

# Spark: The Definitive Guide

Uma visão resumida do livro

Estudos sobre Big Data

Thiago Panini



# Spark: The Definitive Guide

**Uma visão resumida do livro**

by

Thiago Panini

---

# Prefácio

O universo tecnológico vem evoluindo de forma estrondosa nos últimos tempos. Um dos principais cenários criados por esta extraordinária evolução é a imensa quantidade de dados gerada a todo instante. Armazenar, gerenciar e extrair valor deste grande volume de dados, associado ao termo *Big Data*, são ações que possuem um papel fundamental nesta nova dinâmica analítica do mundo moderno.

Com isto em mente, este é um material criado com o intuito de desbravar o universo de *Big Data* e o uso do Spark como uma forma eficiente de processamento paralelo de dados. De forma mais específica, o conteúdo aqui consolidado terá, como principal fonte, o livro *Spark: The Definitive Guide* de Bill Chambers e Matei Zaharia e, cada capítulo deste material trará um resumo geral sobre o capítulo equivalente encontrado no livro citado. Eventualmente, outras fontes de pesquisa poderão enriquecer os detalhes aqui estabelecidos.

Segundo o próprio prefácio do livro *Spark: The Definitive Guide*, Apache Spark é, atualmente, um dos mais populares sistemas para processamento de dados em larga escala, possuindo APIs em diferentes linguagens de programação e uma comunidade altamente ativa em termos de desenvolvimento de novas soluções e apoio a novos entusiastas. Ambos os autores do livro estiveram diretamente envolvidos com a criação do Apache Spark, sendo também responsáveis pela difusão, evolução e distribuição à comunidade de tecnologia como um todo.

*Thiago Panini*

*March 2022*

---

# Contents

<b>Prefácio</b>	i
<b>1 O que é Apache Spark?</b>	1
1.1 Contexto: Desafios em Big Data . . . . .	2
1.2 A História do Spark . . . . .	2
1.3 Executando Spark . . . . .	3
1.4 Bônus: Explorações Práticas . . . . .	4
<b>2 Uma Gentil Introdução ao Spark</b>	6
2.1 Arquitetura Básica do Spark . . . . .	6
2.1.1 Aplicações Spark . . . . .	6
2.2 Linguagens de Programação e Spark . . . . .	7
2.3 DataFrames . . . . .	8
2.4 Transformações . . . . .	8
2.4.1 Lazy Evaluation . . . . .	9
2.5 Ações . . . . .	9
2.6 DataFrames e SQL . . . . .	10
2.7 Bônus: Explorações Práticas . . . . .	11
<b>3 Uma Tour no Kit de Ferramentas do Spark</b>	17
3.1 Executando Aplicações em Produção . . . . .	18

3.2 Datasets: API <i>type-safed</i> . . . . .	19
3.3 Structured Streaming . . . . .	19
3.4 Machine Learning e Advanced Analytics . . . . .	20
3.5 Lower-Level APIs e SparkR . . . . .	21
<b>4 Visão Geral Sobre APIs Estruturadas</b>	<b>22</b>
4.1 DataFrames e Datasets . . . . .	23
4.2 Schemas e Tipos Primitivos do Spark . . . . .	24
4.2.1 Tipo <i>Column</i> e <i>Row</i> . . . . .	24
4.3 Execução de uma API Estruturada . . . . .	24
4.4 Plano Lógico . . . . .	26
4.5 Plano Físico . . . . .	26
<b>5 Operações Básicas Estruturadas</b>	<b>28</b>
5.1 Schema . . . . .	29
5.2 Colunas e Expressões . . . . .	30
5.3 Linhas e Registros . . . . .	32
5.4 Transformações em DataFrames . . . . .	33
5.4.1 Criando DataFrames . . . . .	34
5.4.2 select e selectExpr . . . . .	35
5.4.3 Literais . . . . .	38
5.4.4 Adicionando Colunas . . . . .	40
5.4.5 Renomando Colunas . . . . .	40
5.4.6 Removendo Colunas . . . . .	41
5.4.7 Casting . . . . .	41
5.4.8 Filtrando Registros . . . . .	43

5.4.9 Removendo Duplicatas . . . . .	45
5.4.10 Amostragem . . . . .	45
5.4.11 Unindo Registros . . . . .	46
5.4.12 Ordenando Registros . . . . .	48
5.4.13 Limitando Registros . . . . .	48
5.4.14 Reparticionando DataFrames . . . . .	48
5.4.15 Coletando Registros no Driver . . . . .	49
<b>6 Trabalhando com Diferentes Tipos de Dados</b>	<b>52</b>
6.1 Leitura dos Dados . . . . .	53
6.2 Trabalhando com Booleanos . . . . .	54
6.3 Trabalhando com Números . . . . .	56
6.4 Trabalhando com Strings . . . . .	60
6.5 Trabalhando com Datas . . . . .	65
6.6 Trabalhando Dados Nulos . . . . .	68
6.7 Trabalhando com Tipos Complexos . . . . .	72
6.7.1 Structs . . . . .	72
6.7.2 Arrays . . . . .	73
6.7.3 Maps . . . . .	78
6.8 Trabalhando com JSON . . . . .	80
6.9 UDFs. . . . .	82
<b>7 Agregações</b>	<b>88</b>
7.1 Funções de Agregação . . . . .	89
7.1.1 count . . . . .	89
7.1.2 countDistinct . . . . .	91

7.1.3 approx_count_distinct . . . . .	91
7.1.4 first e last . . . . .	92
7.1.5 min e max . . . . .	93
7.1.6 sum . . . . .	93
7.1.7 sumDistinct . . . . .	94
7.1.8 avg . . . . .	94
7.1.9 var e stddev . . . . .	96
7.1.10 skewness e kurtosis . . . . .	97
7.1.11 corr e covar . . . . .	98
7.1.12 Agregando para Tipos Complexos . . . . .	99
7.2 Agrupamento . . . . .	100
7.3 Funções Window . . . . .	101
7.3.1 Especificando Janela . . . . .	103
7.3.2 Agregando Sobre a Janela . . . . .	104
7.3.3 Construindo Consulta . . . . .	105
7.4 Grouping Sets . . . . .	106
7.4.1 Rollups . . . . .	108
7.4.2 Cube . . . . .	109
7.4.3 Pivot . . . . .	111
<b>8 Joins</b>	<b>115</b>
8.1 Definição Geral de Joins . . . . .	116
8.2 DataFrames para Testes . . . . .	116
8.3 Tipos de Joins . . . . .	118
8.3.1 Inner Join . . . . .	118

8.3.2 Left Join . . . . .	120
8.3.3 Right Join . . . . .	120
8.3.4 Left Semi e Left Anti Join . . . . .	121
8.3.5 Considerações Finais . . . . .	121
8.4 Joins em Tipos Complexos . . . . .	122
8.5 Execução de Joins pelo Spark . . . . .	124
8.5.1 Estratégias de Comunicação . . . . .	125
<b>9 Fontes de Dados</b>	<b>128</b>
9.1 Fundamentos de Leitura de Fontes de Dados . . . . .	129
<b>References</b>	<b>130</b>

---

# 1

## O que é Apache Spark?

O Apache Spark pode ser definido como uma ferramenta computacional unificada que contém um conjunto de bibliotecas para processamento paralelo de dados em clusters de computadores. Em sua natureza, o Spark oferece um vasto leque de possibilidades relacionadas a Big Data, desde o uso de SQL à tarefas de *machine learning*.

Sendo uma ferramenta **computacional unificada**, o Spark traz consigo uma série de APIs e bibliotecas que podem ser utilizadas para os mais variados propósitos: SQL e dados estruturados (Spark SQL), *machine learning* (MLlib), *streaming* de dados (Spark Streaming), análise de grafos (GraphX), entre outros.

Ao mesmo tempo, é correto dizer que o Spark limita seu escopo ao **processamento** de dados, diferente de outras plataformas como, por exemplo, o Apache Hadoop que, por sua vez, inclui tanto um sistema de armazenamento, como também de processamento distribuído. Ao focar apenas na parte computacional, o Spark garante um ambiente que pode facilmente ser integrado a diferentes plataformas, como Amazon S3, Azure Storage, HDFS, entre outras.

## 1.1. Contexto: Desafios em Big Data

Sabe-se que, ao longo dos anos, o poder computacional evoluiu de forma significativa e, até certo ponto, aplicações puderam se aproveitar diretamente do aumento da velocidade de processamento individual de processadores. Entretanto, em 2005, limitações relacionadas a **dissipação de calor** alteraram o propósito: não fazia mais sentido investir na criação de processadores cada vez mais velozes, mas sim em tecnologia capaz de gerenciar o processamento de dados de maneira **paralela**.

Por outro lado, neste mesmo cenário, o **armazenamento** e a **coleta** de dados se tornaram cada vez mais **baratos** e, enquanto o processamento passava por desafios cada vez mais complexos, a obtenção e o armazenamento se tornavam mais “acessíveis” a cada instante. No meio de toda essa complexidade, ferramentas como o Apache Spark se tornaram essenciais para propor soluções que pudessesem equilibrar a balança.

## 1.2. A História do Spark

As raízes do Apache Spark estão na UC Berkeley que, em 2009, foi palco para um projeto de pesquisa posteriormente publicado com o título “*Spark: Cluster Computation with Working Sets*”. Matei Zaharia, um dos autores do livro alvo deste material, foi um dos principais envolvidos na publicação do *paper*.

Entre as ações realizadas para a consolidação do projeto de pesquisa do Spark, pode-se destacar conversas com usuários de MapReduce (dominante na época) para coletar pontos positivos e negativos em diferentes cenários de uso. Com isso, foi possível levantar dificuldades compartilhadas por usuários que executavam aplicações de *machine learning* em larga escala, principalmente em cenários que exigiam múltiplos ciclos de passagem pelos dados. Casos deste tipo requisitavam diversas submissões de *jobs* de Map Reduce de forma separada, impactando diretamente na velocidade das ações.

Para endereçar este problema, os desenvolvedores do Spark desenharam uma

API baseada em **programação funcional** e que posteriormente foi implementada de modo a performar compartilhamento de dados **em memória** ao longo dos passos computacionais exigidos por fluxos.

Ao longo do tempo, os times envolvidos incrementaram o Spark das mais variadas formas, sempre coletando as principais necessidades dos usuários e dentro do objetivo de fornecer uma ferramenta capaz de superar os desafios mais latentes de processamento de dados em larga escala. Neste período, novas bibliotecas foram adicionadas, o número de contribuidores cresceu substancialmente, uma nova companhia (Databricks) foi fundada pelos primeiros desenvolvedores para aprimorar a ferramenta e, por fim, o Spark encontra-se hoje em uma posição de destaque no mercado de desenvolvedores, entusiastas e qualquer um que necessite de suas funcionalidades para lidar com desafios de processamento de Big Data.

## 1.3. Executando Spark

É possível utilizar o Spark através do Python, Scala, R ou SQL. O Spark, em si, é construído em Scala e executado em uma JVM (*Java Virtual Machine*). As duas opções recomendadas pelos autores do livro para execução do Spark são: *download* e instalação do Apache Spark no próprio notebook de trabalho, ou utilização do *Databricks Community Edition* como um ambiente em nuvem que contém o Spark.

No livro, também é possível encontrar diferentes formas de executar o Spark, seja em Python ou em Scala. Existe também uma menção à biblioteca Python `pyspark` como uma forma efetiva de utilizar as funcionalidades do Spark sem a necessidade de realizar qualquer tipo de download consolidado da ferramenta.

Por fim, este primeiro capítulo é finalizado com algumas orientações adicionais relacionadas ao Github do livro contendo os códigos e ferramentas capazes de auxiliar o leitor no decorrer da jornada.

## 1.4. Bônus: Explorações Práticas

**Disclaimer:** esta seção utiliza tópicos do livro para evidenciar experiências pessoais com o Spark.

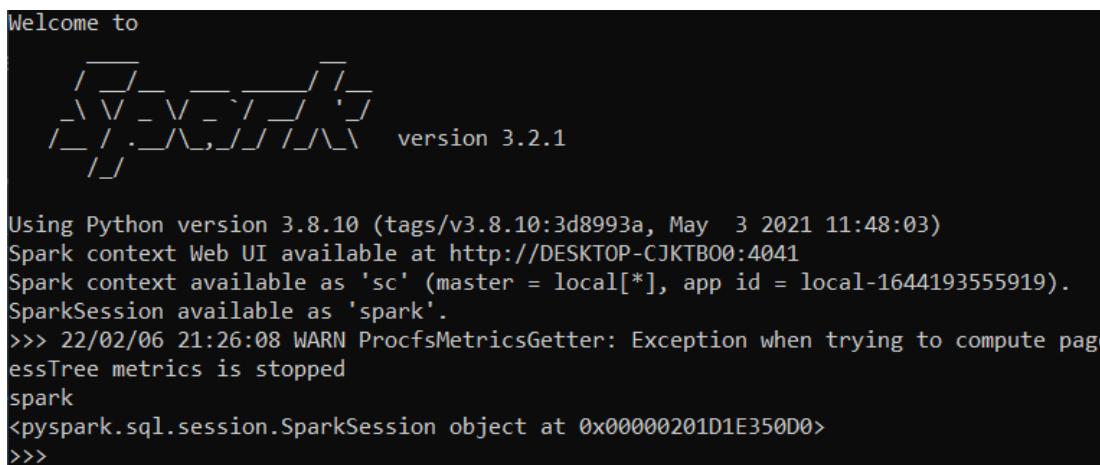
### Links Úteis:

- GitHub com recursos do livro
- Download Apache Spark
- Quick Start Apache Spark

Ao realizar o download e a extração do Apache Spark através do link providenciado acima, é possível iniciar um *shell* interativo utilizando diferentes linguagens de programação. No diretório de instalação do Spark, o comando abaixo é responsável por iniciar uma sessão no prompt de comando utilizando o `pyspark`:

```
$ ./bin/pyspark
```

O resultado pode ser visualizado na figura 1.1 e, como destaque, é possível visualizar a variável “`spark`” representando a sessão ativa de spark inicializada.



```
Welcome to
    _/\_ _\ \ / \ / \ / \ / \
   / \ \ / \ / \ / \ / \ / \ / \
  / \ / \ / \ / \ / \ / \ / \ / \
 / \ / \ / \ / \ / \ / \ / \ / \
version 3.2.1

Using Python version 3.8.10 (tags/v3.8.10:3d8993a, May 3 2021 11:48:03)
Spark context Web UI available at http://DESKTOP-CJKTB00:4041
Spark context available as 'sc' (master = local[*, app id = local-1644193555919].
SparkSession available as 'spark'.
>>> 22/02/06 21:26:08 WARN ProcfsMetricsGetter: Exception when trying to compute page
essTree metrics is stopped
spark
<pyspark.sql.session.SparkSession object at 0x00000201D1E350D0>
>>>
```

**Figure 1.1:** Primeiro contato com Apache Spark utilizando Python através do *shell* inicializado via `pyspark`.

Adicionalmente, é possível inicializar o `pyspark` através da instalação da biblioteca em um ambiente virtual python. Desta forma, é possível gerenciar scripts,

trabalhar em jupyter notebooks ou em outros ambientes de trabalho com maior liberdade. Neste cenário, a sequência de códigos demonstrada abaixo realiza a criação de um ambiente virtual python denominado spark-guide-venv, a instalação das bibliotecas notebook e pyspark.

```
$ python -m venv spark-guide-venv  
$ spark-guide-venv\Scripts\activate  
$ pip install notebook  
$ pip install pyspark
```

Por fim, a figura 1.2 contém uma ilustração de um jupyter notebook inicializado e as devidas importações do pyspark realizadas para proporcionar uma sessão interativa.

```
In [8]: # Importação do módulo pyspark  
from pyspark.sql import SparkSession  
  
# Criação de sessão  
spark = SparkSession.builder.getOrCreate()  
spark
```

Out[8]: **SparkSession - in-memory**  
**SparkContext**

[Spark UI](#)

**Version**  
v3.2.1

**Master**  
local[\*]

**AppName**  
pyspark-shell

**Figure 1.2:** Inicializando uma sessão Spark (SparkSession) através de um jupyter notebook criado a partir de um ambiente virtual python onde a biblioteca pyspark foi instalada via pip.

---

# 2

## Uma Gentil Introdução ao Spark

Após um entendimento geral sobre como o Apache Spark foi concebido, este segundo capítulo traz uma visão prática sobre alguns conceitos essenciais sobre o framework, desde sua arquitetura até exemplos de utilização de algumas APIs para processamento de dados.

### 2.1. Arquitetura Básica do Spark

Como informado anteriormente, o Spark atua em *clusters* de computadores aproveitando-se da habilidade de utilizar processamento paralelo para obter melhores resultados em Big Data. Para coordenar toda esta dinâmica, gerenciadores de clusters são utilizados para garantir recursos para aplicações submetidas. Como exemplos de gerenciadores, é possível citar o YARN [1] e o Mesos [2].

#### 2.1.1. Aplicações Spark

Aplicações Spark (na literatura, *Spark Applications*) são compostas por:

- **driver process:** mantém informações da aplicação, respondem à programas

de usuário ou à inputs e analise, distribui e agenda o trabalho através dos processos executores.

- **executor process:** executam o código associado pelo *driver* e reporta o status do processo computacional.

A figura 2.1 ilustra a relação entre processos *driver* e *executor* em um nó de um cluster. Em uma sessão Spark ativa, diferentes tarefas são associadas a diferentes “executores” através de um processo *driver*. O gerenciador do cluster realiza a orquestração das atividades e garante a disponibilidade dos recursos computacionais para a execução das tarefas.

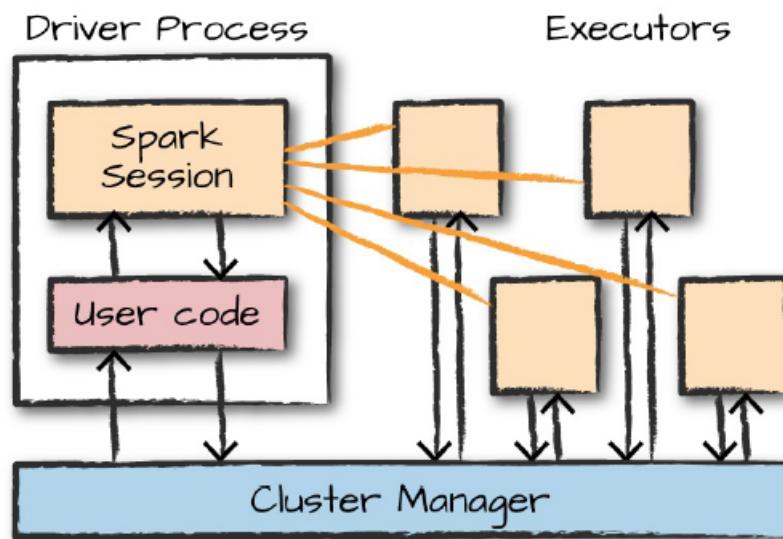


Figure 2.1: Relação entre processos *driver* e *executor* dentro de uma aplicação Spark.

## 2.2. Linguagens de Programação e Spark

Como informado no capítulo anterior, a execução do Spark se dá através da inicialização de uma sessão (comumente referenciada na literatura por `SparkSession`). As figuras 1.1 e 1.2 trazem, respectivamente, exemplos de inicialização de sessões Spark via *shell* ou via *jupyter notebook* (biblioteca `pyspark`) utilizando a linguagem Python.

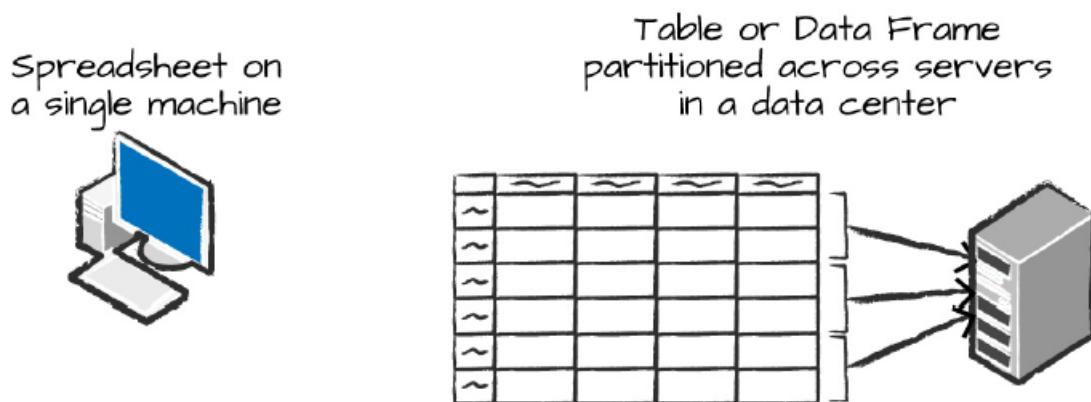
A partir deste ponto, diferentes situações podem ser programadas para alcançar os mais variados objetivos relacionados ao processamento de dados com Spark. Nas

seções a seguir, alguns elementos essenciais desta frente serão detalhados para um melhor entendimento dos processos.

## 2.3. DataFrames

No Spark, um objeto DataFrame representa uma tabela com linhas e colunas e, de maneira geral, pode expandir múltiplos computadores em um cluster (figura 2.2).

De modo a paralelizar tarefas, o Spark divide os dados em *chunks* chamados de *partições* que, por sua vez, podem ser definidas como uma coleção de linhas presentes em uma máquina física do cluster e que são automaticamente distribuídas.



**Figure 2.2:** Diferenciação entre um conjunto de linhas e colunas em uma máquina individual (planilha) e um DataFrame Spark dividido em múltiplos recursos computacionais em um cluster (partições).

## 2.4. Transformações

Transformações, em Spark, nada mais são do que **instruções** de modificação de um DataFrame explicitamente fornecidas via código. No geral, transformações **não retornam um output** até que uma **ação** seja explicitamente chamada. Existem dois tipos de transformações:

- **Narrow Dependencies** (ou transformações de dependência estreita): são aquelas onde cada partição de entrada contribui para uma partição de saída em uma

relação de 1 para 1. Filtros são exemplos de *narrow dependencies* ou *narrow transformations*.

- **Wide Dependencies** (ou transformações de dependência ampla): são aquelas onde partições de entrada contribuem para várias partições de saída. Também chamadas de *shuffle*.

Uma das principais diferenças entre *narrow* e *wide dependencies* diz respeito a forma como o Spark otimiza o processamento. Ao programar múltiplos filtros (*narrow dependencies*) em um DataFrame, por exemplo, o Spark optimiza o processamento através de uma execução contínua em memória. Ao performar uma operação de *shuffle* (*wide dependencies*), o Spark escreve o resultado em disco.

#### 2.4.1. Lazy Evaluation

Este é um conceito muito importante no entendimento do Spark pois, em linhas gerais, indica que o framework literalmente “**espera**” até o último momento para executar o grafo computacional de instruções. Em outras palavras, ao invés de modificar o dado (transformar) imediatamente com o comando solicitado, o Spark cria um **plano de transformações** a ser aplicado nos dados de entrada da forma mais eficiente possível.

Por exemplo, ao criar um *job* Spark complexo e com múltiplas transformações mas, com um filtro no final que requer o retorno de apenas uma única linha, a forma mais eficiente possível de realizar tal processamento é buscar apenas a linha necessária. Ao aguardar e construir o plano otimizado de transformações, o Spark optimiza o processamento.

## 2.5. Ações

Até este momento, foi possível entender transformações como uma construção de lógica capaz de transformar dados de entrada. Além disso, uma menção ao retorno inexistente de resultado até que uma **ação** seja explicitamente programada. Dessa

forma, diz-se de ações como uma forma de instruir o Spark a computar o resultado de uma série de transformações já programadas. As ações podem ser separadas em três grupos:

- Ações para visualizar dados no console
- Ações para coletar dados em objetos nativos na respectiva linguagem (ex: DataFrames)
- ações para escrever dados em outras fontes de saída

Os métodos `count()` e `take()` são exemplos básicos de ações que acionam o gatilho necessário para execução de um *job* Spark para aplicação das transformações e retorno do resultado.

## 2.6. DataFrames e SQL

Uma das características mais animadoras do Spark é a possibilidade de escrever toda a lógica de transformação como queries SQL. Tanto neste cenário, quanto na definição individual de transformações, o Spark compila a lógica em um plano antes de executar o código de fato. Com o Spark SQL, é possível registrar um DataFrame como uma tabela ou uma view (tabela temporária) e executar consultas utilizando puramente linguagem SQL.

Não há diferença de performance entre escrever transformações via Spark SQL ou via métodos nativos. O plano de execução criado pelo Spark será o mesmo para ambos os cenários (desde que as transformações sejam equivalentes, é claro).

Na seção **Bônus: Explorações Práticas**, grande parte dos tópicos discutidos neste capítulo serão compilados em um desafio prático utilizando `pyspark` em um jupyter notebook. No processo, dados reais de planos de vôos serão lidos e transformados com Spark, tanto via métodos, quanto via SQL.

## 2.7. Bônus: Explorações Práticas

**Disclaimer:** esta seção utiliza tópicos do livro para evidenciar experiências pessoais com o Spark.

Na figura 1.2, foi possível visualizar um objeto `SparkSession` inicializado em um jupyter notebook para ser utilizado em futuras ações com Spark. Agora, a sessão criada será alvo de análises reais utilizando dados de vôos retirados do Bureau de estatísticas de transportes dos Estados Unidos. Os dados foram obtidos da página do Github do livro.

Neste primeiro momento, serão utilizados dados em formato csv. A figura 2.3 ilustra uma amostra dos dados obtida através do console.

```
$ head /data/flight-data/csv/2015-summary.csv  
  
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count  
United States,Romania,15  
United States,Croatia,1  
United States,Ireland,344
```

**Figure 2.3:** Amostra dos dados utilizados na seção de exploração prática do capítulo.

A figura 2.4 contém o código Spark utilizado para ler o arquivo csv armazenado localmente e aplicar algumas transformações documentadas nos comentários. Adicionalmente, vale ressaltar que a leitura dos dados foi realizada a partir da configuração das opções `inferSchema` e `header` como `true` indicando, respectivamente, uma leitura com inferência de `schema` (Spark fará o melhor para “entender” os metadados) e considerando que a primeira linha contém o cabeçalho dos arquivos.

