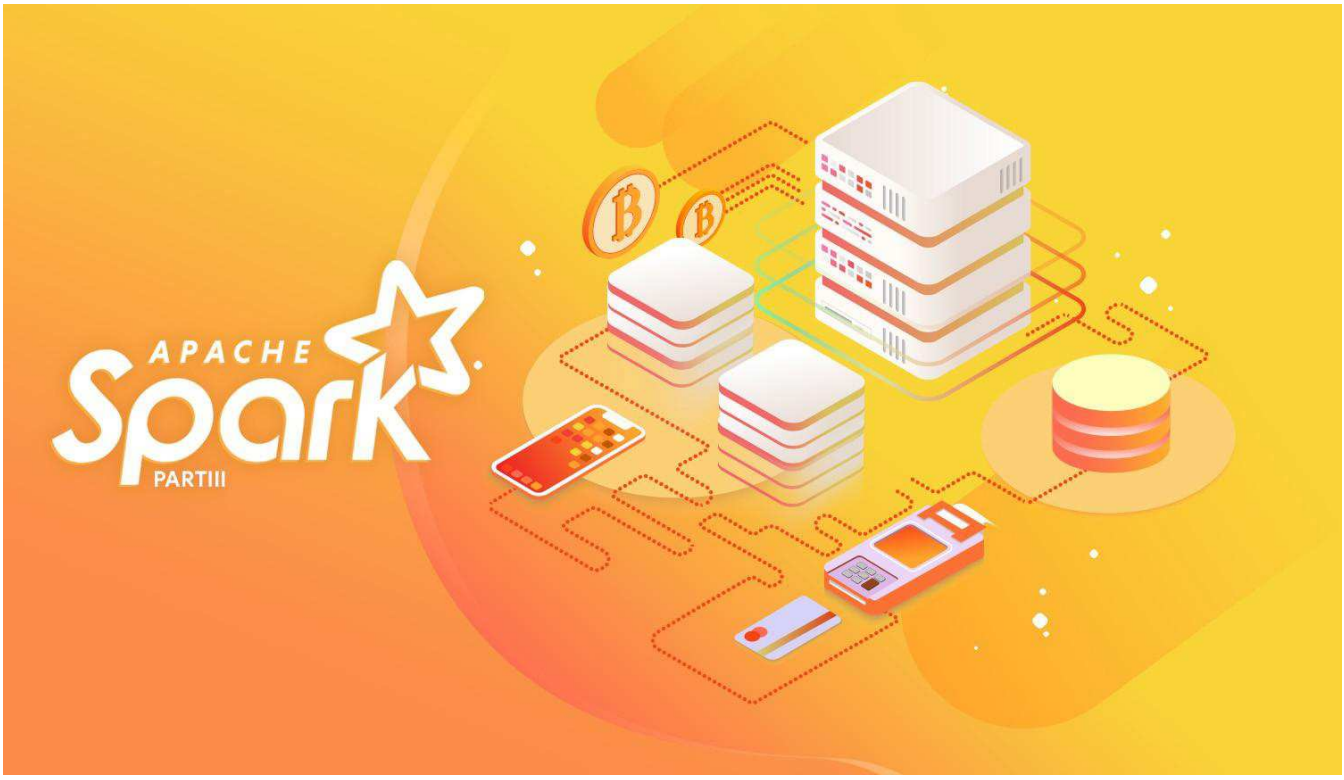
DataFrame Transformations in PySpark (Continued)

Continuing to apply transformations to Spark DataFrames using PySpark.

Todd Birchard
May 7 · 6 min read ★



We've covered a fair amount of ground when it comes to Spark DataFrame transformations in this series. In part 1, we touched on `filter()`, `select()`, `dropna()`, `fillna()`, and `isNull()`. Then, we moved on to `dropDuplicates` and user-defined functions (`udf`) in part 2. This time around, we'll be building on these concepts and introduce some new ways to transform data so you can officially be awarded your **PySpark Guru Certification**, award by us here at Hackers & Slackers.*

Of course, we need to get things started with some sample data. I'm going to use the **2016 MLB postseason** dataset we worked with last time. For a quick refresher, here's a preview:

awayTeamName	homeTeamName	homeFinalRuns	awayFinalRuns	durationMinutes
Red Sox	Indians	6	0	199
Nationals	Dodgers	3	8	252
Orioles	Blue Jays	5	2	205
Dodgers	Nationals	3	4	272
Cubs	Dodgers	2	10	238
Indians	Red Sox	3	4	221
Red Sox	Indians	5	4	213
Nationals	Dodgers	6	5	224
Blue Jays	Rangers	3	5	210
Rangers	Blue Jays	7	6	201
Giants	Cubs	1	0	150
Dodgers	Cubs	0	1	165
Cubs	Dodgers	6	0	198
Dodgers	Cubs	5	0	156
Giants	Mets	0	3	191
Cubs	Giants	6	5	303
Dodgers	Cubs	8	4	217
Blue Jays	Indians	2	1	164
Indians	Blue Jays	5	1	181
Cubs	Giants	5	6	205

Blue Jays	Indians	2	0	164
Cubs	Dodgers	4	8	256
Dodgers	Nationals	3	4	226

Adding and Modifying Columns

As you already know, we can create new columns by calling `withColumn()` operation on a DataFrame, while passing the name of the new column (the first argument), as well as an operation for which values should live in each row of that column (second argument).

