Node Importance in RDF Graph

Masterproject - WS2016/2017

Muazzam Ali
Syed Shahzeb Hassan Kirmani

Period of development
November 2016 - March 2017

Reviewer
Prof. Dr. Georg Lausen

Supervisor
Martin Przyjaciel-Zablocki

Abstract

Over the years, the popularity of the RDF model being used to represent data has increased significantly. This is mainly because it is able to capture and then present the information contained in the data in a very structured way. Therefore most of the big datasets are represented in the form of an RDF graph in order to be visualized and analyzed in a reasonable way. Since the datasets are so huge, the problem arises that what is a good starting point to consider when browsing the RDF graph. This is where the importance of each node in the graph has to be determined in order to separate the important one from the others. For determining this importance, we have used the PageRank algorithm and an algorithm based on a metric called centrality, where centrality is divided further into different categories. These were performed on different datasets with some changing parameters. The exact methodology behind each of the methods is explained and the results are then discussed.

1. Introduction

This project is an extension of the 'Spark RDF Analyzer' framework which was implemented by the previous group last year. The framework is used to analyze and extract useful properties from RDF graphs and is graph independent. Along with this, the other extension to the application is the creation of an 'RDF browser' which would be used to traverse the graph in an ordered way. The problem then becomes apparent that what should be the starting points for the user to traverse the graph from. This is where our work becomes very important and relevant. We use different metrics to determine the importance of the nodes in the graph. Without finding these important concepts in the large RDF datasets, the nodes in the graph are basically unorganized and unordered.

The first method of ranking the node is based on the concept of centrality which states that the more central a node is, the easier it is to reach the rest of the graph. Centrality highlights the 'strategic' nodes by defining their importance. A simple example is that when a person wishes to explore a city, the person does not know where to go and start from. The solution is to start from a central place in the city from where the person has the largest number of options in order to explore the whole city. The types of centrality measures we have used to determine the important nodes are degree and closeness centrality.

Secondly we have used the PageRank algorithm to rank the nodes in the graph. The algorithm is based on what was originally described by Larry Page of Google. PageRank works by counting the number and quality of edges to the node to determine how important it is.

2. Centrality

There were 2 main measures of centrality which were used to calculate the node importance; degree centrality and closeness centrality. The measures each identify a different node as the ‘most central’ but each of them answers different questions.

2.1 Degree Centrality

The degree of a resource is the number of resources adjacent to it. The degree centrality considers nodes with higher degrees as more central, highlighting the local popularity of a node in its neighborhood. Degree centrality is divided further into 2 categories:

1) In-Degree: For a particular node in the graph, it is a count of the number of paths of predicates ending at that node. It can be described as the resources which support/influence a particular node. This is calculated by executing the following Sark SQL query:

```
SELECT COUNT(subject) FROM Graph WHERE object = '" + node
+ "'
```

2) Out-Degree: For a particular node in the graph, it is a measure of the number of paths of predicates starting from that node. It can be described as highlighting the resources supported by a particular node.It is calculated by executing the following Sark SQL query:

```
SELECT COUNT(object) FROM Graph WHERE subject='" + node
+ "'
```

Simply determining the node with the highest in or out degree was not informative enough as different datasets would give us different results. For example in the case of the sib200 dataset which is a relatively smaller dataset, the nodes with highest in degrees were determined to be either “Firefox” or “Chrome” with the respective predicate being ‘sib:browser’. But these nodes had

0 out degree value so they could not really be considered as a useful start node for traversing the RDF graph.

3) Combination of in-degree and out-degree

Our approach was to calculate 4 different values because it is not necessary that the node which has highest in-degree also has highest outdegree. So if a node has highest out-degree we also calculate its in-degree and vice versa.

The Spark SQL query for calculating the node with the highest out-degree is as follows:

```
SELECT subject,COUNT(object) AS OutdegreeCount FROM
Graph GROUP BY subject ORDER BY OutdegreeCount DESC
LIMIT 1
```

For the resultant node which was returned as a result, we then calculated the in-degree for it using the query which was specified earlier.

