

Resumo de Atividades da Semana

Exercício 01: Modifique esse código que conta palavras modificando o modo de saída para Modo Completo (complete)

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql import functions as f
sc = SparkContext()
sc.setLogLevel("ERROR")

spark = SparkSession.builder.appName("ContarPalavrasEstruturadas").getOrCreate()

linhas = spark.readStream\
    .format("socket")\
    .option("host", "localhost")\
    .option("port", 9999)\
    .load()

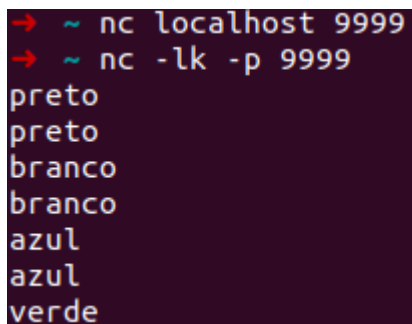
palavras = linhas.select(f.explode(f.split(linhas.value, " ")).alias("palavra"))

contagem = palavras.groupBy('palavra').count()

consulta = contagem\
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

consulta.awaitTermination()
```

- Código do material



```
→ ~ nc localhost 9999
→ ~ nc -lk -p 9999
preto
preto
branco
branco
azul
azul
verde
```

- Simulação usando o nc, com a porta 9999

Batch: 3		Batch: 4	
palavra	count	palavra	count
preto	2	preto	2
branco	2	verde	1
azul	1	branco	2
		azul	2

- Resultado do streaming (terminal do jupyter-notebook)

Exercício 02: Repetir o exercício anterior utilizando o modo Anexo (append)

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql import functions as f
sc = SparkContext()
sc.setLogLevel("ERROR")

spark = SparkSession.builder.appName("exercicio_janela").getOrCreate()

linhas = spark.readStream\
    .format("socket")\
    .option("host", "localhost")\
    .option("port", 9999)\
    .option("includeTimestamp", "true")\
    .load()

palavras = linhas.select(
    f.explode(f.split(linhas.value, " ")).alias("palavra"),
    linhas.timestamp
)

consulta = palavras\
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

consulta.awaitTermination()
```

- Algumas mudanças no código são necessárias, pois o append não permite agregação de dados.

Batch: 1		Batch: 2		Batch: 3	
palavra	timestamp	palavra	timestamp	palavra	timestamp
preto	2021-07-20 16:50:...	preto	2021-07-20 16:50:...	branco	2021-07-20 16:50:...

- No modo append, apenas as palavras novas adicionadas são mostradas por lote.

- **Exercício 03: Repetir o exercício anterior utilizando o modo update**

```
consulta = contagem\  
    .writeStream \  
    .outputMode("update") \  
    .format("console") \  
    .start()  
  
consulta.awaitTermination()
```

- Modificações no código

Batch: 1	Batch: 2	Batch: 3
-----	-----	-----
+-----+-----+	+-----+-----+	+-----+-----+
palavra count	palavra count	palavra count
+-----+-----+	+-----+-----+	+-----+-----+
preto 2	verde 2	preto 3
branco 1	branco 2	
+-----+-----+	+-----+-----+	+-----+-----+

- Execução do código, apenas atualiza os valores atuais

Exercício 04:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql import functions as f
sc = SparkContext()
sc.setLogLevel("ERROR")
```

```
spark = SparkSession.builder.appName("exercicio_janela").getOrCreate()
```

```
linhas = spark.readStream\
    .format("socket")\
    .option("host", "localhost")\
    .option("port", 9999)\
    .option("includeTimestamp", "true")\
    .load()
```

```
palavras = linhas.select(
    f.explode(f.split(linhas.value, " ")).alias("palavra"),
    linhas.timestamp
)
```

```
contagem = palavras\
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        f.window(palavras.timestamp, "10 minutes", "5 minutes"),
        palavras.palavra
    ).count().orderBy('window')
```

```
consulta = contagem\
    .writeStream \
    .option("truncate", "false") \
    .outputMode("complete") \
    .format("console") \
    .start()
```

```
consulta.awaitTermination()
```

- Mudanças do código, desta vez utilizando a função window para fazer o particionamento dos dados por tempo.

