

PRACTICAL NO :06

SPARK

Unit Structure :

- 6.0 Introduction
- 6.1 Spark
- 6.2 Spark RDD – Operations
- 6.3 Create an RDD in Apache Spark
- 6.4 SPARK IN-Memory Computing
- 6.5 Lazy Evaluation In Apache Spark
- 6.6 RDD Persistence and Caching in Spark
- 6.7 Lab Sessions
- 6.8 Exercises
- 6.9 Questions
- 6.10 Quiz
- 6.11 Video Lectures
- 6.12 Moocs
- 6.13 References

6.0 INTRODUCTION

Spark is a unified analytics engine for large-scale data processing including built-in modules for SQL, streaming, machine learning and graph processing.

6.1 SPARK

Apache Spark is an open-source cluster computing framework. Its primary purpose is to handle the real-time generated data. Spark was built on the top of the Hadoop MapReduce. It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives. Apache Spark RDD makes developer's work more efficient, as it divides cluster into nodes to compute parallel operations on each node. Before anything else, we will go through the brief introduction of Spark RDD.

Introduction to Apache Spark RDD

Apache Spark RDDs (Resilient Distributed Datasets) are a basic abstraction of spark which is immutable. These are logically partitioned so that we can also apply parallel operations on them. We can create RDD in three ways:

1. Parallelizing an already existing collection in the driver program.
2. Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
3. Creating RDD from already existing RDDs.

Prominent Features:

There are following traits of Resilient distributed datasets. Those are list-up below:



Fig 1. Spark Prominent Features

i. In-memory computation The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes.

ii. Lazy Evaluation The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do.

iii. Fault Tolerance Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data.

iv. Immutability RDDs are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

v. Persistence We can store the frequently used RDD in in-memory and we can also SPARK retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function. Follow this guide for the detailed study of RDD persistence in Spark.

vi. Partitioning RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

vii. Parallel Rdd, process the data parallelly over the cluster.

viii. Location-Stickiness RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus speed up computation.

ix. Coarse-grained Operation We apply coarse-grained transformations to RDD. Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the data set of RDD.

x. Typed We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

xi. No limitation We can have any number of RDD. There is no limit to its number. The limit depends on the size of disk and memory.

Limitations of Apache Spark – Ways to Overcome Spark Drawbacks



Fig 2. Limitations of Apache Spark

1. Objective

Some of the drawbacks of Apache Spark are there is no support for real-time processing, Problem with small file, no dedicated File management system, Expensive and much more due to these limitations of Apache Spark, industries have started shifting to Apache Flink– 4G of Big Data.

2. Limitations of Apache Spark

As we know Apache Spark is the next Gen Big data tool that is being widely used by industries but there are certain limitations of Apache Spark due to which industries have started shifting to Apache Flink– 4G of Big Data.

a. No Support for Real-time Processing In Spark Streaming, the arriving live stream of data is divided into batches of the pre-defined interval, and each batch of data is treated like Spark Resilient Distributed Database (RDDs). Then these RDDs are processed using the operations like map, reduce, join etc. The result of these operations is returned in batches. Thus, it is not real time processing but Spark is near real-time processing of live data. Micro-batch processing takes place in Spark Streaming.

b. Problem with Small File If we use Spark with Hadoop, we come across a problem of small file. HDFS provides a limited number of large files rather than a large number of small files. Another place where Spark legs behind is we store the data gzipped in S3. This pattern is very nice except when there are lots of small gzipped files. Now the work of the Spark is to keep those files on network and uncompress them. The gzipped files can be uncompressed only if the entire file is on one core. In the resulting RDD, each file will become a partition; hence there will be a large amount of tiny partition within an RDD. Now if we want efficiency in our processing, the RDDs should be repartitioned into some manageable format. This requires extensive shuffling over the network.

c. No File Management System Apache Spark does not have its own file management system, thus it relies on some other platform like Hadoop or another cloud-based platform which is one of the Spark known issues. **d. Expensive** In-memory capability can become a bottleneck when we want costefficient processing of big data as keeping data

in memory is quite expensive, the memory consumption is very high, and it is not Handled

d. Expensive In-memory capability can become a bottleneck when we want costefficient processing of big data as keeping data in memory is quite expensive, the memory consumption is very high, and it is not handled in a user-friendly manner. Apache Spark requires lots of RAM to run in-memory, thus the cost of Spark is quite high.

e. Less number of Algorithms Spark MLlib lags behind in terms of a number of available algorithms like Tanimoto distance.

f. Manual Optimization The Spark job requires to be manually optimized and is adequate to specific datasets. If we want to partition and cache in Spark to be correct, it should be controlled manually.

g. Iterative Processing In Spark, the data iterates in batches and each iteration is scheduled and executed separately.

h. Latency Apache Spark has higher latency as compared to Apache Flink.

i. Window Criteria Spark does not support record based window criteria. It only has timebased window criteria.

j. Back Pressure Handling

Back pressure is build up of data at an input-output when the buffer is full and not able to receive the additional incoming data. No data is transferred until the buffer is empty.

6.2 SPARK RDD – OPERATIONS

We can perform different operations on RDD as well as on data storage to form another RDDs from it. There are two different operations:

- ❖ Spark RDD - Spark RDD operation
- ❖ Spark RDD – Transformation and Action
- ❖ TRANSFORMATION

❖ **ACTION** To modify the available datasets, we need to provide step by step instructions to the spark. Those steps must clearly explain what changes we want. That set of instructions is generally known as Transformations”. Transformations are operations on RDDs which results in new RDD such as Map, Filter.

Actions are operations, triggers the process by returning the result back to program. Transformation and actions work differently.

1. Transformation

Transformation is a process of forming new RDDs from the existing ones. Transformation is a user specific function. It is a process of changing the current dataset in the dataset we want to have. Some common transformations supported by Spark are: For example, Map(func), Filter(func), Mappartitions (func), Flatmap (func) etc. All transformed RDDs are lazy in nature. As we are already familiar with the term “Lazy Evaluations”. That means it does not produce their results instantly. However, we always require an action to complete the computation. To trigger the execution, an action is a must. Up to that action data inside RDD is not transformed or available. After transformation, you incrementally build the lineage. That lineage is which formed by the entire parent RDDs of final RDDs. As soon as the execution process ends, resultant RDDs will be completely different from their parent RDDs. They can be smaller (e.g.

filter, count, distinct, sample), bigger (e.g. flatMap, union, Cartesian) or the same size (e.g. map). Transformation can categorize further as: Narrow Transformations, Wide Transformations.

a. Narrow Transformations

Narrow transformations are the result of map() and filter() functions and these compute data that live on a single partition meaning there will not be any data movement between partitions to execute narrow transformations. An output RDD also has partitions with records. In the parent RDD, that output originates from a single partition. Additionally, to calculate the result Only a limited subset of partitions is used. In Apache spark narrow transformations groups as a stage. That process is mainly known as pipelining. Pipelining is an implementation mechanism. Functions such as map(), mapPartition(), flatMap(), filter(), union() are some examples of narrow transformation.

