Spark Hands On Exercise

- Go to the below location

```
cd /usr/lib/spark
```

- Login to Spark shell using the following command

```
./bin/spark-shell
```

Note: Scala shell comes up

Scenario: 1

- Load a file from local file system and count the occurrence of lines having a particular word in it

```
val textFile = sc.textFile("file:///usr/lib/spark/README.md")
```

Note: sc is sparkContext

```
textFile.filter(line => line.contains("Spark"))
textFile.filter(line => line.contains("Spark")).count()
```

Scenario: 2

- Load data and filter it with single and multiple threads and analyze the performance gain

```
val sampleData = 1 to 5000
val totData = sc.parallelize(sampleData)
val result = totData.filter(_ %2 ==0)
result.collect()
val totDataPar = sc.parallelize(sampleData,2)
val resultPar = totDataPar.filter(_ %2 ==0)
resultPar.collect()
```

Scenario: 3

- Checking if python version of spark works

- To get into pySpark Shell

```
cd /usr/lib/spark
./bin/pyspark
textFile = sc.textFile ("file:///usr/lib/spark/README.md")
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
linesWithSpark.count()
exit()
```

Scenario: 4

- SPARK REPL Demo

```
./bin/spark-shell
val textFile = sc.textFile("file:///usr/lib/spark/README.md")
textFile.count() //number of items in this RDD
textFile.first() //First item in this RDD
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
textFile.filter(line => line.contains("Spark")).count() //How many lines
contain "Spark"
```

- Let's say now you want to find out the line with maximum words

```
textFile.map(line => line.split(" ").size).reduce((a,b) => if(a > b) a else
b)
```

Scenario: 5

- Persistence concept in spark, where data is stored in memory i.e. cached which results in huge performance boost.

```
val a = sc.parallelize(1 to 100)
a.getStorageLevel.description

a.persist(org.apache.spark.storage.StorageLevel.DISK_ONLY)
a.getStorageLevel.description

a.count()
```

Scenario: 6

Simple Word count revision

- inside spark-shell

```
val file = sc.textFile("file:///usr/lib/spark/README.md")
file.collect
val counts1 = file.map(line => line.split(" "))
counts1.collect
file.count

val counts = file.flatMap(line => line.split(" "))
counts.collect
val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1))
counts.collect
val counts = file.flatMap(line => line.split(" ")).map(word => (word,
1)).reduceByKey(_+_)
counts.collect.foreach(println)
```

Scenario: 7

- Multiple partitions

```
val file = sc.textFile("file:///usr/lib/spark/README.md",6)
val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_+_)
counts.collect.foreach(println)
```

Note: o/p would be same. Go to web UI and see DAG cycle and threads (Available for spark version 1.4 and above).

- For lower versions check with command

```
counts.toDebugString
```

Scenario: 8

- Fold - scala sample

```
val listRDD = sc.parallelize(List(1,5,8,6,3,4,5,9,10,2,3,4))
val maxByRddFold = listRDD.fold(Integer.MIN_VALUE)((accu, eachelement) =>accu max eachelement)
```

Scenario: 9

- FoldbyKey - Spark Shell

```
val a = sc.parallelize(List("kim","kumar","muthu","tim","lak","vamsi"),2)

val b = a.map(x => (x.length,x))
b.collect
b.foldByKey("")(_+_).collect
```

Scenario: 10

- Another example of FoldByKey

```
val deptEmployees = List (
("dept1",("kumar1",1000.0)),
("dept1",("kumar2",1200.0)),
("dept2",("kumar3",2200.0)),
("dept2",("kumar4",1400.0)),
("dept2",("kumar5",1000.0)),
("dept2",("kumar6",800.0)),
("dept1",("kumar7",2000.0)),
("dept1",("kumar8",1000.0)),
("dept1",("kumar9",500.0))
)
```

```
val employeeRDD = sc.makeRDD(deptEmployees)
val maxByDept =
employeeRDD.foldByKey(("dummy",Double.MinValue))((acc, element)
=> if(acc._2 > element._2)acc else element)

println("Maximum salaries in each dept" + maxByDept.collect().toList)
```

Scenario: 11

- ReducebyKey concept on wordcount example

```
val input = sc.textFile("file:///usr/lib/spark/README.md",3)
val words = input.flatMap(x =>x.split(" "))
// val result = words.map(x => (x,1)).reduceByKey((x,y) => x + y)

val result = words.map(x => (x,1)).reduceByKey((x,y) => {println("x "+x
+" :: "+"y "+y); x + y})
```

Scenario: 12

- Another example of ReduceByKey

```
val myRDD = sc.parallelize(Seq((1,"A"), (2,"B"), (2,"D"), (3,"C"), (3,"A"), (3,"B"), (3,"A")),1)
val resultRDD = myRDD.reduceByKey((x, y) => {println("x "+x +" :: "+"y "+y); x + y})
resultRDD.foreach(println)
```

