Distributed System Course
Lab 2 - An Introduction to Apache Spark & RDD-based Programming (Tutorial)

I. GOALS

- Can manipulate basic operations with Apache Spark RDD.
- Can develop batch processing applications with Apache Spark.
- Understand the mechanism and infrastructure of Apache Spark

II. PREPARATION

- You should install scala (you should chose scala 2.12) and Apache Spark (you should chose spark 3.3.0) in your local machine:
  https://www.tutorialspoint.com/apache_spark/apache_spark_installation.htm
  https://www.scala-lang.org/download/2.12.2.html
- Install sbt to compile scala code:
  https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html
- Create a directory to store your scala code.
- Create a text file named build.sbt with the following content:

```
name := "test"

version := "1.0"

scalaVersion := "2.12.2"

val sparkVersion = "3.3.1"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion,
  "org.apache.spark" %% "spark-sql" % sparkVersion,
)
```

- To generate a jar file with scala code, you can run the following command and check the generated file in <folder_name>/target/scala-2.12/.

```
sbt package
```

III. PRACTICE WITH APACHE SPARK

1. spark-submit

a. **Introduction**

Spark programs are executed (submitted) by using the **spark-submit** command

- It is a command line program
- It is characterized by a set of parameters
  - E.g., the name of the jar file containing all the classes of the Spark application we want to execute
  - The name of the Driver class
  - The parameters of the Spark application
  - etc.

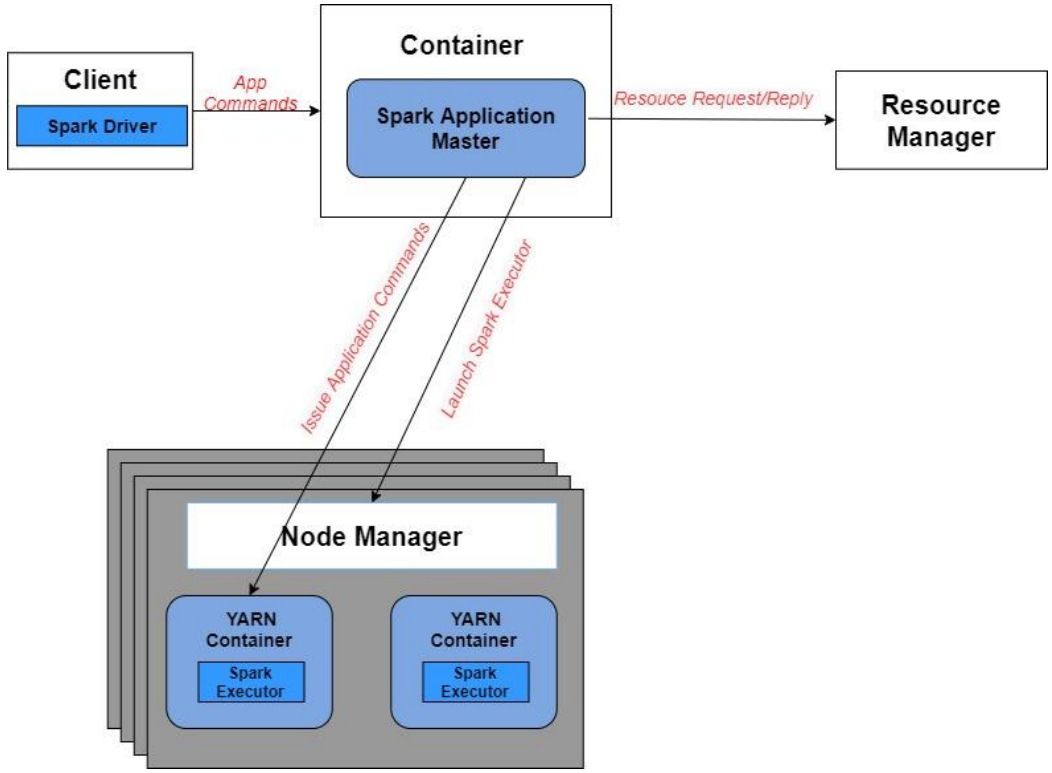
It has two important parameters that are used to specify where the application is executed

- **--master**: Specify which environment/scheduler is used to execute the application (local, standalone, or another node manager, such as Mesos, YARN, etc.)
- **--deploy-mode**: Specify where the Driver is launched/executed (client or cluster)

