**The Problem**

Let me quickly restate the problem from [my original article](http://blog.matthewrathbone.com/2013/01/05/a-quick-guide-to-hadoop-map-reduce-frameworks.html#walkthrough).

I have two datasets:

1. User information (id, email, language, location)
2. Transaction information (transaction-id, product-id, user-id, purchase-amount, item-description)

Given these datasets, I want to find the number of unique locations in which each product has been sold. To do that, I need to join the two datasets together.

Previously I have implemented this solution [in java](http://blog.matthewrathbone.com/2013/02/09/real-world-hadoop-implementing-a-left-outer-join-in-hadoop-map-reduce.html), [with hive](http://blog.matthewrathbone.com/2013/02/20/real-world-hadoop---implementing-a-left-outer-join-in-hive.html) and [with pig](http://blog.matthewrathbone.com/2013/04/07/real-world-hadoop---implementing-a-left-outer-join-in-pig.html). The java solution was ~500 lines of code, hive and pig were like ~20 lines tops.

**The Spark Scala Solution**

[Spark](http://spark.apache.org) is an open source project that has been built and is maintained by a thriving and diverse community of developers. Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab. It was observed that MapReduce was inefficient for some iterative and interactive computing jobs, and Spark was designed in response. Spark’s aim is to be fast for interactive queries and iterative algorithms, bringing support for in-memory storage and efficient fault recovery. Iterative algorithms have always been hard for MapReduce, requiring multiple passes over the same data.

**Demonstration Data**

The tables that will be used for demonstration are called users and transactions.

users

1 matthew@test.com EN US

2 matthew@test2.com EN GB

3 matthew@test3.com FR FR

and

transactions

1 1 1 300 a jumper

2 1 2 300 a jumper

3 1 2 300 a jumper

4 2 3 100 a rubber chicken

5 1 3 300 a jumper

For this task we have used Spark on Hadoop YARN cluster. Our code will read and write data from/to HDFS. Before starting work with the code we have to copy the input data to HDFS.

hdfs dfs -mkdir input

hdfs dfs -put ./users.txt input

hdfs dfs -put ./transactions.txt input

**Code**

All code and data used in this post can be found in my [Hadoop examples GitHub repository](https://github.com/rathboma/hadoop-framework-examples/).

class ExampleJob(sc: SparkContext) {

def run(t: String, u: String) : RDD[(String, String)] = {

val transactions = sc.textFile(t)

val newTransactionsPair = transactions.map{t =>

val p = t.split("\t")

(p(2).toInt, p(1).toInt)

}

val users = sc.textFile(u)

val newUsersPair = users.map{t =>

val p = t.split("\t")

(p(0).toInt, p(3))

}

val result = processData(newTransactionsPair, newUsersPair)

return sc.parallelize(result.toSeq).map(t => (t.\_1.toString, t.\_2.toString))

}

def processData (t: RDD[(Int, Int)], u: RDD[(Int, String)]) : Map[Int,Long] = {

var jn = t.leftOuterJoin(u).values.distinct

return jn.countByKey

}

}

object ExampleJob {

def main(args: Array[String]) {

val transactionsIn = args(1)

val usersIn = args(0)

val conf = new SparkConf().setAppName("SparkJoins").setMaster("local")

val conext = new SparkContext(conf)

val job = new ExampleJob(context)

val results = job.run(transactionsIn, usersIn)

val output = args(2)

results.saveAsTextFile(output)

context.stop()

}

}

Prior to manipulating the data it is required to define a SparkContext. It is enough to set an app name and a location of a master node.

val conf = new SparkConf().setAppName("SparkJoins").setMaster("local")

val sc = new SparkContext(conf)

Spark’s core abstraction for working with data is the resilient distributed dataset (RDD). Explicitely you can see it in the code when looking at processData function:

def processData (t: RDD[(Int, Int)], u: RDD[(Int, String)]) : Map[Int,Long] = {

var jn = t.leftOuterJoin(u).values.distinct

return jn.countByKey

}

Both newTransactionsPair and newUsersPair are RDDs. They are Key/Value RDDs to be more precise.

An RDD in Spark is an immutable distributed collection of objects. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. Computations on RDD’s are designed to feel like Scala’s native [List operations](http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List).

