

pyspark: pandas-on-Spark

Justin Post

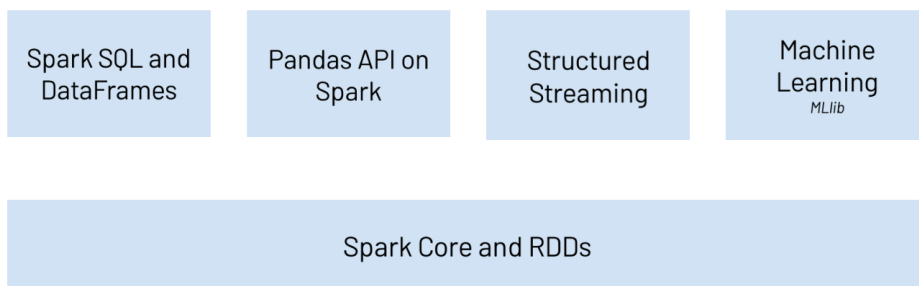
Spark Recap

Spark - Distributed processing software for big data workloads

- Generally faster than Hadoop's MapReduce (and much more flexible)
- DAGs make it fault tolerant and improve computational speed

Five major parts to (py)Spark

- Spark Core and RDDs as its foundation
- Spark SQL and DataFrames
- Pandas on Spark
- Spark Structured Streaming
- Spark Machine Learning (MLlib)



Data Object Used by pyspark

DataFrame APIs are commonly used in pyspark

- DataFrames (think usual relational database table) are created and implemented on top of RDDs
- DataFrames are stored across the cluster
 - When transformations are done, lazy evaluation is used
 - When actions are done, computation starts and results returned

Data Object Used by pyspark

DataFrame APIs are commonly used in pyspark

- DataFrames (think usual relational database table) are created and implemented on top of RDDs
- DataFrames are stored across the cluster
 - When transformations are done, lazy evaluation is used
 - When actions are done, computation starts and results returned

Two major DataFrame APIs in pyspark

- pandas-**on-Spark** DataFrames through the `pyspark.pandas` module
- **Spark SQL** DataFrames through `pyspark.sql` module

pandas-on-Spark

- pandas API on spark is super easy to use since we know pandas!
- First we can import our modules

```
{ import pandas as pd  
import numpy as np  
import pyspark.pandas as ps }
```

pandas-on-Spark

- Now you can create a pandas-on-Spark series or a pandas-on-Spark DataFrame
- Note the ps not pd!

```
ps.Series([1, 3, 5, np.nan, 6, 8])
```

```
0    1.0
```

```
1    3.0
```

```
2    5.0
```

```
3    NaN
```

```
4    6.0
```

```
5    8.0
```

```
dtype: float64
```

pandas-on-Spark

- Now you can create a pandas-on-Spark series or a pandas-on-Spark DataFrame
- Note the ps not pd!

```
ps.DataFrame(  
    {'a': [1, 2, 3, 4, 5, 6],  
     'b': [100, 200, 300, 400, 500, 600],  
     'c': ["one", "two", "three", "four", "five", "six"]},  
    index=[10, 20, 30, 40, 50, 60])
```

| | a | b | c |
|----|---|-----|-------|
| 10 | 1 | 100 | one |
| 20 | 2 | 200 | two |
| 30 | 3 | 300 | three |
| 40 | 4 | 400 | four |
| 50 | 5 | 500 | five |
| 60 | 6 | 600 | six |

pandas-on-Spark

- We can also convert from pandas to pandas-on-Spark

```
pdf = pd.read_csv("https://www4.stat.ncsu.edu/~online/datasets/red-wine.csv", delimiter = ";")  
psdf = ps.from_pandas(pdf)  
psdf.head()
```

