escola britânica de artes criativas & tecnologia

Módulo | Big Data I - Processamento

Caderno de **Aula**
Professor André Perez

Tópicos

1. Introdução;
2. Apache Spark;
3. Data Wrangling com Spark.

Aulas

1. Introdução

1.1. Big data

Big data é um termo que geralmente representa um conjunto de dados muito grande, complexo e de difícil processamento. Apesar do apelo comercial, o termo pouco contribui para a definição de um problema com o volume de dados pois:

- tamanho é relativo, um problema de processamento em um computador pode não ser para outro que contenha mais recursos (RAM, CPU, etc.);
- tamanho não é o único desafio moderno no processamento de dados: a variedade do tipo dos dados e a velocidade com que são produzidos somam-se a essa complexidade;
- etc.

1.2. 3vs: Volume, variedade e velocidade

A criação da *internet* (1990s) e a sua democratização através da adoção em massa de computadores pessoais (2000s), *smartphones* (2010s) e dispositivos da *internet* das coisas (2010s), trouxe diversos novos desafios para o ecossistema de dados em três frentes: **volume**, **variedade** e **velocidade**.

- **Volume**: os recursos de um sistema computacional não são suficientes para processar um determinado volume de dados em uma determinada janela de de tempo / tamanho dos dados representa frações da memória do computador (décimos, etc.). Via de regra, arquivos com mais de 100 MB de tamanho são problemáticos para serem processados em computadores tradicionais.

  > **Exemplo**: um arquivo de texto (txt) de *log* de acesso de usuários a um *website* facilmente atinge 100 MB de tamanho em poucos dias. Com esse arquivo é possível responder perguntas de negócio como: Qual é o período do dia/semana/mês/ano com mais acessos? Qual o tempo médio da etapa de *login*? Qual a taxa de erro de acesso por dia/semana/mês?

- **Variedade**: as fontes de dados modernas armazenam e disponibilizam dados em diversos formatos. Somaram-se aos tradicionais bancos de dados relacionais (SQL) diferentes formatos de arquivos de arquivos (csv, json, txt, html, pdf, jpeg, png, etc.), bases de dados não relacionais (NoSQL ou dados semi/não estruturados), APIs (json), etc.

  > **Exemplo**: os sites dos tribunais de justiça dos estados publicam diariamente o andamento dos processos judiciais que tramitam na segunda instância em arquivos do tipo `pdf`. Como fazer para extrair e armazenar estes arquivos diariamente? Como extrair o número e o *status* do processo do documento?

- **Velocidade**: processamento de dados em lote (*batch*) já não atende mais as necessidades do negócio. Dispositivos permanecem conectados as redes de computadores (*internet*, *internet* móvel, etc.) o tempo todo, logo, continuamente produzindo dados.

  > **Exemplo**: um *e-commerce* registra os *clicks* de um usuário enquanto este navega pelo seu *website*. Com este dados e com o histórico do usuário, seria possível disponibilizar um cupom de desconto para que o usuário não deixe o *website* sem finalizar uma compra? Qual o melhor momento para enviar o cupom?

1.2. Computação distribuída e paralela

A estratégia para lidar com o aumento da demanda por recursos para processamento de dados sempre foi a de melhoria do *hardware* de **um mesmo computador**: mais memória, mais velocidade de processamento, etc. Contudo, após os anos 2000s, a demanda cresceu em um ritmo muito mais acelerado se comparado a capacidade de melhoria de *hardware*. E dessa necessidade nasceu uma nova arquitetura de computadores e um novo paradigma de computação: *clusters* de computadores (**múltiplos computadores**) e computação distribuída e paralela, respectivamente.

- Arquitetura

Um *cluster* é um conjunto de computadores (mesmas configurações, idealmente mesmo *hardware*, etc.) conectados em uma rede privada. Um gerenciador de *cluster* (*cluster manager*) é uma aplicação que orquestra as atividades de armazenamento e processamento de dados distribuído e paralelo, abstraindo a complexidade para usuários e aplicações. Os gerenciadores de *cluster* mais utilizados são o Apache Hadoop e o Kubernetes.

> Computadores de um *cluster* são conhecidos como nós.

- Armazenamento

Dados são armazenados em arquivos (`csv` , `txt` , `parquet` , etc.) e são "quebrados" em blocos (128 MB geralmente), distribuídos e replicados (três vezes geralmente) nos nós. O gerenciador de *cluster* mantém um mapa da distribuição dos blocos.

