Dataframe_Basics

June 11, 2019

0.1 Absolute basics of PySpark DataFrame

0.1.1 Apache Spark

Apache Spark is one of the hottest new trends in the technology domain. It is the framework with probably the **highest potential to realize the fruit of the marriage between Big Data and Machine Learning**. It runs fast (up to 100x faster than traditional Hadoop MapReduce) due to in-memory operation, offers robust, distributed, fault-tolerant data objects (called RDD), and integrates beautifully with the world of machine learning and graph analytics through supplementary packages like Mlib and GraphX.

Spark is implemented on Hadoop/HDFS and written mostly in Scala, a functional programming language, similar to Java. In fact, Scala needs the latest Java installation on your system and runs on JVM. However, for most of the beginners, Scala is not a language that they learn first to venture into the world of data science. Fortunately, Spark provides a wonderful Python integration, called PySpark, which lets Python programmers to interface with the Spark framework and learn how to manipulate data at scale and work with objects and algorithms over a distributed file system.

0.1.2 DataFrame

In Apache Spark, a DataFrame is a distributed collection of rows under named columns. It is conceptually equivalent to a table in a relational database, an Excel sheet with Column headers, or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. It also shares some common characteristics with RDD:

- **Immutable in nature** : We can create DataFrame / RDD once but can't change it. And we can transform a DataFrame / RDD after applying transformations.
- **Lazy Evaluations**: Which means that a task is not executed until an action is performed.
- **Distributed**: RDD and DataFrame both are distributed in nature.

0.1.3 Advantages of the DataFrame

- DataFrames are designed for processing large collection of structured or semi-structured data.
- Observations in Spark DataFrame are organised under named columns, which helps Apache Spark to understand the schema of a DataFrame. This helps Spark optimize execution plan on these queries.
- DataFrame in Apache Spark has the ability to handle petabytes of data.

- DataFrame has a support for wide range of data format and sources.
- It has API support for different languages like Python, R, Scala, Java.

```
In [1]: import pyspark
```

```
In [5]: from pyspark import SparkContext as sc
        from pyspark.sql import Row
```

0.1.4 Create a *SparkSession app* object

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

```
In [3]: spark1 = SparkSession.builder.appName('Basics').getOrCreate()
```

0.1.5 Read in a JSON file and examine

```
In [4]: df = spark1.read.json('Data/people.json')
```

Unlike Pandas DataFrame, it does not show itself when called

```
In [5]: df
```

```
Out[5]: DataFrame[age: bigint, name: string]
```

You have to call `show()` method to evaluate it i.e. show it

```
In [6]: df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Use `printSchema()` to show he schema of the data. Note, how tightly it is integrated to the SQL-like framework. You can even see that the schema accepts `null` values because *nullable* property is set `True`.

```
In [7]: df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

Fortunately a simple `columns` method exists to get column names back as a Python list

```
In [12]: col_list=df.columns
```

```
In [13]: col_list
```

```
Out[13]: ['age', 'name']
```

```
In [14]: type(col_list)
```

```
Out[14]: list
```

Similar to Pandas, the `describe` method is used for the statistical summary

```
In [9]: df.describe
```

```
Out[9]: <bound method DataFrame.describe of DataFrame[age: bigint, name: string]>
```

But unlike Pandas, calling only `describe()` returns a DataFrame!

```
In [10]: df.describe()
```

```
Out[10]: DataFrame[summary: string, age: string, name: string]
```

True to the spirit of lazy evaluation, you have to evaluate the resulting DataFrame by calling `show()`

```
In [11]: df.describe().show()
```

```
+-------+------------------+-------+
|summary|               age|   name|
+-------+------------------+-------+
|  count|                 2|      3|
|   mean|              24.5|   null|
| stddev|7.7781745930520225|   null|
|    min|                19|   Andy|
|    max|                30|Michael|
+-------+------------------+-------+
```

You can also use `summary()` method for more descriptive statistics including quartiles

```
In [42]: df.summary().show()
```

```
+-------+------------------+-------+
|summary|               age|   name|
+-------+------------------+-------+
|  count|                 2|      3|
|   mean|              24.5|   null|
```

```
| stddev|7.7781745930520225|   null|
|    min|                19|   Andy|
|    25%|                19|   null|
|    50%|                19|   null|
|    75%|                30|   null|
|    max|                30|Michael|
+-------+------------------+-------+
```

0.1.6 How you can define your own Data Schema

Import data types and structure types to build the data schema yourself

```
In [16]: from pyspark.sql.types import StructField, IntegerType, StringType, StructType
```

Define your data schema by supplying name and data types to the structure fields you will be importing

```
In [17]: data_schema = [StructField('age',IntegerType(),True),
                        StructField('name',StringType(),True)]
```

Now create a `StrucType` with this schema as field

```
In [18]: final_struc = StructType(fields=data_schema)
```

Now read in the same old JSON with this new schema

```
In [19]: df = spark1.read.json('Data/people.json',schema=final_struc)
```

```
In [20]: df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Now when you print the schema, you will see that the *age* is read as `int` and not `long`. By default Spark could not figure out for this column the exact data type that you wanted, so it went with `long`. But this is how you can build your own schema and instruct Spark to read the data accoridngly.

