Big Data
(Spark)

Instructor: Trong-Hop Do

November 24th 2020

S3Lab
Smart Software System Laboratory


“Big data is at the foundation of all the megatrends that are happening today, from social to mobile to cloud to gaming.”
– Chris Lynch, Vertica Systems

Spark shell

- Open Spark shell
- Command: **spark-shell**

```
[cloudera@quickstart ~]$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/parquet/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/avro/avro-tools-1.7.6-cdh5.13.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
20/11/28 00:15:29 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context available as sc (master = local[*], app id = local-1606551332403).
20/11/28 00:15:35 WARN shortcircuit.DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.
SQL context available as sqlContext.

scala>
```

What is SparkContext?

- Spark SparkContext is an entry point to Spark and defined in org.apache.spark package and used to programmatically create Spark RDD, accumulators, and broadcast variables on the cluster.
- Its object **sc** is default variable available in spark-shell and it can be programmatically created using SparkContext class.
- Most of the operations/methods or functions we use in Spark come from SparkContext for example accumulators, broadcast variables, parallelize, and more.

What is SparkContext?

- Test some methods with the default SparkContext object **sc** in Spark shell

```
scala> sc.applicationId
res9: String = local-1606562891233

scala> sc.appName
res10: String = Spark shell

scala> sc.applicationId
res11: String = local-1606562891233

scala> var map = sc.textFile("/user/cloudera/sample.txt")
map: org.apache.spark.rdd.RDD[String] = /user/cloudera/sample.txt MapPartitionsRDD[8] at textFile at <console>:27
```

- There are many more methods of SparkContext

Spark RDD

- Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark, It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
- Spark RDD can be created in several ways using Scala & Pyspark languages, for example, It can be created by using sparkContext.parallelize(), from text file, from another RDD, DataFrame, and Dataset.

Spark RDD

Create an RDD through Parallelized Collection

- Spark shell provides SparkContext variable “sc”, use **sc.parallelize()** to create an RDD[Int].

```
scala> val no = Array(1,2,3,4,5,6,7,8,9,10)
no: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val noData = sc.parallelize(no)
noData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at parallelize at <console>:29


scala> noData.foreach(println)
1
2
3
4
5
6
7
8
9
10
```

Spark RDD

Create an RDD through Parallelized Collection

- Spark shell provides SparkContext variable “sc”, use **sc.parallelize()** to create an RDD[Int].

```
scala> var pairData = sc.parallelize(Seq(("Java",20000),("Python",100000),("Scala",3000)))
pairData: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[30] at parallelize at <console>:27

scala> pairData.foreach(println)
(Java,20000)
(Python,100000)
(Scala,3000)
```

Spark RDD

Read single or multiple text files into single RDD?

- Create files and put them to HDFS

FILE NAME	FILE CONTENTS
text01.txt	One,1
text02.txt	Two,2
text03.txt	Three,3
text04.txt	Four,4
invalid.txt	Invalid,I

```
[cloudera@quickstart ~]$ cat > text01.txt
One,1
[cloudera@quickstart ~]$ cat > text02.txt
Two,2
[cloudera@quickstart ~]$ cat > text03.txt
Three,3
[cloudera@quickstart ~]$ cat > text04.txt
Four,4
[cloudera@quickstart ~]$ cat > invalid.txt
Invalid,I
```

```
[cloudera@quickstart ~]$ hdfs dfs -put text01.txt
[cloudera@quickstart ~]$ hdfs dfs -put text02.txt
[cloudera@quickstart ~]$ hdfs dfs -put text03.txt
[cloudera@quickstart ~]$ hdfs dfs -put text04.txt
[cloudera@quickstart ~]$ hdfs dfs -put invalid.txt
```

Spark RDD

Read single or multiple text files into single RDD?

- Read single file and create an RDD[String]

```
scala> var textData = sc.textFile("/user/cloudera/text01.txt")
textData: org.apache.spark.rdd.RDD[String] = /user/cloudera/text01.txt MapPartitionsRDD[21] at textFile at <console>:27

scala> textData.foreach(f => {println(f)})
One,1
```

Spark RDD

Read single or multiple text files into single RDD?

- Read multiple files and create an RDD[String]

```
scala> var textData = sc.textFile("/user/cloudera/text*")
textData: org.apache.spark.rdd.RDD[String] = /user/cloudera/text* MapPartitionsRDD[23] at textFile at <console>:27

scala> textData.foreach(f => {println(f)})
One,1
Two,2
Three,3
Four,4
```

Spark RDD

Read single or multiple text files into single RDD?

