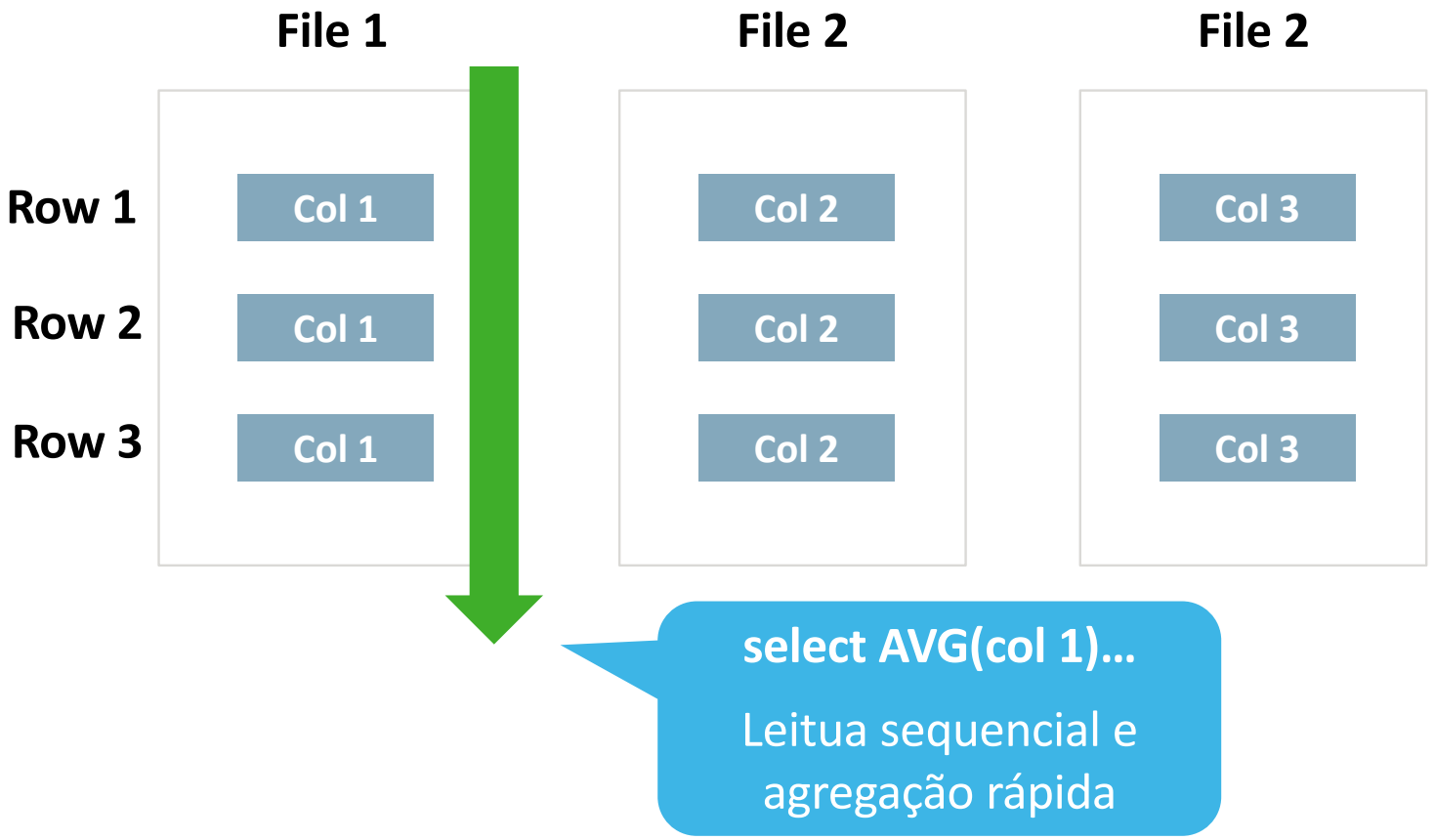
- Linhas armazenada fisicamente juntas
  - Geralmente bastante indexado (e.g. DB relacional)
  - Uma boa escolha para consultas do tipo “select *”
  - Não é uma boa escolha para agregação (e.g. calcular média)



Scala Systems

Visão geral: Armazenamento baseado em colunas

- Armazenamento de **colunas** fisicamente juntas
  - Otimizado para agregação de uma única coluna
    `select MAX(temp) from sensors;`
  - Não é uma boa escolha para buscas do tipo "`select *`"



Scala Systems

Formatos de dados comuns baseados em texto

- Todos são baseados em linhas
- **JSON**: (JavaScript Object Notation)
  - Formato leve de transferência de dados
  - Leitura via `DataFrameReader.json()`
    - Automáticamente **infere** o schema
- **CSV**: (Comma Separated Values)
  - Formato simples de dados tabulares
  - Leitura via `DataFrameReader.csv()`
    - Autimáticamente **infere** o schema
- Formato de texto live
  - Leitura via `DataFrameReader.text()`
  - Geralmente analisa o texto e aplica um schema manualmente