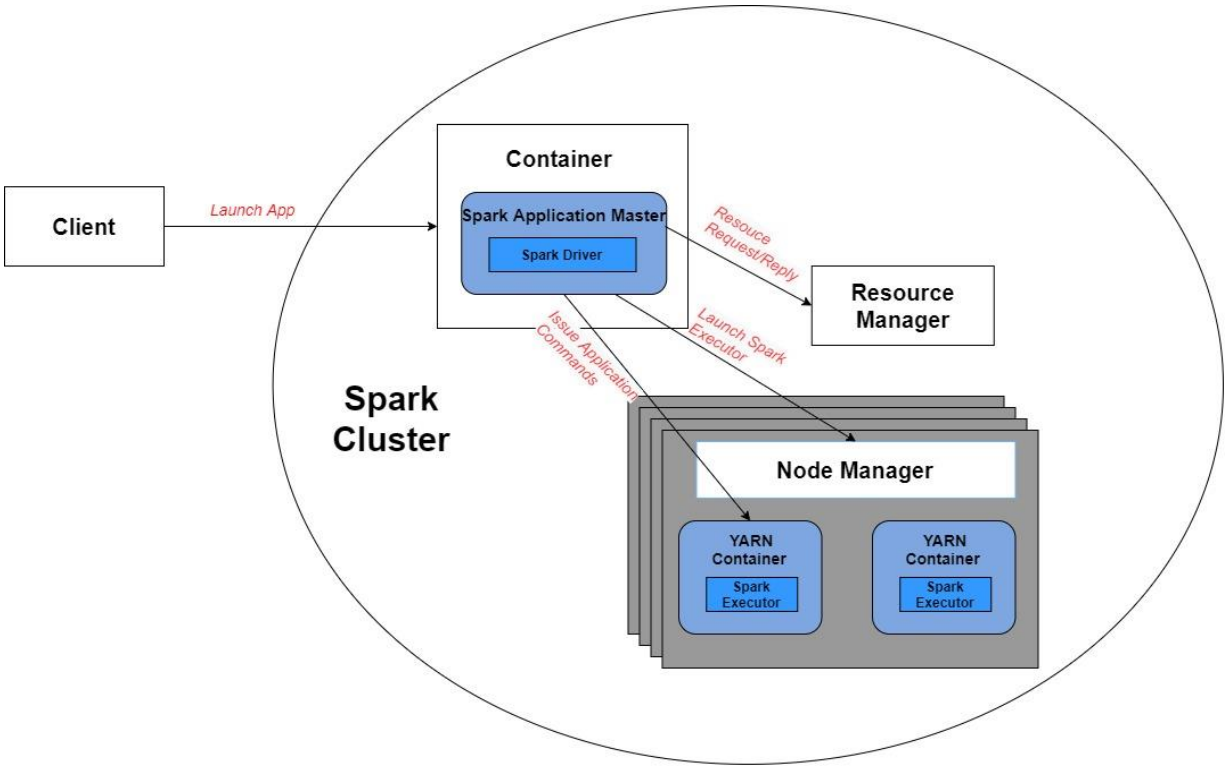
b. Deployment Mode:

There are 2 types of deployment mode:

- **Client mode:** Spark Driver will run on the machine from which the job is submitted.
  - When the submitting machine is near the Spark executors, there is no high network latency of data movement for final result generation. Client mode is a good option.
  - When the submitting machine is very remote to the executors, also has high network latency. In this case, this spark mode does not work in a good manner.



- **Cluster mode:** Spark Driver will **not** run on the machine from which the job is submitted. Alternatively, it will run on a remote node in the cluster.
  - When the submitting machine is remote from executors. Since executors and drivers run nearly. Thus, it reduces data movement between the submitting machine and the executors. => this mode works totally fine.
  - While we work with this mode, the chance of network disconnection between driver and executors reduces. => reduces the chance of job failure.



c. Example

- Create a scala file named test.scala with the following content:

```
package examples

import org.apache.spark.sql.SparkSession

object Test {
  def main(args: Array[String]): Unit = {
      val spark = SparkSession.builder()
      .appName("Test Application")
      .getOrCreate();

      spark.sparkContext.setLogLevel("ERROR")


      val sparkContext = spark.sparkContext
      val sqlCon = spark.sqlContext

      val sqlContext = new
org.apache.spark.sql.SQLContext(spark.sparkContext)

      println("SparkContext:")
      println("APP Name :"+spark.sparkContext.appName);
      println("Deploy Mode :"+spark.sparkContext.deployMode);
      println("Master :"+spark.sparkContext.master);
  }
}
```

