databricks G3G4 transformation extended Notebook 2023-08-28 11:05:13

(https://databricks.com)

```
print("helloworld")
```

```
helloworld
```

```
val accum = sc.longAccumulator("Sum Accumulator")
accum.value
```

```
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(Sum Accumulator), value: 0)
res1: Long = 0
```

```
val broadcastVar = sc.broadcast(Array(1,2,3))
```

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

```
broadcastVar.value
```

```
res2: Array[Int] = Array(1, 2, 3)
```

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("SparkExample").master("local").getOrCreate()
```

```
import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@eb4526a
```

```
val states = Map(("NY", "New York"),("CA", "California"),("FL", "Florida"))
```

```
states: scala.collection.immutable.Map[String,String] = Map(NY -> New York, CA -> California, FL -> Florida)
```

```
val countries = Map(("USA", "United states of america"), ("IN", "India"))
```

```
countries: scala.collection.immutable.Map[String,String] = Map(USA -> United states of america, IN -> India)
```

```
val broadcaststates = spark.sparkContext.broadcast(states)
```

```
broadcaststates: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[String,String]] = Broadcast(1)
```

```
val broadcastcountries = spark.sparkContext.broadcast(countries)
```

```
broadcastcountries: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[String,String]] = Broadcast(2)
```

```
val data = Seq(("james", "Suraj", "USA", "CA"),
               ("mes", "Sura", "USA", "NY"),
               ("megan", "mathews", "USA", "CA"),
               ("Maria", "Mohan", "USA", "FL"))
```

```
data: Seq[(String, String, String, String)] = List((james,Suraj,USA,CA), (mes,Sura,USA,NY), (megan,mathews,USA,CA), (Maria,Mohan,US
A,FL))
```

```
val rdd = spark.sparkContext.parallelize(data)
```

```
rdd: org.apache.spark.rdd.RDD[(String, String, String, String)] = ParallelCollectionRDD[0] at parallelize at command-260941957792899
1:1
```

```
  val rdd2 = rdd.map(f=> {
    val country = f._3
    val state = f._4
    val fullCountry = broadcastcountries.value.get(country).get
    val fullState = broadcaststates.value.get(state).get
    (f._1, f._2,fullCountry, fullState)
  })
```

```
rdd2: org.apache.spark.rdd.RDD[(String, String, String, String)] = MapPartitionsRDD[1] at map at command-2609419577928992:1
```

```
  println(rdd2.collect().mkString("\n"))
```

```
(james,Suraj,United states of america,California)
(mes,Sura,United states of america,New York)
(megan,mathews,United states of america,California)
(Maria,Mohan,United states of america,Florida)
```

```
  val states = Map(("NY", "New York"),("CA", "California"),("FL", "Florida"))
  val countries = Map(("USA", "United states of america"), ("IN", "India"))
  val broadcaststates = spark.sparkContext.broadcast(states)
  val broadcastcountries = spark.sparkContext.broadcast(countries)
  val data = Seq(("james", "Suraj", "USA", "CA"),
                 ("mes", "Sura", "USA", "NY"),
                 ("megan", "mathews", "USA", "CA"),
                 ("Maria", "Mohan", "USA", "FL"))
  val columns = Seq("First name", "Lastname", "Country", "States")
  import spark.sqlContext.implicits._
  val df = data.toDF(columns:_*)
  val df2 = df.map(row =>{
    val country = row.getString(2)
    val state = row.getString(3)
      val fullcountry = broadcastcountries.value.get(country).get
      val fullstate = broadcaststates.value.get(state).get
      (row.getString(0), row.getString(1), fullcountry,fullstate)
  }).toDF(columns:_*)
```

```
states: scala.collection.immutable.Map[String,String] = Map(NY -> New York, CA -> California, FL -> Florida)
countries: scala.collection.immutable.Map[String,String] = Map(USA -> United states of america, IN -> India)
broadcaststates: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[String,String]] = Broadcast(4)
broadcastcountries: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[String,String]] = Broadcast(5)
data: Seq[(String, String, String, String)] = List((james,Suraj,USA,CA), (mes,Sura,USA,NY), (megan,mathews,USA,CA), (Maria,Mohan,US
A,FL))
columns: Seq[String] = List(First name, Lastname, Country, States)
import spark.sqlContext.implicits._
df: org.apache.spark.sql.DataFrame = [First name: string, Lastname: string ... 2 more fields]
df2: org.apache.spark.sql.DataFrame = [First name: string, Lastname: string ... 2 more fields]
```