```
In [21]: df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- name: string (nullable = true)
```

0.1.7 How to grab data from the DataFrame; *Column* and *Row* objects

What is the type of a single column?

```
In [34]: type(df['age'])
```

```
Out[34]: pyspark.sql.column.Column
```

But how to extract a single column as a DataFrame? Use `select()`

```
In [35]: df.select('age')
```

```
Out[35]: DataFrame[age: int]
```

```
In [36]: df.select('age').show()
```

```
+----+
| age|
+----+
|null|
|  30|
|  19|
+----+
```

What is Row object?

```
In [37]: df.head(2)
```

```
Out[37]: [Row(age=None, name='Michael'), Row(age=30, name='Andy')]
```

```
In [38]: df.head(2)[0]
```

```
Out[38]: Row(age=None, name='Michael')
```

```
In [43]: row0=df.head(2)[0]
```

You can get back a normal Python dictionary from the row object

```
In [50]: row0.asDict()
```

```
Out[50]: {'age': None, 'name': 'Michael'}
```

Remember that in Pandas DataFrame we have `pandas.series` object as either column or row. The reason Spark offers separate `Column` or `Row` object is the ability to work over a distributed file system where this distinction will come handy.

0.1.8 Creating new column

You cannot think like Pandas. Following will produce error

```
In [63]: df['newage']=2*df['age']
```

```
    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-63-32731f3b98cc> in <module>()
----> 1 df['newage']=2*df['age']


    TypeError: 'DataFrame' object does not support item assignment
```

Use `useColumn()` method instead

```
In [64]: df.withColumn('double_age',df['age']*2).show()
```

```
+----+-------+----------+
| age|   name|double_age|
+----+-------+----------+
|null|Michael|      null|
|  30|   Andy|        60|
|  19| Justin|        38|
+----+-------+----------+
```

Just for renaming, use `withColumnRenamed()` method

```
In [65]: df.withColumnRenamed('age','my_new_age').show()
```

```
+----------+-------+
|my_new_age|   name|
+----------+-------+
|      null|Michael|
|        30|   Andy|
|        19| Justin|
+----------+-------+
```

You can do operation with multiple columns, like a vector sum

```
In [67]: df2=df.withColumn('half_age',df['age']/2)
         df2.show()
```

```
+----+-------+--------+
| age|   name|half_age|
+----+-------+--------+
|null|Michael|    null|
|  30|   Andy|    15.0|
|  19| Justin|     9.5|
+----+-------+--------+
```

```
In [68]: df2=df2.withColumn('new_age',df2['age']+df2['half_age'])
         df2.show()
```

```
+----+-------+--------+-------+
| age|   name|half_age|new_age|
+----+-------+--------+-------+
|null|Michael|    null|   null|
|  30|   Andy|    15.0|   45.0|
|  19| Justin|     9.5|   28.5|
+----+-------+--------+-------+
```

Now if you print the schema, you will see that the data type of *half_age* and *new_age* are automaically set to `double` (due to floating point operation performed)

```
In [69]: df2.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- half_age: double (nullable = true)
 |-- new_age: double (nullable = true)
```

DataFrame is immutable and there is no `inplace` choice like Pandas! So the original DataFrame has not changed

```
In [66]: df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
```

```
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

0.1.9 Integration with SparkSQL - Run SQL query!

You may be wondering why this `SparkSession` object came out of `spark.sql` class. That is because it is tightly integrated with the SparkSQL and is designed to work with SQL or SQL-like queries seamlessly for data analytics.

It is good to create a temporary view of the DataFrame. Here `people` is the name of the SQL table view.

```
In [70]: df.createOrReplaceTempView('people')
```

Now run a simple SQL query directly on this view. It returns a DataFrame.

```
In [72]: result = spark1.sql("SELECT * FROM people")
         result
```

```
Out[72]: DataFrame[age: int, name: string]
```

```
In [73]: result.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Slightly more complex query

```
In [74]: result_over_25 = spark1.sql("SELECT * FROM people WHERE age > 25")
         result_over_25.show()
```

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```