Putting it all together

Lambdas in Spark Operations

Lambdas expressions allow us to apply a simple operation to an object without needing to define it as a function. This improves readability by condensing what could be a few lines of code into a single line. Utilizing lambdas in Spark operations allows us to apply any arbitrary function to all RDD elements specified by the transformation or action.

```
# start a new SparkSession
from pyspark.sql import SparkSession
spark =
SparkSession.builder.getOrCreate()

# create an RDD
rdd =
spark.sparkContext.parallelize([1,2,3,4,
5])

# transform rdd
transformed_rdd = rdd.map(lambda x: x*2)
# multiply each RDD element by 2

# view the transformed RDD
transformed_rdd.collect()
# output:
# [2,4,6,8,10]
```

codecademy

Viewing RDDs

Two common functions used to view RDDs are:

1. `.collect()`, which pulls the entire RDD into memory. This method will probably max out our memory if the RDD is big.
2. `.take(n)`, which will only pull in the first n elements of the RDD into memory.

```
# start a new SparkSession
from pyspark.sql import SparkSession
spark =
SparkSession.builder.getOrCreate()

# create an RDD
rdd =
spark.sparkContext.parallelize([1,2,3,4,
5])

# we can run collect() on a small RDD
rdd.collect()
# output: [1, 2, 3, 4, 5]

rdd.take(2)
# output: [1, 2]
```

Reducing RDDs

When executing `.reduce()` on an RDD, the reducing function must be both *commutative* and *associative* due to the fact that RDDs are partitioned and sent to different nodes. Enforcing these two properties will guarantee that parallelized tasks can be executed and completed in any order without affecting the output. Examples of operations with these properties include addition and multiplication.

```
# start a new SparkSession
from pyspark.sql import SparkSession
spark =
SparkSession.builder.getOrCreate()

# create an RDD
rdd =
spark.sparkContext.parallelize([1,2,3,4,
5])

# add all elements together
print(rdd.reduce(lambda x,y: x+y))
# output: 15

# multiply all elements together
print(rdd.reduce(lambda x,y: x*y))
# output: 120
```

codecademy

Starting a SparkSession

A SparkSession is the entry point to Spark SQL. The session is a wrapper around a SparkContext and contains all the metadata required to start working with distributed data.

```
# start a SparkSession
spark =
SparkSession.builder.getOrCreate()
```

RDDs to DataFrames

PySpark DataFrames can be created from RDDs using `rdd.toDF()` . They can also be converted back to RDDs with `DataFrame.rdd` .

```
# Create an RDD from a list
hrly_views_rdd  =
spark.sparkContext.parallelize([
    ["Betty_White" , 288886],
    ["Main_Page", 139564],
    ["New_Year's_Day", 7892],
    ["ABBA", 8154]
])

# Convert RDD to DataFrame
hrly_views_df = hrly_views_rdd\
    .toDF(["article_title",
"view_count"])

# Convert DataFrame back to RDD
hrly_views_rdd = hrly_views_df.rdd
```

Reading and Writing using PySpark

PySpark allows users to work with external data by reading from or writing to those files. Developers can use `spark.read.<file-format>(filename)` and `spark.write.<fileformat>(filename)` to read and write data between external files and Spark DataFrames.

```
# read from an external parquet file
df =
spark.read.parquet('parquet_file.parquet
')

# write to an external parquet file
spark.write.parquet('parquet_file.parque
t', mode="overwrite")
```

codecademy

Inspecting DataFrame Schemas

All DataFrames have a schema that defines their structure, columns, datatypes, and value restrictions. We can use `DataFrame.printSchema()` to show a DataFrame's schema.

```
# view schema DataFrame df
df.printSchema()

# output:
root
|-- language_code: string (nullable =
true)
|-- article_title: string (nullable =
true)
|-- hourly_count: integer (nullable =
true)
|-- monthly_count: integer (nullable =
true)
```

DataFrame Columns

Similar to pandas DataFrames, PySpark columns can be dropped and renamed.

```
# Dropping a column
df = df.drop('column_name')

# Renaming a column
df = df.withColumnRenamed('old_name',
'new_name')
```

Creating a Temp View

If there is a query that is often executed, we can save some time by saving that query as a temporary view. This saves the results as a table that can be stored in memory and used for future analysis.

```
# create a view from an existing
dataframe and then query from it
tiny_df.createOrReplaceTempView('tiny_vi
ew')
spark.sql("SELECT * FROM
tiny_view").show()
```

codecademy

Using Parquet Files

Parquet is a file format used with Spark to save DataFrames. Parquet format offers many benefits over traditional file formats like CSV:

- Parquet files are efficiently compressed, so they are smaller than CSV files.
- Parquet files preserve information about a DataFrame's schema.
- Performing analysis on parquet files is often faster than CSV files.

```
# Write DataFrame to Parquet
df.write.parquet('./cleaned/parquet/views/', mode="overwrite")

# Read Parquet as DataFrame
df_restored = spark.read.parquet('./cleaned/parquet/views/')
```

Querying DataFrames with SQL

PySpark allows users to query DataFrames using standard SQL queries.

```
# create a view
df.createTempView("tiny_df")

# query from a PySpark DataFrame
query = """SELECT * FROM tiny_df """
spark.sql(query).show()
```

Print Share