

Semana 9

Objetivo: Aprofundando os conhecimentos sobre Framework Apache Spark, foco em Dataframe e SQL. Desafio XPTO de Processamento do Histórico carregado via Talend para o S3.

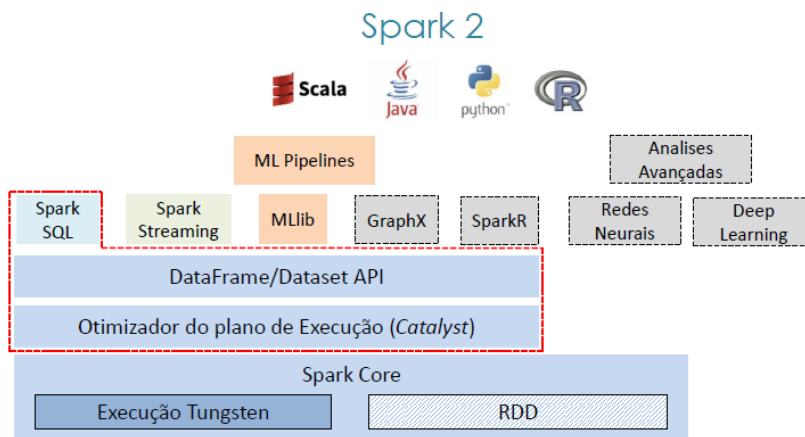
Conteúdo:

- Spark SQL e DataFrame
- Submetendo Aplicações Spark
- Desafio XPTO: Processamento do Histórico

Desafio: Realizar a leitura do material completo e executar os 5 exercícios.

Spark SQL e DataFrame

Anteriormente vimos que RDD é uma API que facilitou bastante a programação, **diminuindo consideravelmente o esforço** do processamento dos dados se comparado com o MapReduce do Hadoop. Porém as versões iniciais dificultavam o uso do Spark pelos usuários finais, uma vez que era necessário compreender o paradigma de programação funcional. A solução para esse problema: **Spark SQL**. Abaixo temos uma representação que mostra os RDDs como parte do Spark-core.



Fonte de dados: Parquet, Hadoop, Cassandra, JSON, CSV, JDBC, entre outros

Empresas lidam com **dados estruturados o tempo todo**, normalmente em grande quantidade. Mesmo que existam diversas formas de lidar com dados não estruturados, **muitas aplicações necessitam que os dados estejam estruturados** para criar algum modelo de aprendizagem ou até mesmo visualizar os dados em ferramentas de BI.

Os fornecedores de sistemas de gerenciamento de banco de dados relacionais se uniram e criaram um **Linguagem de Consulta Estruturada (SQL)**. E nas últimas décadas, SQL se tornou uma das principais linguagens de consulta em dados. As aplicações de grande escala na Internet, como redes sociais, produzem dados além do que as ferramentas tradicionais de processamento de dados conseguem consumir. Portanto, selecionar/buscar o dado corretamente se tornou **essencialmente importante**.

Spark SQL é uma biblioteca que apresenta **uma interface para a linguagem de consulta SQL** e uma **API para DataFrame**. A API para DataFrame suporta as linguagens de programação Scala, Java, **Python** e R. Podemos utilizar diversas fontes de dados desde que os dados se encaixam no modelo de linhas e colunas e **a estrutura dos dados seja conhecida**. O Spark irá utilizá-los como se fosse uma única fonte de dados.

Muitos dos paradigmas de processamento de dados têm adotado o conceito de espelhamento da estrutura dos dados subjacentes **para facilitar o processamento dos dados**, uma vez que estão estruturados. Por exemplo, se tiver um conjunto de dados separados por ponto e vírgula com um número fixo de valores em cada linha e com um tipo de dado específico para cada valor em todas as linhas, temos **um arquivo de dados estruturado**.

