Get Ready with **GraphFrames** in Less than 30 Minutes

This article shows how to work with the GraphFrames package for Apache Spark using the Cloudera QuickStart VM. You will learn how to perform scalable graph analysis on top of your Enterprise Data Hub (EDH). You will experiment with social network data from a public repository. A second exercise uses crawl results obtained from the web by Apache Nutch. Our goal is to inspect the graph's topology using CDH 5.

Introduction

Graphs – also called networks – are everywhere! In general, a graph consists of nodes and edges. A node can be any object, such as a person or an airport, and an edge is a relation between two nodes, such as a friendship or a connection between two cities provided by an airline. Social networks are very common. Many research projects and also very practical use cases are related to social networks. Content networks – which consist of interlinked documents, such as linked web pages or citation networks, are two more examples of graphs. Finally, the "Internet of Things" is also a huge graph that spans multiple existing graphs.

Why should we care about graph theory and its applications? Because the relations between things, or the links in a network, also contain information, one can gain more insights by taking the topology of a system into account, compared to a simple analysis of individual elements. Depending on the structure of the graph, such a system can be robust or fragile. Disturbing influences have more or less no impact on a robust system, but a fragile one can collapse. How can we find out such properties? By analyzing the graph structure and by comparison with well-known graph models, such as the Erdös Renyi model or the Barabasi Albert model.

A very famous metric – PageRank – is calculated per node (or vertex). The PageRank is related to the importance of a node in the network and results from the topology (represented by links). Other metrics exist, such as betweenness centrality, but currently this measure is not yet implemented in GraphX or in the GraphFrames package. This is why we focus on PageRank for now. The clustering coefficient of a graph can be calculated from the number of triplets and the number of fully connected triangles - and both can be obtained from a GraphFrame.

How was the page graph created? First, we defined a seed list - a bunch of URLs. Apache Nutch takes this list, fetches and parses the HTML content in order to get more outgoing links from that page. This happens in multiple iterations and finally, all that links and the raw page content are stored in an Apache HBase table. We wrapped an Apache Hive table around this. This also allows exporting the content as Parquet file. We use Hive because our version of Apache Impala (incubating) did not yet support the complex data type `Map<String,String>`, which is used to hold both link lists (one for incoming links and one for outgoing links).

A common **use case** for graph analysis is "topology analysis". Topology analysis is very relevant in different areas, such as fraud detection analysis and analysis of phase transitions in complex systems. From topology we can derive information about graph properties, and if we are able to track changes in topology, we are able to track changes in the entire system. In this way, topological analysis is a handy tool for purely data driven methods in financial risk management, predictive maintenance, and many other fields.

Prerequisites

You need a cluster with Apache Hadoop (here we work with CDH 5.5.0) and Apache Spark (version 1.5.0). An Internet connection for dynamic download of required packages is also needed. The GraphFrame package (currently version 0.0.1) is available from the spark-packages repository. We grab a public data set from the web and a second one is provided in our Github repository.

You should know the concepts of graph analysis. The introduction section provides links to additional material. This should help you to explore the context and to recap or to warm up a bit faster if graph analysis is new to you. Furthermore, you should know how to use RDDs and DataFrames in Spark.

Finally, you can check out the scripts used in the exercise session together with some example data from our Github repository.

If all the prerequisites are met, you should see first results within the next minutes, but in case you do not have a cluster at hand to start the experiments, the article guides you step by step to graph analysis with GraphFrames. The symbol **$** indicates an action on the Linux shell and scala> is related to commands you enter into the spark-shell. If no such symbol exists please go back to the last one, which highlights the current context, until we have to switch again.

Hands-On Exercise

The next steps explain in general what we have to do. We use a Linux shell and the Spark shell. For simplicity we skip some code, but you find a complete listing at the end of the article and in the Github repository.