It's lit() Fam

It's hard to mention columns without talking about PySpark's `lit()` function. `lit()` is simply one of those unsexy but critically important parts of PySpark that we need to understand, simply because PySpark a Python API which interacts with a Java JVM (as you might be painfully aware).

`lit()` is a way for us to interact with column literals in PySpark: Java expects us to explicitly mention when we're trying to work with a column object. Because Python has no native way of doing, we must instead use `lit()` to tell the JVM that what we're talking about is a *column literal*.

To import `lit()`, we need to import `functions` from `pyspark.sql`:

```
from pyspark.sql.functions import lit, when, col, regexp_extract
```

I've imported a few other things here which we'll get to later. With these imported, we can add new columns to a DataFrame the quick and dirty way:

```
from pyspark.sql.functions import lit, when, col, regexp_extract

df = df_with_winner.withColumn('testColumn', F.lit('this is a
test'))

display(df)
```

string: *this is a test*.

> ***PROTIP!:*** `lit()` is necessary when creating columns with values directly. If we use another function like `concat()`, there is no need to use `lit()` as it is implied that we're working with columns.

Creating Columns Based on Criteria

Another function we imported with `functions` is the `where` function. PySpark's `when()` functions kind of like SQL's `WHERE` clause (remember, we've imported this the from `pyspark.sql` package). In this case, we can use `when()` to create a column *when* the outcome of a conditional is true.

The first parameter we pass into `when()` is the conditional (or multiple conditionals, if you want). I'm not a huge fan of this syntax, but here's the format of this looks:

```
df = df.withColumn([COLUMN_NAME]. F.when([CONDITIONAL],
[COLUMN_VALUE]))
```

Let's use our baseball example to see the `when()` function in action. Remember last time when we added a "winner" column to out DataFrame? Well, we can do this using `when()` instead!

```
df = df_with_test_column.withColumn('game_winner', when(
(col("homeFinalRuns") > col("awayFinalRuns")), col("homeFinalRuns")
))
```

Let's walk through this step by step, shall we? We set the name of our column to be `game_winner` off the bat. Next is the most important part: the conditional. In our example, we have `(col("homeFinalRuns") > col("awayFinalRuns"))`. Remember how we said that The JVM we're interacting absolutely must know which data type we're talking about at all times? `col()` means we're comparing the values of two columns; or more specifically, we're comparing the values in *every row* in these columns.

Also, notice that we don't specify something like `df.col([COLUMN_NAME])`, or even `df.[COLUMN_NAME]`. We instead pass a string containing the name of our columns to `col()`,

DataFrame being acted on. After all, why wouldn't they? See... PySpark isn't annoying *all* the time - it's just *inconsistently* annoying (which may be even more annoying to the aspiring Sparker, admittedly).

But wait, something's missing! We've only added the winner when the winner happens to be the home team! How do we add the away team? Is there some sort of `else` equivalent to `when()`? Why yes, I'm so glad you've asked! It happens to be called `otherwise()`.

With `otherwise()`, we can tack on an action to take when conditional in our `when()` statement returns **False**!

```
df = df_with_test_column.withColumn('gameWinner', when(
(col("homeFinalRuns") > col("awayFinalRuns")), col("homeFinalRuns")
).otherwise(lit('awayTeamName')))

display(df)
```

Excellent, dear boy and/or girl! Let's get a look at what we've done so far:

So far so good, but what about getting rid of that annoying **testColumn?**

Dropping Entire Columns

Dropping columns is easy! we can simply use the `drop()` method on our DataFrame, and pass the name of the column:

```
df = df_with_test_column.drop('testColumn')
```

That does the trick!