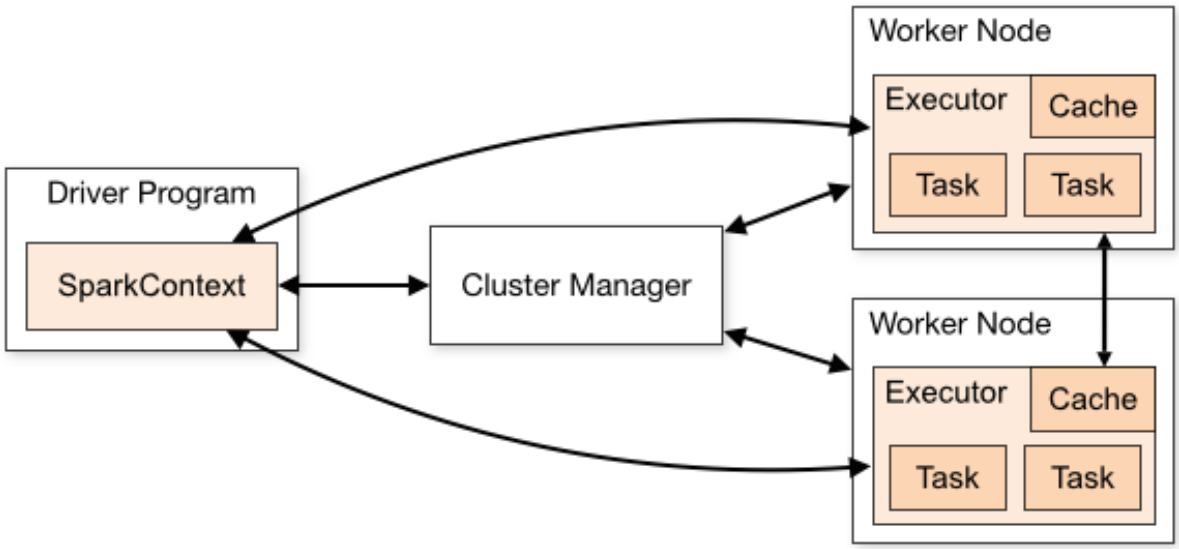
- Processamento

Existem algumas maneiras de se processador dados distribuídos. Uma das maneiras mais eficiente é enviar a operação de processamento (agregações como soma, por exemplo) para o nó em que o dado está armazenado, realizar o processamento localizado e coletar apenas os resultados.

> **Nota**: Operações de junção (joins) constumam ser caras (em termos de tempo de processamento e consumo de memória) pois blocos inteiros de dados devem trafegar pela rede de um nó para o outro.

2. Apache Spark

Spark é um motor de processamento de dados para engenharia, análise e ciência de dados otimizado para *clusters* de computadores. Permite que operações comuns no processamento de dados como seleção de colunas, filtros e *joins* sejam feitas de utilizando o paradigma de computação distribuída e paralela através de APIs de alto nível disponível em diversas linguagens.



O Spark divide as operações em **transformações** (`filter` , `select` , `withColumn` , etc.) e **ações** (`read.csv` , `write.csv` , etc.). Operações de **transformação** são encadeadas até que uma operação de **ação** seja executada, fazendo com que todas as operações sejam executadas de uma vez. Essa característica é conhecida como *lazy evaluation*.

> **Nota**: Em geral, a instalação (item 2.1) e configuração (item 2.2) de um *cluster* Spark é feito por especialistas, como uma pessoa engenheira de dados. É comum uma pessoa analista/cientista de dados começar a interagir com um *cluster* Spark a partir do item 2.3.

> **Nota**: O AWS EMR (*elastic map reduce*) fornece *clusters* com gerenciador Apache Hadoop e com o Apache Spark instalado. Preço computado sobre a hora dos nós, máquinas virtuais do AWS EC2.

2.1. Instalação

Spark é uma aplicação desenvolvida na linguagem de programação Scala, que funciona em uma máquina virtual Java (JVM). Por isso, é necessário fazer o *download* da aplicação e instalar o Java em todas as máquinas (nós) do *cluster*.

- Download do Spark, versão 3.0.0.

In [ ]:
```
%%capture

!wget -q https://archive.apache.org/ +
      dist/spark/spark-3.0.0/spark-3.0.0-bin-hadoop2.7.tgz

!tar xf spark-3.0.0-bin-hadoop2.7.tgz
!rm spark-3.0.0-bin-hadoop2.7.tgz
```

- Download e instalação do Java, versão 8.

In [ ]:
```
%%capture
!apt-get remove openjdk*
!apt-get update --fix-missing
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
```

Mesmo sendo uma aplicação Scala, o Spark disponibiliza APIs de integração em diversas linguagens de programação. O pacote Python PySpark é a API para Python. Com ele, é possível interagir com o Spark como se fosse uma aplicação Python nativa. A API é similar ao pacote Pandas e sua documentação pode ser encontrada neste link.

> **Nota**: a versão do PySpark deve ser a mesma que a versão da aplicação Spark.

In [ ]:
```
!pip install -q pyspark==3.0.0
```

2.2. Configuração

Na etapa de configuração, é necessário configurar as máquinas (nós) do *cluster* para que tanto a aplicação do Spark quanto a instalação do Java possam ser encontrados pelo PySpark e, consequentemente, pelo Python. Para isso, basta preencher as variáveis de ambiente `JAVA_HOME` e `SPARK_HOME` com o seus respectivos caminhos de instalação.