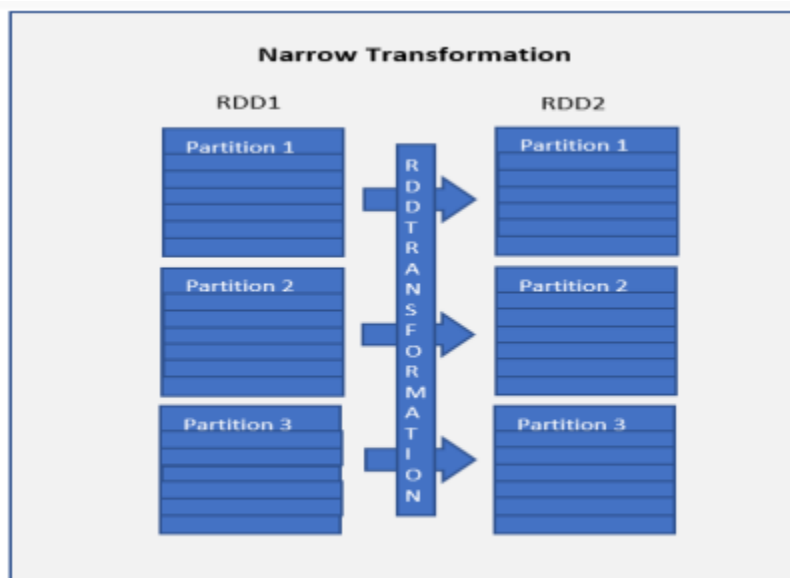


Fig 3. Narrow Transformation

b. Wide Transformations

Wide transformations are the result of groupByKey (func) and reduceByKey (func). As data may reside in many partitions of the parent RDD. These are used to compute the records by data in the single partition. Wide transformations may also know as shuffle transformations. Functions such as groupByKey(), aggregateByKey(), aggregate(), join(), repartition() are some examples of a wider transformations. Note: When compared to Narrow transformations, wider transformations are expensive operations due to shuffling.

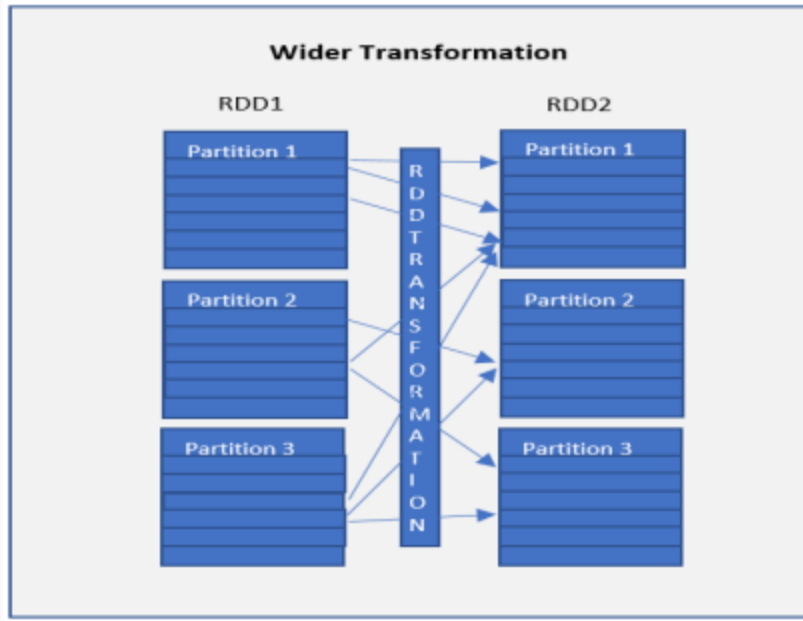


Fig 4. Wider Transformation

TRANSFORMATION METHODS	METHOD USAGE AND DESCRIPTION
<code>cache()</code>	Caches the RDD
<code>filter()</code>	Returns a new RDD after applying filter function on source dataset.
<code>flatMap()</code>	Returns flatmap meaning if you have a dataset with array, it converts each elements in a array as a row. In other words it return 0 or more items in output for each element in dataset.
<code>map()</code>	Applies transformation function on dataset and returns same number of elements in distributed dataset.
<code>mapPartitions()</code>	Similar to map, but executes transformation function on each partition, This gives better performance than map function
<code>mapPartitionsWithIndex()</code>	Similar to map Partitions, but also provides func with an integer value representing the index of the partition.
<code>randomSplit()</code>	Splits the RDD by the weights specified in the argument. For example <code>rdd.randomSplit(0.7,0.3)</code>
<code>union()</code>	Comines elements from source dataset and the argument and returns combined dataset. This is similar to union function in Math set operations.
<code>sample()</code>	Returns the sample dataset.
<code>intersection()</code>	Returns the dataset which contains elements in both source dataset and an argument
<code>distinct()</code>	Returns the dataset by eliminating all duplicated elements.
<code>repartition()</code>	Return a dataset with number of partition specified in the argument. This operation reshuffles the RDD randomly. It could either return lesser or more partitioned RDD based on the input supplied.
<code>coalesce()</code>	Similar to repartition by operates better when we want to decrease the partitions. Betterment achieves by reshuffling the data from fewer nodes compared with all nodes by repartition.

2. Actions

An action is an operation, triggers execution of computations and RDD transformations. Also, returns the result back to the storage or its program. Transformation returns new RDDs and actions returns some other data types. Actions give non-RDD values to the RDD operations. It forces the evaluation of the transformation process need for the RDD they may call on. Since they actually need to produce output. An action instructs Spark to compute a result from a series of transformations.

Actions are one of two ways to send data from executors to the driver. Executors are agents that are responsible for executing different tasks. While a driver coordinates execution of tasks.

