Big Data Management and Analytics
Session: Apache Spark GraphFrames/GraphX

Lecturer: Petar Jovanovic and Sergi Nadal

April 20th, 2016

In this session we will use Spark GraphFrames for distributed graph processing in the Spark ecosystem. GraphFrames is a package for Apache Spark which provides DataFrame-based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames.

It is highly advisable to complete the tasks listed in section 2 prior to attending the class, in order to get full advantage of the available in-class time.

1 Required Tools

- `eclipse` IDE with Java 7 and Maven plugin installed
- The following Java libraries (already included in the provided `pom.xml`):
  - `Spark Core`
  - `Spark GraphFrames`
  - `Spark GraphX`

2 Tasks To Do Before The Session

- Read the slides. It is not necessary to read the examples as they will be discussed in class.
- Checkout `GraphFrames` programming guide (`http://graphframes.github.io/user-guide.html`) and the `GraphX` programming guide (`https://spark.apache.org/docs/latest/graphx-programming-guide.html`).

3 Part A: Examples & Questions (30min)

During the first 30 minutes of session we will go through the 3 proposed examples in the slides. After each one, questions related to that particular topic will be addressed.

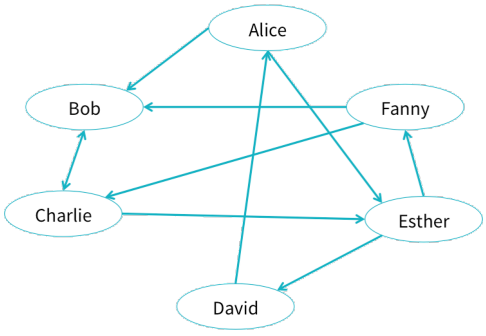
4 Part B: In-class Practice (2h30min)

4.1 Exercise 1 (15 min): Warmup

In the resources folder of the Java project we provide with GraphFrames library (as it is not yet published in Maven Central). Once you import the Java project in eclipse, in order to include it perform the following steps:

1. Right click on the Java project in left's *Package Explorer*.
2. Go to *Properties*.
3. Select *Java Build Path*
4. Go to the third tab, *Libraries*
5. Click *Add External JARs* and look for `graphframes-0.1.0-spark1.6.jar` in the `lib` folder of the Java project.

In `Exercise_1.java` we provide you with some code to introduce yourself to GraphFrames' API. This example depicts a social network. Say we have a social network with users connected by relationships. We can represent the network as a graph, which is a set of vertices (users) and edges (connections between users). A toy example is shown below.



To create a graph you need to provide the following inputs:

- Vertex DataFrame: A vertex DataFrame should contain a special column named `id` which specifies unique IDs for each vertex in the graph.

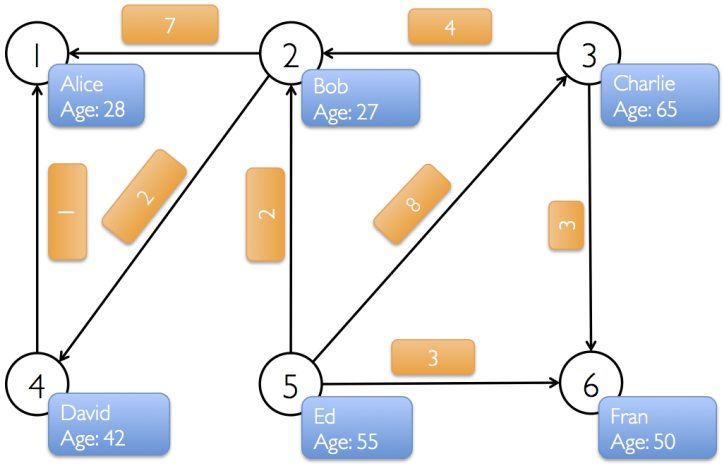
- Edge DataFrame: An edge DataFrame should contain two special columns: `src` (source vertex ID of edge) and `dst` (destination vertex ID of edge).

