Hello,

I consider the 3rd excercise in a tutorial slightly complex. I mean - the code is commented on a very general level. The explanation is sufficient to understand more or less what is going on but not to reuse the code for some other problem.

I decided to deepen my uderstanding. I document the results of my research here. Maybe somebody will find my notes useful.

Important note: I am not an expert. I might be wrong in some of my statements or conclusions.

Whatever code is given here, it has to be typed in spark-shell.

Enabling serialization:

Understanding the steps is much easier if you can see rows in RDD. And you can do it, but there are some additional hacks needed.

RDD has methods like “first”, “take(n)”, which return rows from a given RDD. But if you try to run those methods on, say, order_items, using

```
order_items.first()
```

you will get an error. Spark will complain about a missing serializer. You have to add a serializer to a SparkContext. As far as I understand, SparkContext is a connection with Spark. You cannot change configuration of an existing context. You have to create a new configuration and recreate a context.

So you go with the code like this:

```
import org.apache.spark.SparkContext
val conf = sc.getConf

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[org.apache.avro.generic.GenericData$Record])
)
sc.stop

val sc = new SparkContext(conf)
```

After you launch spark-shell, context is in sc variable. The code above closes sc, then creates a new context, again in sc variable, with KryoSerializer.

After doing this, you will be able to retrieve rows from RDD.

Analysis
So, let's have a look at the code below, taken form the tutorial. Let’s call this code “snippet 1”.

Snippet 1:

```
1:  def rddFromParquetHdfsFile(path: String): RDD[GenericRecord] = {
2:     val job = new Job()
3:     FileInputFormat.setInputPaths(job, path)
4:     ParquetInputFormat.setReadSupportClass(job,
5:        classOf[AvroReadSupport[GenericRecord]])
6:     return sc.newAPIHadoopRDD(job.getConfiguration,
```

```
7:         classOf[ParquetInputFormat[GenericRecord]],
8:         classOf[Void],
9:         classOf[GenericRecord]).map(x => x._2)
10: }
11:
12: val warehouse = "hdfs://quickstart/user/hive/warehouse/"
13: val order_items = rddFromParquetHdfsFile(warehouse + "order_items");
14: val products = rddFromParquetHdfsFile(warehouse + "products");
```

Function rddFromParquetHdfsFile retrieves data from HDFS file and creates RDD.

The actuall creation is in lines 6-9. One migth ask, what is map for in line 9.

```
6:     return sc2.newAPIHadoopRDD(job.getConfiguration,
7:         classOf[ParquetInputFormat[GenericRecord]],
8:         classOf[Void],
9:         classOf[GenericRecord]).map(x => x._2)
```

Let's see what will happen, if the map is not there. Define this function:

```
def rddFromParquetHdfsFile_nm(path: String): RDD[(Void, GenericRecord)] = {
    val job = new Job()
    FileInputFormat.setInputPaths(job, path)
    ParquetInputFormat.setReadSupportClass(job,
        classOf[AvroReadSupport[GenericRecord]])
    return sc.newAPIHadoopRDD(job.getConfiguration,
           classOf[ParquetInputFormat[GenericRecord]],
           classOf[Void],
           classOf[GenericRecord])
}
```

The only difference is that map part was removed from line 9. Let’s create an RDD, using this non-mapped method:

```
val order_items_nm = rddFromParquetHdfsFile_nm(warehouse + "order_items");
```

Let’s have a look at some rows from this RDD

```
scala> order_items_nm
res3: org.apache.spark.rdd.RDD[(Void, org.apache.avro.generic.GenericRecord)] =
NewHadoopRDD[0] at newAPIHadoopRDD at <console>:43

scala> order_items_nm.take(4).foreach(println)
(null,{"order_item_id": 1, "order_item_order_id": 1, "order_item_product_id": 957,
"order_item_quantity": 1, "order_item_subtotal": 299.98,
"order_item_product_price": 299.98})
(null,{"order_item_id": 2, "order_item_order_id": 2, "order_item_product_id": 1073,
"order_item_quantity": 1, "order_item_subtotal": 199.99,
"order_item_product_price": 199.99})
(null,{"order_item_id": 3, "order_item_order_id": 2, "order_item_product_id": 502,
"order_item_quantity": 5, "order_item_subtotal": 250.0, "order_item_product_price":
50.0})
(null,{"order_item_id": 4, "order_item_order_id": 2, "order_item_product_id": 403,
"order_item_quantity": 1, "order_item_subtotal": 129.99,
"order_item_product_price": 129.99})
```

