|  |  |  |
| --- | --- | --- |
| **S.No** | **Topic** | **Command, Coding & Instruction** |
|  |  | **Github code** |
| 1 | ITversity | <https://github.com/dgadiraju/code/blob/master/hadoop/edw/cloudera/spark/spark_demo_scala.txt> |
|  | Spark History Server | Ipaddress:18088  http://bda1node03.k12.com:18088/ |
|  |  |  |
|  |  |  |
|  |  | Scala – Command |
| 1 | Login to scala | * scala |
| 2 | To get help | * :help |
| 3 | To quit from scala | * :quit |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | **SBT – Steps** |
| **\*** | SBT Installation | Refer “QuickRef\_Scala\_SBT” |
| 1 | **Method2:** First Create a project from CMD using SBT and import into Eclipse (Use SBT to generate eclipse project)  For method 1 (refer quick\_ref\_scala\_SBT)  For *plugins.sbt* (refer quick\_ref\_scala\_SBT)  *<pre-request>*  $scala –version  Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_121). | 1. Go to working directory and create directory for project. Run> mkdir simple-scala-sbt 2. Go to the folder “simple-scala-sbt” & Run > mkdir –p /src/main/scala 3. Go to the folder “simple-scala-sbt/src/main” & Run >mkdir resources 4. # so the folder main has two sub folders as scala & resources 5. Go to the folder “simple-scala-sbt” & Run> vi build.sbt   name := "simple-scala-sbt"  version := "1.0"  scalaVersion := "2.11.8"  libraryDependencies += "org.apache.spark" % "spark-core\_2.10" % "1.6.2"  libraryDependencies += "com.typesafe" % "config" % "1.3.0"   1. # After any change to the “build.sbt” run “sbt package” command, so that the plugin will be available for our project in scala IDE (Eclipse) 2. Run > sbt package (or) sbt compile *## It will create the target directory and it will have .Jar* 3. Run > sbt eclipse *# This command works only if we have “plugin.sbt”, this is one time setup* 4. Now project layout is ready, this is called scaffolding. It is nothing but process of creating necessary directory, this is a standard term. 5. Use Scala for IDE to import the project in Eclipse. Go to Eclipse -> File -> Import -> General -> Existing project in workspace -> Next button -> “Choose the root directory” simple-scala-sbt -> Finish -> “ Now the project has been imported into eclipse” -> “But src/main/scala doesn’t have any program, because it is brand new build” -> create a code (or) copy paste the code under src/main/scala -> Save , Right click , Run as scala application 6. Run> sbt ~compile *# Also when SBT & Eclipse are integrated, go to “simple-scala-sbt” and run this command, it will compile the code & Refresh the Jar file, whenever you make changes to the program in Eclipse IDE and save it.* 7. *Note: sbt ~compile, command will keep on running, so kill it by Ctrl+C and run whenever required* 8. Run>sbt "run-main heloworld" *# Go to “simple-scala-sbt” folder and run the program, it will identify the main function and run* 9. ## on hold ## Test it and build it using Run > sbt package |
| 2 | **Spark plugin** in build.sbt | libraryDependencies += "org.apache.spark" % "spark-core\_2.10" % "1.6.2"  so that we can import the following packages in the program:  *import org.apache.spark.SparkContext*  *import org.apache.spark.SparkConf* |
| 3 | **Scala plugin** in build.sbt  *<com.typesafe>*  *Setup “****External properties****” instead of passing unnecessary arguments* | libraryDependencies += "com.typesafe" % "config" % "1.3.0"  For more information, go to google and search for “github typesafe config” and refer above youtube link  <https://github.com/typesafehub/config>  **The benefit of com.typesafe**  There are some cases where we need to keep some “External properties” outside of our JAR file (ex: user, password, etc). In that case, we have to create a common path for Dev/QA/Prod environment , which is outside our class path and provide fully qualified file path to load  so that we can import the following packages in the program:  *import com.typesafe.config.\_* |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | **Spark – General Information** |
| 1 | **Hadoop API/**  **Spark API**  *<refer the topic “Hadoop API/Spark API” in quick\_ref\_spark”>* | 1. Online reference   <https://hadoop.apache.org/docs/r2.7.1/api/index.html> |
| 1.1 | **Frequently used APIs** | import org.apache.spark.SparkContext  import org.apache.spark.SparkConf  import org.apache.spark.sql.hive.HiveContext  import org.apache.hadoop.mapreduce.lib.output  import org.apache.hadoop.io |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | **Spark shell commands** |
| 1 | login | * Spark-shell |
| 2 | To run spark | * Spark-submit |
| 2.1 | To get help | To print all options run   * Spark-submit --help |
| 3 | Spark context | * Scala> sc *# Type sc , it will show the spark context info as follows*   Ex: res0: org.apache.spark.SparkContext = [org.apache.spark.SparkContext@7b1559f1](mailto:org.apache.spark.SparkContext@7b1559f1) |
| 4 | Define variable in Spark using Scala | There are two ways to define variable   1. **Val *##*** *Immutable variable. Can redefine but not able to change* 2. **Var** *## Mutable variable*   **Ex: Val**   * scala> val i=0 * i: Int = 0 * scala> i=1 * <console>:32: error: reassignment to val * i=1 * ^   **Ex: Var**   * scala> var j=1 * j: Int = 1 * scala> j=2 * j: Int = 2 |