Both DataFrames can have arbitrary other columns. Those columns can represent vertex and edge attributes. In a social network, each user might have an age and name, and each connection might have a relationship type.

Run the example, and once you understood how the output and how the code works, proceed to the next exercise.

4.2 Exercise 2 (1h15min): Basic GraphFrames

Similarly as before, in this exercise we will work with a small simple graph. While this is hardly big data, it provides an opportunity to learn about the graph data model and the GraphFrames API. In this example we have a small social network with users and their ages modeled as vertices and likes modeled as directed edges.



4.2.1 Load the Graph

Though simple, it is unrealistic to believe graph data can be hardcoded. Thus, in this exercise you are provided with two data files representing people and likes between people respectively. You can find them under the `resources` folder of your Java project. Note the first line of the file depicts the attributes, for the sake of simplicity all numbers can be treated as Integers.

In this first part of the exercise you are asked to parse the files and load the graph from them. You should finally obtain a `GraphFrame` structure as follows:

```
1 GraphFrame G = new GraphFrame(vertices,edges);
```

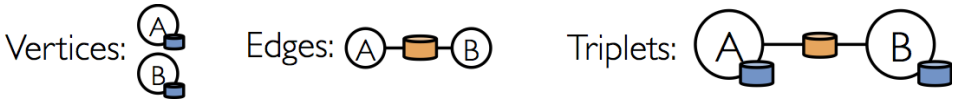
4.2.2 Graph Views

In many cases we will want to extract the vertex and edge RDD views of a graph (e.g., when aggregating or saving the result of calculation). As a consequence, the graph class contains members (`G.vertices` and `G.edges`) to access the vertices and edges of the graph.

Use `G.vertices` to display the names of the users that are at least 30 years old. To do so, obtain the `JavaRDD` structure of the set of vertices and apply the `filter` method. Further, the output should contain (in addition to lots of log messages):

```
David is 42
Fran is 50
Ed is 55
Charlie is 65
```

In addition to the vertex and edge views of the property graph, GraphFrames also exposes a triplet view. The triplet view logically joins the vertex and edge properties, this join can be expressed graphically as



The `EdgeTriplet` class extends the Edge class by adding the `srcAttr` and `dstAttr` members which contain the source and destination properties respectively.

Use the `G.triplets` view to display who likes who. The output should look like:

```
Bob likes Alice
Bob likes David
Charlie likes Bob
Charlie likes Fran
David likes Alice
Ed likes Bob
Ed likes Charlie
Ed likes Fran
```

If someone likes someone else more than 5 times than that relationship is getting pretty serious. Find the lovers:

```
Bob loves Alice
Ed loves Charlie
```

4.2.3 Querying the Graph

GraphFrames provide several simple graph queries, such as node degree. Also, since GraphFrames represent graphs as pairs of vertex and edge DataFrames, it is easy to make powerful queries directly on the vertex and edge DataFrames. Those DataFrames are made available as vertices and edges fields in the GraphFrame.

In this section of the exercise you are asked to implement the following queries:

- Find the youngest user's age in the graph.
- For each user show the number of likes received and given.

4.2.4 Motif Finding

Motif finding refers to searching for structural patterns in a graph.

GraphFrame motif finding uses a simple Domain-Specific Language (DSL) for expressing structural queries. For example, `G.find("(a)-[e]->(b); (b)-[e2]->(a)")` will search for pairs of vertices a,b connected by edges in both directions. It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. In this case, the returned columns will be `a, b, e, e2`.

DSL for expressing structural patterns:

- The basic unit of a pattern is an edge. For example, `(a)-[e]->(b)` expresses an edge `e` from vertex `a` to vertex `b`. Note that vertices are denoted by parentheses `(a)`, while edges are denoted by square brackets `[e]`.
- A pattern is expressed as a union of edges. Edge patterns can be joined with semicolons. Motif `(a)-[e]->(b); (b)-[e2]->(c)` specifies two edges from `a` to `b` to `c`.
- Within a pattern, names can be assigned to vertices and edges. For example, `(a)-[e]->(b)` has three named elements: vertices `a`,`b` and edge `e`. These names serve two purposes:
  - The names can identify common elements among edges. For example, `(a)-[e]->(b); (b)-[e2]->(c)` specifies that the same vertex `b` is the destination of edge `e` and source of edge `e2`.