Preparation

1. Load a data file from the SNAP website: https://snap.stanford.edu/data/egonets-Facebook.html

We need the file named `facebook_combined.txt.gz` from https://snap.stanford.edu/data/facebook_combined.txt.gz. This file can be downloaded, decompressed, and loaded into HDFS with the bootstrap script in our Github repository. But you can do all that manually:

```
$ wget https://snap.stanf ord.edu/data/facebook_combined.txt.gz
$ gunzip *.gz
$ hdfs dfs -put facebook_combined.txt
```

2. Start a spark-shell with the optional packages to work with GraphFrame and the CSV processing libraries:

```
$ spark-shell --packages graphframes:graphframes:0.1.0-spark1.5,\
com.databricks:spark-csv_2.10:1.4.0
```

For the Spark package graphframes version 0.2.0 on Spark 1.6.0 you would have to run the spark-shell with the following arguments:

```
$ spark-shell --master local[2] --packages graphframes:graphframes:0.2.0-spark1.6-s_2.10,
com.databricks:spark-csv_2.10:1.4.0
```

Note, that the `\` symbol indicates that input continues on the next line. The `--package` option allows you to specify a comma separated list of Apache Maven coordinates for required third-party packages used in your spark-shell session. Those will automatically be loaded by the Spark cluster from a central repository. Here we need the GraphFrame package and a library for CSV parsing from Databricks.

Build the Graph

3. Define a DataFrame by loading data from a CSV, which is stored in HDFS:

Our datafile `facebook_combined.txt` contains two columns to represent links between network nodes. The first column is called source (src) and the second is the destination (dst) of the link. Other systems, such as Gephi use *source* and *target* instead. First we define a custom schema, and than we load the DataFrame, using the sql context.

```
scala> val customSchema = StructType(Array(StructField("src", In  tegerType,
true),StructField("dst", IntegerType, true)))
```

```
scala> val df = sqlContext.read.format("com.databricks.spark.csv  ").option("header",
"false").option("delimiter", " ").schema(customSchema).load("facebook_combined.txt")
```

Here we use space (" ") as a specific delimiter. More options can be defined as shown in the documentation of the CSV parsing library.

In theory, this edge list contains all information to build a graph, but a GraphFrame needs nodes data and edges in separate DataFrames. Let's transform the edge list to get also a node list.

First, we have to extract all unique node IDs into a column with name "name" and than we add a column with name "id" to it. Currently, there is no equivalent to a `row_number()` function available, but in version 1.6 the component called **MonotonicallyIncreasingID.scala** https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/MonotonicallyIncreasingID.scala provides monotonically increasing 64-bit integers. For now it is fine to use the node ID as row id as well.

```
val n1 = edges.select("src").distinct()
val n2 = edges.select("dst").distinct()
val n = n1.unionAll(n2).withColumnRenamed("src","name")
val nodes = n.withColumn("id", n("name"))
edges.show()
nodes.show()
val g1 = GraphFrame(nodes, edges)
```

All this was preparation to get the graph data into the right shape by using DataFrames. But now we are ready to analyze the graph.

Graph Analysis with a GraphFrame

Multiple options for scalable graph analysis on top of Hadoop coexist. Nowadays, Apache Spark contains GraphX. Historically, the no longer supported Bagel project was used. Bagel offers an implementation of the Google Pregel graph processing framework on a pretty low level.

The Bagel documentation tells us: *"Bagel operates on a graph represented as a distributed dataset of (K, V) pairs, where keys are vertex IDs and values are vertices plus their associated state."* The core idea is, that: *"In each superstep, Bagel runs a user-specified compute function on each vertex that takes as input the current vertex state and a list of messages sent to that vertex during the previous superstep, and returns the new vertex state and a list of outgoing messages."* This makes Bagel comparable with Apache Giraph which implements the same concept. Both are low level abstractions and useful starting points for implementation and development of new graph analysis algorithms.