Parquet

- Formato de dados colunares (um projeto Apache)
  - Armazenamento baseado em binários, compactação eficiente
  - O schema é armazenado junto do arquivo (o arquivo é **autodescritivo**)
  - Muito eficiente para consultas colunares
  - Possui bom suporte no ecossistema Hadoop e em outras ferramentas
- Escolhido como formato padrão em muitos lugares
- Leitura via `DataFrameReader.parquet()`

Outros formatos

- **Avro**: Formato binário baseado em linhas (projeto Apache Avro)
  - O schema é armazenado junto com o arquivo
  - Suportado pela biblioteca spark-avro externa do Spark
- Formatos **baseados no Hadoop** (binário / sequencial)
  - Baseados em linhas, pares de chave/valor
  - Suportado por métodos do SparkContext (`binaryFile, hadoopFile, newAPIHadoopFile`)
  - Baseados em RDD
- **Optimized Row Columnar** (ORC): Formato hibrido de linhas e colunas
  - Armazena linhas, e dentro de linhas possue-se dados armazenados em formato colunar
  - Comum encontrar em armazenamento de dados do Hive
  - O Spark pode suportar usando as bibliotecas do Hive

Lab 4.1: Formatos de dados

Parte 4.3: Introdução à DataFrame/Dataset

Overview

- **DataFrame**: Coleção distribuída de dado com **schema**
  - **Dado estruturado**: Dado organizado em colunas nomeadas
- O schema pode ser criado de duas formas, sendo elas:
  - **Inferido** a partir do dado (dependendo do formato)
  - **Declarado** explicitando a estrutura
- Diversas APIs para consultas
  - **DSL**: API funcional mais linguagem de expressão
  - Consultas **SQL**
  - Suporte a **lambdas**, mas nem sempre é a melhor opção
- **`Dataset`** adicionam segurança de tipo em tempo de compilação

DataFrame/Dataset: Um pouco de história

- Inicialmente chamado de **`SchemaRDD`** (1.0), depois foi renomeado para **`DataFrame`** (1.3)
- **`Dataset`** foi introduzido no Spark 1.6
  - Criado como um tipo separado de DataFrame para retrocompatibilidade
- Dataset e DataFrame **unificados** na versão 2.0
  - DataFrame é agora um typedef de **`Dataset[Row]`** (Scala)
  - A API é definida no Dataset, e dividida em sessões
    - Operações **sem declaração** derivam de um Dataframe
    - Operações **com declaração** derivam de um Dataset
- Um pouco confuso — ainda mais se você trabalhou com versões anteriores
  - E a documentação pode ser confusa também
- Vamos usar **`DataFrame`** para nos referir à API não tipada

Criando DataFrames/Datasets

- Vimos anteriormente a criação a partir de carregamento de arquivos
- Uma Seq de uma classe sendo fácilmente convertida para `DataFrame/Dataset`
  - Via **`toDF()`** ou **`toDS()`**
- Podem também ser criados a partir de transformações

```
// Declarando uma classe
> case class Person (name: String, gender: String, age: Long)

> val folks = Seq (Person("John", "M", 35), Person("Jane", "F", 40),
Person("Mike", "M", 20), Person("Sue", "F", 52))

// Criando um Dataframe
> val folksDF = folks.toDF
folksDF: org.apache.spark.sql.DataFrame = [name: string … ]

// Create um Dataset
> val peopleDS = folks.toDS
peopleDS: org.apache.spark.sql.Dataset[Person] = [name: string … ]
```

Conversões implícitas: toDF e toDS

- **Seq** é uma classe padrão da biblioteca Scala
  - E isso não define os métodos toDF e toDS do Spark
  - Estes são adicionados por **conversões implícitas** do Scala

- **Conversões implícitas** convertem objetos em outras classes
  - O que pode adicionar novos métodos a um tipo

- O Spark define conversões explicitas em um objeto Seq para os métodos toDF / toDS
  - No pacote `spark.implicits` que é importado automáticamente no Spark Shell
    - Deve ser importado em outros ambientes
  - O Spark utiliza muitos “implicits”
    - Iremos indica-los em próximos exemplos para maior clareza

”Visualizando" o DataFrame

- **`show()`** Exibe os dados do Dataset em formato tabular
  - Diferentes formas de mostrar os dados (veja a documentação)
    - e.g. **`show (numRows: Int)`** exibe a quantidade numRows de linhas
  - Útil para consultas ad-hoc em um Dataset

```
// Exibe os dados
> folksDF.show
+----+------+---+
|name|gender|age|
+----+------+---+
|John|     M| 35|
|Jane|     F| 40|
|Mike|     M| 20|
| Sue|     F| 52|
+----+------+---+
```

