



# Big Data

(Spark)

Instructor: Trong-Hop Do

May 5<sup>th</sup> 2021

**S<sup>3</sup>Lab**

*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

- 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

      /--\
     /    \
    /  V   \
   /__/\___\
  /  /\_/\  \
 /  /\_/\  \
/  /\_/\  \
\  /\_/\  /
 \  /\_/\  /
  \  /\_/\  /
   \  /\_/\  /
    \  /\_/\  /
     \  /\_/\  /
      --\

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?

- 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
```

Note: the printed list might be unordered as foreach runs in parallel across the partitions which creates non-deterministic ordering. The action **collect()** gives you an array of the partitions concatenated in their sorted order. Or you might set the number of partitions to 1.

# 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
```

Each record in the csvData RDD is an **Array[String]**

f(0), f(1) are the first and second element in the array f

# 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> val csvData = textData.map(row => row.split(",")).mapPartitionsWithIndex{(idx,iter) => if (idx ==0) iter.drop(1) else iter}
csvData: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[7] at mapPartitionsWithIndex at <console>:25

scala> csvData.foreach( row => println("Column 1:" + row(0) + ",Column 2:" + row(1)))
Column 1:two,Column 2:2
Column 1:one,Column 2:1
```



# Spark RDD Operations

- <https://spark.apache.org/docs/latest/api/python/reference/pyspark.html>
- <https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/>

# Spark RDD Operations

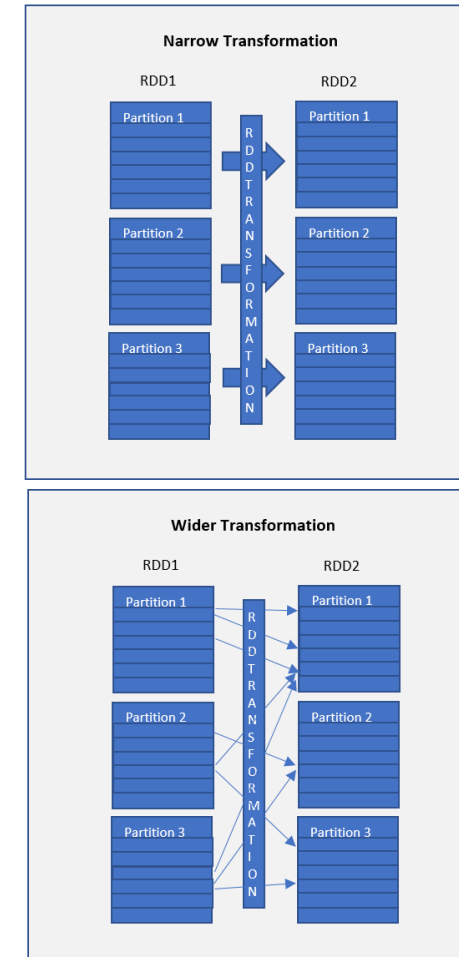
Two type of RDD operations

- 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

<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.

# Spark RDD Operations

## RDD Action

RDD ACTION METHODS	METHOD DEFINITION
<a href="#">aggregate[U](zeroValue: U)(seqOp: (U, T) =&gt; U, combOp: (U, U) =&gt; U)(implicit arg0: ClassTag[U]): U</a>	Aggregate the elements of each partition, and then the results for all the partitions.
<a href="#">collect(): Array[T]</a>	Return the complete dataset as an Array.
<a href="#">count(): Long</a>	Return the count of elements in the dataset.
<a href="#">countApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]</a>	Return approximate count of elements in the dataset, this method returns incomplete when execution time meets timeout.
<a href="#">countApproxDistinct(relativeSD: Double = 0.05): Long</a>	Return an approximate number of distinct elements in the dataset.
<a href="#">countByValue(): Map[T, Long]</a>	Return Map[T, Long] key representing each unique value in dataset and value represent count each value present.
<a href="#">countByValueApprox(timeout: Long, confidence: Double = 0.95)(implicit ord: Ordering[T] = null): PartialResult[Map[T, BoundedDouble]]</a>	Same as countByValue() but returns approximate result.
<a href="#">first(): T</a>	Return the first element in the dataset.
<a href="#">fold(zeroValue: T)(op: (T, T) =&gt; T): T</a>	Aggregate the elements of each partition, and then the results for all the partitions.
<a href="#">foreach(f: (T) =&gt; Unit): Unit</a>	Iterates all elements in the dataset by applying function f to all elements.
<a href="#">foreachPartition(f: (Iterator[T]) =&gt; Unit): Unit</a>	Similar to foreach, but applies function f for each partition.

<a href="#">reduce(f: (T, T) =&gt; T): T</a>	Reduces the elements of the dataset using the specified binary operator.
<a href="#">saveAsObjectFile(path: String): Unit</a>	Saves RDD as a serialized object's to the storage system.
<a href="#">saveAsTextFile(path: String, codec: Class[_ &lt;: CompressionCodec]): Unit</a>	Saves RDD as a compressed text file.
<a href="#">saveAsTextFile(path: String): Unit</a>	Saves RDD as a text file.
<a href="#">take(num: Int): Array[T]</a>	Return the first num elements of the dataset.
<a href="#">takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]</a>	Return the first num (smallest) elements from the dataset and this is the opposite of the take() action. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
<a href="#">takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]</a>	Return the subset of the dataset in an Array. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
<a href="#">toLocalIterator(): Iterator[T]</a>	Return the complete dataset as an Iterator. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
<a href="#">top(num: Int)(implicit ord: Ordering[T]): Array[T]</a>	Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
<a href="#">treeAggregate</a>	Aggregates the elements of this RDD in a multi-level tree pattern.
<a href="#">treeReduce</a>	Reduces the elements of this RDD in a multi-level tree pattern.

# 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 to 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 **PairRDD** 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
```

