What is Spark?

Apache Spark is a general-purpose & lightning fast cluster computing system. It provides a high-level API. For example, Java, Scala, Python, and R. Apache Spark is a tool for Running Spark Applications. Spark is 100 times faster than Bigdata Hadoop and 10 times faster than accessing data from disk.

Spark is written in Scala but provides rich APIs in Scala, Java, Python, and R.
It can be integrated with Hadoop and can process existing Hadoop HDFS data. Follow this guide to learn How Spark is compatible with Hadoop?
It is saying that the images are the worth of a thousand words. To keep this in mind we have also provided Spark video tutorial for more understanding of Apache Spark.

Why Spark?

- Hadoop MapReduce can only perform batch processing.
- Apache Storm / S4 can only perform stream processing.
- Apache Impala / Apache Tez can only perform interactive processing
- Neo4j / Apache Giraph can only perform graph processing

Hence in the industry, there is a big demand for a powerful engine that can process the data in real-time (streaming) as well as in batch mode. There is a need for an engine that can respond in sub-second and perform in-memory processing.
Apache Spark Definition says it is a powerful open-source engine that provides real-time stream processing, interactive processing, graph processing, in-memory processing as well as batch processing with very fast speed, ease of use and standard interface.

Apache Spark Features:

- Speed: Speed always matters for processing data, organizations want to process voluminous data as fast as possible. Spark is Lightning fast processing tool makes it speedier to handle complex processing. As it follows the concept of RDD (Resilient Distributed Dataset) which allows it to store data transparently in memory, which helps in reducing read & write to disc one of the main time-consuming factor.

- Usability: Ability to support multiple languages makes it dynamic. It allows you to quickly write an application in Java, Scala, Python, and R.

- In-Memory Computing: Keeping data in servers' RAM as it makes accessing stored data quickly. In memory, analytics accelerates iterative machine learning algorithms as it saves data read and write round trip from/to disk.

- Pillar to Sophisticated Analytics: Spark comes with tools for interactive/declarative queries, streaming data, machine learning which is an addition to the simple map and reduces, so that users can combine all this into the single workflow.

- Real-Time Stream Processing: Spark streaming can handle real-time stream processing along with the integration of other frameworks which concludes that spark's streaming ability is easy, fault tolerance and Integrated.

- Compatibility with Hadoop & existing Hadoop Data: Spark is compatible with both versions of the Hadoop ecosystem. Be it YARN (Yet Another Resource Negotiator) or SIMR (Spark in MapReduce). It can read anything existing Hadoop data that's what makes it suitable for migration of pure Hadoop-MapReduce applications. It can run independently too.

- Lazy Evaluation: Another outstanding feature of Spark which is called by need or memorization. It waits for instructions before providing the final result which saves significant time.

- Active, progressive and expanding community

Apache Spark Components

- Spark Core
- Spark SQL
- Spark Streaming
- Spark MLlib
- Spark GraphX
- SparkR

Spark Concepts

1) SparkContext :

SparkContext is the entry point of Spark functionality. The most important step of any Spark driver application is to generate SparkContext. It allows your Spark Application to access Spark Cluster with the help of Resource Manager. The resource manager can be one of these three- Spark Standalone, YARN, Apache Mesos.

2) RDD:

RDD stands for “Resilient Distributed Dataset”. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

- Resilient, i.e. fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
- Distributed, since Data resides on multiple nodes.
- Dataset represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Features of Spark RDD

There are several advantages of using RDD. Some of them are-

- In-memory computation
- Lazy Evaluation
- Fault Tolerance
- Immutability
- Persistence
- Partitioning
- Parallel
- Location-Stickiness
- Coarse-grained Operation
- Typed -We can have RDD of various types like: RDD [int], RDD [long], RDD [string].
- No limitation

3) DAG

DAG a finite direct graph with no directed cycles. There are finitely many vertices and edges, where each edge directed from one vertex to another. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence. It is a strict generalization of MapReduce model. DAG operations can do better global optimization than other systems like MapReduce. The picture of DAG becomes clear in more complex jobs.
Each stage is comprised of tasks, based on the partitions of the RDD, which will perform same computation in parallel. The graph here refers to navigation, and directed and acyclic refers to how it is done.

How DAG works in Spark?

- The interpreter is the first layer, using a Scala interpreter, Spark interprets the code with some modifications.
- Spark creates an operator graph when you enter your code in Spark console.

- When we call an Action on Spark RDD at a high level, Spark submits the operator graph to the DAG Scheduler.
- Divide the operators into stages of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators schedule in a single stage.
- The stages pass on to the Task Scheduler. It launches task through cluster manager. The dependencies of stages are unknown to the task scheduler.
- The Workers execute the task on the slave.

4) Transformation

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

Applying transformation built an RDD lineage, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as RDD operator graph or RDD dependency graph. It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).

Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

How to create RDD

There are two popular ways using which you can create RDD in Apache Spark.