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, the new RDD is not formed like a transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the SPARK external storage system. It brings laziness of RDD into motion. An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

`count()`

Action count() returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

Count()

```
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())[/php]
```

Note – In above code flatMap() function maps line into words and count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

collect()

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

Collect() example:

```
[php]val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8) ))
val result = data.join(data2)
println(result.collect().mkString(","))[/php]
```

take(n)

The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements. For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

Take()

example:

```
[php]val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
val twoRec = result.take(2)
twoRec.foreach(println)[/php]
```

Note – The take(2) Action will return an array with the first n elements of the data set defined in the taking argument.

top()

If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.

Top() example:

```
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
```



```
res.foreach(println)[/php]
```

Note – map() operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

```
countByValue()
```

The countByValue() returns, many times each element occur in RDD. For example, RDD has values { 1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.countByValue()” will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

countByValue() example:

```
[php]val data = spark.read.textFile(“spark_test.txt”).rdd
val result= data.map(line => (line,line.length)).countByValue()
result.foreach(println)[/php]
```

```
reduce()
```

The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

```
Reduce() example: [php]val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_ )
println(sum)[/php]
```

Note – The reduce() action in above code will add the elements of the source RDD.

fold() The signature of the fold() is like reduce(). Besides, it takes “zero value” as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation. The key difference between fold() and reduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.

For example, zero is an identity for addition; one is identity element for multiplication. The return type of fold() is same as that of the element of RDD we are operating on.

For example,

```
rdd.fold(0)((x, y) => x + y).
```

Fold() example:

```
[php]val rdd1 = spark.sparkContext.parallelize(List((“maths”, 80),(“science”, 90)))
val additionalMarks = (“extra”, 4)
val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2 (“total”,
add) }
println(sum)[/php]
```

```
aggregate()
```

It gives us the flexibility to get data type different from the input type. The aggregate() takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the driver. In this case, foreach() function is useful. For example, inserting a record into the database.

Foreach() example:

```
[php]val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
```

```
val group = data.groupByKey().collect()
```

```
group.foreach(println)[/php]
```

Note – The foreach() action run a function (println) on each element of the dataset group

6.3 CREATE AN RDD IN APACHE SPARK

1. Spark create RDD from Seq or List (using Parallelize)
2. Creating an RDD from a text file
3. Creating from another RDD
4. Creating from existing DataFrames and DataSet

Spark Create RDD from Seq or List (using Parallelize)

RDD's are generally created by parallelized collection i.e. by taking an existing collection from driver program (scala, python e.t.c) and passing it to SparkContext's parallelize() method. This method is used only for testing but not in realtime as the entire data will reside on one node which is not ideal for production.

```
val rdd=spark.sparkContext.parallelize(Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000)))
rdd.foreach(println)
```

Outputs: (Python,100000) (Scala,3000) (Java,20000)

Create an RDD from a text file

SPARK Mostly for production systems, we create RDD's from files. here will see how to create an RDD by reading data from a file.

```
val rdd = spark.sparkContext.textFile("/path/textFile.txt")
```

This creates an RDD for which each record represents a line in a file. If you want to read the entire content of a file as a single record use wholeTextFiles() method on sparkContext.

```
val rdd2 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
rdd2.foreach(record=>println("FileName : "+record._1+" , FileContents :"+record._2))
```

In this case, each text file is a single record. In this, the name of the file is the first column and the value of the text file is the second column

Creating from another RDD

You can use transformations like map, flatmap, filter to create a new RDD from an existing one.

```
val rdd3 = rdd.map(row=>{(row._1,row._2+100)})
```

Above, creates a new RDD "rdd3" by adding 100 to each record on RDD. this example outputs below.

(Python,100100)

(Scala,3100)
(Java,20100)

From existing DataFrames and DataSet

To convert DataSet or DataFrame to RDD just use rdd() method on any of these data types. `val myRdd2 = spark.range(20).toDF().rdd`
toDF() creates a DataFrame and by calling rdd on DataFrame returns back RDD.

6.4 SPARK IN-MEMORY COMPUTING

The data is kept in random access memory(RAM) instead of some slow disk drives and is processed in parallel. Using this we can detect a pattern, analyze large data. This has become popular because it reduces the cost of memory. So, in-memory processing is economic for applications. The two main columns of in-memory computation are-

- RAM storage
- Parallel distributed processing.

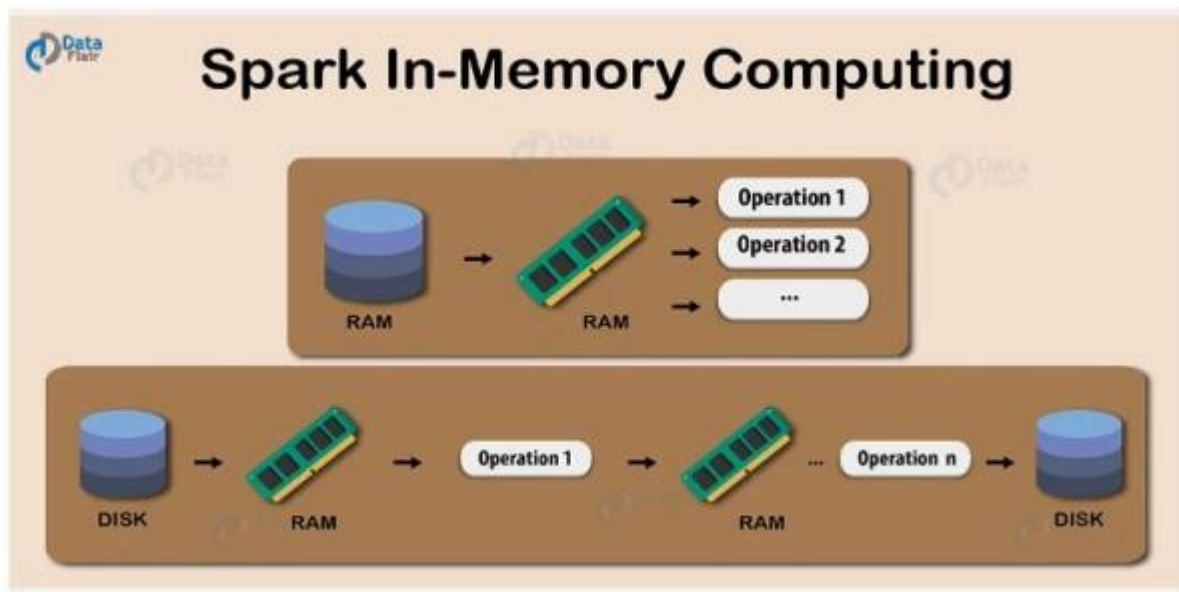


Fig 5. Spark In-Memory Computing

Keeping the data in-memory improves the performance by an order of magnitudes. The main abstraction of Spark is its RDDs. And the RDDs are cached using the cache() or persist() method.

