From Pandas to Apache Spark’s DataFrame

August 12, 2015 | by Olivier Girardot

(httpshttps://twitter.com/linkedin.facebook.com

This is a cross-post from the blog of Olivier Girardot. Olivier is a software engineer and the co-founder of Lateral Thoughts, where he works on Machine Learning, Big Data, and DevOps solutions.

With the introduction in Spark 1.4 of Window operations, you can finally port pretty much any relevant piece of Pandas’ DataFrame computation to Apache Spark parallel computation framework using Spark SQL’s DataFrame. If you’re not yet familiar with Spark’s DataFrame (https://databricks.com/blog/2015/02/17 /introducing-dataframes-in-spark-for-large-scale-data-science.html), don’t hesitate to check out RDDs are the new bytecode of Apache Spark (https://ogirardot.wordpress.com/2015/05/29/rdds-are-the-new-bytecode-of-apache-spark/) and come back here after.

I figured some feedback on how to port existing *complex* code might be useful, so the goal of this article will be to take a few concepts from Pandas DataFrame and see how we can translate this to PySpark’s DataFrame using Spark 1.4.

Disclaimer: a few operations that you can do in Pandas don’t translate to Spark well. Please remember that DataFrames in Spark are like RDD in the sense that they’re an immutable data structure. Therefore things like:

```
# to create a new column "three"
df['three'] = df['one'] * df['two']
```

Can’t exist, just because this kind of affectation goes against the principles of Spark. Another example would be trying to access by index a single element within a DataFrame. Don’t forget that you’re using a distributed data structure, not an in-memory random-access data structure.

To be clear, this doesn’t mean that you can’t do the same kind of thing (i.e. create a new column) using Spark, it means that you have to think immutable/distributed and re-write parts of your code, mostly the parts that are not purely thought of as transformations on a stream of data.

So let's dive in.

Column selection

This part is not that much different in Pandas and Spark, but you have to take into account the immutable character of your DataFrame. First let's create two DataFrames one in Pandas *pdf* and one in Spark *df*:

```

# Pandas => pdf
In [17]: pdf = pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5

In [18]: pdf.A
Out[18]:
0    1
1    2
2    3
Name: A, dtype: int64

# SPARK SQL => df
In [19]: df = sqlCtx.createDataFrame([(1, 4), (2, 5), (3, 6)], ["A",

In [20]: df
Out[20]: DataFrame[A: bigint, B: bigint]

In [21]: df.show()
+-+-+
|A|B|
+-+-+
|1|4|
|2|5|
|3|6|
+-+-+
```

Now in Spark SQL or Pandas you use the same syntax to refer to a column:

```

In [27]: df.A
Out[27]: Column<A>
Out[27]: Column<A>

In [28]: df['A']
Out[28]: Column<A>

In [29]: pdf.A
Out[29]:
0    1
1    2
2    3
Name: A, dtype: int64

In [30]: pdf['A']
Out[30]:
0    1
1    2
2    3
Name: A, dtype: int64
```

The output seems different, but these are still the same ways of referencing a column using Pandas or Spark. The only difference is that in Pandas, it is a mutable data structure that you can change – not in Spark.

Column adding

```
In [31]: pdf['C'] = 0

In [32]: pdf
Out[32]:
   A  B  C
0  1  4  0
1  2  5  0
2  3  6  0

# In Spark SQL you'll use the withColumn or the select method,
# but you need to create a "Column", a simple int won't do :
In [33]: df.withColumn('C', 0)
-------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
<ipython-input-33-fd1261f623cf> in <module>()
----> 1 df.withColumn('C', 0)

/Users/ogirardot/Downloads/spark-1.4.0-bin-hadoop2.4/python/pyspark/s
   1196         """
-> 1197         return self.select('*', col.alias(colName))
   1198
   1199     @ignore_unicode_prefix

AttributeError: 'int' object has no attribute 'alias'

# Here's your new best friend "pyspark.sql.functions.*"
# If you can't create it from composing columns
# this package contains all the functions you'll need :
In [35]: from pyspark.sql import functions as F
In [36]: df.withColumn('C', F.lit(0))
Out[36]: DataFrame[A: bigint, B: bigint, C: int]

In [37]: df.withColumn('C', F.lit(0)).show()
+-+-+-+
|A|B|C|
+-+-+-+
|1|4|0|
|2|5|0|
|3|6|0|
+-+-+-+
```