| 5 | **Tip:** get all the classes under an API | ## Use \_ after dot, it is like \*  import org.apache.spark.\_ |
| 6 | Tip: for auto fill | * Spark-shell * Import org. <<press tab>> it will show all the available options * val data = sc. <<press tab>> it will show all the options available under sc |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | **K12 – Scripts** |
| 1 | SVN path for bkt calculator | <http://subversion.k12.com/svn/appdev/k12_dev/adaptive/bda-config-files/> |
| 1.1 | To submit a spark job for bkt\_calculator | spark-submit --class com.k12.bkt.calculator.BKTJob --master yarn \  --conf spark.yarn.maxAppAttempts=4 \  --conf spark.yarn.am.attemptFailuresValidityInterval=1h \  --conf 'spark.yarn.executor.memoryOverhead=15000' \  --conf 'spark.eventLog.enabled=true' bkt-calculator-1.0.0-20170220.190231-29-jar-with-dependencies.jar bkt\_calculator.ini |
| 1.2 | bkt\_calculator.ini | input\_tables\_schema=recommender  output\_tables\_schema=recommender  input\_table=assessment\_item\_outcome\_student\_objective\_sp  output\_table=proficiencyIn |
| 1.3 | deploy\_bkt\_calculator.sh | echo "Deploying BKT Calculator (Spark JOB)"  hdfs dfs -mkdir hdfs://k12prodbdacluster-ns/user/oracle/bkt\_calculator  hdfs dfs -copyFromLocal -f /home/oracle/workspace/adaptive/bkt\_calculator/bkt-calculator-1.0.0-20170220.190231-29-jar-with-dependencies.jar hdfs://k12prodbdacluster-ns/user/oracle/bkt\_calculator/  hdfs dfs -copyFromLocal -f /home/oracle/workspace/adaptive/bda-config-files/bkt\_calculator/bkt\_calculator.ini hdfs://k12prodbdacluster-ns/user/oracle/bkt\_calculator/  hdfs dfs -copyFromLocal -f /home/oracle/workspace/adaptive/bda-config-files/execute\_bkt\_calculator.sh hdfs://k12prodbdacluster-ns/user/oracle/bkt\_calculator/  hdfs dfs -chmod -R 777 hdfs://k12prodbdacluster-ns/user/oracle/bkt\_calculator/  echo "Deploying Process for BKT Calculator was completed" |
| 1.4 | deploy\_recommender-activity.sh | spark-submit --conf spark.executor.extraClassPath="/opt/cloudera/parcels/CDH-5.7.0-1.cdh5.7.0.p1464.1349/lib/hive/lib/\*" \  --jars $(echo /opt/cloudera/parcels/CDH-5.7.0-1.cdh5.7.0.p1464.1349/jars/\*.jar | tr ' ' ',') \  --master yarn \  --queue realtime\_queue \  --conf spark.hadoop.fs.hdfs.impl.disable.cache=true \  --conf spark.yarn.maxAppAttempts=4 \  --conf spark.yarn.am.attemptFailuresValidityInterval=1h \  --class com.k12.nextgen.consumer.ActivityVisitEventConsumer /home/oracle/workspace/adaptive/adaptive-kafka-consumers/adaptive-kafka-consumers-1.0.0-20170209.204150-10.jar /home/oracle/workspace/adaptive/bda-config-files/recommender\_activityvisitevent\_stream/recommender-activity.ini > /tmp/recommender/recommender\_activityvisitevent\_stream.log 2>&1 & |
| 1.5 | deploy\_recommender-outcome.sh | spark-submit --conf spark.executor.extraClassPath="/opt/cloudera/parcels/CDH-5.7.0-1.cdh5.7.0.p1464.1349/lib/hive/lib/\*" \  --jars $(echo /opt/cloudera/parcels/CDH-5.7.0-1.cdh5.7.0.p1464.1349/jars/\*.jar | tr ' ' ',') \  --master yarn \  --queue realtime\_queue \  --conf spark.hadoop.fs.hdfs.impl.disable.cache=true \  --conf spark.yarn.maxAppAttempts=4 \  --conf spark.yarn.am.attemptFailuresValidityInterval=1h \  --class com.k12.nextgen.consumer.OutcomeResultConsumer /home/oracle/workspace/adaptive/adaptive-kafka-consumers/adaptive-kafka-consumers-1.0.0-20170209.204150-10.jar /home/oracle/workspace/adaptive/bda-config-files/recommender\_outcomeresults\_stream/recommender-outcome.ini > /tmp/recommender/recommender\_outcomeresults\_stream.log 2>&1 & |
| 1.6 | deploy\_recommender  \_algorthm.sh | echo "Deploying Recommender Algorithm (Hive Scripts)"  svn up /home/oracle/workspace/recommender/  hdfs dfs -mkdir hdfs://k12prodbdacluster-ns/user/oracle/recommender/  hdfs dfs -copyFromLocal -f /home/oracle/workspace/recommender/hive/\*.sql hdfs://k12prodbdacluster-ns/user/oracle/recommender/  hdfs dfs -copyFromLocal -f /home/oracle/workspace/recommender/automation/mongo-integration/\*.hql hdfs://k12prodbdacluster-ns/user/oracle/recommender/  echo "Deploying Process for Recommender Algorithm was completed" |
| 1.7 | recommender\_activityvisitevent  \_stream /recommender-activity.ini | # Kafka details  zookeeperAddress=eqa-bda2node03.k12.com:2181  kafkaGroup=rec\_activity\_prod\_consumer\_ash  kafkaTopic=activity\_visit\_events  kafkaConsumerParallelism=5  # Process setup  consumerTimeWindowMinutes=1  table=recommender.activity\_visit\_student\_objective  appName=[PROD] Recommendation - ActivityVisitEvent - Kafka Consumer |
| 1.8 | recommender\_outcomeresults \_stream/recommender-outcome.ini | # Kafka details  zookeeperAddress=eqa-bda2node03.k12.com:2181  kafkaGroup=rec\_outcome\_prod\_consumer\_ash  kafkaTopic=outcome\_item\_results  kafkaConsumerParallelism=5  # Process setup  consumerTimeWindowMinutes=1  table=recommender.assessment\_item\_outcome\_student\_objective  appName=[PROD] Recommendation - OutcomeResultEvent - Kafka Consumer |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | **Use Cases - Scala, SBT using Eclipse** |