When we use cache() method, all the RDD stores in-memory. When RDD stores the value in memory, the data that does not fit in memory is either recalculated or the excess data is sent to disk. Whenever we want RDD, it can be extracted without going to disk. This reduces the space-time complexity and overhead of disk storage.

The in-memory capability of Spark is good for machine learning and microbatch processing. It provides faster execution for iterative jobs.

When we use `persist()` method the RDDs can also be stored in-memory, we can use it 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.

Storage levels of RDD `Persist()` in Spark The various storage level of `persist()` method in Apache Spark RDD are:

- ❖ `MEMORY_ONLY`
- ❖ `MEMORY_AND_DISK`
- ❖ `MEMORY_ONLY_SER`
- ❖ `MEMORY_AND_DISK_SER`
- ❖ `DISK_ONLY`
- ❖ `MEMORY_ONLY_2` and `MEMORY_AND_DISK_2`

6.5 LAZY EVALUATION IN APACHE SPARK

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.

Transformations are lazy in nature meaning when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called (Through DAG).

Apache Spark Lazy Evaluation

Feature. Apache Spark Lazy Evaluation Explanation.

In MapReduce, much time of developer wastes in minimizing the number of MapReduce passes. It happens by clubbing the operations together. While in Spark we do not create the single execution graph, rather we club many simple operations. Thus it creates the difference between Hadoop MapReduce vs Apache Spark.

In Spark, the driver program loads the code to the cluster. When the code executes after every operation, the task will be time and memory consuming. Each time data goes to the cluster for evaluation. **Advantages of Lazy Evaluation in Spark Transformation**

There are some benefits of Lazy evaluation in Apache Spark.

- a. **Increases Manageability** By lazy evaluation, users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.
- b. **Saves Computation and increases Speed** Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get computed. It saves the trip between driver and cluster, thus speeds up the process.
- c. **Reduces Complexities** The two main complexities of any operation are time and space complexity. Using Apache Spark lazy evaluation we can overcome both. Since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.
- d. **Optimization** It provides optimization by reducing the number of queries. Learn more about Apache Spark Optimization.

Example 1

In the first step, we created a list of 10 million numbers and made an RDD with four partitions below. And we can see the result in the below output image.

```
val data = (1 to 1000000).toList
```

```
val rdd = sc.parallelize(data,4) println("Number of partitions is "+rdd.getNumPartitions)
```

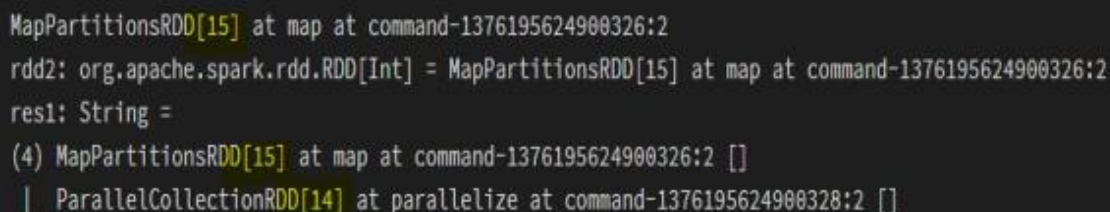


Next, we will perform a fundamental transformation, like adding 4 to each number. Note that Spark, at this point, has not started any transformation. It only records a series of transformations in the form of RDD Lineage. You can see that RDD lineage using the function toDebugString

```
//Adding 5 to each value in rdd val rdd2 = rdd.map(x => x+5)
```

```
//rdd2 objectc println(rdd2)
```

```
//getting rdd lineage rdd2.toDebugString
```



```
MapPartitionsRDD[15] at map at command-1376195624900326:2
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[15] at map at command-1376195624900326:2
res1: String =
(4) MapPartitionsRDD[15] at map at command-1376195624900326:2 []
| ParallelCollectionRDD[14] at parallelize at command-1376195624900328:2 []
```

Now if you observe MapPartitionsRDD[15] at map is dependent on SPARK ParallelCollectionRDD[14]. Now, let's go ahead and add one more transformation to add 20 to all the elements of the list.

```
//Adding 5 to each value in rdd val rdd3 = rdd2.map(x => x+20)
```

```
//rdd2 objectc println(rdd3)
```

```
//getting rdd lineage rdd3.toDebugString rdd3.collect
```

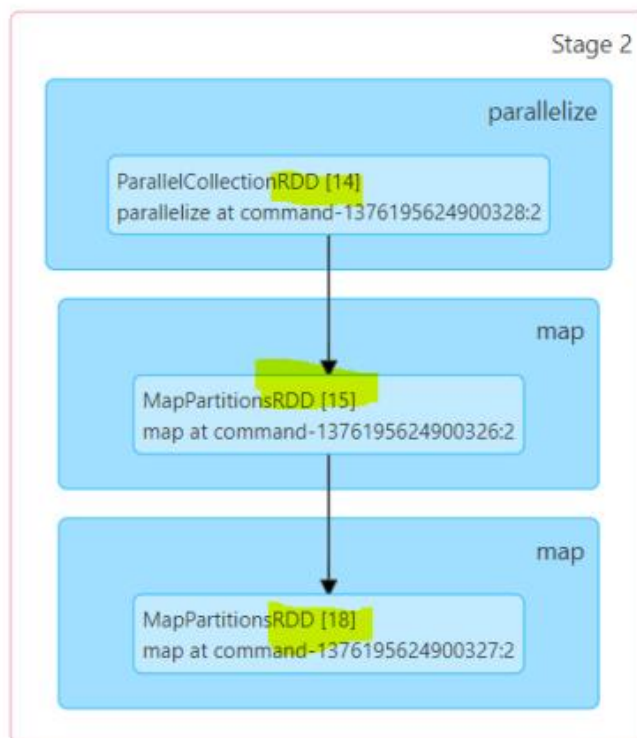
```

MapPartitionsRDD[18] at map at command-1376195624900327:2
rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[18] at map at command-1376195624900327:2
res4: String =
(4) MapPartitionsRDD[18] at map at command-1376195624900327:2 []
| MapPartitionsRDD[15] at map at command-1376195624900326:2 []
| ParallelCollectionRDD[14] at parallelize at command-1376195624900328:2 []

```

Now if you observe MapPartitionsRDD[18] at map is dependent on MapPartitionsRDD[15] and ParallelCollectionRDD[14]. Now, let's go ahead and add one more transformation to add 20 to all the elements of the list. After calling an action using collect we see that three stages of DAG lineage at ParallelCollectionRDD[14], MapPartitionsRDD[15] and MapPartitionsRDD[18].