Each record in testfile3 RDD is a **tuple [String,Int]** (not an array)

# Spark RDD Operations

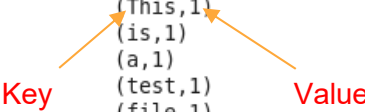
## RDD Transformation – Word count example

- 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
```

- Note: **reduceByKey()** is a transformation, so count is a RDD

# 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

- Another way to use **reduceByKey**

```
scala> testfile3.reduceByKey( (value1,value2) => value1+value2).foreach(println)
(a,1)
(is,2)
(second,1)
(same,1)
(line,2)
(This,2)
(first,1)
(testing,1)
(RDD,1)
(file,2)
(test,1)
(of,1)
(transformation,1)
(for,1)
(the,2)
```

# 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
```

Notice how each element in a tuple is accessed. If `a` is an array, the second and first element are accessed by `a(1)` and `a(0)`. In this case, `a` is a tuple, the second and first element are accessed by `a._2` and `a._1`

# 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)
```

If there are more than 1 partitions, records in each partitions are sorted, but the sorted results of multiple partitions might be mixed up.

To avoid this, use **sortByKey(numPartitions=1)**



# 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 do 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

- Or you can use **sortBy**

```
scala> count.sortBy(_._2,numPartitions=1).foreach(println)
(second,1)
(same,1)
(first,1)
(testing,1)
(RDD,1)
(test,1)
(a,1)
(of,1)
(transformation,1)
(for,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 Action – Easiest Wordcount using **countByValue**

```
scala> testfile2.foreach(println)
```

```
This  
is  
the  
second  
line  
of  
the  
same  
file  
This  
is  
a  
test  
file  
for  
testing
```

```
scala> testfile2.countByValue.foreach(println)
```

```
(for,1)  
(test,1)  
(is,2)  
(This,2)  
(line,2)  
(a,1)  
(transformation,1)  
(testing,1)  
(second,1)  
(same,1)  
(RDD,1)  
(first,1)  
(file,2)  
(of,1)  
(the,2)
```