- Read multiple specific files and create an RDD[String]

```
scala> var textData = sc.textFile("/user/cloudera/text01.txt,/user/cloudera/text03.txt")
textData: org.apache.spark.rdd.RDD[String] = /user/cloudera/text01.txt,/user/cloudera/text03.txt MapPartitionsRDD[25] at
 textFile at <console>:27

scala> textData.foreach(f => {println(f)})
One,1
Three,3
```

Spark RDD

Load CSV File into RDD

- Create .csv files and put them to HDFS

FILE NAME	FILE CONTENTS
text01.csv	Col1,Col2 One,1 Eleven,11
text02.csv	Col1,Col2 Two,2 Twenty One,21

```
[cloudera@quickstart ~]$ cat > text01.csv
Col1,Col2
One,1
Eleven,11
[cloudera@quickstart ~]$ cat > text02.csv
Col1,Col2
Two,2
Twenty One,21
[cloudera@quickstart ~]$ hdfs dfs -put text01.csv
[cloudera@quickstart ~]$ hdfs dfs -put text02.csv
```

Spark RDD

Load CSV File into RDD

- textFile() method read an entire CSV record as a String and returns RDD[String]

```
scala> var textData = sc.textFile("/user/cloudera/text01.txt,/user/cloudera/text01.csv")
textData: org.apache.spark.rdd.RDD[String] = /user/cloudera/text01.txt,/user/cloudera/text01.csv MapPartitionsRDD[27] at
 textFile at <console>:27
```

Spark RDD

Load CSV File into RDD

- we would need every record in a CSV to split by comma delimiter and store it in RDD as multiple columns, In order to achieve this, we should use map() transformation on RDD where we will convert RDD[String] to RDD[Array[String] by splitting every record by comma delimiter. map() method returns a new RDD instead of updating existing.

```
scala> var csvData = textData.map(f=>{f.split(",")})
csvData: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[28] at map at <console>:29

scala> csvData.foreach(f=>{println("Column1:"+f(0)+",Column2:"+f(1))})
Column1:One,Column2:1
Column1:Col1,Column2:Col2
Column1:One,Column2:1
Column1:Eleven,Column2:11
```

Spark RDD

Load CSV File into RDD

- Skip header from csv file
- Command: rdd.mapPartitionsWithIndex { (idx, iter) => if (idx == 0) iter.drop(1) else iter }

```
scala> var csvData = textData.mapPartitionsWithIndex { (idx, iter) => if (idx == 0) iter.drop(1) else iter }
csvData: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[29] at mapPartitionsWithIndex at <console>:29

scala> csvData.foreach(f=>{println("Column1:"+f(0)+",Column2:"+f(1))})
Column1:C,Column2:o
Column1:O,Column2:n
Column1:E,Column2:l
```

Spark RDD Operations

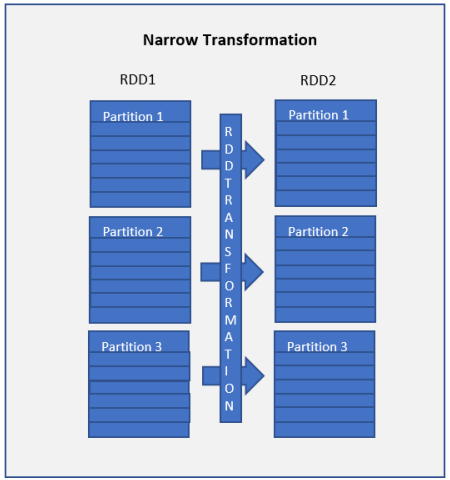
Two type of RDD operations

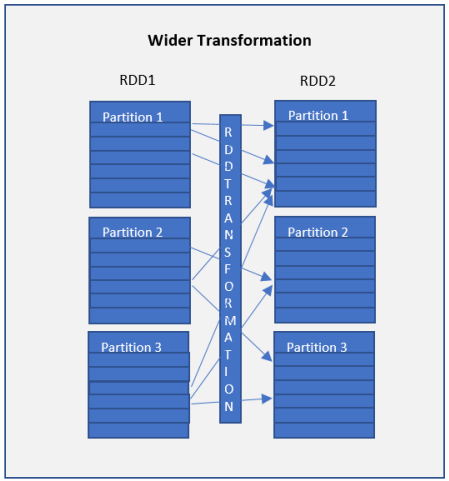
- Apache Spark RDD supports two types of Operations-
    - Transformations
    - Actions
- **Spark Transformation** is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation.
- **RDD actions** are operations that return the raw values, In other words, any RDD function that returns other than RDD is considered as an action in spark programming.