| \* | **Use case:** Understand the significance of => symbol  *<Ex: Syntactic Sugar>* | <http://stackoverflow.com/questions/6951895/what-does-and-mean-in-scala>  => is syntactic sugar for creating instances of functions. Recall that every function in scala is an instance of a class.  For example, the type Int => String, is equivalent to the type Function1[Int,String] i.e. a function that takes an argument of type Int and returns a String. |
| \* | **Use case:** Understand the significance of => symbol with example | **Example1:**   * spark-shell * val j= List(1,2,3,4,5)   j: List[Int] = List(1, 2, 3, 4, 5)   * j.map(x=> x\*2)   res0: List[Int] = List(2, 4, 6, 8, 10)  **Example2:** |
| \* | **Use case:** Invoke the functions | **Example1:**   * def sum(a:Int, b:Int):Int = (a+b)   sum: (a: Int, b: Int)Int   * sum(4,8)   res1: Int = 12  **Example2:**   * val sum1=(a:Int,b:Int) => (a+b)   sum1: (Int, Int) => Int = <function2>   * sum1(4,8)   res2: Int = 12 |
| **1** | **Use Case1:** Word count program in Eclipse  <eclipse import is not working due to some configuration issue in my laptop with windows> | **# Create a word count program with “External Properties”. Here External properties will hold the “setMaster” value. Instead of hardcode to local (or) yarn-client, it will be stored in External properties. So that depends upon the environment we run, it will change to local/yarn-client automatically**  val conf = new SparkConf().  setAppName("Word Count").  setMaster("**local**")   1. Go to “/cygdrive/c/Kumar/Java/scala-sbt/word-count-sbt” ## Work space 2. $mkdir word-count-sbt and $cd word-count-sbt 3. $mkdir -p src/main/scala 4. $cd src/main and $mkdir resources and back to “word-count-sbt” folder 5. # main folder has two sub folders now, scala & resources 6. $vi build.sbt   name := "word-count"  version := "1.0"  scalaVersion := "2.11.8"  libraryDependencies += "org.apache.spark" % "spark-core\_2.10" % "1.6.2"  libraryDependencies += "com.typesafe" % "config" % "1.3.0"   1. $sbt eclipse 2. Use Scala for IDE to import the project in Eclipse. Go to Eclipse -> File -> Import -> General -> Existing project in workspace -> Next button -> “Choose the root directory” simple-scala-sbt -> Finish -> “ Now the project has been imported into eclipse”   **Known issue:** Getting the workspace already exists issue. Because .classpath is referring to “/cygdrive/c/users/sarkumar” directory, which is an issue in my windows configuration   1. $sbt package ## Do NOT use ‘s’ at the end (packages) 2. Sdf   **TBD**   1. scp simple-spark\_2.10-1.0.jar [saranvisa@gw01.itversity.com:~](mailto:saranvisa@gw01.itversity.com:~) 2. Note: In the above command ~ refers to home directory, or we can use fully qualified path as follows /home/saranvisa/filepath 3. Note: The jar file has been copied in local (not in HDFS) but the output of Jar will be stored in HDFS 4. Run the below command in linux local   spark-submit --class "SimpleApp" \  --master yarn \  --executor-memory 512m \  --total-executor-cores 1 \  simple-spark\_2.10-1.0.jar yarn-client /user/saranvisa/spark\_demo/simplesparkoutput   1. The output will be stored in : /user/saranvisa/spark\_demo/simplesparkoutput/part-00001 |
| **2** | **Use case 2**: write a simple Scala program in Linux and build a Jar using SBT.  CCA175 : Load data from HDFS and store results back to HDFS using Spark  **Method1:** build a JAR using SBT and execute the JAR  Project name: Simple project | 1. <https://www.youtube.com/watch?v=iclGhV3s98o&index=53&list=PLf0swTFhTI8rJvGpOp-LujOcpk-Rlz-yE> 2. Pre-request, make sure SBT is installed in Linux. For more details about installation refer “Quick\_Ref\_Scala\_SBT” 3. Go to this link and get content for simple.sbt This explains the dependencies for Spark   <http://spark.apache.org/docs/1.2.1/quick-start.html>  name := "Simple Project"  version := "1.0"  scalaVersion := "2.10.4"  libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.1"   1. mkdir -p ./src/main/scala 2. vi SimpleApp.scala   [saranvisa@gw01 scala]$ cat SimpleApp.scala  import org.apache.spark.SparkContext, org.apache.spark.SparkConf  import org.apache.spark.SparkContext, org.apache.spark.SparkConf  object SimpleApp {  def main(args: Array[String]) {  val conf = new SparkConf().setAppName("scala spark")  val sc = new SparkContext(conf)  val dataRDD = sc.textFile("/apps/hive/warehouse/qa.db/orders") ## Should be HDFS path  dataRDD.saveAsTextFile("/user/saranvisa/scala-sbt-output/orders") ## Should be **unavailable** HDFS path  }  }   1. run> sbt package *# from the parent path “simple project”* 2. run> spark-submit --class “SimpleApp” \   --master local \  Taget… copy past the path -- TBD   1. sdf |
| 2.1 | Method2: Run the command directly in the spark-shell | 1. >spark-shell # login to spark 2. # Create RDD based on a folder contains data   scala> val dataRDD = sc.textFile("/apps/hive/warehouse/qa.db/orders")   1. # Create RDD based on a textFile   scala> val dataRDD1 = sc.textFile("/user/saranvisa/scala-sbt-input/textfile.txt")   1. scala> dataRDD.count() 2. scala> dataRDD1.count() 3. ## Should be unavailable HDFS path   scala> dataRDD.saveAsTextFile("/user/saranvisa/scala-sbt-output/orders")   1. ## Should be unavailable HDFS path   scala> dataRDD1.saveAsTextFile("/user/saranvisa/scala-sbt-output/file1") |
| 2.2 | Tips: To view the directory structure | In the above parent path “simple project” run the below find . command, it will show the directory structure as follows  [saranvisa@gw01 simple-project]$ find .  .  ./src  ./src/main  ./src/main/scala  ./src/main/scala/SimpleApp.scala  ./simple.sbt |