We then determine the node with the highest in-degree and then calculate the out-degree for that particular node.

Finally the following calculations are performed in the specified order:

1) Sum of the out-degree and in-degree of the highest out-degree node
2) Sum of the out-degree and in-degree of the highest in-degree node
3) If the sum in step 1 is greater than that in step 2, we return the highest out-degree node as the start node.
4) If the sum in step 1 is less than that in step 2, we return the highest in-degree node as the start node.

By combining both degree values for the particular node, we get more useful suggestions for a start node. This is when compared to the ‘browser’ node suggestion which was given when only one of the degree types was considered.

2.2 Closeness centrality

The second centrality metric which we used to determine the start node for the RDF graph is closeness centrality. The closeness centrality of a resource represents its capacity to join (and to be reached by) any resource in a network.

For a particular node k, the closeness centrality is the inverse sum of its shortest distances to all the other nodes in the graph.
Where n is the number of nodes and d(k,i) the length of a shortest path from k to another node i. Shown by the following formula:

$$C_C(k) = \left[\sum_{i=1}^{n} d(k,i) \right]^{-1}$$

It should be noted that the interpretation of whether a high or less value calculated using the formula means a high or low closeness centrality respectively for a particular node depends on the type of graph. That is undirected and directed graphs will give different results based on what data is being represented in the graph.

Since closeness centrality is based on the shortest paths between the source node and all the other nodes in the graph, we first go into detail about shortest path algorithms in large graphs.

3. Shortest path

To calculate the shortest path between a given pair of nodes, there are 2 approaches: single source shortest path (SSSP) and all pair shortest path (APSP). SSSP involves finding the shortest path from a single source node to all other destination nodes. APSP involves finding the shortest path between all the pairs of nodes in the graph but due to this it is significantly slower in runtime as compared to the single source algorithm. The best case running time for APSP algorithm is O(vertices ^ 2) considering our graph is an unweighted and directed graph.

Our approach was to first go about implementing the SSSP algorithm and later on enhance it to support APSP. Since the graph we were working on is an unweighted directed graph. The approach we took for implementing SSSP was using a Breadth First Search (BFS) through Spark Map and Reduce functions.

3.1 Implementation of SSSP

For implementing the SSSP algorithm we first wanted to organize our data in terms of id's and relations. This helped us gain more efficiency in terms of using nodes with large texts and special characters in terms of ids. To achieve this we did two things.

3.1.1 Unique nodes parquet generation

The first thing we did was to convert our data into a parquet file called *UniqueNodes.* We did this by selecting all distinct nodes from the graph and assigning them unique ids. This helped us in getting rid of nodes having long texts. For e.g.

"status across Europe. In Germany, the album was certified 4x Platinum, with more than 2 million units sold, making it one of the"

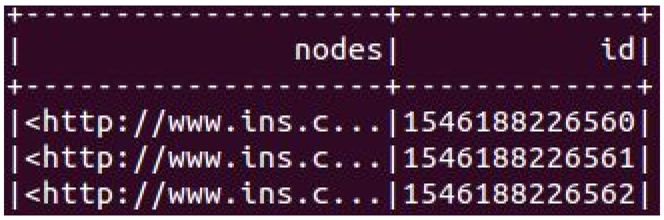
We save the resulting data in a parquet file called **UniqueNodes.parquet**. Which helps us easy retrieval of data for all algorithms we will be applying in future.