Most of the time in Spark SQL you can use Strings to reference columns but there are two cases where you'll want to use the Column objects rather than Strings :

- In Spark SQL DataFrame columns are allowed to have the same name, they'll be given unique names inside of Spark SQL, but this means that you can't reference them with the column name only as this becomes ambiguous.
- When you need to manipulate columns using expressions like *Adding two columns to each other*, *Twice the value of this column* or even *Is the column value larger than 0 ?*, you won't be able to use simple strings and will need the Column reference.

- Finally if you need renaming, cast or any other complex feature, you'll need the Column reference too.

Here's an example:

```
In [39]: df.withColumn('C', df.A * 2)
Out[39]: DataFrame[A: bigint, B: bigint, C: bigint]

In [40]: df.withColumn('C', df.A * 2).show()
+-+-+-+
|A|B|C|
+-+-+-+
|1|4|2|
|2|5|4|
|3|6|6|
+-+-+-+

In [41]: df.withColumn('C', df.B > 0).show()
+-+-+----+
|A|B|   C|
+-+-+----+
|1|4|true|
|2|5|true|
|3|6|true|
+-+-+----+
```

When you're selecting columns, to create another *projected* DataFrame, you can also use expressions:

```
In [42]: df.select(df.B > 0)
Out[42]: DataFrame[(B > 0): boolean]

In [43]: df.select(df.B > 0).show()
+--------+
|(B > 0)|
+--------+
|    true|
|    true|
|    true|
+--------+
```

s you can see the column name will actually be computed according to the expression you defined, if you want to rename this, you'll need to use the **alias** method on Column:

```
In [44]: df.select((df.B > 0).alias("is_positive")).show()
+-----------+
|is_positive|
+-----------+
|       true|
|       true|
|       true|
+-----------+
```

All of the expressions that we're building here can be used for Filtering, Adding a new column or even inside Aggregations, so once you get a general idea of how it works, you'll be fluent throughout all of the DataFrame manipulation framework.

Filtering

Filtering is pretty much straightforward too, you can use the *RDD-like* `filter` method and copy any of your existing Pandas expression/predicate for filtering:

```
In [48]: pdf[(pdf.B > 0) & (pdf.A < 2)] Out[48]:    A   B   C 0   1   4   
+-+-+
|A|B|
+-+-+
|1|4|
+-+-+
```

Aggregations

What can be confusing at first in using aggregations is that the minute you write `groupBy` you're not using a DataFrame object, you're actually using a `GroupedData` object and you need to precise your aggregations to get back the output DataFrame:

```

In [77]: df.groupBy("A")
Out[77]: <pyspark.sql.group.GroupedData at 0x10dd11d90>

In [78]: df.groupBy("A").avg("B")
Out[78]: DataFrame[A: bigint, AVG(B): double]

In [79]: df.groupBy("A").avg("B").show()
+-+------+
|A|AVG(B)|
+-+------+
|1|   4.0|
|2|   5.0|
|3|   6.0|
+-+------+
```

As a syntactic sugar if you need only one aggregation, you can use the simplest functions like: *avg, cout, max, min, mean* and *sum* directly on `GroupedData`, but most of the time, this will be too simple and you'll want to create a few aggregations during a single groupBy operation. After all (c.f. RDDs are the new bytecode of Apache Spark (https://ogirardot.wordpress.com/2015/05/29/rdds-are-the-new-bytecode-of-apache-spark/)) this is one of the greatest features of the DataFrames. To do so you'll be using the `agg` method:

```

In [83]: df.groupBy("A").agg(F.avg("B"), F.min("B"), F.max("B")).show(
+-+------+------+------+
|A|AVG(B)|MIN(B)|MAX(B)|
+-+------+------+------+
|1|   4.0|     4|     4|
|2|   5.0|     5|     5|
|3|   6.0|     6|     6|
+-+------+------+------+
```