The GraphX framework uses a property graph model. The property graph is a widely used high level abstraction to represent a graph. In GraphX, the data is managed by RDDs, one RDD for nodes and one for edges. The graph operations have not to be implemented by a particular compute function like in Bagel. Instead, the GraphX library offers the graph class which provides access to nodes, edges, and even to functionality to analyze the graph. The Pregel API is also available on this level in GraphX.

Recently, the GraphFrame library (version 0.0.1) for Apache Spark was released by Databricks. Although this software is not ready for production - nor is it currently supported by Cloudera - I suggest to get in touch with it early. You may ask: Will this become a replacement for GraphX? I personally like to answer with an analogy: *What a DataFrame is to an RDD, a GraphFrame is to the graph class in GraphX.*

With GraphFrames you can simply define a graph from your DataFrames. One holds the node list and the other the edge list. No matter how the data was stored in EDH, DataFrames allow all the preparation like filtering grouping and projections before the specific graph algorithms, implemented in the GraphFrame class, are applied. Results of such computations can be new columns in the node list. And because this is a DataFrame, we can easily join and finally export results from multiple operations as demonstrated later.

4. Perform some graph analysis on the facebook ego-net graph:

We sort the nodes by degree and calculate the PageRank. Finally we count the triangles a node is embedded in.

```
val k = g1.degrees.sort(desc("degree"))
k.show()

val pr2 = g1.pageRank.resetProbability(0.15).maxIter(10).run()
pr2.vertices.show()
```

Which are the nodes with the highest PageRank?

```
val pr3 = pr2.vertices.sort(desc("pagerank"))
pr3.show()
```

```
+----+----+------------------+
|name|  id|          pagerank|
+----+----+------------------+
|3434|3434|18.184854992469376|
|1911|1911|18.123485211404812|
|2655|2655|17.525235604357597|
|1902|1902| 17.35563914066005|
|1888|1888| 13.34316669756043|
+----+----+------------------+
only showing top 5 rows
```

Into how many triangles are our nodes embedded?

```
val tcr = g1.triangleCount.run()
```

Because we plan to use this data later again, we can use the persist() function, but this makes only sense if the dataset is not too big.

```
tcr.persist()
tcr.sort(desc("count")).limit(4).show()
```

```
+-----+----+----+
|count|name|  id|
+-----+----+----+
|30025|1912|1912|
|15502|2543|2543|
|15471|2233|2233|
|15213|2464|2464|
+-----+----+----+
only showing top 4 rows
```

Finally, we use this results to evaluate if the number of triangles is related to the PageRank. We join both results and export the table as a CSV file. This allows us to draw the scatter plot in gnuplot or R.

`tcr` is the triangle count result. We did not persist it, so we have to calculate it again next time we do the join. The PageRank was calculated as a per vertex value in `pr2.vertices` DataFrame.

```
scala> tcr.printSchema
root
 |-- count: long (nullable = true)
 |-- name: integer (nullable = true)
 |-- id: integer (nullable = true)

scala> pr2.vertices.printSchema
root
 |-- name: integer (nullable = true)
 |-- id: integer (nullable = true)
 |-- pagerank: double (nullable = true)
```

Since both datasets are DataFrames, we can simply use the DataFrame functionality to join both on the id column:

```
scala> val scatter = pr2.vertices.join(tcr, pr2.vertices.col("id").equalTo(tcr("id")))
scala> scatter.limit(10).show
```

```
16/05/10 03:24:53 INFO DAGScheduler: Job 18 finished: show at <console>:51,   took 653,309080 s
+----+----+-------------------+-----+----+----+
|name|  id|           pagerank|count|name|  id|
+----+----+-------------------+-----+----+----+
|  31|  31|0.16137995810046757|  110|  31|  31|
| 231| 231|0.35635621183428823|   95| 231| 231|
| 431| 431| 0.2495015120580725| 1248| 431| 431|
| 631| 631|0.23890006729816274|   41| 631| 631|
| 831| 831| 0.3651072380090138|   64| 831| 831|
|1031|1031|0.15013201313406596|    5|1031|1031|
|1231|1231| 0.2566120648129746| 1452|1231|1231|
|1431|1431|0.44670540616673726|10441|1431|1431|
|1631|1631| 0.285116150576581|   35|1631|1631|
|1831|1831| 2.1506290231917156|  226|1831|1831|
+----+----+------------------+-----+----+----+
```