  - The names are used as column names in the result DataFrame. If a motif contains named vertex `a`, then the result DataFrame will contain a column `a` which is a StructType with sub-fields equivalent to the schema (columns) of [[GraphFrame.vertices]]. Similarly, an edge `e` in a motif will produce a column `e` in the result DataFrame with sub-fields equivalent to the schema (columns) of [[GraphFrame.edges]].

- It is acceptable to omit names for vertices or edges in motifs when not needed. E.g., `(a)-[]->(b)` expresses an edge between vertices `a`,`b` but does not assign a name to the edge. There will be no column for the anonymous edge in the result DataFrame. Similarly, `(a)-[e]->()` indicates an out-edge of vertex a but does not name the destination vertex.

To end this exercise, you are asked to implement the following motif findings:

- Recommend who to follow, find triplets of users A,B,C where A follows B and B follows C, but A does not follow C.
- Find the subgraph of people with 3 likes or more.

4.3 Exercise 3 (1h): Analyzing Wikipedia Articles Relevance

Wikipedia provides XML dumps[1] of all articles in the encyclopedia. For the sake of this exercise, you are provided with a shortened dataset. The resulting dataset is stored in two files in the `resources` folder of your Java project: `wiki-vertices.txt` and `wiki-edges.txt`. The former contains articles by ID and title and the latter contains the link structure in the form of source-destination ID pairs.

4.3.1 Load the Graph

Similarly as before, first load the graph from the files. However, note now that headers are not specified. Also, take into account that names have multiple spaces between them, do not trim any data.

4.3.2 Relevance Analysis

We can now do some actual graph analytics. For this example, we are going to run PageRank to evaluate what the most important pages in the Wikipedia graph are. PageRank is one of the methods Google uses to determine a page's relevance or importance. It is only one part of the story when

[1] `https://en.wikipedia.org/wiki/Wikipedia:Database_download#English-language_Wikipedia`

it comes to the Google listing, but the other aspects are discussed elsewhere (and are ever changing) and PageRank is interesting enough. In this exercise you are asked to use PageRank to provide the 10 most relevant articles from the provided Wikipedia dataset. The following subsections provide a little more detail in order to understand how PageRank works.

4.3.2.1 About PageRank

In short PageRank is a "vote", by all the other pages on the Web, about how important a page is. A link to a page counts as a vote of support. If there is no link there is no support (but it is an abstention from voting rather than a vote against the page). Quoting from the original Google paper, PageRank is defined like this:

> We assume page A has pages $T_1, \ldots, T_n$ which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also $C(A)$ is defined as the number of links going out of page A. The PageRank PR of a page A is given as follows:
>
> $PR(A) = (1-d) + d(PR(T_1)/C(T_1) + ... + PR(T_n)/C(T_n))$
>
> Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.
>
> PageRank or $PR(A)$ can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

but that's not too helpful so let's break it down into sections.

1. $PR(T_n)$ - Each page has a notion of its own self-importance. That is $PR(T_1)$ for the first page in the web all the way up to $PR(T_n)$ for the last page

2. $C(T_n)$ - Each page spreads its vote out evenly amongst all of it's outgoing links. The count, or number, of outgoing links for page 1 is $C(T_1)$, $C(T_n)$ for page n, and so on for all pages.

3. $PR(T_n)/C(T_n)$ - so if our page (page A) has a backlink from page n the share of the vote page A will get is $PR(T_n)/C(T_n)$.

4. $d(...$ - All these fractions of votes are added together but, to stop the other pages having too much influence, this total vote is "damped down" by multiplying it by 0.85 (the factor d).