Em linguagens de programação como R, existe uma abstração utilizada para armazenar tabelas de dados em memória, chamado de **DataFrame**. No Python, **a biblioteca de análise de dados Pandas**, também tem um conceito similar e é chamado de **DataFrame**. Uma vez que os dados

estejam em memória, os programas podem extrair os dados conforme a necessidade. Esse conceito de tabela de dados é estendido para Spark, conhecido como DataFrame, **que foi construído em cima do RDD e tem uma API muito abrangente no Spark SQL para processar os dados.**

A biblioteca Spark SQL tem por objetivo:

- Suportar **processamento relacional** dentro de programas Spark (nativo em RDDs) e em origem externa de dados utilizando uma API amigável.
- Fornecer **alto desempenho** utilizando as técnicas bem estabelecidas (DBMS).
- Suportar **novas origens de dados**, incluindo dados semiestruturados.
- Habilitar extensão com **algoritmos de análises avançadas** como processamento em grafo e aprendizado de máquina.

DataFrame contém dados estruturados e é distribuído. Permite seleção, filtragem e agregação dos dados, entre outras operações. A principal diferença entre RDD e DataFrame é que o DataFrame armazena mais informações sobre a estrutura dos dados, como os tipos dos dados e nomes de colunas. Isso permite que o DataFrame otimize o processamento efetivamente do que as transformações e ações realizadas no RDD. De modo geral, Spark SQL é um **mecanismo SQL distribuído.**

Podemos dizer que o DataFrame e o uso de SQL irá substituir o modelo de programação baseada em RDD?

Definitivamente não! Pois o modelo de programação baseado em RDD é genérico e é o modelo de processamento de dados básico no Spark.

Tipicamente, durante a fase de projeto, os analistas de negócio geralmente realizam diversas análises com os dados utilizando SQL. O mesmo acontece em projetos de Big Data. Por exemplo, **em ecossistemas baseados no Hadoop, é possível utilizar o Hive para realizar essas análises.** Essa estratégia de utilizar SQL no Hadoop permitiu a criação de diversas aplicações, como **Hive** e **Impala**, que fornecem uma interface SQL para acessar o armazenamento dos dados no HDFS.

Apache Hive: É uma tecnologia de armazenamento de dados baseado em MapReduce, e que de fato utiliza MapReduce para processar as consultas. As consultas em Hive necessitam de muitas operações de I/O (Input/Output) antes de completar uma consulta.

Apache Impala: É o mecanismo de **consulta SQL** de roda no Hadoop. Surgiu com uma solução para realizar o processamento em memória **utilizando os metadados** do Hive. Desta forma, **não é necessário** realizar a movimentação dos conjuntos de dados para os sistemas especializados com o objetivo de realizar as análises.

Spark SQL utiliza o SQLContext para realizar todas as operações com os dados. Porém, é possível utilizar o HiveContext que tem funcionalidades mais avançadas, como por exemplo, utilizar as funções definidas por usuários e acessar as tabelas de metadados do Hive. Para utilizar o **HiveContext**, é necessário que exista uma instalação do Hive disponível. Desta forma, é possível que o **Spark SQL e o Hive coexistam.**

A biblioteca Spark SQL, bem como a API do DataFrame fornecem interfaces que podem ser acessadas via **JDBC/ODBC**, possibilitando que ferramentas de inteligência de negócio (BI) tenham uma conexão direta com os nós trabalhadores (workers) de um cluster Spark.

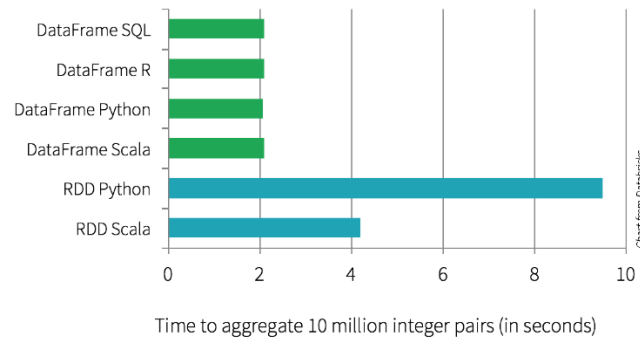
SparkSession é o novo ponto de entrada de aplicações baseadas em SQL, desta forma, é possível combinar o SQLContext e HiveContext suportando a compatibilidade dos contextos existentes.