▼ DAG Visualization



From the above examples, we can able to understand that spark lineage is maintained using DAG.

Example 2

Here we see how unnecessary steps or transformations are skipped due to spark Lazy evaluation during the execution process.

```

import org.apache.spark.sql.functions._
val df1 = (1 to 100000).toList.toDF("col1")
//scenario 1
println("scenario 1")

```

```
df1.withColumn("col2",lit(2)).explain(true);
//scenario 2
println("scenario 2")
df1.withColumn("col2",lit(2)).drop("col2").explain(true);
```

In this, we created a dataframe with column "col1" at the very first step. If you observe Scenario-1, I have created a column "col2" using withColumn() function and after that applied explain() function to analyze the physical execution plan. The below image contains a logical plan, analyzed logical plan, optimized logical plan, and physical plan. In the analyzed logical plan, if you observe there is only one projection stage, Projection main indicates the columns moving forward for further execution.

```
scenario 1
== Parsed Logical Plan ==
Project [col1#236, 2 AS col2#238]
+- Project [value#233 AS col1#236]
   +- LocalRelation [value#233]

== Analyzed Logical Plan ==
col1: int, col2: int
Project [col1#236, 2 AS col2#238]
+- Project [value#233 AS col1#236]
   +- LocalRelation [value#233]

== Optimized Logical Plan ==
LocalRelation [col1#236, col2#238]

== Physical Plan ==
LocalTableScan [col1#236, col2#238]
```

If you observe Scenario-1, I have created a column "col2" using the withColumn() function, and we are dropping that column and after that applied explain() function to analyze the physical execution plan. The below image contains a logical plan, analyzed logical plan, optimized logical plan, and physical plan. In the analyzed logical plan, if you observe there are only two projection stages, Projection main indicates the columns 127 moving forward for further execution. In finalized physical plan, there is no task of creation and of "col2"


```

scenario 2
== Parsed Logical Plan ==
Project [col1#236]
+- Project [col1#236, 2 AS col2#241]
   +- Project [value#233 AS col1#236]
      +- LocalRelation [value#233]

== Analyzed Logical Plan ==
col1: int
Project [col1#236]
+- Project [col1#236, 2 AS col2#241]
   +- Project [value#233 AS col1#236]
      +- LocalRelation [value#233]

== Optimized Logical Plan ==
LocalRelation [col1#236]

== Physical Plan ==
LocalTableScan [col1#236]

```

Advantages of Spark Lazy Evaluation

Users can divide the entire work into smaller operations for easy readability and management. But Spark internally groups the transformations reducing the number of passes on data. What this means is, if Spark could group two transformations into one, then it had to read the data only once to apply the transformations rather than reading twice. In one of the above examples[Scenario 2], we saw that only the necessary computation is done by Spark, which increases Speed.

As the action is triggered only when data is required, this reduces unnecessary overhead. Let's say a program does the following steps (i)Read a file, (ii)Does a function call unrelated to the file (iii)Lloads the file into a table. If Spark read the file as soon as it met the first transformation, it had to wait for the function call to finish before it loads the data, and all this while the data had to sit in memory.

6.6 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.

Persistence in Apache Spark

In Spark, we can use some RDD's multiple times. If honestly, we repeat the same process of RDD evaluation each time it required or brought into action. This task can be time and memory consuming, especially for iterative algorithms that look at data multiple times. To solve the problem of repeated computation the technique of persistence came into the picture.

