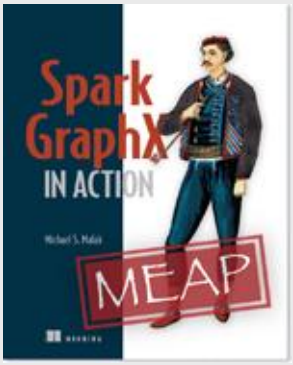


First steps with GraphX using the Spark Shell

By Michael S. Malak

In this article, we will download some sample graph data, and using the Spark Shell, quickly determine which out of a series of papers has been cited the most frequently.

This article is excerpted from Spark GraphX in Action. Save 39% on Spark GraphX in Action *with code 15dzamia at manning.com.*

The Spark Shell is the easiest way to quickly start using Spark. No compilation is necessary. In this article, we will download some sample graph data, that of bibliographic citations. Using the Spark Shell, we will quickly determine which paper has been cited the most frequently. More interestingly, we will invoke the PageRank algorithm built into GraphX to find the "most influential" paper in the graph network.

This article is intended to walk you through the steps of working with GraphX without delving into the details.

Getting Set Up and Getting Data

While normally you would write a Spark program in Scala (or Java or Python), compile it, and submit it to a Spark cluster, Spark also offers the Spark Shell, which is an interactive shell where you can quickly test out ideas.

Now, assuming you have Spark installed, simply type

```
./spark-shell
```

This assumes the spark/bin directory is in your path, but no paths really need to be set up. You can just execute it from any directory. You should see something similar to:

```
[mmalak@localhost bin]$ ./spark-shell
Welcome to                                                    #A
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.1
      /_/

Using Scala version 2.10.4
(OpenJDK 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.                               #B

scala>
```

#A Many lines of log output not shown here
#B More log lines not shown here

You'll notice toward the end, the Spark Shell helpfully alerts us to the variable sc being available to us. The Spark Shell helpfully instantiates an `org.apache.spark.SparkContext` for us with variable name `sc`. `sc` is our handle to the Spark world as we will see later.

The next part is getting some data to work with. Perhaps you have your own. But in this article we'll be downloading some data from the Stanford Network Analysis Project (SNAP) at http://snap.stanford.edu/data.

We'll be using the "High-energy physics theory citation network" data set (not to be confused with the collaboration network also available there), available for download from http://snap.stanford.edu/data/cit-HepTh.html. It is a little over 1 MB compressed as cit-HepTh.txt.gz, and decompresses to 6 MB. The start of this cit-HepTh.txt looks like:

```
# Directed graph (each unordered pair of nodes is saved once):
# Paper citation network of Arxiv High Energy Physics Theory category
# Nodes: 27770 Edges: 352807
# FromNodeId  ToNodeId
1001          9304045
1001          9308122
1001          9309097
1001          9311042
1001          9401139
1001          9404151
1001          9407087
1001          9408099
```

Comment lines begin with `#` and each data line represents one edge of the graph, with the vertex IDs of the start and end vertices. In this case, each vertex ID refers to a particular physics paper listed in the companion file cit-HepTh-abstracts.tar.gz, which you can optionally download if you wish to try to match up these bare numbers with something tangible.

This happens to be the file format recognized by GraphX, and this is the case with most or all of the data from SNAP.

NOTE The other major standard graph file format is called Resource Description Framework (RDF), plus derivatives such as N3 (Notation 3). As of Spark 1.2.0, GraphX does not have built-in the capability to read RDF.

Interactive GraphX Querying Using the Spark Shell

Now we'll be using the Spark Shell to load the HepPh data set and query it.

DEFINITION The Spark Shell also is an example of a REPL, which stands for Read-Eval Print Loop, which are the names of the four LISP programming language primitives used to implement the first interactive LISP shell in the 1960's. Scala, Python and other languages commonly have REPLs now, and the Spark REPL builds on the Scala REPL.

To avoid worrying about paths, it can be simpler to just copy Cit-HepTh.txt right into the same directory as spark-shell:

1. `cp Cit-HepTh.txt` into the same directory as `spark-shell`
2. `./spark-shell`
3. Now, with three lines entered into the Spark Shell, we can find the most-referenced paper:

```
import org.apache.spark.graphx._
val graph = GraphLoader.edgeListFile(sc, "Cit-HepTh.txt")
graph.outDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
```

We'll enter these lines one by one and explain each one as we go:

```
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._

scala>
```

Similar to the Java import, here Scala uses an underscore as a wildcard instead of Java's use of asterisk for the same purpose.

