Clearning Data with PySpark (Dataframe Details)

Defining a schema

Creating a defined schema helps with data quality and import performance. As mentioned during the lesson, we'll create a simple schema to read in the following columns:

- Name
- Age
- City

The `Name` and `City` columns are `StringType()` and the `Age` column is an `IntegerType()` .

```
# Import the pyspark.sql.types library
from pyspark.sql.types import *

# Define a new schema using the StructType method
people_schema = StructType([
  # Define a StructField for each field
  StructField('name', StringType(), False),
  StructField('age', IntegerType(), False),
  StructField('city', StringType(), False)
])
```

Use lazy processing

Lazy processing operations will usually return in about the same amount of time regardless of the actual quantity of data. Remember that this is due to Spark not performing any transformations until an action is requested.

For this exercise, we'll be defining a Data Frame (`aa_dfw_df`) and add a couple transformations. Note the amount of time required for the transformations to complete when defined vs when the data is actually queried. These differences may be short, but they will be noticeable. When working with a full Spark cluster with larger quantities of data the difference will be more apparent.

```
# Load the CSV file
aa_dfw_df = spark.read.format('csv').options(Header=True).load('AA_DFW_2018.csv.gz')

# Add the airport column using the F.lower() method
aa_dfw_df = aa_dfw_df.withColumn('airport', F.lower(aa_dfw_df['Destination Airport']))

# Drop the Destination Airport column
aa_dfw_df = aa_dfw_df.drop(aa_dfw_df['Destination Airport'])

# Show the DataFrame
aa_dfw_df.show()
```

Saving a DataZFrame in Parquet format

When working with Spark, you'll often start with CSV, JSON, or other data sources. This provides a lot of flexibility for the types of data to load, but it is not an optimal format for Spark. The `Parquet` format is a columnar data store, allowing Spark to use *predicate pushdown*. This means Spark will only process the data necessary to complete the operations you define versus reading the entire dataset. This gives Spark more flexibility in accessing the data and often drastically improves performance on large datasets.

In this exercise, we're going to practice creating a new Parquet file and then process some data from it.

The `spark` object and the `df1` and `df2` DataFrames have been setup for you.

```
# View the row count of df1 and df2
print("df1 Count: %d" % df1.count())
print("df2 Count: %d" % df2.count())

# Combine the DataFrames into one
df3 = df1.union(df2)

# Save the df3 DataFrame in Parquet format
df3.write.parquet('AA_DFW_ALL.parquet', mode='overwrite')

# Read the Parquet file into a new DataFrame and run a count
print(spark.read.parquet('AA_DFW_ALL.parquet').count())
```

SQL and Parquest

Parquet files are perfect as a backing data store for SQL queries in Spark. While it is possible to run the same queries directly via Spark's Python functions, sometimes it's easier to run SQL queries alongside the Python options.

For this example, we're going to read in the Parquet file we created in the last exercise and register it as a SQL table. Once registered, we'll run a quick query against the table (aka, the Parquet file).

The `spark` object and the `AA_DFW_ALL.parquet` file are available for you automatically.

```
# Read the Parquet file into flights_df
flights_df = spark.read.parquet('AA_DFW_ALL.parquet')

# Register the temp table
flights_df.createOrReplaceTempView('flights')

# Run a SQL query of the average flight duration
avg_duration = spark.sql('SELECT avg(flight_duration) from flights').collect()[0]
print('The average flight time is: %d' % avg_duration)
```

Dog Parsing

You've done a considerable amount of cleanup on the initial dataset, but now need to analyze the data a bit deeper. There are several questions that have now come up about the type of dogs seen in an image and some details regarding the images. You realize that to answer these questions, you need to process the data into a specific type. Before you can use it, you'll need to create a schema / type to represent the dog details.

The `joined_df` DataFrame is as you last defined it, and the `pyspark.sql.types` have all been imported.

```
# Select the dog details and show 10 untruncated rows
print(joined_df.select('dog_list').show(truncate=False))

# Define a schema type for the details in the dog list
DogType = StructType([
    StructField("breed", StringType(), False),
    StructField("start_x", IntegerType(), False),
    StructField("start_y", IntegerType(), False),
    StructField("end_x", IntegerType(), False),
    StructField("end_y", IntegerType(), False)
])
```

Per image count

Your next task in building a data pipeline for this dataset is to create a few analysis oriented columns. You've been asked to calculate the number of dogs found in each image based on your `dog_list` column created earlier. You have also created the `DogType` which will allow better

parsing of the data within some of the data columns.

The `joined_df` is available as you last defined it, and the `DogType` structtype is defined. `pyspark.sql.functions` is available under the `F` alias.

```
# Create a function to return the number and type of dogs as a tuple
def dogParse(doglist):
  dogs = []
  for dog in doglist:
    (breed, start_x, start_y, end_x, end_y) = dog.split(',')
    dogs.append((breed, int(start_x), int(start_y), int(end_x), int(end_y)))
  return dogs

# Create a UDF
udfDogParse = F.udf(dogParse, ArrayType(DogType))

# Use the UDF to list of dogs and drop the old column
joined_df = joined_df.withColumn('dogs', udfDogParse('dog_list')).drop('dog_list')

# Show the number of dogs in the first 10 rows
joined_df.select(F.size('dogs')).show(10)
```

Percentage dog pixels

The final task for parsing the dog annotation data is to determine the percentage of pixels in each image that represents a dog (or dogs). You'll need to use the various techniques you've learned in this course to help calculate this information and add it as columns for later analysis.

To calculate the percentage of pixels, first calculate the total number of pixels representing each dog then sum them for the image. You can calculate the bounding box with the formula:

(Xend - Xstart) * (Yend - Ystart)

NOTE: You can ignore the possibility of overlapping bounding boxes in this instance.

For the percentage, calculate the total number of "dog" pixels divided by the total size of the image, multiplied by 100.
The `joined_df` DataFrame is as you last used it. `pyspark.sql.functions` is aliased to `F` .

```
# Define a UDF to determine the number of pixels per image
def dogPixelCount(doglist):
  totalpixels = 0
  for dog in doglist:
    totalpixels += (dog[3] - dog[1])*(dog[4] - dog[2])
  return totalpixels

```

```
# Define a UDF for the pixel count
udfDogPixelCount = F.udf(dogPixelCount, IntegerType())
joined_df = joined_df.withColumn('dog_pixels', udfDogPixelCount('dogs'))

# Create a column representing the percentage of pixels
joined_df = joined_df.withColumn('dog_percent', (joined_df.dog_pixels / (joined_df.width * joined_df.height)) * 100)

# Show the first 10 annotations with more than 60% dog
joined_df.filter(joined_df.dog_percent > 60).show(10)
```