Benefits of RDD Persistence in Spark

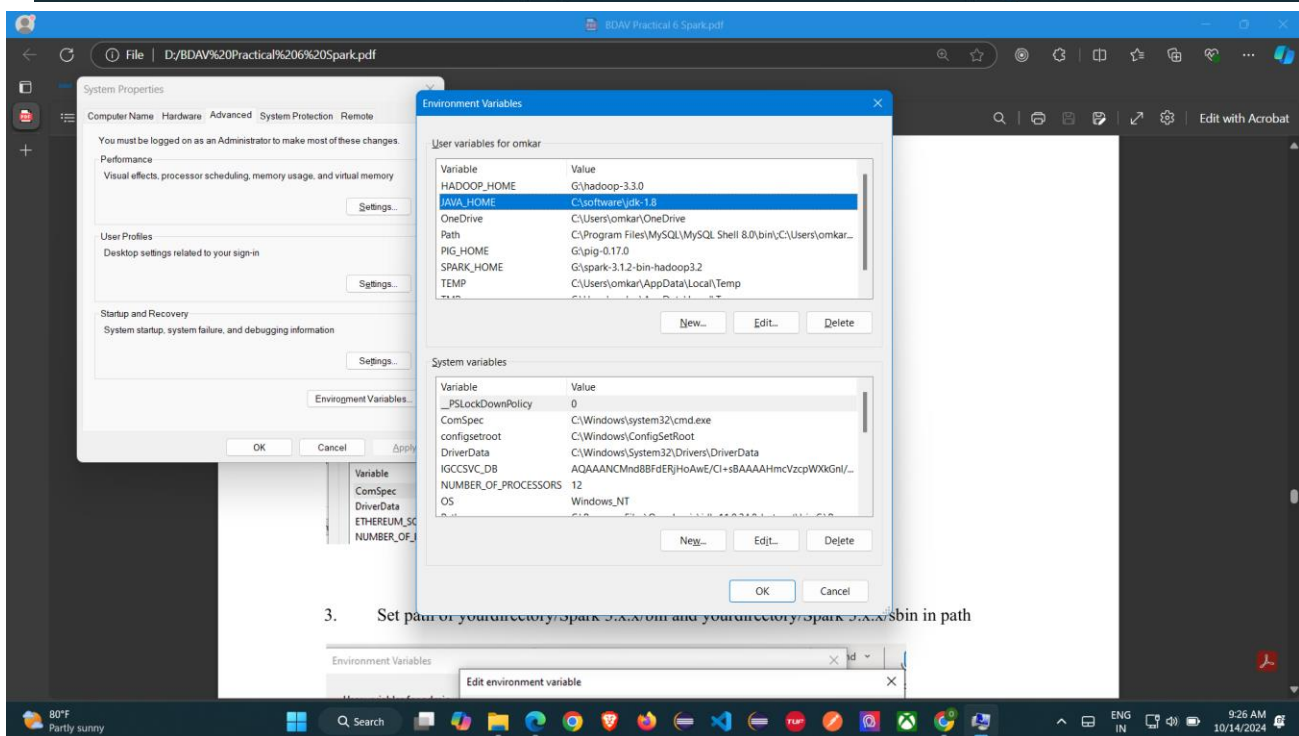
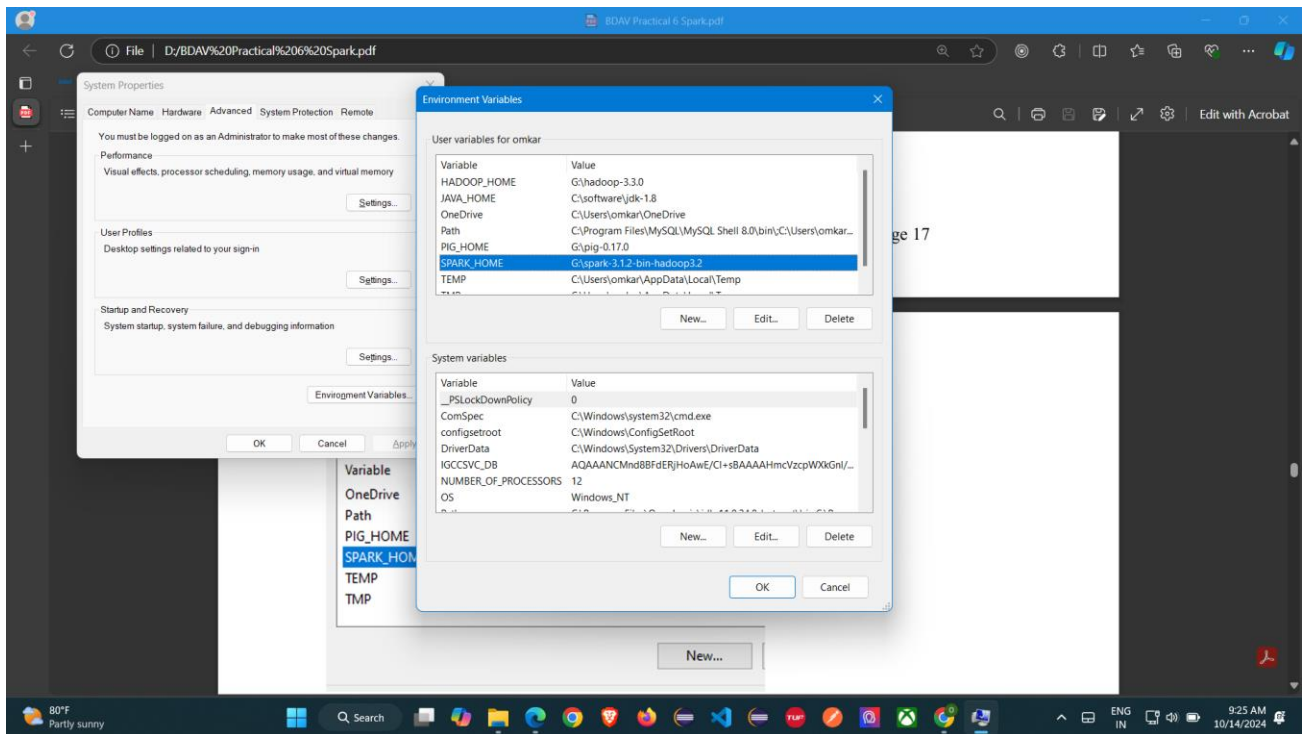
There are some advantages of RDD caching and persistence mechanism in spark. It makes the whole system