| **3** | **Use Case 3:** **RDD**: Simple Commands in RDD | *# Create RDD from a file & Folder and save into another folder*   1. Login to spark   >spark-shell   1. Create RDD based on a folder contains data   scala> val dataRDD = sc.textFile("/apps/hive/warehouse/qa.db/orders")   1. Create RDD based on a textFile   scala> val dataRDD1 = sc.textFile("/user/saranvisa/scala-sbt-input/textfile.txt")   1. Count records from RDD   scala> dataRDD.count()  scala> dataRDD1.count()   1. SaveAs RDD to a different location. ## Should be unavailable HDFS path   scala> dataRDD.saveAsTextFile("/user/saranvisa/scala-sbt-output/orders")  scala> dataRDD1.saveAsTextFile("/user/saranvisa/scala-sbt-output/file1") |
| 3.1 | Create RDD based on **Fully qualified HDFS path &** Save it as **text file & Object** | 1. To get the ipaddress   Linux> uname -a   1. Create RDD based on Fully qualified HDFS path, the target file/folder should NOT be available   scala> val dataRDD2 = sc.textFile("hdfs://gw01.itversity.com:8022/user/saranvisa/scala-sbt-input/textfile.txt")   1. Save RDD as text file   scala> dataRDD2.saveAsTextFile("/user/saranvisa/scala-sbt-output/file2")   1. Save RDD as object, it will create a folder   scala> dataRDD2.saveAsObjectFile("/user/saranvisa/scala-sbt-output/file2") |
| 3.2 | To print the data from RDD | 1. Create RDD based on a folder contains data   scala> val dataRDD = sc.textFile("/apps/hive/warehouse/qa.db/orders")   1. Print the RDD   scala> dataRDD.collect().foreach(println) |
| **4** | **Use case 4: Sequence File**  *<with default KEY and VALUE>* | # To handle sequence file   1. Login to >spark-shell 2. Import required API, org.apache.hadoop.io.\_ is mandatory   import org.apache.spark.SparkContext, org.apache.spark.SparkConf  import org.apache.hadoop.mapreduce.lib.output  import org.apache.hadoop.io.\_   1. Cannot read a text file as sequence file, so read a text file and write it into to sequence file then read it as sequence file. So to read a sequence file, it has a pre-request that the source should be already a sequence file   *# Consider there is a comma separated file in path /user/saranvisa/scala-sbt-input/deptfile.txt*  val dataRDD = sc.textFile("/user/saranvisa/scala-sbt-input/deptfile.txt")  dataRDD.foreach(println)  *# Make sure the target directory is not available*  dataRDD.map(rec=>(NullWritable.get(),rec)).saveAsSequenceFile("/user/saranvisa/scala-sbt-input/dept\_seqfile")  NullWritable.get() ## refers to the default KEY  rec *## refers to the default VALUE* |
| 4.1 | **Sequence File (Write) :** Specify the column for KEY and VALUE  *<using column1 for KEY and column 2 for VALUE>* | *# To specify the KEY and VALUE instead of default*  dataRDD.map(rec=> (rec.split(",")(0), rec.split(",")(1))).saveAsSequenceFile("/user/saranvisa/scala-sbt-input/dept\_seqfile1")  rec.split(",") *## refers to comma separated file*  rec.split(",")(0) *## (0) Refers to the first column to be used for KEY*  rec.split(",")(1)  *## (1) Refers to the second column to be used for VALUE* |
| 4.2 | **Sequence File (Read) :** from an existing sequence file and few more formats – pending (starting from 15th min) | <https://www.youtube.com/watch?v=2p8jcw7troM&list=PLf0swTFhTI8rJvGpOp-LujOcpk-Rlz-yE&index=55>  import org.apache.hadoop.mapreduce.lib.output.\_  val path="/user/saranvisa/scala-sbt-input/dept\_seqfile"  sc.sequenceFile(path,classOf[IntWritable], classOf[Text]).map(rec => rec.toString()).collect().foreach(println) |
| 5 | **Use case 5:** **Access Hive table via Spark**  ***<HiveContext>*** | **Pre-request:**  Make sure hive-site.xml is present in /etc/spark/conf. Either copy pate or create a soft link  # import the HiveContext API  import org.apache.spark.sql.hive.HiveContext  # Initialize SQLContext via HiveContext  val sqlContext = new HiveContext(sc)  # With the help of SQLContext, query the table. The data will be collected and stored in testRDD  val testRDD = sqlContext.sql("select \* from qa.test1")  # Collect the RDD and print the data  testRDD.collect().foreach(println) |
| 5.1 | To create a new table in Hive using Spark and access it | # *To create a new table in Hive using Spark and access it*  sqlContext.sql("create table qa.sprk\_tbl1 as select \* from qa.test1")  val test1RDD = sqlContext.sql("select \* from qa.sprk\_tbl1")  test1RDD.collect().foreach(println) |
| 6 | **Use case 6: INSERT, LOAD a JSON file to hive table**  ***<SQLContext>*** | # We can run hive INSERT, LOAD and any valid hive query  **Pre-request:**  # Create a following JSON file in local and Make sure to copy to HDFS  {"department\_id":2, "department\_name":"Fitness"}  {"department\_id":3, "department\_name":"Footwear"}  {"department\_id":4, "department\_name":"Apparel"}  {"department\_id":5, "department\_name":"Golf"}  # program using SQLContext  import org.apache.spark.sql.SQLContext  val sqlContext = new SQLContext(sc)  val deptJson = sqlContext.jsonFile("/user/saranvisa/scala-sbt-input/dept.json")  val deptJson1 = sqlContext.jsonFile("/user/saranvisa/scala-sbt-input/dept1.json")  deptJson1.registerTempTable("deptjson")  val deptJsonData = sqlContext.sql("select \* from deptjson")  deptJsonData.collect().foreach(println) |