We export this result as a CSV file but in order to get only one result file we use the `coalesce` function of the DataFrame:

```
scatter.coalesce(1).write.format("com.databricks.spark.csv").option("header",
"true").save("scatter.csv")
```

Alternatively you could also have used the getmerge command from HDFS to combine all the parts files into one single file, but in this case it is important to use the option header=false which turns the header lines off. Now, let's plot our previous result data set in gnuplot. It was already exported into a folder names `scatter_1.csv` but to HDFS.

If gnuplot is not available, simply install it, for example by typing `sudo yum install gnuplot` on CentOS. We have to load it to the local folder and then using the following gnuplot commands stored in a script file named `plot.cmd` :

```
Commands for plotting the scatterplot from two combined result sets.
Filename: plot.cmd
```

```
set datafile separator ","
set term png
set output "g1.png"
#
#  Add titles and labels.
#
set xlabel "PageRank"
set ylabel "Triangles"
set title "PR vs. T"
unset key
#
#  Add grid lines.
#
set grid
set size ratio -1
#
#  Timestamp the plot.
#
set timestamp
plot 'scatter_1.csv/part-00000' using 3:4
```

Listing 1

we can easily plot the figure as shown in table 1 (left), assuming, that the data set and the plot commands are in the same directory:

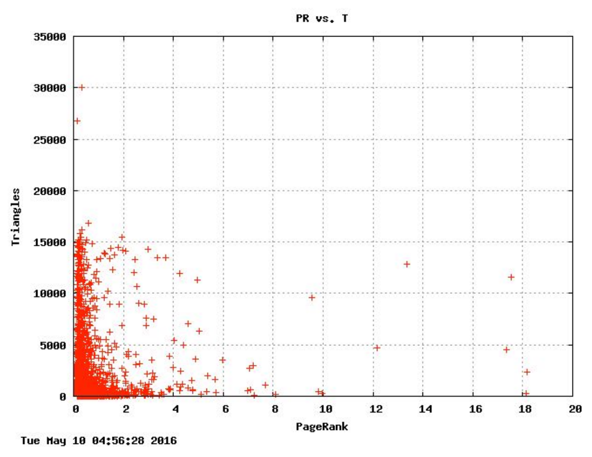
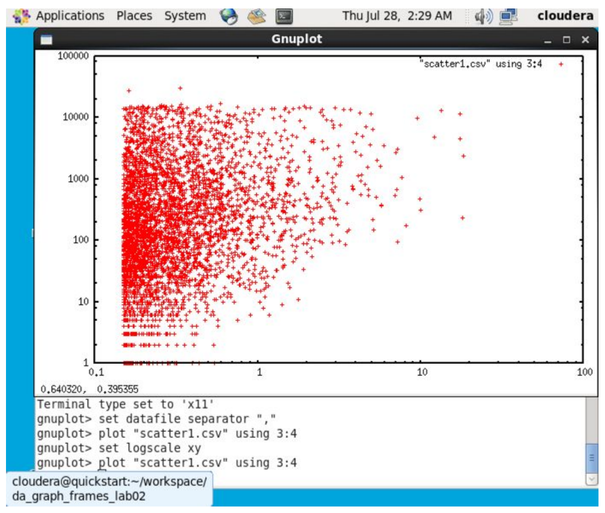
```
$ gnuplot plot.cmd
```

In a previous post we used the library scalaplot to create charts. You can simply add the Maven coordinates during startup:

```
$  spark-shell --packages graphframes:graphframes:0.1.0-spark1.5,\
com.databricks:spark-csv_2.10:1.4.0, org.sameersingh.scalaplot:scalaplot:0.0.4
```

And now, those imports `import org.sameersingh.scalaplot.Implicits._` allow you to plot charts in gnuplot or based on the JFreeChart library.