```
-----
Batch: 1
-----
```

```
+-----+-----+
|window|palavra|count|
+-----+-----+
|{2021-07-20 15:55:00, 2021-07-20 16:05:00}|preto|1|
|{2021-07-20 16:00:00, 2021-07-20 16:10:00}|preto|1|
+-----+-----+
```

```
-----
Batch: 2
-----
```

window	palavra	count
{2021-07-20 15:55:00, 2021-07-20 16:05:00}	preto	1
{2021-07-20 15:55:00, 2021-07-20 16:05:00}	vermelho	1
{2021-07-20 15:55:00, 2021-07-20 16:05:00}	verde	1
{2021-07-20 15:55:00, 2021-07-20 16:05:00}	branco	1
{2021-07-20 16:00:00, 2021-07-20 16:10:00}	vermelho	1
{2021-07-20 16:00:00, 2021-07-20 16:10:00}	branco	1
{2021-07-20 16:00:00, 2021-07-20 16:10:00}	verde	1
{2021-07-20 16:00:00, 2021-07-20 16:10:00}	preto	1

```
-----
```

- Resultado do exercício, desta vez contém uma nova coluna com os timestamps.

DESAFIO XPTO: INGESTÃO DOS DADOS DO TWITTER

Exercício 05: Fazendo a ingestão de dados da api do Twitter, usando Tweepy:

Análise do código:

```
import tweepy
from tweepy import Stream
from tweepy.streaming import StreamListener
from smart_open import open
import json
from datetime import datetime

twitter_keys = {
    'consumer_key': 'xsLmIoXBMS5AyP9Apgj\lVD
    'consumer_secret': 'efjjVQrG0ZfMQwNSZPX
    'access_token_key': '841044884048007170
    'access_token_secret': '7VVCyHDZ79GfU1K
}
```

- Importamos todas as bibliotecas necessárias para a ingestão de dados, e configuramos as credenciais de acesso para a API do Twitter.

```

if __name__ == '__main__':
    search_words = ['Bolsonaro', 'Presidente do Brasil']
    file = 's3://xptoraw/stream/twitter_'

    auth = tweepy.OAuthHandler(twitter_keys['consumer_key'], twitter_keys['consumer_secret'])
    auth.set_access_token(twitter_keys['access_token_key'], twitter_keys['access_token_secret'])
    api = tweepy.API(auth)
    twitter_stream = Stream(auth = api.auth, listener=TweetsListener(file), lang='pt-br')
    twitter_stream.filter(track = search_words)

```

- No módulo main definimos os parâmetros de pesquisa do twitter, fizemos a autenticação utilizando as chaves com a lib do tweepy
- Após isso, executamos a classe Stream, passando a autenticação e a classe Tweets Listener, como parâmetro passamos o arquivo de saída, e a linguagem buscadas pelos tweets.
- Finalmente, fizemos um filtro da Stream criada para procurar as palavras chaves definidas inicialmente.

```

class TweetsListener(StreamListener):

    def __init__(self, file):
        self.file = file
        self.array = []

    def on_data(self, data):
        try:
            tweet = json.loads(data)
            formattedDate = datetime.strftime(datetime.strptime(tweet['created_at'], '%a %b %d %H:%M:%S %Z %Y'), '%Y-%m-%d %H:%M:%S')
            strTweet = json.dumps({"id": tweet['id'], "text": tweet['text'], "created_at": formattedDate},
                                  indent=4, sort_keys=True)
            tweet_py = json.loads(strTweet)
            self.array.append(tweet_py)
            print(tweet_py)
            if len(self.array) > 100:

                formatted_date_first_tweet = datetime.strftime(datetime.strptime(self.array[0]['created_at'], '%Y-%m-%d %H:%M:%S'), '%Y-%m-%d %H:%M:%S')
                with open(self.file + formatted_date_first_tweet + ".json", 'a+') as f:
                    f.write(json.dumps(self.array, indent=4, sort_keys=True))
                    self.array = []

            return True

        except BaseException as e:
            print("Erro:", e)
            return False