In [ ]:
```
import os

os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.0.0-bin-hadoop2.7"
```

Por fim, para conectar o PySpark (e o Python) ao Spark e ao Java, pode-se utilizar o pacote Python FindSpark.

In [ ]:
```
!pip install -q findspark==1.4.2
```

O método `init()` injeta as variáveis de ambiente `JAVA_HOME` e `SPARK_HOME` no ambiente de execução Python, permitindo assim a correta conexão entre o pacote PySpark com aplicação Spark.

In [ ]:
```
import findspark

findspark.init()
```

2.3. Conexão

Com o *cluster* devidamente configurado, vamos criar uma aplicação Spark. O objeto `SparkSession` do pacote PySpark (e seu atributo `builder`) auxiliam na criação da aplicação:

- `master`: endereço (local ou remoto) do *cluster*;
- `appName`: nome da aplicação;
- `getOrCreate`: método que de fato cria os recursos e instância a aplicação.

In [ ]:
```
from pyspark.sql import SparkSession

spark = SparkSession.\
    builder.\
    master("local[*]").\
    appName("pyspark-notebook").\
    getOrCreate()
```

Com o objeto `SparkSession` devidamente instanciado, podemos começar a interagir com os dados utilizando os recursos do *cluster* através de uma estrutura de dados que já conhecemos: `DataFrames`:

In [ ]:
```
!wget -q "https://raw.githubusercontent.com/"+
    "cluster-apps-on-docker/spark-standalone-cluster-on-docker/"+
    "master/build/workspace/data/uk-macroeconomic-data.csv"
-O "uk-macroeconomic-data.csv"
```

In [ ]:
```
data = spark.read.csv(
    path="uk-macroeconomic-data.csv",
    sep=",",
    header=True
)
```

In [ ]:
```
data.show()
```

In [ ]:
```
data.printSchema()
```

3. Data Wrangling

Vamos utilizar a API Python do Spark, o pacote PySpark, para limpar os dados da aula 2.

> **Nota**: Atente-se sempre a natureza distribuída das operações.

3.1. Exploração

In [ ]:
```
data.count()
```

In [ ]:
```
data.columns
```

In [ ]:
```
len(data.columns)
```

3.2. Limpeza

O método `select` seleciona colunas do `DataFrame`. Já o método `withColumnRenamed` renomeia colunas.

In [ ]:
```
data = data.select([
    "Description",
    "Population (GB+NI)",
    "Unemployment rate"
])
```

In [ ]:
```
data = data.\
  withColumnRenamed(
    "Description", 'year'
).\
  withColumnRenamed(
    "Population (GB+NI)", "population"
).\
  withColumnRenamed(
    "Unemployment rate", "unemployment_rate"
)
```

In [ ]:
```
data.show(n=10)
```

O método `filter` seleciona linhas do `DataFrame` baseado no conteúdo de uma coluna.

In [ ]:
```
data_description = data.filter(data['year'] == 'Units')
```

In [ ]:
```
data_description.show(n=10)
```

3.3. Junção

In [ ]:
```
(data.count(), len(data.columns))
```

In [ ]:
```
(data_description.count(), len(data_description.columns))
```

O método `join` faz a junção distribuída de dois `DataFrames`. Já o método `broadcast` "marca" um `DataFrame` como "pequeno" e força o Spark a trafega-lo pela rede.

In [ ]:
```
from pyspark.sql.functions import broadcast
```

In [ ]:
```
data = data.join(
    other=broadcast(data_description),
    on=['year'],
    how='left_anti'
)
```

In [ ]:
```
data.show(n=10)
```

O método `dropna` remove todas as linhas que apresentarem ao menos um valor nulo.

In [ ]:
```
data = data.dropna()
```

In [ ]:
```
data.show(n=10)
```

O método `withColumn` ajuda a criar novas colunas.

In [ ]:
```
data = data.withColumn(
    'century',
    1 + (data['year']/100).cast('int')
)
```

In [ ]:
```
data.\
    select(['century', 'year']).\
    groupBy('century').\
    agg({'year': 'count'}).show()\
```

O método `collect` é uma ação que coleta os resultados dos nós e retorna para o Python.

In [ ]:
```
timing = data.\
    select(['century', 'year']).\
    groupBy('century').\
    agg({'year': 'count'}).\
    collect()
```

In [ ]:
```
timing
```

In [ ]:
```
timing[0].asDict()
```

3.4. Escrita

O método `write.csv` persiste o `DataFrame` em formato `csv`. Já o método `repartition` controla o número de partições da escrita.

In [ ]:

```
data.\
    repartition('century').\
    write.\
    csv(
        path="uk-macroeconomic-data-clean",
        sep=",",
        header=True,
        mode="overwrite"
    )
```