As you can see, each element of an array contains a tuple (key,row). Example:

```
(null,{"order_item_id": 1, "order_item_order_id": 1, "order_item_product_id": 957,
"order_item_quantity": 1, "order_item_subtotal": 299.98,
"order_item_product_price": 299.98}),
```

Notice, that key is always null. So this map

```
map(x => x._2)
```

for each row replaces the row with a second element of a (key,row) tuple ("_2" is an index of the second element in a tuple). Rows from RDD created with this map look like this:

```
scala> order_items
res5: org.apache.spark.rdd.RDD[org.apache.avro.generic.GenericRecord] =
MapPartitionsRDD[2] at map at <console>:46

scala> order_items.take(4).foreach(println)
{"order_item_id": 1, "order_item_order_id": 1, "order_item_product_id": 957,
"order_item_quantity": 1, "order_item_subtotal": 299.98,
"order_item_product_price": 299.98}
{"order_item_id": 2, "order_item_order_id": 2, "order_item_product_id": 1073,
"order_item_quantity": 1, "order_item_subtotal": 199.99,
"order_item_product_price": 199.99}
{"order_item_id": 3, "order_item_order_id": 2, "order_item_product_id": 502,
"order_item_quantity": 5, "order_item_subtotal": 250.0, "order_item_product_price":
50.0}
{"order_item_id": 4, "order_item_order_id": 2, "order_item_product_id": 403,
"order_item_quantity": 1, "order_item_subtotal": 129.99,
"order_item_product_price": 129.99}
```

So, after line 14 from snippet 1 is executed we have order_items and products loaded into RDDs and you can view some rows from them, using "first" or "take". Each element of an according RDD contains one order_item and product row respectively

Now, let's analyze the next chunk of the code.

Snippet 2:

```
 1  val orders = order_items.map { x => (
 2      x.get("order_item_product_id"),
 3      (x.get("order_item_order_id"), x.get("order_item_quantity")))
 4  }.join(
 5    products.map { x => (
 6      x.get("product_id"),
 7      (x.get("product_name")))
 8    }
 9  ).map(x => (
10      scala.Int.unbox(x._2._1._1), // order_id
11      (
12          scala.Int.unbox(x._2._1._2), // quantity
13          x._2._2.toString // product_name
14      )
15  )).groupByKey()
```

Let's analyze step by step what is being done here. For convenience let's create temporary variables, which will keep the results of analyzed steps.

This code:

```
1  val orders = order_items.map { x => (
2      x.get("order_item_product_id"),
3      (x.get("order_item_order_id"), x.get("order_item_quantity")))
4  }
```

creates an RDD, which contains following tuples - (order_item_product_id,(order_item_order_id, order_item_quantity))

Let's see how do rows look like:

```
val temp_oi = order_items.map { x => (
    x.get("order_item_product_id"),
    (x.get("order_item_order_id"), x.get("order_item_quantity")))
}


scala> temp_oi
res9: org.apache.spark.rdd.RDD[(Object, (Object, Object))] = MapPartitionsRDD[5] at
map at <console>:44

scala> temp_oi.take(4).foreach(println)
(957,(1,1))
(1073,(2,1))
(502,(2,5))
(403,(2,1))
```

So, from the business point of view, it contains all order items for all products. Row one, (957,(1,1)) says: For product id 957 there is an order 1, in which product 957 is ordered in quantity 1.

This code:

```
5    products.map { x => (
6      x.get("product_id"),
7      (x.get("product_name")))
8    }
```

Returns all product ids with their names. See below:

```
scala> val temp_prod = products.map { x => (
     |     x.get("product_id"),
     |     (x.get("product_name")))
     |   }
temp_prod: org.apache.spark.rdd.RDD[(Object, Object)] = MapPartitionsRDD[4] at map
at <console>:44

scala> temp_prod
res3: org.apache.spark.rdd.RDD[(Object, Object)] = MapPartitionsRDD[4] at map at
<console>:44

scala> temp_prod.take(4).foreach(println)
(1,Quest Q64 10 FT. x 10 FT. Slant Leg Instant U)
(2,Under Armour Men's Highlight MC Football Clea)
(3,Under Armour Men's Renegade D Mid Football Cl)
(4,Under Armour Men's Renegade D Mid Football Cl)
```