Of course, just like before, you can use any expression especially column compositions, alias definitions etc… and some other non-trivial functions:

```

In [84]: df.groupBy("A").agg(
   ....: F.first("B").alias("my first"),
   ....: F.last("B").alias("my last"),
   ....: F.sum("B").alias("my everything")
   ....: ).show()
+-+--------+-------+-------------+
|A|my first|my last|my everything|
+-+--------+-------+-------------+
|1|       4|      4|            4|
|2|       5|      5|            5|
|3|       6|      6|            6|
+-+--------+-------+-------------+
```

Complex operations & Windows

Now that Spark 1.4 is out, the Dataframe API provides an efficient and easy to use Window-based framework – this single feature is what makes any Pandas to Spark migration actually do-able for 99% of the projects – even considering some of Pandas' features that seemed *hard* to reproduce in a distributed environment.

A simple example that we can pick is that in Pandas you can compute a diff on a column and Pandas will compare the values of one line to the last one and compute the difference between them. Typically the kind of feature hard to do in a distributed environment because each line is supposed to be treated independently, now with Spark 1.4 window operations you can define a window on which Spark will *execute some aggregation functions* but relatively to a specific line. Here's how to port some existing Pandas code using diff:

```

In [86]: df = sqlCtx.createDataFrame([(1, 4), (1, 5), (2, 6), (2, 6),

In [95]: pdf = df.toPandas()

In [96]: pdf
Out[96]:
   A  B
0  1  4
1  1  5
2  2  6
3  2  6
4  3  0

In [98]: pdf['diff'] = pdf.B.diff()

In [102]: pdf
Out[102]:
   A  B  diff
0  1  4   NaN
1  1  5     1
2  2  6     1
3  2  6     0
4  3  0    -6
```

In Pandas you can compute a diff on an arbitrary column, with no regard for keys, no regards for order or anything. It's cool… but most of the time not exactly what you want and you might end up cleaning up the mess afterwards by setting the column value back to NaN from one line to another when the keys changed.

Here's how you can do such a thing in PySpark using Window functions, a Key and, if you want, in a specific order:

```
In [107]: from pyspark.sql.window import Window

In [108]: window_over_A = Window.partitionBy("A").orderBy("B")

In [109]: df.withColumn("diff", F.lead("B").over(window_over_A) - df.
+---+---+----+
|  A|  B|diff|
+---+---+----+
|  1|  4|   1|
|  1|  5|null|
|  2|  6|   0|
|  2|  6|null|
|  3|  0|null|
+---+---+----+
```

With that you are now able to compute a diff line by line – ordered or not – given a specific key. The great point about Window operation is that you're not actually breaking the structure of your data. Let me explain myself.

When you're computing some kind of aggregation (once again according to a key), you'll usually be executing a `groupBy` operation given this key and compute the multiple metrics that you'll need (*at the same time* if you're lucky, otherwise in multiple `reduceByKey` or `aggregateByKey` transformations).

But whether you're using RDDs or DataFrame, if you're not using window operations then you'll actually crush your data in a part of your flow and then you'll need to join back again the results of your aggregations to the *main* – dataflow. Window operations allow you to execute your computation and copy the results as additional columns without any explicit join.

This is a quick way to enrich your data adding rolling computations as just another column directly. Two additional resources are worth noting regarding these new features, the official Databricks blog article on Window operations (https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html) and Christophe Bourguignat (http://twitter.com/chris_bour)'s article evaluating Pandas and Spark DataFrame differences (https://medium.com/@chris_bour/6-differences-between-pandas-and-spark-dataframes-1380cec394d2).

To sum up, you now have all the tools you need in Spark 1.4 to port any Pandas computation in a distributed environment using the *very* similar DataFrame API.