# 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

- **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 Pair RDD Functions

## Pair RDD Transformation

Spark defines **PairRDDFunctions** class with several functions to work with Pair RDD or RDD key-value pair

PAIR RDD FUNCTIONS	FUNCTION DESCRIPTION
aggregateByKey	Aggregate the values of each key in a data set. This function can return a different result type than the values in input RDD.
combineByKey	Combines the elements for each key.
combineByKeyWithClassTag	Combines the elements for each key.
flatMapValues	It's flatten the values of each key without changing key values and keeps the original RDD partition.
foldByKey	Merges the values of each key.
groupByKey	Returns the grouped RDD by grouping the values of each key.
mapValues	It applied a map function for each value in a pair RDD without changing keys.
reduceByKey	Returns a merged RDD by merging the values of each key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and final result will be sent to the master.

sampleByKey	Returns the subset of the RDD.
subtractByKey	Return an RDD with the pairs from this whose keys are not in other.
keys	Returns all keys of this RDD as a RDD[T].
values	Returns an RDD with just values.
partitionBy	Returns a new RDD after applying specified partitioner.
fullOuterJoin	Return RDD after applying fullOuterJoin on current and parameter RDD
join	Return RDD after applying join on current and parameter RDD
leftOuterJoin	Return RDD after applying leftOuterJoin on current and parameter RDD
rightOuterJoin	Return RDD after applying rightOuterJoin on current and parameter RDD

# Spark Pair RDD Functions

## Pair RDD Action

Spark defines **PairRDDFunctions** class with several functions to work with Pair RDD or RDD key-value pair

PAIR RDD ACTION FUNCTIONS	FUNCTION DESCRIPTION
collectAsMap	Returns the pair RDD as a Map to the Spark Master.
countByKey	Returns the count of each key elements. This returns the final result to local Map which is your driver.
countByKeyApprox	Same as countByKey but returns the partial result. This takes a timeout as parameter to specify how long this function to run before returning.
lookup	Returns a list of values from RDD for a given input key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and final result will be sent to the master.

saveAsHadoopDataset	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop JobConf object to save.
saveAsHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop OutputFormat class to save.
saveAsNewAPIHadoopDataset	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c) with new Hadoop API, It uses Hadoop Configuration object to save.
saveAsNewAPIHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses new Hadoop API OutputFormat class to save.

# Spark Pair RDD Functions

Wordcount using *reduceByKey*

```
[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 Pair RDD Functions

Wordcount using *reduceByKey*

- Save the output in a text file

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

# Spark Pair RDD Functions

Wordcount using *reduceByKey*

- 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)
(programme,1)
(file,1)
(sample,2)
[cloudera@quickstart ~]$
```



# Spark Pair RDD Functions

Another usage of *reduceByKey*

```
scala> val testRDD = sc.parallelize(List(("key1",(1,10)),("key2",(1,15)),("key2",(2,20)),("key3",(3,30))))
testRDD: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ParallelCollectionRDD[654] at parallelize at <console>:24
```

```
scala> testRDD.foreach(println)
(key1,(1,10))
(key3,(3,30))
(key2,(1,15))
(key2,(2,20))
```

```
scala> testRDD.reduceByKey( (row1,row2) => (row1._1+row2._1,row1._2+row2._2)).foreach(println)
(key3,(3,30))
(key1,(1,10))
(key2,(3,35))
```

```
scala> testRDD.reduceByKey( (row1,row2) => (row1._1*row2._2,row1._2*row2._1)).foreach(println)
(key3,(3,30))
(key1,(1,10))
(key2,(20,30))
```