- ☐ Time efficient
 - ☐ Cost efficient
 - ☐ Lessen the execution time.
-

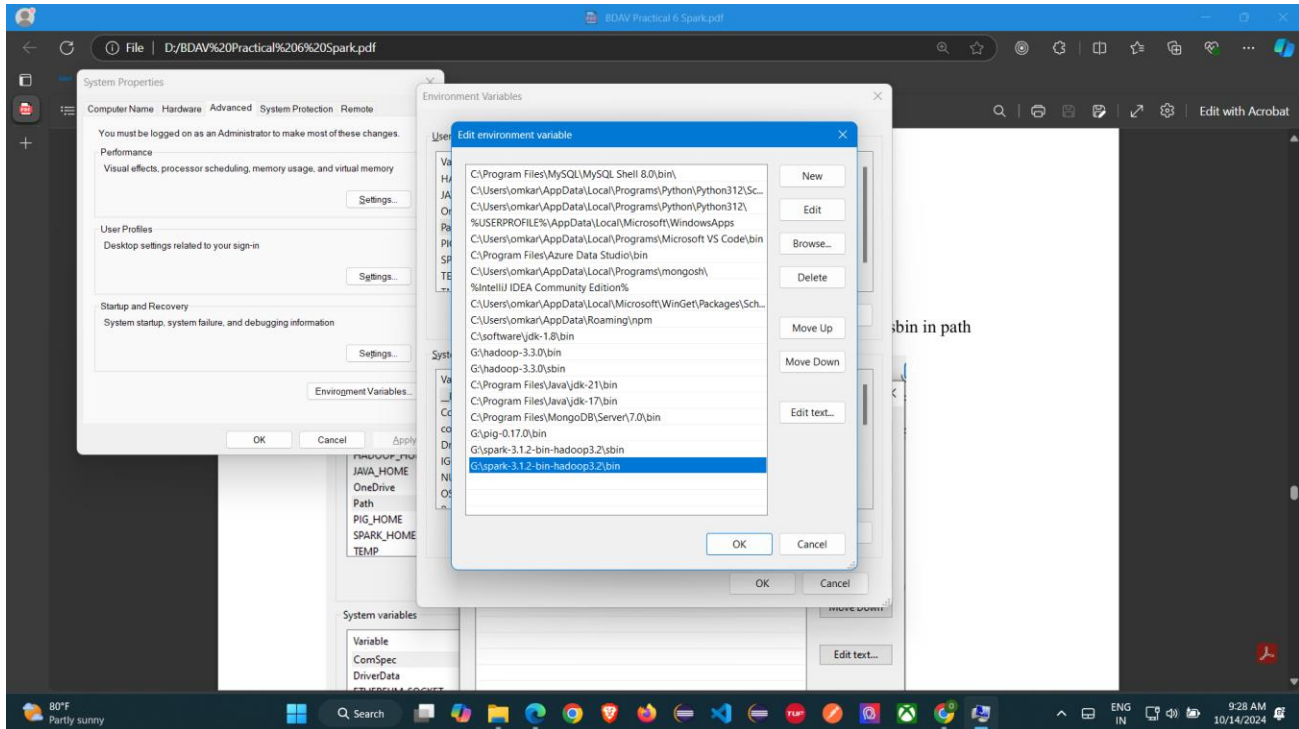
6.7 LAB SESSIONS

Install spark and extract it

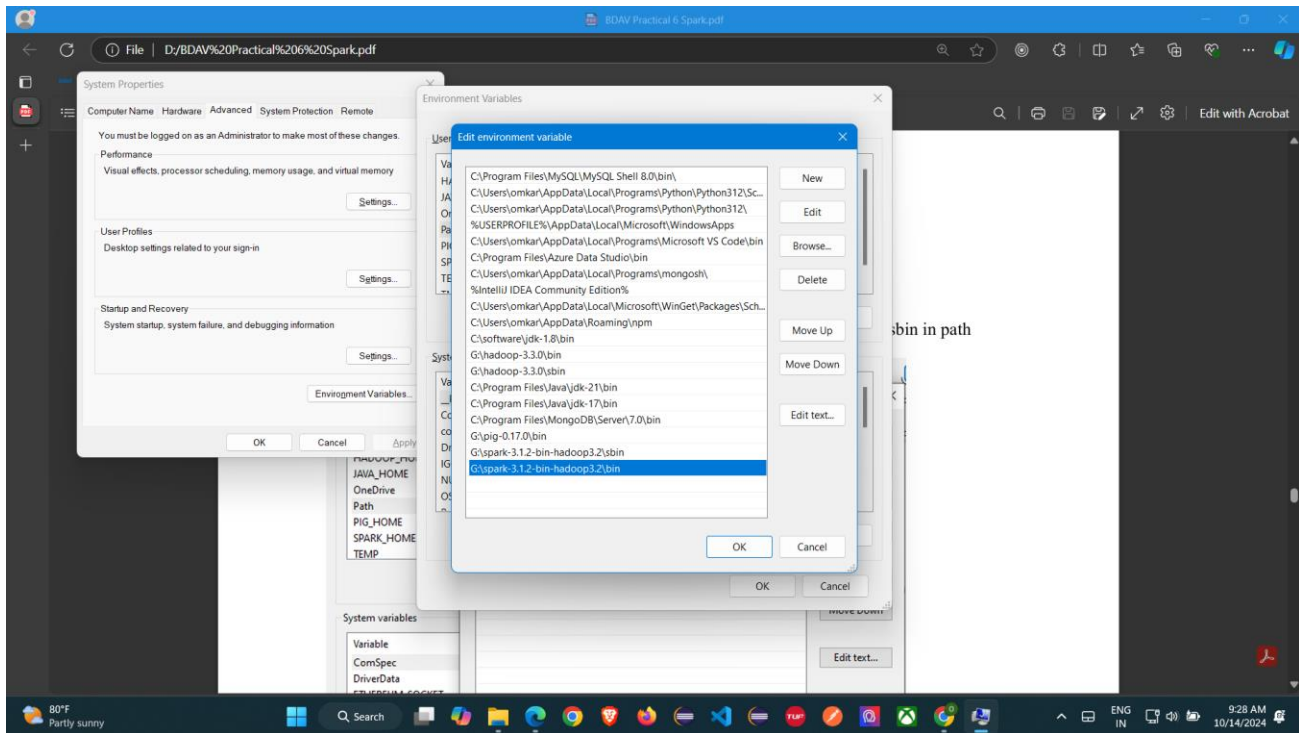
1. Set SPARK_HOME and JAVA_HOME in environment variable
 - a. [SPARK_HOME to /yourdirectory/Spark 3.x.x]
 - b. [JAVA_HOME to /java folder in Program files]



2. Set path of Java/bin in environment variable



3. Set path of yourdirectory/Spark 3.x.x/bin and yourdirectory/Spark 3.x.x/sbin in path



- ```
G:\>cd spark-3.1.2-bin-hadoop3.2\sbin
G:\spark-3.1.2-bin-hadoop3.2\sbin>
```

- ```
G:\spark-3.1.2-bin-hadoop3.2\sbin>spark-shell
```

```
G:\spark-3.1.2-bin-hadoop3.2\sbin>spark shell
'spark' is not recognized as an internal or external command,
operable program or batch file.
```



```
G:\spark-3.1.2-bin-hadoop3.2\sbin>spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://172.16.4.54:4040
Spark context available as 'sc' (master = local[*, app id = local-1728878497215).
Spark session available as 'spark'.
Welcome to
```



```
 _   _      _ 
| | | |    / \     version 3.1.2
| |_| |___/_\\_    _ \\_
|  __//__ \_\_\_\_/ \_\_\_\_
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```



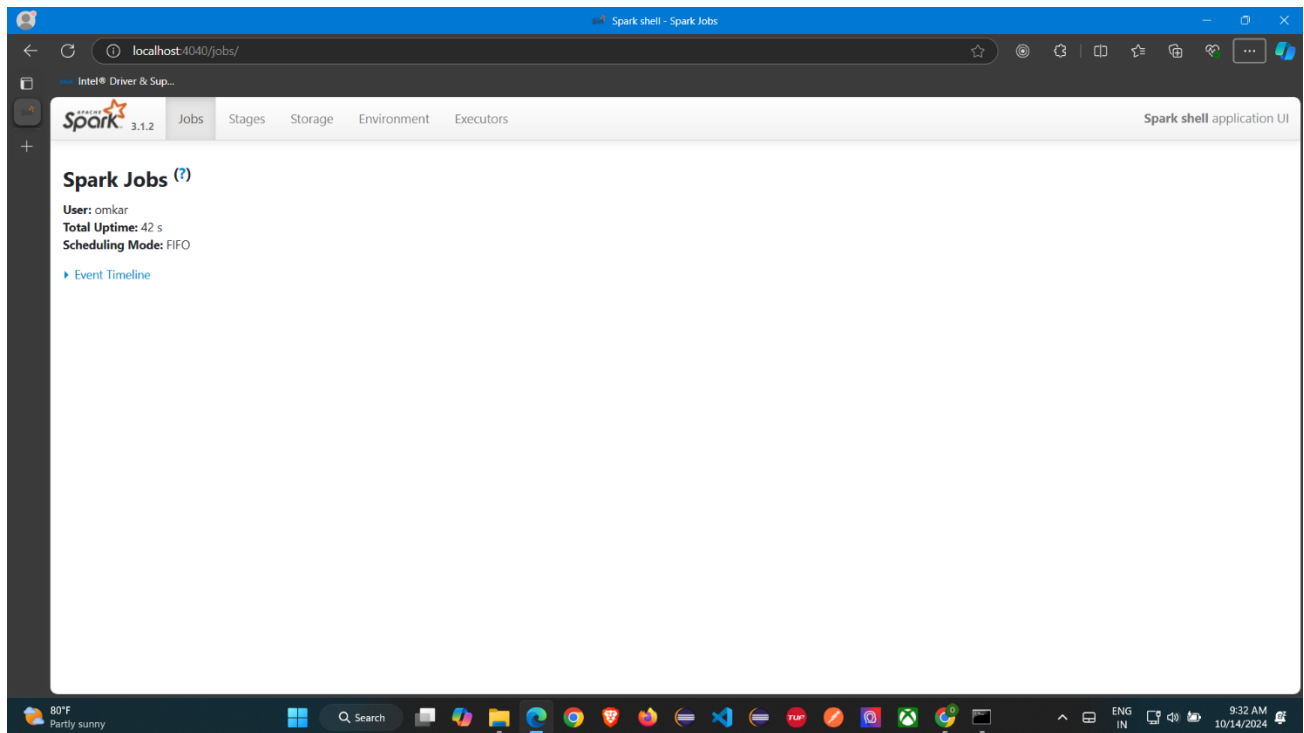
```
Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_411)
Type in expressions to have them evaluated.
Type :help for more information.
```