    def on_error(self, status):
        print(status)

```

- Na classe Tweets Listener é executado todo o processo de retirada dos dados do twitter e todo o processo necessário para enviar para o Bucket RAW do S3:
- Inicialmente definimos o construtor da classe, onde inicializamos o arquivo que foi mandado como parâmetro, e um array auxiliar adicional.
- Na função on_data, o data que é recebido como parâmetro é todos os dados em formato RAW dos tweets retirados.
- Então inicialmente convertemos para um array python, selecionamos os dados que iremos precisar e convertemos novamente para um array python, para depois juntar neste array auxiliar.
- Também formatamos a data para um formato que fique fácil de utilizar para a segunda parte do processamento (data irá para um formato timestamp)
- Para verificar que temos 100 tweets por arquivo, é feito uma verificação no tamanho deste array, se tiver mais que 100 tweets é executado o processo a seguir:
- Formatamos a data do primeiro tweet para anexar no nome do arquivo, abrimos o arquivo com todo o seu diretório, e escrevemos no arquivo o array completo contendo os 100 tweets. Após isso zeramos todo o dado do array, para iniciar o processo novamente.

Execução deste código via AWS EMR:

Cluster: xpto_ingestão_streaming **Aguardando** Cluster ready after last step completed.

Resumo Histórico do aplicativo Monitoramento Hardware Configurações Eventos Etapas Ações de bootstrap

Resumo

ID: j-1EDWZLWXF5Q0Z

Data de criação: 2021-07-25 10:18 (UTC-3)

Tempo decorrido: 1 dia

Encerramento automático: Cluster waits

Proteção contra encerramento: Desativado [Alterar](#)

Tags: -- [Visualizar todas/Editar](#)


DNS público principal: ec2-35-171-18-164.compute-1.amazonaws.com [Connect to the Master Node Using SSH](#)

Detalhes da configuração

Rótulo da versão: emr-6.1.0

Distribuição do Hadoop: Amazon


Aplicativos: Spark 3.0.0, Zeppelin 0.9.0


URI do log: s3://aws-logs-575556700570-us-east-1/elasticmapreduce/ 

Visualização consistente do EMRFS: Desativado

ID personalizado de AMI: --


Application user interfaces

Serviço de histórico:  [Spark history server, YARN timeline server](#)

Conexões:  Not Enabled [Habilitar conexão da web](#)

Rede e hardware

Zona de disponibilidade: us-east-1c

ID da sub-rede: [subnet-005d1421](#) 

Principal: Running 1 m4.large

Serviços: --

Tarefa: --

Cluster scaling: Not enabled

- Primeiro é necessário criar um cluster EMR

SSH

TCP

22

Anywhere-I...

Q

0.0.0.0/0 X

- Após isso , é necessário configurar o grupo de segurança do cluster para acesso ao master node (instalação de extras) via SSH.

```

EEEEEEEEEEEEEEEEEEEE MMMMMMM      MMMMMMM RRRRRRRRRRRRRR
E::::::::::::::::::::E M::::::::M      M::::::::M R:::::::::R
EE::::::::EEEEEEEE::::E M::::::::M      M::::::::M R::::::::RRRRRR::::R
  E::::E      EEEEE M::::::::M      M::::::::M RR::::R      R::::R
  E::::E      M::::M:M:M      M::M::::M      R:::R      R::::R
  E::::EEEEEEEEEE M::::M M::M M::M M::::M      R::RRRRRR::::R
  E:::::::::::::E M::::M M::M:M:M      M::::M      R:::::::::RR
  E::::EEEEEEEEEE M::::M M::::M      M::::M      R::RRRRRR::::R
  E::::E      M::::M      M::M      M::::M      R:::R      R::::R
  E::::E      EEEEE M::::M      MMM      M::::M      R:::R      R::::R
EE::::::::EEEEEEEE::::E M::::M      M::::M      R:::R      R::::R
E:::::::::::::E M::::M      M::::M      RR::::R      R::::R
EEEEEEEEEEEEEEEEEEEE MMMMMMM      MMMMMMM RRRRRRR      RRRRRR

[hadoop@ip-172-31-84-130 ~]$
[hadoop@ip-172-31-84-130 ~]$ pip install tweepy smart_open[s3]

```




- Conectamos ao cluster EMR (Master Node) via SSH e instalamos os pacotes necessários para execução do script.