# Spark Pair RDD Functions

Wordcount using **countByKey** action

```
scala> val countNonRDD = testfile3.countByKey()
countNonRDD: scala.collection.Map[String,Long] = Map(for -> 1, test -> 1, is -> 2, This -> 2, line -> 2, a -> 1,
transformation -> 1, testing -> 1, second -> 1, same -> 1, RDD -> 1, first -> 1, file -> 2, of -> 1, the -> 2)

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

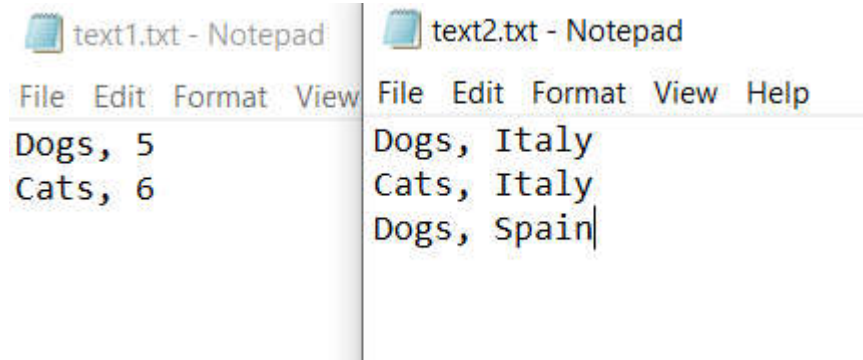
# Spark Pair RDD Functions

*Join two RDDs*

- Create two RDDs from text files

```
scala> rdd1.foreach(println)
Cats, 6
Dogs, 5

scala> rdd2.foreach(println)
Dogs, Spain
Dogs, Italy
Cats, Italy
```



# Spark Pair RDD Functions

*Join two RDDs*

- Check the result

```
scala> val data1 = rdd1.map(line => line.split(",").map(word => word.trim))
data1: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[678] at map at <console>:25
```

```
scala> val data2 = rdd2.map(line => line.split(",").map(word => word.trim))
data2: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[679] at map at <console>:25
```

```
scala> data1.map(line => (line(0),line(1))).join(data2.map(line => (line(0),line(1)))).foreach(println)
(Dogs,(5,Italy))
(Dogs,(5,Spain))
(Cats,(6,Italy))
```

If you have duplicates key in any RDD, join will result in cartesian product.

# Spark Pair RDD Functions

*Join two RDDs*

- Use **cogroup** or **groupByKey** to get unique keys in the joining result

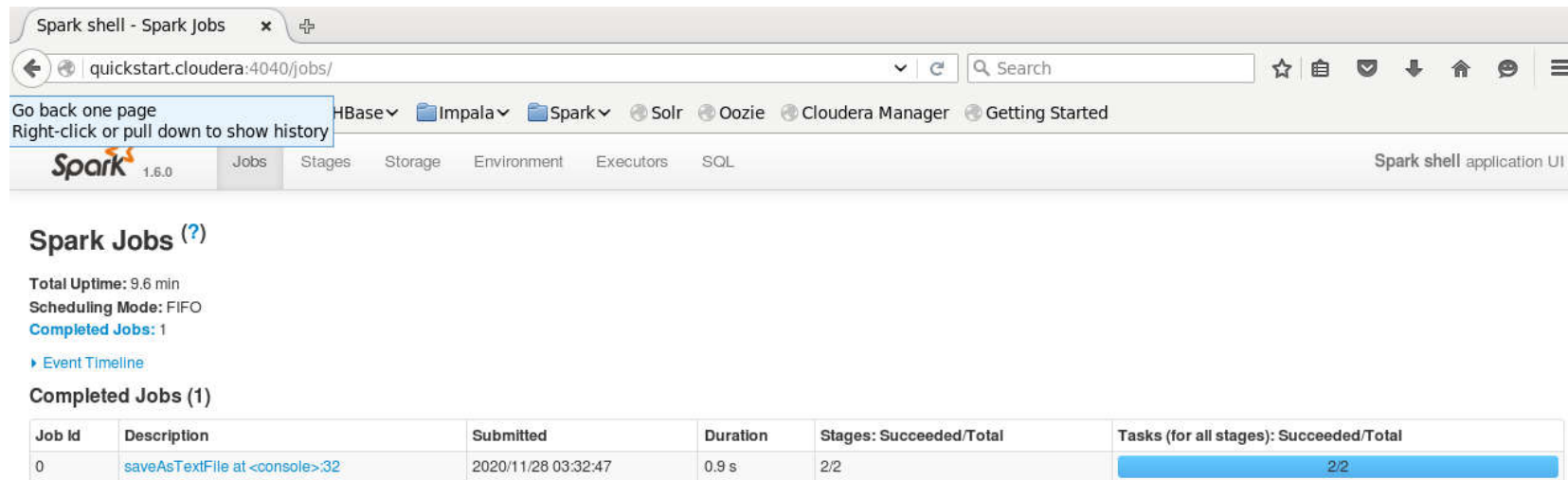
```
scala> data1.map(line => (line(0),line(1))).cogroup(data2.map(line => (line(0),line(1)))).  
map(row => (row._1,List.concat(row._2._1.toList,row._2._2.toList))).foreach(println)  
(Dogs,List(5, Italy, Spain))  
(Cats,List(6, Italy))
```