Scenario: 13

- Another example of ReduceByKey

```
val myRDD = sc.parallelize(Seq(("A",1), ("B",2), ("D",4), ("C",2), ("A",1), ("B",2), ("A",1)))
myRDD.foreach(println)

val resultRDD = myRDD.reduceByKey((x, y) => {println("x "+x +" :: "+"y "+y); x + y})
```

Scenario: 14

- Line length count (Another reduce by key)

```
val input = sc.textFile("file:///usr/lib/spark/README.md")
val lengthCounts = input.map(line => (line.length, 1)).reduceByKey(_+_)
lengthCounts.foreach(println)
```

```
val lengthCounts = input.map(line => (line.length, 1)).reduceByKey(_+_)
lengthCounts.collect.foreach(println)
```

Scenario: 15

- Lookup

```
val deptEmployees = List (
("dept1",("kumar1",1000.0)),
("dept1",("kumar2",1200.0)),
("dept2",("kumar3",2200.0)),
("dept2",("kumar4",1400.0)),
("dept2",("kumar5",1000.0)),
("dept2",("kumar6",800.0)),
("dept1",("kumar7",2000.0)),
("dept1",("kumar8",1000.0)),
("dept1",("kumar9",500.0))
)
```

```
val employeeRDD = sc.makeRDD(deptEmployees)
employeeRDD.lookup("dept1")
```

Scenario: 16

- mapValues

```
employeeRDD.mapValues(":::" + _ + ":::").collect
employeeRDD.mapValues(_._2*3).collect
```

Scenario: 17

- collectAsMap

```
val myRDD = sc.parallelize(Seq((1,"A"), (2,"B"), (2,"D"), (3,"C"), (3,"A"),
(3,"B"), (3,"A")))
myRDD.collectAsMap()
```

Scenario: 18

- countByKey

```
myRDD.countByKey()
```

Scenario: 19

- PartitionBy(We can define a new type of partition with this)

```
val myRDD = sc.parallelize(List((1,"A"), (2,"B"), (2,"D"), (3,"C"), (3,"A"), (3,"B"), (3,"A")))
myRDD.partitioner
import org.apache.spark.HashPartitioner
val afterPartition = myRDD.partitionBy(new HashPartitioner(3))
afterPartition.partitioner
```

```
afterPartition.countByKey()
myRDD.countByKey()
```

Scenario: 20

- FlatMapValues

```
val myFlatMapRDD = sc.parallelize(List((1,"Apple"), (2,"Ball"), (2,"Dog"), (3,"Cat"), (3,"Ant")))
myFlatMapRDD.flatMapValues(_.toUpperCase).collect()
```

Scenario: 21

- mean

```
val a = sc.parallelize(List(4.1, 1.0, 1.2, 6.3, 1.3, 2.0, 2.1, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5))
val b = sc.parallelize(List(2,3,4,5,6))
a.mean
a.sum
a.variance
```

```
a.stats
```

- General RDD functions

```
val b = sc.parallelize(List(2,3,4,5,6,2,3,4))
```

Scenario: 22

- sample

```
val samplerange = sc.parallelize(1 to 1000, 3)
samplerange.sample(false, 0.1, 0).count
samplerange.sample(false, 0.1, 45).take(10)
samplerange.sample(false, 0.1, 345).take(10)
```

Scenario: 23

- union

```
val seta = sc.parallelize(1 to 10)
val setb = sc.parallelize(5 to 15)
(seta union setb).collect
seta.filter(_ % 2 == 0).collect
```

Scenario: 24

- groupBy

```
val a = sc.parallelize(1 to 15)
a.groupBy(x => { if (x % 2 == 0) "even" else "odd"}).collect
```

Scenario: 25

- groupByKey

```
val aaa = sc.parallelize(List("kim", "kumar", "muthu", "tim", "lak",
"vams"))
val foraaakey = aaa.keyBy(_.length)
val mycounter = foraaakey.map(x => (x._1,1))
mycounter.collect
```

```
foraaakey.groupByKey.collect
```

Scenario: 26

- reduceByKey

```
foraaakey.reduceByKey(_+_).collect
mycounter.reduceByKey(_+_).collect
```

Scenario: 27

- join

```
val a = sc.parallelize(List("kim", "kumar", "muthu", "tim", "lak",
"vamsi"))
val akeyby = a.keyBy(_.length)
```

```
akeyby.collect
```

```
val b = sc.parallelize(List("English", "Maths", "Tamil", "Science"))
val bkeyby = b.keyBy(_.length)
bkeyby.collect
```

```
akeyby.join(bkeyby).collect
```

Scenario: 28

- cartesian

```
val carta = sc.parallelize(List("kim", "kumar", "muthu", "tim", "lak", "vamsi"))
val cartb = sc.parallelize(List("English", "Maths", "Tamil", "Science"))
carta.cartesian(cartb).collect
carta.cartesian(cartb).count
```

Scenario: 29

- intersection

```
val setintera = sc.parallelize(1 to 10)
val setinterb = sc.parallelize(5 to 15)
(setintera intersection setinterb).collect
```

