Quickstart: Pandas API on Spark

This is a short introduction to pandas API on Spark, geared mainly for new users. This notebook shows you some key differences between pandas and pandas API on Spark. You can run this examples by yourself in 'Live Notebook: pandas API on Spark' at the quickstart page.

Customarily, we import pandas API on Spark as follows:

[1]:
```
import pandas as pd
import numpy as np
import pyspark.pandas as ps
from pyspark.sql import SparkSession
```

Object Creation

Creating a pandas-on-Spark Series by passing a list of values, letting pandas API on Spark create a default integer index:

[2]:
```
s = ps.Series([1, 3, 5, np.nan, 6, 8])
```

[3]:
```
s
```

[3]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a pandas-on-Spark DataFrame by passing a dict of objects that can be converted to series-like.

[4]:
```
psdf = 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])
```

[5]:
```
psdf
```

[5]:

	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

Creating a pandas DataFrame by passing a numpy array, with a datetime index and labeled columns:

[6]:
```
dates = pd.date_range('20130101', periods=6)
```

[7]:
```
dates
```

[7]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

[8]:
```
pdf = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
```

[9]:
```
pdf
```

[9]:

	A	B	C	D
2013-01-01	0.912558	-0.795645	-0.289115	0.187606
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021
2013-01-05	-0.772209	-1.228099	0.068901	0.896679
2013-01-06	1.485582	-0.709306	-0.202637	-0.248766

Now, this pandas DataFrame can be converted to a pandas-on-Spark DataFrame

[10]:
```
psdf = ps.from_pandas(pdf)
```

[11]:
```
type(psdf)
```

[11]:
```
pyspark.pandas.frame.DataFrame
```

It looks and behaves the same as a pandas DataFrame.

[12]:
```
psdf
```

[12]:

	A	B	C	D
2013-01-01	0.912558	-0.795645	-0.289115	0.187606
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021
2013-01-05	-0.772209	-1.228099	0.068901	0.896679
2013-01-06	1.485582	-0.709306	-0.202637	-0.248766

Also, it is possible to create a pandas-on-Spark DataFrame from Spark DataFrame easily.

Creating a Spark DataFrame from pandas DataFrame

[13]:
```
spark = SparkSession.builder.getOrCreate()
```

[14]:
```
sdf = spark.createDataFrame(pdf)
```

[15]:
```
sdf.show()
```

Creating pandas-on-Spark DataFrame from Spark DataFrame.

[16]:
```
psdf = sdf.pandas_api()
```

[17]:
```
psdf
```

[17]:

	A	B	C	D
0	0.912558	-0.795645	-0.289115	0.187606
1	-0.059703	-1.233897	0.316625	-1.226828
2	0.332871	-1.262010	-0.434844	-0.579920
3	0.924016	-1.022019	-0.405249	-1.036021
4	-0.772209	-1.228099	0.068901	0.896679
5	1.485582	-0.709306	-0.202637	-0.248766

Having specific [dtypes](#) . Types that are common to both Spark and pandas are currently supported.

[18]:
```
psdf.dtypes
```

[18]:
```
A    float64
B    float64
C    float64
D    float64
dtype: object
```

Here is how to show top rows from the frame below.

Note that the data in a Spark dataframe does not preserve the natural order by default. The natural order can be preserved by setting `compute.ordered_head` option but it causes a performance overhead with sorting internally.

[19]:
```
psdf.head()
```

[19]:

	A	B	C	D
0	0.912558	-0.795645	-0.289115	0.187606
1	-0.059703	-1.233897	0.316625	-1.226828
2	0.332871	-1.262010	-0.434844	-0.579920
3	0.924016	-1.022019	-0.405249	-1.036021
4	-0.772209	-1.228099	0.068901	0.896679

Displaying the index, columns, and the underlying numpy data.