```
scala> data1.map(line => (line(0),line(1))).groupByKey().join(data2.map(line => (line(0),line(1))).  
groupByKey()).map(row => (row._1,List.concat(row._2._1.toList,row._2._2.toList))).foreach(println)  
(Dogs,List(5, Italy, Spain))  
(Cats,List(6, Italy))
```

# Spark shell Web UI

## Accessing the Web UI of Spark

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



The screenshot shows the Spark shell Web UI in a web browser. The browser tab is titled "Spark shell - Spark Jobs". The address bar shows the URL "quickstart.cloudera:4040/jobs/". The page has a navigation bar with links for HBase, Impala, Spark, Solr, Oozie, Cloudera Manager, and Getting Started. Below the navigation bar, there is a section for "Spark Jobs" with a help icon. It displays "Total Uptime: 9.6 min", "Scheduling Mode: FIFO", and "Completed Jobs: 1". A link for "Event Timeline" is also present. Below this, a section titled "Completed Jobs (1)" contains a table with job details.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	saveAsTextFile at <console>:32	2020/11/28 03:32:47	0.9 s	2/2	2/2

# Spark shell Web UI

Explore Spark shell Web UI

- Click DAG Visualization

quickstart.cloudera:4040/jobs/job?id=0

Cloudera Hue Hadoop HBase Impala Spark Solr Oozie Cloudera Manager Getting Started

Spark 1.6.0 Jobs Stages Storage Environment Executors SQL Spark shell application UI

### Details for Job 0

Status: SUCCEEDED  
Completed Stages: 2

▶ Event Timeline  
▶ DAG Visualization ← Click

#### Completed Stages (2)

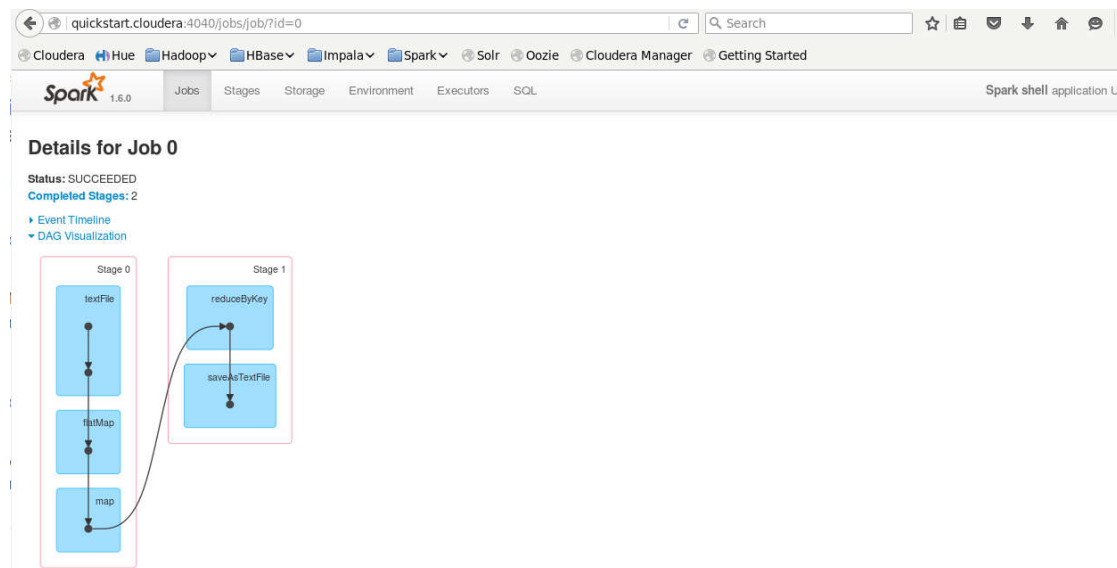
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	saveAsTextFile at <console>:32	2020/11/28 03:32:47	0.6 s	1/1		112.0 B	218.0 B	
0	map at <console>:27	2020/11/28 03:32:47	0.1 s	1/1	77.0 B			218.0 B

Number of partitions

# Spark shell Web UI

Explore Spark shell Web UI

- View the Direct Acyclic 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*

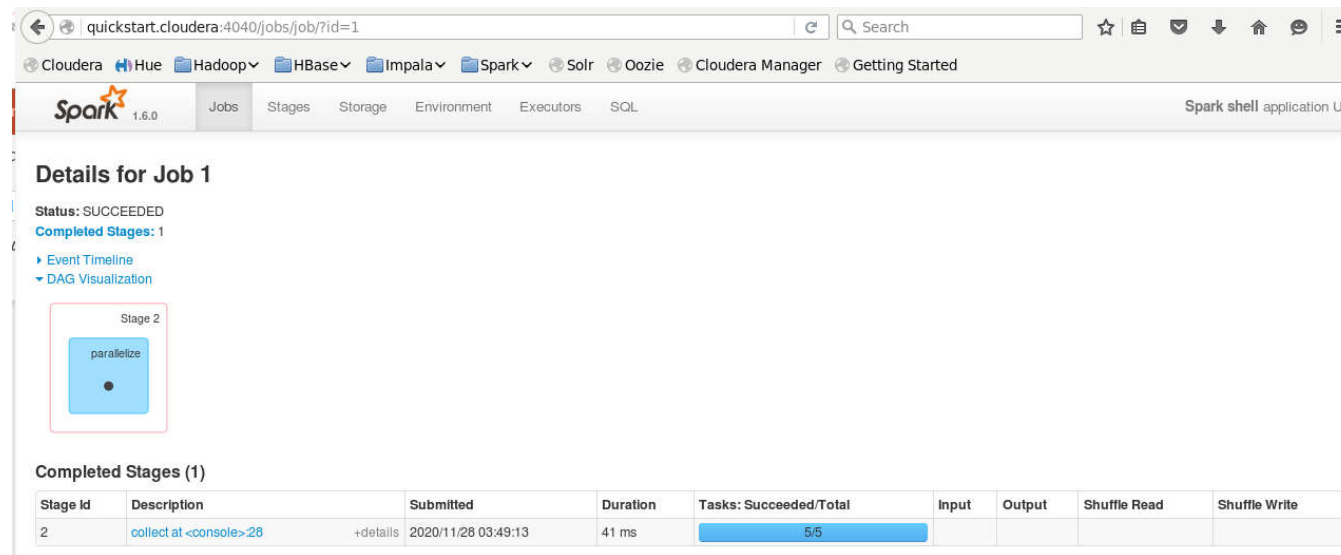
- Execute a parallel task in the shell

