LAB Assignment

This is a hands-on about distributed graphs. We will use Spark GraphX for distributed processing in the Spark ecosystem and, additionally, we will introduce you to the Spark Graph Frames (more information in page 5). GraphX is a component of Spark for graphs and graph-parallel computation, providing a collection of graph algorithms (e.g., Page Rank, Triangle Counting and Connected Components) and some fundamental operations on graphs based on TLAV. Relevantly, GraphX implements an optimized variant of the Pregel API (most popular implementation of the TLAV programming model), which is a popular graph processing system developed by Google in 2010. Next, we provide the environment setup and then the exercises to be solved. Each team must upload the solutions to the exercises to the Learn-SQL platform (look for the corresponding assignment event). Only one team member must submit the solutions (check below for a precise enumeration of what you need to submit) and list all group members in such document. Check the assignment deadline and be sure to meet it. It is a strict deadline!

Setup instructions

For working on the exercises, follow these steps:

1. Download the Java project zip (`SparkGraphXassignment.zip`) provided in LearnSQL.
2. Import the project in you preferred IDE as a maven project (Eclipse, IntelliJ). Manuals for both cases are provided in LearnSQL.
3. Create Run Configurations for running all the exercises. Consult the `Main.java` for the expected input.

Note
When running the code in a Windows environment, you must set up the `HADOOP_COMMON_PATH` to the resource directory where the bin with the `winutils.exe` is located. For example, if the project directory is placed in C, the absolute path should look like this:

```
C:\\SparkGraphXassignment\\src\\main\\resources
```

After this, you are ready to run the exercises.

To deliver

Upload **ONE zip archive** that includes:

- The java project with the solution of Exercises 2, 3, and 4.
- A document of max 4 pages with i) the solution of Exercise 1 and ii) explanation of any specificities and assumptions of your solutions (related to all exercises).

Exercise 1: Getting familiar with GraphX's Pregel API

Acknowledgement. The content of this exercise is based on Professor Eric Lo's material for the course Big Data Analytics.

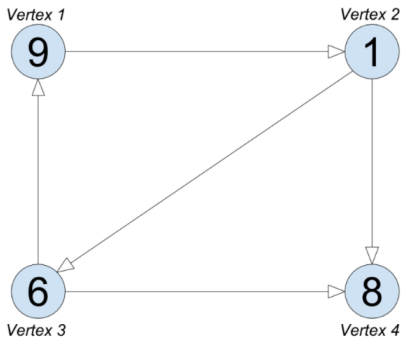
In this exercise we will introduce GraphX's Pregel-style vertex-centric programming model and apply this model to implement some graph algorithms. Pregel-style vertex-centric computation model consists of a sequence of iterations, called supersteps. Assuming an input graph with type G and a message type A, each superstep S follows a GAS (Gather-Apply-Scatter) paradigm:

Gather: also known as *merge*, receives and reads messages that are sent to a node v from the previous superstep $S-1$. This function must be commutative and associative.

Apply: also known as *vertex program*, applies a user-defined function f to each vertex in parallel; meaning that f specifies the behavior of a single vertex v at a particular superstep S. On the first iteration, the vertex program is invoked on all vertices and the pre-defined message is passed. On subsequent iterations, the vertex program is only invoked on those vertices that receive messages.

Scatter: also known as *send message*, may send messages to other vertices, such that those vertices will receive the messages in the next superstep $S+1$.

Now let's turn to the code in class `Exercise 1.java` in the method `max-Value`. The first step is to create the following graph:



The vertices store the value we aim to maximize, while the edges contain an additional *dummy* attribute. Now look at the implementation of the three functions sent to *Pregel*:

VProg (corresponding to the *Apply* phase): in the case of the first superstep the vertex value, otherwise sends the maximum of the vertex value and the received message.

sendMsg (corresponding to the *Scatter* phase): if the destination vertex value is smaller than the current value, we will send the current value to it. Otherwise, we do not send anything.

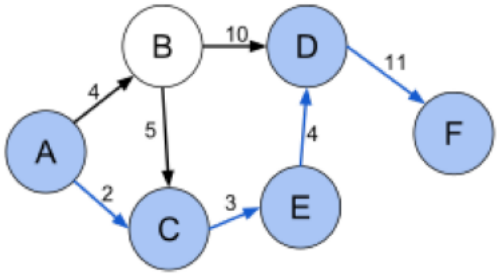
merge (corresponding to the *Gather* phase): in this case, the logic for the function is the same as the vertex program.

Finally, we have the call to the *Pregel* framework. The parameters are as follows:

- `initialMsg` is the message that all vertices will receive at the start of superstep 0. Here we use `Int.MaxValue` just for the purpose of identifying superstep 0 (as we shall see later).
- `maxIter` indicates the maximum number of iterations (i.e., supersteps). Here we set it as `Int.MaxValue` as convergence is guaranteed.
- `activeDir` refers to the edge direction in which to send the message. Take the graph above with four vertices as an example, assume Vertex 2 becomes active and wants to send a message to its neighbors. If:
  - `EdgeDirection.Out` is set, the message is sent to Vertex 3 and Vertex 4 (i.e., along outgoing direction).
  - `EdgeDirection.In` is set, the message is sent to Vertex 1 (i.e., along ingoing direction).
  - `EdgeDirection.Either` is set, the message is sent to all the other three vertices.
  - `EdgeDirection.Both` is set, the message is sent to Vertex 1/Vertex 3/Vertex 4 only when Vertex 1/Vertex 3/Vertex 4 are also active.
- The three next parameters are references to each of the three previously described functions.
- `evidence`, the definition of the class being passed as message. Note the usage of `scala.reflect.ClassTag$.MODULE$.apply(Integer.class)`, which uses Scala's API.