Spark RDD Operations

RDD Transformation

- Two types of RDD Transformation
  - **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.
  - **Wider transformations** are the result of groupByKey() and reduceByKey() functions and these compute data that live on many partitions meaning there will be data movements between partitions to execute wider transformations.





Spark RDD Operations

RDD Transformation – Word count example

- Create a .txt file and put it to HDFS

```
[cloudera@quickstart ~]$ cat > test.txt
This is a test file for testing RDD transformation first line
This is the second line of the same file
[cloudera@quickstart ~]$ hdfs dfs -put test.txt
```

Spark RDD Operations

RDD Transformation – Word count example

- Read the file and create an RDD[String]

```
scala> var testfile = sc.textFile("/user/cloudera/test.txt")
testfile: org.apache.spark.rdd.RDD[String] = /user/cloudera/test.txt MapPartitionsRDD[3] at textFile at <console>:27
```

Spark RDD Operations

RDD Transformation – Word count example

- Apply **flatMap()** Transformation
- flatMap() transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
scala> var testfile2 = testfile.flatMap(f=>f.split(" "))
testfile2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at flatMap at <console>:29
```

Spark RDD Operations

RDD Transformation – Word count example

- Apply **map()** Transformation
- map() transformation is used the apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.
- In this word count example, we are adding a new column with value 1 for each word, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

```
scala> var testfile3 = testfile2.map(m=>(m,1))
testfile3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[5] at map at <console>:31
```

Spark RDD Operations

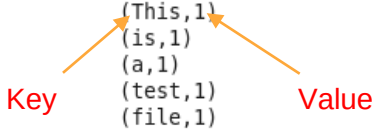
RDD Transformation – Word count example

- Let's print out the content of these RDDs (each record is printed in a single line)

```
scala> testfile.foreach(println)
This is a test file for testing RDD transformation first line
This is the second line of the same file
```

```
scala> testfile2.foreach(println)
This
is
a
test
file
for
testing
RDD
transformation
first
line
This
is
the
second
line
of
the
same
file
```

```
scala> testfile3.foreach(println)
(This,1)
(is,1)
(a,1)
(test,1)
(file,1)
(for,1)
(testing,1)
(RDD,1)
(transformation,1)
(first,1)
(line,1)
(This,1)
(is,1)
(the,1)
(second,1)
(line,1)
(of,1)
(the,1)
(same,1)
(file,1)
```



Spark RDD Operations

RDD Transformation – Word count example

- Apply **reduceByKey()** Transformation
- reduceByKey() merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on **value**. The result of our RDD contains unique words and their count.

```
scala> var count = testfile3.reduceByKey(_ + _)
count: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[6] at reduceByKey at <console>:33
```

Spark RDD Operations

RDD Transformation – Word count example

- Let’s print out the result

```
scala> count.foreach(println)
(is,2)
(second,1)
(same,1)
(a,1)
(line,2)
(This,2)
(first,1)
(testing,1)
(of,1)
(transformation,1)
(for,1)
(RDD,1)
(file,2)
(the,2)
(test,1)
```

Spark RDD Operations

RDD Transformation – Word count example

- Try **sortByKey()** Transformation
- sortByKey() transformation is used to sort RDD elements on **key**.
- In this example, we want to sort RDD elements on the number of each word. Therefore, we need convert RDD[(String,Int]) to RDD[(Int,String]) using map transformation and then apply sortByKey which ideally does sort on an integer key value.

```
scala> var count2 = count.map(a=>(a._2,a._1))
count2: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[13] at map at <console>:35
```

Spark RDD Operations

RDD Transformation – Word count example

- Let's print out the result (and see that the count number of each word now become the **key**)

```
scala> count.foreach(println)
(is,2)
(second,1)
(same,1)
(a,1)
(line,2)
(This,2)
(first,1)
(testing,1)
(of,1)
(transformation,1)
(for,1)
(RDD,1)
(file,2)
(the,2)
(test,1)
```

```
scala> count2.foreach(println)
(2,is)
(1,second)
(1,same)
(1,a)
(2,line)
(2,This)
(1,first)
(1,testing)
(1,of)
(1,transformation)
(1,for)
(1,RDD)
(2,file)
(2,the)
(1,test)
```

Spark RDD Operations

RDD Transformation – Word count example

- Apply sortByKey() transformation

