**SPARK**

**Chapter – 1:**

**Introduction to Data Analysis with Spark:**

**Spark:** Apache Spark is an in-memory cluster computing technology designed for fast computing.

Spark is designed to be highly accessible, it offers simple APIs in Python, Scala, Java and SQL and in-built libraries.

At its core, Spark is a “computational engine” that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a computing cluster.

The spark stack consists of following:

Spark Core, Spark SQL, Spark Streaming, MLlib and GraphX.

**Spark Core:**

It contains basic functionality of Spark like task scheduling, memory management, fault tolerance, interacting with storage systems, etc.

Spark main programming abstraction is RDD (Resilient Distributed Datasets).

**Spark SQL:**

It is a Spark’s package to work with structured data. In it also supports HQL(Hive Query Language).

And it supports different sources of data like parquet, hive tables, json, avro, csv.

**Spark Streaming:**  It allows us to process the real time data. Eg: Web server logs generated continuously.

**MLlib:** Spark has a library, which provides us the functionality of Machine Learning called MLlib.

It provides machine learning algorithms like classification, regression, clustering etc..

**Cluster Managers:**  Spark can run over a variety of cluster managers including Hadoop Yarn, Mesos.

It can run on even Stand-alone mode.

**Chapter-2:**

**Getting started:**

To open the scala version of shell we need to go to bin directory of spark.

Then execute the following command

./spark-shell

**Counting lines in a text file**

scala>val numoflines = sc.textFile(“filename”) // RDD will be created

scala> numoflines.count() // counts the num of lines and displays on console

scala> numoflines.first()// displays first line in the file

**Filtering:**

scala> val lines = sc.textFile("README.md")

scala> val pythonLines = lines.filter(line => line.contains("Hadoop"))

scala> pythonLines.first()

**chapter-3:**

**Programming with RDD’s:**

RDD is a distributed collection of objects. It is an immutable.

**Creating RDDs:**

We can create RDDs in 2 ways:

* loading an external dataset
* parallelizing a collection in your driver program.

Creating RDD by loading external dataset:

val input = sc.textFile(“filepath”)

Using parallelize method:

val p= sc.parallelize(List(“flowers”, “i like flowers” ))

**RDD operations:**

There are 2 types of operations.

* Transformations
* Actions: Actions are operations that return a result to the driver program or write it to storage.

**Transformations:**

These are operations on RDD’s, return new RDD’s.

These transformed RDD’s are lazily evaluated means they are evaluated only those are used in actions.

**filter() transformation:**

val inputRDD = sc.textFile("log.txt")

val errorsRDD = inputRDD.filter(line => line.contains("error"))

**Actions:**

Actions are operations that return a result to the driver program or write it to storage.

When an action is triggered RDDs are evaluated.

Eg: count(), take(), etc..

**Lazy Evaluation:**

Transformations on RDDs are lazily evaluated, means Spark will not begin to execute until it sees an action.

Loading data into an RDD is lazily evaluated in the same way transformations are.

When we call sc.textFile(), the data is not loaded until it is necessary.

**Basic RDDs:**

Element-wise transformations:

The two most common transformations are map() and filter().

The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

The filter() transformation takes in a function and returns an RDD that only has elements that pass the filter() function.

Some examples of transformations are

rdd.distinct()

rdd1.union(rdd2)

rdd1.intersection(rdd2)

rdd1.subtract(rdd2)

rdd1.cartesian(rdd2)

**chapter-3:**

**Programming with RDD’s:**

RDD is a distributed collection of objects. It is an immutable.

**Creating RDDs:**

We can create RDDs in 2 ways:

* loading an external dataset
* parallelizing a collection in your driver program.

Creating RDD by loading external dataset:

val input = sc.textFile(“filepath”)

Using parallelize method:

val p= sc.parallelize(List(“flowers”, “i like flowers” ))

**RDD operations:**

There are 2 types of operations.

* Transformations
* Actions: Actions are operations that return a result to the driver program or write it to storage.

**Transformations:**

These are operations on RDD’s, return new RDD’s.

These transformed RDD’s are lazily evaluated means they are evaluated only those are used in actions.

**filter() transformation:**

val inputRDD = sc.textFile("log.txt")

val errorsRDD = inputRDD.filter(line => line.contains("error"))

**Actions:**

Actions are operations that return a result to the driver program or write it to storage.

When an action is triggered RDDs are evaluated.

Eg: count(), take(), etc..

**Lazy Evaluation:**

Transformations on RDDs are lazily evaluated, means Spark will not begin to execute until it sees an action.

Loading data into an RDD is lazily evaluated in the same way transformations are.

When we call sc.textFile(), the data is not loaded until it is necessary.

**Basic RDDs:**

Element-wise transformations:

The two most common transformations are map() and filter().

The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

The filter() transformation takes in a function and returns an RDD that only has elements that pass the filter() function.

Some examples of transformations are

rdd.distinct()

rdd1.union(rdd2)

rdd1.intersection(rdd2)

rdd1.subtract(rdd2)

rdd1.cartesian(rdd2)

**Chapter-4:**

**Key/value pairs in spark:**

**Paired RDDs:**

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs.

Creating a pair RDD using the first word as the key in Scala

val pairs = lines.map(x => (x.split(" ")(0), x))

**Transformations using one pair RDD:**

rdd.reduceByKey( (x, y) => x + y)

rdd.groupByKey()

rdd.mapValues(x => x+1)

rdd.flatMapValues(x => (x to 5))

rdd.keys()

rdd.values()

rdd.sortByKey()

**Transformations using two pair RDD:**

rdd.subtractByKey(other)

rdd.rightOuterJoin(other)

rdd.join(other)

rdd.leftOuterJoin(other)

**Word count job in scala:**

val input = sc.textFile("s3://...")

val words = input.flatMap(x => x.split(" "))

val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)

**Tuning the level of parallelism:**

Every RDD has a fixed number of partitions that determine the degree of parallelism to use when executing operations on the RDD.

We can increase the number of partitions of an RDD. This we call repartitioning.

val data = Seq(("a", 3), ("b", 4), ("a", 1))

sc.parallelize(data).reduceByKey((x, y) => x + y) // Default parallelism sc.parallelize(data).reduceByKey((x, y) => x + y, 10) // Custom parallelism

Spark also has an optimized version of repartition() called coalesce() that allows avoiding data movement, but only if you are decreasing the number of RDD partitions.

To check the size of the RDD in scala/java

rdd.partitions.size()

**Actions on paired RDDs:**

rdd.collectAsMap()--- collect the result as map

rdd.countByKey()--- returns number of values related to each key

rdd.lookup(key) --- gives values related to that key