The following query is used to generate the UniqueNodes for the graph.

```
SELECT DISTINCT  a.nodes FROM
(SELECT subject as nodes from Graph  UNION ALL
 SELECT object as nodes FROM Graph) a)
.withColumn("id", functions.monotonically_increasing_id()
```

The results are collected in a Dataframe in the following format:

-



```
+--------------------+-------------+
|               nodes|           id|
+--------------------+-------------+
|<http://www.ins.c...|1546188226560|
|<http://www.ins.c...|1546188226561|
|<http://www.ins.c...|1546188226562|
```

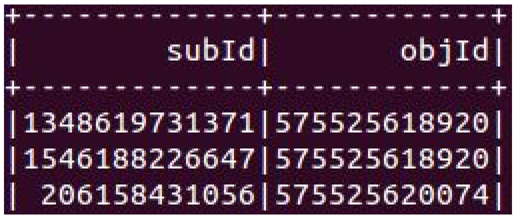
3.1.2 Relations parquet generation

After creating the UniqueNodes we also needed one more measure to keep track of the relations between the nodes in the graph i.e which object belongs to which subject or vice versa. For this we went about generating a relations.parquet file to keep track of all the relations with respect to the node ids we generated in the previous step.

The query is as follows:

```
SELECT unSub.id as subId,unObj.id as objId FROM Graph g
INNER JOIN UniqueNodes unSub ON unSub.nodes=g.subject
INNER JOIN UniqueNodes unObj ON unObj.nodes=g.object
WHERE g.subject != g.object
```

The results are collected in a Dataframe in the following format:



```
+-------------+------------+
|        subId|       objId|
+-------------+------------+
|1348619731371|575525618920|
|1546188226647|575525618920|
| 206158431056|575525620074|
```

3.2 Single Source Shortest Path Algorithm

The core algorithm for SSSP is implemented in **SSSP.java** which is run through the **DataFramePartitionLooper.java(DFPL)**. DFPL is responsible for preparing the data structures required to run SSSP. The first thing that DFPL does is create an adjacency matrix for the provided graph. It does this by accepting the relations dataframe (explained in previous section) and outputting an adjacencyMatrix for the graph in the following format.

Tuple4<List<Long>, Integer, Integer, Integer>

Each node has a specific id as key and for each key we maintain a data structure called ‘scala tuple’ which can contain a miscellaneous collection of elements. In our case we have 4 elements so it is the following form.

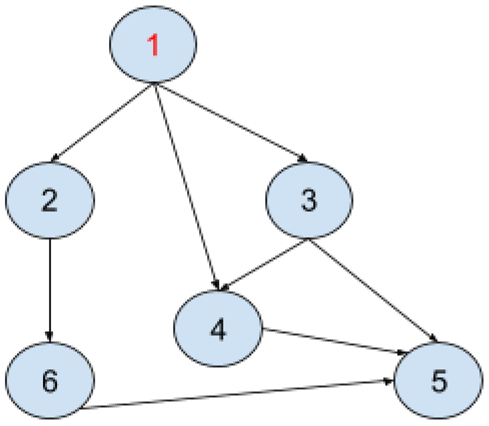
1) The first element is a list containing the ids of the immediate neighbours of the particular node.
2) The second integer type element represents the distance from the source node.
3) The third integer type element represents the current status of what action has to be taken regarding the node during traversing. Only 3 values (0, 1, and 2) are used and each of them relates to the following criteria.

- 0 means that the node still has to be visited
- 1 means that the node will be expanded next
- 2 means node has already been expanded

4) The fourth integer type element represents the number of shortest paths between the source node and current node being considered.

3.2.1 Example

The following graph will now be used as a reference for the map and reduce phases. Taking node 1 as the source node.



Map phase 1

The map phase of the algorithm consists of multiple steps.

Step 1

For better understanding, here we are only showing the row with respect to node/key 1. But in actual case, these will be the values of row with node 1 contained in the adjacency matrix.

There are 3 values for a node **status** in the data structure.

0 = This node is not reached yet.
1 = This node needs to be expanded next.
2 = This node has already been expanded.

Node ID	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	1	1

Explanation:

- Node 1 in the graph has nodes 2, 3, 4 as direct neighbours.
- Distance of node 1 from itself is 0.
- Status is 1 as node 1 has to be expanded.
- For this case, the no. of shortest paths is assigned value of 1 by default.

Step 2

Here we're showing the complete adjacency matrix and how it will be updated.