Let's join those temporary rdds:

```
scala> val orders = temp_oi.join(temp_prod)
orders: org.apache.spark.rdd.RDD[(Object, ((Object, Object), Object))] =
MapPartitionsRDD[10] at join at <console>:50
```

The join method, will join those 2 rdds using first element of each tuple (row) in both of them. After a join we will will all order items, with each of them having a product name assigned. See below.

```
scala> val orders_tmp1 = temp_oi.join(temp_prod)
orders_tmp1: org.apache.spark.rdd.RDD[(Object, ((Object, Object), Object))] =
MapPartitionsRDD[11] at join at <console>:50
```

```
scala>

scala> orders_tmp1
res8: org.apache.spark.rdd.RDD[(Object, ((Object, Object), Object))] =
MapPartitionsRDD[11] at join at <console>:50

scala> orders_tmp1.take(4).foreach(println)
(226,((68691,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68699,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68703,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68710,1),Bowflex SelectTech 1090 Dumbbells))
```

Line (226,((68691,1),Bowflex SelectTech 1090 Dumbbells)) means: Product with ID 226, name "Bowflex SelectTech 1090 Dumbbells" was ordered in order 68691 in amount 1.

So this is what we have analyzed so far:

```
 1  val orders = order_items.map { x => (
 2      x.get("order_item_product_id"),
 3      (x.get("order_item_order_id"), x.get("order_item_quantity")))
 4  }.join(
 5    products.map { x => (
 6      x.get("product_id"),
 7      (x.get("product_name")))
 8    }
 9  )
```

The result above is the a subject of the following map:

```
 9  map(x => (
10      scala.Int.unbox(x._2._1._1), // order_id
11      (
12          scala.Int.unbox(x._2._1._2), // quantity
13          x._2._2.toString // product_name
14      )
```

What are those unbox(x._2._1._1) and like?

First – what is unbox? Basically, unbox converts a general Object type to Int. Look below:

Before map

```
scala> orders_tmp1
res8: org.apache.spark.rdd.RDD[(Object, ((Object, Object), Object))] =
MapPartitionsRDD[11] at join at <console>:50

scala> orders_tmp1.take(4).foreach(println)
(226,((68691,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68699,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68703,1),Bowflex SelectTech 1090 Dumbbells))
(226,((68710,1),Bowflex SelectTech 1090 Dumbbells))
```

After map

```
scala> orders_tmp2
res10: org.apache.spark.rdd.RDD[(Int, (Int, String))] = MapPartitionsRDD[12] at map
at <console>:52

scala> orders_tmp2.take(4).foreach(println)
(68691,(1,Bowflex SelectTech 1090 Dumbbells))
(68699,(1,Bowflex SelectTech 1090 Dumbbells))
```

```
(68703,(1,Bowflex SelectTech 1090 Dumbbells))
(68710,(1,Bowflex SelectTech 1090 Dumbbells))
```

All Objects are replaced by Int and String.

And this: x._2._1._1?

_1 or _2 are indices in a tuple. So to x._2._1._1 we have to:

1. Take 2nd tupled from the row (indexing is 1 based)
2. From tuple in step 1 take first element, which is a tuple
3. From tuple in step 2 take first element.

So when you compare RDDs before and after this last mapping, you can see 2 changes:

1. Object types were replaced by Int and String
2. Product id was removed (only product name remained)

The final step is line 15, grouping by a key. And a key is an order number. So, after executing a groupByKey, we get RDD, with each row containing one order i.e.:

1. Order id
2. All items (product name and amount)

See below:

```
scala> val orders_tmp3 = orders_tmp2.groupByKey()
orders_tmp3: org.apache.spark.rdd.RDD[(Int, Iterable[(Int, String)])] =
ShuffledRDD[10] at groupByKey at <console>:54

scala> orders_tmp3
res3: org.apache.spark.rdd.RDD[(Int, Iterable[(Int, String)])] = ShuffledRDD[10] at
groupByKey at <console>:54

scala> orders_tmp3.take(4).foreach(println)
(65722,CompactBuffer((1,Field & Stream Sportsman 16 Gun Fire Safe), (5,Under Armour
Girls' Toddler Spine Surge Runni), (2,Perfect Fitness Perfect Rip Deck), (5,LIJA
Women's Argyle Golf Polo), (2,Nike Men's Free 5.0+ Running Shoe)))
(68522,CompactBuffer((1,Stiga Master Series ST3100 Competition Indoor)))
(23776,CompactBuffer((1,Nike Men's CJ Elite 2 TD Football Cleat), (1,Pelican
Sunstream 100 Kayak)))
(34207,CompactBuffer((2,O'Brien Men's Neoprene Life Vest), (2,Perfect Fitness
Perfect Rip Deck)))
```

At this stage we have an RDD. Each row contains an order ID and all items belonging to this order.