| 7 | **Use case 7: Get hive table and write into HDFS as JSON file** | # Get Hive table & Writing data in JSON format in HDFS  # import the HiveContext API  import org.apache.spark.sql.hive.HiveContext  # Initialize SQLContext via HiveContext  val sqlContext = new HiveContext(sc)  # With the help of SQLContext, query the table. The data will be collected and stored in testRDD  val dataRDD = sqlContext.sql("select \* from qa.emp")  dataRDD.toJSON.saveAsTextFile("/user/saranvisa/scala-sbt-input/emp.json")  hdfs dfs -cat /user/saranvisa/scala-sbt-input/emp.json/part-0000\* |
| 8 | **Use case 8: Word Count – group by word**  *<using 3* ***Transformation*** *of RDD>*   1. *flatMap* 2. *map* 3. *ReducedbyKey* | *# RDD Transformations used for word count program*   1. **flatMap(Func):** Ex: To split each line into words in the below example 2. **Map(func):** Ex: To associate numeric one with each word in the example: (Hardware,1), (Telephone,1), (Hardware,1) 3. **ReducedbyKey.** Ex: Includes combiner as well as reducer to get the value: (Hardware,2), (Telephone,1) |
| 8.1 | *Word count steps*  *To Split the words*  *RDD.flatMap(Rec=> rec.split(“,”))*  *To setup KEY VALUE (Tuple)*  *RDD.map(rec=> (rec,1))*  *Kind of Groupby*  RDD.reduceByKey(( acc, value) => acc + value)  *Acc is 0 for the first iteration and add values group by KEY*  *split by comma vs space*  *rec.split(",")*  *rec.split(" ")* | [*https://www.youtube.com/watch?v=B6sh4JI-pMc&list=PLf0swTFhTI8rJvGpOp-LujOcpk-Rlz-yE&index=57*](https://www.youtube.com/watch?v=B6sh4JI-pMc&list=PLf0swTFhTI8rJvGpOp-LujOcpk-Rlz-yE&index=57)  *# Pre-request: Move a file to HDFS. Consider this has department details. Consider it has a duplicate value ‘Hardware’*  deptid,deptname  10,HumanResource  20,InformationTechnology  30,Hardware  40,Telephone  50,Marketting  60,Security  70,Management  80,Hardware  val dataRDD = sc.textFile("/user/saranvisa/scala-sbt-input/deptdup.txt")  dataRDD.collect().foreach(println)  *## Use* ***flatMap*** *property of RDD. Rec is just a name for the function flatMap and also mentioned to split the word by comma*  *## to split by space, use the function as follows: dataRDD.flatMap(rec => rec.split(" "))*  val dataRDDflatMap = dataRDD.flatMap(rec => rec.split(","))  *# To get what is available in flatMap variable*  dataRDDflatMap.collect().foreach(println)  #The output will be as follows:  deptid  deptname  10  HumanResource  20  InformationTechnology  30  Hardware  40  Telephone  50  Marketting  60  Security  70  Management  80  Hardware  *## To get the word count, create a tuple with KEY as the word itself and VALUE as count 1. Need to use the* ***Map*** *Function for the same. Map function, will take one line at a time, so loop it*  val dataRDDmap = dataRDDflatMap.map(rec => (rec,1))  *# To get what is available in Map variable*  dataRDDmap.collect().foreach(println)  *# The output of map variable. Now we get tuple with KEY as word and VALUE as 1*  (deptid,1)  (deptname,1)  (10,1)  (HumanResource,1)  (20,1)  (InformationTechnology,1)  (30,1)  (Hardware,1)  (40,1)  (Telephone,1)  (50,1)  (Marketting,1)  (60,1)  (Security,1)  (70,1)  (Management,1)  (80,1)  (Hardware,1)  # Need to aggregate the tuple to get the word count  # Use **ReduceByKey** to group the tuple by KEY, so that it will count two Hardware  ## acc is the variable to accumulate the KEY  val dataRDDreduceByKey = dataRDDmap.reduceByKey((acc,value) => acc + value)  ## To print the word count by word  dataRDDreduceByKey.collect().foreach(println)  ## The output will be displayed as follows  (60,1)  **(Hardware,2)**  (80,1)  (20,1)  (deptname,1)  (Security,1)  (40,1)  (Marketting,1)  (InformationTechnology,1)  (deptid,1)  (Telephone,1)  (HumanResource,1)  (Management,1)  (50,1)  (30,1)  (70,1)  (10,1)  ## Save the collection as TextFile  dataRDDreduceByKey.saveAsTextFile("/user/saranvisa/scala-sbt-output/wordcountbyword") |
| **8.2** | **Word count – using ReduceByKey vs groupByKey** | <http://www.ruxizhang.com/blog/spark-difference-between-reducebykey-groupbykey>  # There is two different ways to compute counts:  val words = Array("one", "two", "two", "three", "three", "three") val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))  val wordCountsWithReduce = wordPairsRDD .reduceByKey(\_ + \_) .collect() ​ val wordCountsWithGroup = wordPairsRDD .groupByKey() .map(t => (t.\_1, t.\_2.sum)) .collect() |
| **9** | **Use case 9: Joining datasets & applying filter before join** | # **Problem statement:** Get the revenue and number of orders from order\_items on daily basis  **# ITversity doc & vids:** <http://www.itversity.com/topic/joining-data-sets-scala/>  **GitHub code**: <https://github.com/dgadiraju/code/blob/master/hadoop/edw/cloudera/spark/spark_demo_scala.txt>  GitHub topic: # Join disparate datasets together using Spark  # RDD Transformations used for joining datasets   1. **join(otherDataset, [numTasks]):** # To join two data sets. Pre-request: common KEY. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. 2. **Map(func):** Ex: To associate numeric one with each word in the example: |