O Schema

- Todo DataFrame/Dataset possue um **schema**
  - Descreve aos seus campos (nome, tipo e nulidade)
  - Os dados precisam estar de acordo com o schema
- **`printSchema`** exibe no console as informações do schema de forma "amigável"
- **`schema`** exibe a representação interna do schema

```
> folksDF.printSchema // Exibição do schema amigável
root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)

> folksDF.schema  // Estrutura interna do schema
res10: org.apache.spark.sql.types.StructType =
StructType(StructField(name,StringType,true),
StructField(gender,StringType,true), StructField(age,LongType,false))
```

Como o Schema é determinado

- De muitas formas, sendo algumas delas:
- **Inferindo a partir de Meta-dados**: O Spark utiliza informações disponibilizadas pelo formato
  - Arquivos Parquet, DB schema
  - Classes do Scala, classes do JavaBean
- **Inferindo a partir dos dados**: O Spark descobre as informações olhando para os dados
  - Suportado para os formatos JSON e CSV
- **Especificando no código**: Feito pelo client no momento de criação do Dataset
  - e.g. para dados em formato de texto

Inferindo o Schema de um JSON

- Parece simples: `spark.read.json("path-to-file")`
  - E imediatamente os seus dados estão com o schema correto

- Algumas coisas que devemos saber sobre este processo:
  - **Não** é um processo de execução **lazy**: O Spark escanea o dado **imediatamente** para inferir o schema
  - É **frágil**: Qualquer problema nos dados influencia na qualidade da inferição

```
> var folksBadDF = spark.read.json("data/people-bad.json")

> folksBadDF.show
+--------------------+----+------+----+
|     _corrupt_record| age|gender|name|
+--------------------+----+------+----+
|{"name": "John", ...|null|  null|null|
|                null|  40|     F|Jane|
```

Declarando o Schema no código

- Abaixo, criamos o Schema usando o StructType
  - Objeto usado para Schemas internos do Spark
  - Sem a necessidade de escanear os dados para inferir um schema
  - Continua frágil caso existam dados errados — e.g. considerando que uma linha possue um atributo (age) errado:
    - `{"name": "John", "gender": "M", "age": "35"}`

```
> import org.apache.spark.sql.types._ // Importando os tipos do Schema
> val mySchema = (new StructType).add("name", StringType).add("gender",
StringType).add("age", IntegerType)

// Criação do DF com o Schema criado anteriormente
> var folksDF = spark.read.schema(mySchema).json("data/people.json")

> folksDF.show
17/06/01 13:57:38 WARN JacksonParser: Found at least one malformed records //
remaining warning omitted …
+----+------+----+
|name|gender| age|
+----+------+----+
|null|  null|null| // A linha com problema fica com todos os dados nulos.
|Jane|     F|  40|
```

Criando o Schema a partir de consultas

- Primeiro, especifique o schema como totalmente string
  - Então use as habilidades do DataFrame para alterar o schema

```
> val mySchema = (new StructType).add("name", StringType).add("gender",
StringType).add("age", StringType)  // Veja que o campo age está definido como
                                    // string

// O DF possue o schema definido anteriormente
> var folksDF = spark.read.schema(mySchema).json("data/people.json")

> val folksWithNewSchemaDF = folksDF.select(
  'age.cast("integer"), 'name, 'gender )

> folksWithNewSchemaDF.schema
… StructType(StructField(age,IntegerType,true),
StructField(name,StringType,true), StructField(gender,StringType,true))

> folksWithNewSchemaDF.show
+---+----+------+
|age|name|gender|
+---+----+------+
| 35|John|     M| // Funcionou com os mesmos dados do slide anterior
| 40|Jane|     F|
```

API do Dataset : Categorias de operações

- **Ações**: Retornam valores, exibe-os, processamento local
  - e.g. `collect()`, `first()`, etc. — similar as ações do RDD
- **Funções Básicas**: Funções de uso geral
  - e.g. `cache()`, `unpersist()`, `schema()`, `explain()`, `write()`
- **Transformações tipadas**: Retornam um `Dataset[T]`
  - Algumas são parecidas com o RDD, e.g. função de filtro a partir de um lambda
    - `filter(func: (T) ⇒ Boolean): Dataset[T]`
  - Outras não tão parecidas, e.g. função de filtro a partir de um objeto Column
    - `filter(condition: Column): Dataset[T]`
- Transformações não tipadas: Retornam um `DataFrame`
  - `select()`, `groupBy()`, `join()`, `agg()` (aggregate)

LAB

MINI-LAB — Reveja a Documentação

- Acesse os documentos do Spark em:
  http://spark.apache.org/docs/latest/
  - Procure por **`org.apache.spark.sql`**
  - Perceba que `DataFrame` está definido como `Dataset[Row]`
- Reveja brevemente a **classe Dataset**
  - Observe como a lista é dividida em ações, funções básicas, transformações tipadas e transformações não tipadas
  - Reveja cada sessão brevemente
- Reveja **SparkSession.implicits**
  - Na página da API do `SparkSession`, reveja seus objetos **`implicitos`**

Lab 4.2: Schema