Cheatsheets / **Introduction to Big Data with PySpark**

codecademy

Spark RDDs with PySpark

Properties of RDDs

The three key properties of RDDs:

- **Fault-tolerant** (resilient): data is recoverable in the event of failure
- **Partitioned** (distributed): datasets are cut up and distributed to nodes
- **Operated on in parallel** (parallelization): tasks are executed on all the chunks of data at the same time

Transforming an RDD

A transformation is a Spark operation that takes an existing RDD as an input and provides a new RDD that has been modified by the transformation as an output.

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

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

# transform tiny_rdd
transformed_tiny_rdd =
tiny_rdd.map(lambda x: x+1) # apply x+1
to all RDD elements

# view the transformed RDD
transformed_tiny_rdd.collect()
# output:
# [2, 3, 4, 5, 6]
```

codecademy

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

Executing Actions on RDDs

An action is a Spark operation that takes an RDD as input, but always outputs a value instead of a new RDD.

```
# 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])

# execute action
print(rdd.count())

# output:
# 5
```

Spark Transformations are Lazy

Transformations in Spark are not performed until an action is called. Spark optimizes and reduces overhead once it has the full list of transformations to perform. This behavior is called **lazy** evaluation. In contrast, **eager** evaluation is how Pandas transformations behave.

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

Aggregating with Accumulators

Accumulator variables are shared variables that can only be updated through associative and commutative operations. They are primarily used as counters or sums in parallel computing since they operate on each node separately and adhere to both the associative and commutative properties. However, they are only infallible when used in actions because Spark transformations can re-execute after failure, which would wrongfully increment the accumulator.

```
# 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])

# start the accumlator at zero
counter =
spark.sparkContext.accumulator(0)

# add 1 to the accumulator for each
element
rdd.foreach(lambda x: counter.add(1))

print(counter)
# output: 5
```

codecademy

Sharing Broadcast Variables

In Spark, broadcast variables are cached input datasets that are sent to each node. This provides a performance boost when running operations that utilize the broadcasted dataset since all nodes have access to the same data. We would never want to broadcast large amounts of data because the size would be too much to serialize and send through the network.

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

# create an RDD
rdd =
spark.sparkContext.parallelize(["Plane",
"Plane", "Boat", "Car", "Car", "Boat",
"Plane"])

# dictionary to broadcast
travel = {"Plane":"Air", "Boat":"Sea",
"Car":"Ground"}

# create broadcast variable
broadcast_travel =
spark.sparkContext.broadcast(travel)

# map the broadcast variable to the RDD
result = rdd.map(lambda x:
broadcast_travel.value[x])

# view first four results
result.take(4)
# output : ['Air', 'Air', 'Sea',
'Ground']
```

Spark Overview

Spark is an application that was designed to process large amounts of data. Originally designed for creating data pipelines for machine learning workloads, Spark is capable of querying, transforming, and analyzing big data on a variety of data systems.

codecademy

Spark Process Overview

Spark is able to process data quickly because it leverages the Random Access Memory (RAM) of a computing cluster. When processing data, Spark keeps the data in RAM, which is a faster processing part of a computing node. Spark does this in parallel across all worker nodes in a cluster. This differs from MapReduce, which processes data on the node's disk, and explains why Spark is a faster framework than MapReduce.

Pyspark Overview

The Spark framework is written in Scala but can be used in several languages, namely Python, Java, SQL, and R.
Pyspark is the Python API for Spark that can be installed directly from the leading Python repositories (PyPI and conda). Pyspark is a particularly popular framework because it makes the big data processing of Spark available to Python programmers. Python is a more approachable and familiar language for many data practitioners than Scala.

Print Share