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---------------|------------------|-------------|----------------|-----------|---------------------|----------------------|---------|------|-----------|---------|---------|
| 0 | 7.4 | 0.70 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | 9.4 | 5 |
| 1 | 7.8 | 0.88 | 0.00 | 2.6 | 0.098 | 25.0 | 67.0 | 0.9968 | 3.20 | 0.68 | 9.8 | 5 |
| 2 | 7.8 | 0.76 | 0.04 | 2.3 | 0.092 | 15.0 | 54.0 | 0.9970 | 3.26 | 0.65 | 9.8 | 5 |
| 3 | 11.2 | 0.28 | 0.56 | 1.9 | 0.075 | 17.0 | 60.0 | 0.9980 | 3.16 | 0.58 | 9.8 | 6 |
| 4 | 7.4 | 0.70 | 0.00 | 1.9 | 0.076 | 11.0 | 34.0 | 0.9978 | 3.51 | 0.56 | 9.4 | 5 |

pandas-on-Spark

- We now have a much of the same functionality from pandas available through pandas-on-Spark ([API reference guide](#))
 - `.index`, `.columns`, `.shape`, `.info`
 - `.head()`, `.tail()`
 - `[["column1", "column2"]].loc[]`
 - `.mean()`, `.sum()`, `.groupby()`, `.describe()`, `.value_counts()`

pandas-on-Spark

- We now have a much of the same functionality from pandas available through pandas-on-Spark ([API reference guide](#))
 - .index, .columns, .shape, .info
 - .head(), tail()
 - [["column1", "column2"]], .loc[]
 - .mean(), .sum(), .groupby(), .describe(), .value_counts()

```
psdf.loc[psdf.quality > 5, ["alcohol", "quality"]].head()
```

| | alcohol | quality |
|----|---------|---------|
| 3 | 9.8 | 6 |
| 7 | 10.0 | 7 |
| 8 | 9.5 | 7 |
| 16 | 10.5 | 7 |
| 19 | 9.2 | 6 |

pandas-on-Spark

- We now have a much of the same functionality from pandas available through pandas-on-Spark ([API reference guide](#))
 - .index, .columns, .shape, .info
 - .head(), tail()
 - [["column1", "column2"]], .loc[]
 - .mean(), .sum(), .groupby(), .describe(), .value_counts()

```
titanic_ps = ps.read_csv("titanic.csv") #data uploaded to jhub in data folder  
titanic_ps["survived"].value_counts()
```

```
0.0    809  
1.0    500  
Name: survived, dtype: int64
```

pandas-on-Spark

- We now have a much of the same functionality from pandas available through pandas-on-Spark ([API reference guide](#))
 - `.index`, `.columns`, `.shape`, `.info`
 - `.head()`, `tail()`
 - `[["column1", "column2"]]`, `.loc[]`
 - `.mean()`, `.sum()`, `.groupby()`, `.describe()`, `.value_counts()`

```
titanic_ps.groupby("survived").mean()
```

| | pclass | age | sibsp | parch | fare | body |
|----------|----------|-----------|----------|----------|-----------|------------|
| survived | | | | | | |
| 1 | 1.962000 | 28.918228 | 0.462000 | 0.476000 | 49.361184 | NaN |
| 0 | 2.500618 | 30.545369 | 0.521632 | 0.328801 | 23.353831 | 160.809917 |

pandas-on-Spark

- We now have a much of the same functionality from pandas available through pandas-on-Spark ([API reference guide](#))
 - .index, .columns, .shape, .info
 - .head(), tail()
 - [["column1", "column2"]], .loc[]
 - .mean(), .sum(), .groupby(), .describe(), .value_counts()

```
titanic_ps.describe()
```