Considerando a possibilidade de utilizar SQL para realizar transformações nos dados, a figura 2.5 ilustra os processos de criação de uma tabela temporária no Spark (view) através do método `createOrReplaceTempView()` de um Spark DataFrame. Em seguida, a função `spark.sql()` pode ser executada para responder às mais variadas perguntas de negócio utilizando strings SQL como parâmetro. No exemplo, uma contagem de registros por destino de voo (coluna `DEST_COUNTRY_NAME`) é retornada.

```
In [25]: # Lendo arquivo via Spark
flights_data_2015 = spark\
    .read\
    .option("inferSchema", "true")\
    .option("header", "true")\
    .csv(DATA_PATH)

# Verificando "conteúdo" do objeto
print(f'DataFrame lido: \n{flights_data_2015}')
print(f'\nTipo primitivo: \n{type(flights_data_2015)}')

DataFrame lido:
DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: int]

Tipo primitivo:
<class 'pyspark.sql.dataframe.DataFrame'>

In [28]: # Coletando algumas linhas do DataFrame
flights_data_2015.take(3)

Out[28]: [Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344)]

In [29]: # Ordenando por "count" e coletando nova amostra
flights_data_2015.sort("count").take(3)

Out[29]: [Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1),
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1)]
```

**Figure 2.4:** Leitura de um arquivo csv local e armazenamento em um DataFrame do Spark.

Adicionalmente, a ação de coleta de amostras `take()` e a transformação de ordenação `sort()` são exploradas de forma a propor um entendimento geral sobre como funciona a dinâmica de utilização do Spark.

Analogamente, seria possível construir a mesma análise de dados através de métodos nativos do objeto Spark DataFrame como, por exemplo, o `groupBy()` e o `count()`. A figura 2.6 demonstra a mesma aplicação.

É extremamente importante citar que as análises de dados dispostas nas figuras 2.5 (via Spark SQL) e 2.6 (via Spark DataFrame) **compilam para o mesmo plano de execução**. Em outras palavras, isto indica que **não há diferença de performance** em ambos os formatos pois o Spark “traduz” as solicitações para um plano exatamente igual. A figura 2.7 prova este ponto através da ilustração dos planos (via método `explain()`) de ambos os objetos Spark DataFrame gerados pelas consultas.

Assim, após as explorações detalhadas nas figuras 2.5 e 2.6, a figura 2.8 traz um processo relativamente mais complexo onde o objetivo é utilizar a mesma base

```
In [13]: # Criando uma view (tabela temporária no Spark)
flights_data_2015.createOrReplaceTempView('flights_view')

# Contando registros por destino de voo via Spark SQL
sql_way = spark.sql("""
    SELECT DEST_COUNTRY_NAME, count(1) FROM flights_view
    GROUP BY DEST_COUNTRY_NAME
""")

# Mostrando resultados
sql_way.show(5)

+-----+-----+
|DEST_COUNTRY_NAME|count(1)|
+-----+-----+
|      Anguilla|      1|
|       Russia|      1|
|     Paraguay|      1|
|     Senegal|      1|
|      Sweden|      1|
+-----+-----+
only showing top 5 rows
```

**Figure 2.5:** Utilização de Spark SQL através do método `spark.sql()` para execução de queries utilizando o objeto de sessão do Spark.

```
In [19]: # Contando registros por destino de voo via DataFrame
df_way = flights_data_2015\
    .groupBy("DEST_COUNTRY_NAME")\
    .count()

# Mostrando resultados
df_way.show(5)

+-----+-----+
|DEST_COUNTRY_NAME|count|
+-----+-----+
|      Anguilla|      1|
|       Russia|      1|
|     Paraguay|      1|
|     Senegal|      1|
|      Sweden|      1|
+-----+-----+
only showing top 5 rows
```

**Figure 2.6:** Realizando a mesma análise de dados mostrada pela figura 2.5, porém utilizando métodos nativos do Spark DataFrame.

de dados para retornar os 5 principais destinos de vôos registrados. Para tal, os dois conceitos abordados são explorados (Spark SQL e Spark DataFrame) para que seja fornecida uma visão prática sobre ambas as práticas de realizar transformações e executar ações no Spark.

Algumas informações adicionais sobre as transformações aplicadas com Spark

```
In [24]: # Plano de execução das queries
sql_way.explain()
print()
df_way.explain()

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[count(1)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#16, 200), ENSURE_REQUIREMENTS, [id=#468]
      +- HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[partial_count(1)])
         +- FileScan csv [DEST_COUNTRY_NAME#16] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 pat
hs)[file:/D:/Users/thiagoPanini/OneDrive/Desenvolvimento/estudos/big data/...], PartitionFilters: [], PushedFilters: [], ReadSch
ema: struct<DEST_COUNTRY_NAME:string>

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[count(1)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#16, 200), ENSURE_REQUIREMENTS, [id=#481]
      +- HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[partial_count(1)])
         +- FileScan csv [DEST_COUNTRY_NAME#16] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 pat
hs)[file:/D:/Users/thiagoPanini/OneDrive/Desenvolvimento/estudos/big data/...], PartitionFilters: [], PushedFilters: [], ReadSch
ema: struct<DEST_COUNTRY_NAME:string>
```

**Figure 2.7:** Planos de execução das consultas compilados. Na imagem, percebe-se que, tanto para o modo utilizando Spark SQL quanto para o modo utilizando Spark DataFrame, o plano é exatamente igual.

DataFrame na figura 2.8:

- Foi preciso importar a função `desc` do módulo `pyspark.sql.functions` para aplicar uma ordenação descendente nos dados agrupados.
- O método `withColumnRenamed()` foi introduzido como uma forma de renomear colunas geradas nas transformações aplicadas.
- O método `limit()` foi apresentado como uma forma de limitar os resultados de transformações.

Por fim, visando trazer uma ideia visual sobre as transformações aplicadas nos cenários detalhados, a figura 2.9 ilustra o fluxo de transformações do DataFrame relacionado ao processo de retorno dos top 5 destinos escolhidos por viajantes. A figura atua como uma proxy do plano de execução da transformação que, de fato, possui apenas algumas diferenças atreladas a otimizações.

O plano de execução pode ser definido como um *directed acyclic graph* (ou DAG) de transformações, cada um resultando em um novo DataFrame imutável que pode ser utilizado para chamada de uma ação.

```
In [32]: # Utilizando Spark SQL
sql_top5 = spark.sql("""
    SELECT
        DEST_COUNTRY_NAME,
        sum(count) AS total_destination
    FROM flights_view
    GROUP BY DEST_COUNTRY_NAME
    ORDER BY total_destination DESC
    LIMIT 5
""")
sql_top5.show()
```

DEST_COUNTRY_NAME	total_destination
United States	411352
Canada	8399
Mexico	7140
United Kingdom	2025
Japan	1548

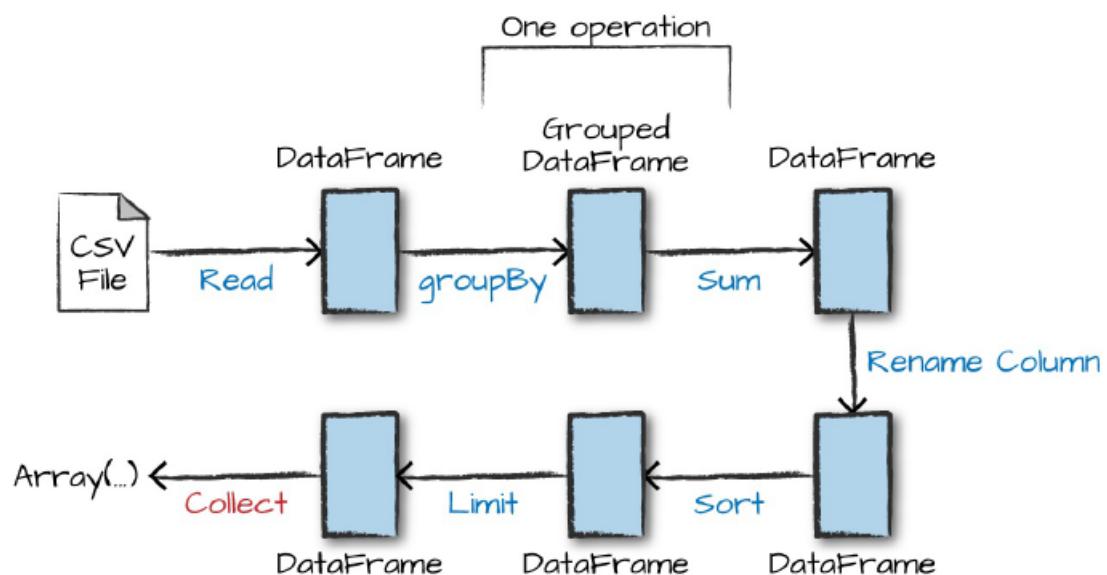
None

```
In [33]: # Utilizando Spark DataFrame
from pyspark.sql.functions import desc

df_top5 = flights_data_2015\
    .groupBy("DEST_COUNTRY_NAME")\
    .sum("count")\
    .withColumnRenamed("sum(count)", "total_destination")\
    .sort(desc("total_destination"))\
    .limit(5)\
    .show()
```

DEST_COUNTRY_NAME	total_destination
United States	411352
Canada	8399
Mexico	7140
United Kingdom	2025
Japan	1548

**Figure 2.8:** Aplicando transformações em Spark para retornar os 5 principais destinos escolhidos pelos viajantes na base de dados de vôos utilizada para análise. Na imagem, é possível visualizar as diferenças de sintaxes entre a utilização do Spark SQL e de métodos do Spark DataFrame para alcançar um mesmo resultado.



**Figure 2.9:** Fluxo de transformações no DataFrame ao retornar os top 5 destinos escolhidos por viajantes. Na imagem, é possível analisar o passo a passo de execução das transformações (DAG) em um cenário ilustrativo onde cada etapa retorna um DataFrame imutável.

---

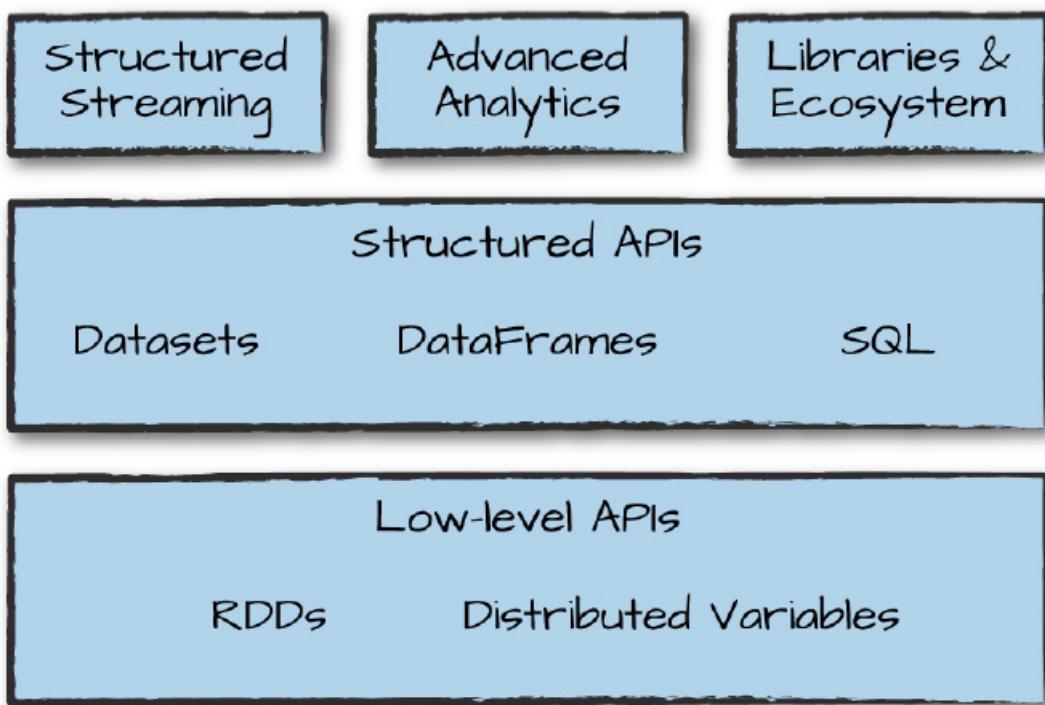
# 3

## Uma Tour no Kit de Ferramentas do Spark

Este é um capítulo genérico e amplo no sentido de fornecer ao leitor uma visão resumida sobre algumas das principais possibilidades em Spark. De forma consolidada, a figura 3.1 ilustra o ecossistema de componentes do Apache Spark.

Objetivando ser um capítulo exploratório e que abre portas para detalhamentos posteriores ao longo de outras seções, o conteúdo aqui consolidado tratá exemplos de alguns componentes do Spark de maneira prática. No cenário detalhado, os tópicos contidos neste capítulo do livro abordam os seguintes assuntos:

- Executando aplicações em produção com `spark-submit`
- Datasets: APIs “*type-safe*” para dados estruturados
- Streaming de dados estruturados
- Machine learning e advanced analytics
- *Resilient Distributed Datasets* (RDD)
- SparkR
- Ecossistema de pacotes *third-party*



**Figure 3.1:** Kit de ferramentas do Apache Spark

Para fins didáticos, neste resumo serão comportados apenas alguns pontos principais de parte dos tópicos acima listados.

## 3.1. Executando Aplicações em Produção

Com a ferramenta de linha de comando `spark-submit`, o Spark permite transformar aplicações interativas em produtivas da seguinte forma: a ferramenta permite enviar o código da aplicação construído para execução em um cluster. Na submissão, a aplicação irá executar até que seja encerrada (*task completa*) ou até que encontre um erro.

Ao mencionar que o `spark-submit` envia o código da aplicação para execução em um cluster, entende-se que o gerenciamento poderá ser aplicado por um dos gerenciadores suportados pelo Spark, como Standalone (modo de instalação “solo”), Mesos ou YARN.

No exemplo de código abaixo, o `spark-submit` é utilizado para inicializar uma aplicação de testes contida na própria instalação do Spark que calcula os dígitos do

número pi até uma certa precisão:

```
$ ./bin/spark-submit \
--master local \
.examples/src/main/python/pi.py 10
```

## 3.2. Datasets: API *type-safed*

A API Datasets é utilizada para escrita de código estático tipado em Java ou Scala. Esta API não está disponível em Python ou R, uma vez que estas são dinamicamente tipadas.

Comparando com os DataFrames vistos anteriormente, aos quais poderiam ser definidos como uma coleção distribuída de objetos do tipo `Row`, os Datasets fornecem aos usuários a habilidade de assimilar **classes Java ou Scala** de modo manipular os dados como objetos tipados. Por exemplo, o elemento `Dataset[Person]` contém obrigatoriamente objetos da classe `Person`.

## 3.3. Structured Streaming

De forma direta, *Structured Streaming* é uma API Spark *high-level* para processamento de streaming. Um dos pontos fortes desta API é a possibilidade de extrair valor de streaming de dados sem alterações significativas em um código batch já construído.

Em um exemplo prático demonstrado no livro, dados de revenda obtidos através do repositório Github do livro serão utilizados para simular uma aplicação de *streaming* que coleta as informações, agrupa o total de vendas por hora a partir de uma função *window* e escreve o resultado em uma tabela em memória que será populada a cada *trigger* que, neste caso, foi configurado para ser baseado em um arquivo individual.

Para demonstrar a extrema facilidade em converter fluxos de preparação batch em streaming de dados, a construção das transformações do processo detalhado no

parágrafo acima foi proposta, em um primeiro momento, com base em código batch. Para tal, utiliza-se a função `read()`. Posteriormente, visando agora converter o fluxo de preparação em streaming, utiliza-se a função `readStream()`.

Como a escrita do fluxo em streaming é realizada em uma tabela em memória, é preciso executar a função `writeStream()` do DataFrame de streaming gerado após as preparações adequadas (o código para as transformações em streaming é exatamente o mesmo para transformações em batch).

## 3.4. Machine Learning e Advanced Analytics

O Spark também possibilita a criação de aplicações para solução de problemas envolvendo machine learning. Para tal, existe uma biblioteca *built-in* do Spark chamada `MLlib`, possibilitando o desenvolvimento de pré-processamento, preparações, treinamento e predições em dados.

No exemplo demonstrado no livro, um modelo de clusterização é utilizado para agrupar dados lidos e armazenados em um DataFrame do Spark. Como preparação, algumas funções nativas do Spark são importadas e aplicadas aos dados como, por exemplo, as funções `date_format` e `col` do módulo `pyspark.sql.functions`.

Vale citar que `date_format` e `col` são transformações de DataFrame. A API `MLlib` também contém uma série de transformadores a serem aplicados nos dados como, por exemplo, `StringIndexer`, `OneHotEncoder` e `VectorAssembler`, ambos importados do módulo `pyspark.ml.feature`.

Após aplicar as transformações necessárias na base, preparando os dados e construindo um *Pipeline* para padronizar as modificações propostas, é chegado o momento de treinar o modelo de clusterização. Para este propósito, o exemplo contido no livro realizou a importação da classe `KMeans` contida no módulo `pyspark.ml.clustering`, aplicou o método `fit()` para treinar e o método `computeCost()` para verificar o custo.

O passo a passo deste procedimento encontra-se detalhado no livro mas, apenas para reforçar, a intenção do autor não foi demonstrar um processo produtivo de treinamento de um modelo de machine learning, mas sim exemplificar um caso

prático de uso em uma das diversas possibilidades do Spark.

## 3.5. Lower-Level APIs e SparkR

Um termo relativamente famoso quando se fala em Spark é conhecido por *Resilient Distributed Datasets* ou RDDs. Virtualmente, tudo em Spark é construído sob os RDDs.

Por exemplo, operações em DataFrames são construídas sob RDDs e compiladas para ferramentas de baixo nível para uma extrema eficiência na execução distribuída. No geral, os usuários utilizam as APIs estruturadas do Spark e, apenas em casos específicos, os RDDs são aplicáveis em cenários práticos.

SparkR, por sua vez, é intuitivamente definida como uma ferramenta para execução de Spark em R. A ideia é proporcionar todo o kit de operações do Spark para a linguagem R, assim como a biblioteca pyspark atua no Python.

---

# 4

## Visão Geral Sobre APIs Estruturadas

O quarto capítulo do livro *Spark: The Definitive Guide* é o primeiro daquela que é chamada de **Parte II** e traz consigo conteúdos focados em APIs Estruturadas, sendo elas:

- `DataFrames`
- `Datasets`
- Tabelas SQL e Views

Em outras palavras, abrindo o leque de definições relacionadas às APIs Spark, este capítulo aborda uma visão geral sobre os três elementos que compõem as APIs estruturas e que, de certa forma, podem ser considerados os objetos mais comuns de trabalho envolvendo aplicações gerais em Spark. Aqui, será possível entender, de maneira introdutória, como o Spark atua considerando códigos escritos em cada uma das três principais APIs estruturadas: `DataFrames`, `Datasets` e `SQL`.

Como uma forma de relembrar e retomar conceitos importantes, temos o Spark como modelo de programação distribuída ao qual especifica-se

**transformações.** Múltiplas transformações compõem uma DAG (*Directed Acyclic Graph*) de instruções. Uma **ação** inicia o processo de execução do grafo de instruções, como um *job* único, separando-o em estágios e tarefas que executam em um cluster. A estrutura lógica manipulada com as transformações e ações são os **DataFrames** e os **Datasets**. Para criar um novo DataFrame ou Dataset, é preciso chamar uma **transformação**. Para iniciar a computação, é preciso chamar uma **ação**.

## 4.1. DataFrames e Datasets

DataFrames e Datasets nada mais são que **coleções estruturadas** e distribuídas com linhas e colunas bem definidas. Para o Spark, ambas as estruturas representam planos imutáveis, avaliados e de comportamento *lazily* que especificam **operações** a serem aplicadas aos dados de modo a gerar algum *output*.

Quando uma **ação** é performada em um DataFrame, o Spark é instruído a aplicar as **transformações** codificadas para retorno do resultado. Esta sequência de operações representam **planos** de manipulação das linhas e colunas de acordo com o resultado esperado pelo usuário.

É importante citar que `tabelas` e `views` representam a mesma coisa que DataFrames. A principal diferença é que, ao invés de realizar as transformações utilizando métodos e funções aplicadas aos DataFrames, utiliza-se `comandos SQL` para geração dos planos de execução.

Por fim, vale citar que a principal diferença entre DataFrames e Datasets é que, no primeiro, apesar de ser chamado de *untyped*, os tipos primitivos existem e só são checados pelo Spark em tempo de execução. Já o segundo, esta checagem é realizada em tempo de compilação. Datasets estão disponíveis apenas em Scala e Java e, na grande maioria das vezes, os usuários utilizam DataFrames que, por sua vez, são simplesmente Datasets do tipo `Row` (tipo interno do Spark que representa um formato otimizado para computação em memória).

## 4.2. Schemas e Tipos Primitivos do Spark

Schemas são uma forma de definir os tipos primitivos dos dados armazenados em uma das coleções distribuídas discutidas acima. Basicamente, schemas definem nomes de colunas e tipos primtiivos e podem ser definidos manualmente ou lidos diretamente da fonte de dados (este último procedimento é conhecido por *schema on read*).

Internamente, o Spark utiliza uma *engine* chamada **Catalyst** que mantém **informações próprias sobre tipos primitivos** através do planejamento e processamento do código. Com isso, o Spark se aproveita de otimizações significantes ao mesmo tempo em que garante que as linguagens de programação, como Scala, Python e R, utilizem seus próprios tipos primitivos durante a etapa de configuração e programação dos processos de ETL. Neste formato, mesmo que toda a programação seja realizada utilizando APIs em outras linguagens, a manipulação irá operar em tipos primitivos nativos do Spark (e não em tipos primitivos das linguagens utilizadas nas transformações).

A coordenação entre tipos primitivos das linguagens utilizadas nas transformações e os tipos primitivos nativos do Spark ocorre através de um mapeamento interno e é gerenciado pela engine Catalyst.

### 4.2.1. Tipo *Column* e *Row*

De forma intuitiva, o tipo *Column* no Spark representa colunas em uma tabela ou coleção distribuída. Colunas podem representar diferentes tipos primitivos e uma série de transformações podem ser aplicadas. Já o tipo *Row* é basicamente um registro na base de dados.

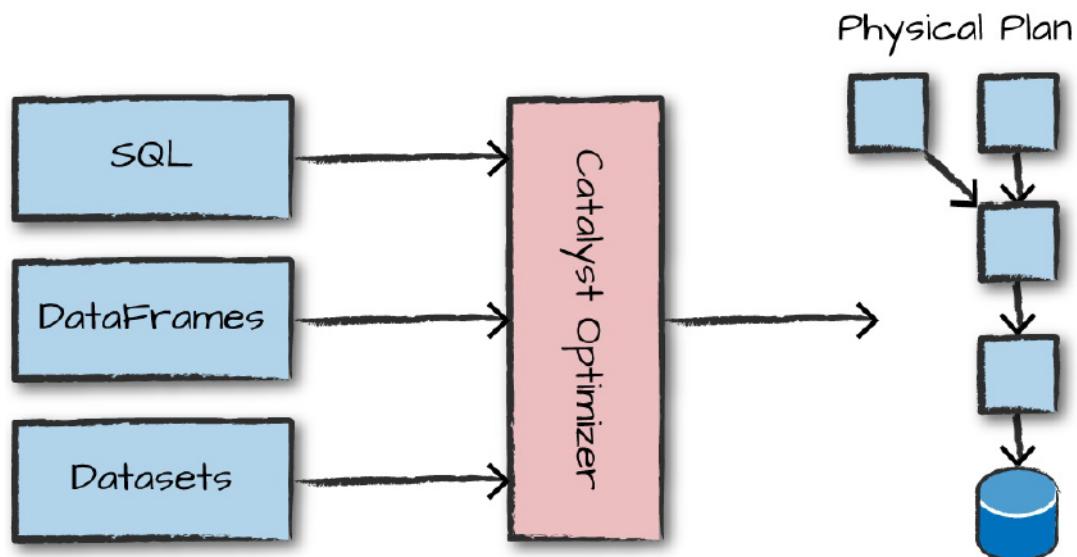
## 4.3. Execução de uma API Estruturada

Ao escolher o Spark como ferramenta de transformação de dados, é preciso definir um código associado à manipulação a ser aplicada em dados coletados. Entretanto,

o que ocorre após a codificação das transformações e operações em uma coleção distribuída (ex: DataFrame)? De maneira geral, o Spark realiza as seguinte operações até retornar os resultados aos usuários:

1. Escrita de código para manipulação dos dados (DataFrame/Dataset/SQL)
2. Se o código for válido, o Spark o converte para um *Plano Lógico*
3. O Spark transforma o *Plano Lógico* em um *Plano Físico*, procurando por otimizações ao longo do processo
4. O Spark, então, executa o *Plano Físico* (manipulações RDD - Resilient Distributed Datasets) no cluster

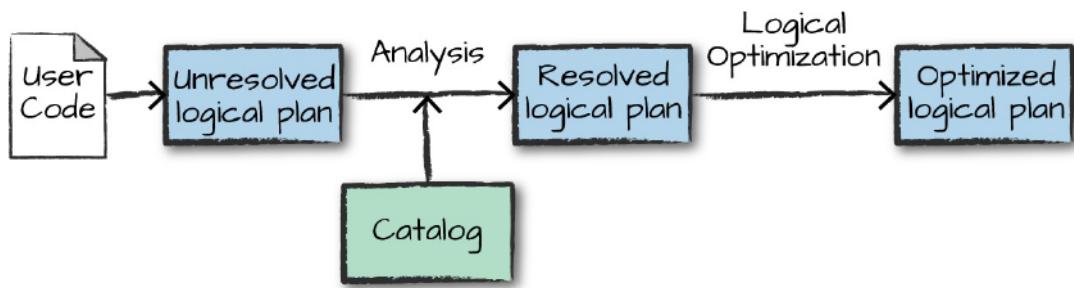
A figura 4.1 ilustra o processo seguido pelo Spark desde a obtenção do código, escrito pelo usuário, até a submissão e a construção do plano de execução. Como detalhe adicional, é possível visualizar na figura um bloco de otimização proporcionado pela engine *Catalyst* que, por sua vez, é responsável por procurar meios de otimizar a execução dos comandos configurados pelo usuário via DataFrames, Datasets ou SQL.



**Figure 4.1:** Transformação de código escrito em Spark em um Plano Físico de execução após otimizações proporcionadas pela engine interna do Spark chamada Catalyst.

## 4.4. Plano Lógico

Como informado anteriormente, a primeira fase de execução de aplicações Spark envolvem a conversão do código escrito pelo usuário em um **Plano Lógico**. A figura 4.2 ilustra esta conversão e contempla alguns blocos fundamentais no entendimento do processo de execução. Na prática, um Plano Lógico representa um conjunto abstrato de transformações que não se referem a executores ou *drivers*: trata-se apenas da versão das expressões escritas pelo usuário em sua forma mais otimizada.



**Figure 4.2:** Representação do Plano Lógico como uma conversão otimizada do código escrito pelo usuário contendo transformações Spark a serem aplicadas.