SCALA TIP Scala uses underscore in about a dozen different distinct ways. All the ways are some sort of wildcard or placeholder capacity, which makes it seem like the underscore has one solitary meaning. But it doesn't. Don't assume the underscore means something specific to like what you've seen previously when you see it in a new context.

```
scala> val graph = GraphLoader.edgeListFile(sc, "cit-HepTh.txt")
14/12/14 23:04:06 INFO MemoryStore: ensureFreeSpace...              #A
graph: org.apache.spark.graphx.Graph[Int,Int] =
 org.apache.spark.graphx.impl.GraphImpl@15721cbd

scala>
```

#A Many log lines not shown

The Spark Shell tells us it successfully created a variable called `graph` of type `org.apache.spark.graphx.Graph[Int,Int]`. Let's look at that one line in more detail:

Every "variable" in Scala is either `val` (constant, like Java's `final`) or `var` (variable). Good Scala programmers try to make everything `val` whenever possible, as "immutable" data is preferred when doing functional programming.

Recall that `sc` is that `SparkContext` that the Spark Shell helpfully created for us upon startup.

```
val graph = GraphLoader.edgeListFile(sc, "Cit-HepTh.txt")
```

`graph` is the name of the value (variable) being declared here, but where is the type? Scala is statically typed but uses inferred typing. The Scala compiler knows what type to make `graph` due to the return type of the `edgeListFile()` method, and that return type happens to be `org.apache.spark.graphx.Graph`. Once the compiler makes its decision, the type for `graph` can never change. Even though it might look like it due to its brevity, Scala is not like interpreted scripting languages like Perl where Perl's variables can change types while the program is running. Scala is a strictly and statically typed language; it's just not wordy.

Finally, for the third last line to be entered into the Spark Shell, and this will give us the ID of the theoretical physics paper that is most frequently cited:

```
scala> graph.outDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
14/12/14 23:50:56 INFO SparkContext: Starting job: ...
#A
14/12/14 23:50:59 INFO SparkContext: Job finished: reduce at
 <console>:18, took 3.16079562 s
res0: (org.apache.spark.graphx.VertexId, Int) = (9905111,562)

scala>
```

#A Many log lines not shown

Thus, paper ID 9905111 (the 111th paper from May, 1999) was cited the most, by 562 other papers to be specific. But take a look at the breakdown of that line of code for everything that is going on there.

We were able to get the most cited paper with just three lines of code, one of which was an import, and the last two of which we could have combined into a single line if we really wanted to show off.

But this example really hasn't taken advantage of the power of graphs. We could have done this in SQL on a relational database using a GROUP BY. Next, we'll use the power of GraphX by using its PageRank algorithm on this same data.

PageRank Example

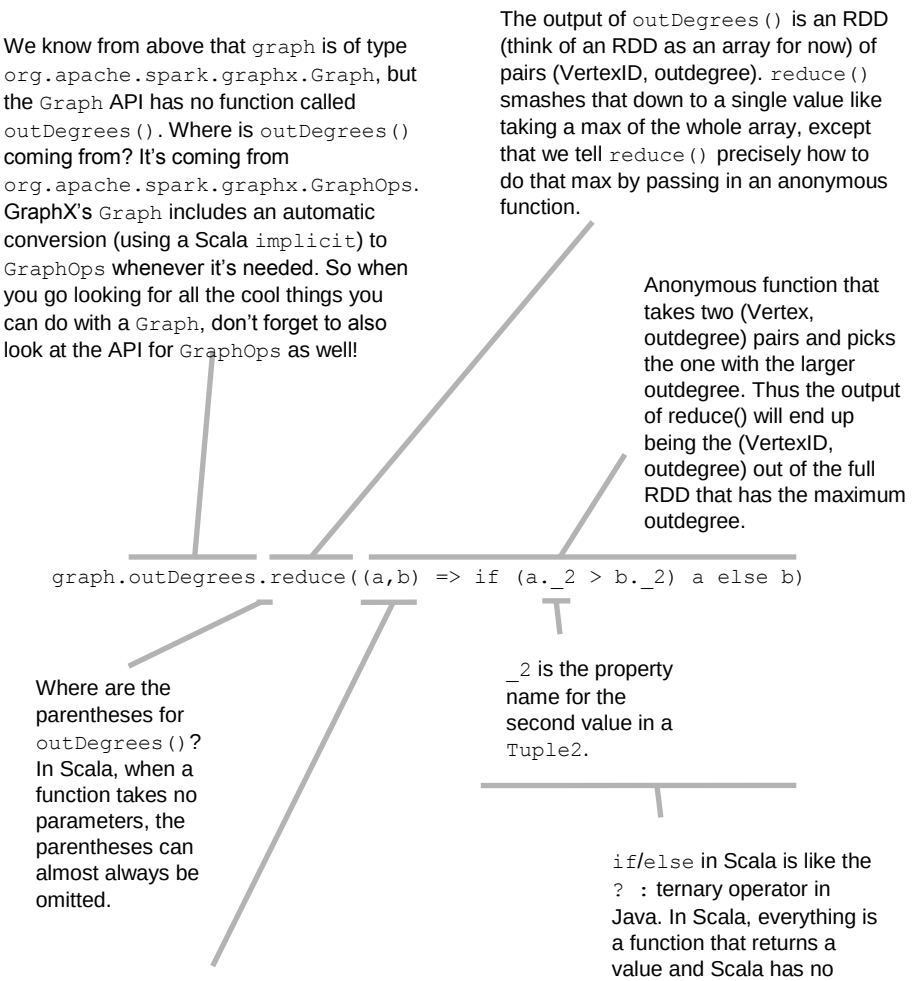
Although Larry Page of Google invented the PageRank algorithm (hence the name) to rank webpages on the entire World Wide Web, the algorithm can be used to measure the influence of vertices in any graph. Before we apply PageRank to our theoretical physics citation network, though, let me explain one thing lurking behind the scenes. Let's take a look at the vertices of our graph:

```
scala> graph.vertices.take(10)
res2: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((9405166,1),
 (108150,1), (110163,1), (204100,1), (9407099,1), (9703222,1),
 (9709148,1), (9905115,1), (103184,1), (211245,1))
```