First is Parallelize and other is text File method. Here is quick explanation how both methods can be used for RDD creation.

```
val a= Array(5,7,8,9)
val b= sc.parallelize(a)
val c = sc.textFile(“demo.txt”);
```

Example 1: Even Odd Number Filter

```
object EvenOddFilterTest {

  def main(args: Array[String]): Unit = {

          var sparkConfig = new
SparkConf().setMaster("local").setAppName("OddEventFilterTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")
          sparkContext.setJobDescription("The program design to filter odd and even no")

          var item: Range = 1 to 10
          var data = sparkContext.parallelize(item)

          var evenRDD = data.filter((key)=> key % 2 == 0)
          println("Printing Even nuber")
          evenRDD.collect().foreach(key=>{
          println(s"Collected Even Numbers Are: $key")
          })

          var oddRDD= data.filter(key=>key%2!=0)
          println("Printing Odd Number")
          oddRDD.collect().foreach(key=>{
          println(s"Collected Odd Numbers Are: $key")
          })
  }
}
```

Example 2: Even Odd Number Count

```
object EvenOddNumberCountTest {

  def main(args: Array[String]): Unit = {

          var sparkConfig = new
SparkConf().setMaster("local").setAppName("EvenOddNumberCountTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setJobDescription("This is to determine no of odd and even number")
          sparkContext.setLogLevel("info")
```

```
        //var data =
sparkContext.textFile("file:///home/debdutta/HadoopEnviornment/data/input/number.txt")
        var data = sparkContext.textFile("hdfs://localhost:9000/user/root/input/number.txt")
        var dataRDD = data.flatMap(line => line.split(","))
        var numberRDD = dataRDD.map(item => item.toInt)

        var evenRDD = numberRDD.filter(_ % 2 == 0)
        println("Printing Even No from File")
        evenRDD.collect().foreach(key => {
        println(s"Collected Even Numbers Are: $key")
        })

        var oddRDD = numberRDD.filter(key => key % 2 != 0)
        println("Printing Odd Number")
        oddRDD.collect().foreach(key => {
        println(s"Collected Odd Numbers Are: $key")
        })

        println("Finding Counts")
        var evenMapRDD = evenRDD.map(number => (number, 1))
        var evenCountRDD = evenMapRDD.reduceByKey((sum, b) => sum + b)
        evenCountRDD.collect().foreach(key => {
        println(s"Collected Even Count Are: $key")
        })

        println("Sorting Elements")
        var sortedRDD = evenMapRDD.sortByKey()
        sortedRDD.collect().foreach(key => {
        println(s"Collected Sorted Even Number Are: $key")
        })
  }
}
```

Example 3: Word Count

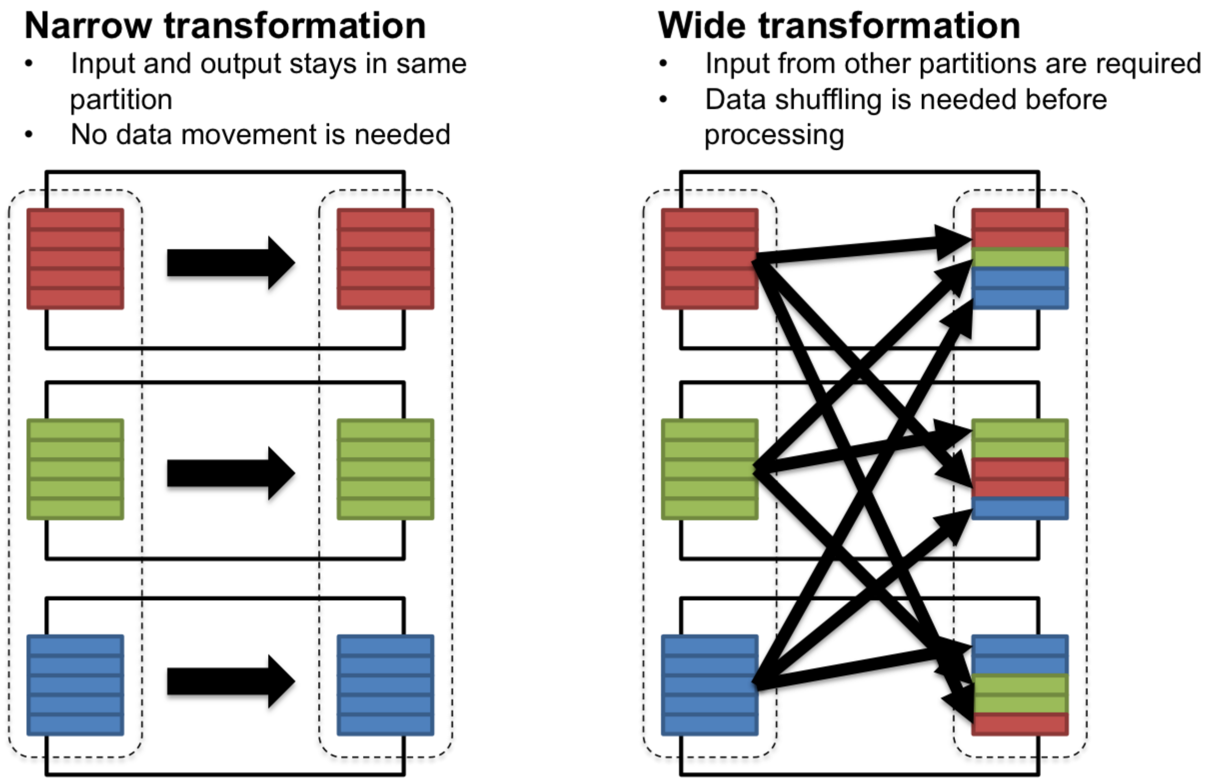
```
object WordCountTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("WordCountTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")
        var textRDD =
sparkContext.textFile("file:///home/debdutta/HadoopEnviornment/data/input/demo.txt")
        var lineRDD = textRDD.flatMap(line => line.split(" "))

        var wordRDD = lineRDD.map(word => (word, 1))
        var wordCountRDD = wordRDD.reduceByKey((sum, b) => sum + b)
        wordCountRDD.collect().foreach(key => {
        println(s"Collected Words Are: $key")
        })

        println("Coding in Shorts")
        var shortWordCountRDD =
sparkContext.textFile("file:///home/debdutta/HadoopEnviornment/data/input/demo.txt")
        .flatMap(_.split(" ")).map((_, 1)).reduceByKey(_ + _).collect().foreach(println)
  }
}
```

Working With RDD:

There are two types of transformations:

Narrow transformation – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of map(), filter().