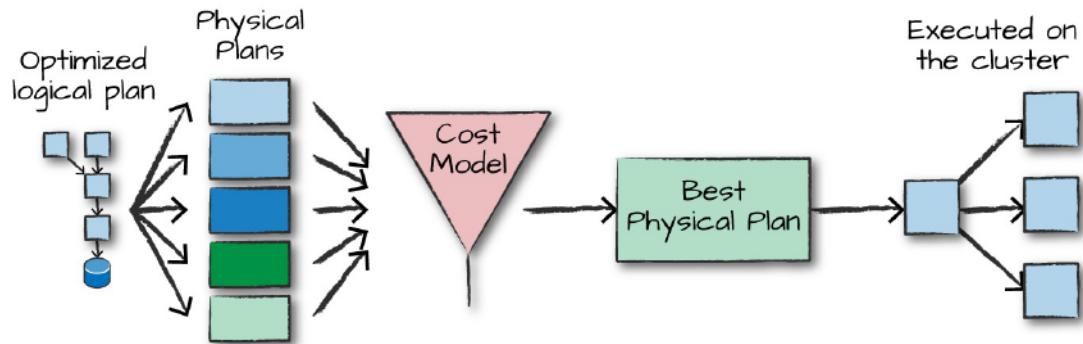
Nota-se que o Spark utiliza um **catálogo** como um repositório com informações de todas as tabelas e DataFrames utilizado para resolver linhas e colunas em um **analisador** que, por sua vez, pode rejeitar o plano lógico se as tabelas ou DataFrames requisitados nas transformações do usuário não existirem.

Uma vez analisadas as tabelas e DataFrames envolvidos, o “plano resolvido” (representado na figura 4.2 pelo bloco *Resolved logical plan*) passa por um otimizador do **Catalyst** onde aplicam-se uma série de regras visando garantir a construção do melhor plano possível (bloco *Optimized logical plan*).

## 4.5. Plano Físico

Uma vez construído um plano lógico otimizado para execução das instruções Spark codificadas pelo usuário, é iniciado o processo de definição de um **Plano Físico**. A figura 4.3 ilustra este processo também conhecido simplesmente como *Spark plan* onde diferentes planos físicos são gerados a partir de diferentes estratégias. Os

planos, então, são comparados entre si através de um **modelo de custo** que atua de modo a selecionar o plano que contém as melhores estratégias de execução.



**Figure 4.3:** Representação do Plano Físico como um conjunto de instruções e transformações RDDs para execução do código do usuário no cluster

Ao final, o melhor plano físico escolhido (representado na figura 4.3 pelo bloco *Best Physical Plan*) é então utilizado para execução das instruções no cluster.

O Plano Físico resulta em uma série de **transformações RDDs** e, por conta disso, é possível encontrar em outras referências uma associação do Spark a um compilador: no fundo, as queries e operações em DataFrames e Datasets são compiladas em transformações RDDs (interface de baixo nível do Spark), proporcionando assim otimizações em tempo de execução e gerando Java bytecode nativo que pode remover tarefas ou estágios inteiros de execução. Por fim, o resultado é então retornado ao usuário.

---

# 5

## Operações Básicas Estruturadas

Após uma análise estrutural das APIs e dos planos de execução criados pelo Spark, este capítulo comporta as operações fundamentais utilizadas para manipulação de dados em DataFrames. Relembrando os conceitos:

Um DataFrame consiste em uma série de **registros** (como linhas em uma tabela), do tipo Row, e um número de **colunas** que representam expressões computacionais que podem ser performadas em cada registro individual. **Schemas** definem os nomes e também os tipos primitivos de cada coluna. **Particionar** um DataFrame significa definir um layout de distribuição física dos dados no cluster. O **esquema de particionamento** define como isso será alocado e pode ser configurado de forma não determinística ou de acordo com valores presentes em uma dada coluna.

A criação de um DataFrame, em Spark, pode ser feita, por exemplo, através da leitura de uma base de dados salva localmente. A figura 5.1 utiliza alguns métodos nativos do objeto de sessão `spark` da biblioteca Python `pyspark` para ler um arquivo `json` presente no HD da máquina do usuário executor do código. Adicionalmente, o

método `DataFrame.printSchema()` pode ser utilizado para retornar informações formatadas sobre o *schema* dos dados.

```
In [5]: # Lendo arquivo
df = spark.read.format('json').load(DATA_PATH)

# Printando schema
df.printSchema()

root
| -- DEST_COUNTRY_NAME: string (nullable = true)
| -- ORIGIN_COUNTRY_NAME: string (nullable = true)
| -- count: long (nullable = true)
```

**Figure 5.1:** Leitura de dados salvos localmente a partir da execução dos métodos `read`, `format` e `load`. O método `printSchema()` retorna um *print* formatado do *schema* dos dados.

## 5.1. Schema

Como detalhado logo acima, o *schema* define nome de colunas e tipos primitivos de um DataFrame. É possível permitir que a própria fonte defina o *schema* (*schema-on-read*) ou definir manualmente de maneira explícita. Em termos gerais, trabalhar com *schema-on-read* pode ser mais fácil em análises *ad-hoc* mas, em ambientes produtivos, definir explicitamente o *schema* pode ser considerada uma boa prática, visto que, neste cenário, são evitados possíveis problemas relacionados a tipos primitivos assinalados incorretamente para algumas colunas.

Após a criação de um DataFrame em Spark, é possível ainda analisar, em detalhes, o *schema* através do atributo `DataFrame.schema`. A figura 5.2 traz o retorno do *schema* no formato trabalhado pelo Spark: na prática, o *schema* é um tipo conhecido como `StructType` composto por uma série de campos, `StructField`, que possuem um **nome**, um **tipo** e uma **flag booleana** que especifica se a coluna pode conter valores nulos. Opcionalmente, os usuários podem armazenar informações sobre a coluna como metadados ou comentários adicionais.

```
In [19]: # Atributo schema do DataFrame
df.schema

out[19]: structType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField(ORIGIN_COU
nt,LongType,true)))

In [20]: # Verificando campos
df.schema.fields

out[20]: [StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true)]
```

**Figure 5.2:** Representação do *schema* de um DataFrame através do Spark.

Já na figura 5.3, objeto StructType é criado manualmente para armazenar os metadados da base lida. É importante citar que os tipos primitivos são configurados utilizando tipos nativos do Spark importados do módulo `pyspark.sql.types`.

```
In [26]: # Importando tipos primitivos Spark
from pyspark.sql.types import StructField, StructType, StringType, LongType

# Criando metadados manualmente
DF_SCHEMA = StructType([
    StructField("DEST_COUNTRY_NAME", StringType(), True),
    StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
    StructField("count", LongType(), False, metadata={"hello": "world"})
])

# Lendo dados com schema explícito
df = spark.read.format("json")\
    .schema(DF_SCHEMA)\n    .load(DATA_PATH)
```

**Figure 5.3:** Definição manual do *schema* de uma base de dados a partir de tipos nativos do Spark.

## 5.2. Colunas e Expressões

Colunas no Spark são similares a colunas em uma planilha Excel ou em um DataFrame do Pandas. É possível selecionar, manipular e remover colunas de DataFrames Spark e, à estas operações, dá-se o nome de **expressões**.

Provavelmente as duas formas mais intuitivas de se referir a colunas no Spark é a partir das funções `col` e `column`. Para utilizá-las, basta realizar as devidas importações e passar uma referência de coluna como argumento. A figura 5.4 ilustra este

processo.

```
In [31]: # Importando funções
from pyspark.sql.functions import col, column

# Referenciando colunas
col('algum_nome_de_coluna')
column('algum_nome_de_coluna')

Out[31]: Column<'algum_nome_de_coluna'>
```

**Figure 5.4:** Referenciando colunas no Spark utilizando as funções `col` ou `column` do módulo `pyspark.sql.functions`

Neste ponto, vale reforçar que as colunas referenciadas pelas funções citadas **podem ou não** existir no DataFrame. Para entender este conceito, é preciso lembrar que as colunas não são *resolvidas* até que haja uma comparação entre os nomes fornecidos e aqueles contidos no *catálogo*. As resoluções de colunas e tabelas ocorrem na fase de *analyzer* (ver figura 4.2).

Já as `expressões` são um *set* de transformações em um ou mais valores em um registro contido em um DataFrame. Da forma mais direta possível, uma expressão é dada quando se cria um valor único para cada registro no dataset a partir de um input que pode ser uma ou mais colunas. Uma fórmula seria uma associação razoavelmente coerente para as expressões. O exemplo mais básico é uma expressão que reflete basicamente uma outra coluna. Em outras palavras, `expr("coluna")` é equivalente a `col("coluna")`.

Um ponto extremamente importante é que transformações em colunas podem ser realizadas tanto utilizando operações com a função `col()` quanto com a função `expr()`. Na figura 5.5, ambas as transformações resultam no mesmo resultado e isso pode ser explicado por dois principais pontos:

- Colunas nada mais são que expressões
- Colunas e transformações das colunas compilam para o mesmo plano lógico como “expressões traduzidas”

```
In [37]: # Importando funções
from pyspark.sql.functions import col, expr

# Transformação via "col"
(((col("column_A") + 5) * 200) - 6) < col("column_B")

# Transformação via "expr"
expr("((column_A + 5) * 200) - 6 < column_B")

Out[37]: column<'((((column_A + 5) * 200) - 6) < column_B)'>
```

**Figure 5.5:** Aplicando operações e realizando transformações utilizando referência de colunas via `col` ou expressões em string via `expr`

Este é um ponto extremamente importante de ser reforçado:

As transformações aplicadas via `expr()`, na figura 5.5, se assemelham muito à sintaxe `SQL`. De fato, seria algo que caberia facilmente em um comando `SELECT` que, no fundo, equivale à mesma operação escrita a partir da referência de colunas via função `col()`. Isto ocorre pois expressões SQL e transformações em DataFrames compilam para a mesma árvore lógica de execução, permitindo com que ambas a abordagens sejam aplicadas sem nenhuma alteração de performance.

## 5.3. Linhas e Registros

No Spark, cada linha em um DataFrame representa um registro único do tipo `Row`. A manipulação de objetos do tipo `Row` é feita por `expressões`.

A criação de linhas pode ser feita manualmente a partir da instância do objeto `Row` com valores que façam sentido para o usuário. Como linhas individuais não armazenam `schema`, caso os valores utilizados na criação manual da linha sejam utilizados para adicionar dados a um DataFrame já definido, a ordem e o tipo primitivo do `schema` do DataFrame existente devem ser respeitados.

A figura 5.6 ilustra como objetos do tipo `Row` normalmente se mostram aos usuários. Adicionalmente, a indexação em Python é utilizada para acessar valores

específicos em um registro retornado. Por fim, a criação manual de linhas é demonstrada através da instância do objeto Row importado do módulo pyspark.sql.

```
In [43]: # Verificando linhas
print(df.first())

# Acessando valores
print()
print(f'Dest: {df.first()[0]}')
print(f'Orig: {df.first()[1]}')
print(f'Count: {df.first()[-1]}')

Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15)

Dest: United States
Orig: Romania
Count: 15

In [44]: # Criando linhas manualmente
from pyspark.sql import Row

row = Row('Brasil', 'United States', 10)

out[44]: <Row('Brasil', 'United States', 10)>
```

**Figure 5.6:** Exemplo de atividades realizadas no objeto Row do Spark para acesso a registros em um DataFrame.

## 5.4. Transformações em DataFrames

Após a apresentação de elementos extremamente relevantes no universo de DataFrames, é chegado o momento de explorar em detalhes diferentes situações relacionadas a **transformação** de dados. De modo a facilitar o entendimento dos tópicos a serem explorados deste ponto em diante, a tabela 5.1 retoma alguns dos conceitos tratados até o momento:

Em termos de transformações possíveis em DataFrames, pode-se:

- Adicionar linhas ou colunas
- Remover linhas ou colunas
- Transformar linhas em colunas (e vice e versa)
- Alterar a ordem de linhas baseada em valores de colunas

**Table 5.1:** Resumo de Tópicos 1 - DataFrames

Elemento	Descrição
col()	Função para referenciar colunas em DataFrames
expr()	Forma alternativa para referenciar colunas ou criar expressões em um formato de string
Row()	Objeto que representa um registro em um DataFrame Spark

### 5.4.1. Criando DataFrames

Na seção anterior, foi realizada a leitura de uma base de dados salva localmente para a criação de um objeto DataFrame no Spark. Em um cenário alternativo e, de certa forma, ilustrativo, é possível definir um *schema* (assim como na figura 5.3) e criar objetos do tipo Row() que condizem com o *schema* definido para, então, construir um DataFrame através do método spark.createDataFrame(). As figuras 5.7 e 5.8 ilustram, respectivamente, a criação de DataFrames através da leitura de dados via spark.read.format().load() e a criação de DataFrames manualmente.

```
# Lendo base de dados
df = spark.read.format('json').load(DATA_PATH)
df.show(5)

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States| Romania| 15|
| United States| Croatia| 1|
| United States| Ireland| 344|
| Egypt| United States| 15|
| United States| India| 62|
+-----+-----+-----+
only showing top 5 rows
```

**Figure 5.7:** Criação de um DataFrame Spark a partir da leitura de uma base de dados local.

Após uma visão geral sobre a obtenção de DataFrames no Spark, é possível navegar em dois dos métodos mais utilizados para trabalhar com colunas e expressões: select e selectExpr.

```

# Importando módulos
from pyspark.sql.types import StructType, StructField, StringType, LongType
from pyspark.sql import Row

# Criando schema manualmente
MY_SCHEMA = StructType([
    StructField("col_1", StringType(), True),
    StructField("col_2", StringType(), True),
    StructField("col_3", LongType(), False)
])

# Criando registro manualmente
MY_ROW = Row("Hello", "World", 1)

# Criando DataFrame manualmente
df = spark.createDataFrame([MY_ROW], MY_SCHEMA)
df.show()

```

col_1	col_2	col_3
Hello	World	1

**Figure 5.8:** Criação manual de um DataFrame a partir da definição de linhas com o objeto Row()

### 5.4.2. select e selectExpr

Em linhas gerais, o método `select()` pode ser utilizado em DataFrames para criar operações de manipulação através de colunas e expressões. Já o método `selectExpr()` pode ser utilizada para criar expressões em formato de string.

Trazendo uma visão prática deste tópico, a figura 5.9 ilustra exemplos de utilização do método `select()` aplicado a um DataFrame Spark para seleção de uma ou múltiplas colunas. Adicionalmente, os comandos análogos na linguagem SQL são também evidenciados.

A referência a **colunas** de um DataFrame pode ser feita de diferentes formas (`col()`, `column()` e `expr()`, por exemplo). Para demonstrar este conceito, a figura 5.10 traz três diferentes exemplos de seleção de colunas utilizando as formas de referenciamento discutidas.

Na prática, a função de referenciamento de colunas mais comum é a `expr()` pois, com ela, existe um maior nível de flexibilidade em relação às operações possíveis. Na figura 5.11, é fornecido um exemplo comparativo de uma mudança de nome de coluna através da função `expr()` (semelhante à sintaxe SQL) e através da

```
# Selecionando coluna
df.select('DEST_COUNTRY_NAME').show(2)

# Análogo a:
# SELECT DEST_COUNTRY_NAME FROM df LIMIT 2
```

```
+-----+
|DEST_COUNTRY_NAME|
+-----+
|  United States|
|  United States|
+-----+
only showing top 2 rows
```

```
# Selecionando múltiplas colunas
df.select('DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME').show(2)

# Análogo a:
# SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM df LIMIT 2
```

```
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----+-----+
|  United States|          Romania|
|  United States|          Croatia|
+-----+-----+
only showing top 2 rows
```

Figure 5.9: Utilização do método select() para seleção de colunas em um DataFrame.

```
# Importando funções
from pyspark.sql.functions import col, column, expr
```

```
# Referenciando colunas
df.select(
    col('DEST_COUNTRY_NAME'),
    column('DEST_COUNTRY_NAME'),
    expr('DEST_COUNTRY_NAME')
).show(2)
```

```
+-----+-----+-----+
|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|
+-----+-----+-----+
|  United States|      United States|      United States|
|  United States|      United States|      United States|
+-----+-----+-----+
only showing top 2 rows
```

Figure 5.10: Diferentes formas de referenciar colunas em um DataFrame. Em todas elas, o resultado é o mesmo.

função `col()` (utilizando o método `alias()`).

```
# Renomeando coluna via expr()
df.select(expr('DEST_COUNTRY_NAME AS destination')).show(2)
```

```
+-----+
| destination|
+-----+
|United States|
|United States|
+-----+
only showing top 2 rows
```

```
# Renomeando coluna via col()
df.select(col('DEST_COUNTRY_NAME').alias('destination')).show(2)
```

```
+-----+
| destination|
+-----+
|United States|
|United States|
+-----+
only showing top 2 rows
```

**Figure 5.11:** Renomeando uma coluna via `select()` através das funções `expr()` e `col()`. No primeiro cenário, o novo nome para a coluna pode ser dado em uma sintaxe estritamente semelhante ao SQL. No segundo cenário, este processo precisa ser realizado através da execução do método `alias()`

Como a utilização do método `select` seguido por uma série de funções `expr` é considerado um padrão extremamente comum em manipulações de DataFrame em Spark, criou-se o método `selectExpr` como uma forma de facilitar a construção de expressões de seleção. Este é provavelmente o jeito mais intuitivo para usos diários de operações deste tipo em DataFrames Spark. A figura 5.12 exemplifica sua utilização comparando a seleção de múltiplas colunas (renomeando com `alias`) utilizando `select` com múltiplos `expr()` e, depois, apenas a função `selectExpr`.

Na prática, o método `selectExpr` pode ser utilizado como uma forma simples de construir expressões complexas e criar novos DataFrames. Qualquer “comando” SQL de não agregação será válido (desde que as colunas existam). Em um outro exemplo, representado pela figura 5.13, o método `selectExpr` é utilizado para adição de uma nova coluna com uma base em uma operação de validação de igualdade. Na figura, vale citar a presença do asterisco (\*) como uma forma de selecionar todas as

```
# Selecionando múltiplas colunas com select + expr
df.select(
    expr('DEST_COUNTRY_NAME AS destination'),
    expr('ORIGIN_COUNTRY_NAME AS origin')
).show(2)
```

```
+-----+-----+
| destination| origin|
+-----+-----+
|United States|Romania|
|United States|Croatia|
+-----+-----+
only showing top 2 rows
```

```
# Selecionando múltiplas colunas com selectExpr
df.selectExpr('DEST_COUNTRY_NAME AS destination',
               'ORIGIN_COUNTRY_NAME AS origin').show(2)
```

```
+-----+-----+
| destination| origin|
+-----+-----+
|United States|Romania|
|United States|Croatia|
+-----+-----+
only showing top 2 rows
```

**Figure 5.12:** Exemplo da facilidade proporcionada pelo método `selectExpr` na dinâmica geral de seleção de múltiplas colunas com a capacidade de adicionar operações e outras manipulações possíveis. Este procedimento se assemelha muito à sintaxe SQL.

colunas existentes no DataFrame e, diferente do combo `select` seguido por múltiplos `expr()`, o método `selectExpr` proporciona uma maior facilidade em referenciar as colunas.

### 5.4.3. Literais

De forma direta, a inclusão de **literais** em um DataFrame Spark pode ser realizada através da função `lit()` importada do módulo `pyspark.sql.functions`. A figura 5.14 ilustra um exemplo de adição de coluna em um DataFrame com a presença de um literal.

```
# Adicionando colunas
df.selectExpr(
    "*", # todas as colunas
    "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) AS within_country"
).show(2)

# Em SQL:
# SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) AS within_country
# FROM df LIMIT 2

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|within_country|
+-----+-----+-----+
| United States| Romania| 15| false|
| United States| Croatia| 1| false|
+-----+-----+-----+
only showing top 2 rows
```

**Figure 5.13:** Exemplo de adição de colunas em um DataFrame utilizando o método `selectExpr` em um formato bem semelhante à sintaxe SQL.

```
# Importando função
from pyspark.sql.functions import lit

# Adicionando coluna com literal (select)
df.select(expr("*"), lit(1).alias("One")).show(2)

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----+-----+-----+
| United States| Romania| 15| 1|
| United States| Croatia| 1| 1|
+-----+-----+-----+
only showing top 2 rows

# Adicionando coluna com literal (selectExpr)
df.selectExpr("*", "1 AS One").show(2)

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----+-----+-----+
| United States| Romania| 15| 1|
| United States| Croatia| 1| 1|
+-----+-----+-----+
only showing top 2 rows
```

**Figure 5.14:** Exemplos de adição de literais em um DataFrame Spark através dos métodos `select` e `selectExpr`

### 5.4.4. Adicionando Colunas

Nos últimos exemplos ilustrados pelas figuras 5.13 e 5.14, novas colunas foram adicionadas à DataFrames utilizando os métodos `select` ou `selectExpr`. Nestes cenários, uma exigência para adição de novas colunas foi referenciar o asterisco (\*) para indicar a seleção de todas as colunas já existentes no DataFrames antes da inclusão (isso é feito via `expr()`, `col()` ou `column()` no caso do método `select` ou via strings no caso do método `selectExpr`).

Por mais intuitivo que possa parecer, esta pode ser uma ação facilmente ignorada por programadores. Pensando nisso, o Spark propõe a utilização do método `withColumn()` para adição de novas colunas em DataFrame de uma forma mais simples e direta. A figura 5.15 ilustra um exemplo de adição de literais e de um flag booleano com base em uma expressão.

```
# Adição de literais
df.withColumn("One", lit(1)).show(2)

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----+-----+-----+
| United States| Romania| 15| 1|
| United States| Croatia| 1| 1|
+-----+-----+-----+
only showing top 2 rows
```

```
# Adição de flags
df.withColumn("within_country", expr("(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME)").show(2)

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|within_country|
+-----+-----+-----+
| United States| Romania| 15| false|
| United States| Croatia| 1| false|
+-----+-----+-----+
only showing top 2 rows
```

**Figure 5.15:** Exemplos de adição de colunas com o método `withColumn()`

### 5.4.5. Renomando Colunas

Ao explorar o método `withColumn()` e seus dois argumentos (nome da coluna e expressão), percebe-se a possibilidade de utilizá-lo para renomear colunas existentes. Neste cenário, bastaria fornecer o nome da nova coluna no primeiro argumento e a

referência (via `expr()`, `col()` ou `column()`) à uma coluna já existentes no DataFrame.

Apesar de parecer uma forma simplificada de alterar nomes de colunas, esta solução **replica** os dados de uma coluna já existente com outro nome, mantendo a coluna original no DataFrame e, por consequência, uma redundância dos dados (até que algum outro método de exclusão de colunas fosse executado para eliminar a coluna “original”).

Para operações de renomeação de colunas, o Spark proporciona o método `withColumnRenamed()` que também recebe dois argumentos: o primeiro, representando o nome da coluna já existente a ser renomeada e, o segundo, o novo nome a ser direcionado para a coluna. A figura 5.16 ilustra uma aplicação do tipo.

```
# Renomeando colunas
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").show(2)

+-----+-----+-----+
|      dest|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
|United States|          Romania|    15|
|United States|          Croatia|     1|
+-----+-----+-----+
only showing top 2 rows
```

**Figure 5.16:** Exemplos de alteração de nomes de colunas com o método `withColumnRenamed()`

## 5.4.6. Removendo Colunas

A remoção ou exclusão de colunas em um DataFrame Spark, de forma direta, é realizada através do método `drop()` do objeto DataFrame. A figura 5.17 ilustra um exemplo de exclusão de uma coluna.

## 5.4.7. Casting

Em alguns cenários, é preciso alterar o tipo primitivo de colunas existentes em um DataFrame. A figura 5.18 exemplifica a conversão do tipo primitivo da coluna `count`, originalmente `long`, para o tipo `integer` utilizando o método `cast()` após a referência da coluna (feito via `col()`).

```
# Removendo uma coluna
df.drop("DEST_COUNTRY_NAME").show(2)
```

```
+-----+-----+
|ORIGIN_COUNTRY_NAME|count|
+-----+-----+
|          Romania|   15|
|          Croatia|    1|
+-----+-----+
only showing top 2 rows
```

```
# Removendo múltiplas colunas
df.drop("ORIGIN_COUNTRY_NAME", "count").show(2)
```

```
+-----+
|DEST_COUNTRY_NAME|
+-----+
|      United States|
|      United States|
+-----+
only showing top 2 rows
```

Figure 5.17: Eliminando colunas em um DataFrame com o método drop()

```
print('Schema original:')
df.printSchema()

# Criando nova coluna com tipo primitivo alterado
df_cast = df.withColumn("count_2", col("count").cast("int"))
print('Schema após casting:')
df_cast.printSchema()
```

```
Schema original:
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: long (nullable = true)
```

```
Schema após casting:
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: long (nullable = true)
 |-- count_2: integer (nullable = true)
```

Figure 5.18: Adicionando nova coluna e convertendo tipo primitivo via cast()

Já a figura 5.19 exemplifica um cenário estritamente semelhante, porém utilizando uma expressão de *casting* via `expr()`, nos moldes da sintaxe SQL, convertendo o tipo da coluna `count` para `string` em uma nova coluna chamada `count_3`.

```
# Alterando schema via selectExpr
df_cast2 = df_cast.withColumn("count_3", expr("cast(count AS STRING)"))
df_cast2.printSchema()
```

```
root
| -- DEST_COUNTRY_NAME: string (nullable = true)
| -- ORIGIN_COUNTRY_NAME: string (nullable = true)
| -- count: long (nullable = true)
| -- count_2: integer (nullable = true)
| -- count_3: string (nullable = true)
```

**Figure 5.19:** Adicionando nova coluna e convertendo tipo primitivo via `expr()` em uma sintaxe semelhante ao SQL

#### 5.4.8. Filtrando Registros

De forma intuitiva, a filtragem de registros em DataFrames Spark ocorre a partir da criação de expressões que avaliam em `true` ou `false`. Para isso, é possível utilizar dois métodos: `where()` ou `filter()`. Ambos levam ao mesmo resultado e também aceitam, como lógica de filtragem, referências de colunas ou expressões em String.

A figura 5.20 exemplifica a filtragem de registros utilizando o método `where()` (mais intuitivo e próximo à linguagem SQL) em cenários de referência de colunas e em expressões String.

Em SQL, para filtrar registros utilizando múltiplas condições, aplica-se funções lógicas como, por exemplo, E e OU. Apesar do Spark também abrir este tipo de possibilidade com o método `where()`, na execução da tarefa, todos os filtros são aplicados de forma simultânea. Dessa forma, pode-se encadear múltiplas chamadas do `where()` para aplicar filtros com mais de uma condição lógica. A figura 5.21 demonstra um exemplo de aplicação.

```
# Filtrando registros (referência de colunas)
df.where(col("count") < 2).show(2)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Croatia	1
United States	Singapore	1

only showing top 2 rows

```
# Filtrando registros (expressões em String)
df.where("count < 2").show(2)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Croatia	1
United States	Singapore	1

only showing top 2 rows

**Figure 5.20:** Filtrando registros utilizando referência de colunas e expressões em String.

```
# Aplicando múltiplos filtros (forma não usual)
df.where("count < 2 AND ORIGIN_COUNTRY_NAME != 'Croatia'").show(2)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Singapore	1
Moldova	United States	1

only showing top 2 rows

```
# Aplicando múltiplos filtros (forma usual)
df.where("count < 2")\
  .where("ORIGIN_COUNTRY_NAME != 'Croatia'")\
  .show(2)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Singapore	1
Moldova	United States	1

only showing top 2 rows

**Figure 5.21:** Filtrando registros a partir de múltiplas condições lógicas utilizando expressões SQL e encadeamento de métodos `where()`.

