PySpark Cheat Sheet

Initializing a SparkContext and SparkSession

- **SPARKCONTEXT**

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName('AppName').setMaster('local')
sc = SparkContext(conf=conf)
```

- **SPARKSESSION**

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('myAppName').master('local').getOrCreate()
```

Creating a Resilient Distributed Dataset (RDD)

- **CREATE AN RDD FROM A TEXT FILE**

```
rdd = sc.textFile("/path/textfile.txt")
```

- **CREATING AN RDD FROM AN EXISTING RDD**

```
rdd2 = rdd.map(lambda x: x * x)
```

- **FROM A CSV FILE**

```
rdd = sc.textFile("/path/csvfile.csv")
```

- **FROM A JSON FILE**

```
import json
rddFromJson = sc.textFile("/path/to/your/jsonfile.json").map(json.loads)
```

- **FROM AN HDFS FILE**

```
rddFromHdfs = sc.textFile("hdfs://localhost:9000/path/to/your/file")
```

- **FROM A SEQUENCE FILE**

```
rddFromSequenceFile = sc.sequenceFile("/path/to/your/sequencefile")
```

- **PARALLELIZING AN EXISTING COLLECTION**

```
#Initialize a SparkContext
....
#Create an RDD from a Python list
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

Creating a DataFrame

- **FROM A LIST OF TUPLES**

```
data = [("John", "Doe", 30), ("Jane", "Doe", 25)]
df = spark.createDataFrame(data, ["First_Name", "Last_Name", "Age"])
df.show()
```

- **FROM A CSV FILE**

```
#Create a DataFrame by reading a CSV file
df = spark.read.csv("/path/to/your/csvfile.csv", inferSchema=True, header=True)
df.show()
```

- **FROM A PANDAS DATAFRAME**

```
import pandas as pd
#Create a pandas DataFrame
pandas_df = pd.DataFrame({ "First_Name":["John","Jane"],"Last_Name":["Doe","Doe"],"Age":[30,25]})
#Convert the pandas DataFrame to PySpark DataFrame
df = spark.createDataFrame(pandas_df)
df.show()
```

- **FROM AN RDD**

```
from pyspark.sql import SparkSession
#initialize a SparkSession
...
#Create an RDD
rdd = spark.sparkContext.parallelize([("John", "Doe", 30),("Jane", "Doe", 25)])
#Convert the RDD to DataFrame
df = rdd.toDF(["First_Name", "Last_Name", "Age"])
df.show()
```

- **FROM A LIST OF ROW OBJECTS**

```
from pyspark.sql import Row
data = [Row(F_Name="John",L_Name="Doe",Age=30),Row(F_Name="Jane", L_Name="Doe", Age=25)]
df = spark.createDataFrame(data)
df.show()
```

- **FROM A JSON FILE**

```
df = spark.read.json("/path/jsonfile.json")
df.show()
```

- **FROM A PARQUET FILE**

```
df = spark.read.parquet("/path/pqtfile.parquet")
df.show()
```

Transformations

NARROW TRANSFORMATIONS

- **map(func)**

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
mapped_rdd = rdd.map(lambda x: x * x)
mapped_rdd.collect()  #Output: [1, 4, 9, 16, 25]
```

- **flatMap(func)**

```
rdd = spark.sparkContext.parallelize([2, 3, 4])
flat_mapped_rdd = rdd.flatMap(lambda x: range(x, 6))
flat_mapped_rdd.collect()  #Output: [2, 3, 4, 5, 3, 4, 5, 4, 5]
```

- **union(dataset)**

```
rdd1 = spark.sparkContext.parallelize([1, 2, 3])
rdd2 = spark.sparkContext.parallelize([4, 5, 6])
union_rdd = rdd1.union(rdd2)
union_rdd.collect()  #Output: [1, 2, 3, 4, 5, 6]
```

- **filter(func)**

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
filtered_rdd.collect()  #Output: [2, 4]
```

- **distinct()**

```
rdd = spark.sparkContext.parallelize([1, 1, 2, 2, 3, 3])
distinct_rdd = rdd.distinct()
distinct_rdd.collect()  #Output: [1, 2, 3]
```

- **mapPartitions(func)**

```
def process_partition(iterator):
    yield sum(iterator)
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5], 2)
result_rdd = rdd.mapPartitions(process_partition)
result_rdd.collect()  #Output: [3, 12]
```

**mapPartitions(func) can be either narrow or wide depending on your function func. If it demands data from other partitions, then it's a wide transformation, otherwise it's a narrow one.*

WIDE TRANSFORMATIONS

- **groupByKey()**