String Operations & Filters

We've looked at how `filter()` works pretty extensively. Why don't we pair this with some common string operations to see what we can filter by?

like() and related operators

The PySpark `like()` method works exactly like the SQL equivalent: `%` denotes a wild card which means "any character or number of characters". Take a look at how we'd use `like()` to find winning teams whose name start with "Nat":

```
df = df.filter(df.winner.like('Nat%'))

display(df)
```

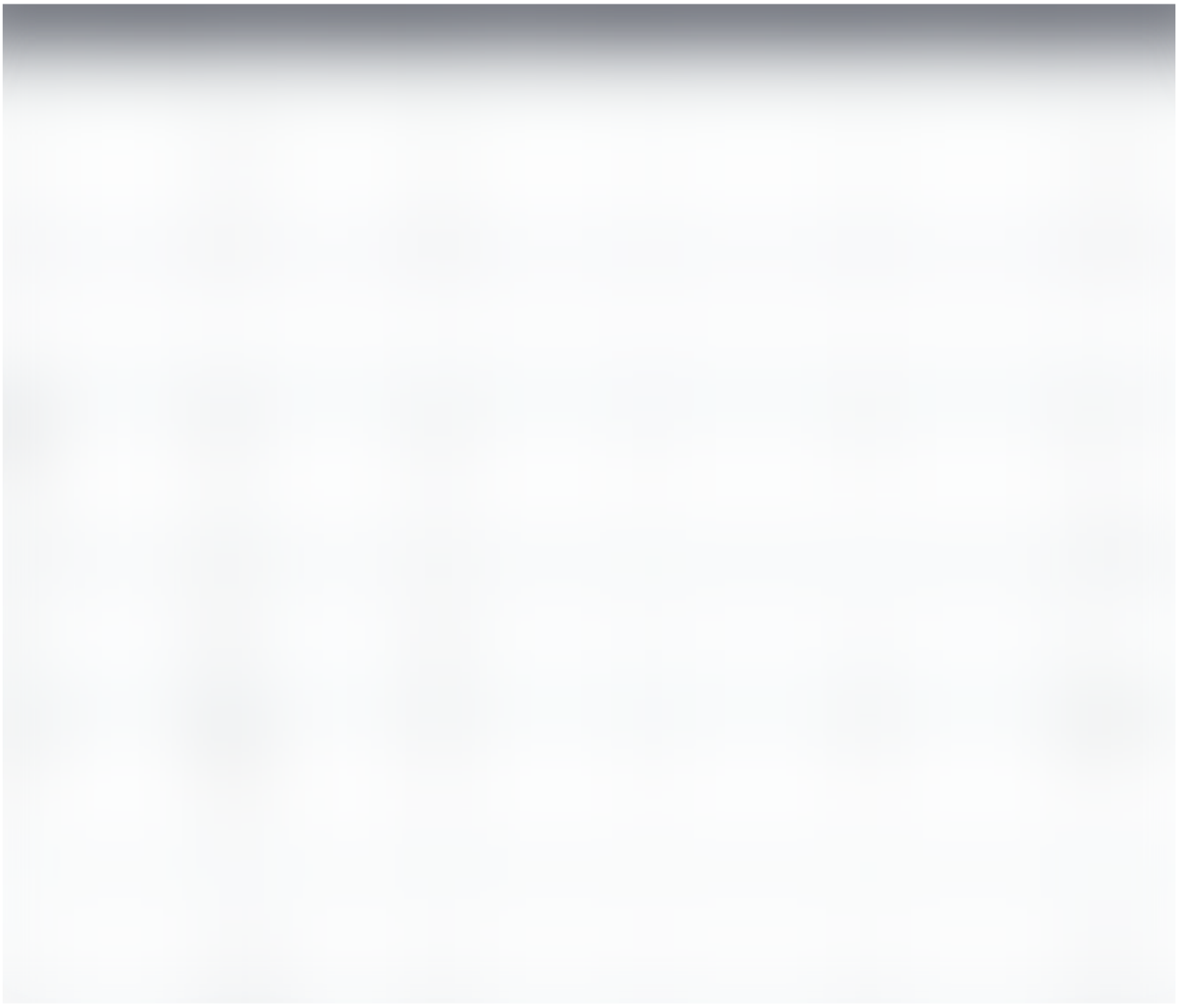
Wow, the Nationals only won two games! Hah, they sure do suck. Sorry about Harper, by the way.

cup of tea, you can use `contains()`, `startswith()`, and `endsWith()` instead. They all do what you'd imagine.