### 5.4.9. Removendo Duplicatas

A remoção de duplicatas de registros contidos em um DataFrame Spark pode ser feita a partir da execução do método `distinct()`. Ao ser utilizado em conjunto com métodos de seleção de colunas (como por exemplo, o `select`), o método `distinct()` pode ser aplicado a uma ou mais colunas de uma vez.

Na figura 5.22, é demonstrado dois exemplos de aplicação: o primeiro, removendo duplicatas de uma única coluna e, o segundo, removendo duplicatas de duas colunas simultaneamente. Em ambos, o método `count()` auxilia na verificação da efetividade do processo.

```
# Removendo duplicatas em uma coluna
df.select('DEST_COUNTRY_NAME').distinct().count()
```

132

```
# Removendo duplicatas em duas colunas
df.select('DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME').distinct().count()
```

256

**Figure 5.22:** Remoção de duplicatas em uma ou mais colunas de um DataFrame.

### 5.4.10. Amostragem

Em algumas situações, pode ser preciso analisar algumas amostras aleatórias de um DataFrame. Para isso, pode-se utilizar o método `sample()` com os argumentos `withReplacement` (configura se a amostragem será com reposição), `fraction` (configura a fração de dados a ser amostrada) e `seed` (configura uma semente aleatória). A figura 5.23 ilustra essa aplicação.

```
# Realizando amostragem em um DataFrame
df.sample(withReplacement=False, fraction=0.5, seed=42).count()
```

132

**Figure 5.23:** Amostrando registros de um DataFrame de forma aleatória.

Adicionalmente, uma outra possibilidade envolvendo amostragem é a separação

de um DataFrame em diferentes “conjuntos” ou “blocos”. Algoritmos de Machine Learning normalmente utilizam esta filosofia para obter dados de treino e teste a partir de uma base única. Em Spark, este processo pode ser realizado pelo método `randomSplit()` que, por sua vez, recebe uma lista como argumento responsável por configurar as proporções de separação a serem aplicadas. A figura 5.24 exemplifica sua aplicação.

```
# Separando DataFrame em blocos distintos
dataframes = df.randomSplit(weights=[0.25, 0.75], seed=42)
print(f'DF Original: {df.count()}')
print(f'DF[0]: {dataframes[0].count()}')
print(f'DF[1]: {dataframes[1].count()}'')
```

DF Original: 256  
DF[0]: 63  
DF[1]: 193

Como a função `randomSplit()` retorna uma lista composta por DataFrames de acordo com os pesos passados, seu retorno pode já ser associado a novos DataFrames, evitando assim a necessidade de realizar indexação.

```
# Separando e associando DataFrames
train, test = df.randomSplit(weights=[0.25, 0.75], seed=42)
print(f'train: {train.count()}')
print(f'test: {test.count()}'')
```

train: 63  
test: 193

**Figure 5.24:** Separando um DataFrames em blocos distintos de acordo com proporções definidas.

No primeiro bloco de código da imagem, o retorno do método `randomSplit()` é associado a um bloco único (lista) composto por DataFrames separados de acordo com a proporção definida. No segundo bloco de código, o retorno já é associado a diferentes DataFrames chamados de `train` e `test`, exemplificando essa possibilidade em uma situação prática.

### 5.4.11. Unindo Registros

Em tópicos anteriores, foi reforçado que DataFrames são **imutáveis**. Isto significa que usuários não podem realizar **append** em DataFrames, pois isto implicaria em uma alteração. Para realizar este processo, é preciso unir um DataFrame original à um novo DataFrame garantindo que ambos tenham o **mesmo schema** e o mesmo

número de **colunas**. Caso contrário, o processo irá falhar.

A figura 5.25 ilustra a criação de novas linhas manualmente a partir da função `Row()` (já explorada anteriormente). A construção de um DataFrame é realizada a partir da “paralelização” das novas linhas criadas através do método `parallelize()` e a subsequente criação do DataFrame via `spark.createDataFrame()`.

```
from pyspark.sql import Row

# Criando registros manualmente
newRows = [
    Row('Brazil', 'Canada', 5),
    Row('Brazil', 'Argentina', 1)
]

# Construindo um novo DataFrame
originalSchema = df.schema
parallelizedRows = spark.sparkContext.parallelize(newRows)
newDF = spark.createDataFrame(parallelizedRows, originalSchema)

# Unindo DataFrames e realizando consulta
df.union(newDF)\n    .where(col("DEST_COUNTRY_NAME") == "Brazil").show()

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
|          Brazil|      United States|   853|
|          Brazil|              Canada|     5|
|          Brazil|        Argentina|     1|
+-----+-----+-----+
```

**Figure 5.25:** Exemplo de união de DataFrames a partir da criação manual de um DataFrame novo e a concatenação com um DataFrame já existente. DataFrames são imutáveis e, dessa forma, não é possível realizar alterações na estrutura de um DataFrame já existente. No exemplo, duas linhas com *DEST\_COUNTRY\_NAME* igual a “Brazil” são criadas e unidas ao DataFrame original. No filtro, percebe-se que o DataFrame original já continha uma linha com *DEST\_COUNTRY\_NAME* igual a “Brazil” e, após a adição das duas linhas criadas manualmente com esta mesma condição, o resultado final traz três registros.

Ainda sobre a figura 5.25, percebe-se que, se fosse necessário realizar operações ou transformações adicionais no DataFrame de união, seria preciso atribuir o resultado do `union` a um novo DataFrame e, dali em diante, continuar o trabalho. Uma forma de realizar este processo de forma mais dinâmico é criar uma view ou tabela

temporária no Spark com o resultado do union.

#### 5.4.12. Ordenando Registros

A ordenação de registro em DataFrames Spark pode ser realizada pelos métodos `sort()` ou `orderBy()`. Ambos funcionam da mesma forma e aceitam referências de colunas ou expressões em string. Por padrão, as ordenações são realizadas de forma ascendente mas, em necessidades de ordenação de forma descendente, é possível importar e utilizar a função `desc()`, ou mesmo explicitar a ordenação descendente via função `asc()`. Ambas as funções são importadas do módulo `pyspark.sql.functions`.

A figura 5.26 ilustra exemplos de ordenação de registros utilizando uma ou mais colunas com o método `sort()`.

#### 5.4.13. Limitando Registros

De forma intuitiva, o método `limit()` pode ser aplicado em DataFrames para que apenas um número específico de registros seja utilizado em transformações posteriores da DAG. A figura 5.27 ilustra um exemplo de utilização.

#### 5.4.14. Reparticionando DataFrames

Sabe-se que o particionamento de dados é uma oportunidade de otimização de fluxos de trabalho. No Spark, assim como em diversas outras ferramentas, é possível gerenciar o particionamento de DataFrames a partir de um número fixo ou então utilizando entradas de dados presentes em uma determinada coluna do DataFrame.

Na prática, se as consultas realizadas utilizam filtros frequentes relacionados a uma ou mais colunas, pode-se analisar a utilização dessa(s) coluna(s) como partição(ões), desde que a quantidade de entradas distintas seja razoavelmente pequena.

A figura 5.28 ilustra alguns métodos utilizados para coletar a quantidade de partições de um DataFrame lido em memória, a repartição por um número fixo de par-

```
# Importando funções
from pyspark.sql.functions import asc, desc

# Ordenando registros - descendente
df.sort(col("count").desc()).show(5)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	United States	370002
United States	Canada	8483
Canada	United States	8399
United States	Mexico	7187
Mexico	United States	7140

only showing top 5 rows

```
# Ordenando registros - múltiplas colunas
df.sort(expr("count asc"), col("DEST_COUNTRY_NAME").asc()).show(5)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
Burkina Faso	United States	1
Cote d'Ivoire	United States	1
Cyprus	United States	1
Djibouti	United States	1
Indonesia	United States	1

only showing top 5 rows

**Figure 5.26:** Exemplos de ordenação de registros em um DataFrame. No primeiro bloco de código, a ordenação é realizada em uma única coluna a partir de uma referência de coluna (col()) e a função desc() para garantir uma ordem descendente. No segundo bloco de código, uma mistura de possibilidades é demonstrada para que o usuário perceba as diferentes formas de especificar a ordenação em registros, seja com uma ou mais colunas, ou utilizando referências e expressões.

tições e também a repartição utilizando uma coluna da base de dados.

#### 5.4.15. Coletando Registros no Driver

Como informado em capítulos anteriores, o Spark mantém o estado do cluster no *driver* (ver figura 2.1). Em alguns casos, pode ser necessário colher dados do processo *driver* para realizar as manipulações na própria máquina local do usuário. Ex-

```
# Limitando registros
df.sort(col("count").desc()).limit(3).show()

+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+-----+-----+
| United States|      United States| 370002|
| United States|              Canada|   8483|
|       Canada|      United States|   8399|
+-----+-----+
```

**Figure 5.27:** Exemplo de ordenação e limitação de registros em um DataFrame.

```
# Coletando o número de partições
print(f'Partições do DataFrame: {df.rdd.getNumPartitions()}')

# Reparticionando - número fixo
df_rep5 = df.repartition(5)
print(f'Partições do DataFrame: {df_rep5.rdd.getNumPartitions()}')

# Reparticionando - coluna da base
df_base5 = df.repartition(5, col("DEST_COUNTRY_NAME"))
print(f'Partições do DataFrame: {df_base5.rdd.getNumPartitions()}')


Partições do DataFrame: 1
Partições do DataFrame: 5
Partições do DataFrame: 5
```

**Figure 5.28:** Exemplo de configuração de partições em um DataFrame utilizando o método `repartition()`

emplos de métodos responsáveis por realizar esta coleta são:

- `collect()`: coleta dados de todo o DataFrame
- `take()` coleta apenas as primeiras N linhas
- `show()` printa as linhas de maneira formatada

Além dos métodos acima citados, existe o método `toLocalIterator()` responsável por coletar partições para a máquina local como um *iterator*. Em outras palavras, este método permite iterar sobre partições de um DataFrame retirado do cluster Spark direto da máquina local. A figura 5.29 ilustra algumas aplicações relacionadas a coleta de registros do cluster e tratamento em máquina local do usuário.

```
# Preparando DataFrame reduzido
df_collect = df.limit(5)

# Coletando N registros
df_collect.take(3)

[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344)]

# Coletando todos os registros
df_collect.collect()

[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344),
 Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count=62)]

# Coletando DataFrame como um iterator de partições
for i in df_collect.toLocalIterator():
    print(i)

Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15)
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1)
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344)
Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count=15)
Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count=62)
```

Figure 5.29: Exemplos de utilização de métodos responsáveis por coletar dados do cluster Spark.

Neste cenário, é importante ressaltar que:

Qualquer coleta de dados para tratamento local pode ser uma operação extremamente **custosa**. A execução do método `collect()` em um dataset de muitos registros pode facilmente ``quebrar'' o disco, assim como a iteração via `toLocalIterator()` em um número grande de partições. É preciso lembrar que, neste cenário, as operações são realizadas **individualmente** e não mais em **paralelo**.

---

# 6

## Trabalhando com Diferentes Tipos de Dados

No capítulo anterior, foram explorados diversas formas de transformação de DataFrames Spark, além do fornecimento de um entendimento geral sobre as APIs e as estruturas básicas de linhas e colunas em um DataFrame. Neste capítulo, o conhecimento abordado girará em torno da **construção de expressões** em cada um dos seguintes cenários envolvendo tipos primitivos e situações específicas:

- Booleanos
- Numéricos
- Strings
- Dates e Timestamps
- Valores nulos
- Tipos complexos
- Funções definidas pelo usuário (UDFs)

Por mais que o conteúdo abordado no livro seja um *snapshot* no tempo, os detalhes aqui inseridos visam proporcionar não só um entendimento geral sobre algumas

das expressões ou transformações de dados, mas também visam garantir uma maior autonomia ao desenvolvedor em pesquisar em documentações e selecionar as melhores formas de aplicar as necessidades existentes ao processo de ETL.

## 6.1. Leitura dos Dados

Pensando em implementar cenários práticos de construção e aplicação de expressões, uma base de dados contida no repositório do livro será lida e uma view em SQL será criada de modo a propor comparações entre transformações realizadas a partir de métodos de DataFrames e queries SQL. Para tal, a figura 6.1 ilustra a leitura da base de dados em um formato DataFrame e a criação de uma view SQL pronta para ser consultada. Adicionalmente, é possível visualizar uma amostra da base via `show()` e seu respectivo **schema** via `printSchema()`.

```
# Lendo base de dados
df = spark.read.format('csv')\
    .option('header', 'true')\
    .option('inferSchema', 'true')\
    .load(DATA_PATH)

# Criando view
df.createOrReplaceTempView('vw_retail_data')

# Verificando dados e schema
df.show(2)
df.printSchema()

+-----+-----+-----+-----+-----+-----+
|InvoiceNo|StockCode|Description|Quantity|InvoiceDate|UnitPrice|CustomerID|Country|
+-----+-----+-----+-----+-----+-----+
| 536365 | 85123A | WHITE HANGING HEA... |       6 | 2010-12-01 08:26:00 |     2.55 | 17850.0 | United Kingdom |
| 536365 | 71053  | WHITE METAL LANTERN |       6 | 2010-12-01 08:26:00 |     3.39 | 17850.0 | United Kingdom |
+-----+-----+-----+-----+-----+-----+
only showing top 2 rows

root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

**Figure 6.1:** Leitura e apresentação da base de dados a ser utilizada nos exemplos de demonstração deste capítulo.

## 6.2. Trabalhando com Booleanos

Booleanos são tipos essenciais utilizados principalmente na filtragem de dados. Expressões booleanas são basicamente formadas por 4 elementos: *and*, *or*, *true* e *false*. A figura 6.2 exemplifica aplicações de filtros utilizando referências de colunas (parte superior da figura) ou expressões (parte inferior da figura). Em complemento, a figura 6.3 referencia como as diferentes expressões são consideradas para o Spark no ato da filtragem.

```
# Filtrando dados com referência de colunas
df.where(col("InvoiceNo") != 536365) \
    .select("InvoiceNo", "Description") \
    .show(2, truncate=False)
```

```
+-----+-----+
|InvoiceNo|Description          |
+-----+-----+
|536366  |HAND WARMER UNION JACK |
|536366  |HAND WARMER RED POLKA DOT|
+-----+-----+
only showing top 2 rows
```

```
# Filtrando dados direto com expressões
df.where("InvoiceNo <> 536365") \
    .select("InvoiceNo", "Description") \
    .show(2, truncate=False)
```

```
+-----+-----+
|InvoiceNo|Description          |
+-----+-----+
|536366  |HAND WARMER UNION JACK |
|536366  |HAND WARMER RED POLKA DOT|
+-----+-----+
only showing top 2 rows
```

**Figure 6.2:** Exemplo de aplicação de filtros utilizando diferentes abordagens.

Para mais exemplos de aplicação de filtros, é possível analisar as figuras 5.20 e 5.21 consolidadas no capítulo anterior.

Ainda sobre o capítulo anterior, foi informado sobre as possibilidades de encadeamento de filtros utilizando as operações lógicas *AND* e *OR*. Como o Spark consolida todas as operações em um formato único a ser executado no cluster, para lóg-

```

expr("InvoiceNo <> 536365")
Column<'(NOT (InvoiceNo = 536365))'>

col("InvoiceNo") != 536365
Column<'(NOT (InvoiceNo = 536365))'>

```

**Figure 6.3:** Visualização sobre como o Spark considera expressões booleanas para filtragem.

icas AND, pode-se encadear filtros `where()` em uma única operação. A figura 5.21, evidenciada no capítulo anterior, traz esse conceito. A figura 6.4, presente neste capítulo, traz um outro exemplo relativamente mais “robusto”, considerando a criação de expressões booleanas de filtro e a subsequente aplicação em um DataFrame.

```

# Importando função instr
from pyspark.sql.functions import instr

# Criação de filtros
stockFilter = col("StockCode").contains("DOT")
priceFilter = expr("UnitPrice") > 600
descFilter = instr("Description", "POSTAGE") >= 1

# Lista de seleção de colunas
selectList = ["InvoiceNo", "StockCode", "Description", "UnitPrice"]

# Aplicando filtros
df.where(stockFilter)\.
    .where(priceFilter | descFilter)\.
    .select(selectList)\.
    .show()

+-----+-----+-----+-----+
|InvoiceNo|StockCode| Description|UnitPrice|
+-----+-----+-----+-----+
| 536544|      DOT|DOTCOM POSTAGE|   569.77|
| 536592|      DOT|DOTCOM POSTAGE|   607.49|
+-----+-----+-----+-----+

# Análogo em SQL
spark.sql("""
    SELECT InvoiceNo, StockCode, Description, UnitPrice
    FROM vw_retail_data
    WHERE StockCode IN ("DOT")
        AND (UnitPrice > 600 OR instr>Description, "POSTAGE") >= 1
""").show()

```

**Figure 6.4:** Aplicação de múltiplos filtros em um DataFrame considerando múltiplas lógicas booleanas (AND e OR).

No exemplo ilustrado na figura 6.4, os filtros booleanos são criados e armazenados em variáveis que, posteriormente, são aplicadas nos métodos `where()` de acordo com a lógica desejada. Na camada inferior da image, demonstra-se como obter o mesmo resultado utilizando puramente SQL.

Expressões booleanas também podem ser utilizadas para criação de colunas em um DataFrame e a subsequente filtragem. Em outras palavras, define-se uma expressão booleana com as regras desejadas (ANDs e ORs) para popular uma nova coluna (booleana) no DataFrame. No filtro `where()`, pode-se basicamente especificar esta nova coluna booleana para filtrar apenas casos onde a regra de filtro é verdadeira. A figura 6.5 ilustra esta aplicação.

```
# Filtrando dados a partir de coluna booleana
df.withColumn('isExpensive', stockFilter & (priceFilter | descFilter))\
  .where('isExpensive')\
  .select(selectList + ['isExpensive'])\
  .show()
```

InvoiceNo	StockCode	Description	UnitPrice	isExpensive
536544	DOT	DOTCOM POSTAGE	569.77	true
536592	DOT	DOTCOM POSTAGE	607.49	true

**Figure 6.5:** Exemplo de criação de coluna utilizando uma expressão booleana e a subsequente filtragem.

## 6.3. Trabalhando com Números

Em grande parte das ocasiões, as tratativas em campos numéricos se dará a partir da construção e execução de expressões computacionais, criando ou implementando cenários baseados em operações matemáticas. Em um exemplo básico, a figura 6.6 exemplifica a criação de um novo campo de quantidade na base de dados utilizada a partir de uma fórmula matemática que envolve potenciação, multiplicação e soma.

Ainda sobre a figura 6.6, vale citar que as funções e operações matemáticas realizadas contemplam:

```
# Importando funções
from pyspark.sql.functions import expr, pow, round

# Criando novo campo com base em fórmula (jeito 1)
fabricatedQuantity = round(pow(col("Quantity") * col("UnitPrice"), 2) + 5, 2)
df.select(
    "Quantity", "UnitPrice",
    fabricatedQuantity.alias("realQuantity")
).show(3)

+-----+-----+-----+
|Quantity|UnitPrice|realQuantity|
+-----+-----+-----+
|      6|     2.55|      239.09|
|      6|     3.39|      418.72|
|      8|     2.75|      489.0|
+-----+-----+-----+
only showing top 3 rows
```

```
# Criando novo campo com base em fórmula (jeito 2)
df.selectExpr(
    "Quantity", "UnitPrice",
    "round(power((Quantity * UnitPrice), 2) + 5, 2) AS realQuantity"
).show(3)

+-----+-----+-----+
|Quantity|UnitPrice|realQuantity|
+-----+-----+-----+
|      6|     2.55|      239.09|
|      6|     3.39|      418.72|
|      8|     2.75|      489.0|
+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.6:** Exemplo de aplicação de operações matemáticas em colunas numéricas para retornar valores baseados em uma ou mais colunas existentes de uma base de dados. Na primeira célula da figura, essa aplicação é realizada através da utilização de funções nativas do Spark para a criação e armazenamento da regra em uma variável Python utilizada posteriormente no método select() (um alias é adicionado de modo a providenciar um nome à coluna). Na segunda célula de código da figura, toda a construção matemática é feita através do método selectExpr() em uma linguagem estritamente semelhante ao SQL o que, em alguns cenários, pode facilitar a obtenção dos resultados necessários sem qualquer perda de performance (assunto já abordado em capítulos anteriores)

- `round()`: arredondamento de um numeral de acordo com as casas decimais fornecidas no argumento da função;
- `pow()`: aplicação da potência de N em numeral;

- As outras duas operações envolvidas são multiplicação e soma.

Em um outro exemplo de aplicação envolvendo variáveis numéricas com Spark, a figura 6.7 traz um cenário de cálculo de **correlação** entre duas variáveis a partir da função `corr`. Na figura, o cálculo é demonstrando utilizando tanto a função nativa do Spark como também uma expressão em string via `selectExpr`.

```
# Importando função
from pyspark.sql.functions import corr

# Correlação entre variáveis via select
df.select(corr("Quantity", "UnitPrice").alias("PriceQtyCorr")).show()

# Correlação entre variáveis via selectExpr
df.selectExpr("corr(Quantity, UnitPrice) AS PriceQtyCorr").show()

+-----+
|      PriceQtyCorr|
+-----+
|-0.04112314436835551|
+-----+

+-----+
|      PriceQtyCorr|
+-----+
|-0.04112314436835551|
+-----+
```

**Figure 6.7:** Exemplos de cálculo de correlação Pearson entre duas colunas numéricas de DataFrames Spark.

Continuando com os exemplos comuns envolvendo colunas numéricas, o método `describe()` de DataFrames Spark permitem computar estatísticas essenciais de colunas, incluindo a quantidade de registros, média, desvio padrão, valor mínimo e valor máximo. Por padrão, o método `describe()` considera todas as colunas do DataFrame, inclusive strings. Dessa forma, de modo a exemplificar o processo somente considerando colunas numéricas, a figura 6.8 realiza um `select` baseado em uma lista construída em Python (via *list comprehension*) contemplando apenas atributos numéricos.

Apesar do método `describe()` fornecer uma forma fácil e intuitiva de calcular estatísticas de colunas de um DataFrame, existem funções Spark capazes de calcu-

```
# Coletando colunas numéricas do DataFrame
num_cols = [s.name for s in df.schema if s.dataType.typeName() != 'string']

# Computando estatísticas de colunas numéricas
df.select(num_cols).describe().show()

+-----+-----+-----+-----+
|summary|    Quantity|   UnitPrice|CustomerID|
+-----+-----+-----+-----+
|  count|      3108|       3108|        1968|
|  mean| 8.627413127413128| 4.151946589446603|15661.388719512195|
| stddev|26.371821677029203|15.638659854603892|1854.4496996893627|
|  min|        -24|         0.0|      12431.0|
|  max|         600|      607.49|     18229.0|
+-----+-----+-----+-----+
```

**Figure 6.8:** Extrairando estatísticas relevantes de colunas em um DataFrame.

lar pontualmente ou individualmente cada uma das estatísticas contempladas pelo método `describe()`. De modo a proporcionar um entendimento claro sobre isso, a figura 6.9 exemplifica o processo necessário para computar algumas estatísticas de uma base de dados.

```
# Importando funções estatísticas
from pyspark.sql.functions import count, mean, stddev_pop, min, max

# Calculando estatísticas na unha
function_list = [count, mean, stddev_pop, min, max]
stats_list = [func(col).alias(col + '_' + func.__name__) for func in function_list for col in num_cols]

# Realizando consulta (mostrando apenas as colunas de count e mean - espaço)
df.select(stats_list[:6]).show()

+-----+-----+-----+-----+
|Quantity_count|UnitPrice_count|CustomerID_count|  Quantity_mean|   UnitPrice_mean|CustomerID_mean|
+-----+-----+-----+-----+-----+-----+
|        3108|          3108|           1968|8.627413127413128| 4.151946589446603|15661.388719512195|
+-----+-----+-----+-----+-----+-----+
```

**Figure 6.9:** Calculando estatísticas pontualmente a partir de funções nativas do Spark.

Adicionalmente, é possível utilizar funções estatísticas no pacote `StatFunctions` acessível através do módulo `stat` de `DataFrames`. Alguns exemplos:

- `df.stat.approxQuantile()`
- `df.stat.crosstab()`
- `df.stat.freqItems()`
- `df.stat.corr()`

## 6.4. Trabalhando com Strings

A manipulação de strings é uma ação comum em fluxos de dados, desde simples substituições até expressões regulares. Iniciando com alguns exemplos envolvendo *case*, a figura 6.10 utiliza a função `initcap()` para aplicar um *title case* em um campo string.

```
# Importando função
from pyspark.sql.functions import initcap

# Aplicando title case
df.select(
    "Description",
    initcap("Description").alias("TitledDescription")
).show(3, truncate=False)

+-----+-----+
|Description|TitledDescription|
+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|White Hanging Heart T-light Holder|
|WHITE METAL LANTERN|White Metal Lantern|
|CREAM CUPID HEARTS COAT HANGER|Cream Cupid Hearts Coat Hanger|
+-----+-----+
only showing top 3 rows
```

**Figure 6.10:** Aplicação de *title case* em um campo string.

Nesta mesma abordagem, a figura 6.11 exemplifica a aplicação das funções `lower()` e `upper()` para alterar o *case* de uma string para caixa baixa e caixa alta, respectivamente.

Por fim, visando relembrar algumas possibilidades de Spark em meio a utilização de SQL, a figura 6.12 aplica as mesmas transformações demonstradas nas figuras 6.10 e 6.11 via SparkSQL.

Outro exemplo de tarefa trivial envolvendo strings é a aplicação dos processos de *paddings* e *trimming*, ambos exemplificados em detalhes através da figura 6.13. Nela, utilizam-se as funções `ltrim`, `rtrim`, `lpad`, `rpad` e `trim`. A string utilizada no exemplo é criada pontualmente como um literal através da função `lit()`.

- `ltrim`: elimina espaços em branco à esquerda da string;

```
# Importando funções
from pyspark.sql.functions import lower, upper

# Alterando case
df.select(
    "Description",
    lower("Description").alias("LowerDescription"),
    upper(lower("Description")).alias("UpperDescription")
).show(3, truncate=False)

+-----+-----+-----+
|Description|LowerDescription|UpperDescription|
+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|white hanging heart t-light holder|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN|white metal lantern|WHITE METAL LANTERN|
|CREAM CUPID HEARTS COAT HANGER|cream cupid hearts coat hanger|CREAM CUPID HEARTS COAT HANGER|
+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.11:** Aplicação das funções `lower()` e `upper()` para alteração do `case`

```
# Realizando via SQL
spark.sql("""
    SELECT
        Description AS original,
        initcap>Description) AS titled,
        lower>Description) AS lower,
        upper(lower>Description)) AS upper

    FROM vw_retail_data
""").show(3)

+-----+-----+-----+-----+
|original|titled|lower|upper|
+-----+-----+-----+-----+
|WHITE HANGING HEA...|White Hanging Hea...|white hanging hea...|WHITE HANGING HEA...|
| WHITE METAL LANTERN| White Metal Lantern| white metal lantern| WHITE METAL LANTERN|
|CREAM CUPID HEART...|Cream Cupid Heart...|cream cupid heart...|CREAM CUPID HEART...|
+-----+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.12:** Transformando `case` de strings via SparkSQL.