```
rdd = spark.sparkContext.parallelize([("a", 1), ("b", 1), ("a", 1)])
grouped_rdd = rdd.groupByKey()
grouped_rdd.collect()  #Output:[('a', <pyspark.resultiterable.ResultIterable object at 0x10a6d0410>), ('b', <pyspark.resultiterable.ResultIterable object at 0x10a6d0510>)]
```

- **reduceByKey(func)**

```
rdd = spark.sparkContext.parallelize([("a", 1), ("b", 1), ("a", 1)])
reduced_rdd = rdd.reduceByKey(lambda a, b: a + b)
reduced_rdd.collect()  #Output: [('a', 2), ('b', 1)]
```

- **aggregateByKey(zeroValue)(seqOp, combOp)**

```
seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
rdd = spark.sparkContext.parallelize([("a", 1), ("b", 1), ("a", 2)], 2)
agg_rdd = rdd.aggregateByKey((0, 0))(seqOp, combOp)
agg_rdd.collect()  #Output: [('a', (3, 2)), ('b', (1, 1))]
```

- **sortBy(keyfunc)**

```
rdd = spark.sparkContext.parallelize([("a", 3), ("b", 1), ("a", 2)])
sorted_rdd = rdd.sortBy(lambda x: x[1])
sorted_rdd.collect()  #Output: [('b', 1), ('a', 2), ('a', 3)]
```

- **join(otherDataset)**

```
rdd1 = spark.sparkContext.parallelize([("a", 1), ("b", 4)])
rdd2 = spark.sparkContext.parallelize([("a", 2), ("a", 3)])
join_rdd = rdd1.join(rdd2)
join_rdd.collect()  #Output: [('a', (1, 2)), ('a', (1, 3))]
```

DataFrame API

DATAFRAME OPERATIONS

- **select()**
```
df.select("column1", "column2").show()
```
- **groupBy()**
```
df.groupBy("column1").count().show()
```
- **drop()**
```
df.drop("column1", "column2").show()
```
- **limit()**
```
df.limit(10).show()
```
- **union()**
```
df1.union(df2).show()
```
- **filter()**
```
df.filter(df["column1"] > 0).show()
```
- **orderBy()**
```
df.orderBy(df["column1"].desc()).show()
```
- **ordedistinct()**
```
df.distinct().show()
```
- **repartition()**
```
df.repartition(10)
```
- **withColumn()**
```
from pyspark.sql.functions import col
df.withColumn("new_column", col("column1") * 2).show()
```
- **withColumnRenamed()**
```
df.withColumnRenamed("old_name", "new_name").show()
```
- **join()**
```
df1.join(df2, df1["column1"] == df2["column2"]).show()
```

DATAFRAME STATISTICAL FUNCTIONS

- **describe()**
```
df.describe().show()
```
- **cov()**
```
df.stat.cov("column1", "column2")
```
- **corr()**
```
df.stat.corr("column1", "column2")
```
- **crosstab()**
```
df.stat.crosstab("col1", "col2").show()
```
- **freqItems()**
```
df.stat.freqItems(["column1", "column2"]).show()
```
- **sampleBy()**
```
fractions = {"female": 0.2, "male": 0.8}
df.stat.sampleBy("gender", fractions).show()
```
- **approxQuantile()**
```
df.stat.approxQuantile("column1", [0.25, 0.5, 0.75], 0.05)
```
- **histogram()**
```
df.select("column1").rdd.flatMap(lambda x: x).histogram(5)
```

HANDLING MISSING DATA

- **dropna()**

```
df.dropna().show() #Drop rows that have at least one null value
df.dropna(subset=["column1", "column2"]).show() #Drop rows that have null values in specific cols
df.dropna(how="all").show() #Drop rows that have null values in all columns
```

- **fillna()**

```
df.fillna(-1).show() #Fill all null values with a specified value
#Fill null values in specific columns with a specified value
df.fillna({"column1": -1, "column2": "unknown"}).show()
```

- **replace()**

```
df.replace(1, 2, subset=["column1"]).show() #Replace all occurrences of 1 with 2 in column1
```

SQL QUERIES WITH createOrReplaceTempView() AND spark.sql()

- **createOrReplaceTempView()**