```
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



quickstart.cloudera:4040/jobs/job?id=1

Cloudera Hue Hadoop HBase Impala Spark Solr Oozie Cloudera Manager Getting Started

Spark 1.6.0 Jobs Stages Storage Environment Executors SQL Spark shell application UI

### Details for Job 1

Status: SUCCEEDED  
Completed Stages: 1

Event Timeline  
DAG Visualization

Stage 2

parallelize

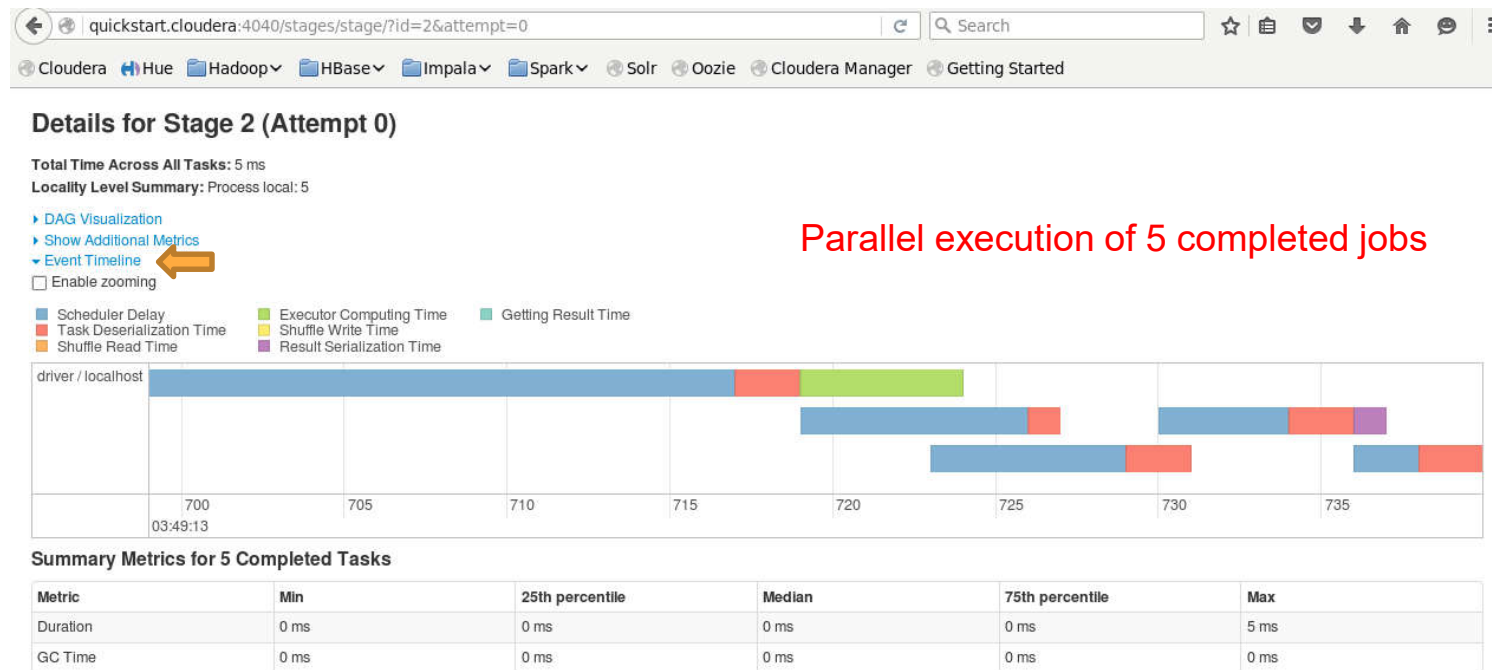
### Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	collect at <console>:28	2020/11/28 03:49:13	41 ms	5/5				

↑  
Totally 5 partitions are done

# Spark shell Web UI

## Partition and parallelism in RDDs



## Q & A



## Cảm ơn đã theo dõi

Chúng tôi hy vọng cùng nhau đi đến thành công.