- `rtrim`: elimina espaços em branco à direita da string;
- `trim`: elimina espaços em ambos os lados da string;
- `lpad`: adiciona espaços à esquerda da string. Também pode funcionar como um “left”;
- `rpad`: adiciona espaços à direita da string;

Provavelmente uma das tarefas mais interessantes ao trabalhar com strings envolve a utilização de **expressões regulares**, popularmente conhecidas como RegEx. Em linhas gerais, as RegEx proporcionam a habilidade de especificar uma série de re-

```
# Importando funções
from pyspark.sql.functions import lit, ltrim, rtrim, trim, lpad, rpad

# Exemplificando tratativas em string
df.select(
    ltrim(lit("    HELLO    ")).alias("ltrim"),
    rtrim(lit("    HELLO    ")).alias("rtrim"),
    trim(lit("    HELLO    ")).alias("trim"),
    lpad(lit("HELLO"), 2, " ").alias("lpad"),
    rpad(lit("HELLO"), 10, " ").alias("rpad")
).show(2)

+-----+-----+-----+-----+
| ltrim| rtrim| trim|lpad| rpad|
+-----+-----+-----+-----+
|HELLO |    HELLO|HELLO| HE|HELLO   |
|HELLO |    HELLO|HELLO| HE|HELLO   |
+-----+-----+-----+-----+
only showing top 2 rows
```

**Figure 6.13:** Exemplos de aplicação dos processos de *trimming* e *padding* em uma string literal do Spark.

gras a serem utilizadas para extrair valores específicos de uma string ou substituí-los com outra entrada. Basicamente, as duas principais funções a serem utilizadas neste universo são `regexp\_extract` e `regexp\_replace` para, respectivamente, extraír ou substituir valores a partir de expressões regulares.

A figura 6.14 demonstra um exemplo de utilização de ambas as funções para substituição ou extração de strings a partir de expressões regulares. Como lógica adotada, foi definido um padrão para procurar por 5 diferentes cores na coluna “Description” do DataFrame, substituindo-as pela string ‘COLOR’ via `regexp\_replace()` ou retornando a primeira aparição de uma das cores via `regexp\_extract()`.

Em um outro exemplo de aplicação, a função `translate()` poderia ser utilizada para substituição a nível de caractere. Em outras palavras, além da coluna de string do DataFrame, são fornecidos para a função dois argumentos especificando elementos a serem procurados e elementos a serem substituídos, nesta ordem. A substituição ocorre a nível de caractere na ordem dos índices de cada elemento dos argumentos fornecidos.

```
# Importando funções
from pyspark.sql.functions import regexp_extract, regexp_replace

# Definindo RegEx
pattern = "(BLACK|WHITE|RED|GREEN|BLUE)"

# Construindo consulta
df.select(
    "Description",
    regexp_replace(col("Description"), pattern, "COLOR").alias("color_clean"),
    regexp_extract(col("Description"), pattern, 1).alias("first_color")
).show(3, truncate=False)

+-----+-----+-----+
|Description|color_clean|first_color|
+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|COLOR HANGING HEART T-LIGHT HOLDER|WHITE
|WHITE METAL LANTERN|COLOR METAL LANTERN|WHITE
|CREAM CUPID HEARTS COAT HANGER|CREAM CUPID HEARTS COAT HANGER|
+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.14:** Construção de consulta baseada em utilização de RegEx para substituição ou extração de determinados valores de acordo com um padrão definido.

Relacionando com processos de filtragem de dados, a função `instr` ou o método `contains` podem ser utilizados para verificar ocorrências de uma determinada palavra em uma string. A figura 6.15 ilustra exemplos de aplicações deste tipo.

```
# Importando funções
from pyspark.sql.functions import instr

# Criando filtros
black_color_filter = col("Description").contains("BLACK")
white_color_filter = instr(col("Description"), "WHITE") >= 1

# Aplicando consulta
df.select("Description")\
    .where(black_color_filter & white_color_filter)\
    .show(3, truncate=False)

+-----+
|Description|
+-----+
|JUMBO BAG BAROQUE BLACK WHITE|
|WOOD BLACK BOARD ANT WHITE FINISH|
|JUMBO BAG BAROQUE BLACK WHITE|
+-----+
only showing top 3 rows
```

**Figure 6.15:** Verificando ocorrências de palavras em string utilizando `contains` ou `instr`.

Por fim, visando propor uma aplicação relativamente mais complexa, o conceito de `list comprehension` em Python será mais uma vez utilizado para construir uma lista de funções capazes de retornar colunas dinamicamente, aproveitando-se da habilidade do Spark em receber um número dinâmico de argumentos. Em outros termos, significa dizer que iremos construir uma lista onde cada elemento é uma função Spark responsável por aplicar uma lógica definida de transformação de dados. Esta lista então é fornecida ao `select` como `vargars`, isto é, argumentos dinâmicos. Com isso, a consulta irá considerar cada resultado individual das funções contidas na lista como uma nova coluna.

Para traduzir esta tarefa em um exemplo prático, a figura 6.16 consolida a utilização da função `locate()` para procurar determinadas cores na coluna “Description”. Os resultados são então transformados em colunas booleanas (uma para cada cor) e adicionadas ao `select` via argumentos dinâmicos.

```
# Importando funções
from pyspark.sql.functions import locate

# Definindo lista de cores
find_list = ["black", "white", "red", "green", "blue"]
func_list = [locate(f.upper(), col("Description")).cast("boolean").alias(f"is_{f}") for f in find_list]

# Aplicando select
df.select("Description", *func_list).show(5, truncate=False)

+-----+-----+-----+-----+
|Description|is_black|is_white|is_red|is_green|is_blue|
+-----+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|false|true|false|false|false|
|WHITE METAL LANTERN|false|true|false|false|false|
|CREAM CUPID HEARTS COAT HANGER|false|false|false|false|false|
|KNITTED UNION FLAG HOT WATER BOTTLE|false|false|false|false|false|
|RED WOOLLY HOTTIE WHITE HEART.|false|true|true|false|false|
+-----+-----+-----+-----+
only showing top 5 rows
```

**Figure 6.16:** Construção de uma consulta dinâmica em Spark utilizando `list comprehensions` para armazenar funções. A lista é então fornecida ao método `select` como um argumento dinâmico, permitindo assim com que o Spark considere cada elemento individual como uma nova expressão e, consequentemente, uma nova coluna no resultado final.

## 6.5. Trabalhando com Datas

Datas sempre foram atributos desafiadores em linguagens de programação e bancos de dados. Ao ler uma base de dados com Spark com a configuração “inferSchema” igual a “true”, é realizado o melhor esforço para identificação de colunas contendo informações de datas em seus mais variados formatos.

Entretanto, nem sempre o resultado esperado é alcançado, principalmente quando a base de dados possui valores de datas mal formatados ou bem específicos. Em alguns cenários práticos, atributos de data em bases de dados são armazenados ou lidos como strings para posterior transformação em tempo de execução,

Em uma abordagem inicial, a figura 6.17 realiza a importação de duas funções interessantes: `current_date()` e `current_timestamp()` responsáveis por retornar, respectivamente a data atual sem hora e a data atual com hora.

```
# Importando funções
from pyspark.sql.functions import current_date, current_timestamp

# Gerando datas e horas
df_date = spark.range(1) \
    .withColumn("today", current_date()) \
    .withColumn("now", current_timestamp())
df_date.show(truncate=False)

# Criando view
df_date.createOrReplaceTempView("tbl_date")

# Verificando schema
df_date.printSchema()

+---+-----+-----+
|id |today      |now          |
+---+-----+-----+
|0  |2022-02-22|2022-02-22 20:52:38.881|
+---+-----+-----+

root
 |-- id: long (nullable = false)
 |-- today: date (nullable = false)
 |-- now: timestamp (nullable = false)
```

Figure 6.17: Exemplo de retorno das funções `current_date()` e `current_timestamp()`.

Ao trabalhar com uma base de dados com colunas de data, um leque de possibilidades e funções se abre de acordo com as mais variadas necessidades de análise. A figura 6.18 exemplifica o uso das funções `date_add()` e `date_sub()` como uma forma de adicionar ou subtrair, respectivamente, datas de uma coluna do tipo Date.

```
# Importando funções
from pyspark.sql.functions import date_add, date_sub

# Adicionando datas
df_date.select(
    "today",
    date_add("today", 5).alias("today_plus_5"),
    date_sub("today", 5).alias("today_minus_5")
).show()

# Em SQL
spark.sql("""
    SELECT
        today,
        date_add(today, 5) AS today_plus_5,
        date_sub(today, 5) AS today_minus_5
    FROM tbl_date
""").show()
```

today	today_plus_5	today_minus_5
2022-02-22	2022-02-27	2022-02-17

today	today_plus_5	today_minus_5
2022-02-22	2022-02-27	2022-02-17

**Figura 6.18:** Adição e subtração de dias em campos de data utilizando as funções `date_add()` e `date_sub()`. No segundo bloco de código implementado, é possível visualizar a consulta análoga utilizando SparkSQL direto na tabela temporária criada anteriormente.

Uma outra tarefa comum em transformações de data é a de verificar a diferença, em dias, entre duas datas. Para tal, é possível utilizar a função `datediff()`. Como a quantidade de dias em diferentes meses é variável, o Spark traz consigo a função `months_between()` com a capacidade de calcular a diferença entre datas em uma base mensal. A figura 6.19 ilustra a aplicação dos dois cenários acima descritos.

```
# Importando funções
from pyspark.sql.functions import date_add, date_sub

# Adicionando datas
df_date.select(
    "today",
    date_add("today", 5).alias("today_plus_5"),
    date_sub("today", 5).alias("today_minus_5")
).show()

# Em SQL
spark.sql("""
    SELECT
        today,
        date_add(today, 5) AS today_plus_5,
        date_sub(today, 5) AS today_minus_5
    FROM tbl_date
""").show()
```

	today	today_plus_5	today_minus_5
2022-02-22	2022-02-27	2022-02-17	

	today	today_plus_5	today_minus_5
2022-02-22	2022-02-27	2022-02-17	

**Figure 6.19:** Computando diferenças entre datas a partir das funções datediff() e months\_between()

Adicionalmente, o bloco inferior de código contido na figura ?? utiliza a função `to_date()` para converter uma string em uma variável do tipo `date` (eventualmente em um formato específico que pode ser evidenciado nos argumentos). A figura 6.20 ilustra mais alguns exemplos de conversão considerando diferentes formatos de datas. Nela, é possível notar o comportamento do Spark quando uma data não consegue ser “convertida” pelos padrões fornecidos na função `to_date()` ou `to_timestamp()`.

Por fim, finalizando o universo de datas e timestamps, a comparação entre estes elementos é um tanto quanto intuitiva. A figura 6.21 traz um exemplo comparativo de uma data já presente em um DataFrame e um literal criado pontualmente. A única premissa é que o literal precisa ser devidamente criado no formato padrão `yyyy-MM-dd`

```
# Importando funções
from pyspark.sql.functions import to_date, to_timestamp

# Conversão de datas
spark.range(1)\.
    withColumn("date_default", to_date(lit("02-02-2022")))\.
    withColumn("date_formatted", to_date(lit("02-02-2022"), "dd-MM-yyyy"))\.
    withColumn("timestamp_formatted", to_timestamp(lit("02-02-2022"), "dd-MM-yyyy"))\.
.show()

+-----+-----+-----+
| id|date_default|date_formatted|timestamp_formatted|
+-----+-----+-----+
|  0|        null|   2022-02-02|2022-02-02 00:00:00|
+-----+-----+-----+
```

**Figure 6.20:** Exemplo de conversão de strings em datas com as funções `to_date()` e `to_timestamp()`. Na consulta, é utilizada um data (literal) no formato “dd-MM-yyy” e as funções são executadas sem a especificação de um formato (default) e com a especificação de um formato. Ao não reconhecer o formato de uma data, o Spark retorna `null`.

(ou então transformado em data via `to_date()` utilizando um formato específico).

```
# Comparando datas
df_date.select(
    "today",
    lit("2022-02-23").alias("tomorrow"),
    col("today") > lit("2022-02-23")
).show()

+-----+-----+-----+
| today| tomorrow|(today > 2022-02-23)|
+-----+-----+-----+
| 2022-02-22| 2022-02-23|           false|
+-----+-----+-----+
```

**Figure 6.21:** Comparação de datas em Spark utilizando colunas já existentes em DataFrames e literais.

## 6.6. Trabalhando Dados Nulos

Como ponto de partida em relação a dados nulos, há uma recomendação em realmente utilizar dados `null` para representar nulos ou vazios. Isto pois o Spark atua mais rápido em cenários deste tipo do que representando nulos com strings vazias. Nesta seção, serão fornecidos alguns exemplos de gerenciamento de dados nulos,

desde comportamentos específicos até aplicação de funções úteis para retornar resultados com base em dados nulos.

Logo de início, a figura 6.22 faz referência à aplicação da função `coalesce()` como uma forma de retornar o primeiro valor não-nulo entre uma ou mais colunas. Em caso de inexistência de valores nulos, a função retorna o dado da primeira coluna fornecida no argumento. O DataFrame utilizado na aplicação da função foi criado pontualmente através do método `spark.createDataFrame()`.

```
# Importando funções
from pyspark.sql.functions import coalesce

# Gerando DataFrame
df_null = spark.createDataFrame(
    [(1, None), (None, 2)], ["col_1", "col_2"]
)

# Aplicando função
df_null.select(
    "col_1", "col_2",
    coalesce("col_1", "col_2")
).show()

+-----+-----+
|col_1|col_2|coalesce(col_1, col_2)|
+-----+-----+
|    1|  null|                  1|
|  null|     2|                  2|
+-----+-----+
```

**Figure 6.22:** Aplicação da função `coalesce()` para retornar a primeira aparição de um registro não nulo em um *set* de colunas fornecidas.

Considerando as diversas possibilidades de operações e verificações com dados nulos, a figura 6.23 traz uma série de exemplos utilizando as funções `ifnull()`, `nullif()`, `nvl()` e `nvl2()`. Abaixo, uma breve descrição sobre cada uma das funções citadas:

- `ifnull()`: retorna o segundo valor/argumento caso o primeiro seja nulo;
- `nullif()`: retorna nulo caso os argumentos sejam iguais;
- `nvl()`: retorna o segundo valor/argumento caso o primeiro seja nulo;

- `nvl2()`: retorna o segundo valor/argumento caso o primeiro seja nulo, caso contrário, retorna o terceiro valor/argumento

```
# Criando view
df_null.createOrReplaceTempView("tbl_null")

# Testando diversas outras funções
spark.sql("""
    SELECT
        col_1,
        ifnull(col_1, "Retorno") AS ifnull,
        nullif("teste", "teste") AS nullif,
        nvl(col_1, "Retorno") AS nvl,
        nvl2(col_1, "Retorno", "Else") AS nvl2
    FROM tbl_null
""").show()
```

col_1	ifnull	nullif	nvl	nvl2
1	1	null	1 Retorno	
null Retorno	null Retorno		Else	

**Figure 6.23:** Exemplo de resultados obtidos com a aplicação de diferentes funções para tratamento de dados nulos. As funções foram aplicadas utilizando SparkSQL.

Para **dropar** ou **eliminar** dados nulos em um DataFrame Spark, é possível utilizar a função `drop()` acessível através do módulo `DataFrame.na`. Em seu argumento, é possível especificar duas opções distintas:

- “any”: elimina a linha caso **qualquer** valor em qualquer coluna seja nulo
- “all”: elimina a linha apenas se todos os valores forem `null` ou `NaN`

Além disso, é possível especificar o argumento `subset` para eliminar dados nulos considerando apenas uma lista específica de colunas. A figura 6.24 exemplifica um cenário utilizando um DataFrame criado manualmente.

No outro espectro operacional o **preenchimento** de dados nulos pode ser feito através da função `fill()` também presente no módulo `DataFrame.na`. Como principais argumentos, a função `fill()` recebe um valor a ser utilizado no preenchimento

```
# Adicionando coluna totalmente nula
df_null = df_null.withColumn("col_3", lit(None))

# Utilizando drop
df_null.na.drop().show()
df_null.na.drop("all").show()
df_null.na.drop("all", subset=["col_2", "col_3"]).show()

+---+---+---+
|col_1|col_2|col_3|
+---+---+---+
|      |
+---+---+---+

+---+---+---+
|col_1|col_2|col_3|
+---+---+---+
|   1| null| null|
| null|   2| null|
+---+---+---+

+---+---+---+
|col_1|col_2|col_3|
+---+---+---+
| null|   2| null|
+---+---+---+
```

**Figure 6.24:** Exemplos de eliminação de dados nulos através da função `DataFrame.na.drop()`.

Com diferentes configurações, a função pode apresentar comportamentos de acordo com as necessidades do usuário em relação ao gerenciamento de dados nulos em um DataFrame inteiro ou em um subconjunto de colunas.

dos nulos e opcionalmente uma lista de colunas a serem consideradas. Em uma configuração mais refinada, é possível criar um **dicionário** com chaves representando as colunas alvo do DataFrame e os valores representando os respectivos valores de preenchimento de cada coluna/chave. A figura 6.25 traz alguns exemplos práticos de utilização da função `fill()`.

Uma outra aplicação interessante que pode ser análoga aos processos de `drop` ou `fill` é a **substituição** de valores em uma ou mais colunas a partir da função `replace()`.

```
# Preenchendo dados nulos
df_null = df_null.select("col_1", "col_2")
df_null.na.fill(0).show()

# Preenchendo dados nulos com subset
df_null.na.fill(0, subset=["col_1"]).show()

# Preenchendo dados nulos com dicionário
fill_dict = {"col_1": -1, "col_2": 10}
df_null.na.fill(fill_dict).show()
```

col_1	col_2
1	0
0	2

col_1	col_2
1	null
0	2

col_1	col_2
1	10
-1	2

**Figure 6.25:** Exemplos de preenchimento de dados nulos em diferentes cenários utilizando a função `fill()`.

## 6.7. Trabalhando com Tipos Complexos

Em alguns cenários, tipos primitivos complexos auxiliam na construção e definição de *schemas* capazes de atender as mais variadas necessidades em fluxos de Big Data. Em geral, existem três tipos primitivos complexos: structs, arrays e maps.

### 6.7.1. Structs

Structs podem ser imaginados como “DataFrames dentro de DataFrames”. A figura 6.26 ilustra a criação de um DataFrame apartado a partir da seleção de campos es-

pecíficos de um DataFrame e a construção de um campo do tipo struct utilizando a função `struct()`. A estrutura do campo complexo criado se assemelha a um dicionário composto por chaves (implicitamente representadas pelos nomes dos campos) e valores (valores dos próprios campos no DataFrame original).

```
# Importando função
from pyspark.sql.functions import struct

# Criando DataFrame com tipo complexo
df_complex = df.select(
    "Description",
    "InvoiceNo",
    struct("Description", "InvoiceNo").alias("complex")
)
df_complex.show(5, truncate=False)

# Criando view
df_complex.createOrReplaceTempView("df_complex")

+-----+-----+
|Description|InvoiceNo|complex
+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|536365|{WHITE HANGING HEART T-LIGHT HOLDER, 536365}
|WHITE METAL LANTERN|536365|{WHITE METAL LANTERN, 536365}
|CREAM CUPID HEARTS COAT HANGER|536365|{CREAM CUPID HEARTS COAT HANGER, 536365}
|KNITTED UNION FLAG HOT WATER BOTTLE|536365|{KNITTED UNION FLAG HOT WATER BOTTLE, 536365}
|RED WOOLLY HOTTIE WHITE HEART.|536365|{RED WOOLLY HOTTIE WHITE HEART., 536365}
+-----+-----+
only showing top 5 rows
```

**Figure 6.26:** Exemplo de criação de um campo do tipo struct em um DataFrame.

O procedimento para seleção de elementos ou atributos em um campo do tipo struct se assemelha ao próprio processo de seleção de colunas em um DataFrame. A sintaxe utilizada é `nome_campo.nome_atributo`. No exemplo ilustrado pela figura 6.27, o atributo “Description” do campo complexo “complex” é acessado via `complex.Description`. Para acessar todos os elementos de um campo struct, a sintaxe é `complex.*`, assim como em DataFrames. Uma outra forma de acessar atributos em campos struct é através do método `getField()`.

## 6.7.2. Arrays

De forma direta, arrays são tipos complexos semelhantes a listas. Em um cenário prático de entendimento, seria possível criar um array a partir do campo “Description” onde cada palavra seria um elemento de um array em um novo campo complexo. A

```
# Selecionando atributos de campo struct
df_complex.select("complex.Description").show(5)

# Selecionando todos os atributos de um struct
df_complex.select("complex.*").show(5)

# Selecionando atributo de um struct via getField
df_complex.select(col("complex").getField("InvoiceNo")).show(5)

+-----+
|       Description|
+-----+
|WHITE HANGING HEA...|
| WHITE METAL LANTERN|
|CREAM CUPID HEART...|
|KNITTED UNION FLA...|
|RED WOOLLY HOTTIE...|
+-----+
only showing top 5 rows

+-----+-----+
|       Description|InvoiceNo|
+-----+-----+
|WHITE HANGING HEA...| 536365|
| WHITE METAL LANTERN| 536365|
|CREAM CUPID HEART...| 536365|
|KNITTED UNION FLA...| 536365|
|RED WOOLLY HOTTIE...| 536365|
+-----+-----+
only showing top 5 rows

+-----+
|complex.InvoiceNo|
+-----+
|      536365|
|      536365|
|      536365|
|      536365|
|      536365|
+-----+
only showing top 5 rows
```

**Figure 6.27:** Acessando atributos de um campo struct.

figura 6.28 ilustra um exemplo de criação de um campo do tipo array a partir da aplicação da função `split()` em um campo de textos, separando assim cada elemento considerando um delimitador fornecido no argumento da função. O resultado é um array composto por uma série de elementos em um formato semelhante a uma lista.

Para acessar elementos em um array, é possível utilizar conceitos de indexação.

```
# Importando função
from pyspark.sql.functions import split

# Splitando texto
df.select("Description", split("Description", " ")).show(5, truncate=False)

+-----+-----+
|Description |split	Description, , -1|
+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]| |
|WHITE METAL LANTERN|[WHITE, METAL, LANTERN]|
|CREAM CUPID HEARTS COAT HANGER|[CREAM, CUPID, HEARTS, COAT, HANGER]|
|KNITTED UNION FLAG HOT WATER BOTTLE|[KNITTED, UNION, FLAG, HOT, WATER, BOTTLE]|
|RED WOOLLY HOTTIE WHITE HEART.||[RED, WOOLLY, HOTTIE, WHITE, HEART.]|
+-----+-----+
only showing top 5 rows
```

**Figure 6.28:** Exemplo de criação de um array através da separação (split()) de um campo string a partir de um delimitador.

Na figura 6.29 é realizada uma consulta em um DataFrame criado especificamente para demonstrar situações envolvendo arrays onde é retornado apenas o primeiro elemento do array criado.

```
# Criando DataFrame para uso com arrays
df_array = df.select(split("Description", " ")).alias("split_desc")

# Selecionando elementos
df_array.selectExpr("split_desc[0] AS first_word").show(5)

+-----+
|first_word|
+-----+
|  WHITE|
|  WHITE|
|  CREAM|
|  KNITTED|
|  RED|
+-----+
only showing top 5 rows
```

**Figure 6.29:** Acessando elementos dentro de um array a partir de indexação (início em 0)

Em alguns casos, pode ser interessante retornar a quantidade de elementos presentes em um campo do tipo array. Para tal, pode-se utilizar a função size(). A figura 6.30 ilustra um exemplo desta aplicação.

```
# Importando função
from pyspark.sql.functions import size

# Elementos em um array
df_array.select(size("split_desc")).show(5)

+-----+
|size(split_desc)|
+-----+
|      5|
|      3|
|      5|
|      6|
|      5|
+-----+
only showing top 5 rows
```

**Figure 6.30:** Coletando a quantidade de elementos de um array através da função `size()`

Uma outra função interessante a ser aplicada em arrays é a `array_contains()`. Com ela, é possível validar se um determinado valor está presente em um array. A figura 6.31 exemplifica sua aplicação prática.

```
# Importando função
from pyspark.sql.functions import array_contains

# Verificando se o array contém determinado valor
df_array.select("split_desc", array_contains("split_desc", "WHITE")).show(5, False)

+-----+-----+
|split_desc |array_contains(split_desc, WHITE)|
+-----+-----+
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER] |true
|[WHITE, METAL, LANTERN] |true
|[CREAM, CUPID, HEARTS, COAT, HANGER] |false
|[KNITTED, UNION, FLAG, HOT, WATER, BOTTLE]|false
|[RED, WOOLLY, HOTTIE, WHITE, HEART.] |true
+-----+-----+
only showing top 5 rows
```

**Figure 6.31:** Verificando se um valor está presente em um array através da função `array_contains()`

Por fim, imaginando um cenário onde seja necessário coletar cada elemento de um array e transformá-los em linhas, a função `explode()` possui um papel fundamental para alcançar este objetivo. Com ela, é possível “explodir” todos os elementos de um array em linhas, permitindo análises específicas de acordo com necessi-

dades específicas. A figura 6.32 ilustra sua aplicação utilizando transformações em DataFrames.

```
# Importando função
from pyspark.sql.functions import explode

# "Explodindo" array em múltiplas linhas
df_array.select(
    "split_desc",
    explode("split_desc").alias("exploded")
).show(13, truncate=False)

+-----+-----+
|split_desc          |exploded|
+-----+-----+
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|WHITE   |
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HANGING |
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HEART   |
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|T-LIGHT  |
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HOLDER  |
|[WHITE, METAL, LANTERN]                   |WHITE   |
|[WHITE, METAL, LANTERN]                   |METAL   |
|[WHITE, METAL, LANTERN]                   |LANTERN |
|[CREAM, CUPID, HEARTS, COAT, HANGER]     |CREAM   |
|[CREAM, CUPID, HEARTS, COAT, HANGER]     |CUPID   |
|[CREAM, CUPID, HEARTS, COAT, HANGER]     |HEARTS  |
|[CREAM, CUPID, HEARTS, COAT, HANGER]     |COAT    |
|[CREAM, CUPID, HEARTS, COAT, HANGER]     |HANGER  |
+-----+-----+
only showing top 13 rows
```

Figure 6.32: “Explodindo” um array em linhas a partir da função `explode()`

Ao utilizar a função `explode()` como uma transformação em DataFrames, o Spark cuida de algumas situações que não são automaticamente gerenciadas via SQL. Em outras palavras, a aplicação da função `explode()` diretamente via Spark-SQL, no formato exemplificado pela figura 6.32, demanda a utilização de um outro comando conhecido como LATERAL VIEW.

Com o LATERAL VIEW, é possível realizar a “explosão” dos elementos de um array ao mesmo tempo que outras colunas do DataFrame (ou da View) sejam trazidas também no *record set*. A figura 6.33 ilustra uma aplicação do tipo.