```
#Create DataFrame
data = [("John", "Doe", 30), ("Jane", "Doe", 25)]
df = spark.createDataFrame(data, ["FirstName", "LastName", "Age"])
#Create Temporary View
df.createOrReplaceTempView("people")
```

**Once a temporary view is created, you can run SQL queries on the DataFrame as if it was a SQL table using the spark.sql() function.*

- **spark.sql()**

```
#SELECT Query
results = spark.sql("SELECT * FROM people WHERE Age > 28")
results.show()
#Aggregation
results_agg = spark.sql("SELECT AVG(Age) as average_age FROM people")
results_agg.show()
#Join Operations
data2 = [("Doe", "New York"), ("Doe", "San Francisco")]
df2 = spark.createDataFrame(data2, ["LastName", "City"])
df2.createOrReplaceTempView("locations")
results_join = spark.sql("SELECT p.FirstName, p.LastName, l.City FROM people p INNER JOIN
locations l ON p.LastName = l.LastName")
results_join.show()
#Subqueries
results_subquery = spark.sql("SELECT * FROM people WHERE Age > (SELECT AVG(Age) FROM people)")
results_subquery.show()
```

Working with Different Data Formats

READING AND WRITING DATA

- **CSV:**
  - **Reading**
    ```
    df = spark.read.format("csv").option("header", "true").load("<path>")
    ```
  - **Writing**
    ```
    df.write.format("csv").option("header", "true").save("<path>")
    ```
- **JSON:**
  - **Reading**
    ```
    df=spark.read.format("json").load("<path>")
    ```
  - **Writing**
    ```
    df.write.format("json").save("<path>")
    ```
- **PARQUET:**
  - **Reading**
    ```
    df = spark.read.format("parquet").load("<path>")
    ```
  - **Writing**
    ```
    df.write.format("parquet").save("<path>")
    ```
- **AVRO:**
  - **Reading**
    ```
    df=spark.read.format("avro").load("<path>")
    ```
  - **Writing**
    ```
    df.write.format("avro").save("<path>")
    ```
- **JDBC:**
  - **Reading**
    ```
    df = spark.read.format("jdbc").option("url", "jdbc:postgresql:dbserver") \
        .option("dbtable", "schema.tablename").option("user", "username") \
        .option("password", "password").option("driver", "org.postgresql.Driver").load()
    ```
  - **Writing**
    ```
    df.write.format("jdbc").option("url", "jdbc:postgresql:dbserver") \
        .option("dbtable", "schema.tablename").option("user", "username") \
        .option("password", "password").option("driver", "org.postgresql.Driver").save()
    ```
- **XML** (need to ensure spark-xml package is available)**:**
  - **Reading**
    ```
    df = spark.read.format('com.databricks.spark.xml').options(rowTag='book') \
        .load('/path/to/xml')
    ```
  - **Reading**
    ```
    df.write.format('com.databricks.spark.xml').options(rowTag='book') \
        .save('/path/to/xml')
    ```

DEALING WITH SCHEMA DURING DATA INGESTION

- **INFERRING SCHEMA AUTOMATICALLY**
  ```
  df = spark.read.format("csv").option("header", "true") \
      .option("inferSchema", "true").load("/path/to/csv")
  ```

- **DEFINING SCHEMA EXPLICITLY**

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
#Define schema
schema = StructType([
    StructField("FirstName", StringType(), True),
    StructField("LastName", StringType(), True),
    StructField("Age", IntegerType(), True)
])
#Read data with schema
df = spark.read.format("csv").schema(schema).load("/path/to/csv")
```

- **MODIFYING SCHEMA AFTER INGESTION**

```
#Add new column
df = df.withColumn("NewColumn", df["Age"] * 2)
#Drop column
df = df.drop("NewColumn")
#Rename column
df = df.withColumnRenamed("Age", "UserAge")
```

- **INSPECTING SCHEMA**

```
df.printSchema()
```

PySpark MLlib

DATA PREPARATION

- **STRINGINDEXER**

```
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()
```

- **VECTORASSEMBLER**

```
from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")
output = assembler.transform(df)
output.show()
```

- **ONEHOTENCODER**

```
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder(inputCol="categoryIndex", outputCol="categoryVec")
encoded = encoder.transform(indexed)
encoded.show()
```

ALGORITHMS

- **LINEAR REGRESSION**

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
#Fit the model
lrModel = lr.fit(trainingData)
#Print the coefficients and intercept for linear regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