Resumo de RDD e SparkSQL:

- As transformações e ações do Spark são **convertidos em funções Java** que agem no topo do RDD, que nada mais são do que objetos agindo diretamente nos dados.
- Como o **RDD é objeto Java**, não tem como saber qual dado será processado no tempo de compilação e execução.
- Pois os **metadados não estão** disponíveis para o mecanismo de execução, não sendo possível otimizar as transformações e ações.
- Desta forma, não é possível criar múltiplos caminhos de execução ou plano de consultas antes da execução.
- Em resumo, não existe um plano de consulta otimizado para ser executado pois com RDD não existe um esquema associado aos dados.
- No caso do DataFrame, **a estrutura é bem conhecida** antecipadamente, possibilitando criar **otimizações nas consultas** e realizar cache dos dados antes da execução.
- Em resumo, o DataFrame é um conjunto de dados distribuídos e carregados em colunas nomeadas. Em termos simples, podemos dizer que é uma tabela em base de dados relacional ou uma planilha Excel com cabeçalho.
- Pode ser construído através de:
 - RDDs
 - Arquivos de dados estruturados
 - Tabelas no HIVE
 - Base de dados externas
- APIs disponíveis para Python, Scala, R e Java.
- Tem algumas características comuns entre RDD e DataFrame:
 - Imutável: podemos criar DataFrame/RDD uma vez, porém não podemos mudá-los.
 - As transformações não são executadas até que uma ação seja realizada.
 - RDD e DataFrame são distribuídos.
- Inspirada no estilo do R/pandas DataFrame para manipulação de dados tabulares.
- Interoperabilidade com Spark SQL
 - É possível salvar DataFrames como sendo uma tabela.
 - É possível fazer consultas no formato SQL

Por que utilizar DataFrames ao invés de RDD?

Para facilitar a adesão de novos usuários que já estão familiarizados com o conceito de DataFrame em outras linguagens de programação. Para os usuários do Spark, essa API permite que a programação seja mais simples do que utilizamos no RDD. Para ambos os usuários, a API DataFrame irá melhorar o desempenho através das otimizações inteligentes e também pela geração de código. E principalmente porque **DataFrames podem ser significativamente mais rápidos pelo RDDs** independente da linguagem escolhida, o desempenho será parecido.



Agora vamos apresentar um notebook demonstrando os usos básicos do DataFrame, voltado principalmente para novos usuários. As aplicações PySpark começam com a inicialização do SparkSession, que é o ponto de entrada do PySpark conforme abaixo. No caso de executá-lo no shell PySpark por meio do executável pyspark, o shell cria automaticamente a sessão na variável spark para os usuários. Abaixo um exemplo de criação do SparkSession.

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Criação de um DataFrame

Um PySpark DataFrame pode ser criado através de `pyspark.sql.SparkSession.createDataFrame`, normalmente passando uma lista de listas, tuplas, dicionários, `pyspark.sql.Rows`, um pandas DataFrame e um RDD consistindo dessa lista. `pyspark.sql.SparkSession.createDataFrame` usa o argumento do esquema para especificar o esquema/estrutura do DataFrame. Quando é omitido, o PySpark infere o esquema/estrutura correspondente a partir de uma amostra dos dados.

Por exemplo, em primeiro lugar, você pode criar um PySpark DataFrame a partir de uma lista de linhas

```
[2]: from datetime import datetime, date
import pandas as pd
from pyspark.sql import Row

df = spark.createDataFrame([
    Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
    Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
    Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3, 12, 0))
])
df

[2]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Abaixo um exemplo de criação de um PySpark DataFrame com um esquema explícito

```
[3]: df = spark.createDataFrame([
      (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
      (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
      (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
    ], schema='a long, b double, c string, d date, e timestamp')
df
```

```
[3]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Abaixo um exemplo de criação de um PySpark DataFrame a partir de um pandas DataFrame

```
[4]: pandas_df = pd.DataFrame({
      'a': [1, 2, 3],
      'b': [2., 3., 4.],
      'c': ['string1', 'string2', 'string3'],
      'd': [date(2000, 1, 1), date(2000, 2, 1), date(2000, 3, 1)],
      'e': [datetime(2000, 1, 1, 12, 0), datetime(2000, 1, 2, 12, 0), datetime(2000, 1, 3, 12, 0)]
    })
df = spark.createDataFrame(pandas_df)
df
```

```
[4]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Abaixo um exemplo de criação de um PySpark DataFrame a partir de um RDD que consiste em uma lista de tuplas.