| 9.1 | JOIN vs SET | Two tables are available in MySQL   1. Orders 2. Order\_item   JOIN vs SET   1. INNER = INTERSECTION 2. LEFT/RIGHT OUTER = (INTERSECTION + MINUS) 3. FULL OUTER = (LEFT OUTER union RIGHT OUTER) |
| 9.2 | Table/File structure  Of   1. Orders 2. Order\_items | **Table: Orders (File is comma separated)**  +----------+---------------------+-------------------+-----------------+  | **order\_id** | order\_date | order\_customer\_id | **order\_status** |  +----------+---------------------+-------------------+-----------------+  | 1 | 2013-07-25 00:00:00 | 11599 | CLOSED |  | 2 | 2013-07-25 00:00:00 | 256 | PENDING\_PAYMENT |  | 3 | 2013-07-25 00:00:00 | 12111 | **COMPLETE** |  | 4 | 2013-07-25 00:00:00 | 8827 | CLOSED  **Table: Order\_items**  +---------------+---------------------+-----------------------+---------------------+---------------------+--------------------------+  | order\_item\_id | **order\_item\_order\_id** | order\_item\_product\_id | order\_item\_quantity | **order\_item\_subtotal** | order\_item\_product\_price |  +---------------+---------------------+-----------------------+---------------------+---------------------+--------------------------+  | 1 | 1 | 957 | 1 | 299.98 | 299.98 |  | 2 | 2 | 1073 | 1 | 199.99 | 199.99 |  | 3 | 2 | 502 | 5 | 250 | 50 |  | 4 | 2 | 403 | 1 | 129.99 | 129.99 |  | 5 | 4 | 897 | 2 | 49.98 | 24.99 |
| 9.3 | Perform necessary transformations before joining the data.  *<Before JOIN RDD>* | [*http://www.itversity.com/topic/applying-simple-transformations-scala/*](http://www.itversity.com/topic/applying-simple-transformations-scala/)  *# perform necessary transformations before joining the data.*   * *Filter for “COMPLETED” orders* * *Extract the key from orders and order\_items (using map)* * *Get order\_date along with order\_id from orders* * *Get order\_item\_subtotal along with order\_id from order\_items* * *Get revenue per order from order\_items*   *# Create RDD with orders and order\_items*   * val filePath="/user/saranvisa/sqoop\_import" * val ordersRDD = sc.textFile(filePath + "/orders") * val orderItemsRDD = sc.textFile(filePath + "/order\_items") |
| 9.3.1 | *Transformation:* ***Filter on RDD***  *Vs*  *Transformation:* ***Filter on Tuple*** | # As part of this topic we will see how we can apply filter transformation for completed orders (**Filter** for “COMPLETED” orders)   * orders **file** has 4 columns * 4th column is order\_status * Data is comma separated * rec.split(“,”) will return an array * rec.split(“,”)(3) will get order\_status (the 4th column) * rec.split(“,”)(3).equals(“COMPLETE”) will check for completed orders * If it returns true record will be returned for downstream processing, else record will be discarded * In this case rec as well as rec.split(“,”)(3) is of type string * Hence we can use any string operations, casting functions, conditional operators etc   **Filter on RDD: (First column starts with 0)**  **Filter by String (Use case: filter only “COMPELTED” orders)**  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3) == "COMPLETE"))  --or--  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3).equals "COMPLETE"))  **Filter by Integer(Use case: Filter the order\_id =1)**  val ordersOne = ordersRDD.filter(rec => (rec.split(",")(0).toInt.equals(1)))  --or--  val ordersOne = ordersRDD.filter(rec => (rec.split(",")(0).toInt == 1))  **Filter on Tuple: (First element starts with 0)**  val filePath="/user/saranvisa/sqoop\_import"  val ordersRDD = sc.textFile(filePath + "/orders")  val orderItemsRDD = sc.textFile(filePath + "/order\_items")  ## To get ONLY "COMPLETE" orders  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3) == "COMPLETE"))  ## To Map Order\_id and Order\_Date assigned to a Tuple  val ordersMap = ordersCompleted.map(rec => (rec.split(",")(0).toInt, rec.split(",")(1)))  ## Filter on Tuple. First element of Tuple = 32. Using “Underscore notation” on the record  ordersMap.filter(rec => (rec.\_1 == 32)).foreach(println) |
| 9.3.2 | **Filter**: Few more examples | ***# To take first 5 records***  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3).equals "COMPLETE")).take(5)  ***# To take first 5 records and print***  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3).equals "COMPLETE")).take(5).foreach(println)  ***# To get Top 5 records using take(5)***  val ordersRDD = sc.textFile("")  ordersRDD.filter(line => line.split(",")(3).**equals**("COMPLETE")).**take(5).**foreach(println)  ***# Example for Contains # Similar to LIKE***  ordersRDD.filter(line => line.split(",")(3).**contains**("PENDING")).take(5).foreach(println)  ***# Example for toInt***  ordersRDD.filter(line => line.split(",")(0).**toInt** > 100).take(5).foreach(println)  **# Example for toFloat**  orderItems.map(rec => (rec.split(“,”)(1).toInt, rec.split(“,”)(4).toFloat))  ordersRDD.filter(line => line.split(",")(0).**toInt** > 100 || line.split(",")(3).contains("PENDING")).take(5).foreach(println)  ordersRDD.filter(line => line.split(",")(0).**toInt** > 1000 &&  (line.split(",")(3).contains("PENDING") || line.split(",")(3).equals("CANCELLED"))).  take(5).foreach(println)  ordersRDD.filter(line => line.split(",")(0).**toInt** > 1000 &&  !line.split(",")(3).equals("COMPLETE")).  take(5).foreach(println) |