```
# Criando view
df.select("Description").createOrReplaceTempView("vw_description")

# Explodindo array em SQL com LATERAL VIEW
spark.sql("""
    SELECT
        Description,
        split>Description, " ") AS splitted,
        exploded

    FROM vw_description

    LATERAL VIEW
        explode(split>Description, " ")) t AS exploded
""").show(8, truncate=False)

+-----+-----+-----+
|Description|splitted|exploded|
+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|WHITE |
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HANGING |
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HEART |
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|T-LIGHT |
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|HOLDER |
|WHITE METAL LANTERN|[WHITE, METAL, LANTERN]|WHITE |
|WHITE METAL LANTERN|[WHITE, METAL, LANTERN]|METAL |
|WHITE METAL LANTERN|[WHITE, METAL, LANTERN]|LANTERN |
+-----+-----+-----+
only showing top 8 rows
```

**Figure 6.33:** “Explodindo” um array em linhas via SparkSQL utilizando a cláusula LATERAL VIEW

### 6.7.3. Maps

Campos do tipo map são criados utilizando a função `map` (SQL ou Scala) ou `create_map()` (Python). A figura 6.34 ilustra o processo de criação de um campo map e sua apresentação na base de dados.

O acesso a atributos presentes em um campo do tipo map é feito a partir da **chave**. Registros que não possuem valor para determinada chave são retornados como valores nulos, conforme figura 6.35.

Assim como nos arrays, é possível “explodir” elementos de um map com a função `explode()`. As figuras 6.36 e 6.37 ilustram, respectivamente, este processo aplicado através de transformações em DataFrames e em SparkSQL. É possível notar, especificamente na figura 6.37, que não é necessário aplicar a cláusula LATERAL VIEW, dado que os elementos de um map são quebrados em diferentes colunas de chave

```
# Importando função
from pyspark.sql.functions import create_map

# Criando um campo com map
df_map = df.select(
    "Description",
    "InvoiceNo",
    create_map("Description", "InvoiceNo").alias("complex_map")
)
df_map.show(5, truncate=False)

+-----+-----+
|Description |InvoiceNo|complex_map
+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER |536365 |{WHITE HANGING HEART T-LIGHT HOLDER -> 536365}
|WHITE METAL LANTERN |536365 |{WHITE METAL LANTERN -> 536365}
|CREAM CUPID HEARTS COAT HANGER |536365 |{CREAM CUPID HEARTS COAT HANGER -> 536365}
|KNITTED UNION FLAG HOT WATER BOTTLE |536365 |{KNITTED UNION FLAG HOT WATER BOTTLE -> 536365}
|RED WOOLLY HOTTIE WHITE HEART. |536365 |{RED WOOLLY HOTTIE WHITE HEART. -> 536365}
+-----+-----+
only showing top 5 rows
```

**Figure 6.34:** Criação de um campo complexo map via `create_map()`

```
# Selecionando valores em um campo map
df_map.selectExpr("complex_map['WHITE METAL LANTERN']").show(3)

+-----+
|complex_map[WHITE METAL LANTERN]|
+-----+
|           null|
|      536365|
|           null|
+-----+
only showing top 3 rows
```

**Figure 6.35:** Acessando valores em um campo map através da utilização de chaves (semelhante ao processo de um dicionário em Python).

e valor.

```
# 'Explodindo' valores em um campo map
df_map.select("Description", "InvoiceNo", explode("complex_map")).show(3, truncate=False)

+-----+-----+-----+
|Description |InvoiceNo|key          |value |
+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|536365 |WHITE HANGING HEART T-LIGHT HOLDER|536365|
|WHITE METAL LANTERN |536365 |WHITE METAL LANTERN |536365|
|CREAM CUPID HEARTS COAT HANGER |536365 |CREAM CUPID HEARTS COAT HANGER |536365|
+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.36:** Aplicando a função `explode()` para “quebrar” elementos de um campo map em chaves e valores.

```
# 'Explodindo' valores em um campo map
df_map.select("Description", "InvoiceNo", explode("complex_map")).show(3, truncate=False)

+-----+-----+-----+
|Description|InvoiceNo|key      |value   |
+-----+-----+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|536365|WHITE HANGING HEART T-LIGHT HOLDER|536365|
|WHITE METAL LANTERN             |536365|WHITE METAL LANTERN             |536365|
|CREAM CUPID HEARTS COAT HANGER |536365|CREAM CUPID HEARTS COAT HANGER |536365|
+-----+-----+-----+
only showing top 3 rows
```

**Figure 6.37:** Aplicando o mesmo processo de “explosão” em um campo map utilizando SparkSQL.

## 6.8. Trabalhando com JSON

Existem formas específicas proporcionadas pelo Spark para se trabalhar com arquivos JSON, incluindo operar diretamente em “strings de JSON”, converter de arquivos JSON ou extrair objetos JSON. Na figura 6.38 é demonstrado um exemplo de criação de uma coluna JSON.

```
# Criando coluna JSON
df_json = spark.range(1) \
    .selectExpr("""
        '{"myJSONKey": {"myJsonValue": [1, 2, 3]}}' as jsonString
    """)
df_json.show(truncate=False)

+-----+
|jsonString          |
+-----+
|[{"myJSONKey": {"myJsonValue": [1, 2, 3]}}]|
+-----+
```

**Figure 6.38:** Exemplo de criação de um campo JSON a partir de uma string formatada.

Para acessar uma coluna JSON existente em um DataFrame, é possível utilizar as funções `get_json_object()` e `json_tuple()` que, respectivamente, acessam elementos específicos em um objeto JSON e retornam campos JSON como tuplas. A figura 6.39 exemplifica suas aplicações.

Ainda na figura 6.39, é possível perceber que a função `get_json_object()` permite abertamente acessar elementos em um objeto JSON independente do nível de encadeamento presente. Para tal, utiliza-se uma sintaxe de acesso com o caractere

```
# Importando funções
from pyspark.sql.functions import get_json_object, json_tuple

# Retornando valores JSON
df_json.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJsonValue[1]").alias("column"),
    json_tuple(col("jsonString"), "myJSONKey")
).show(truncate=False)

+-----+
|column|c0
+-----+
|2      |{"myJsonValue": [1,2,3]}|
+-----+
```

**Figure 6.39:** Retorna elementos de um objeto JSON armazenado em um DataFrame.

“\$” para referenciar os níveis como se fossem atributos.

Na sequência, uma outra funcionalidade interessante no universo de objetos JSON é a possibilidade de converter registros do tipo Struct (detalhados na seção de tipos complexos) em JSON. Para tal, utiliza-se a função `to_json()` em um campo Struct já definido. A figura 6.40 ilustra o processo de criação de um campo Struct a subsequente conversão em um objeto JSON.

```
# Transformando structType em JSON
from pyspark.sql.functions import to_json

df.selectExpr("(InvoiceNo, Description) as myStruct")\
    .select(to_json(col('myStruct')))\n    .show(5, truncate=False)

+-----+
|to_json(myStruct)
+-----+
|{"InvoiceNo": "536365", "Description": "WHITE HANGING HEART T-LIGHT HOLDER"}|
|{"InvoiceNo": "536365", "Description": "WHITE METAL LANTERN"}|
|{"InvoiceNo": "536365", "Description": "CREAM CUPID HEARTS COAT HANGER"}|
|{"InvoiceNo": "536365", "Description": "KNITTED UNION FLAG HOT WATER BOTTLE"}|
|{"InvoiceNo": "536365", "Description": "RED WOOLLY HOTTIE WHITE HEART."}|
+-----+
only showing top 5 rows
```

**Figure 6.40:** Conversão de um tipo primitivo complexo Struct em JSON.

Por fim, a conversão de objetos JSON em elementos do tipo Struct é feita a partir da função `from_json()`. Com ela, é possível passar uma coluna contendo o objeto JSON e obter, como resultado um elemento do tipo Struct. Para tal, eventualmente é

necessário fornecer um *schema* como argumento para que a conversão seja feita respeitando tipos primitivos dos valores retirados da estrutura JSON. A figura ?? ilustra todo este processo.

```
# Importando bibliotecas
from pyspark.sql.functions import from_json
from pyspark.sql.types import *

# Definindo schema
parseSchema = StructType([
    StructField("InvoiceNo", StringType(), True),
    StructField("Description", StringType(), True)
])

# Criando consulta para conversão de json em struct
df.selectExpr("(InvoiceNo, Description) as myStruct")\
    .select(to_json(col("myStruct")).alias("newJSON"))\
    .select(col("newJSON"), from_json(col("newJSON"), parseSchema))\
    .show(2, truncate=True)

+-----+-----+
|      newJSON| from_json(newJSON)|
+-----+-----+
|{"InvoiceNo": "536...|{536365, WHITE HA...|
| {"InvoiceNo": "536...|{536365, WHITE ME...|
+-----+-----+
only showing top 2 rows
```

**Figure 6.41:** Conversão de um objeto JSON armazenado em um valor do tipo primitivo Struct.

Neste ponto, vale reforçar que, a partir do momento em que se tem o dado no tipo Struct, todas as operações em Structs listadas na seção anterior são válidas, desde o acesso a determinados elementos ou atributos em um Struct até transformações adicionais.

## 6.9. UDFs

Uma possibilidade poderosa no Spark envolve a definição das próprias funções customizadas de transformação: as UDFs (*User Defined Functions*). Com elas, é possível criar transformações específicas utilizando linguagens como Python ou Scala ou mesmo bibliotecas externas.

De forma geral, as UDFs podem receber uma ou mais valores como argumentos de entrada e retornar uma ou mais valores como saída. No final, as UDFs são

funções que operam nos dados, registro por registro, armazenadas como **funções temporárias** a serem utilizadas em uma `SparkSession` específica ou em um `SparkContext` específico.

Por mais que seja possível escrever UDFs em Python, Java ou Scala, existem considerações de performance relacionadas a linguagem escolhida. Para entender este *trade off*, vamos ilustrar o processo completo, desde a criação de uma função até sua submissão e vinculação no Spark. A figura 6.42 ilustra um exemplo de função capaz de retornar a potência de 3 de um número passado como argumento.

```
# Definindo função
def power3(value):
    return value ** 3

power3(2.0)
```

8.0

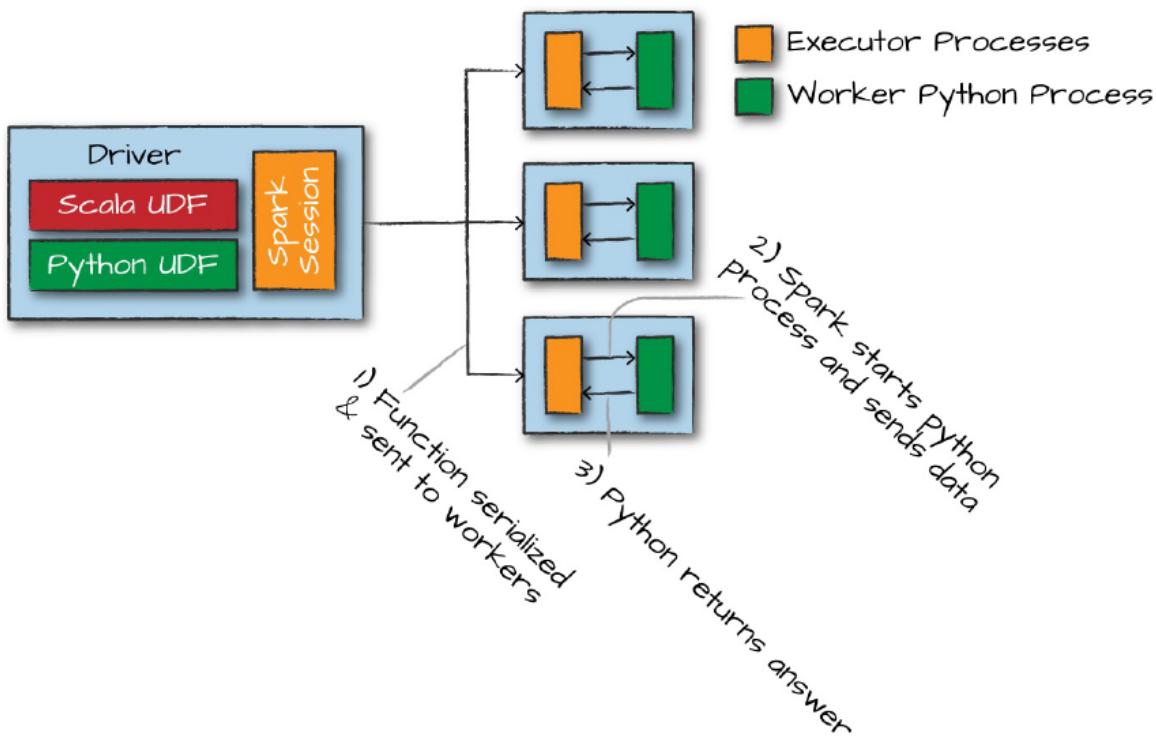
**Figure 6.42:** Criação de uma função em Python para futuramente ser registrada como uma UDF.

Por mais que tenhamos criado e testado a função (figura ??), ainda é preciso registrar no Spark para poder utilizar nas máquinas do cluster que, de fato, realizam o trabalho operacional de transformação. Dessa forma, o Spark irá **serializar** a função no *driver* e transferir os elementos através da rede para todos os processos **executores**. Isso ocorre **independente** da linguagem.

Entretanto, caso a função tenha sido escrita em Scala ou Java, é possível utilizá-la dentro da JVM (*Java Virtual Machine*), obtendo o mínimo de perda de performance. Caso a função tenha sido escrita em Python, existem algumas diferenças: neste caso, o Spark inicia um processo Python no *worker*, serializa os dados em um formato legível para que o Python possa ler, executar a função (linha a linha) e retornar os dados para a JVM e o Spark.

A figura 6.43 ilustra como UDFs escritas em Python são gerenciadas no cluster. Como principal ponto de atenção, por necessitar de uma inicialização do processo Python na máquina e uma subsequente serialização, existem altos custos computacionais envolvidos, além de uma competição de memória entre o Python e o Spark

nas máquinas **workers**. Se a UDF demandar um uso excessivo de memória, a máquina que processa o trabalho pode falhar.



**Figure 6.43:** Exemplificação do processo de execução de UDFs escritas em Python no cluster de execução do Spark.

Retomando o processo de submissão das UDFs, após a criação da função, é necessário registrá-la para que a mesma se torne acessível como uma função a ser executada em DataFrames. Para o registro de uma UDF, utiliza-se a função `udf()` do módulo `pyspark.sql.functions`. Como argumento, basta passar a função criada para que o registro seja feito e sua execução seja habilitada em DataFrames. A figura 6.44 ilustra este processo em Python.

Até este ponto, a função registrada somente pode ser executada como uma função de DataFrame e não por outros meios, como via SparkSQL ou expressões em strings, por exemplo). Para tal, é preciso registrar a função definida utilizando o método `spark.udf.register()` onde, como argumentos, passa-se o nome da função que será utilizada como uma expressão em string ou via SQL, além da função propriamente dita. Após isso, a função estará acessível não apenas como uma expressão de DataFrame, mas também como expressões em string. A figura 6.45 demonstra

```
# Importando função
from pyspark.sql.functions import udf

# Registrando udf
power3udf = udf(power3)

# Executando udf
df_udf = spark.range(5).toDF("num")
df_udf.select("num", power3udf(col("num"))).show()

+---+-----+
|num|power3(num)|
+---+-----+
| 0 |      0 |
| 1 |      1 |
| 2 |      8 |
| 3 |     27 |
| 4 |     64 |
+---+-----+
```

Figure 6.44: Registro de UDFs para habilitar a execução como funções de DataFrames no Spark

todo este processo.

Uma prática importante ao definir UDFs é a de explicitar o tipo de retorno dos dados. Em um exemplo real, a função `power3()` desenvolvida não possui nenhum *casting* explícito em seu retorno, permitindo assim que o Python siga suas próprias regras de cálculo. Em outras palavras, `power3(2)` retornará um inteiro, pois o argumento é um inteiro. Já `power3(2.0)` retornará um `float`, dado que o argumento é um `float`.

No ato de registro da UDF é possível explicitar o tipo primitivo esperado como retorno. Considerando o exemplo do parágrafo acima, se fosse especificado que a função `power3()` espera um retorno do tipo `DoubleType` (tipo primitivo do Spark), os resultados aplicados ao DataFrame `df_udf` seriam todos nulos. A explicação para isso é que `df_udf` contém uma coluna “`num`” apenas com números inteiros e, dessa forma, ao verificar que a função espera um retorno do tipo `DoubleType`, o Spark simplesmente retorna nulo. A figura 6.46 ilustra este caso.

Dessa forma, é sempre uma boa prática explicitar o tipo de retorno da função, evitando assim possíveis problemas de argumentos inválidos ou inespera-

```
# Registrando função para uso em SQL
spark.udf.register("power3", power3)

# Executando função como uma expressão
df_udf.selectExpr("power3(num)").show(3)

# Executando função com SparkSQL
df_udf.createOrReplaceTempView("df_udf")
spark.sql("""SELECT power3(num) FROM df_udf""").show(3)
```

```
+-----+
|power3(num)|
+-----+
|      0|
|      1|
|      8|
+-----+
only showing top 3 rows
```

```
+-----+
|power3(num)|
+-----+
|      0|
|      1|
|      8|
+-----+
only showing top 3 rows
```

**Figure 6.45:** Registro e execução de UDFs como expressões em String ou como queries SQL.

```
# Registrando função com retorno explícito
spark.udf.register("power3_double", power3, DoubleType())

# Executando função em inteiros
df_udf.selectExpr("num", "power3_double(num)").show()
```

```
+-----+
|num|power3_double(num)|
+-----+
|  0|          null|
|  1|          null|
|  2|          null|
|  3|          null|
|  4|          null|
+-----+
```

**Figure 6.46:** Exemplo de registro de UDF com tipo retorno explícito e execução com retorno diferente do tipo esperado.

dos. No caso mostrado acima, uma solução possível seria explicitamente realizar um *casting* no retorno da função Python com `float(x ** 3)`. A figura 6.47 demonstra uma possível solução para o problema evidenciado na figura 6.46. Para facilitar a construção do código, foi utilizada uma função `lambda` para cálculo da terceira potência com transformação explícita para `float`.

```
# Registrando função com casting explícito
spark.udf.register("power3_final", lambda x: float(x ** 3), DoubleType())

# Executando função em inteiros
df_udf.selectExpr("num", "power3_final(num)").show()
```

num	power3_final(num)
0	0.0
1	1.0
2	8.0
3	27.0
4	64.0

**Figure 6.47:** Exemplo de criação de função com transformação explícita no retorno da função ainda no ato de definição, garantindo que o código seja executado e retornado de acordo com o tipo primitivo de retorno configurado no ato do registro.

---

# 7

## Agregações

Após uma longa jornada de entendimento de transformações em DataFrames e nas diferentes formas de operar com diferentes tipos primitivos, é chegado o momento de adicionar mais um grande tópico no leque de conhecimentos do Spark: agregações.

Até este ponto, os ensinamentos consolidados giravam em torno de operações “de 1 para 1”, ou seja, seleções baseadas em expressões que literalmente transformavam colunas utilizando funções que resultavam em cenários diferentes porém de mesma granularidade.

De maneira introdutória, o Spark proporciona uma série de tipos de agrupamento, os quais serão detalhados ao longo de todo o capítulo:

- O agrupamento mais simples possível envolve uma summarização completa de um DataFrame a partir de uma agregação em uma cláusula select, resultando assim em um único registro;
- Um “**group by**” permite especificar uma ou mais chaves, assim como uma ou mais funções de agregação para transformar o valor das colunas;
- Uma “**window**” se assemelha ao group by, entretanto, as linhas de entrada da função são, de alguma forma, relacionadas a linha atual de resultado;

- Um “**grouping set**” pode ser usado para agregações em múltiplos níveis. Conjuntos de agrupamento são primitivos do SQL e podem ser utilizados via *rollups* e *cubes* em um DataFrame;
- Um “**rollup**” permite agregações de uma ou mais chaves e um ou mais valores, aplicando a sumarização de forma hierárquica;
- Um “**cube**” permite agregações de uma ou mais chaves e um ou mais valores, aplicando a sumarização em todas as combinações de colunas.

Cada agrupamento retorna um objeto do tipo `RelationalGroupedDataset`, no qual as agregações e sumarizações são aplicadas.

Como uma nota importante, o livro destaca:

Uma consideração extremamente relevante a ser feita é **como as respostas às perguntas envolvendo Big Data são esperadas**. Ao performar cálculos em Big Data, pode ser um tanto quanto complexo e custoso extrair uma resposta **exata** à uma questão e, dessa forma, pode ser muito mais simples realizar uma **aproximação** com um alto grau de confiança ou acurácia. Funções de aproximação podem ser uma boa oportunidade de agilizar a velocidade computacional do Spark, especialmente em análises *ad hoc* ou interativas.

## 7.1. Funções de Agregação

Todas as agregações aparecem como funções. Grande parte delas poderão ser importadas no `pyspark` via `pyspark.sql.functions`. Nos tópicos a seguir, algumas das principais funções de agragação serão abordadas de modo a propor um entendimento claro de suas atuações baseadas em exemplos práticos.

### 7.1.1. count

A primeira função de agregação a ser abordada é o `count`. De maneira geral, a contagem de linhas de um DataFrame pode ser realizada através do **método count** ou da

**transformação count.** No primeiro caso, o método `count()` existe como uma **ação** aplicada a um DataFrame e, dessa forma, os dados são retornados imediatamente. No segundo caso, o `count()` é considerado uma **transformação** aplicada dentro de operações de seleção. A figura 7.1 exemplifica todos estes cenários abordados com adição de um exemplo utilizando SparkSQL.

```
# Importando função
from pyspark.sql.functions import count

# count como ação
print(f'Ação: {df.count()}\n')

# count como transformação
df.select(count("*")).show()

# count via SparkSQL
spark.sql("""
    SELECT count(1) FROM vw_retail
""").show()
```

Ação: 541909

```
+-----+
|count(1)|
+-----+
| 541909|
+-----+
```

```
+-----+
|count(1)|
+-----+
| 541909|
+-----+
```

**Figure 7.1:** Contextos de utilização da função `count`

Nota importante:

Ao performar um `count(*)`, o Spark irá contar valores nulos. Entretanto, ao contar uma coluna individual (`count("col")`), o Spark não irá contar valores nulos.

As demais funções de agregação abordadas aqui não possuem “ações” de agregação assim como `count`. Dessa forma, todos os exemplos demonstrados daqui em

diantre serão dados como “transformações”.

### 7.1.2. countDistinct

De forma intuitiva, a função `countDistinct()` é utilizada para retornar uma contagem de valores distintos de uma determinada coluna ou grupo. A figura 7.2 exemplifica sua aplicação na contagem distinta de valores da coluna “StockCode” da base de dados de vendas utilizada, além de uma versão da mesma aplicação utilizando SparkSQL.

```
# Importando função
from pyspark.sql.functions import countDistinct

# Contando valores distintos
df.select(countDistinct("StockCode")).show()

# utilizando SparkSQL
spark.sql("""
    SELECT COUNT(DISTINCT StockCode) FROM vw_retail
""").show()
```

+-----+
count(DISTINCT StockCode)
+-----+
4070
+-----+
+-----+
count(DISTINCT StockCode)
+-----+
4070
+-----+

Figure 7.2: Utilização da função `countDistinct`.

O resultado acima indica que existem 4070 valores ou entradas distintas na coluna “StockCode”.

### 7.1.3. approx\_count\_distinct

Ao trabalhar com quantidades de dados realmente grandes, a contagem distinta exata de valores pode ser irrelevante. Ao invés disso, uma aproximação pode ser utilizada para retornar resultados com um certo grau de acuracidade de uma forma bem mais

veloz computacionalmente.

Neste ponto, a função `approx_count_distinct` surge como uma alternativa à `countDistinct` e com um parâmetro adicional: a taxa máxima de erro permitida. A figura 7.3 exemplifica sua aplicação no mesmo conjunto de dados.

```
# Importando função
from pyspark.sql.functions import approx_count_distinct

# Contando valores distintos com taxa de erro permitida
df.select(approx_count_distinct("StockCode", 0.1)).show()

# Via SparkSQL
spark.sql("""
    SELECT approx_count_distinct(StockCode, 0.1) FROM vw_retail
""").show()

+-----+
|approx_count_distinct(StockCode)| 
+-----+
|                            3364|
+-----+ 

+-----+
|approx_count_distinct(StockCode)| 
+-----+
|                            3364|
+-----+
```

Figure 7.3: Utilização da função `approx_count_distinct`.

Os resultados são significativamente diferentes dos obtidos com o `countDistinct` na figura ?? pois, de certa forma, estamos falando de uma alta taxa de erro permitida (10%) e um dataset bem longe do que podemos chamar de Big Data (cenário onde a contagem distinta aproximada pode ser extremamente relevante). Em contrapartida, pode-se dizer que os resultados com a função de aproximação são obtidos de forma bem mais rápida que a contagem distinta exata.

#### 7.1.4. `first` e `last`

Intuitivamente, as funções `first` e `last` permitem retornar, respectivamente, o primeiro e o último valor de um DataFrame. A figura 7.4 traz um exemplo de seleção do

primeiro e último valor da coluna “SparkSQL” via DataFrames e via SparkSQL.

```
# Importando funções
from pyspark.sql.functions import first, last

# Retornando valores
df.select(
    first("StockCode").alias("first"),
    last("StockCode").alias("last")
).show()

# Via SparkSQL
spark.sql("""
    SELECT
        first(StockCode) AS first,
        last(StockCode) AS last
    FROM vw_retail
""").show()
```

first	last
85123A	22138

first	last
85123A	22138

Figure 7.4: Utilização das funções `first` e `last`.

### 7.1.5. min e max

De forma direta e objetiva, a figura 7.5 exemplifica o retorno dos valores mínimo e máximo da coluna “Quantity” do DataFrame utilizado como exemplo através das funções `min` e `max`.

### 7.1.6. sum

Outra tarefa a ser explicada de forma direta e objetiva é a soma. A figura 7.6 demonstra uma soma de valores da coluna “Quantity” através da função `sum`.

```
# Importando funções
from pyspark.sql.functions import min, max

# Executando consulta
df.select(
    min("Quantity").alias("min"),
    max("Quantity").alias("max")
).show()

# Via SparkSQL
spark.sql("""
    SELECT
        min(Quantity) AS min,
        max(Quantity) AS max
    FROM vw_retail
""").show()
```

min	max
-80995	80995

min	max
-80995	80995

Figure 7.5: Utilização das funções `min` e `max`.