5. $(1-d)$ - The $(1–d)$ bit at the beginning is a bit of probability math magic so the "sum of all web pages' PageRanks will be one": it adds in the bit lost by the $d(....$ It also means that if a page has no links to it (no backlinks) even then it will still get a small PR of 0.15 (i.e. 1–0.85).

4.3.2.2 How is PageRank calculated?

The PR of each page depends on the PR of the pages pointing to it. But we won't know what PR those pages have until the pages pointing to them have their PR calculated and so on... And when you consider that page links can form circles it seems impossible to do this calculation! But actually it's not that bad. Remember this bit of the Google paper:

> PageRank or $PR(A)$ can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

What that means to us is that we can just go ahead and calculate a page's PR without knowing the final value of the PR of the other pages. That seems strange but, basically, each time we run the calculation we are getting a closer estimate of the final value. So all we need to do is remember the each value we calculate and repeat the calculations lots of times until the numbers stop changing much.

5 Part C: Optional Practice (3h)

In this optional exercise, it is proposed that you create and analyze co-occurrence[2] graphs coming from more complex input structures (i.e., XML files). In this exercise, you should use Spark and GraphFrames to load, transform, and then analyze the major topics on a recently published subset of citation data from the Medical Literature Analysis and Retrieval System Online (MEDLINE).

5.1 Load the MEDLINE citation graph

MEDLINE is a database of academic papers that have been published in journals covering the life sciences and medicine. MEDLINE database can be obtained in the format of multiple XML files. For simplicity, we provide a single XML file (medline_example.xml), located within the `resources` folder of your Java project.

For loading and parsing an XML file in Spark, you are provided with a utility function `loadMajorTopicsSequence` located inside the

[2]More about co-occurrence at: `https://en.wikipedia.org/wiki/Co-occurrence`

`utils.UtilsXML` class of the project. Parsing and loading XML in Spark requires the scala XML parser library and the spark-xml library to be included your maven project. We have already added the following dependencies to the *pom.xml* of your project.

```
<dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-xml</artifactId>
    <version>2.11.0-M4</version>
</dependency>
<dependency>
 <groupId>com.databricks</groupId>
 <artifactId>spark-xml_2.11</artifactId>
 <version>0.3.2</version>
</dependency>
```

For clarity, the provided utility class, obtains the `<MedlineCitation>` nodes from the XML file located at the provided *filepath*, and extracts the content of all `DescritptorName` elements, where the attribute `MajorTopicsYN` equals 'Y'.

The result of this part should be RDDs of sequences of XML nodes, each node in a sequence representing one topic in the citation (i.e., `JavaRDD<Seq<Node>>`).

5.2 Find all combinations of co-occurring pairs of topics within citations

Given the parsed XML data from above, you are now asked to find all co-occurring pairs of topics for each citation. More precisely, you need to look for all combinations of 2 elements within the sequence of nodes in each citation.

Hints: You first need to sort the sequence of topics' nodes (`Seq<Node>`) of one citation (`sortWith(new AbstractFunction2<Node,Node,Object>`), and then get all combinations of 2 elements for a given sequence (`combinations(2)`).

Important note: To avoid using scala XML structures that are not needed in the rest of the exercise, extract the text from the given XML node (`Node`) elements representing the topics (`text()`).

As a result, you should obtain RDDs with one or more pairs of strings representing the topics (`JavaRDD<Tuple2<String,String>>`)

5.3 Count the co-occurrences of the topics among all the citations

For each distinct pair of topics, find the number of how many times these two topics appear together (within the same citation) in the given document.

5.4 Creating the co-occurrence graph

Now, you are asked to create a *GraphFrames* structure representing the co-occurrence graph. Different topics should represent the graph vertices, while the co-occurrence relationships represent the edges. In addition, include the calculated number of co-occurrences on the graph edges.

Hints: Use the MD5 hash values of the topics as a vertex id, and as src and dst of the edges.

5.5 Analyzing the relationships among topics.

Finally, in this exercise, you should analyze how the topics of the given citation document are related with each other.

Hints: You should check the connectivity of the created graph and analyze the size of each connected component in the graph.

Explain shortly what the given numbers represent.