```
scala> var count3 = count2.sortByKey()
count3: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[14] at sortByKey at <console>:37

scala> count3.foreach(println)
(1,second)
(1,same)
(1,a)
(1,first)
(1,testing)
(1,of)
(1,transformation)
(1,for)
(1,RDD)
(1,test)
(2,is)
(2,line)
(2,This)
(2,file)
(2,the)
```

Spark RDD Operations

RDD Transformation – Word count example

- You can also apply **map** transformation to switch the word back to the key

```
scala> var count4 = count3.map(a=>(a._2,a._1))
count4: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[15] at map at <console>:39

scala> count4.foreach(println)
(second,1)
(same,1)
(a,1)
(first,1)
(testing,1)
(of,1)
(transformation,1)
(for,1)
(RDD,1)
(test,1)
(is,2)
(line,2)
(This,2)
(file,2)
(the,2)
```

Spark RDD Operations

RDD Transformation – Word count example

- You can to all these steps in one line of code

```
scala> var count5 = count.map(a=>(a._2,a._1)).sortByKey().map(a=>(a._2,a._1))
count5: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[18] at map at <console>:35

scala> count5.foreach(println)
(second,1)
(same,1)
(a,1)
(first,1)
(testing,1)
(of,1)
(transformation,1)
(for,1)
(RDD,1)
(test,1)
(is,2)
(line,2)
(This,2)
(file,2)
(the,2)
```

Spark RDD Operations

RDD Transformation – Word count example

- Apply **filter()** Transformation
- filter() transformation is used to filter the records in an RDD. In this example we are filtering all words starts with “t”.

```
scala> var count6 = count.filter(a=>a._1.startsWith("t"))
count6: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[21] at filter at <console>:35

scala> count6.foreach(println)
(testing,1)
(transformation,1)
(the,2)
(test,1)
```

Spark RDD Operations

RDD Transformation – Word count example

- Merge two RDD

```
scala> var count7 = count.filter(a=>a._1.startsWith("T"))
count7: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[22] at filter at <console>:35

scala> var merged = count6.union(count7)
merged: org.apache.spark.rdd.RDD[(String, Int)] = PartitionerAwareUnionRDD[23] at union at <console>:39

scala> merged.foreach(println)
(testing,1)
(transformation,1)
(the,2)
(test,1)
(This,2)
```

Spark RDD Operations

RDD Transformation

cache()	Caches the RDD
filter()	Returns a new RDD after applying filter function on source dataset.
flatMap()	Returns flattern map 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.
map()	Applies transformation function on dataset and returns same number of elements in distributed dataset.
mapPartitions()	Similar to map, but executs transformation function on each partition, This gives better performance than map function
mapPartitionsWithIndex()	Similar to map Partitions, but also provides func with an integer value representing the index of the partition.
randomSplit()	Splits the RDD by the weights specified in the argument. For example rdd.randomSplit(0.7,0.3)

union()	Comines elements from source dataset and the argument and returns combined dataset. This is similar to union function in Math set operations.
sample()	Returns the sample dataset.
intersection()	Returns the dataset which contains elements in both source dataset and an argument
distinct()	Returns the dataset by eliminating all duplicated elements.
repartition()	Return a dataset with number of partition specified in the argument. This operation reshuffles the RDD randamly, It could either return lesser or more partioned RDD based on the input supplied.
coalesce()	Similar to repartition by operates better when we want to the decrease the partitions. Betterment acheives by reshuffling the data from fewer nodes compared with all nodes by repartition.

Spark RDD Operations

RDD Action

- Let’s create some RDD

```
scala> var inputrdd = sc.parallelize(List(("Z",1),("A",2),("B",30),("C",40),("B",30),("B",60)))
inputrdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[24] at parallelize at <console>:27

scala> var listrdd = sc.parallelize(List(1,2,3,4,5,3,2))
listrdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[25] at parallelize at <console>:27
```

Spark RDD Operations

RDD Action

- aggregate – action
- **aggregate()** the elements of each partition, and then the results for all the partitions.

```
scala> def param0 = (accu:Int, v:Int) => accu + v
param0: (Int, Int) => Int

scala> def param1 = (accu1:Int,accu2:Int) => accu1 + accu2
param1: (Int, Int) => Int

scala> println("aggregate : "+listrdd.aggregate(0)(param0,param1))
aggregate : 20

scala> def param3= (accu:Int, v:(String,Int)) => accu + v._2
param3: (Int, (String, Int)) => Int

scala> def param4= (accu1:Int,accu2:Int) => accu1 + accu2
param4: (Int, Int) => Int

scala> println("aggregate : "+inputrdd.aggregate(0)(param3,param4))
aggregate : 163
```

Spark RDD Operations

RDD Action

- **reduce()** - Reduces the elements of the dataset using the specified binary operator.