NOTE Henceforward, I will omit all the log lines that Spark Shell spews out, and I won't even mention anymore that I am omitting them. Whenever you see a Spark Shell interaction from now on, assume there is a bunch of log output that is not shown here.

So, surprise, surprise: the vertices in our graph aren't just the vertex IDs from the Cit-HepTh.txt file; they're actually Scala pairs where the second number of the pair is always the number 1. GraphX is actually natively a property graph processor, which means it allows vertices and edges to have their own properties. Here, the 1 values are the properties of the vertices, and these 1 values are arbitrarily attached to the vertices by that `GraphLoader.edgeListFile()` function we used – just so that the vertices have some property even though it has no meaning. GraphX can handle vertices with properties but edge list files have no properties.

The reason I bring this up now is because the pageRank() method of `Graph` returns a new `Graph`, where each vertex has a property which is of type `Double` and is the PageRank for that vertex.


We know from above that graph is of type org.apache.spark.graphx.Graph, but the Graph API has no function called outDegrees(). Where is outDegrees() coming from? It's coming from org.apache.spark.graphx.GraphOps. GraphX's Graph includes an automatic conversion (using a Scala implicit) to GraphOps whenever it's needed. So when you go looking for all the cool things you can do with a Graph, don't forget to also look at the API for GraphOps as well!
The output of outDegrees() is an RDD (think of an RDD as an array for now) of pairs (VertexID, outdegree). reduce() smashes that down to a single value like taking a max of the whole array, except that we tell reduce() precisely how to do that max by passing in an anonymous function.
Anonymous function that takes two (Vertex, outdegree) pairs and picks the one with the larger outdegree. Thus the output of reduce() will end up being the (VertexID, outdegree) out of the full RDD that has the maximum outdegree.
graph.outDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
Where are the parentheses for outDegrees()? In Scala, when a function takes no parameters, the parentheses can almost always be omitted.
_2 is the property name for the second value in a Tuple2.
if/else in Scala is like the ? : ternary operator in Java. In Scala, everything is a function that returns a value and Scala has no "imperative" if/else at all!
The parameters to our anonymous function are (VertexID, outdegree). Here we declare the parameter names to be a and b, and we declare they are coming in in the form of a Scala Pair (Tuple2). Parentheses are all that are need to declare a Tuple2 in Scala, and the Scala compiler is smart enough to figure out that these particular parentheses mean a Tuple2 declaration as opposed to some more common numeric use of parentheses.

Let's run PageRank:

```
scala> val v = graph.pageRank(0.001).vertices
v: org.apache.spark.graphx.VertexRDD[Double] = VertexRDD[1264] at RDD at
 VertexRDD.scala:58
```

And now, let's look at the first ten vertices of our PageRank graph:

```
scala> v.take(10)
res3: Array[(org.apache.spark.graphx.VertexId, Double)] =
 Array((9405166,1.336783076434938), (108150,0.5836164464324066),
 (110163,0.15), (204100,0.19080382117882116),
 (9407099,0.8271044254712047), (9703222,0.16521611205688394),
 (9709148,0.22176221523472583), (9905115,0.38267418598941183),
 (103184,0.20437621370972553), (211245,0.2298299371239072))
```

Those floating point numbers are the PageRanks; as you can see, for the first ten at least, they range from 0.15 up to 1.34.

Now let's run reduce() on v to find the vertex with highest PageRank:

```
scala> v.reduce((a,b) => if (a._2 > b._2) a else b)
res4: (org.apache.spark.graphx.VertexId, Double) =
 (9207016,85.27317386053808)
```

So some paper from July, 1992 is the most influential, at least according to the PageRank algorithm.

Summary

The Spark REPL is very powerful and should not be discounted. It has been touted as a tool for "data scientists." It is also useful, though, for Spark developers, in the same way as, for example, the JavaScript console in web browsers for JavaScript developers.