Wide transformation – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. Wide transformations are the result of groupbyKey() and reducebyKey().


Narrow transformation
• Input and output stays in same partition
• No data movement is needed
Wide transformation
• Input from other partitions are required
• Data shuffling is needed before processing

Paired RDDs
Spark Paired RDDs are nothing but RDDs containing a key-value pair. Basically, key-value pair (KVP) consists of a two linked data item in it.
We can use map method to create paired RDDs

```
 var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop"))
  dataRDD.collect()
var resultRDD = dataRDD.map(_.toUpperCase())
println("Printing Result")
resultRDD.foreach(println)
```

<u>Partitions</u>:

Basically there are two types of Partitioning in spark:

- **Hash Partitioning in Spark**
  Hash Partitioning attempts to spread the data evenly across various partitions based on the **key. Object.hashCode** method is used to determine the partition in Spark as partition = key.hashCode () % numPartitions.

- **Range Partitioning in Spark**

Some Spark RDDs have keys that follow a particular ordering, for such RDDs, range partitioning is an efficient partitioning technique. In range partitioning method, tuples having keys within the same range will appear on the same machine. Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

Example: Hash Partitions

```
object HashPartionerTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new
SparkConf().setMaster("local").setAppName("HashPartionerTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(1 to 10).map((_, 1)).partitionBy(new
HashPartitioner(3))
        dataRDD.partitioner
        dataRDD.glom().collect()

dataRDD.saveAsTextFile("file:///home/debdutta/HadoopEnviornment/data/output/hashpartitoner"
)
  }
}
```

Example Range Partition:

```
object RangePartitionerTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new
SparkConf().setMaster("local").setAppName("HashPartionerTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(1 to 10).map((_, 1))
        var resultRDD = dataRDD.partitionBy(new RangePartitioner(3, dataRDD))
        resultRDD.glom().collect()

resultRDD.saveAsTextFile("file:///home/debdutta/HadoopEnviornment/data/output/rangepartiton
er")
  }
}
```

Example Custom Partitions:

```
class CustomPartitioner(noOfPartitioner: Int) extends Partitioner {

  override def numPartitions: Int = noOfPartitioner

  override def getPartition(key: Any): Int = {
          key.toString().toInt % noOfPartitioner
  }

  override def equals(other: Any): Boolean = {
          other match {
          case obj: CustomPartitioner =>
          obj.numPartitions == numPartitions
          case _ => false
          }
  }
}

object CustomPartitionerTest {

  def main(args: Array[String]): Unit = {

          var sparkConfig = new
SparkConf().setMaster("local").setAppName("HashPartionerTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")

          var dataRDD = sparkContext.parallelize(1 to 10).map((_, 1)).partitionBy(new
CustomPartitioner(3))
          dataRDD.partitioner
          dataRDD.glom().collect()

dataRDD.saveAsTextFile("file:///home/debdutta/HadoopEnviornment/data/output/custom")
  }
}
```

Spark Transformation Example:

1) Map:

```
object MapTest {

  def main(args: Array[String]): Unit = {
        var sparkConfig = new SparkConf().setMaster("local").setAppName("MapTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop"))
        dataRDD.collect()
        var resultRDD = dataRDD.map(_.toUpperCase())
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

2) MapValues:

```
object MapValueTest {

  //Paird RDDs- Works Similar to Map
  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("MapValueTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10)))
        dataRDD.collect()
        var resultRDD = dataRDD.mapValues(item => item * 10)
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

3) FlatMap

```
object MapValueTest {

  //Paird RDDs- Works Similar to Map
  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("MapValueTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10)))
        dataRDD.collect()
        var resultRDD = dataRDD.mapValues(item => item * 10)
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

4) FlatMapValue:

```
object FlatMapValueTest {

  //Paird RDDs- Works Similar to Flatmap

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("FlatMapValueTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10)))
        var resultRDD = dataRDD.flatMapValues(item => Array(item * 10)).collect()
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

5) Filter:

```
object FilterTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("FilterTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go"))
        var resultRDD = dataRDD.filter(item => item.length() > 3)
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

6) GroupBy:

```
object GroupByTest {

  def main(args: Array[String]): Unit = {
        var sparkConfig = new SparkConf().setMaster("local").setAppName("GroupByTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go",
"Scala", "Spark", "Android", "J2ME"))
        var resultRDD = dataRDD.groupBy(item => item.charAt(0))
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

7) GroupByKey:

```
object GroupByKeyTest {

  //Paird RDDs- Group based on key
  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("GroupByKeyTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10),
("Java", 40), ("Spring", 15), ("Scala", 25)))
        var resultRDD = dataRDD.groupByKey()
        println("Printing Result")
        resultRDD.foreach(println)
  }
}
```

8) Sort By:

```
object SortByTest {

   def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("SortByTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go",
"Scala", "Spark", "Android", "J2ME"))
        var resultRDD = dataRDD.sortBy(_.charAt(0))
```

```
          println("Printing Result")
          resultRDD.foreach(println)
  }
}
```

9) SortByKey:

```
object SortByKeyTest {

  //Paird RDDs- Sort based on key
  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("SortByKeyTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")

          var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10),
("HBase", 40), ("Go", 15), ("Scala", 25)))
          var resultRDD = dataRDD.sortByKey()
          println("Printing Result")
          resultRDD.foreach(println)
  }
}
```