Let look at this code now

```
1  val cooccurrences = orders.map(order =>
2    (
3      order._1,
4      order._2.toList.combinations(2).map(order_pair =>
5          (
6              if (order_pair(0)._2 < order_pair(1)._2)
7                  (order_pair(0)._2, order_pair(1)._2)
8              else
```

```
 9                  (order_pair(1)._2, order_pair(0)._2),
10              order_pair(0)._1 * order_pair(1)._1
11          )
12      )
13    )
14  )
```

Line 4 is the most complex one. Let's observe what happens, step by step:

```
order._2.toList
```

It is takes a second element of an order row, which is a list of all items in the order.

```
order._2.toList.combinations(2)
```

Creates all possible pairs of order items in one order. Example result:

```
scala> orders.first()._2.toList.combinations(2).foreach(println)
List((1,Field & Stream Sportsman 16 Gun Fire Safe), (5,Under Armour Girls' Toddler
Spine Surge Runni))
List((1,Field & Stream Sportsman 16 Gun Fire Safe), (2,Perfect Fitness Perfect Rip
Deck))
List((1,Field & Stream Sportsman 16 Gun Fire Safe), (5,LIJA Women's Argyle Golf
Polo))
List((1,Field & Stream Sportsman 16 Gun Fire Safe), (2,Nike Men's Free 5.0+ Running
Shoe))
List((5,Under Armour Girls' Toddler Spine Surge Runni), (2,Perfect Fitness Perfect
Rip Deck))
List((5,Under Armour Girls' Toddler Spine Surge Runni), (5,LIJA Women's Argyle Golf
Polo))
List((5,Under Armour Girls' Toddler Spine Surge Runni), (2,Nike Men's Free 5.0+
Running Shoe))
List((2,Perfect Fitness Perfect Rip Deck), (5,LIJA Women's Argyle Golf Polo))
List((2,Perfect Fitness Perfect Rip Deck), (2,Nike Men's Free 5.0+ Running Shoe))
List((5,LIJA Women's Argyle Golf Polo), (2,Nike Men's Free 5.0+ Running Shoe))
```

And finally:

```
scala> val co1 = orders.first()._2.toList.combinations(2).map(order_pair =>
     |         (
     |             if (order_pair(0)._2 < order_pair(1)._2)
     |                 (order_pair(0)._2, order_pair(1)._2)
     |             else
     |                 (order_pair(1)._2, order_pair(0)._2),
     |             order_pair(0)._1 * order_pair(1)._1
     |         )
     |     )
co1: Iterator[((String, String), Int)] = non-empty iterator

scala>

scala> co1.foreach(println)
((Field & Stream Sportsman 16 Gun Fire Safe,Under Armour Girls' Toddler Spine Surge
Runni),5)
((Field & Stream Sportsman 16 Gun Fire Safe,Perfect Fitness Perfect Rip Deck),2)
((Field & Stream Sportsman 16 Gun Fire Safe,LIJA Women's Argyle Golf Polo),5)
((Field & Stream Sportsman 16 Gun Fire Safe,Nike Men's Free 5.0+ Running Shoe),2)
((Perfect Fitness Perfect Rip Deck,Under Armour Girls' Toddler Spine Surge
Runni),10)
((LIJA Women's Argyle Golf Polo,Under Armour Girls' Toddler Spine Surge Runni),25)
((Nike Men's Free 5.0+ Running Shoe,Under Armour Girls' Toddler Spine Surge
Runni),10)
((LIJA Women's Argyle Golf Polo,Perfect Fitness Perfect Rip Deck),10)
```

```
((Nike Men's Free 5.0+ Running Shoe,Perfect Fitness Perfect Rip Deck),4)
((LIJA Women's Argyle Golf Polo,Nike Men's Free 5.0+ Running Shoe),10)
```