Node ID	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	2	1
2	Null	1	1	1
3	Null	1	1	1

4	Null	1	1	1
3	[4,5]	0	0	1
2	[6]	0	0	1
6	[5]	0	0	1
4	Null	0	0	1
5	Null	0	0	1

Explanation:

- We expanded the direct neighbors of Node 1 i.e Node 2,3,4 and changed the status of node 1 to 2 since it has been now expanded.
- Node 2,3,4 have not yet been expanded so their status is 1.
- The distance is incremented by 1. This is done whenever we expand nodes.
- The number of shortest path from node 1 to node 2,3,4 remains 1.

Reduce phase

After achieving the result from step 2 of map phase, we simply reduce the data with the following rules. We will be considering the example of NodeID 2 to explain how reduce happens for each column.

1. The first thing we check is the status of the two nodes being reduced. If one node has a status of 2 while the other has 1. This means the node with status 2 has already been expanded before, hence we directly consider the one with status 2 with all it’s values. This ensures shortest paths.

2. Now suppose if one node has status 1 while the other has 0. This means we’re facing this node for the first time. Hence we consider the listOfNeighbors of the node with non null values. We consider the distance which is greater among the two. We keep the No. of shortest paths as same.

After reducing the result from the step 2 of map phase, we get the following data structure:

Node ID / key	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	2	1
2	[6]	1	1	1
3	[4,5]	1	1	1
4	[5]	1	1	1
5	Null	0	0	1
6	[5]	0	0	1

Map Phase 2
Step 1

Node id	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	2	1
2	[6]	1	2	1
6	Null	2	1	1

Explanation

- The status of node 2 is 1 so it is expanded next and its status is changed to 2.
- Like the last time, we increment the distance by one for the expanded row which gives us a distance of 2 now from the sourceNode 1.

Step 2

Node id	List of	Distance	Status	No. of

	neighbours			shortest paths
1	[2,3,4]	0	2	1
2	[6]	1	2	1
6	Null	2	1	1
3	[4,5]	1	2	1
4	Null	2	1	1
5	Null	2	1	1

Explanation
- The node 3 has status 1 so it is expanded and its status is changed to 2.
- The neighbours of node 3 are node 4 and 5; they are at a distance of 2 from the source node 1 and their status is set as 1.

Step 3

Node id	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	2	1
2	[6]	1	2	1
6	Null	2	1	1
3	[4,5]	1	2	1
4	Null	2	1	1
5	Null	2	1	1
4	[5]	1	2	1
5	Null	0	0	1
6	[5]	0	0	1
5	Null	2	1	1

Explanation

- The node 4 has status 1 so it is expanded and its status is changed to 2.
- The neighbour of node 4 is node 5; the distance from the source node is 2 and the status is set as 1.
- It should be noticed that there are 2 entries for node 5 with distance as 2 meaning that there are 2 shortest paths between node and source node 1 so in the next phase, the value for the no. of shortest paths for node 5 will be increased to 2 (see entry with * in following table).

Reduce phase 2

Node id	List of neighbours	Distance	Status	No. of shortest paths
1	[2,3,4]	0	2	1
2	[6]	1	2	1
6	[5]	2	1	1
3	[4,5]	1	2	1
4	[5]	1	2	1
5	Null	2	1	2 *

After this step, node 6 will be expanded based on its status of 1 and so on. The purpose of showing the computation steps until here were to demonstrate the main steps involved in the algorithm. And in the previous reduce phase, the specific scenario of a node having more than 1 shortest path was also shown.