10) ReduceByKey:

```
object ReduceByKeyTest {

  //Paird RDDs- Combine values based on key
  def main(args: Array[String]): Unit = {

          var sparkConfig = new
SparkConf().setMaster("local").setAppName("ReduceByKeyTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")

          var dataRDD = sparkContext.parallelize(List(("Java", 20), ("Spring", 30), ("Scala", 10),
("Java", 40), ("Spring", 15), ("Scala", 25)))
          var resultRDD = dataRDD.reduceByKey(_+_)
          println("Printing Result")
          resultRDD.foreach(println)
  }
}
```

11) Distinct:

```
object DistinctTest {
  //Remove Duplicate Data
  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("DistinctTest")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")

          var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "Java", "Go",
"Hadoop", "Spark", "Android", "J2ME"))
          var resultRDD = dataRDD.distinct()
          println("Printing Result")
          resultRDD.foreach(println)
  }
}
```

12) KeyBy:

```
object KeyByExample {

  //Convert Into Paired RDD By Executing Function
  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("KeyByExample")
          var sparkContext = new SparkContext(sparkConfig)
          sparkContext.setLogLevel("info")

          var techRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go",
"Scala", "Spark", "Android", "J2ME"))
          var pairedData = techRDD.keyBy(_.charAt(0))
          println(s"Key By Example: ")
          pairedData.foreach(item => {
          println(s"Key By Data : ${item._1} \t  ${item._2}")
          })
  }
}
```

13) Partitions:

```
object PartitionTest {

  //Iterates over every partition
  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("PartitionTest")
```

```
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go"), 3)
        var mapPartitionsRDD = dataRDD.mapPartitions(x => x.map(y => (y, 1)))
        mapPartitionsRDD.glom.collect()
        println("Partition Example with mapPartitions: Iterates over every partition ")
        println(s"Printing Partition Result ${mapPartitionsRDD.foreach(println)}")

        println("Coalesce Example: Reduce Partitions")
        var coalesceRDD = dataRDD.coalesce(2)
        println(s"Updating Partition Result ${coalesceRDD.foreach(println)}")

        println("Repartition Example")
        var repartitionRDD = dataRDD.coalesce(4)
        println(s"Repartition Result ${repartitionRDD.foreach(println)}")
  }
}
```

<u>14) Join:</u>

```
object JoinTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("JoinTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(Array(("JA", "Java"), ("HD", "Hadoop"), ("HB",
"HBase")))
        var dataRDD1 = sparkContext.parallelize(Array(("SP", "Spark"), ("SC", "Scala"), ("HB",
"HBase")))
        println("Join Example: Return Matching Data")
        var resultRDD = dataRDD.join(dataRDD1).collect()
        resultRDD.foreach(println)

        println("Left OuterJoin Example: Return Matching data and First RDD data")
        var resultRDD1 = dataRDD.leftOuterJoin(dataRDD1).collect()
        resultRDD1.foreach(println)

        println("Right OuterJoin Example: Return Matching data and Second RDD data")
        var resultRDD2 = dataRDD.rightOuterJoin(dataRDD1).collect()
        resultRDD2.foreach(println)
        println("Full OuterJoin Example: Return both 1st & 2nd RDD data including matching
ones")
        var resultRDD3 = dataRDD.fullOuterJoin(dataRDD1).collect()
        resultRDD3.foreach(println)
  }
}
```

<u>15) Union, Intersect, Substract:</u>

```
object UISTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("UISTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var rdd1 = sparkContext.parallelize(List(15, 8, 10, 2, 4))
        var rdd2 = sparkContext.parallelize(List(15, 25, 8, 19, 11))

        //Example 1
        println("******************************")
        println("Example 1: Union Example: Add both RDD")
        var unionRDD = rdd1.union(rdd2)
        println("Union Result")
        unionRDD.foreach(println)

        //Example 2
        println("******************************")
        println("Example 2: Intersection Example: Return Common RDD")
        var intersectionRDD = rdd1.intersection(rdd2)
        println("Intersection Result")
        intersectionRDD.foreach(println)

        //Example 3
        println("******************************")
        println("Example 3: Subtract Example: Return value which are not present in 2nd RDD")
        var subtractRDD = rdd1.subtract(rdd2)
        println("Subtract Result")
        subtractRDD.foreach(println)

        //Example 4
        println("******************************")
        println("Example 4: Subtract By Key Example:Return value which are not present in 2nd
RDD Based on Key")
        var techRDD = sparkContext.parallelize(List(("Java", 30), ("Spring", 30), ("Scala", 10)))
        var dbRDD = sparkContext.parallelize(List(("Java", 20), ("Oracle", 30), ("Mysql", 10)))
        var subtractByKeyRDD = techRDD.subtractByKey(dbRDD)
        println("Subtract By Key Result")
        subtractByKeyRDD.foreach(println)
  }
}
```