- **LOGISTIC REGRESSION**

```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
#Fit the model
lrModel = lr.fit(trainingData)
#Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

- **DECISION TREE CLASSIFIER**

```
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures")
#Fit the model
dtModel = dt.fit(trainingData)
#Make predictions
predictions = dtModel.transform(testData)
```

- **RANDOM FOREST CLASSIFIER**

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures",
numTrees=10)
#Fit the model
rfModel = rf.fit(trainingData)
#Make predictions
predictions = rfModel.transform(testData)
```

- **KMEANS**

```
from pyspark.ml.clustering import KMeans
kmeans = KMeans(k=2, seed=1)  #Initialize model
#Fit the model
model = kmeans.fit(dataset)
#Get the cost (Squared Euclidean Distance)
wssse = model.computeCost(dataset)
print("Within Set Sum of Squared Errors = " + str(wssse))
#Shows the result
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

Spark Streaming

CREATING DISCRETIZED STREAM

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
#Create a local StreamingContext with two working threads and a batch interval of 2 seconds
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 2)
lines = ssc.socketTextStream("localhost", 9999)
words = lines.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
#Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
ssc.start()             #Start the computation
ssc.awaitTermination()  #Wait for the computation to terminate
```

TRANSFORMATIONS ON DSTREAMS

- **map()**

```
numbers = dstream.map(lambda x: int(x))
```

- **flatMap()**

```
words = lines.flatMap(lambda line: line.split(" "))
```

- **filter()**

```
errors = lines.filter(lambda line: "error" in line)
```

- **reduceByKey()**

```
wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

- **window()**

```
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```

- **updateStateByKey()**

```
def updateFunc(new_values, last_sum):
    return sum(new_values) + (last_sum or 0)
runningCounts = pairs.updateStateByKey(updateFunc)
```

- **Sliding window**

```
windowedDStream = dStream.window(windowDuration, slideDuration)
```

- **tumbling window**

```
windowedDStream = dStream.window(windowDuration, windowDuration)
```

OUTPUT OPERATIONS ON DSTREAMS

- **pprint()**

```
dstream.pprint()
```

- **saveAsTextFiles()**

```
dstream.saveAsTextFiles(prefix, [suffix])
```

PySpark Commands when Interacting with Hive

- **INITIALIZING A SPARKSESSION WITH HIVE SUPPORT**

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Hive with PySpark").enableHiveSupport().getOrCreate()
```

- **CREATING HIVE TABLES**

```
spark.sql("CREATE TABLE IF NOT EXISTS employees (name STRING, age INT, department STRING) USING hive")
```

- **LOADING DATA INTO HIVE TABLES**

```
spark.sql("LOAD DATA LOCAL INPATH 'input/file/path' INTO TABLE employees")
```

- **INSERTING DATA INTO HIVE TABLES**

```
spark.sql("INSERT INTO TABLE employees VALUES ('John', 30, 'Sales')")
```

- **RUNNING SQL QUERIES**

```
results = spark.sql("SELECT name, age, department FROM employees WHERE age > 30")
```

- **WRITING DATAFRAME TO HIVE TABLE**

```
df.write.saveAsTable("employees")
```

- **READING FROM HIVE TABLE TO DATAFRAME**

```
df = spark.table("employees")
```

- **CREATING HIVE TABLES WITH PARTITIONING**

```
spark.sql("CREATE TABLE employees (name STRING, age INT) PARTITIONED BY (department STRING) USING hive")
```

- **LOADING DATA INTO HIVE PARTITIONED TABLES**

```
spark.sql("LOAD DATA LOCAL INPATH 'input/file/path' INTO TABLE employees PARTITION (department='Sales')")
```

- **INSERTING DATA INTO HIVE PARTITIONED TABLES**

```
spark.sql("INSERT INTO TABLE employees PARTITION (department='Sales') VALUES ('John', 30)")
```

- **READING FROM A SPECIFIC HIVE PARTITION TO DATAFRAME**

```
df = spark.sql("SELECT * FROM employees WHERE department = 'Sales'")
```

- **ADDING A NEW PARTITION TO HIVE TABLE**

```
spark.sql("ALTER TABLE employees ADD PARTITION (department='HR')")
```

- **DROP A PARTITION FROM HIVE TABLE**

```
spark.sql("ALTER TABLE employees DROP PARTITION (department='HR')")
```

- **REFRESH TABLE TO MAKE ALL DATA IMMEDIATELY VISIBLE**

```
spark.catalog.refreshTable("employees")
```