```
[5]: rdd = spark.sparkContext.parallelize([
      (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
      (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
      (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
    ])
df = spark.createDataFrame(rdd, schema=['a', 'b', 'c', 'd', 'e'])
df
```

```
[5]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Os DataFrames criados acima têm os mesmos resultados e esquema.



```
[6]: # All DataFrames above result same.
df.show()
df.printSchema()
```

```
+---+---+---+---+
| a| b| c| d| e|
+---+---+---+---+
| 1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
| 2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
| 3|4.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+---+---+
```

```
root
|-- a: long (nullable = true)
|-- b: double (nullable = true)
|-- c: string (nullable = true)
|-- d: date (nullable = true)
|-- e: timestamp (nullable = true)
```

Para visualizar os dados de um DataFrame pode-se usar o `DataFrame.show()`. Para exibir as primeiras linhas de um DataFrame pode-se:

```
[7]: df.show(1)
```

```
+---+---+---+---+---+
| a| b| c| d| e|
+---+---+---+---+---+
| 1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+---+---+---+
only showing top 1 row
```

As linhas também podem ser mostradas verticalmente. Isso é útil quando as linhas são muito longas para serem exibidas horizontalmente.

```
[9]: df.show(1, vertical=True)
```

```
-RECORD 0-----
a | 1
b | 2.0
c | string1
d | 2000-01-01
e | 2000-01-01 12:00:00
only showing top 1 row
```

Você pode ver o esquema do DataFrame e os nomes das colunas da seguinte maneira:

```
[10]: df.columns
```

```
[10]: ['a', 'b', 'c', 'd', 'e']
```

```
[11]: df.printSchema()
```

```
root
|-- a: long (nullable = true)
|-- b: double (nullable = true)
|-- c: string (nullable = true)
|-- d: date (nullable = true)
|-- e: timestamp (nullable = true)
```



Exibindo o resumo do DataFrame

```
[12]: df.select("a", "b", "c").describe().show()
```

```
+---+---+---+---+
|summary| a| b| c|
+---+---+---+---+
| count| 3| 3| 3|
| mean| 2.0| 3.0| null|
| stddev| 1.0| 1.0| null|
| min| 1| 2.0| string1|
| max| 3| 4.0| string3|
+---+---+---+---+
```

`DataFrame.collect()` coleta os dados distribuídos e envia para o driver como os dados locais em Python. Observe que isso pode gerar um erro de falta de memória quando o conjunto de dados for muito grande para caber no driver, pois ele coleta todos os dados dos executores.

```
[13]: df.collect()
```

```
[13]: [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12, 0, 0)),
      Row(a=2, b=3.0, c='string2', d=datetime.date(2000, 2, 1), e=datetime.datetime(2000, 1, 2, 12, 0, 0)),
      Row(a=3, b=4.0, c='string3', d=datetime.date(2000, 3, 1), e=datetime.datetime(2000, 1, 3, 12, 0, 0))]
```

Para evitar o uma exceção de falta de memória, use `DataFrame.take()` ou `DataFrame.tail()`.

```
[14]: df.take(1)
[14]: [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12:00:00))]
```

O PySpark DataFrame também fornece a conversão de volta para um DataFrame do Pandas. Observe que o `toPandas` também coleta todos os dados para o driver que podem facilmente causar um erro de falta de memória quando os dados são muito grandes para caber no driver.

```
[15]: df.toPandas()
[15]:
```

	a	b	c	d	e
0	1	2.0	string1	2000-01-01	2000-01-01 12:00:00
1	2	3.0	string2	2000-02-01	2000-01-02 12:00:00
2	3	4.0	string3	2000-03-01	2000-01-03 12:00:00

O PySpark DataFrame é avaliado lentamente e simplesmente selecionar uma coluna não dispara nenhum poder computacional, mas retorna uma instância de coluna.

```
[16]: df.a
[16]: Column<b'a'>
```

Na verdade, a maioria das operações de colunas retornam Colunas

```
[17]: from pyspark.sql import Column
      from pyspark.sql.functions import upper
      type(df.c) == type(upper(df.c)) == type(df.c.isNull())
[17]: True
```

Essas colunas podem ser usadas para selecionar as colunas de um DataFrame. Por exemplo, `DataFrame.select()` obtém as instâncias de Coluna que retornam outro DataFrame.

```
[18]: df.select(df.c).show()
+-----+
|      c|
+-----+
|string1|
|string2|
|string3|
+-----+
```