This was a fun ride, starting with some public data, loading the edges, creating the node list and defining a GraphFrame in Apache Spark followed by analyzing the graph and merging partial results for the final plot. Finally you should get this result:

Scatter Plot	Log-Log Plot
	 Applications Places System Thu Jul 28, 2:29 AM cloudera Gnuplot "scatter1.csv" using 3:4 1 10 100 1000 10000 100000 0.1 1 10 100 0.640320, 0.395355 cloudera@quickstart:~/workspace/da_graph_frames_lab02

```
Terminal type set to 'x11'
gnuplot> set datafile separator ","
gnuplot> plot "scatter1.csv" using 3:4
gnuplot> set logscale xy
gnuplot> plot "scatter1.csv" using 3:4
```

Table 1

This scatter plot (left image) is not yet the best way to understand the data, but a good starting point. You should consider to plot a 2D histogram[1], which encodes the number of (PageRank,Triangle) pairs as well. The right panel shows a second approach to inspect the data, using a log-log plot. This would reveal a power law distribution if a straight line would be shown, or a stretched power-low in case of a bended curve. The relation between PageRank of a node and the number of triangles a node is part of is not a power law. But from the left panel, we can already conclude, that a high number of triangles does not guarantee a high page rank and also, that a high PageRank can be found for nodes which are embedded in a small number of triangles and also for nodes with higher triangle embedding, but the two highest values were found for a pretty small PageRank.

[1] You can find an example here: http://pelican.rsvs.ulaval.ca/mediawiki/index.php/Making_density_maps_using_Gnuplot

Now, let’s do it again with a different data set.

Example 2: Analyze the Topology of Nutch Crawl Results

To prepare the data set for this example I used the Apache Nutch web crawler engine. Apache Nutch version 2.3.1 was built on CDH 5.5.0 in order to collect real content from the world wide web. Although, data ingestion from SQL databases and huge document repositories dominate in commercial use cases, a data scientist should also be aware of the advantages of Apache Nutch beside using our good old friends: Apache Flume and Apache Sqoop.

In some cases, for example if you have to do a market screening you want to load data from specific web pages, portals, or social media applications. Some have a specific API, such as Twitter and Facebook. But instead of learning all the REST APIs and writing another client you can use a crawler to get the data shown by your browser into your cluster. Here you apply specific parsing, such as HTML parsing with JSOUP or more advanced triple extraction using Any23. What we do here can already be seen as the groundwork for advanced web scraping projects. An additional article will tell you exactly how to setup Nutch and how to crawl the web for your own data sets to conduct your own research or simply funny experiments.

Access the Page Graph Data

Our crawl data is loaded into a staging table (here it was an HBase table but MySQL or Cassandra could be used as well). Beside the full HTML content it contains inlinks and outlinks. Because HBase is managing the crawl data (and the state of the crawl dataset), we can add more links, especially inlinks to a particular page while we crawl more and more data. To distinguish both in visualization and we use link type 1 is for inlinks and type 2 for outlinks. Finally, in order to analyze the overall network, we need both lists in one homogeneous collection, where each link is described by `source`, `target`, and `type`.

Creation of Node- and Link-Lists

Using Spark SQL and DataFrames allows us the inspection and rearrangement of the data in just a few steps. We start with a query which transforms the map of string pairs into individual rows. During the crawl procedure, Apache Nutch aggregates all links in an *adjacency* list. We have to “explode” the data structure with all links per page to create a *link list* from the previously optimized representation.

```
scala> val r1 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT explode(inlinks) as
(source,page), baseurl as target, '1' as type")
r1.limit(4).show()
```

```
+--------------------+--------------------+--------------------+----+
|                 _c0|                 _c1|              target|type|
+--------------------+--------------------+--------------------+----+
|https://creativec...|       Marking guide|http://wiki.creat...|   1|
|https://creativec...|           More info|http://wiki.creat...|   1|
|https://creativec...|          Learn more|http://wiki.creat...|   1|
|https://creativec...|           More info|http://wiki.creat...|   1|
+--------------------+--------------------+--------------------+----+
```