For this exercise, explain **each superstep that will be performed for the given graph**. In particular, **for each superstep** you should specify:

- The initial state of each node and the messages that it receives.
- The result of calculations performed on each node and its resulting state.
- The message(s) to be sent (if any) to other nodes in the next superstep.



Exercise 2: Computing shortest paths using Pregel

In this exercise, we ask you to implement, (in class `Exercise_2`), the computation of the shortest paths from a source in the provided graph. We provide you with the creation of a graph as depicted in the following figure:

You will also see that the classes corresponding to the functions `VProg`, `sendMsg` and merge has already been defined for you. You just need to implement their logic.

Precisely, we ask you to compute the cost for the shortest paths from node 1 to the rest of the network. You should obtain the following output:

```
Minimum cost to get from A to A is 0
Minimum cost to get from A to B is 4
Minimum cost to get from A to C is 2
Minimum cost to get from A to D is 9
Minimum cost to get from A to E is 5
Minimum cost to get from A to F is 20
```

Excercise 3: Extending shortest path's computation

You might have realized that the previous computation of shortest paths was a bit limited, as we could know the minimum cost to get from A to any other node but not the actual path. To this end, in this exercise you are asked to extend such implementation (in class `Exercise_3`) to include the minimum path.

Note that this entails considering additional data structures in the graph and properly update them in *Pregel*. Given the graph from the previous exercise you should obtain the following output:

```
Minimum cost to get from A to A is [A] with cost 0
Minimum cost to get from A to B is [A,B] with cost 4
Minimum cost to get from A to C is [A,C] with cost 2
Minimum cost to get from A to D is [A,C,E,D] with cost 9
Minimum cost to get from A to E is [A,C,E] with cost 5
Minimum cost to get from A to F is [A,C,E,D,F] with cost 20
```

Excercise 4: Spark Graph Frames

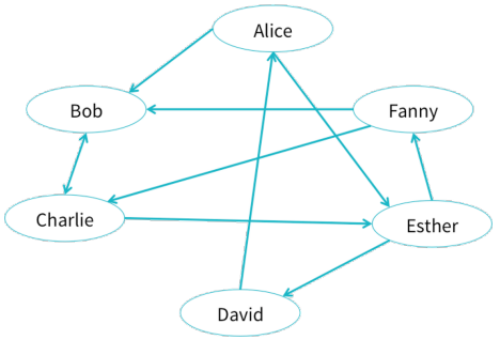
In this exercise we will use Spark GraphFrames for distributed graph processing in the Spark ecosystem. GraphFrames is a package for Apache Spark that provides DataFrame-based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX seen in the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

Warmup excercise (with solution provided)

In the Java project, we provide the GraphFrames library (as it is not yet published in Maven Central), make sure that the locally specified dependency in pom.xml is working. Otherwise, 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.jar` in the `lib` folder of the Java project.

In `Exercise_4_warmup.java` we provide you with some code to introduce yourself to the 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 that 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 other arbitrary 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.

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.

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.

Entity Link 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. Appendix A: About PageRank provides more important detail in order to understand how PageRank works and what are its main parameters.

Therefore, in this exercise, you are asked to use PageRank from GraphFrames library to provide the 10 most relevant articles from the provided Wikipedia dataset. Importantly, you are asked to optimally parametrize the PageRank algorithm, providing the values for the **damping factor**[2] and the **maximum number of iterations**[3] see Appendix A: About PageRank).

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

[2] In Spark GraphFrames, damping factor is set through the resetProbability method which sets the value of $(1 - dampingFactor)$. For example, for $d = 0.85$, the resetProbability should be called with 0.15.

[3] In Spark GraphFrames, maximum number of iterations is set through maxIter method.

Appendix A: About PageRank

In short PageRank is a "vote", given by all the other pages on the Web, about how important a page is. A click to a link that leads to a page counts as a vote of support. If there is no click 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. *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*:

$PR(A) = (1-d) + dx(PR(T_1)/C(T_1) + \ldots + PR(T_n) = C(T_n))$

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."*

Let us detail each of the components of the formula:

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 its 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. $dx(\ldots)$ - 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 the damping factor (e.g., 0.85). Damping factor practically represents the probability that the "clicking" on the link that leads to the page will continue.

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 also means that if a page has no links to it (no backlinks) even then it will still get a small PR of 1–d (e.g., 0.15).

How is PageRank calculated?

The PR of each page depends on the PR of the pages pointing to it. But we will not know what PR those pages have until the pages pointing to them have their PR calculated and so on. To avoid potential loops:

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 from the local PR we, at each iteration, spread the PR one hop. We will repeatedly iterate propagating such values until they converge or we reach a predefined maximum number of iterations.