E para criar uma nova coluna no DataFrame. Neste exemplo a coluna 'upper_c' está sendo adicionada ao DataFrame.

```
[19]: df.withColumn('upper_c', upper(df.c)).show()
+---+---+-----+-----+-----+
| a | b | c | d | e | upper_c |
+---+---+-----+-----+-----+
| 1 | 2.0 | string1 | 2000-01-01 | 2000-01-01 12:00:00 | STRING1 |
| 2 | 3.0 | string2 | 2000-02-01 | 2000-01-02 12:00:00 | STRING2 |
| 3 | 4.0 | string3 | 2000-03-01 | 2000-01-03 12:00:00 | STRING3 |
+---+---+-----+-----+-----+
```

Para selecionar um subconjunto de linhas, use `DataFrame.filter()`. Aqui filtrando os dados que onde o campo 'a' é igual a 1.


```
[20]: df.filter(df.a == 1).show()
```

a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00

O PySpark oferece suporte a vários user-defined function (UDFs) e APIs para permitir que os usuários executem funções nativas do Python. Uma UDF é uma função definida pelo usuário que usa Apache Arrow para transferir dados e pandas para manipular os dados. Os UDFs do pandas permitem operações vetorizadas que podem aumentar o desempenho em até 100x em comparação com os UDFs Python linha a linha. Para saber mais sobre UDFs veja:

<https://docs.databricks.com/spark/latest/spark-sql/udf-python-pandas.html>

O exemplo abaixo permite que os usuários usem diretamente as APIs em uma série pandas dentro da função nativa Python. Veja que neste exemplo é criada uma função chamada 'pandas_plus_one' que é utilizada em tempo de seleção dos dados do DataFrame, recebendo como parâmetro a coluna 'a' do DataFrame.

```
[21]: import pandas
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(series: pd.Series) -> pd.Series:
    # Simply plus one by using pandas Series.
    return series + 1

df.select(pandas_plus_one(df.a)).show()
```

pandas_plus_one(a)
2
3
4

Outro exemplo é o `DataFrame.mapInPandas`, que permite aos usuários usar diretamente as APIs em um DataFrame do pandas sem quaisquer restrições, como o tamanho do resultado.

```
[22]: def pandas_filter_func(iterator):
    for pandas_df in iterator:
        yield pandas_df[pandas_df.a == 1]

df.mapInPandas(pandas_filter_func, schema=df.schema).show()
```

a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00

O PySpark DataFrame também oferece uma maneira simples de lidar com dados agrupados através de uma abordagem comum, estratégia de divisão-aplicação-combinação ou do inglês split-apply-combine. Ele agrupa os dados por uma determinada condição, aplica uma função a cada grupo e os combina de volta ao DataFrame.

```
[23]: df = spark.createDataFrame([
    ['red', 'banana', 1, 10], ['blue', 'banana', 2, 20], ['red', 'carrot', 3, 30],
    ['blue', 'grape', 4, 40], ['red', 'carrot', 5, 50], ['black', 'carrot', 6, 60],
    ['red', 'banana', 7, 70], ['red', 'grape', 8, 80]], schema=['color', 'fruit', 'v1', 'v2'])
df.show()
```

color	fruit	v1	v2
red	banana	1	10
blue	banana	2	20
red	carrot	3	30
blue	grape	4	40
red	carrot	5	50
black	carrot	6	60
red	banana	7	70
red	grape	8	80

Agrupando os dados da coluna 'color' e aplicando a função `avg()`.

```
[24]: df.groupby('color').avg().show()
```

color	avg(v1)	avg(v2)
red	4.8	48.0
black	6.0	60.0
blue	3.0	30.0

Você também pode aplicar uma função nativa do Python em cada grupo usando APIs do pandas.

```
[25]: def plus_mean(pandas_df):
    return pandas_df.assign(v1=pandas_df.v1 - pandas_df.v1.mean())

df.groupby('color').applyInPandas(plus_mean, schema=df.schema).show()
```

color	fruit	v1	v2
red	banana	-3	10
red	carrot	-1	30
red	carrot	0	50
red	banana	2	70
red	grape	3	80
black	carrot	0	60
blue	banana	-1	20
blue	grape	1	40