Spark Action Example:

```
object ActionTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("ActionTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var dataRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go",
"Scala", "Spark", "Android", "J2ME"), 4)
        var techRDD = sparkContext.parallelize(List("Java", "HBase", "Hadoop", "C", "Go",
"Scala", "Spark", "Android", "J2ME"))
        var digitRDD = sparkContext.parallelize(List(10, 20, 90, 55, 1, 77))
        var pairedData = techRDD.keyBy(_.charAt(0))

        println(s"No Of Partition: ${dataRDD.getNumPartitions}")
        println(s"Partition Size: ${dataRDD.partitions.size}")

        println(s"Collect Example: ${dataRDD.collect().foreach(println)}")
        println(s"Glom Collect TO Show Partition Data Example:
${dataRDD.glom.collect().foreach(println)}")
        println(s"Collect As Map: Convert Paired RDD into Map: ${pairedData.collectAsMap()}")

        println(s"Printing First 2 Elements using take ${techRDD.take(2).foreach(println)}")
        println(s"Printing Top 2 Elements using top ${techRDD.top(2).foreach(println)}")

        println(s"Reduce Using Action: ${digitRDD.reduce(_ + _)}")
        println(s"Max Element: ${digitRDD.max}")
        println(s"Min Element: ${digitRDD.min}")
        println(s"Sum Element: ${digitRDD.sum}")
        println(s"Mean Element: ${digitRDD.mean}")
        println(s"Count Element: ${digitRDD.count}")

        println(s"Count By Key Example: ${pairedData.countByKey()}")
        println(s"Count By Value Example: ${pairedData.countByValue()}")

        //Others foreach & SaveAsTextFile

pairedData.saveAsTextFile("file:///home/debdutta/HadoopEnviornment/data/spark/output/result.t
xt")
  }
}
```

What is RDD Persistence and Caching in Spark?

Spark RDD persistence is an optimization technique in which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.

We can make persisted RDD through cache() and persist() methods. When we use the cache() method we can store all the RDD in-memory. We can persist the RDD in memory and use it efficiently across parallel operations.

The difference between cache() and persist() is that using cache() the default storage level is MEMORY_ONLY while using persist() we can use various storage levels (described below). It is a key tool for an interactive algorithm. Because, when we persist RDD each node stores any partition of it that it computes in memory and makes it reusable for future use. This process speeds up the further computation ten times.

When the RDD is computed for the first time, it is kept in memory on the node. The cache memory of the Spark is fault tolerant so whenever any partition of RDD is lost, it can be recovered by transformation Operation that originally created it.

Storage levels of Persisted RDDs

Using persist() we can use various storage levels to Store Persisted RDDs in Apache Spark.

a) MEMORY_ONLY In this storage level, RDD is stored as deserialized Java object in the JVM. If the size of RDD is greater than memory, It will not cache some partition and recompute them next time whenever needed. In this level the space used for storage is very high, the CPU computation time is low, the data is stored in-memory. It does not make use of the disk.

b) MEMORY_AND_DISK: In this level, RDD is stored as deserialized Java object in the JVM. When the size of RDD is greater than the size of memory, it stores the excess partition on the disk, and retrieve from disk whenever required. In this level the space used for storage is high, the CPU computation time is medium, it makes use of both in-memory and on disk storage.

c) MEMORY_ONLY_SER: This level of Spark store the RDD as serialized Java object (one-byte array per partition). It is more space efficient as compared to deserialized objects, especially when it uses fast serializer. But it increases the overhead on CPU. In this level the storage space is low, the CPU computation time is high and the data is stored in-memory. It does not make use of the disk.

d) MEMORY_AND_DISK_SER: It is similar to MEMORY_ONLY_SER, but it drops the partition that does not fits into memory to disk, rather than recomputing each time it is needed. In this storage level, The space used for storage is low, the CPU computation time is high, it makes use of both in-memory and on disk storage.

e) DISK_ONLY: In this storage level, RDD is stored only on disk. The space used for storage is low, the CPU computation time is high and it makes use of on disk storage.

f) MEMORY_ONLY_2 - Replication factor is 2

g) MEMORY_AND_DISK- Replication factor is 2

Cache Example:

```
object CacheTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("BroadcastTest")
        var sparkContext = new SparkContext(sparkConfig)
        sparkContext.setLogLevel("info")

        var itemRDD= sparkContext.parallelize(1 to 10)
        itemRDD.cache()
        itemRDD.persist()
        println(s"Result: ${itemRDD.count()}")
  }
}
```

Broadcast Variables:

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

```
object BroadcastTest {


  def main(args: Array[String]): Unit = {

            var sparkConfig = new SparkConf().setMaster("local").setAppName("BroadcastTest")
            var sparkContext= new SparkContext(sparkConfig)
            sparkContext.setLogLevel("info")

            var itemRDD= sparkContext.parallelize(List(1,2,3))
            var brodcastVal= sparkContext.broadcast(2)

            println("Printing Result")
            var resultRDD= itemRDD.map(key=>brodcastVal.value+key)
            resultRDD.collect().foreach(println)

  }
}
```