```

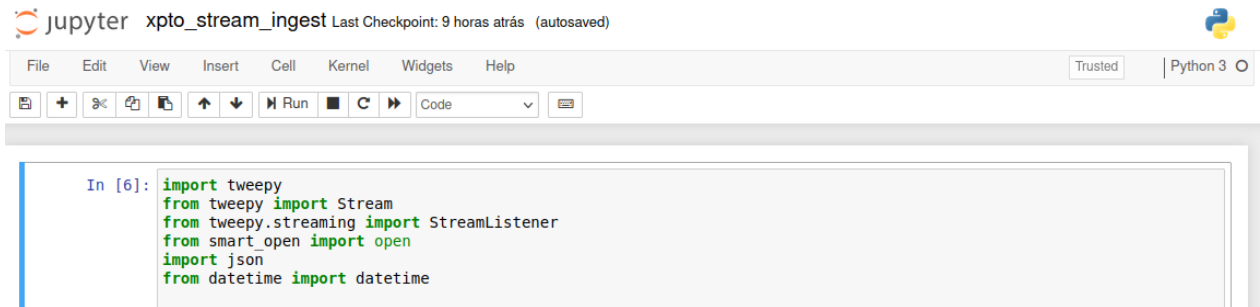
[hadoop@ip-172-31-84-130 ~]$ spark-submit --deploy-mode cluster s3://xpto-scripts/xpto_raw_streaming.py

```

- Após a instalação de todos os pacotes, executamos o comando spark-submit passando o URI do script Python para ingestão dos dados.
- Após executar o script por alguns minutos, temos o resultado (dentro do bucket de destino do S3), a ingestão de tweets pronta no formato JSON.

-  [twitter_20210725_233206.json](#)
-  [twitter_20210725_233250.json](#)
-  [twitter_20210725_233331.json](#)

Usando Glue:



- Neste caso, utilizamos o glue notebook, e executamos o mesmo script que foi executado no EMR, obtendo o mesmo resultado.

Exercício 06: Processamento dos dados streaming

Análise do Código:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Spark Twitter Transformation"
    )
    parser.add_argument("--src", required=True)
    parser.add_argument("--dest", required=True)
    args = parser.parse_args()

    twitter_transform(args.src, args.dest)
```

- Criação da função main, onde apenas é criado os argumentos necessários para execução do script, nesse caso a entrada de dados e a saída de dados.

```
def twitter_transform(src, dest):
    with SparkSession.builder.appName("Twitter Transformation").getOrCreate() as spark:
        df = import_json(spark, src)
        df2 = preprocessing(df)
        df3 = create_columns_data(df2)
        export_parquet(df3, dest)
```

- Primeiramente, é iniciada uma sessão do Spark utilizando o comando SparkSession. Após isso, foi dividido cada etapa do processamento em 4 funções, que são elas:

- Import Json: Essa função faz a importação do Json de origem.
- Pré-processamento: Esta etapa é necessária para limpar os caracteres inválidos existentes nos textos dos tweets.
- Criação de colunas de dados: Essa função é a principal responsável pela criação de 3 colunas a mais, ajustando o dataframe.
- Export Parquet: Função para exportar o arquivo em formato Parquet para uma saída.

```
def import_json(spark, src):  
    tweets = t.StructType([  
        t.StructField("id", t.LongType(), False),  
        t.StructField("text", t.StringType(), False),  
        t.StructField("created_at", t.TimestampType(), False)  
    ])  
  
    df = spark \  
        .readStream\  
        .schema(tweets) \  
        .option("multiline", 'true')\  
        .json(src)  
  
    return df
```

- Nesta função, primeiramente é definido o schema (a tipagem dos dados) que vamos receber. Neste caso temos um id como um long int, o texto em si no formato string e a data de publicação no formato TimeStamp.
- Após isso, executamos a função readStream, passando o esquema definido anteriormente e o caminho de dados de entrada(em formato JSON). Neste caso, é ativada a opção multilinha, prática comum para ler arquivos json com mais de uma variável.

Positivos, enquanto símbolos tristes como (D: , :() são considerados como Negativos . Os textos que não aparecem no símbolo são considerados Neutros.

- Finalmente, criamos uma coluna contendo o símbolo que foi identificado, conforme os padrões dos símbolos comentados anteriormente.

```
def export_parquet(df, dest):
    query = df\
        .writeStream\
        .outputMode("append").format("parquet")\
        .partitionBy('created_date')\
        .option("path", dest)\
        .option("checkpointLocation", dest + "/check")\
        .trigger(processingTime='30 seconds')\
        .start()
    query.exception()
    query.awaitTermination()
```

- E por último, ocorre a exportação desses arquivos. Neste caso iremos exportar no formato Parquet, no modo Acrescentar(append), e particionado cada um deles pelo dia de criação (mostrado no created_date). Também é passado o destino, que foi configurado como um argumento no início do processo. Por fim passamos um gatilho de tempo de execução da stream, neste caso de 30 segundos.

Testes no EMR:

```
[hadoop@ip-172-31-84-130 ~]$ spark-submit --deploy-mode cluster s3://xpto-scripts/xpto_processing_streaming.py --src s3://xptoraw/stream/ --dest s3://xpto-refined/stream/
```

- Neste caso podemos utilizar o mesmo cluster que foi executado o script de ingestão de dados. Para isso, utilizamos o comando spark-submit passando o script(no bucket S3) e os parâmetros de entradas e saídas de dados.