Co-agrupamento e aplicação de uma função.

```
[26]: df1 = spark.createDataFrame(
      [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1, 3.0), (20000102, 2, 4.0)],
      ('time', 'id', 'v1'))

df2 = spark.createDataFrame(
      [(20000101, 1, 'x'), (20000101, 2, 'y')],
      ('time', 'id', 'v2'))

def asof_join(l, r):
    return pd.merge_asof(l, r, on='time', by='id')

df1.groupby('id').cogroup(df2.groupby('id')).applyInPandas(
    asof_join, schema='time int, id int, v1 double, v2 string').show()

+-----+-----+-----+
|   time| id| v1| v2|
+-----+-----+-----+
|20000101| 1|1.0| x|
|20000102| 1|3.0| x|
|20000101| 2|2.0| y|
|20000102| 2|4.0| y|
+-----+-----+-----+
```

Para trabalhar com arquivos CSV é direto e fácil de usar. Parquet e ORC são formatos de arquivo compactos e eficientes para leitura e gravação mais rápida e também podem ser utilizados. Existem muitas outras fontes de dados disponíveis no PySpark, como JDBC, text, binaryFile, Avro, etc. Consulte também o Spark SQL, DataFrames e Guia de conjuntos de dados mais recentes na documentação do Apache Spark (<https://spark.apache.org/docs/latest/sql-programming-guide.html>).

CSV

```
[27]: df.write.csv('foo.csv', header=True)
spark.read.csv('foo.csv', header=True).show()

+-----+-----+-----+
|color| fruit| v1| v2|
+-----+-----+-----+
| red|banana| 1| 10|
| blue|banana| 2| 20|
| red|carrot| 3| 30|
| blue|grape| 4| 40|
| red|carrot| 5| 50|
| black|carrot| 6| 60|
| red|banana| 7| 70|
| red|grape| 8| 80|
+-----+-----+-----+
```

Parquet

```
[28]: df.write.parquet('bar.parquet')
spark.read.parquet('bar.parquet').show()

+-----+-----+-----+
|color| fruit| v1| v2|
+-----+-----+-----+
| red|banana| 1| 10|
| blue|banana| 2| 20|
| red|carrot| 3| 30|
| blue|grape| 4| 40|
| red|carrot| 5| 50|
| black|carrot| 6| 60|
| red|banana| 7| 70|
| red|grape| 8| 80|
+-----+-----+-----+
```

ORC

```
[29]: df.write.orc('zoo.orc')
      spark.read.orc('zoo.orc').show()
```

```
+-----+-----+-----+
|color| fruit| v1| v2|
+-----+-----+-----+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red| carrot|  3| 30|
| blue|  grape|  4| 40|
|  red| carrot|  5| 50|
| black| carrot|  6| 60|
|  red|banana|  7| 70|
|  red|  grape|  8| 80|
+-----+-----+-----+
```

DataFrame e Spark SQL compartilham o mesmo mecanismo de execução para que possam ser usados de forma intercambiável. Por exemplo, você pode registrar o DataFrame como uma tabela e executar um SQL facilmente como a seguir:

```
[30]: df.createOrReplaceTempView("tableA")
      spark.sql("SELECT count(*) from tableA").show()
```

```
+-----+
|count(1)|
+-----+
|      8|
+-----+
```

Além disso, UDFs podem ser registrados e invocados em SQL prontos para uso:

```
[31]: @pandas_udf("integer")
      def add_one(s: pd.Series) -> pd.Series:
          return s + 1

      spark.udf.register("add_one", add_one)
      spark.sql("SELECT add_one(v1) FROM tableA").show()
```

```
+-----+
|add_one(v1)|
+-----+
|          2|
|          3|
|          4|
|          5|
|          6|
|          7|
|          8|
|          9|
+-----+
```

Essas expressões SQL podem ser combinadas diretamente e usadas como colunas PySpark.

```
[32]: from pyspark.sql.functions import expr
```

```
df.selectExpr('add_one(v1)').show()  
df.select(expr('count(*)') > 0).show()
```

```
+-----+  
|add_one(v1)|  
+-----+  
|          2|  
|          3|  
|          4|  
|          5|  
|          6|  
|          7|  
|          8|  
|          9|  
+-----+
```

```
+-----+  
|(count(1) > 0)|  
+-----+  
|          true|  
+-----+
```