As the listing shows, an exploded map has not yet any useful column name. Let's change this and by defining the names for the two columns, represented by the string pair.

```
val r1 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT explode(inlinks) as
(source,page), baseurl as  target, '1' as type")
```

For our network, we need one column named source, which contains the URL and the page name. This is why we first explode the inlinks map into "virtual columns" called `mt` and `mp`. We can now concatenate both and create a new source column:

```
val r1 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT   concat(mt,mp) as source,
baseurl as target, '1' as type,  explode(inlinks) as (mt,mp)")
```

```
+--------------------+--------------------+----+--------------------+--------------------+
|              source|              target|type|                  mt|                  mp|
+--------------------+--------------------+----+--------------------+--------------------+
|https://creativec...|http://wiki.creat...|   1|https://creativec...|       Marking guide|
|https://creativec...|http://wiki.creat...|   1|https://creativec...|           More info|
|https://creativec...|http://wiki.creat...|   1|https://creativec...|          Learn more|
|https://creativec...|http://wiki.creat...|   1|https://creativec...|           More info|
|https://creativec...|http://wiki.creat...|   1|https://creativec...|           More info|
|https://creativec...|http://wiki.creat...|   1|https://creativec...|           More info|
|https://creativec...|http://creativeco...|   1|https://creativec...|    Creative Commons|
|https://creativec...|https://creativec...|   1|https://creativec...|https://creativec...|
|https://creativec...|https://creativec...|   1|https://creativec...|             العربية|
|https://creativec...|https://creativec...|   1|https://creativec...|          Беларуская|
+--------------------+--------------------+----+--------------------+--------------------+
```

Now we apply this procedure to the second set of links, to our outlinks.

```
val r2 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT baseurl as source,
concat(mt,mp) as target, '2' as type, explode(outlinks) as (mt,mp)")
```

In order to merge both link lists into one we can ignore the temporary columns. We simply select the required columns, use the `unionAll` operator and store our page network as Parquet file in HDFS.

```
val r1c = r1.select("source","target","type")
val r2c = r2.select("source","target","type")
val rAll = r1c.unionAll( r2c )
rAll.writer.parquet( "full_link_list" )
```

With this intermediate result we can go back to our main topic, to GraphFrames. But let us recap quickly: **What was the role of Spark SQL in this situation?** *First, Spark gives us access to* *many different data sources, no matter if we use the Hive table in Parquet format or even the HBase table via Hive - the same query language allows us to prepare the raw data for graph analysis. Second, in many use cases we can not use the existing table layout. But using Spark SQL makes all the filtering, grouping and projection really easy.*

The variable `g1` represents the graph, loaded by the Nutch crawler. We can apply the same analysis steps as in exercise 1, but this is your homework!

Conclusion

Using GraphFrames it is now possible to turn data tables into graphs with just a few lines of code. The full power of the Pregel API, implemented in GraphX, is available in combination with Spark SQL. This means, raw data in different flavors can easily be combined to huge multi-layer graphs and the graph analysis is done in that place - inside the Enterprise Data Hub - using the scalability of Spark.

The next part of this series will present more background on GraphFrames and two more practical exercises. Please share your ideas or post any question in the comments section, and enjoy all your connected data!

Although the initial claim, expressed in the title of the article, was a bit "provocative" I hope you could get some experience or at least an idea of the GraphFrame package and how it is related to Spark DataFrames. If you plan to go deeper, it is worth to get familiar with the Google Pregel API. Developing your own graph analysis algorithms definitively requires more time and some experience with Spark, graph theory, and specific use cases, but this was not the scope of this introduction.