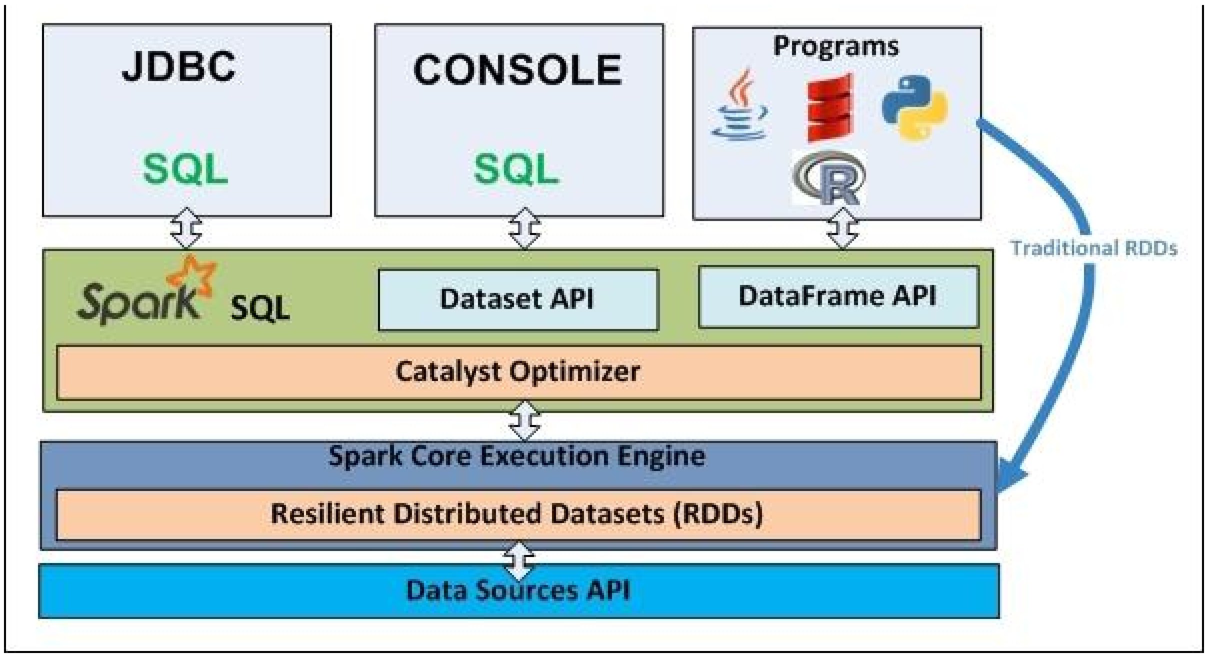
Accumulators:

Accumulate Values

```
object AccumulatorsTest {
   def main(args: Array[String]): Unit = {

            var sparkConfig = new SparkConf().setMaster("local").setAppName("BroadcastTest")
            var sparkContext= new SparkContext(sparkConfig)
            sparkContext.setLogLevel("info")

            var accum= sparkContext.accumulator(0, "TestAccume")
            var dataRDD= sparkContext.parallelize(List(1,2,3))
            dataRDD.foreach(x=>accum+=x)
            println(s"The result is: ${accum.value}")
  }
}
```

Spark SQL

Architecture:



What is Spark SQL DataFrame?

DataFrame appeared in Spark Release 1.3.0. We can term DataFrame as Dataset organized into named columns. DataFrames are similar to the table in a relational database or data frame in R /Python. It can be said as a relational table with good optimization technique.

What is Spark Dataset?

Dataset is a data structure in SparkSQL which is strongly typed and is a map to a relational schema. It represents structured queries with encoders. It is an extension to data frame API. Spark Dataset provides both type safety and object-oriented programming interface. We encounter the release of the dataset in Spark 1.6.

Spark SQL Optimization

The term optimization refers to a process in which a system is modified in such a way that it work more efficiently or it uses fewer resources."

Spark SQL is the most technically involved component of Apache Spark. Spark SQL deals with both SQL queries and DataFrame API. In the depth of Spark SQL there lies a catalyst optimizer. Catalyst optimization allows some advanced programming language features that allow you to build an extensible query optimizer.

Catalyst Optimizer supports both rule-based and cost-based optimization. In rule-based optimization the rule based optimizer use set of rule to determine how to execute the query. While the cost based optimization finds the most suitable way to carry out SQL statement. In cost-based optimization, multiple plans are generated using rules and then their cost is computed.

Need of Catalyst Optimizer

There are two purposes behind Catalyst's extensible design:

- We want to add the easy solution to tackle various problems with Bigdata like a problem with semi-structured data and advanced data analytics.
- We want an easy way such that external developers can extend the optimizer.

Fundamentals of Catalyst Optimizer

Catalyst optimizer makes use of standard features of Scala programming like pattern matching. In the depth, Catalyst contains the tree and the set of rules to manipulate the tree. There are specific libraries to process relational queries. There are various rule sets which handle different phases of query execution like analysis, query optimization, physical planning, and code generation to compile parts of queries to Java bytecode.

- **Trees -** A tree is the main data type in the catalyst. A tree contains node object. For each node, there is a node. A node can have one or more children. New nodes are defined as subclasses of TreeNode class. These objects are immutable in nature. The objects can be manipulated using functional transformation.

- **Rules -** We can manipulate tree using rules. We can define rules as a function from one tree to another tree. With rule we can run arbitrary code on input tree, the common approach to use a pattern matching function and replace subtree with a specific structure. In a tree with the help of transform function, we can recursively apply pattern matching on all the node of a tree. We get the pattern that matches each pattern to a result.

Spark SQL Execution Plan

- Analysis- Looks at the query/Dataframe and create a logical pla,
- Logical Optimization- Rule based optimization is applied on logical plan.
- Physical planning- Takes logical plan and compute one or more physical plan. Compute cost of each physical plan & select the the physical plan based on cost model.
- Code generation-Java bytecode will be generated for each machine.

Example 1: Loading Data from Json

```
object ReadJsonTest {
  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("ReadJsonTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        var studentDF =
sqlContext.read.json("file:///home/debdutta/HadoopEnviornment/data/spark/input/student.json")
        studentDF.registerTempTable("STUDENT")
        var studentDF1 =
sqlContext.read.json("file:///home/debdutta/HadoopEnviornment/data/spark/input/student.json")
        studentDF1.registerTempTable("STUDENT1")

        println("Example 1- Run Raw Query")
        var result = sqlContext.sql("SELECT * FROM STUDENT")
        result.show

        println("Example 2")
        result = sqlContext.sql("SELECT ID,NAME FROM STUDENT")
        result.show

        println("Example 3- Print Schema")
        studentDF.printSchema()

        println("Example 4- Select Statement")
        studentDF.select(studentDF("NAME"), studentDF("CITY")).show

        println("Example 5- Sort By Statement")
        studentDF.sort(studentDF("CITY")).show

        println("Example 6- Filter Statement")
        studentDF1.filter(studentDF1("ID") >2).show
  }
}
```