| | pclass | survived | age | sibsp | parch | fare | body |
|-------|-------------|-------------|-------------|-------------|-------------|-------------|------------|
| count | 1309.000000 | 1309.000000 | 1046.000000 | 1309.000000 | 1309.000000 | 1308.000000 | 121.000000 |
| mean | 2.294882 | 0.381971 | 29.881135 | 0.498854 | 0.385027 | 33.295479 | 160.809917 |
| std | 0.837836 | 0.486055 | 14.413500 | 1.041658 | 0.865560 | 51.758668 | 97.696922 |
| min | 1.000000 | 0.000000 | 0.166700 | 0.000000 | 0.000000 | 0.000000 | 1.000000 |
| 25% | 2.000000 | 0.000000 | 21.000000 | 0.000000 | 0.000000 | 7.895800 | 72.000000 |
| 50% | 3.000000 | 0.000000 | 28.000000 | 0.000000 | 0.000000 | 14.454200 | 155.000000 |
| 75% | 3.000000 | 1.000000 | 39.000000 | 1.000000 | 0.000000 | 31.275000 | 256.000000 |
| max | 3.000000 | 1.000000 | 80.000000 | 8.000000 | 9.000000 | 512.329200 | 328.000000 |

```
]
```

pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

`def standardize(pser) -> ps.Series[np.float64]:`
 `return (pser - pser.mean())/pser.std() # should always return the same length as input.`

ps series

*specify exactly what the output
of your function should be*

pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

```
def standardize(pser) -> ps.Series[np.float64]:  
    return (pser + pser.mean())/pser.std() # should always return the same length as input.
```

```
std_res = titanic_ps[["age", "fare"]] \  
    .rename(columns = {"age": "o_age", "fare": "o_fare"}) \  
    .join(titanic_ps[["age", "fare"]] \  
        .transform(standardize))
```

std versions

| | o_age | o_fare | age | fare |
|---|---------|----------|----------|----------|
| 0 | 29.0000 | 211.3375 | 4.085138 | 4.726416 |
| 1 | 0.9167 | 151.5500 | 2.136735 | 3.571295 |
| 2 | 2.0000 | 151.5500 | 2.211894 | 3.571295 |
| 3 | 30.0000 | 151.5500 | 4.154517 | 3.571295 |
| 4 | 25.0000 | 151.5500 | 3.807620 | 3.571295 |

pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

```
def standardize(pser) -> ps.Series[np.float64]:  
    return (pser + pser.mean())/pser.std() # should always return the same length as input.
```

```
std_res = titanic_ps[["age", "fare"]] \  
    .rename(columns = {"age": "o_age", "fare": "o_fare"}) \  
    .join(titanic_ps[["age", "fare"]]  
        .transform(standardize))  
std_res.shape
```

```
(1310, 4)
```


pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

```
def standardize_positives(pser) -> ps.Series[np.float64]:  
    return (pser[pser>30] + pser[pser>30].mean())/pser[pser>30].std()  
# can return something short than input length
```

pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

```
def standardize_positives(pser) -> ps.Series[np.float64]:  
    return (pser[pser>30] + pser[pser>30].mean())/pser[pser>30].std()  
# can return something short than input length
```

```
std_pos = titanic_ps[["age"]].apply(standardize_positives)  
std_pos.head()
```

| | age |
|---|-----------|
| 0 | 9.135889 |
| 1 | 10.636052 |
| 2 | 8.235791 |
| 3 | 9.635943 |
| 4 | 11.436139 |

pandas-on-Spark

- `.transform()` and `.apply()` methods allow you to perform **operations on columns or rows**

```
def standardize_positives(pser) -> ps.Series[np.float64]:  
    return (pser[pser>30] + pser[pser>30].mean())/pser[pser>30].std()  
# can return something short than input length
```

```
std_pos = titanic_ps[["age"]].apply(standardize_positives)  
std_pos.shape
```

(437, 1)

To Jupyterlab

- Let's more easily handle the counting of words in our Oliver Twist example!

Recap

- **DataFrames** are the type of object (and name of the API) commonly used in `pyspark`
 - DataFrames built on RDDs
- **pandas-on-Spark** DataFrames through the `pyspark.pandas` module
 - Most of the usual pandas functionality!
- Lazy eval allows you to build up your transformations and then execute only when an action is performed

Important to know **limitations on** `pandas` **functionality**