****** END ******

Final Notes:
The exercises are scripted and available in a Github repository. Please start with cloning the repository:

```
$ git clone https://github.com/kamir/da_graph_frames_lab02.git
$ cd da_graph_frames_lab02
$ ./load_examples.sh
```

Now you start the spark-shell:

```
$ spark-shell --packages graphframes:graphframes:0.1.0-spark1.5,\
com.databricks:spark-csv_2.10:1.4.0,org.sameersingh.scalaplot:scalaplot:0.0.4
```

Both exercises can be loaded with the :load operator of the Spark REPL:

```
scala> :load ex1.scala
scala> :load ex2.scala
```

Good Luck!

Scala code for analyzing the social network obtained from SNAP using the spark-shell.
Filename: ex1.scala

```
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType};
import org.graphframes._

val customSchema = StructType(Array(StructField("src", IntegerType, true),StructField("dst", IntegerType, true)))

val edges = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("delimiter", "
").schema(customSchema).load("facebook_combined.txt")

val n1 = edges.select("src").distinct()
val n2 = edges.select("dst").distinct()
val n = n1.unionAll( n2 ).withColumnRenamed("src","name").distinct()

val nodes = n.withColumn("id", n("name") )

edges.show()
nodes.show()

val g1 = GraphFrame(nodes, edges)

val k = g1.degrees.sort(desc("degree"))
k.show()

val pr2 = g1.pageRank.resetProbability(0.15).maxIter(10).run()

pr2.vertices.show()

val pr3 = pr2.vertices.sort(desc("pagerank"))
pr3.show()

val tcr = g1.triangleCount.run()
tcr.sort(desc("count")).show()

tcr.printSchema
pr2.vertices.printSchema

val scatter = pr2.vertices.join(tcr, pr2.vertices.col("id").equalTo(tcr("id")))

scatter.limit(20).show

scatter.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("scatter_1.csv")
```

Listing 2

Scala code for analysing the page graph provided as Parquet file using the spark-shell.
Filename: ex2.scala

```
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType};
import org.graphframes._

// We start loading the raw data from a parquet file and register it as a temporary table
//
val crawlDataFrame = sqlContext.read.parquet("yarn_ds1_run_3_webpage_parquet_1k")
crawlDataFrame.registerTempTable("yarn_ds1_run_3_webpage_parquet")

// Here we assume that we have a Hive table named yarn_ds1_run_3_webpage_parquet
//
val e1 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT concat(mt,mp) as source, baseurl as target, '1' as type,
explode(inlinks) as (mt,mp)")
val e2 = sqlContext.sql("FROM yarn_ds1_run_3_webpage_parquet SELECT baseurl as source, concat(mt,mp) as target, '2' as type,
explode(outlinks) as (mt,mp)")

val e1c = e1.select("source","target","type")
val e2c = e2.select("source","target","type")
val eAll = e1c.unionAll( e2c )
eAll.limit(10).show()

val nAll1 = eAll.select("source").distinct().withColumnRenamed("source","name")
val nAll2 = eAll.select("target").distinct().withColumnRenamed("target","name")
val nAll = nAll1.unionAll( nAll2 ).distinct()

val nodes = nAll.withColumn("id", nAll("name") )
nodes.limit(10).show()

val e2 = eAll.withColumnRenamed("source","src").withColumnRenamed("target","dst")

val g1 = GraphFrame(nodes, e2)

val k = g1.degrees.sort(desc("degree"))
k.show()

val pr2 = g1.pageRank.resetProbability(0.15).maxIter(10).run()

val pr3 = pr2.vertices.sort(desc("pagerank"))

val tcr = g1.triangleCount.run()
tcr.sort(desc("count")).show()

tcr.printSchema
pr2.vertices.printSchema

val scatter = pr2.vertices.join(tcr, pr2.vertices.col("id").equalTo(tcr("id")))

scatter.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("scatter_2.csv")
```

Listing 3