In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. Spark automatically distributes the data contained in RDDs across the cluster and parallelizes the operations that are performed on them.

[Transforming existing RDDs](http://spark.apache.org/docs/latest/programming-guide.html#transformations) is different from [calling an action](http://spark.apache.org/docs/latest/programming-guide.html#actions) to compute a result. Actions trigger actual computations, where transformations are lazy, so transformation code is not executed until a downstream action is called.

In our code we utilize a lot of Key/Value RDDs. Key/Value RDDs are commonly used to perform aggregations, such as countByKey(), and are useful for joins, such as leftOuterJoin().

In our case we use the action countByKey() (and saveAsTextFile() that is used to output result to HDFS). Where a transformation only returns info about the format the data after the transformation (because it doesn’t actually do anything), calling an action will immediately result in logs about what is being done and the progress of the computation pipeline.

It’s really easy to see the transaction/action interplay by using the Spark CLI, an interactive Spark shell.

**Transforming our data**

The process of transforming the input text file into a Key/value RDD is rather self-explanatory:

val transactions = sc.textFile(t)

val newTransactionsPair = transactions.map{t =>

val p = t.split("\t")

(p(2).toInt, p(1).toInt)

}

After calling an action and computing a result, we transform it back into an RDD so we can use the saveAsTextFile function to store the result elsewhere in HDFS.

val r = sc.parallelize(result.toSeq).map(t => (t.\_1.toString, t.\_2.toString))

Here toSeq transforms the Map that countByKey of the processData function returns into an ArrayBuffer. This ArrayBuffer can be given as an input to parallelize function of SparkContext to map it back into an RDD.

Spark is designed with workflows like ours in mind, so join and key count operations are provided out of the box.

var jn = t.leftOuterJoin(u).values.distinct

return jn.countByKey

The leftOuterJoin() function joins two RDDs on key, that is why it was important that our RDDs are Key/Value RDDs. The result of the join is an RDD of a form RDD[(Int, (Int, Option[String]))].

The values() functions allows to omit the key of the join (user\_id) as it is not needed in the operations that follow the join.

The distinct() function selects distinct Tuples from the values of the join.

The result of values() and distinct() functions is in a form of RDD[(Int, Option[String])].

countByKey() counts the number of countries where the product was sold. It will return a Map[Int,Long].

**Running the resulting jar**

The best way to run a spark job is using spark-submit.

/usr/bin/spark-submit --class main.scala.com.matthewrathbone.spark.Main --master local ./target/scala-spark-1.0-SNAPSHOT-jar-with-dependencies.jar /path/to/transactions\_test.txt /path/to/users\_test.txt /path/to/output\_folder

1 3

2 1

**Testing**

As with other frameworks the idea was to follow closely the existing official tests in [Spark GitHub](https://github.com/apache/spark/blob/master/core/src/test/java/org/apache/spark/JavaAPISuite.java), using scalatests and JUnit in our case.

class SparkJoinsScalaTest extends AssertionsForJUnit {

var sc: SparkContext = \_

@Before

def initialize() {

val conf = new SparkConf().setAppName("SparkJoins").setMaster("local")

sc = new SparkContext(conf)

}

@After

def tearDown() {

sc.stop()

}

@Test

def testExamleJobCode() {

val job = new ExampleJob(sc)

val result = job.run("./transactions.txt", "./users.txt")

assert(result.collect()(0).\_1 === "1")

assert(result.collect()(0).\_2 === "3")

assert(result.collect()(1).\_1 === "2")

assert(result.collect()(1).\_2 === "1")

}

}

The test is fairly straightforward. We have checked at the end that the expected result is equal to the result that was obtained through Spark.

**Thoughts**

Spark is used for a diverse range of applications. It contains different components: Spark Core, Spark SQL, Spark Streaming, MLlib, and GraphX. These libraries solve diverse tasks from data manipulation to performing complex operations on data.

In addition, Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

The tool is very versatile and useful to learn due to variety of usages. It’s easy to get started running Spark locally without a cluster, and then upgrade to a distributed deployment as needs increase.

**Spark Resources**

The Spark [official site](http://spark.apache.org) and [Spark GitHub](https://github.com/apache/spark) contain many resources related to Spark.