Scenario: 30

- coalesce

```
val acoalesce = sc.parallelize(1 to 15,3)
```

```
acoalesce.cache
acoalesce.collect
```

```
val onepart = acoalesce.coalesce(1)
```

```
onepart.cache // see at UI nothing in storage tab
onepart.collect // after executing this you will see a record in storage tab
```

Scenario: 31

- cogroup

```
val a = sc.parallelize(List("kim", "kumar", "muthu", "tim", "lak",
"vamsi"))
val akeyby = a.keyBy(_.length)
```

```
akeyby.collect
```

```
val b = sc.parallelize(List("English", "Maths", "Tamil", "Science"))
val bkeyby = b.keyBy(_.length)
bkeyby.collect
```

```
akeyby.cogroup(bkeyby).collect
```

Scenario: 32

- reduce

```
val ared = sc.parallelize(1 to 1000)
ared.reduce(_+_)
```

Scenario: 33

```
val aordered = sc.parallelize(List("kim", "kumar", "muthu", "tim", "lak",
"vamsi"))
aordered.takeOrdered(4)
```

Scenario: 34

- save as text file

```
aordered.saveAsTextFile("file:///home/training/spark_test")
```

Scenario: 35

- Broadcast variables

//defining class

```
case class Urls (url_id: Int, url_name: String)
case class Visits (visit_id: Int, url_id: Int, duration: Int)
```

//Load RDD of (id, url) pairs

```
val urlnames =
sc.textFile("file:///usr/lib/spark/pages.txt").map(_.split(" "))
urlnames.collect
val urlnames_recs = urlnames.map( r => (r(0).toInt,
Urls(r(0).toInt,r(1))))
urlnames_recs.collect
```

//Load RDD of (URL, visit) pairs

```
val visit_duration =
sc.textFile("file:///usr/lib/spark/visits_duration.txt").map(_.split(" "))
val visit_duration_recs = visit_duration.map( r => (r(1).toInt,
Visits(r(0).toInt, r(1).toInt,r(2).toInt)))
```

```
val joined = urlnames_recs.join(visit_duration_recs)
joined.collect
```

```
val joined_rev = visit_duration_recs.join(urlnames_recs)
joined_rev.collect
```

//Here sorting and shuffling happens over the network ref image

// alternative is to send the map to each executor along with the task

```
val urlnames_recs_map = urlnames_recs.collect.toMap
```

//toMap - array to Map conversion

```
urlnames_recs_map(1)
```

```
val altjoined = visit_duration_recs.map(v => (v._1,
(urlnames_recs_map(v._1), v._2)))
visit_duration_recs.first
```

```
altjoined.collect
```

```
// More optimized way is to use broadcast variable
// bc will get copied to each executor
val mybc = sc.broadcast(urlnames_recs_map)
```

```
val bcjoined = visit_duration_recs.map(v => (v._1, (mybc.value(v._1), v._2)))
```

```
bcjoined.collect
```

```
val joined_rev = visit_duration_recs.join(urlnames_recs)
val altjoined = visit_duration_recs.map(v => (v._1, (urlnames_recs_map(v._1), v._2)))
```

```
val bcjoined = visit_duration_recs.map(v => (v._1, (mybc.value(v._1), v._2)))
```

Scenario: 36

- Accumulator
- normal way

```
val myrangeacc = sc.parallelize(1 to 1000000, 5)
val myrangecount = myrangeacc.count
val mysum = myrangeacc.reduce(_+_)
val myaverage = mysum/myrangecount
```

```
// with accumulator
val sumacc = sc.accumulator(0)
val mycount = sc.accumulator(0)
myrangeacc.foreach(r => {
sumacc += r
```

```
mycount += 1
})
```

```
val myaverageacc = sumacc.value / mycount.value
```

Scenario: 37

- Join example

```
sc.setLocalProperty("spark.sql.shuffle.partitions","10")
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
//import sqlContext1.createSchemaRDD
```

```
import sqlContext.implicits._
```

```
case class Employee(name: String, sal: Int, city: String)
case class Email(name: String, email: String)
```

```
val employee = sc.textFile("file:///usr/lib/spark
/employee.txt").map(_.split(",")).map(p => Employee(p(0),
p(1).trim.toInt, p(2))).toDF()
val email = sc.textFile("file:///usr/lib/spark
/email.txt").map(_.split(",")).map(p => Email(p(0), p(1))).toDF()
```

```
employee.registerTempTable("employee1")
email.registerTempTable("email1")
```

```
val joineddata = sqlContext.sql("SELECT a.name, a.city, a.sal, b.email
FROM employee1 a JOIN email1 b on a.name = b.name")
joineddata.map(t => "Name: " + t(0) +" :: "+"City: " + t(1) + " :: " + "Sal:
" + t(2) + " :: " + "Email: " + t(3)).collect().foreach(println)
```

```
sqlContext.cacheTable("employee1")
sqlContext.cacheTable("email1")
```