```
scala> println("reduce : "+listrdd.reduce(_ + _))
reduce : 20

scala> println("reduce alternate : "+listrdd.reduce((x,y) => x + y))
reduce alternate : 20

scala> println("reduce : "+inputrdd.reduce((x,y) => ("Total",x._2 + y._2)))
reduce : (Total,163)
```

Spark RDD Operations

RDD Action

- **collect()** -Return the complete dataset as an **Array**.

```
scala> var data = listrdd.collect()
data: Array[Int] = Array(1, 2, 3, 4, 5, 3, 2)
```

Spark RDD Operations

RDD Action

- **count()** – Return the count of elements in the dataset.
- **countApprox()** – Return approximate count of elements in the dataset, this method returns incomplete when execution time meets timeout.
- **countApproxDistinct()** – Return an approximate number of distinct elements in the dataset.

```
scala> println("Count : "+listrdd.count)
Count : 7

scala> println("CountApprox : "+listrdd.countApprox(1200))
CountApprox : (final: [7.000, 7.000])

scala> println("CountApproxDistinct : "+listrdd.countApproxDistinct())
CountApproxDistinct : 5

scala> println("CountApproxDistinct : "+inputrdd.countApproxDistinct())
CountApproxDistinct : 5
```

Spark RDD Operations

RDD Action

- first() – Return the first element in the dataset.
- top() – Return top n elements from the dataset.
- min() – Return the minimum value from the dataset.
- max() – Return the maximum value from the dataset.
- take() – Return the first num elements of the dataset.
- takeOrdered() – Return the first num (smallest) elements from the dataset and this is the opposite of the take() action.
- takeSample() – Return the subset of the dataset in an Array.

Spark shell Web UI

Another wordcount program

```
[cloudera@quickstart ~]$ cat > sample.txt
This is a sample file to demonstrate a sample wordcount programe using Spark
[cloudera@quickstart ~]$ hdfs dfs -put sample.txt
```

```
scala> var map = sc.textFile("/user/cloudera/sample.txt").flatMap(line => line.split(" ")).map(word => (word,1));
map: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[12] at map at <console>:27

scala> var counts = map.reduceByKey(_+_);
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[13] at reduceByKey at <console>:29
```

Spark shell Web UI

Another wordcount program

- Save the output in a text file

```
scala> counts.saveAsTextFile("/user/cloudera/outputWCspark");
```

Spark shell Web UI

Another wordcount program

- Check the result

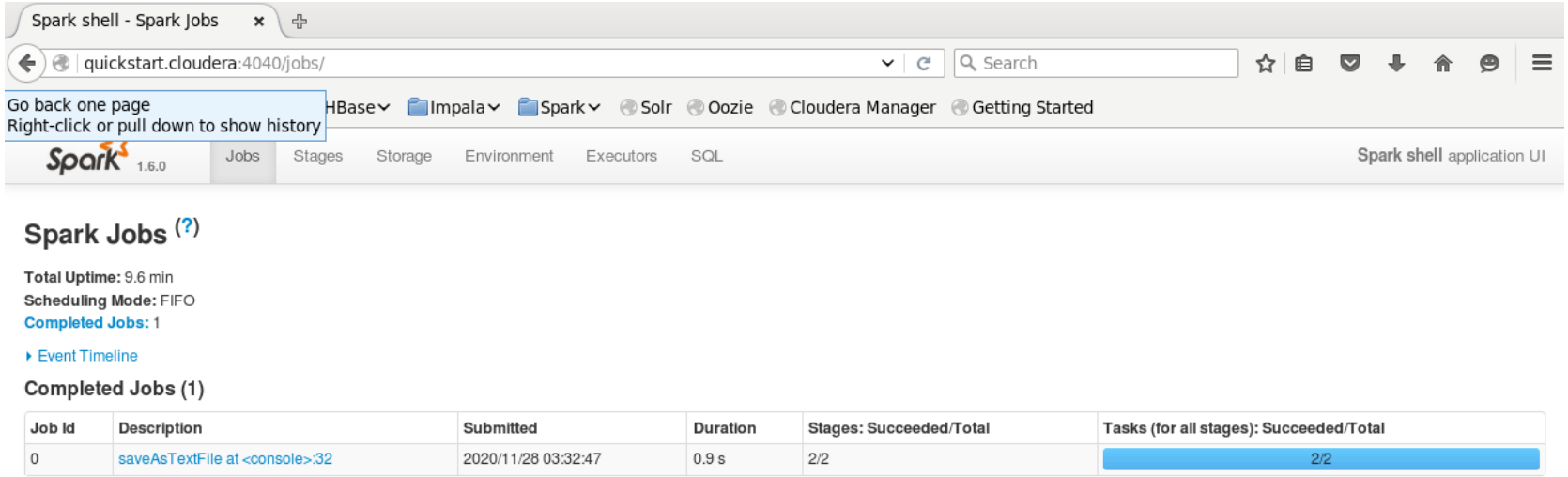
```
[cloudera@quickstart ~]$ hdfs dfs -ls
Found 7 items
-rw-r--r--   1 cloudera cloudera        217 2020-10-19 09:33 employeeDetails.txt