- Package the above scala code into a .jar file

- Submit the above application with the following deploy modes:
  - Cluster mode:

```
spark-submit --deploy-mode cluster --master spark://master-ip:7077
test.py
```

  - Client mode:

```
spark-submit --deploy-mode client --master spark://master-ip:7077
test.py
```

- Interactive mode:
  - Spark-shell is **Scala** interactive mode for spark

```
$ spark-shell
....
....
Spark context available as 'sc'
Spark session available as 'spark'
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/
....
....
scala> import scala.math.random
scala> val slices = 1000
scala> val n = math.min(100000L * slices, Int.MaxValue).toInt
scala> val squares_sum = spark.sparkContext.parallelize(1 until n,
slices).map { i => i * i }.reduce(_ + _)
scala> println(s"Sum of squares is ${squares_sum}")
scala> :quit
```

  - Pyspark is **Python** interactive mode for spark

```
$ pyspark
....
....
Spark context available as 'sc'
Spark session available as 'spark'
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/
....
```

```
....
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder \
       .master("yarn") \
       .appName("SparkByExamples.com") \
       .getOrCreate()
>>> dataList = [("Java", 20000), ("Python", 100000), ("Scala", 3000)]
>>> rdd=spark.sparkContext.parallelize(dataList)
>>>  df = spark.read.option("header",True).csv("hdfs://10.1.7.194:9000/share/test/Uber-Jan-Feb-FOIL.csv")
>>> df.printSchema()
## create a temp view
>>> df.createOrReplaceTempView("uber")
## simple query: finding the distinct NYC Uber bases  in the CSV
>>> distinct_bases = sqlContext.sql("select distinct dispatching_base_number from uber")
>>> for b in distinct_bases.collect(): print(b)

## determining which Uber bases is the busiest based on the number of trips
>>> sqlContext.sql("""select distinct(`dispatching_base_number`),
                        sum(`trips`) as cnt from uber
                        group by `dispatching_base_number`
                        order by cnt desc""").show()

## 5 busiest days based on the number of trips in the time range of the data
>>> sqlContext.sql("""select distinct(date),sum(trips) as cnt
                       from uber group by date
                       order by cnt desc limit 5""").show()
```

2. Resilient Distributed Dataset:

a. Introduction

A Spark RDD is an immutable distributed collection of objects. Each RDD is split in partitions. This choice allows parallelizing the code based on RDDs. Code is independently executed on each partition. (data cannot be shared between applications). RDDs can contain any type of **Scala**, Java, and **Python** objects, including user-defined classes.

b. Create, Read and Save

- **Create by reading files:** An RDD can be built from an input file. The input file can be a text file or binary file. For example,
  - Read a text file

```
val lines = sc.textFile("/path/to/file")
val lines = sc.textFile("/path/to/file", 5) // create a RDD with
partition = 5.
```

  - Read a binary file (e.g. a .png file)

```
spark.read.format("binaryFile").option("pathGlobFilter",
"*.png").load("/path/to/data")
```

- **Create by local lists:** You can convert a local list into a RDD object by using parallelize() function. For example,

```
val list = List(1,2,3)
val distList = sc.parallelize(list)
val distList = sc.parallelize(list, 5) // partition = 5
```

Note: If the input file is located in your local machine, the file path should have a prefix as "file://". Otherwise, it will be understood as being located in the HDFS cluster.

- **Save RDD to storage:**
  You can use the saveAsTextFile to store a RDD into a text file.

```
lines.saveAsTextFile("/path/to/folder")
```

3. Transformation and Action Operators in RDD:

a. Introduction

- RDD supports 2 types of operation:
  - Transformation:
    - Are operations on RDDs that return a new RDD.
    - Apply a transformation on the elements of the input RDD(s) and the result of the transformation is "stored in/associated to" a **new RDD**. (RDD is immutable).
    - Are computed lazily.
      - Transformations are computed ("executed") only when an action is applied on the RDDs generated by the transformation operations
    - When a transformation is invoked
      - Spark keeps only track of the dependency between the input RDDand the new RDD returned by the transformation
      - The content of the new RDD is not computed
  - Actions
    - Are operations that
      - Return results to the Driver program
        - i.e., return local variables