### 7.1.7. `sumDistinct`

Talvez menos conhecida ou praticamente exercita que o `sum` “tradicional”, a função `sumDistinct` pode ser utilizada para somar apenas valores distintos de uma coluna. A figura 7.7 traz um exemplo de aplicação.

### 7.1.8. `avg`

A média de uma coluna numérica pode ser calculada através da função `avg`. Em expressões strings, pode-se também utilizar a expressão `mean` para um cálculo análogo.

Dessa forma, a figura 7.8 exemplifica três diferentes cálculos para a média: divisão entre a soma e a contagem (`sum / count`), utilizando a função `avg` e utilizando

```
# Importando função
from pyspark.sql.functions import sum

# Realizando consulta
df.select(sum("Quantity")).show()

# Via SparkSQL
spark.sql("""
    SELECT sum(Quantity) FROM vw_retail
""").show()
```

+-----+
sum(Quantity)
+-----+
5176450
+-----+

+-----+
sum(Quantity)
+-----+
5176450
+-----+

Figure 7.6: Utilização da função sum.

```
# Importando função
from pyspark.sql.functions import sum_distinct

# Realizando consulta
df.select(sum_distinct("Quantity")).show()

# Via SparkSQL
spark.sql("""
    SELECT sum(DISTINCT Quantity) FROM vw_retail
""").show()
```

+-----+
sum(DISTINCT Quantity)
+-----+
29310
+-----+

+-----+
sum(DISTINCT Quantity)
+-----+
29310
+-----+

Figure 7.7: Utilização da função sum\_distinct.

a expressão `mean`.

```
# Importando funções
from pyspark.sql.functions import sum, count, avg, expr

# Realizando seleção
df.select(
    (sum("Quantity") / count("Quantity")).alias("sum/count"),
    avg("Quantity").alias("avg"),
    expr("mean(Quantity)").alias("mean")
).show()
```

sum/count	avg	mean
9.55224954743324	9.55224954743324	9.55224954743324

Figure 7.8: Utilização da função `avg`.

### 7.1.9. `var` e `stddev`

O cálculo da média abre margem para questões envolvendo os conceitos de variância e desvio padrão: ambos são medidas para avaliar o “espalhamento” dos dados ao redor da média. Em resumo, suas definições são:

- **Variância:** média das diferenças quadradas da média;
- **Desvio Padrão:** raíz quadrada da variância.

No Spark (e em diversos outros sistemas), existem separações para estes cálculos considerando a **população** ou a **amostra**. Para cada cenário, diferentes funções devem ser utilizadas. Em caso de não especificação da função de população ou amostra, isto é, considerando as funções `variance` ou `stddev`, o Spark realiza o cálculo com a fórmula que considera a **amostra**. A figura 7.9 envolve diferentes cenários de utilização do cálculo de variância e desvio padrão na coluna “Quantity” da base de dados.

```

# Importando funções
from pyspark.sql.functions import variance, stddev, \
    var_pop, stddev_pop, \
    var_samp, stddev_samp

# Realizando consultas
df.select(
    var_samp("Quantity").alias("var_samp"),
    stddev_samp("Quantity").alias("stddev_samp"),
    var_pop("Quantity").alias("var_pop"),
    stddev_pop("Quantity").alias("stddev_pop")
).show()

# Sem especificação do público
df.select(
    variance("Quantity").alias("variance"),
    stddev("Quantity").alias("stddev")
).show()

```

var_samp	stddev_samp	var_pop	stddev_pop
47559.39140929892	218.08115785023455	47559.30364660923	218.08095663447835

variance	stddev
47559.39140929892	218.08115785023455

**Figure 7.9:** Utilização das funções var e stddev

### 7.1.10. skewness e kurtosis

*Skewness* e *Kurtosis* são ambas medidas para avaliação do comportamento da distribuição dos dados.

- **Skewness:** indica a assimetria dos valores ao redor da média;
- **Kurtosis:** indica uma medida de “causa” ou extremidade dos dados.

A figura 7.10 traz exemplos de execução de ambas as funções e, por mais que o conteúdo aqui proposto não tenha aprofundamento teórico sobre a matemática por trás das medidas, é importante conhecer que tanto **skewness** quanto **kurtosis** podem ser estatisticamente relevantes em cenários de modelagem de dados.

```
# Importando funções
from pyspark.sql.functions import skewness, kurtosis

# Definindo consulta
df.select(
    skewness("Quantity"),
    kurtosis("Quantity")
).show()

# Via SparkSQL
spark.sql("""
    SELECT
        skewness(Quantity),
        kurtosis(Quantity)
    FROM vw_retail
""").show()
```

skewness(Quantity)	kurtosis(Quantity)
-0.26407557610528376	119768.05495530753
skewness(Quantity)	kurtosis(Quantity)
-0.26407557610528376	119768.05495530753

Figure 7.10: Utilização das funções `skewness` e `kurtosis`

### 7.1.11. `corr` e `covar`

Até aqui, foram discutidas agregações envolvendo uma única coluna. Os resultados das funções abordadas nesta subseção trazem uma relação estatística entre duas colunas em um único valor.

A correlação entre duas colunas (extraída pela função `corr`) retorna a correlação utilizando o coeficiente Pearson e seu resultado abrange o intervalo entre -1 e 1.

A covariância, assim como a variância, necessita de uma especificação de cálculo baseada na **população** ou na **amostra**. Dessa forma, as funções responsáveis por calcular a covariância são `covar_pop` ou `covar_samp`. A figura 7.11 exemplifica as aplicações de ambas as funções.

```
# Importando funções
from pyspark.sql.functions import corr, covar_pop, covar_samp

# Definindo consulta
df.select(
    corr("Quantity", "UnitPrice").alias("corr"),
    covar_pop("Quantity", "UnitPrice").alias("covar_pop"),
    covar_samp("Quantity", "UnitPrice").alias("covar_samp")
).show()
```

corr	covar_pop	covar_samp
-0.00123492454487...	-26.058713170968105	-26.058761257937057

Figure 7.11: Utilização das funções corr e covar

### 7.1.12. Agregando para Tipos Complexos

No Spark, as funções de agregação não estão restritas apenas ao universo de variáveis numéricas e utilização de fórmulas para retorno de números. Em aplicações específicas, é possível aplicar agregação em outras colunas para retornar **tipos complexos**.

Um exemplo prático é a coleta de uma lista de valores presentes em uma coluna. A figura 7.12 ilustra a aplicação das funções collect\_set e collect\_list.

```
# Importando funções
from pyspark.sql.functions import collect_set, collect_list, size

# Definido consultas
df.select(
    collect_set("Country"),
    collect_list("Country")
).show()
```

collect_set(Country)	collect_list(Country)
[Portugal, Italy, ...]	[United Kingdom, ...]

Figure 7.12: Utilização das funções collect\_set e collect\_list

Em geral, a “fabricação” de tipos complexos através da agregação pode ser uti-

lizada fortemente para acesso programático, em pipelines de dados ou para passar listas completas para UDFs. No exemplo da figura 7.12, as funções de agregação são aplicadas na coluna “Country”, resultando assim em listas com todos os elementos presentes no atributo e que podem ser trabalhadas posteriormente em transformações adicionais.

## 7.2. Agrupamento

Até este ponto, foram abordadas diversas funções de agregação aplicadas em um DataFrame por completo. Uma outra aplicação familiar é a agregação de dados utilizando categorias ou grupos.

A sintaxe para realizar os agrupamentos é proposta através do método `groupBy()`, onde são fornecidas as colunas (normalmente categóricas) alvo do agrupamento. Na sequência, dentro do método `agg()`, pode-se então especificar as agregações a serem realizadas (funções abordadas anteriormente). A figura 7.13 ilustra um exemplo de contagem e soma de itens para cada “InvoiceNo” e “CustomerId”.

```
# Importando funções
from pyspark.sql.functions import count, sum, avg

# Definindo consulta com groupBy
df.groupBy("InvoiceNo", "CustomerId").agg(
    count("Quantity").alias("count_qty"),
    sum("Quantity").alias("sum_qty")
).show(5)
```

InvoiceNo	CustomerId	count_qty	sum_qty
536846	14573	76	134
537026	12395	12	528
537883	14437	5	60
538068	17978	12	499
538279	14952	7	472

only showing top 5 rows

**Figure 7.13:** Exemplo de agrupamento em Spark utilizando múltiplas colunas categóricas como alvo e múltiplas funções de agregação.

Em outro exemplo ilustrado pela figura 7.14, o agrupamento é proposto através de expressões em strings em uma maior liberdade de utilização de funções que, como informado anteriormente ao longo deste resumo, se assemelham à sintaxe SQL.

```
# Agrupando com expressões
df.groupBy("InvoiceNo", "CustomerId").agg(
    expr("round(avg(Quantity), 2) AS avg_qty"),
    expr("round(stddev(Quantity), 2) AS stddev")
).show(5)

+-----+-----+-----+-----+
|InvoiceNo|CustomerId|avg_qty|stddev|
+-----+-----+-----+-----+
|      536846|     14573|   1.76|  1.61|
|      537026|     12395|  44.0| 48.29|
|      537883|     14437|  12.0|  0.0|
|      538068|     17978| 41.58|123.03|
|      538279|     14952|  67.43| 76.8|
+-----+-----+-----+-----+
only showing top 5 rows
```

Figure 7.14: Exemplo de agrupamento em Spark utilizando uma sintaxe de expressões em strings.

Por fim, de forma ilustrativa, a figura 7.15 exemplifica a obtenção de resultados de agrupamentos utilizando unicamente SparkSQL.

## 7.3. Funções Window

A aplicação do processo de *windowing* em análises de dados pode parecer relativamente complexo no início, mas sua importância é notória em um vasto leque de aplicações. De certa forma, a principal diferença em analisar dados com funções **window** é a especificação de uma literal “janela” de análise, a qual é definida utilizando referências na própria base de dados.

O livro *Spark - The Definitive Guide* traz uma definição comparativa interessante entre *group by* e funções *window*:

Um *group by* coleta os dados em um formato que cada linha seja direcionada para um grupo. Uma função *window* calcula um valor de retorno para cada linha de entrada da tabela com base em um **grupo de linhas**,

```
# Realizando os mesmos cálculos via SparkSQL
spark.sql("""
    SELECT
        InvoiceNo,
        CustomerId,
        count(Quantity) AS count,
        sum(Quantity) AS sum_qty,
        round(avg(Quantity), 2) AS avg_qty,
        round(stddev(Quantity), 2) AS stddev

    FROM vw_retail

    GROUP BY
        InvoiceNo,
        CustomerId
""").show(5)

+-----+-----+-----+-----+-----+
|InvoiceNo|CustomerId|count|sum_qty|avg_qty|stddev|
+-----+-----+-----+-----+-----+
| 536846|     14573|   76|    134|   1.76|  1.61|
| 537026|     12395|   12|    528|   44.0| 48.29|
| 537883|     14437|    5|     60|   12.0|  0.0|
| 538068|     17978|   12|    499|   41.58|123.03|
| 538279|     14952|    7|    472|   67.43|  76.8|
+-----+-----+-----+-----+-----+
only showing top 5 rows
```

**Figure 7.15:** Utilizando SparkSQL para propor agrupamentos em views ou tabelas temporárias salvas no Spark.

chamado de *frame*. Neste caso, cada linha pode ser direcionada para um ou mais *frames*. Um caso comum de uso é analisar a média móvel de um valor onde cada linha representa um dia. Neste exemplo, cada linha será direcionada em 7 diferentes *frames*. O Spark suporta três tipos de funções *window*: ranking, analytics e aggregate.

Para demonstração, será construído um *case* para análise da quantidade máxima de itens adquirida por cada cliente em cada janela temporal. Como primeiro passo, será criada uma nova coluna no DataFrame utilizado nos exemplos para extrair a data da coluna “InvoiceDate” de modo a utilizarmos esta informação futuramente na definição da janela de análise. A figura 7.16 ilustra a forma utilizada para realização deste processo de conversão. Ao final, é gerada uma nova coluna (de-

nominada “date”) com tipo primitivo de data (e não mais string como a coluna original “InvoiceDate”).

```
# Importando funções
from pyspark.sql.functions import col, to_date

# Transformando coluna de data
df_date = df.withColumn(
    "date", to_date(col("InvoiceDate"), "MM/d/yyyy H:mm")
)
df_date.createOrReplaceTempView("vw_retail_date")

# Verificando dados
df_date.select("InvoiceDate", "date").show(5)

+-----+-----+
| InvoiceDate|      date|
+-----+-----+
|12/1/2010 8:26|2010-12-01|
|12/1/2010 8:26|2010-12-01|
|12/1/2010 8:26|2010-12-01|
|12/1/2010 8:26|2010-12-01|
|12/1/2010 8:26|2010-12-01|
+-----+-----+
only showing top 5 rows
```

**Figure 7.16:** Transformação da coluna “InvoiceDate”, originalmente em string, na coluna “date” do tipo Date. Esta coluna será utilizada como ponto chave na definição da janela de análise da função *window*.

### 7.3.1. Especificando Janela

Como primeiro passo, a especificação da função *window* será definida como uma forma de formalizar a janela de análise a ser utilizada como alvo para as funções de agregação aplicadas. Este procedimento é feito através da classe *Window* importada do módulo *pyspark.sql.window*. Em seu conteúdo, são utilizadas as cláusulas *partitionBy*, *orderBy* e *rowsBetween*. A figura 7.17 ilustra o processo de criação via código e a tabela 7.1 pode ser utilizada como base para entendimento geral da construção da janela de análise.

```

# Importando funções
from pyspark.sql.window import Window
from pyspark.sql.functions import desc, max, rank, dense_rank

# Especificando janela de análise
windowSpec = Window\
    .partitionBy("CustomerId", "date")\
    .orderBy(desc("Quantity"))\
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

```

**Figure 7.17:** Especificação da janela de análise através da classe `Window`. Nela, são definidas as partições de agrupamento, a ordenação aplicada e a configuração do *frame* com base nas linhas de entrada.

**Table 7.1:** Definição das cláusulas utilizadas na construção da janela de análise da função `window`.

Cláusula	Descrição
<code>partitionBy</code>	Define como os grupos da janela serão “quebrados”. No caso do exemplo, a especificação comporta uma análise por cliente e data.
<code>orderBy</code>	Especifica a ordem na qual a partição definida será analisada. No exemplo, a análise é ordenada por quantidade de forma descendente.
<code>rowsBetween</code>	Determina a especificação do <i>frame</i> e indica quais linhas serão incluídas no <i>frame</i> com base na referência da linha de entrada.

### 7.3.2. Agregando Sobre a Janela

Sequencialmente, são definidas então as funções de agregação a serem aplicadas na janela `windowSpec` definida. A figura 7.18 ilustra os cálculos da quantidade máxima de itens (`maxPurchaseQuantity`), o ranking de cada quantidade (`purchaseRank`) e o dense ranking de cada quantidade (`purchaseDenseRank`). Todas as funções definidas são aplicadas à janela `windowSpec` através da cláusula `over()`. Adicionalmente, a tabela 7.2 detalha um pouco melhor as agregações aplicadas.

```
# Especificando agregação
maxPurchaseQty = max(col("Quantity")).over(windowSpec)

# Criando rankings
purchaseRank = rank().over(windowSpec)
purchaseDenseRank = dense_rank().over(windowSpec)
```

**Figure 7.18:** Construindo e aplicando agregações sobre a janela de análise especificada na figura

7.17

**Table 7.2:** Definição das agregações aplicadas à janela de análise especificada.

Cláusula	Descrição
max(Quantity)	Computa a quantidade máxima de itens aplicada à janela especificada (partições por cliente e data).
rank()	Calcula o rank de cada quantidade aplicada à janela (partições por cliente e data). A função rank() continua incrementando o <i>ranking</i> mesmo ao encontrar elementos repetidos.
dense_rank()	Calcula o rank de cada quantidade aplicada à janela (partições por cliente e data). A função dense_rank() não incrementa o <i>ranking</i> ao encontrar elementos repetidos.

### 7.3.3. Construindo Consulta

Por fim, após as definições da janela e das agregações a serem aplicadas sobre a janela, é chegado o momento de unir todos os elementos criados em uma consulta única. A figura 7.19 utiliza os objetos e as definições de colunas construídas e ilustradas pela ?? para montar uma cláusula select de modo a retornar a análise de janela especificada.

Ainda sobre a figura 7.19, é possível perceber como a quantidade máxima por janela se comporta para cada partição definida (cliente e data). Além disso, a diferença entre rank() e dense\_rank() pode ser analisada em detalhes, principalmente quando tem-se quantidades duplicadas.

De modo a enriquecer esta seção, a figura 7.20 realiza todo o procedimento

```
# Selecionando dados
df_date.where("CustomerId IS NOT NULL").orderBy("CustomerId")\
    .select(
        "CustomerId",
        "date",
        "Quantity",
        purchaseRank.alias("qtyRank"),
        purchaseDenseRank.alias("qtyDenseRank"),
        maxPurchaseQty.alias("maxPurchaseQty")
    ).show(20)
```

CustomerId	date	Quantity	qtyRank	qtyDenseRank	maxPurchaseQty
12346	2011-01-18	74215	1	1	74215
12346	2011-01-18	-74215	2	2	74215
12347	2010-12-07	36	1	1	36
12347	2010-12-07	30	2	2	36
12347	2010-12-07	24	3	3	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	6	17	5	36
12347	2010-12-07	6	17	5	36

only showing top 20 rows

Figure 7.19: Construção de consulta utilizando os elementos de janela especificados.

especificado nas subseções acima utilizando SparkSQL.

## 7.4. Grouping Sets

Até este ponto, foram abordados agrupamentos envolvendo um conjunto de colunas como grupos e funções de agregação como valores para os tais grupos. Entretanto, em algumas aplicações, pode-se necessitar realizar agregações ao longo de **múltiplos grupos**. Para tal, são utilizados os *grouping sets* como uma ferramenta de baixo

```
# Aplicando a mesma análise via SparkSQL
spark.sql("""
    SELECT
        CustomerId,
        date,
        Quantity,
        rank(Quantity) OVER (PARTITION BY CustomerId, date
                             ORDER BY Quantity DESC NULLS LAST
                             ROWS BETWEEN
                             UNBOUNDED PRECEDING AND
                             CURRENT ROW) AS rank,
        dense_rank(Quantity) OVER (PARTITION BY CustomerId, date
                                   ORDER BY Quantity DESC NULLS LAST
                                   ROWS BETWEEN
                                   UNBOUNDED PRECEDING AND
                                   CURRENT ROW) AS dRank,
        max(Quantity) OVER (PARTITION BY CustomerId, date
                            ORDER BY Quantity DESC NULLS LAST
                            ROWS BETWEEN
                            UNBOUNDED PRECEDING AND
                            CURRENT ROW) AS maxPurchase
    FROM vw_retail_date WHERE CustomerId IS NOT NULL ORDER BY CustomerId
""").show(20)
```

CustomerId	date	Quantity	rank	dRank	maxPurchase
12346	2011-01-18	74215	1	1	74215
12346	2011-01-18	-74215	2	2	74215
12347	2010-12-07	36	1	1	36
12347	2010-12-07	30	2	2	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	24	3	3	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	12	4	4	36
12347	2010-12-07	6	17	5	36
12347	2010-12-07	6	17	5	36

**Figure 7.20:** Exemplo de construção de análise utilizando *windowing* puramente em SQL. Aqui, percebe-se alguns elementos chave de especificação em cada função de agregação definida (max, rank e dense\_rank).

nível para combinar conjuntos de agregação.

O uso dos *grouping sets* de maneira correta depende da utilização de uma base

de dados **sem nulos**. Caso contrário, valores incorretos poderão ser obtidos na análise. Seu uso está restrito ao SparkSQL e, para alcançar resultados semelhantes utilizando métodos de DataFrames, será necessário aplicar `rollup` e `cube` (abordados logo a seguir).

Visando propor um entendimento claro dos *grouping sets*, a figura 7.21 ilustra uma query em SparkSQL capaz de realizar a totalização de item (“Quantity”) por “CustomerId” e “StockCode”. Na query, é possível perceber a cláusula `GROUPING SETS` aplicada após o `GROUP BY` e especificando as colunas envolvidas no set de agrupamento.

No exemplo da figura 7.21, o resultado obtido é exatamente igual a um agrupamento simples aplicado apenas com a cláusula `GROUP BY`. A primeira pergunta que se faz é: qual então é a vantagem em utilizar *grouping sets*? A resposta pode ser visualizar na figura 7.22: ao modificar e customizar os elementos contidos na cláusula `GROUPING SETS` da query, é possível obter um resultado que totaliza uma coluna de agregação em relação a múltiplos grupos. Neste caso, o resultado inclui também um total de itens independente de clientes e códigos de estoque e isto está representado quando as colunas “CustomerId” e “StockCode” são nulas. No fundo, *grouping sets* atua como **uma junção de agrupamentos em um mesmo resultado**.

Como informado anteriormente, para configuração de consultas com a mesma abordagem em DataFrames, utiliza-se os métodos `rollup` e `cube`.

## 7.4.1. Rollups

Um *rollup* pode ser definido como uma agregação multidimensional que realiza uma variedade de cálculos de agrupamento em uma única operação. A figura 7.23 exemplifica sua aplicação a partir da agregação entre as colunas “Date” e “Country”, totalizando a quantidade “Quantity” em um cenário multidimensional.

Detalhando ainda mais a figura 7.23, é possível perceber que, além de somar a quantidade para cada data e cada país da base, o resultado também comporta uma summarização de quantidade total, independente da data e do país. Isto está

```

# Dropando nulos e criando nova view
df_not_null = df_date.na.drop()
df_not_null.createOrReplaceTempView("vw_retail_not_null")

# Aplicando grouping set
spark.sql("""
    SELECT
        CustomerId,
        StockCode,
        sum(Quantity)

    FROM vw_retail_not_null

    GROUP BY CustomerId, StockCode

    GROUPING SETS((CustomerId, StockCode))

    ORDER BY CustomerId DESC, StockCode DESC
""").show(5)

```

CustomerId	StockCode	sum(Quantity)
18287	85173	48
18287	85040A	48
18287	85039B	120
18287	85039A	96
18287	84920	4

only showing top 5 rows

**Figure 7.21:** Construindo conjuntos de agrupamento utilizando a cláusula GROUPING SETS em SparkSQL. No exemplo da figura, os agrupamentos selecionados na cláusula indicam que a agregação será realizada nos níveis das colunas indicadas o que, de fato, traz o mesmo efeito de um GROUP BY “padrão”.

representado quando as colunas de agrupamento são nulas.

### 7.4.2. Cube

Um cube leva o rollup para um nível mais profundo: ao invés de tratar os elementos de forma hierárquica, o processo de cube realiza a mesma agregação ao longo de **todas as dimensões**. No exemplo proposto, isto significaria adicionar respostas para as seguintes tópicos:

```

spark.sql("""
    SELECT
        CustomerId,
        StockCode,
        sum(Quantity)

    FROM vw_retail_not_null

    GROUP BY CustomerId, StockCode

    GROUPING SETS((CustomerId, StockCode), ())
    ORDER BY sum(Quantity) DESC, CustomerId DESC, StockCode DESC
""").show(5)

```

CustomerId	StockCode	sum(Quantity)
null	null	4906888
13256	84826	12540
17949	22197	11692
16333	84077	10080
16422	17003	10077

only showing top 5 rows

**Figure 7.22:** Configuração da cláusula GROUPING SETS para adição de diferentes níveis de agrupamento. No exemplo da figura, além da agregação “padrão” pelas colunas indicadas, um nível adicional é responsável por totalizar a função de agregação independente dos grupos selecionados e indicados no agrupamento.

- Quantidade total ao longo de todas as datas e países (*grouping set* vazio - data e país)
- Quantidade total para cada data ao longo de todos os países (*grouping set* por data)
- Quantidade total para cada país em cada data (*group by* padrão)
- Quantidade total para cada país ao longo de todas as datas (*grouping set* por país)

A figura 7.24 traz um exemplo prático de aplicação do método `cube` no mesmo cenário exemplificado anteriormente (totalização de quantidade por data e país). Na parte superior da figura, a ordenação por data (nulos primeiro) indica que a totaliza-

```
# Aplicando rollup
df_date.rollup("date", "Country")\
    .agg(sum("Quantity"))\
    .orderBy("Date")\
    .show(5)
```

date	Country	sum(Quantity)
null	null	5176450
2010-12-01	United Kingdom	23949
2010-12-01	null	26814
2010-12-01	Australia	107
2010-12-01	France	449

only showing top 5 rows

**Figure 7.23:** Exemplo de aplicação do método `rollup` para retorno de agregações multidimensionais.

ção de quantidade está sendo realizada por país. Na parte inferior da figura, a ordenação por país (nulos primeiro) indica a totalização por cada data. Quando as colunas “date” e “country” forem nulas, a totalização está sendo realizada para a base inteira. Quando “date” e “country” não forem nulas, a totalização é idêntica a um **group by** por data e país.

De modo a propor uma visão geral sobre as 4 possibilidades detalhadas no exemplo fornecido, a figura 7.25 aplica os filtros necessários para demonstrar os diferentes níveis de agrupamentos presentes no cubo criado. Para cada filtro, um nível de análise diferente é considerado, seja totalizando por data, país, ambas ou nenhuma.

