Day3

Transformations Operations

ReduceByKey

Merge the values for each key using an **associative** and **commutative** reduce function.
Associative : $a + (b + c) = (a + b) + c$

```
from operator import add
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.reduceByKey(add).collect())
# [('a', 2), ('b', 1)
```

SortByKey

```
tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
sc.parallelize(tmp).sortByKey().first()
```

groupByKey

Group the values for each key in the RDD into a single sequence. Hash-partitions

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.groupByKey().mapValues(len).collect())
[('a', 2), ('b', 1)]
```

```
sortByKey(ascending=True, numPartitions=None, keyfunc=<function RDD.<lambda>>)
```

join

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in `self` and (k, v2) is in `other`.

```
x = sc.parallelize([("a", 1), ("b", 4)])
y = sc.parallelize([("a", 2), ("a", 3)])
sorted(x.join(y).collect())
[('a', (1, 2)), ('a', (1, 3))]
```

Actions Operation

reduce

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

```
from operator import add
sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
# 15
sc.parallelize((2 for _ in range(10))).map(lambda x: 1).cache().reduce(add)
# 10
sc.parallelize([]).reduce(add)
```

saveAsTextFile

Save this RDD as a text file, using string representations of elements.
can use file format

```
tempFile = NamedTemporaryFile(delete=True)
tempFile.close()
sc.parallelize(range(10)).saveAsTextFile(tempFile.name)
from fileinput import input
from glob import glob
''.join(sorted(input(glob(tempFile.name + "/part-0000*"))))
# '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

coalesce

coalesce(numPartitions, shuffle=False)
Return a new RDD that is reduced into numPartitions partitions.

```
sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
[[1], [2, 3], [4, 5]]
sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()
[[1, 2, 3, 4, 5]]
```

countByKey()

paired rdd
Count the number of elements for each key, and return the result to the master as a dictionary.

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.countByKey().items())
# [('a', 2), ('b', 1)]
```

collectAsMap

Return the key-value pairs in this RDD to the master as a dictionary.

```
m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
m[1]
# 2
m[3]
# 4
```

Running .py file in spark

```
spark-submit Ex4_Printwordfrequencies.py
```

Reduce Verbose Logs

cp log4j.properties.template log4j.properties
edit this file nano log4j.properties

```
log4j.rootCategory=ERROR, console
log4j.logger.org.apache.spark.repl.Main=ERROR
```

Classwork

Challenge Labs

4 Qs in RDD

PySpark DataFrames

The entry point to programming Spark with the Dataset and DataFrame API.

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files. To create a SparkSession, use the following builder pattern:

sparkSession -> entry -> spark dataframes
schemas

RDD Objects

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# 1. Initialize SparkSession
spark = SparkSession.builder \
    .appName("RDD to DataFrame Example") \
    .getOrCreate()

# 2. Define your list of data (4 rows)
data = [
    ("Alice", 1, "New York"),
    ("Bob", 2, "London"),
    ("Charlie", 3, "Paris"),
    ("David", 4, "Tokyo")
]

# 3. Create an RDD from the list
rdd = spark.sparkContext.parallelize(data)

# 4. Define the schema for your DataFrame (optional but recommended for
clarity and type safety)
schema = StructType([ StructField("Name", StringType(), True),
StructField("ID", IntegerType(), True), StructField("City", StringType(),
True) ])

# 5. Create DataFrame from RDD with schema
df = spark.createDataFrame(rdd, schema)

# 6. Show the DataFrame
df.show()

# 7. Print the schema to verify types
df.printSchema()

# Stop the SparkSession
spark.stop()
```

Other Sources

```
# Other data sources
# read methods
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder \
.appName("Example App") \
.getOrCreate()

# Use the Spark session
file_path = 'file:///home/talentum/test-
jupyter/P2/M3/SM1/1_AbstractingDatawithDataFrames/Dataset/people.csv'

df_csv = spark.read.csv(file_path, header=True, inferSchema=True)

df_json = spark.read.json(file_path, header=True, inferSchema=True)

df_txt = spark.read.txt(file_path, header=True, inferSchema=True)

df_csv.show()
```

Dataframes

Transformations

filter

```
people_df_male = people_df.filter(people_df.sex == "male")
people_df_sub_dup1 = people_df_sub_count.filter('count > 1')
people_df_sub_dup1 = people_df_sub_count.filter(people_df_sub_count["count"] >
1)

# Does Not work count is a keyword reserved for count method
people_df_sub_dup3 = people_df_sub_count.filter(people_df_sub_count.count > 1)
# Error

# Renaming Works
people_df_sub_count = people_df_sub_count.withColumnRenamed("count", "Count")
people_df_sub_dup3 = people_df_sub_count.filter(people_df_sub_count.Count > 1)
```

select

```
people_df_sub = people_df.select('name','sex', 'date of birth')
```

orderBy
Returns a new DataFrame sorted by the specified column(s).

```
people_df_sub_nodup_rename_ordered =
people_df_sub_nodup_rename.orderBy("name")
```

dropDuplicates

```
people_df_sub_nodup = people_df_sub.dropDuplicates()
```

withColumnRenamed

```
df_csv = df_csv.withColumnRenamed("sex", "gender")
```

Actions

printSchema

```
print(people_df.printSchema())
```

columns

Returns a Column based on the given column name.

```
people_df.columns
```

describe

Computes basic statistics for numeric and string columns.

```
print(people_df.describe().show())
```

count

```
people_df.count()
```

groupBy

```
df.groupBy().avg().collect()
```

DataFrame API vs SQL queries

Executing SQL Queries

Need Hive

```
df.createOrReplaceTempView("table1")

df2 = spark.sql("SELECT field1, field2 FROM table1")
df2.collect()

spark.sql(query).show()
```

```
# start the script before running sql
./Start-Hadoop-Hive.sh

# Check Java Processes
jps
```

select
filter

Deep Dive into DataFrame
Real Time Streaming
Spark ML