Cleaning Data with PySpark (Improving Performance)

Caching a DataFrame

You've been assigned a task that requires running several analysis operations on a DataFrame. You've learned that caching can improve performance when reusing DataFrames and would like to implement it.

You'll be working with a new dataset consisting of airline departure information. It may have repetitive data and will need to be de-duplicated.

The DataFrame `departures_df` is defined, but no actions have been performed.

```
start_time = time.time()

# Add caching to the unique rows in departures_df
departures_df = departures_df.distinct().cache()

# Count the unique rows in departures_df, noting how long the operation takes
print("Counting %d rows took %f seconds" % (departures_df.count(), time.time() - start_time))

# Count the rows again, noting the variance in time of a cached DataFrame
start_time = time.time()
print("Counting %d rows again took %f seconds" % (departures_df.count(), time.time() - start_time))
```

Removeing a DataFrame from cache

You've finished the analysis tasks with the departures_df DataFrame, but have some other processing to do. You'd like to remove the DataFrame from the cache to prevent any excess memory usage on your cluster.

The DataFrame `departures_df` is defined and has already been cached for you.

```
# Determine if departures_df is in the cache
print("Is departures_df cached?: %s" % departures_df.is_cached)
```

```python
print("Removing departures_df from cache")

# Remove departures_df from the cache
departures_df.unpersist()

# Check the cache status again
print("Is departures_df cached?: %s" % departures_df.is_cached)
```

File import performance

You've been given a large set of data to import into a Spark DataFrame. You'd like to test the difference in import speed by splitting up the file.

You have two types of files available: `departures_full.txt.gz` and `departures_xxx.txt.gz` where **xxx** is **000** - **013**. The same number of rows is split between each file.

```python
# Import the full and split files into DataFrames
full_df = spark.read.csv('departures_full.txt.gz')
split_df = spark.read.csv('departures_000.txt.gz')

# Print the count and run time for each DataFrame
start_time_a = time.time()
print("Total rows in full DataFrame:\t%d" % full_df.count())
print("Time to run: %f" % (time.time() - start_time_a))

start_time_b = time.time()
print("Total rows in split DataFrame:\t%d" % split_df.count())
print("Time to run: %f" % (time.time() - start_time_b))
```

Reading Spark configurations

You've recently configured a cluster via a cloud provider. Your only access is via the command shell or your python code. You'd like to verify some Spark settings to validate the configuration of the cluster.

The `spark` object is available for use.

```python
# Name of the Spark application instance
app_name = spark.conf.get('spark.app.name')

# Driver TCP port
driver_tcp_port = spark.conf.get('spark.driver.port')

# Number of join partitions
num_partitions = spark.conf.get('spark.sql.shuffle.partitions')
```

```python

# Show the results
print("Name: %s" % app_name)
print("Driver TCP port: %s" % driver_tcp_port)
print("Number of partitions: %s" % num_partitions)
```

Writing Spark configurations

Now that you've reviewed some of the Spark configurations on your cluster, you want to modify some of the settings to tune Spark to your needs. You'll import some data to review that your changes have affected the cluster.

The spark configuration is initially set to the default value of 200 partitions.

The `spark` object is available for use. A file named `departures.txt.gz` is available for import. An initial DataFrame containing the distinct rows from `departures.txt.gz` is available as `departures_df` .

```python
# Store the number of partitions in variable
before = departures_df.rdd.getNumPartitions()

# Configure Spark to use 500 partitions
spark.conf.set('spark.sql.shuffle.partitions', 500)

# Recreate the DataFrame using the departures data file
departures_df = spark.read.csv('departures.txt.gz').distinct()

# Print the number of partitions for each instance
print("Partition count before change: %d" % before)
print("Partition count after change: %d" % departures_df.rdd.getNumPartitions())
```

Normal joins

You've been given two DataFrames to combine into a single useful DataFrame. Your first task is to combine the DataFrames normally and view the execution plan.

The DataFrames `flights_df` and `airports_df` are available to you.

```python
# Join the flights_df and aiports_df DataFrames
normal_df = flights_df.join(airports_df, \
    flights_df["Destination Airport"] == airports_df["IATA"] )

# Show the query plan
normal_df.explain()
```

Using broadcasting on Spark joins

Remember that table joins in Spark are split between the cluster workers. If the data is not local, various shuffle operations are required and can have a negative impact on performance. Instead, we're going to use Spark's `broadcast` operations to give **each** node a copy of the specified data.

A couple tips:

- Broadcast the smaller DataFrame. The larger the DataFrame, the more time required to transfer to the worker nodes.
- On small DataFrames, it may be better skip broadcasting and let Spark figure out any optimization on its own.
- If you look at the query execution plan, a broadcastHashJoin indicates you've successfully configured broadcasting.

The DataFrames `flights_df` and `airports_df` are available to you.

```
# Import the broadcast method from pyspark.sql.functions
from pyspark.sql.functions import broadcast

# Join the flights_df and airports_df DataFrames using broadcasting
broadcast_df = flights_df.join(broadcast(airports_df), \
    flights_df["Destination Airport"] == airports_df["IATA"] )

# Show the query plan and compare against the original
broadcast_df.explain()
```

Comparing broadcast vs normal joins

You've created two types of joins, normal and broadcasted. Now your manager would like to know what the performance improvement is by using Spark optimizations. If the results are promising, you'll be given more opportunity to tweak the Spark setup as needed.

Your DataFrames `normal_df` and `broadcast_df` are available for your use.

```
start_time = time.time()
# Count the number of rows in the normal DataFrame
normal_count = normal_df.count()
normal_duration = time.time() - start_time

start_time = time.time()
# Count the number of rows in the broadcast DataFrame
```

```python
broadcast_count = broadcast_df.count()
broadcast_duration = time.time() - start_time

# Print the counts and the duration of the tests
print("Normal count:\t\t%d\tduration: %f" % (normal_count, normal_duration))
print("Broadcast count:\t%d\tduration: %f" % (broadcast_count, broadcast_duration))
```