Example 2: Loading Data from CSV

```
object ReadCSVFiles {

  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("ReadCSVFiles")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        var studentDF =
sqlContext.read.csv("file:///home/debdutta/HadoopEnviornment/data/spark/input/student.csv")
        println("Showing Student Data")
        studentDF.show()

        println("Showing Student Count")
        println(s"Total Count ${studentDF.count()}")

        println("Create Table Using Code")
        var studentSchema = new StructType().add("id", "int").add("name", "String")
        var studentDF1 =
sqlContext.read.format("com.databricks.spark.csv").schema(studentSchema).load("file:///home/
debdutta/HadoopEnviornment/data/spark/input/student.csv")
        studentDF1.show
  }
}
```

Example 3: Case class Example

```
object CaseTest {

  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("CaseTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        import sqlContext.implicits._
        var file =
sparkContext.textFile("file:///home/debdutta/HadoopEnviornment/data/spark/input/stu.csv")
        var rows = file.map(_.split(","))
        var studentRDD = rows.map(s => Student(s(0).toInt, s(1), s(2)))
        var studentDF = studentRDD.toDF()
        var studentDS = studentDF.as[Student] //This is Data Source
        studentDS.show
  }
}
```

```
case class Student(id: Int, name: String, city: String)
```

Example 4: Loading data from Relational Database

```
object DBTest {

  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("DBTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        val url = "jdbc:mysql://localhost:3306/play"
        val prop = new java.util.Properties
        prop.setProperty("user", "root")
        prop.setProperty("password", "Deb123")

        var studentDF = sqlContext.read.jdbc(url, "STUDENT", prop)
        studentDF.show()
  }
}
```

Example 5: Data Set Example

```
object DataSetTest {

  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("DataSetTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        import sqlContext.implicits._
        var studentDS =
sqlContext.read.json("file:///home/debdutta/HadoopEnviornment/data/spark/input/student.json").
as[Stu]
        studentDS.show()
  }
}

case class Stu(id: String, name: String, city: String)
```

Example 6: User Defined Function in Spark SQL

```
object UDFTest {

  def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("UDFTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        //Register UDF
        sqlContext.udf.register("UPPERCASE", convertToUpperCase(_: String))

        var studentDF =
sqlContext.read.json("file:///home/debdutta/HadoopEnviornment/data/spark/input/student.json")
        studentDF.registerTempTable("STUDENT")

        var result = sqlContext.sql("SELECT ID,UPPERCASE(NAME) AS NAME FROM
STUDENT")
        result.show
  }

  def convertToUpperCase(value: String) = {
        value.toUpperCase()
  }
}
```

Example 7: Data Set Multiple Example

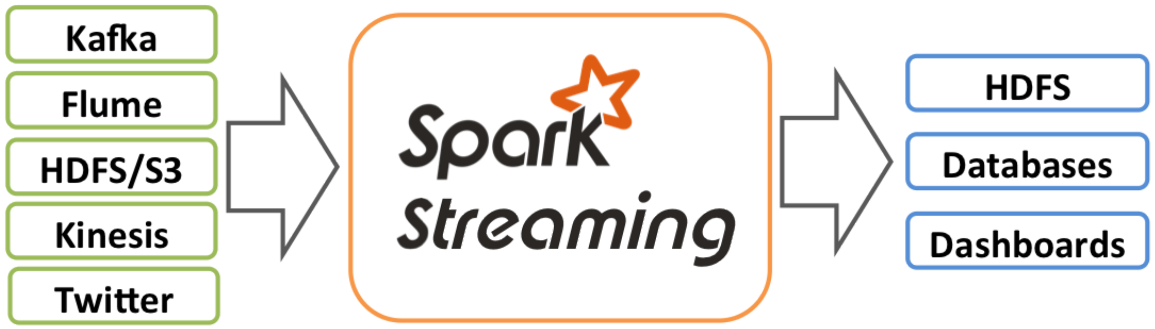
```
object DatasetExampleTest {

  def main(args: Array[String]): Unit =
        {
        val sparkConfig = new
SparkConf().setMaster("local").setAppName("DatasetExampleTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        val url = "jdbc:mysql://localhost:3306/play"
        val prop = new java.util.Properties
        prop.setProperty("user", "root")
        prop.setProperty("password", "Deb123")
        var studentDF = sqlContext.read.jdbc(url, "STUDENT", prop)
        println("Show AllRecords:")
        studentDF.show()

        println("Select Query Using Equal To")
        studentDF.filter(studentDF("name").equalTo("Rahul")).show()

        println("Select Query Using ==")
        studentDF.filter(studentDF("name") === "Rahul").show()

        println("Select Query Using >=")
        studentDF.filter(studentDF("id") > 9).show()

        println("Select Query Using gt=")
        studentDF.filter(studentDF("id").gt(9)).show()

        println("Select Query Using geq (Greater or Equal)=")
        studentDF.filter(studentDF("id").geq(9)).show()

        println("Select Query Using leq (Less or Equal)=")
        studentDF.filter(studentDF("id").leq(2)).show()

        println("Select Using And")

studentDF.filter(studentDF("name").equalTo("Priya").and(studentDF("city").equalTo("Bangalore")
)).show()

        println("Select Using Or")

studentDF.filter(studentDF("name").equalTo("Priya").or(studentDF("mobile").equalTo("21474836
47"))).show()

        println("Select Using Between")
        studentDF.filter(studentDF("id").between(8, 10)).show()

        println("Select Using In")
        studentDF.filter(studentDF("id").isin(8, 10)).show()

        println("Select Using Like Operator")
        studentDF.filter(studentDF("name") like "Pri%").show()

        println("Select Using Sorting")
        studentDF.sort(studentDF("name").asc).show()

        println("Select Using Order By")
        studentDF.orderBy(studentDF("name").asc).show()

        println("Select Using Group By- Count")
        studentDF.groupBy(studentDF("name")).count.show()

        }
}
```