Material complementar:

Caso queiram conhecer mais sobre Apache Spark, seguem alguns cursos muito bons:

- Mini Curso Getting started with Spark & Python
<https://www.coursera.org/lecture/big-data-essentials/getting-started-with-spark-python-o6oKt>
- Vídeo para Iniciantes:
https://youtu.be/dK_F1tuK9C4
- Tutorial para aprender Spark com Python:
<https://www.roseindia.net/bigdata/pyspark/index.shtml>
- Spark para Iniciantes do Canal do YouTube Apache Spark Tutorial - Frank Kane
<https://www.youtube.com/playlist?list=PL6cactdCCnTJ2XZYIwLperpbKB86jv>

Exercícios:

Para execução dos exercícios utilize a instalação realizada na sprint anterior.

- 1) Crie um array de 250 `ints` e aplique o método `reverse` para inverter o conteúdo.
- 2) Crie uma lista de 20 animais, ordene-os e itere para imprimi-las individualmente cada um deles. Depois salve num arquivo texto em formato CSV
- 3) Executar o Laboratório: “Gerar dados processar e criar arquivo texto”

Submetendo Aplicações Spark

O Spark possui scripts facilitadores. O script `spark-submit` localizado na pasta `bin`, é usado para iniciar aplicativos em um cluster. Ele pode usar todos os gerenciadores de cluster suportados do Spark por meio de uma interface uniforme, para que você não precise configurar seu aplicativo especialmente para cada um[2]. Ou seja, um processo Spark, independente do módulo onde o código foi desenvolvido pode ser iniciado através do script `Spark-submit` e ser gerenciado pelo gerenciador de recursos previamente definido. No caso de uma EMR da AWS o gerenciador utilizado é o YARN.

Depois que um aplicativo de usuário é empacotado, ele pode ser iniciado usando o script `spark-submit`. Esse script cuida da configuração do classpath com o Spark e suas dependências, e pode suportar diferentes gerenciadores de cluster e modos de deploy. A estrutura suportada pelo `spark-submit`:

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Algumas das opções mais usadas são:

- **--class**: o ponto de entrada do seu aplicativo (por exemplo, `org.apache.spark.examples.SparkPi`)
- **--master**: o URL principal do cluster (por exemplo, `spark: //23.195.26.187: 7077`)
- **--deploy-mode**: implantar seu driver nos nodes do worker (cluster) ou localmente como um cliente externo (client) (padrão: client)
- **--conf**: propriedade de configuração forçadas do Spark no formato `key=value`. Para valores que contêm espaços, envolva “`key=value`” entre aspas (como mostrado).
- **application-jar**: caminho para um jar adicional, incluindo seu aplicativo e todas as dependências. A URL deve estar globalmente visível dentro do seu cluster, por exemplo, um caminho `hdfs: //` ou um arquivo: `//` que esteja presente em todos os nós.
- **application-arguments**: argumentos passados para o método principal da sua classe principal, se houver

Uma estratégia comum de deploy é submeter sua aplicação a partir de uma máquina gateway que esteja fisicamente localizada com as máquinas de works (por exemplo, nó mestre em um cluster EC2 independente). Nesta configuração, o modo `client` é apropriado. No modo `client`, o driver é iniciado diretamente no processo `spark-submit` que atua como um cliente para o cluster. A entrada e a saída da aplicação estão anexadas ao console. Portanto, esse modo é especialmente adequado para aplicações que envolvem o REPL (por exemplo, `shell Spark`).

Como alternativa, se sua aplicação for enviada de uma máquina distante das máquinas worker (por exemplo, localmente no seu laptop), é comum usar o modo de `cluster` para minimizar a latência da rede entre os drivers e os executores. Atualmente, o modo `standalone` não suporta o modo de `cluster` para aplicações Python.