[20]:
```
psdf.index
```

[20]:
```
Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

[21]:
```
psdf.columns
```

[21]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

[22]:
```
psdf.to_numpy()
```

[22]:
```
array([[ 0.91255803, -0.79564526, -0.28911463,  0.18760567],
       [-0.05970271, -1.23389695,  0.31662465, -1.2268284 ],
       [ 0.33287107, -1.26201008, -0.43484443, -0.57991997],
       [ 0.92401585, -1.0220191 , -0.40524889, -1.03602121],
       [-0.772209  , -1.22809864,  0.06890115,  0.89667907],
       [ 1.4855823 , -0.70930564, -0.20263668, -0.2487662 ]])
```

Showing a quick statistic summary of your data

[23]:
```
psdf.describe()
```

[23]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.470519	-1.041829	-0.157720	-0.334542
std	0.809428	0.241511	0.294520	0.793014
min	-0.772209	-1.262010	-0.434844	-1.226828
25%	-0.059703	-1.233897	-0.405249	-1.036021
50%	0.332871	-1.228099	-0.289115	-0.579920
75%	0.924016	-0.795645	0.068901	0.187606
max	1.485582	-0.709306	0.316625	0.896679

Transposing your data

[24]:
```
psdf.T
```

[24]:

	0	1	2	3	4	5
A	0.912558	-0.059703	0.332871	0.924016	-0.772209	1.485582
B	-0.795645	-1.233897	-1.262010	-1.022019	-1.228099	-0.709306
C	-0.289115	0.316625	-0.434844	-0.405249	0.068901	-0.202637
D	0.187606	-1.226828	-0.579920	-1.036021	0.896679	-0.248766

Sorting by its index

[25]:
```
psdf.sort_index(ascending=False)
```

[25]:

	A	B	C	D
5	1.485582	-0.709306	-0.202637	-0.248766
4	-0.772209	-1.228099	0.068901	0.896679
3	0.924016	-1.022019	-0.405249	-1.036021
2	0.332871	-1.262010	-0.434844	-0.579920
1	-0.059703	-1.233897	0.316625	-1.226828
0	0.912558	-0.795645	-0.289115	0.187606

Sorting by value

[26]:
```
psdf.sort_values(by='B')
```

[26]:

	A	B	C	D
2	0.332871	-1.262010	-0.434844	-0.579920
1	-0.059703	-1.233897	0.316625	-1.226828
4	-0.772209	-1.228099	0.068901	0.896679
3	0.924016	-1.022019	-0.405249	-1.036021
0	0.912558	-0.795645	-0.289115	0.187606
5	1.485582	-0.709306	-0.202637	-0.248766

Missing Data

Pandas API on Spark primarily uses the value `np.nan` to represent missing data. It is by default not included in computations.

[27]:
```
pdf1 = pdf.reindex(index=dates[0:4], columns=list(pdf.columns) + ['E'])
```

[28]:
```
pdf1.loc[dates[0]:dates[1], 'E'] = 1
```

[29]:
```
psdf1 = ps.from_pandas(pdf1)
```

[30]:
```
psdf1
```

[30]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920	NaN
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021	NaN

To drop any rows that have missing data.

[31]:
```
psdf1.dropna(how='any')
```

[31]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0

Filling missing data.

[32]:
```
psdf1.fillna(value=5)
```

[32]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920	5.0
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021	5.0

Operations

Stats

Performing a descriptive statistic:

[33]:
```
psdf.mean()
```

[33]:
```
A    0.470519
B   -1.041829
C   -0.157720
D   -0.334542
dtype: float64
```

Spark Configurations

Various configurations in PySpark could be applied internally in pandas API on Spark. For example, you can enable Arrow optimization to hugely speed up internal pandas conversion. See also PySpark Usage Guide for Pandas with Apache Arrow in PySpark documentation.

[34]:
```
prev = spark.conf.get("spark.sql.execution.arrow.pyspark.enabled")  # Keep its default value.
ps.set_option("compute.default_index_type", "distributed")  # Use default index prevent overhead.
import warnings
warnings.filterwarnings("ignore")  # Ignore warnings coming from Arrow optimizations.
```

[35]:
```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", True)
%timeit ps.range(300000).to_pandas()
```

[36]:
```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", False)
%timeit ps.range(300000).to_pandas()
```

[37]:
```
ps.reset_option("compute.default_index_type")
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", prev)  # Set its default value back.
```

Grouping

By "group by" we are referring to a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria
- Applying a function to each group independently
- Combining the results into a data structure

[38]:
```
psdf = ps.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar', 'foo', 'foo'],
                     'B': ['one', 'one', 'two', 'three',
                           'two', 'two', 'one', 'three'],
                     'C': np.random.randn(8),
                     'D': np.random.randn(8)})
```

[39]:
```
psdf
```

[39]:

	A	B	C	D
0	foo	one	1.039632	-0.571950
1	bar	one	0.972089	1.085353
2	foo	two	-1.931621	-2.579164
3	bar	three	-0.654371	-0.340704
4	foo	two	-0.157080	0.893736
5	bar	two	0.882795	0.024978
6	foo	one	-0.149384	0.201667
7	foo	three	-1.355136	0.693883

Grouping and then applying the sum() function to the resulting groups.

[40]:
```
psdf.groupby('A').sum()
```

[40]:

A	C	D
bar	1.200513	0.769627
foo	-2.553589	-1.361828

Grouping by multiple columns forms a hierarchical index, and again we can apply the sum function.

[41]:
```
psdf.groupby(['A', 'B']).sum()
```

[41]:

A	B	C	D
foo	one	0.890248	-0.370283
	two	-2.088701	-1.685428
bar	three	-0.654371	-0.340704
foo	three	-1.355136	0.693883
bar	two	0.882795	0.024978
	one	0.972089	1.085353

Plotting

[42]:
```
pser = pd.Series(np.random.randn(1000),
                 index=pd.date_range('1/1/2000', periods=1000))
```


[43]:
```
psser = ps.Series(pser)
```

[44]:
```
psser = psser.cummax()
```

[45]:
```
psser.plot()
```

On a DataFrame, the plot() method is a convenience to plot all of the columns with labels:

[46]:
```
pdf = pd.DataFrame(np.random.randn(1000, 4), index=pser.index,
                   columns=['A', 'B', 'C', 'D'])
```

[47]:
```
psdf = ps.from_pandas(pdf)
```

[48]:
```
psdf = psdf.cummax()
```

[49]:
```
psdf.plot()
```

For more details, Plotting documentation.

Getting data in/out

CSV

CSV is straightforward and easy to use. See here to write a CSV file and here to read a CSV file.

[50]:
```
psdf.to_csv('foo.csv')
ps.read_csv('foo.csv').head(10)
```

[50]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

Parquet

Parquet is an efficient and compact file format to read and write faster. See here to write a Parquet file and here to read a Parquet file.

[51]:
```
psdf.to_parquet('bar.parquet')
ps.read_parquet('bar.parquet').head(10)
```

[51]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

Spark IO

In addition, pandas API on Spark fully supports Spark's various datasources such as ORC and an external datasource. See here to write it to the specified datasource and here to read it from the datasource.

[52]:

```
psdf.to_spark_io('zoo.orc', format="orc")
ps.read_spark_io('zoo.orc', format="orc").head(10)
```

[52]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

See the Input/Output documentation for more details.

Previous
Quickstart: Spark Connect

Next
Testing PySpark