```
scala>
```

5. Check spark jobs using following url in browser

<http://localhost:4040/jobs/>



6. `scala> spark.version` run this command to check the version of spark RDD (Resilient Distributed Datasets (RDD))

```
scala> spark.version
res3: String = 3.1.2

scala>
```

7. `val info = Array(1, 2, 3, 4)`

```
scala> val info = Array(1, 2, 3, 4)
info: Array[Int] = Array(1, 2, 3, 4)

scala>
```

```
val distinfo = sc.parallelize(info)
```

```
scala> val distinfo = sc.parallelize(info)
distinfo: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26
scala> _
```

Create an RDD using a parallelized collection.

```
scala> val data = sc.parallelize(List(10,20,30))
```

```
data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24
```

```
scala> val data = sc.parallelize(List(10,20,30))
data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24
scala>
```

```
scala> data.collect
```

```
scala> data.collect
res4: Array[Int] = Array(10, 20, 30)
scala> _
```

Apply the map function and pass the expression required to perform.

```
scala> val mapfunc = data.map(x => x+10)
```

```
mapfunc: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at map at <console>:25
```

```
scala> val mapfunc = data.map(x => x+10)
mapfunc: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at map at <console>:25
scala>
```

Now, we can read the generated result by using the following command.

```
scala> mapfunc.collect
```

```
res2: Array[Int] = Array(20, 30, 40)
```

```
scala> mapfunc.collect
res5: Array[Int] = Array(20, 30, 40)
scala> _
```

Create an RDD using a text file.

Worcount program using map reduce in Spark

1. `hdfs dfs -mkdir /spark` #Create a directory in HDFS, where to kept text file.

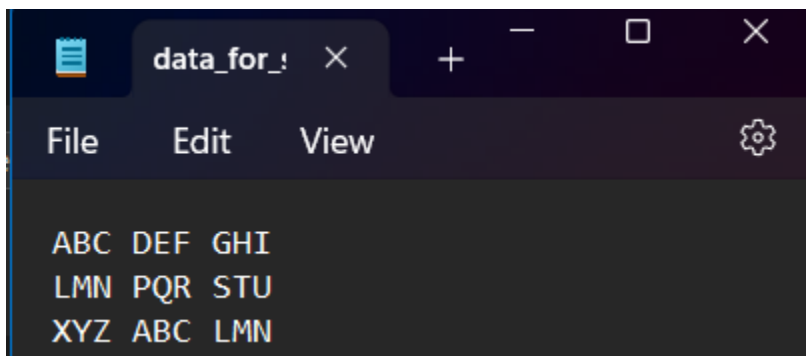
```
C:\Windows\System32>hdfs dfs -mkdir /spark
C:\Windows\System32>hdfs dfs -put G:\spark-3.1.2-bin-hadoop3.2\data_for_spark_practice.txt /spark
C:\Windows\System32>_
```

2. Upload the following data_for_spark_practical.txt file on HDFS in the specific directory. Check <http://localhost:9870/> and spark directory

ABC DEF GHI

LMN PQR STU

XYZ ABC LMN



```
hdfs dfs -put G:\spark-3.1.2-bin-hadoop3.2\data_for_spark_practice.txt /spark
```


Let's create an RDD by using the following command.

4. `val data=sc.textFile("data_for_spark_practice.txt")`

```
scala> val data=sc.textFile("data_for_spark_practical.txt")
data: org.apache.spark.rdd.RDD[String] = data_for_spark_practical.txt MapPartitionsRDD[4] at textFile at <console>:24
scala> _
```

Now, we can read the generated result by using the following command.

5. Splitting data:

`>val splitdata = data.flatMap(line => line.split(" "));`

`> splitdata.collect;`

```
scala> val splitdata = data.flatMap(line => line.split(" "));
splitdata: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at flatMap at <console>:25
scala>
scala> splitdata.collect;
res6: Array[String] = Array(ABC, DEF, GHI, LMN, PQR, STU, XYZ, ABC, LMN)
scala>
```

6. Now, we are going to map data or keys with value 1

`>val mapdata = splitdata.map(word => (word,1));`

`>mapdata.collect;`

```
scala> mapdata.collect;
res7: Array[(String, Int)] = Array((ABC,1), (DEF,1), (GHI,1), (LMN,1), (PQR,1), (STU,1), (XYZ,1), (ABC,1), (LMN,1))
scala> _
```

7. Now, we are going to reduce data.

```
>val reducedata = mapdata.reduceByKey(_+_);
```

```
>reducedata.collect;
```

```
scala> val reducedata = mapdata.reduceByKey(_+_);
reducedata: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[7] at reduceByKey at <console>:25

scala> reducedata.collect;
res8: Array[(String, Int)] = Array((ABC,2), (STU,1), (GHI,1), (DEF,1), (PQR,1), (XYZ,1), (LMN,2))

scala> _
```