Para aplicações Python, basta passar um arquivo .py no lugar de um arquivo JAR, e adicionar arquivos Python .zip, .egg ou .py ao caminho de pesquisa com --py-files. Abaixo alguns exemplos do site[2]:

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# Run a Python application on a Spark standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```



Exercício:




4) Realize a execução de código de cada exemplo e exercício dos itens anteriores via spark-submit

Desafio XPTO: Processamento do Histórico

EMR

Anteriormente, na semana 6, vimos o que é uma EMR. Agora vamos saber como executar uma aplicação Spark numa EMR. Uma EMR possui diversas versões e diferentes produtos disponíveis em cada uma das versões. Uma versão do Amazon EMR é um conjunto de aplicações de código aberto no ecossistema de big data. Cada versão contém diferentes aplicações de big data, componentes e recursos que você seleciona para que o Amazon EMR instale e configure quando você criar um cluster. As aplicações são empacotadas usando um sistema baseado no Apache BigTop, que é um projeto de código aberto associado ao ecossistema do Hadoop.

Ao iniciar um cluster, você pode escolher entre várias versões do Amazon EMR. Isso permite que você teste e use versões de aplicativos que atendam aos requisitos de compatibilidade. Especifique a versão usando o rótulo da versão. Os rótulos de versão estão no formato emr-x.x.x. For example, emr-6.2.0. Abaixo uma lista de componentes desde exemplo:



6.0.0 Mar 2020	6.1.0 Sep 2020	6.2.0 Dec 2020	
3.2.1	3.2.1	3.2.1	Hadoop
	1.11.0	1.11.2	Flink
3.7.2	3.7.2	3.7.2	Ganglia
2.2.3	2.2.3	2.2.6	HBase
3.1.2	3.1.2	3.1.2	Hive & HCatalog
0.5.0-inc	0.5.2-inc	0.6.0	Hudi
4.4.0	4.7.1	4.8.0	Hue
1.0.0	1.1.0	1.1.0	JupyterHub
0.6.0	0.7.0	0.7.0	Livy
1.5.1	1.6.0	1.7.0	MXNet
5.1.0	5.2.0	5.2.0	Oozie
5.0.0	5.0.0	5.0.0	Phoenix
	0.17.0	0.17.0	Pig
	0.230	0.238.3	Presto(DB)
0.230	338	343	PrestoSQL
2.4.4	3.0.0	3.0.1	Spark
	N/A	1.4.7	Sqoop
1.14.0	2.1.0	2.3.1	Tensorflow
0.9.2	0.9.2	0.9.2	Tez
0.9.0-SNAPSHOT	0.9.0-preview1	0.9.0-preview1	Zeppelin
3.4.14	3.4.14	3.4.14	Zookeeper
1.11.711	1.11.828	1.11.880	AWS SDK for Java

Vamos usar o EMR com o Spark para executar o processamento dos dados históricos da RAW para a REF.

Dicas:

- Crie um Cluster somente com o nó master e com a menor configuração possível com Spark
- Para rodar a carga histórica da RAW para a REF use o comando do Spark: `spark-submit`.

Glue

Anteriormente, na semana 6, vimos o que é o Glue e como utilizá-lo. Use o material da semana 6 como apoio para a execução dos desafios.

Vamos usar um Glue Job com Spark para o processamento dos dados históricos da RAW para a REF.

Dicas:

- Crie Role com Permissões de GLUE, S3 e CloudWatch
- Utilize um Glue Job com Spark para executar o processamento dos dados Históricos da RAW para a REF.

Exercícios:

- 5) **Desafio XPTO:** Carregar os dados da RAW Zone para a REF Zone, unificando os dados num Bucket S3 persistente no formato parquet. Particionar os dados por Ano/Mês/Dia conforme a criação do Twitter. Incluir 2 colunas novas:
- Sentimento: indicando Positivo quando encontrar algum símbolo como :D ou :) ou :) etc; Indicando Negativo quando encontrar algum símbolo como :(ou :[ou :{ etc. Indicando Neutro, quando o tweet não tiver nenhum dos símbolos analisados. Se um tweet tiver vários símbolos apenas o primeiro encontrado deve ser utilizado
 - Símbolo: Nesta coluna você deve adicionar o símbolo encontrado no tweet. Se um tweet tiver vários símbolos apenas o primeiro encontrado deve ser utilizado

Vamos fazer o código Spark criado executar via EMR e Glue com a finalidade de comparação das soluções.

Referências

[1] <https://www.devmedia.com.br/introducao-ao-apache-spark/34178>

[2] <https://spark.apache.org/docs/latest/quick-start.html>