| 9.3.3 | how to use map function and create paired RDD with order\_id as key  *Map to setup KEY & VALUE*  *RDD.map(rec=> (rec.split(“,”)(0).toInt, rec.split(“,”)(1))* | # **Problem statement:** Get the revenue and number of orders from order\_items on daily basis for COMPLETE orders   * orders table has 4 columns and order\_items has 6 columns * order\_id is the First column(0) in orders and Second column(1) in order\_items * order\_date is the Second column(1) in orders and order\_item\_subtotal (to compute revenue) is the Fifth column(4) in order\_items * order\_status is the Fourth Column(3) in orders * From orders we need order\_date (as we need daily average revenue) * Use map to generate paired RDD of order\_id and order\_date. Ex: The output will be in a tuple like (3, 2013-07-25 00:00:00.0)   + ordersCompleted.map(rec => (rec.split(“,”)(0).toInt, rec.split(“,”)(1)))   + As order\_Id is of type int we are using toInt function to cast it to integer. We can join without cast to Int but it is recommended to cast to int, so that the performance will be better * Use map to generate paired RDD of order\_id and order\_item\_subtotal   + orderItems.map(rec => (rec.split(“,”)(1).toInt, rec.split(“,”)(4).toFloat))   + As order\_Id is of type int we are using toInt function to cast it to integer   + As order\_item\_subtotal is of type float we are using toFloat to cast it to float |
| 9.3.4 | Use **reduceByKey** to get revenue for each order  *<<Coding starts here for the problem stmt>>*  *Key aggregate functions can be applied on top of Map (as map will setup KEY & VALUE)*  *<reduceByKey>* | <https://www.youtube.com/watch?v=eXBMkwxxqg8>   * There are several “ByKey” aggregate functions   + groupByKey *## Suitable for Sorting, Ranking, etc*   + reduceByKey *## Suitable for simple aggregation*   + aggregateByKey *## Suitable for complex aggregation* * groupByKey is more generic function and it does not use combiner. It is more appropriate to use for complex group transformations such as sorting and ranking * aggregateByKey is used in the scenarios where we have different logic for combiner and reducer * reduceByKey is more appropriate function to aggregate the data for current scenario * Using new paired RDD of order\_items, we can compute total revenue for each order   + Remember each order will have multiple records in order\_items   + reduceByKey((acc, value) => acc + value) *## function to generate revenue for each order\_id* * Now we are ready to join orders and order\_items after extracting required fields and performing aggregations   val filePath="/user/saranvisa/sqoop\_import"  val ordersRDD = sc.textFile(filePath + "/orders")  val orderItemsRDD = sc.textFile(filePath + "/order\_items")  ## To get ONLY "COMPLETE" orders  val ordersCompleted = ordersRDD.filter(rec => (rec.split(",")(3) == "COMPLETE"))  ## Orders table: To Map Order\_id and Order\_Date to a Tuple (KEY, VALUE)  val ordersMap = ordersCompleted.map(rec => (rec.split(",")(0).toInt, rec.split(",")(1)))  ## Order\_item table: To Map order\_id and order\_item\_subtotal to a Tuple (KEY, VALUE)  val orderItemsMap = orderItemsRDD.map(rec => (rec.split(",")(1).toInt, rec.split(",")(4).toFloat))  ## We can apply reduceByKey BEFORE JOIN or AFTER JOIN and check the performance. (Here applying Before JOIN)  val orderItemsReduce = orderItemsMap.reduceByKey((acc,value) => (acc + value)) |
| 9.3.5 | JOIN | <http://www.itversity.com/topic/joining-data-sets-scala/>   * All types of joins work on paired RDDs (KEY, VALUE) . Joins require 2 paired RDDs which means each RDD should have Tuple * The Tuple should have a common KEY * Supporting Join Types  1. leftOuterJoin 2. rightOuterJoin 3. fullOuterJoin  * Output will be paired RDD with KEY & VALUE. VALUE is **nested tuple** with values from both the data sets. * Ex: [(Int, (String, Float))] , * KEY is Order\_ID (Int data type) * VALUE are Order\_Date (STRING) and order\_item\_subtotal (FLOAT)   *## Join*  val ordersJoin = ordersMap.join(orderItemsReduce)  ordersJoin.take(5).foreach(println)  (23776,(2013-12-20 00:00:00.0,329.98))  (5384,(2013-08-26 00:00:00.0,241.96))  (57800,(2013-07-26 00:00:00.0,889.94))  (16132,(2013-11-04 00:00:00.0,119.98))  (61948,(2014-01-03 00:00:00.0,599.97003)) |
| 9.3.6 | Tuple:  Convert the final output to JSON and setup | as separator instead of comma | # By default Tuple has comma separater between elements, This will modify the comma to |  orderJoin.take(10).foreach(rec => (println(rec.\_1) + "|" + rec.\_2.\_1 + "|" + rec.\_2.\_2))  ## To Convert the Tuple outout to JSON  ordersJoin.take(5).foreach(rec => (println("{\" order\_id \ ":" + rec.\_1 + "}") |
| 10 | **Use Case: Compute the average revenue from previous use case result using aggregateByKey**  *<aggregateByKey>* | * <http://www.itversity.com/topic/compute-average-revenue-scala/> * Implement aggregateByKey to get both revenue as well as number of orders * Compute average revenue for each day * Sort the data by date * Save the output to a file/directory |