So the final map does 2 things –

1. Orders items alphabetically
2. Calculates some kind of weight which is a mutliplication of amounts in a pair.

We are almost there. Let look at this line:

```
val combos = cooccurrences.flatMap(x => x._2).reduceByKey((a, b) => a + b)
```

Again, step by step:

```
scala> val combos_tmp1 = cooccurrences.flatMap(x => x._2)
combos_tmp1: org.apache.spark.rdd.RDD[((String, String), Int)] =
MapPartitionsRDD[14] at flatMap at <console>:60
```

Compare with “coocurrences”:

```
scala> cooccurrences
res38: org.apache.spark.rdd.RDD[(Int, Iterator[((String, String), Int)])] =
MapPartitionsRDD[11] at map at <console>:58
```

After flatMap there is no iterator any more, the result is flattened.

So, 4 rows of coocurrances:

```
scala> cooccurrences.take(4).foreach(println)
(65722,non-empty iterator)
(68522,empty iterator)
(23776,non-empty iterator)
(34207,non-empty iterator)
```

After flatMap-ing:

```
scala> combos_tmp1.take(4).foreach(println)
((Field & Stream Sportsman 16 Gun Fire Safe,Under Armour Girls' Toddler Spine Surge
Runni),5)
((Field & Stream Sportsman 16 Gun Fire Safe,Perfect Fitness Perfect Rip Deck),2)
((Field & Stream Sportsman 16 Gun Fire Safe,LIJA Women's Argyle Golf Polo),5)
((Field & Stream Sportsman 16 Gun Fire Safe,Nike Men's Free 5.0+ Running Shoe),2)
```

Note that order_id is also gone. We don’t need for further calculations.

Finally:

```
val combos = cooccurrences.flatMap(x => x._2).reduceByKey((a, b) => a + b)
```

```
As you can see above, product names  pair is a key. So reduceByKey find all rows
with the same product names pair and sums up the weights (multiplied amounts). So
finaly we get this:
```

```
scala> combos.take(4).foreach(println)
((Perfect Fitness Perfect Rip Deck,Top Flite Women's 2014 XL Hybrid),855)
((Bag Boy Beverage Holder,Glove It Women's Mod Oval 3-Zip Carry All Gol),10)
((TYR Boys' Team Digi Jammer,TYR Boys' Team Digi Jammer),24)
((Hirzl Women's Hybrid Golf Glove,Titleist Pro V1x High Numbers Personalized
Go),28)
```

So at this step we have an RDD (combos) which contains all pairs of products, which appeared on the same order. Each such pair contains sum of weights from all orders in which this paid has appeared.

So the very last step it get the most correlated ones (the ones with the highest weights). Here is how we do it:

```
val mostCommon = combos.map(x => (x._2, x._1)).sortByKey(false).take(10)
```

First we change the order of elements in the row, so the weight is a key now. Then we sort it descending by a key (which is weight) and take 10 top values.

Voila:

```
scala> mostCommon.foreach(println)
(67876,(Nike Men's Dri-FIT Victory Golf Polo,Perfect Fitness Perfect Rip Deck))
(62924,(O'Brien Men's Neoprene Life Vest,Perfect Fitness Perfect Rip Deck))
(54399,(Nike Men's Dri-FIT Victory Golf Polo,O'Brien Men's Neoprene Life Vest))
(39656,(Nike Men's Free 5.0+ Running Shoe,Perfect Fitness Perfect Rip Deck))
(39314,(Perfect Fitness Perfect Rip Deck,Perfect Fitness Perfect Rip Deck))
(35092,(Perfect Fitness Perfect Rip Deck,Under Armour Girls' Toddler Spine Surge
Runni))
(33750,(Nike Men's Dri-FIT Victory Golf Polo,Nike Men's Free 5.0+ Running Shoe))
(33406,(Nike Men's Free 5.0+ Running Shoe,O'Brien Men's Neoprene Life Vest))
(29835,(Nike Men's Dri-FIT Victory Golf Polo,Nike Men's Dri-FIT Victory Golf Polo))
(29342,(Nike Men's Dri-FIT Victory Golf Polo,Under Armour Girls' Toddler Spine
Surge Runni))
```

I hope it helps. If you find any mistakes – please comment in this thread.