[cloudera@quickstart ~]$ hdfs dfs -ls outputWCspark
Found 2 items
-rw-r--r--   1 cloudera cloudera          0 2020-11-28 00:58 outputWCspark/_SUCCESS
-rw-r--r--   1 cloudera cloudera        112 2020-11-28 00:58 outputWCspark/part-00000
[cloudera@quickstart ~]$ hdfs dfs -cat outputWCspark/part*
(Spark,1)
(is,1)
(wordcount,1)
(demonstrate,1)
(a,2)
(to,1)
(This,1)
(using,1)
(programe,1)
(file,1)
(sample,2)
[cloudera@quickstart ~]$
```

Spark shell Web UI

Accessing the Web UI of Spark

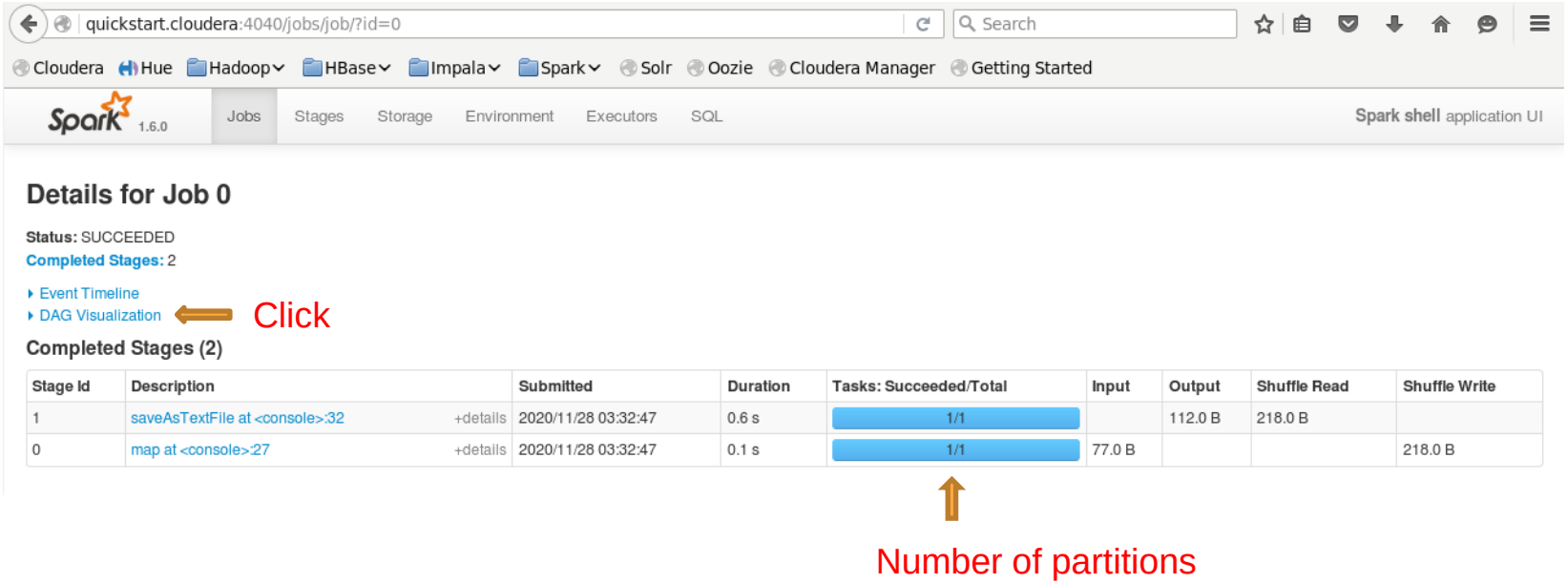
- Open http://quickstart.cloudera:4040 in web browser



Spark shell Web UI