isIn() to Match Multiple Values

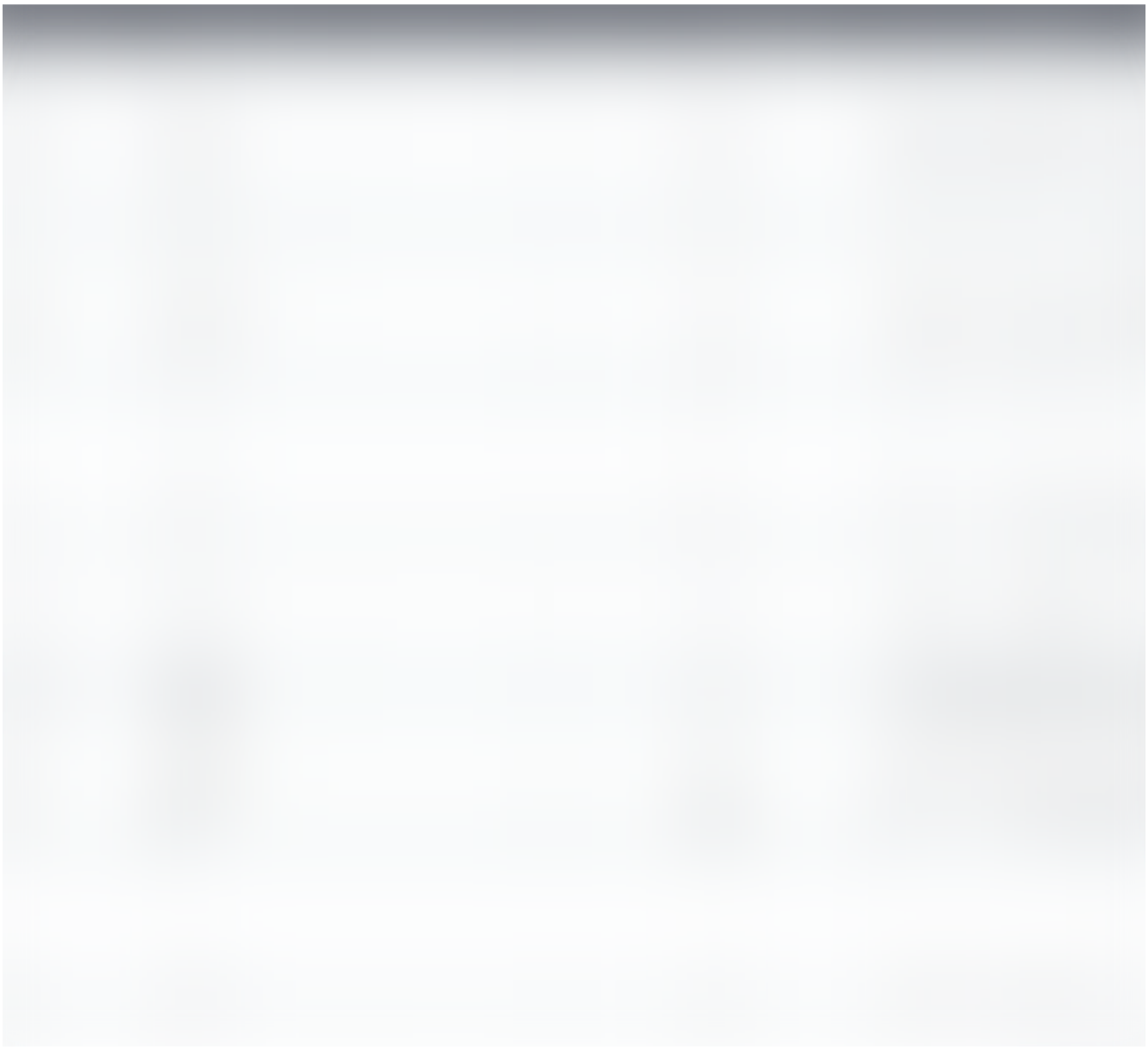
If we want to match by multiple values, `isIn()` is pretty great. This takes multiple values as it's parameters, and will return all rows where the columns of column X match any of *n* values:

```
df = df.filter(df.gameWinner.isin('Cubs', 'Indians'))

display(df)
```



concat() For Appending Strings

```
df = df.withColumn('gameTitle', concat(df.homeTeamName, lit(' vs. '), df.awayTeamName) )

display(df)
```



.regexp_extract() for Regex Functions

For real programmers who know what they're actually doing, you can disregard all those other lame string operations: `regexp_extract()` is the moment you've been waiting for. This takes one parameter: your regex. I could *totally* demonstrate my 1337 regex skills right here, but uh, I just don't feel like it right now:

Number Operators

You should be getting the point by now! Here are some number operators, which you can use just like the string operators (speaking syntactically, which is apparently a word):

- `round` : Rounds to the nearest scale. Accepts a column as the first parameter, and a scale as the second.
- `floor` : Returns the lowest number in a set. Accepts a column name as the first parameter.
- `ceil` : Returns the highest number in a set. Accepts a column name as the first parameter.

That'll Do, Pig (or Equivalent Spirit Animal)

Once again, I've dumped an ungodly amount of documentation veiled as a tutorial in our lap. Thank you for putting up with my nonsense by continuing to click on links to these stupid posts.

There's still more to do! Join us next time when we walk through the **extracting** and **loading** of data in PySpark while we give this transformation stuff a rest. Peace fam.

• • •

Originally published at https://hackersandslackers.com on May 7, 2019.

Data Engineering Python Apache Spark Apache Big Data