Spark Hive Integration:

Copy paste hive-site.xml to spark config folder. Otherwise it will create its own metastore and version will mismatch.

```
def main(args: Array[String]): Unit = {

        val sparkConfig = new SparkConf().setMaster("local").setAppName("DBTest")
        val sparkContext = new SparkContext(sparkConfig)
        val sqlContext = new SQLContext(sparkContext)
        sparkContext.setLogLevel("info")

        sqlContext.sql("create table if not exists student (id int, name string, city string,state
string) " +
        "row format delimited fields terminated by '\t' lines terminated by '\n' stored as textfile")
        sqlContext.sql("load data local inpath
'/home/debdutta/HadoopEnviornment/data/hive/student.txt' into table student")
        var result = sqlContext.sql("from student select id,name,state")
        result.show
  }
```

Spark Streaming



Streaming Context:

- Responsible for consuming data from various source.
- Register an Input DStream to produce Receiver Object

D-Stream

Spark DStream (Discretized Stream) is the basic abstraction of Spark Streaming. DStream is a continuous stream of data. It receives input from various sources like Kafka, Flume, Kinesis, or TCP sockets. It can also be a data stream generated by transforming the input stream.
Spark DStream also support two types of Operations:

- **Transformation**: There are two types of transformation in DStream:
    - ❖ **Stateless Transformations-** The processing of each batch has no dependency on the data of previous batches. Stateless transformations are simple RDD transformations.

    - ❖ **Stateful Transformations-**It uses data or intermediate results from previous batches and computes the result of the current batch. Stateful transformations are operations on DStreams that track data across time. Thus it makes use of some data from previous batches to generate the results for a new batch.

- **Operation-** Once we get the data after transformation, on that data output operation are performed in Spark Streaming. After the debugging of our program, using output operation we can only save our output. Some of the output operations are print(), save() etc..

Example 1: Stream from File

object FileStreamTest {

```
  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("FileStreamTest")
   // var sparkContext = new SparkContext(sparkConfig)
          //sparkContext.setLogLevel("info")

          var streamingContext = new StreamingContext(sparkConfig, Seconds(20))
          //Hdfs Location "hdfs://localhost:9000/user/root/input/b2c.txt"
          var result =
streamingContext.textFileStream("hdfs://localhost:9000/user/root/spark/input/")
          var words = result.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_ + _)

          words.print()
          //
result.saveAsTextFiles("file:///home/debdutta/HadoopEnviornment/data/spark/output/result.txt")
          streamingContext.start
          streamingContext.awaitTermination
  }
}
```

Example 2: Stream from Netcat

```
object NetcatTest {

  //To Run Netcat - nc -lk 9998

  def setupLogging() = {

          import org.apache.log4j.{ Level, Logger }
          val rootLogger = Logger.getRootLogger()
          rootLogger.setLevel(Level.ERROR)
  }

  def main(args: Array[String]): Unit = {

          var sparkConfig = new SparkConf().setMaster("local").setAppName("FileStreamTest")
   // var sparkContext = new SparkContext(sparkConfig)
          //sparkContext.setLogLevel("info")
          //setupLogging()

          var streamingContext = new StreamingContext(sparkConfig, Seconds(5))
          var result = streamingContext.socketTextStream("localhost", 9996).flatMap(_.split("
")).map((_, 1)).reduceByKey(_ + _)
```

```
        result.print
        streamingContext.start
        streamingContext.awaitTermination
  }
}
```

Example 3: Stream from Twitter

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds
import twitter4j.conf.ConfigurationBuilder
import org.apache.spark.streaming.twitter.TwitterUtils
import twitter4j.auth.OAuthAuthorization

object TwitterTest {

  def main(args: Array[String]): Unit = {

        var sparkConfig = new SparkConf().setMaster("local").setAppName("TwitterStreamTest")
        var streamingContext = new StreamingContext(sparkConfig, Seconds(20))
        var configurationBuilder = new ConfigurationBuilder


configurationBuilder.setDebugEnabled(true).setOAuthConsumerKey("q2zKAwuKcmuLJsF551fK
NI6vw")

.setOAuthConsumerSecret("vKoe2qed7avMQkWggvikOOQetbh8hCvcCjLbwuYXeboR0pr826")

.setOAuthAccessToken("162516802-kooFfVynRIXc3GZF7JD8TX35r6sixNgXW4G5g6aX")

.setOAuthAccessTokenSecret("Ao3rawgTn6gSqjdTO8wWXsDedeSDT9WsELvDeKcDtrJeR")
        val auth = new OAuthAuthorization(configurationBuilder.build)

        val tweets = TwitterUtils.createStream(streamingContext, Some(auth))

tweets.saveAsTextFiles("file:///home/debdutta/HadoopEnviornment/data/spark/output/twitter/res
ult")

        //Filter Result
        var result = tweets.filter { t =>
        val tags = t.getText.split("").filter(_.startsWith("#")).map(_.toLowerCase())
```

```
          tags.contains("#India")
          }
          streamingContext.start
          streamingContext.awaitTermination
  }
}
```