Explore Spark shell Web UI

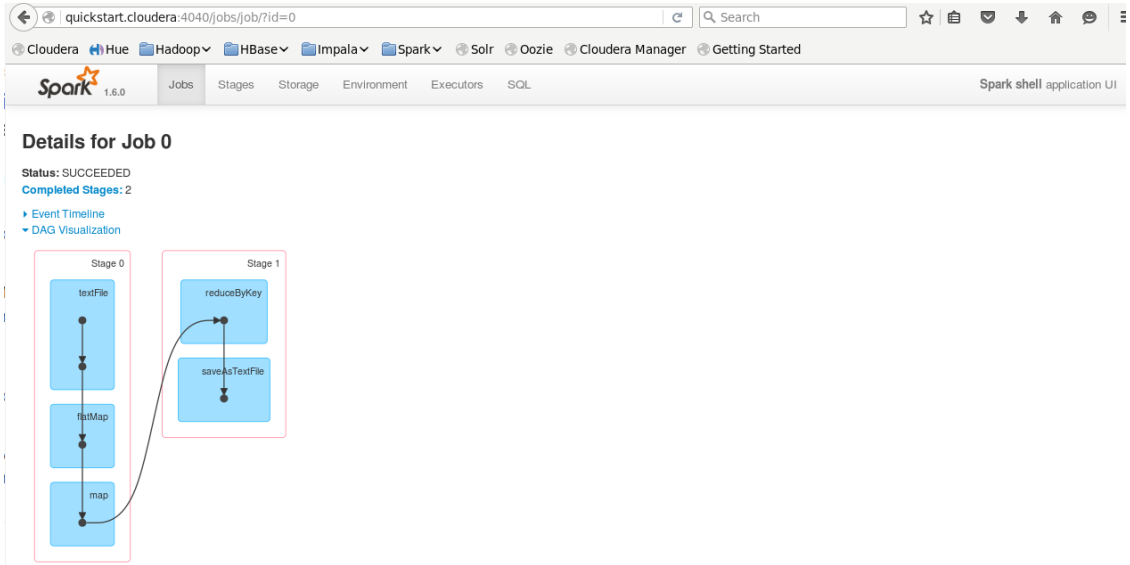
- Click DAG Visualization



Spark shell Web UI

Explore Spark shell Web UI

- View the Direct Acylic Graph (DAG) of the completed job



Spark shell Web UI

Partition and parallelism in RDDs

Now, let's understand about partitions and parallelism in RDDs.

- A ***partition*** is a *logical chunk* of a *large distributed data set.*
- By default, Spark tries to *read data into an RDD* from the *nodes* that are *close to it.*

Spark shell Web UI

Partition and parallelism in RDDs

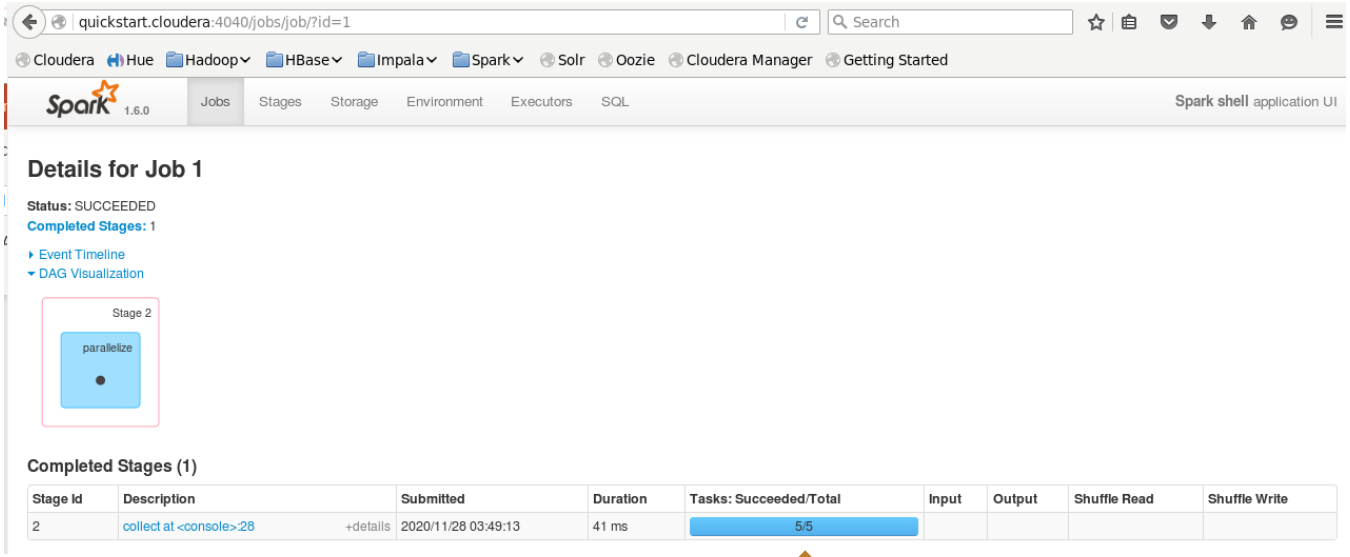
- Execute a parallel task in the shell