 [_spark_metadata/](#)

 [check/](#)

 [created_date=2021-07-25/](#)

 [created_date=2021-07-26/](#)

- Script finalizado, com a criação dos arquivos do spark, o checkpoint, e os arquivos particionados conforme a data do tweet.

Glue:

Trabalho: xpto_stream_processing Ação Salvar Executar trabalho Gerar diagrama ⓘ

```

1 import argparse
2 from pyspark.sql import SparkSession
3 from pyspark.sql import functions as f
4 from pyspark.sql import types as t
5 from awsglue.utils import getResolvedOptions
6 from awsglue.context import GlueContext
7 from awsglue.dynamicframe import DynamicFrame
8 from awsglue.job import Job
9
10
11 def create_columns_data(df):
12
13     df2 = df\
14         .withColumn("created_date", f.to_date("created_at")).repartition("created_date")\
15         .withColumn("sentimento",
16             f.when((df.text.contains(":D") | (df.text.contains(":") | (df.text.contains(":") | (df.text.contains(":P"))), "Positivo")
17             .when((df.text.contains("D:") | (df.text.contains(":(") | (df.text.contains(":["), "Negativo")
18             .otherwise("Neutro"))\
19         .withColumn("sinbolo",
20             f.when(df.text.contains(":D"), ":D")
21             .when(df.text.contains(":("), ":(")
22             .when(df.text.contains(":["), ":[")
23             .when(df.text.contains(":P"), ":P")
24             .when(df.text.contains("D:"), "D:")
25             .when(df.text.contains(":("), ":(")
26             .when(df.text.contains(":["), ":[")
27             .otherwise(":)")
28     return df2
29
30 def preprocessing(df):
31     df = df.withColumn('text', f.regexp_replace('text', r'http\S+', ''))
32     df = df.withColumn('text', f.regexp_replace('text', '@\w+', ''))
33     df = df.withColumn('text', f.regexp_replace('text', '#', ''))
34     df = df.withColumn('text', f.regexp_replace('text', 'RT', ''))
35     df = df.withColumn('text', f.regexp_replace('text', ':', ''))
36     df = df.na.replace('', None)
37     df = df.na.drop()
38     return df
39


```


- No glue podemos aproveitar o código utilizado no EMR, aproveitar o crawler utilizado na Semana 09 de processamento glue, e apenas rodar um trabalho com o script criado, que dará o mesmo resultado.

Exercício 07: Configuração do AWS Athena para realizar queries no bucket REF.

Choose where your data is located


Athena queries data where it is. Data is not loaded or moved. [Learn more](#)


☒ Query data in Amazon S3
Choose an external data catalog.


☐ Query a data source
Configure a connector for common data sources.


Choose a metadata catalog

The catalog contains the schema for the source data such as column names, data types and table names. [Learn more](#)

☒  AWS Glue Data Catalog

☐  Apache Hive metastore

Cancel Next

- Configuração da localização dos dados. No nosso caso utilizaremos o bucket S3 REF como fonte de dados para o AWS Athena.

Column Name
Column name must be single words that start with a letter or a digit.

Column type
Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

Column Name
Column name must be single words that start with a letter or a digit.

Column type
Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

Column Name
Column name must be single words that start with a letter or a digit.

Column type
Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

Column Name
Column name must be single words that start with a letter or a digit.

Column type
Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

Column Name
Column name must be single words that start with a letter or a digit.

Column type
Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

- Configuração de tipagem de todas as colunas processadas.

Configure Partitions (Optional)

Partitions are a way to group specific information together. Partitions are virtual columns. In case of partitioned tables, subdirectories are created based on the values of the partition columns. If the table is partitioned on multiple columns, then nested subdirectories are created based on the order of partition columns in the table definition.

Column Name

Column name must be single words that start with a letter or a digit.

Column type

Type for this column. Certain advanced types (namely, structs) are not exposed in this interface.

- Configuração das partições dos arquivos, no nosso caso estão particionados pela data de criação do tweet.

```
CREATE EXTERNAL TABLE IF NOT EXISTS xpto.refined_tweets (  
  `id` bigint,  
  `text` string,  
  `created_at` timestamp,  
  `sentimento` string,  
  `simbolo` string  
) PARTITIONED BY (  
  created_date date  
)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'  
WITH SERDEPROPERTIES (  
  'serialization.format' = '1'  
) LOCATION 's3://xpto-refined/'  
TBLPROPERTIES ('has_encrypted_data'='false');
```

- Código SQL Gerado pela criação da tabela, pronto para realizar queries com os dados já processados.