```
  df2.show(false)
```

```
+----------+--------+-----------------------+----------+
|First name|Lastname|Country                |States    |
+----------+--------+-----------------------+----------+
|james     |Suraj   |United states of america|California|
|mes       |Sura    |United states of america|New York  |
|megan     |mathews |United states of america|California|
|Maria     |Mohan   |United states of america|Florida   |
+----------+--------+-----------------------+----------+
```

```
  var longAcc = spark.sparkContext.longAccumulator("SumAccumulator")
```

```
longAcc: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 151, name: Some(SumAccumulator), value: 0)
```

```
  val rdd  = spark.sparkContext.parallelize(Array(1,2,3))
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at command-2609419577928997:1
```

```
  rdd.foreach(x => longAcc.add(x))
```

```
  println(longAcc.value)
```

```
6
```

```
  spark.sparkContext.setLogLevel("Error")
```

```
  val inputRDD = spark.sparkContext.parallelize(List(("Z", 1),("A", 20),("B", 30),("C", 40),("B", 30),("B", 60)))
```

```
inputRDD: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[5] at parallelize at command-2609419577929001:1
```

```
  val listRDD = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))
```

```
listRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at parallelize at command-2609419577929002:1
```

```
  // aggregate

  def param0 = (accu:Int, v:Int) => accu + v
  def param1 = (accu1:Int, accu2:Int) => accu1 + accu2
  print("Aggregate :" +listRDD.aggregate(0)(param0, param1))
```

```
Aggregate :20param0: (Int, Int) => Int
param1: (Int, Int) => Int
```

```
  // aggregate

  def param3 = (accu:Int, v:(String, Int)) => accu + v._2
  def param4 = (accu1:Int, accu2:Int) => accu1 + accu2
  print("Aggregate :" +inputRDD.aggregate(0)(param3, param4))
```

```
Aggregate :181param3: (Int, (String, Int)) => Int
param4: (Int, Int) => Int
```

```
  // tree Aggregate

  def param8 = (accu:Int, v: Int) => accu + v
  def param9 = (accu1:Int, accu2:Int) => accu1 + accu2
  print("Tree Aggregate :" +listRDD.treeAggregate(0)(param8, param9))
```

```
Tree Aggregate :20param8: (Int, Int) => Int
param9: (Int, Int) => Int
```

```
  // fold action

  println("fold :" +listRDD.fold(0) {(acc,v) =>
   val sum = acc+ v
    sum
  })
```

```
fold :20
```

```
  println("fold :" +inputRDD.fold(("total", 0)) {(acc:(String, Int), v:(String, Int)) =>
   val sum = acc._2+v._2
    ("Total",sum)
  })
```

```
fold :(Total,181)
```

```
  // what is difference between aggregate, tree aggregate and fold
```

```
  val data = listRDD.collect()
```

```
data: Array[Int] = Array(1, 2, 3, 4, 5, 3, 2)
```

```
  data.foreach(println)
```

```
1
2
3
4
5
3
2
```

```
  // reduce

  println("Reduce :" +listRDD.reduce(_ + _))
```

```
Reduce :20
```

```
  println("Reduce :" +listRDD.reduce(_ - _))
```

```
Reduce :-10
```

```
  println("reduce Alternate :" + inputRDD.reduce((x,y)=> ("Total",x._2+y._2)))
```

```
reduce Alternate :(Total,181)
```

```
  println("tree reduce: " +listRDD.treeReduce(_+_))
```

```
tree reduce: 20
```

```
inputRDD.count
```

```
res24: Long = 6
```

```
inputRDD.countApproxDistinct()
```

```
res26: Long = 5
```

```
listRDD.first
```

```
res27: Int = 1
```

```
listRDD.take(2)
```

```
res28: Array[Int] = Array(1, 2)
```

```
listRDD.top(2)
```

```
res29: Array[Int] = Array(5, 4)
```

```
inputRDD.min._2
```

```
res30: Int = 20
```

```
inputRDD.max._2
```

```
res31: Int = 1
```

```
inputRDD.min
```

```
res32: (String, Int) = (A,20)
```

```
inputRDD.max
```

```
res33: (String, Int) = (Z,1)
```

```
<console>:8: error: identifier expected but integer literal found.
       partitionied_rdd = rdd.partitionBy(2, lambda x: x % 2)
                                                       ^
```