```
scala> sc.parallelize(1 to 100, 5).collect()
res1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

Spark shell Web UI

Partition and parallelism in RDDs

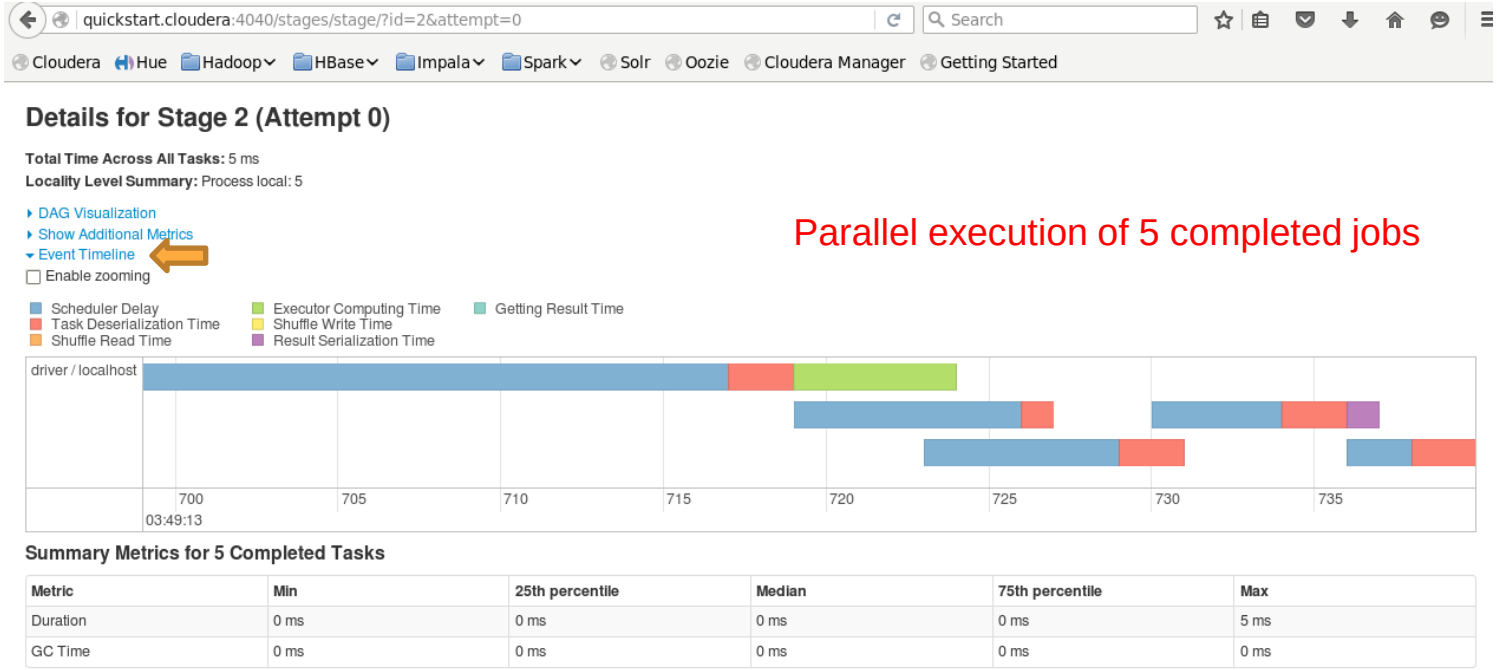
- Open Spark shell Web UI



Totally 5 partitions are done

Spark shell Web UI

Partition and parallelism in RDDs



Parallel execution of 5 completed jobs

Q & A

Cảm ơn đã theo dõi

Chúng tôi hy vọng cùng nhau đi đến thành công.