

To run these examples, we mostly use PySpark, which is Spark's interactive shell for Python.

You can run PySpark either using Google Colab or in a Docker container.

In the latter case, you can use the official Docker image from Apache Spark (<https://hub.docker.com/r/apache/spark>).

To pull the image

```
docker pull spark:python3
```

We then run the Docker container and mount a volume with some input data (located in your local data/ directory):

```
docker run -it -v $PWD/data:/sabd-data -p 8080:8080  
-p:4040:4040 --rm spark /opt/spark/bin/pyspark
```

PySpark will start inside the container.

The Spark Web UI is accessible from your browser at <http://localhost:4040>

As an alternative, you can use the image provided by Bitnami, which also includes an Apache Spark cluster setup with Docker Compose <https://hub.docker.com/r/bitnami/spark>

The docker-compose.yml file is provided and includes a bind mount.

To run a cluster composed of 1 master and 3 worker nodes, use:

```
docker compose up --scale spark-worker=3
```

To submit jobs, you must enter the spark master container:

```
docker exec -it spark-master bash
```

Let's now play with Spark using pyspark.

--- Create RDDs

```
lines = sc.textFile("/sabd-data/input.txt")  
lines.collect()
```

--- Transformations

```
# map: transform each element through a function  
nums = sc.parallelize([1, 2, 3, 4])  
squares = nums.map(lambda x: x * x)  
nums.collect()  
squares.collect()
```

```
# filter: select those elements that func returns true  
even = squares.filter(lambda num: num % 2 == 0)  
even.collect()
```

```
# flatMap: map each element to zero or more others  
ranges = nums.flatMap(lambda x: range(0, x, 1))  
ranges.collect()  
# splitting input lines into words  
lines = sc.parallelize(["hello world", "hi"])  
words = lines.flatMap(lambda line: line.split(" "))  
words.collect()
```

```
# union: return the union of two RDDs
```

```

rdd1 = sc.parallelize([2, 4, 7, 9])
rdd2 = sc.parallelize([1, 4, 5, 8, 9])
rdd3 = rdd1.union(rdd2)
rdd3.collect()

# distinct: return a new RDD that contains the distinct elements
# of the source RDD: helpful to remove duplicate data
rdd3.distinct().collect()

# mapPartitions: return a new RDD by applying a function to each
# partition of the source RDD
rdd1 = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 3)
# Define a function to add the elements into each partition
def f(iterator): yield sum(iterator)
rdd2 = rdd1.mapPartitions(f)
rdd2.collect()

# glom: return a new RDD by coalescing all elements within each
# partition into a list
rdd1.glom().collect()

# partitionBy: returns a new RDD that contains the RDD partitioned
# using the specified partitioner and number of partitions
rdd = sc.parallelize([("apple", 1), ("banana", 2), ("orange", 3),
("apple", 4), ("banana", 1)])
rdd.collect()
partitioned_rdd = rdd.partitionBy(3)
partitioned_rdd.glom().collect()

# reduceByKey: aggregate values with identical key
x = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b",
1), ("b", 1), ("b", 1), ("b", 1)], 3)
y = x.reduceByKey(lambda accum, n: accum + n)
y.collect()

# join: perform equi-join on the keys of two RDDs
users = sc.parallelize([(0, "Alex"), (1, "Bert"), (2, "Curt"), (3,
"Don")])
hobbies = sc.parallelize([(0, "writing"), (0, "gym"), (1,
"swimming")])
users.join(hobbies).collect()
# Let's analyze the DAG using the Spark Web UI

# mapValues: pass each value in the key-value pair RDD through
# a map function without changing the keys; this also retains
# the original RDD's partitioning.
# Differently from map, it operates only on the value.
fruits = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b",
["grapes"])]])
fruits.mapValues(len).collect()

# groupByKey: group the values for each key in the RDD into a single
# sequence. Hash-partitions the resulting RDD with numPartitions
# partitions.

```

```

# If you are grouping in order to perform an aggregation
# (e.g., sum or average) over each key, using reduceByKey or
# aggregateByKey will provide better performance.
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.groupByKey().mapValues(len).collect())

# repartition: change the number of partitions (i.e., the level
# of parallelism) in the source RDD
rdd = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 3)
rdd.glom().collect()
repartitioned_rdd = rdd.repartition(4)
repartitioned_rdd.glom().collect()

# coalesce: decrease the number of partitions in the source RDD
rdd = sc.parallelize([1, 2, 3, 4, 5], 3)
rdd.glom().collect()
repartitioned_rdd = rdd.coalesce(1)
repartitioned_rdd.glom().collect()

```

--- Actions

```

nums = sc.parallelize([1, 2, 3, 4])
nums.collect()

nums.take(3)

nums.count()

sum = nums.reduce(lambda x, y: x + y)
sum

nums.saveAsTextFile("/sabd-data/out")

```

--- Greek Pi calculation using Monte Carlo method

```

import random
NUM_SAMPLES = 100000

def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

samples = sc.parallelize(range(0, NUM_SAMPLES))
within_circle = samples.filter(inside)
count = within_circle.count()
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))

```

--- WordCount

```

text_file = sc.textFile("/sabd-data/input.txt")

```

```
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("/sabd-data/output")
```

--- WordCount using countByValue: when can we use it?

```
text_file = sc.textFile("/sabd-data/input.txt")

counts = text_file.flatMap(lambda line: line.split(" "))
wordCount = words.countByValue()
print(wordCount)
```

--- Compute mean value

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules",
30), ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
agesRDD = (dataRDD
.map(lambda x: (x[0], (x[1], 1)))
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
.map(lambda x: (x[0], x[1][0]/x[1][1])))

# Let's examine the code snippet step by step
mapRDD = dataRDD.map(lambda x: (x[0], (x[1], 1)))
mapRDD.collect()
redRDD = mapRDD.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
redRDD.collect()
agesRDD = redRDD.map(lambda x: (x[0], x[1][0]/x[1][1]))
agesRDD.collect()
```

#####

Launch applications on Spark

```
docker run -it -p 8080:8080 -p:4040:4040 --rm spark:python3 /bin/
bash
```

```
# Inside the container
cd ..
```

```
# Submit PageRank application in Python
./bin/spark-submit examples/src/main/python/pagerank.py data/mllib/
pagerank_data.txt 10
```

```
# Submit Pi estimation application in Java
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master local \
```

```
--deploy-mode client \  
--num-executors 2 \  
--driver-memory 512m \  
--executor-memory 512m \  
--executor-cores 1 \  
examples/jars/spark-examples_2.12-3.5.5.jar 100
```