### 7.4.3. Pivot

O processo de *pivot* basicamente indica a transformação de linhas em colunas. De forma direta e objetiva, a figura 7.26 cria uma coluna adicional na tabela para referenciar o mês de cada data presente. Sequencialmente, é realizado um pivot para transformar cada mês em uma coluna distinta da base. O agrupamento é feito por país e a agregação é feita pela soma de quantidade.

```
# Aplicando agregações multidimensionais
df_date.cube("Date", "Country")\
    .agg(sum("Quantity"))\
    .orderBy("Date")\
    .show(5)
```

```
+-----+-----+
|Date|          Country|sum(Quantity)|
+-----+-----+
|null|United Arab Emirates|      982|
|null|                  Italy|     7999|
|null|              Singapore|      5234|
|null|                 Finland|    10666|
|null|                  Greece|      1556|
+-----+-----+
only showing top 5 rows
```

```
df_date.cube("Date", "Country")\
    .agg(sum("Quantity"))\
    .orderBy("Country")\
    .show(5)
```

```
+-----+-----+
|      Date|Country|sum(Quantity)|
+-----+-----+
|2011-01-11|  null|      29093|
|2011-01-13|  null|      10114|
|2010-12-19|  null|       3795|
|2011-01-06|  null|      22461|
|2011-02-10|  null|      11447|
+-----+-----+
only showing top 5 rows
```

**Figure 7.24:** Criação de cubo e ordenação por diferentes colunas para exemplificar as diferentes agregações e agrupamentos inclusos.

```

# Criando cubo
df_cube = df_date.cube("Date", "Country")\
    .agg(sum("Quantity"))

# Totalização por país
df_cube.where("date is null").show(5)

# Totalização por data
df_cube.where("country is null").show(5)

# Totalização independente de data e país
df_cube.where("date is null AND country is null").show()

# Totalização por data e país
df_cube.where("date is not null AND country is not null").show(5)

```

Date	Country	sum(Quantity)
null	Finland	10666
null	Lithuania	652
null	Poland	3653
null	Iceland	2458
null	United Arab Emirates	982

only showing top 5 rows

Date	Country	sum(Quantity)
2010-12-17	null	16069
2010-12-14	null	20098
2011-02-13	null	2715
2011-01-06	null	22461
2011-02-17	null	14544

only showing top 5 rows

Date	Country	sum(Quantity)
null	null	5176450

Date	Country	sum(Quantity)
2010-12-17	Germany	1657
2011-04-01	France	327
2011-04-07	EIRE	390
2011-01-12	United Kingdom	8622
2011-01-19	Spain	48

only showing top 5 rows

**Figure 7.25:** Exemplo de retorno dos diferentes níveis de agregação contidos no cubo. Na figura, é possível visualizar claramente como cada filtro se comporta em meio à respectiva totalização.

```
# Criando coluna de mês e pivotando dados
df_date.withColumn("month_dt", month("date"))\
    .groupBy("Country")\
    .pivot("month_dt")\
    .agg(sum("Quantity"))\
    .show(5)
```

Country	1	2	3	4	5	6	7	8	9	10	11	12
Sweden	3096	250	5263	310	2829	404	6006	1308	4344	6151	1962	3714
Singapore	1091	null	null	1384	null	null	2160	null	null	599	null	null
Germany	8906	4083	7675	5692	12951	7348	8991	9560	11028	17636	12922	10656
RSA	null	null	null	null	null	null	null	null	null	352	null	null
France	9155	5301	8639	2216	9780	9441	5656	7948	12912	13737	17086	8609

only showing top 5 rows

**Figure 7.26:** Realizando pivot nos dados.

---

# 8

## Joins

Após uma longa e gratificante jornada nas principais formas de transformar dados em Spark, sejam estes presentes como `DataFrames` ou como `views`, é chegado o momento de conhecer um pouco melhor técnicas de *joins*.

Até o momento, todas as transformações aplicadas tiveram, como origem, uma única fonte de dados. Deste ponto em diante, as técnicas de *joins* irão auxiliar em situações onde necessita-se juntar lateralmente diferentes fontes de dados a partir de uma chave de cruzamento.

Além do conhecimento básico de sintaxe para aplicação de operações de *joins*, este capítulo trará uma visão essencial sobre como o Spark atua no ato de execução de *joins* dentro do cluster. Este conhecimento é extremamente importante em ambientes produtivos para otimização de fluxos e mitigação de erros de memória ou problemas do tipo.

## 8.1. Definição Geral de Joins

De maneira geral, temos o *join* como uma operação que utiliza dois *datasets*, chamados de *left* e de *right*, em uma comparação de valor de uma ou mais *chaves* presentes em ambos.

A forma mais comum de *joins* são os *equi-joins* em cenários de comparação de igualdade entre as expressões da chave. Se os valores das chaves de ambos os datasets forem iguais, o Spark irá combinar ambas as bases.

Considerando todos os tipos de *joins* possíveis, temos:

- **Inner Joins:** mantém linhas com chaves que existem nas bases *left* e *right*
- **Outer Joins:** mantém linhas com chaves que podem existir nas bases *left* ou *right*
- **Left Outer Joins:** mantém linhas com chaves existentes no dataset *left*
- **Right Outer Joins:** mantém linhas com chaves existentes no dataset *right*

Existem outros tipos de *joins* presentes e detalhados no livro e que podem ser consultados eventualmente.

## 8.2. DataFrames para Testes

Como forma de proporcionar uma visão prática de aplicação dos mais variados tipos de *joins*, o livro estabelece a construção de “DataFrames de exemplo” apenas para testar os resultados das consultas. O processo de criação dos DataFrames está ilustrado pela figura 8.1 e cada uma das bases pode ser visualizada pela figura 8.2.

Visão geral das bases:

```

# Criando DataFrame de pessoas
person = spark.createDataFrame([
    (0, "Bill Chambers", 0, [100]),
    (1, "Matei Zaharia", 1, [500, 250, 100]),
    (2, "Michael Armbrust", 1, [250, 100])
]).toDF("id", "name", "grad_program", "spark_status")

# Criando DataFrame de programas de grauação
gradProgram = spark.createDataFrame([
    (0, "Masters", "School of Information", "UC Berkeley"),
    (2, "Masters", "EECS", "UC Berkeley"),
    (1, "Ph.D", "EECS", "UC Berkeley")
]).toDF("id", "degree", "department", "school")

# Criando DataFrame de status Spark
sparkStatus = spark.createDataFrame([
    (500, "Vice President"),
    (250, "PMC Member"),
    (100, "Contributor")
]).toDF("id", "status")

# Registrando views
person.createOrReplaceTempView("person")
gradProgram.createOrReplaceTempView("grad_program")
sparkStatus.createOrReplaceTempView("spark_status")

```

**Figure 8.1:** Código de criação de DataFrames a serem utilizados como exemplos nas construções de *joins*

**Table 8.1:** Descrição geral dos DataFrames utilizados na aplicação prática de *joins* neste capítulo.

DataFrame	Descrição
person	Descreve diferentes pessoas que participaram de diferentes programas de graduação (grad_program) e possuem diferentes papéis no universo Spark (spark_status)
gradProgram	Descreve detalhes sobre os programas de graduação
sparkStatus	Descreve detalhes sobre os papéis no universo Spark

```
# Visualisando bases
person.show()
gradProgram.show()
sparkStatus.show()

+---+-----+-----+
| id|      name|grad_program| spark_status|
+---+-----+-----+
| 0| Bill Chambers|          0| [100]|
| 1| Matei Zaharia|          1| [500, 250, 100]|
| 2|Michael Armbrust|          1| [250, 100]|
+---+-----+-----+

+---+-----+-----+
| id| degree|      department|      school|
+---+-----+-----+
| 0|Masters|School of Informa...|UC Berkeley|
| 2|Masters|                  EECS|UC Berkeley|
| 1|   Ph.D|                  EECS|UC Berkeley|
+---+-----+-----+


+---+
| id|      status|
+---+
|500|Vice President|
|250|    PMC Member|
|100| Contributor|
+---+
```

Figure 8.2: Visualização dos DataFrames criados.

## 8.3. Tipos de Joins

### 8.3.1. Inner Join

Basicamente, os *joins* em Spark são realizados a partir da aplicação do método `join()` de DataFrames. Em SparkSQL, a sintaxe SQL prevalece. De forma direta, a figura 8.3 demonstra um exemplo de **inner join** aplicado entre as bases.

Para complementar o entendimento sobre o método `join` de DataFrames, a figura 8.4 traz a mesma aplicação de um **inner join** (exemplificada pela figura 8.3), porém com a descrição explícita dos argumentos. Com isso, percebe-se que, por padrão, o método `join` aplica um **inner join** e que, de certa forma, este pode ser explicitado através do argumento `how`.

```
# Criando expressão join
join_expr = (person['grad_program'] == gradProgram['id'])

# Aplicando join
person.join(gradProgram, join_expr).show()
```

id	name   grad_program	spark_status	id	degree	department	school
0	Bill Chambers	0   [100]	0   Masters	School of Informa...	UC Berkeley	
1	Matei Zaharia	1   [500, 250, 100]	1   Ph.D		EECS	UC Berkeley
2	Michael Armbrust	1   [250, 100]	1   Ph.D		EECS	UC Berkeley

**Figure 8.3:** Exemplo de aplicação de **inner join** para retornar uma tabela com detalhes do programa de graduação.

```
# Aplicando joins com argumentos explícitos
person.join(other=gradProgram, on=join_expr, how="inner").show()
```

id	name   grad_program	spark_status	id	degree	department	school
0	Bill Chambers	0   [100]	0   Masters	School of Informa...	UC Berkeley	
1	Matei Zaharia	1   [500, 250, 100]	1   Ph.D		EECS	UC Berkeley
2	Michael Armbrust	1   [250, 100]	1   Ph.D		EECS	UC Berkeley

**Figure 8.4:** Aplicação de inner join através da passagem explícita dos argumentos do método `join()`.

Por fim, a figura 8.5 ilustra o mesmo resultado através da aplicação de um **inner join** exclusivamente via SparkSQL.

```
# Em SparkSQL
spark.sql("""
    SELECT
        *
    FROM person
    INNER JOIN grad_program
        ON person.grad_program = grad_program.id
""").show()
```

id	name   grad_program	spark_status	id	degree	department	school
0	Bill Chambers	0   [100]	0   Masters	School of Informa...	UC Berkeley	
1	Matei Zaharia	1   [500, 250, 100]	1   Ph.D		EECS	UC Berkeley
2	Michael Armbrust	1   [250, 100]	1   Ph.D		EECS	UC Berkeley

**Figure 8.5:** Exemplo de aplicação de **inner join** via SparkSQL.

### 8.3.2. Left Join

Left Joins são operações que avaliam duas bases de dados e incluem todas as linhas da base da esquerda e também linhas da direita que são avaliadas como verdadeiras na expressão vinculada. A figura 8.6 exemplifica uma consulta de pessoas em cada programa de graduação. Nela, é possível perceber que o programa de graduação com `id` igual a 2 não contém nenhuma pessoa e, neste caso, um valor nulo é retornado.

```
# Aplicando join
gradProgram.join(person, join_expr, how='left_outer').show()
```

	id	degree	department	school	id	name	grad_program	spark_status
	0	Masters	School of Informa...	UC Berkeley	0	Bill Chambers	0	[100]
	1	Ph.D	EECS	UC Berkeley	1	Matei Zaharia	1	[500, 250, 100]
	1	Ph.D	EECS	UC Berkeley	2	Michael Armbrust	1	[250, 100]
	2	Masters	EECS	UC Berkeley	null	null	null	null

Figure 8.6: Exemplo de aplicação de **left join**.

### 8.3.3. Right Join

Right Joins são mais incomuns que operações de inner e left, porém podem ser úteis em cenários onde se deseja manter todas as linhas da base da direita ao mesmo tempo que as linhas da base da esquerda que convergem para a expressão são trazidas. A figura 8.7 exemplifica este cenário.

```
# Aplicando join
person.join(gradProgram, join_expr, how="right_outer").show()
```

	id	name	grad_program	spark_status	id	degree	department	school
	0	Bill Chambers	0	[100]	0	Masters	School of Informa...	UC Berkeley
	1	Matei Zaharia	1	[500, 250, 100]	1	Ph.D	EECS	UC Berkeley
	2	Michael Armbrust	1	[250, 100]	1	Ph.D	EECS	UC Berkeley
	null	null	null	null	2	Masters	EECS	UC Berkeley

Figure 8.7: Exemplo de aplicação de **right join**.

### 8.3.4. Left Semi e Left Anti Join

Ainda mais incomuns, porém aplicáveis em algumas ocasiões, os tipos de *joins* Left Semi e Left Anti são detalhados abaixo:

- *Left Semi Join*: avalia a expressão de junção e retorna apenas as linhas que aparecem na base da esquerda;
- *Left Anti Join*: avalia a expressão de junção e retorna apenas as linhas que não aparecem na base da esquerda. Contrário ao *left semi join*.

A figura 8.8 ilustra aplicações destes dois cenários.

```
# Aplicando left semi join
gradProgram.join(person, join_expr, how="left_semi").show()

# Aplicando left anti join
gradProgram.join(person, join_expr, how="left_anti").show()

+---+-----+-----+
| id| degree|      department|      school|
+---+-----+-----+
| 0|Masters|School of Informa...|UC Berkeley|
| 1|   Ph.D|                  EECS|UC Berkeley|
+---+-----+-----+

+---+-----+-----+
| id| degree|department|      school|
+---+-----+-----+
| 2|Masters|      EECS|UC Berkeley|
+---+-----+-----+
```

Figure 8.8: Exemplo de aplicação de **left semi join** e **left anti join**.

### 8.3.5. Considerações Finais

Existem ainda outros tipos de joins não explorados neste material porém presentes no livro: Natural Joins e Cartesian Joins.

De forma geral, a tabela 8.2 traz detalhes importantes sobre cada um dos diferentes tipos de joins e a respectiva string a ser passada para o método `join()`.

**Table 8.2:** String de aplicação de joins no método `join()` do objeto DataFrame do Spark para execução dos mais variados tipos de junção de tabelas e bases de dados.

Tipo de Join	Parâmetro <code>how</code> do método <code>join</code>
Inner Join	<code>how="inner"</code>
Left Join	<code>how="left_outer"</code>
Right Join	<code>how="right_outer"</code>
Left Semi Join	<code>how="left_semi"</code>
Left Anti Join	<code>how="left_anti"</code>
Cross Join	<code>how="cross"</code>

## 8.4. Joins em Tipos Complexos

Por mais que o título da seção assuste, este tipo de operação é perfeitamente possível. Para validar sua aplicabilidade, é preciso lembrar que o DataFrame `person` contém uma coluna do tipo `array` (isto é, uma lista) com cada um dos papéis referenciados a cada linha (pessoa) da base de dados.

Como esta informação está originalmente em um formato `array`, pode parecer complexo realizar qualquer tipo de expressão *join* com este dado. Entretanto, é correto dizer que joins deste tipo podem ser implementados desde que a expressão aplicada retorne um booleano. Para entender um pouco sobre este conceito, é preciso relembrar algumas funções utilizadas no capítulo 6 deste material que podiam ser utilizadas em tipos complexos (mais especificamente, o tipo `array`). Neste caso, a figura 8.9 ilustra a aplicação da função `array_contains()` para validar se determinada string está presente em uma lista de elementos. Como pode ser visto, o resultado é um campo booleano que retorna `true` ou `false`.

Assim, entendendo a aplicabilidade da função `array_contains`, é possível criar uma consulta onde a expressão *join* verifica se a chave de cruzamento de uma das tabelas está presente no campo do tipo primitivo `array` da outra tabela, retornando assim verdadeiro caso o elemento esteja presente e falso caso o elemento não esteja presente. De forma geral, as figuras 8.10 e 8.11 trazem este cenário através da

```
# Importando função
from pyspark.sql.functions import array_contains, col

df_array.select(
    "split_desc",
    array_contains("split_desc", "WHITE").alias("flag_white")
).show(5, truncate=False)

+-----+-----+
|split_desc |flag_white|
+-----+-----+
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER] |true
|[WHITE, METAL, LANTERN] |true
|[CREAM, CUPID, HEARTS, COAT, HANGER] |false
|[KNITTED, UNION, FLAG, HOT, WATER, BOTTLE]|false
|[RED, WOOLLY, HOTTIE, WHITE, HEART.] |true
+-----+-----+
only showing top 5 rows
```

**Figure 8.9:** Exemplo de aplicação da função `array_contains` como uma forma de validar se um elemento está presente em um campo do tipo primitivo `array`

verificação da coluna `spark_status` em uma das bases juntamente com a coluna de `id` da base que descreve os status Spark. A diferença entre as figuras é que, na primeira, o processo é feito através de indexação de colunas em DataFrames e, na segunda, um processo de alteração de nome de coluna é aplicado previamente para evitar erros de ambiguidade (ambas as bases possuem a coluna `id`).

```
# Importando funções
person.join(
    other=sparkstatus,
    on=array_contains(person["spark_status"], sparkstatus["id"]))
.show()

+-----+-----+-----+-----+
| id | name | grad_program | spark_status | id | status |
+-----+-----+-----+-----+
| 0 | Bill Chambers | 0 | [100] | 100 | Contributor |
| 1 | Matei Zaharia | 1 | [500, 250, 100] | 500 | Vice President |
| 1 | Matei Zaharia | 1 | [500, 250, 100] | 250 | PMC Member |
| 1 | Matei Zaharia | 1 | [500, 250, 100] | 100 | Contributor |
| 2 | Michael Armbrust | 1 | [250, 100] | 250 | PMC Member |
| 2 | Michael Armbrust | 1 | [250, 100] | 100 | Contributor |
+-----+-----+-----+-----+
```

**Figure 8.10:** Execução de join com tipo primitivo complexo (indexação de colunas).

Por fim, este mesmo processo de join também pode ser realizado via SparkSQL.

Sua exemplificação pode ser visualizada na figura 8.12.

```
# Renomeando coluna antes de aplicar join
person.withColumnRenamed("id", "person_id")\
    .join(sparkStatus, on=expr("array_contains(spark_status, id)"))\
    .show()
```

person_id	name	grad_program	spark_status	id	status
0	Bill Chambers		[100]	100	Contributor
1	Matei Zaharia		[500, 250, 100]	500	Vice President
1	Matei Zaharia		[500, 250, 100]	250	PMC Member
1	Matei Zaharia		[500, 250, 100]	100	Contributor
2	Michael Armbrust		[250, 100]	250	PMC Member
2	Michael Armbrust		[250, 100]	100	Contributor

**Figure 8.11:** Execução de join com tipo primitivo complexo (renomeando colunas).

```
# Realizando consulta análoga em SparkSQL
spark.sql("""
    SELECT * FROM person AS p
    INNER JOIN spark_status AS s
        ON array_contains(p.spark_status, s.id)
""").show()
```

id	name	grad_program	spark_status	id	status
0	Bill Chambers		[100]	100	Contributor
1	Matei Zaharia		[500, 250, 100]	500	Vice President
1	Matei Zaharia		[500, 250, 100]	250	PMC Member
1	Matei Zaharia		[500, 250, 100]	100	Contributor
2	Michael Armbrust		[250, 100]	250	PMC Member
2	Michael Armbrust		[250, 100]	100	Contributor

**Figure 8.12:** Execução de join com tipo primitivo complexo via SparkSQL

## 8.5. Execução de Joins pelo Spark

Para entender como o Spark gerencia a execução de operações de *join*, é importante ter em mente dois principais tópicos:

- *node-to-node communication strategy*
- *per node computation strategy*

Mesmo que o detalhamento dessas duas estratégias citadas acima seja algo específico e direcionado apenas a possíveis desenvolvedores e contribuidores do próprio Spark, uma visão geral sobre as abordagens de alto nível pode auxiliar entusiastas em otimizar a performance de fluxos de trabalho que utilizam de tal funcional-

idade.

### 8.5.1. Estratégias de Comunicação

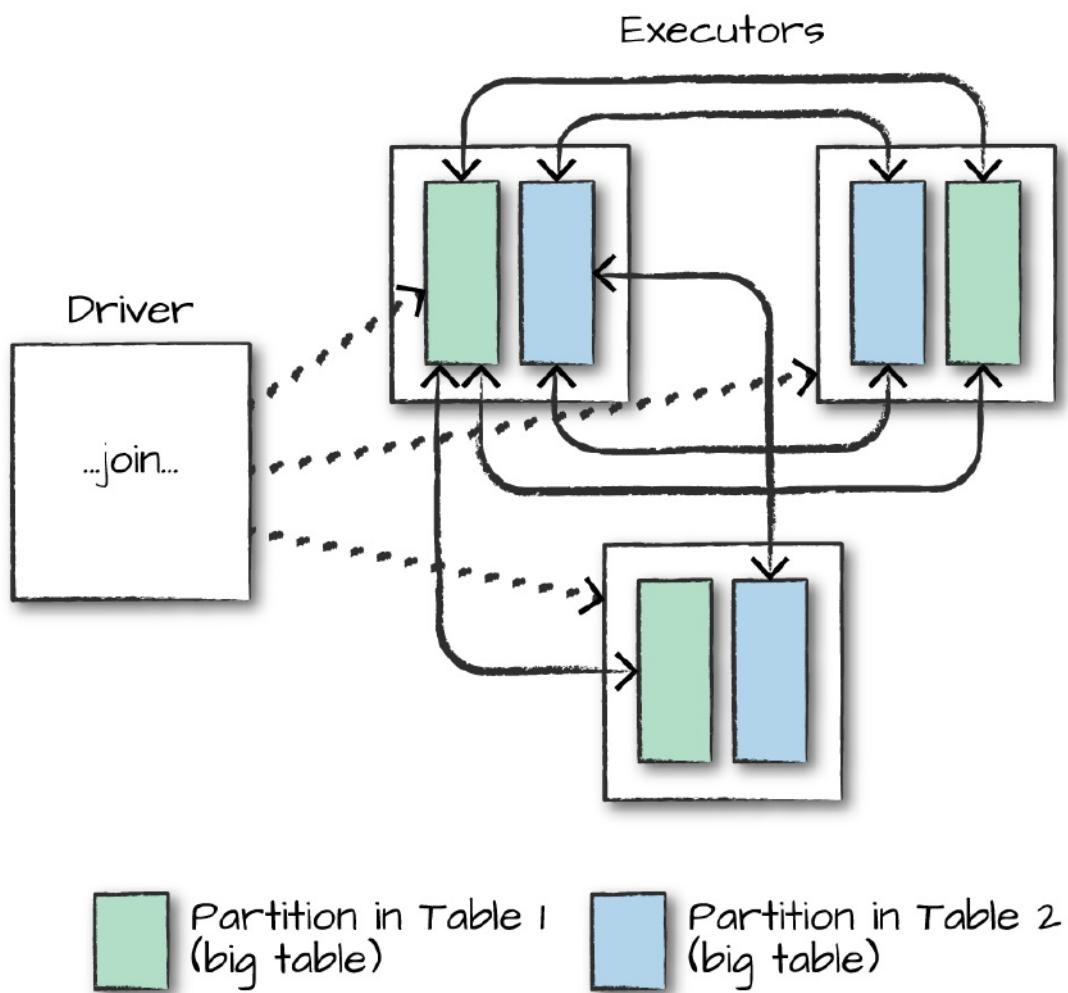
O Spark realiza comunicações dentro do *cluster* de duas formas diferentes durante operações de *joins*:

- *shuffle join (all-to-all communication)*: cada nó se comunica entre si, compartilhando dados conforme a presença das chaves de cruzamento nos nós. Estes são *joins* computacionalmente custosos pois o tráfego de rede pode ser intenso, especialmente se as bases de dados não estão bem particionadas.
- *broadcast join*: o DataFrame com menor quantidade de dados (uma das bases do processo de *join*) é replicado entre todos os nós *worker* do cluster. Apesar de parecer custoso em termos de memória, a comunicação entre nós é feita apenas no início (processo de replicação) e não durante todo o processo, permitindo que os nós realizem o trabalho sem riscos de congestionamento de tráfego de rede.

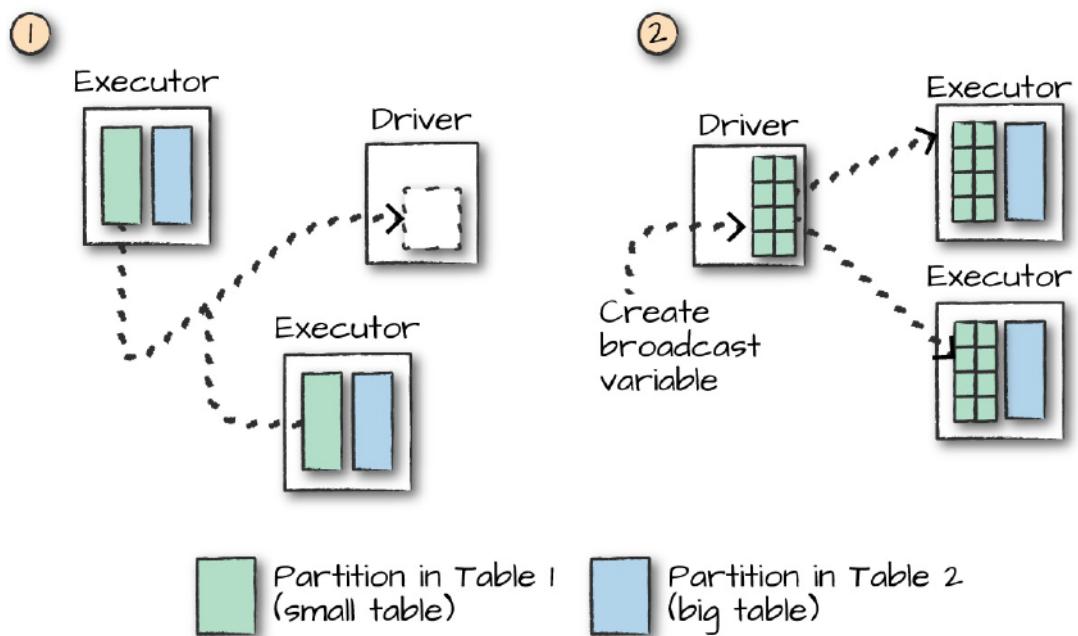
A principal maneira de entender detalhes sobre estas abordagens é referenciar cenários de *joins* com “**grandes tabelas**” e “**pequenas tabelas**” em termos de quantidade de dados e volume.

Quando são realizados *joins* envolvendo duas “grandes tabelas”, o Spark realiza o processo de *shuffle join* ilustrado pela figura 8.13. Na figura, ambos os DataFrames contém grandes quantidades de dados e, por ser um *shuffle join*, todos os nós *worker* se comunicarão entre si durante todo o processo.

Em um segundo cenário, ao performar *joins* que envolvem uma “grande tabela” com uma “pequena tabela”, normalmente uma maior eficiência será obtida com *broadcast join* o qual é ilustrado pela figura 8.14. Um risco associado ao *broadcast join*, por mais que pareça ser mais eficiente do que o *shuffle join* é, sem dúvida, o poder computacional da CPU. Aplicar *broadcast join* em grandes volumes de dados pode fatalmente sobrecarregar as máquinas e prejudicar todo o cluster.



**Figure 8.13:** Ilustração do processo de *shuffle join* em operações envolvendo dois grandes conjuntos de dados



**Figure 8.14:** Ilustração do processo de *broadcast join* em operações envolvendo um grande conjunto de dados e um conjunto de dados de menor porte em termos de volume.

---

# 9

## Fontes de Dados

De forma geral, existem uma variedade de fontes de dados a serem trabalhados no Spark. Ao todo, é possível dizer que o Spark possui 6 fontes de dados “core”, sendo elas:

- CSV
- JSON
- Parquet
- ORC
- Conectores JDBC/ODBC
- Plain text

Em um outro espectro, existem outra infinidade de fontes de dados externas e que também podem ser trabalhadas dentro do Spark. Exemplos:

- Cassandra
- HBase
- MongoDB

- AWS Redshift
- XML
- e outras...

## 9.1. Fundamentos de Leitura de Fontes de Dados

De modo a consolidar um pouco do trabalho já realizado até o momento, a estrutura básica para leitura de fontes de dados em Spark pode ser dada através do trecho de código abaixo:

```
DataFrameReader.format(...)

$ ./bin/spark-submit \
--master local \
.examples/src/main/python/pi.py 10

teste
```

---

## References

- [1] Apache Software Foundation. *Apache Hadoop YARN*. 2022. URL: <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html> (visited on 02/06/2022).
- [2] Apache Software Foundation. *Apache Mesos*. 2022. URL: <https://mesos.apache.org/> (visited on 02/06/2022).