| 10.1 | Compute Revenue and count of orders for each day  *<aggregateByKey>* | * <https://www.youtube.com/watch?v=xh3uEop3nvI> * Apply map function to discard order\_id and create paired RDD of order\_date and per order revenue * per order revenue is computed before joining the data sets by using reduceByKey * Implement aggregateByKey to get both revenue as well as number of orders * **Input for aggregateByKey is revenue for each order for each day** * **Output for aggregateByKey is one record**   + **with order\_date as key per date**   + **total revenue and total number of orders per date**   **Output of ordersJoin**  ordersJoin.take(5).foreach(println)  (23776,(2013-12-20 00:00:00.0,329.98))  (5384,(2013-08-26 00:00:00.0,241.96))  (57800,(2013-07-26 00:00:00.0,889.94))  (16132,(2013-11-04 00:00:00.0,119.98))  (61948,(2014-01-03 00:00:00.0,599.97003))  val ordersJoinMap = ordersJoin.map(rec => (rec.\_2.\_1, rec.\_2.\_2))  val revenuePerDay = ordersJoinMap.aggregateByKey((0.0, 0))(  (acc, value) => (acc.\_1 + value, acc.\_2 + 1),  (total1, total2) => (total1.\_1 + total2.\_1, total1.\_2 + total2.\_2)  )   1. **ZeroValue (To initialize) :** (0.0, 0)   **Note:** It is based on our output type. In this case, the we are expecting the output of total revenue (which is float) and count of orders per day (which is Integer), so initialized it as (0.0,0)   1. **SeqOp (For Combiner logic):**(acc, value) => (acc.\_1 + value, acc.\_2 + 1) 2. **combOp (for reduce logic):** (total1, total2) => (total1.\_1 + total2.\_1, total1.\_2 + total2.\_2) |
| 10.2 | **Average revenue for each day**  *<aggregateByKey>* | * <https://www.youtube.com/watch?v=QcNRtQm8lOI> * We have order\_date as key and revenue, total number of orders as value * Now we need to apply map function * Divide revenue with total number of orders to get average revenue * If the average revenue has so many decimal, we need to round the result using ‘Scala’ API. * **Note:** there is no Spark API to round the numbers as of spark version 1.2   **Example for Round in Scala**  scala> BigDecimal(123.45634567).setScale(2, BigDecimal.RoundingMode.HALF\_UP) *## Note: setScale not Scala*  res5: scala.math.BigDecimal = 123.46  scala> BigDecimal(123.45434567).setScale(2, BigDecimal.RoundingMode.HALF\_UP)  res6: scala.math.BigDecimal = 123.45  val averageRevenuePerDay = revenuePerDay.  map(rec => (rec.\_1, BigDecimal(rec.\_2.\_1 / rec.\_2.\_2).  setScale(2, BigDecimal.RoundingMode.HALF\_UP).toFloat)) |
| 11 | **Use case:** Sort the data by date using sortByKey  *<sortByKey>* | * <https://www.youtube.com/watch?v=tcp_ltcZZEo>   Now let us sort the data by date   * We need to make sure the key on which data needs to be sorted is part of the key * order\_date in this case * Apply sortByKey to sort the data based on key * sortByKey also takes true or false as parameter for ascending or descending   **For ascending order:**  val averageRevenuePerDaySorted = averageRevenuePerDay.  sortByKey()  **For descending order:**  val averageRevenuePerDaySorted = averageRevenuePerDay.  sortByKey(false) |
| 12 | **Use case:** Save the output to the file system | * Now let us see how we can store output to file system * RDD can be written to any of the file systems   + local   + HDFS   + s3 * It can be written in any file format   + saveAsTextFile   + saveAsSequenceFile   + saveAsHadoopFile   + saveAsNewAPIHadoopFile * In this case we will try to save as text file   averageRevenuePerDaySorted.map(rec => rec.\_1 + "," + rec.\_2).  saveAsTextFile(outputPath) |
| 13 | **Use case:** How to convert the Individual CLI spark script to Eclipse program | # This is not appropriate video, still use it just for reference  # After 4th min: <http://www.itversity.com/topic/compute-average-revenue-scala/> |
| 14 | **Use case:**  reduceByKey vs reduce | TBD  val orderItemsReduce = orderItemsMap.reduce((acc, val) => acc + val)   i am using reduce() function in order to perform aggregation (sum). It is generally used to get the sum, min, max values in any table. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | **Known issue** |  |
| 1 | object apache is not a member of package org sbt | Issue:   1. Created a simple.sbt file with below content and executed > sbt package   name := "Simple Project"  version := "1.0"  scalaVersion := "2.10.4"  libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.1"   1. Error: object apache is not a member of package org sbt   Soln:  Check the scala version and make sure to use suitable version is mentioned in scalaVersion. In this scenario, the actual scala version is 2.11.8 (run> scala –version) but I’m using 2.10.4, it sbt package command has failed |
| 2 | ForeachRDD is slow | <https://community.cloudera.com/t5/Advanced-Analytics-Apache-Spark/Spark-map-vs-foreachRdd/m-p/51302>  requestsWithState.foreachRDD { rdd =>  println("Data being populated to Pulsar")  rdd.foreach { case (id, eventStream) =>  println("id is " + id + " Event is " + eventStream)  DBUtils.putItem(dynamoConnection, id, eventStream.toString())  }  }  The alternate would be the below code, refer the link above  requestsWithState.map (rdd => { rdd match {  case (id, eventStream) => {  println("id is " + id + " Event is " + eventStream)  val dynamoConnection = setupDynamoClientConnection()  DBUtils.putItem(dynamoConnection, id, eventStream.toString())  }  }  }) |
| 3 | Spark commands are not working | Make sure it doesn’t goes running out of memory. If so, Try restart the process, it will work |