Quickstart: DataFrame

This is a short introduction and quickstart for the PySpark DataFrame API. PySpark DataFrames are lazily evaluated. They are implemented on top of RDDs. When Spark transforms data, it does not immediately compute the transformation but plans how to compute later. When actions such as `collect()` are explicitly called, the computation starts. This notebook shows the basic usages of the DataFrame, geared mainly for new users. You can run the latest version of these examples by yourself in 'Live Notebook: DataFrame' at the quickstart page.

There is also other useful information in Apache Spark documentation site, see the latest version of Spark SQL and DataFrames, RDD Programming Guide, Structured Streaming Programming Guide, Spark Streaming Programming Guide and Machine Learning Library (MLlib) Guide.

PySpark applications start with initializing `SparkSession` which is the entry point of PySpark as below. In case of running it in PySpark shell via pyspark executable, the shell automatically creates the session in the variable spark for users.

[1]:
```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

DataFrame Creation

A PySpark DataFrame can be created via `pyspark.sql.SparkSession.createDataFrame` typically by passing a list of lists, tuples, dictionaries and `pyspark.sql.Row`s, a pandas DataFrame and an RDD consisting of such a list. `pyspark.sql.SparkSession.createDataFrame` takes the `schema` argument to specify the schema of the DataFrame. When it is omitted, PySpark infers the corresponding schema by taking a sample from the data.

Firstly, you can create a PySpark DataFrame from a list of rows

[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]
```

Create a PySpark DataFrame with an explicit schema.

[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]
```

Create a PySpark DataFrame from a 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]
```

The DataFrames created above all have the same results and schema.

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

Viewing Data

The top rows of a DataFrame can be displayed using `DataFrame.show()`.

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

Alternatively, you can enable `spark.sql.repl.eagerEval.enabled` configuration for the eager evaluation of PySpark DataFrame in notebooks such as Jupyter. The number of rows to show can be controlled via `spark.sql.repl.eagerEval.maxNumRows` configuration.

[8]:
```
spark.conf.set('spark.sql.repl.eagerEval.enabled', True)
df
```

[8]:

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

The rows can also be shown vertically. This is useful when rows are too long to show horizontally.

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

You can see the DataFrame's schema and column names as follows:

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

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

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

Show the summary of the DataFrame

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

`DataFrame.collect()` collects the distributed data to the driver side as the local data in Python. Note that this can throw an out-of-memory error when the dataset is too large to fit in the driver side because it collects all the data from executors to the driver side.

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

In order to avoid throwing an out-of-memory exception, use `DataFrame.take()` or `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, 0))]
```

PySpark DataFrame also provides the conversion back to a pandas DataFrame to leverage pandas API. Note that `toPandas` also collects all data into the driver side that can easily cause an out-of-memory-error when the data is too large to fit into the driver side.

[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

Selecting and Accessing Data

PySpark DataFrame is lazily evaluated and simply selecting a column does not trigger the computation but it returns a `Column` instance.

[16]:
```
df.a
```

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

In fact, most of column-wise operations return `Column`s.

[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
```

These `Column`s can be used to select the columns from a DataFrame. For example, `DataFrame.select()` takes the `Column` instances that returns another DataFrame.

[18]:
```
df.select(df.c).show()
```

Assign new `Column` instance.

```
[19]: df.withColumn('upper_c', upper(df.c)).show()
```

To select a subset of rows, use `DataFrame.filter()`.

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

Applying a Function

PySpark supports various UDFs and APIs to allow users to execute Python native functions. See also the latest [Pandas UDFs] and [Pandas Function APIs]. For instance, the example below allows users to directly use the APIs in [a pandas Series] within Python native function.

```
[21]: import pandas as pd
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()
```

Another example is `DataFrame.mapInPandas` which allows users directly use the APIs in a [pandas DataFrame] without any restrictions such as the result length.

```
[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()
```

Grouping Data

PySpark DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition applies a function to each group and then combines them back to the

Search the docs ...

Installation
Quickstart: DataFrame
Quickstart: Spark Connect
Quickstart: Pandas API on Spark
Testing PySpark

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()
```

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

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

You can also apply a Python native function against each group by using pandas API.

[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()
```

Co-grouping and applying a function.

[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 merge_ordered(l, r):
    return pd.merge_ordered(l, r)

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

Getting Data In/Out

CSV is straightforward and easy to use. Parquet and ORC are efficient and compact file formats to read and write faster.

There are many other data sources available in PySpark such as JDBC, text, binaryFile, Avro, etc. See also the latest Spark SQL, DataFrames and Datasets Guide in Apache Spark documentation.

CSV

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

Parquet

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

ORC

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

Working with SQL

DataFrame and Spark SQL share the same execution engine so they can be interchangeably used seamlessly. For example, you can register the DataFrame as a table and run a SQL easily as below:

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

In addition, UDFs can be registered and invoked in SQL out of the box:

[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()
```

These SQL expressions can directly be mixed and used as PySpark columns.

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

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

Previous
Installation

Next
Quickstart: Spark Connect