      (Note: Pay attention to the size of the returned results because they must be stored in the main memory of the Driver program)
      - Or write the result in the storage (output file/folder)
        - The size of the result can be large in this case since it is directly stored in the (distributed) file system.

b. Basic Transformations:

Transformation operators can be categorized into two types:

- Some basic transformations analyze the content of one **single** RDD and return a new RDD
  - E.g., filter(), map(), flatMap(), etc.
- Some other transformations analyze the content of **two** (input) RDDs and return a new RDD
  - E.g., union(), intersection(), substract(), etc.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func*

	with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command, [envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

c. Basic Actions:

Spark actions can retrieve the content of an RDD or the result of a function applied on an RDD and

- "Store" it in a local variable of the Driver program
  - Pay attention to the size of the returned value
- Or store the content of an RDD in an output folder or database
- The spark actions that return a result that is stored in local variables of the Driver
  - Are executed locally on each node containing partitions of the RDD on which the action is invoked
    - Local results are generated in each node
  - Local results are sent on the network to the Driver that computes the final result and store it in local variables of the Driver

The basic actions returning objects to the Driver are

- **collect**(), count(), countByValue(), take(), top(), takeSample(), **reduce**(), fold(), **aggregate**(), foreach()

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n, [ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators

	outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

4. Transformation and Action Operators in PairRDD:

a. What is a PairRDD?

- Spark supports also RDDs of key-value pairs, a.k.a pair RDDs. Pair RDDs are characterized also by the operations available for the "standard" RDDs. Many applications are based on pair RDDs
  - Pair RDDs allow "grouping" data by key performing computation by key (i.e., group)
  - The basic idea is similar to the one of the MapReduce-based programs
  - But there are more operations already available
- From a local list of pairs, (Scala) It is similar to "standard" RDD, we can use parallelize() method to convert a local list of pairs to a PairRDD. For example,

```
val inputRDD = sc.parallelize(List(("A", 1), ("B", 2)), numPartitions)
```

- From "standard" RDDs, we can use methods, such as: map and flatMap to convert them into PairRDDs.

b. Transformations on PairRDDs

Pair RDD Functions	Function Description
aggregateByKey	Aggregate the values of each key in a data set. This function can return a different result type then the values in input RDD.
combineByKey	Combines the elements for each key.
combineByKeyWithClassTag	Combines the elements for each key.
flatMapValues	It's flatten the values of each key with out changing key values and keeps the original RDD partition.
foldByKey	Merges the values of each key.
groupByKey	Returns the grouped RDD by grouping the values of each key.
mapValues	It applied a map function for each value in a pair RDD with out changing keys.
reduceByKey	Returns a merged RDD by merging the values of each key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and final result will be sent to the master.
sampleByKey	Returns the subset of the RDD.

subtractByKey	Return an RDD with the pairs from this whose keys are not in other.
keys	Returns all keys of this RDD as a RDD[T].
values	Returns an RDD with just values.
partitionBy	Returns a new RDD after applying specified partitioner.
fullOuterJoin	Return RDD after applying fullOuterJoin on current and parameter RDD
join	Return RDD after applying join on current and parameter RDD
leftOuterJoin	Return RDD after applying leftOuterJoin on current and parameter RDD
rightOuterJoin	Return RDD after applying rightOuterJoin on current and parameter RDD

c. Actions on PairRDDs

Pair RDD Action functions	Function Description
collectAsMap	Returns the pair RDD as a Map to the Spark Master.
countByKey	Returns the count of each key elements. This returns the final result to local Map which is your driver.
countByKeyApprox	Same as countByKey but returns the partial result. This takes a timeout as a parameter to specify how long this function will run before returning.
lookup	Returns a list of values from RDD for a given input key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and the final result will be sent to the master.
saveAsHadoopDataset	Saves RDD to any Hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop JobConf objects to save.
saveAsHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop OutputFormat class to save.
saveAsNewAPIHadoopDataset	Saves RDD to any Hadoop supported file

	system (HDFS, S3, ElasticSearch, e.t.c) with new Hadoop API, It uses Hadoop Configuration object to save.
saveAsNewAPIHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses the new Hadoop API OutputFormat class to save.