3.2.2 Experiments and Results

The algorithm was used to calculate the closeness centrality for some random nodes and the dataset used was sib200. There were 3 main implementations:

1) Closeness centrality against whole graph
2) Closeness centrality for 3 hop neighbourhood
3) Closeness centrality for 2 hop neighbourhood

The results are shown in the following table:

	Closeness centrality value		
Node	**Whole Graph**	**3 hop neighbourhood**	**2 hop neighbourhood**
sibpo:po12549	0.016667	0.016667	0.016667
sibu:u7	0.000000625	0.000000629	0.000001343
sibc:co27859	0.005587	0.005587	0.011765
sibu:u154	0.000000614	0.00000062	0.000001467
sibpha:pa1145	0.000000473	0.000001289	0.000014185
sibfr:fr1808	0.000000533	0.000000834	0.000003356
sibfr:fr2870	0.000000529	0.000000884	0.000003705
sibc:co146175	0.004651	0.004651	0.012658

Analysis
If we consider the first node 'po12549' whose type is post, over 2 hops it has a value of 0.016667 which is the largest as compared to the others. This means it has the lowest closeness centrality value. The explanation for this is that the final value calculated is an inverse of the sum of node distances according to the formula specified earlier. So for this node, the total sum being less means that it is further away from many of the nodes of the graph and hence it is the least central in terms of closeness from among the other nodes.

Consequently if we now consider the node 'u7' whose type is user, over 2 hops it has the lowest value meaning that it has the highest closeness centrality among the other nodes. It has a large value for the sum of the other node distances which shows that the 'u7' node is near to many of the other nodes of the graph hence it is most central in terms of closeness. Being connected and nearer to more nodes means that this node has the capacity to receive information flowing through the graph very quickly. This also makes sense since a user in a network can be considered to have many connections to diverse entities as compared to a single post.

In following table, the percentage change in values of centrality between the different hop methods is done:

Node	% difference between 3 hop and whole graph	% difference between 2 hop and whole graph
sibpo:po12549	0.00	0.00
sibu:u7	0.64	53.46
sibc:co27859	0.00	52.51
sibu:u154	0.97	58.15
sibpha:pa1145	63.30	96.67
sibfr:fr1808	36.09	84.12
sibfr:fr2870	40.16	85.72
sibc:co146175	0.00	63.26
	Avg: 17.64	**Avg: 61.74**

Analysis
It is observed that for the node ‘sibpo:po12549’, the closeness centrality value calculated over 2 hops did not change when calculated over 3 hops and whole graph. Meaning that all the neighbours for that node were covered over the 2 hops. For two of the ‘sibc’ nodes, there was no change in value of centrality when going from 3 hops to whole graph.

For the rest of the nodes, there was some change in value of centrality from 3 hops to whole graph but the change was not too significant. In fact for the two ‘sibu’ nodes representing a user, there was only a 0.64 and 0.94 % change even though they would be thought of as nodes with a lot of connections and hence paths in the graph.

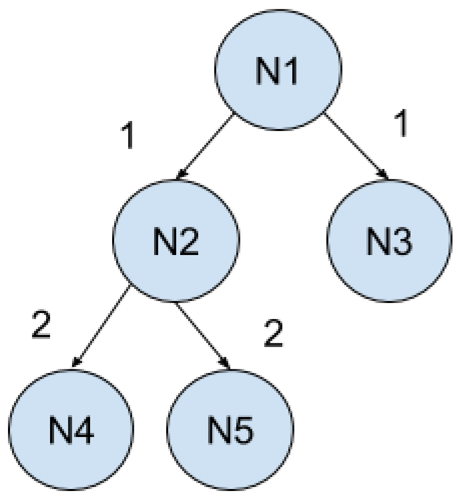
The maximum percentage change when comparing 3 hops against whole graph was observed for the ‘sibfr’ nodes and a ‘sibpa’ node. Overall when the percentage change in centrality for 3 hops against whole graph for all the eight nodes in table was considered by calculating an average, the value came came out to be 17.64 %. The significance of this value was made clear when the average percentage change in centrality for 2 hops against whole graph was calculated to be 61.75 %.

What this means is that in general, calculating the closeness centrality over 3 hops for the nodes was sufficient. And the values obtained are considered to be accurate enough in comparison to the values with respect to the whole graph. The difference in values between 2 hops and whole graph then also gives credence to obtaining more accurate centrality values over 3 hops.

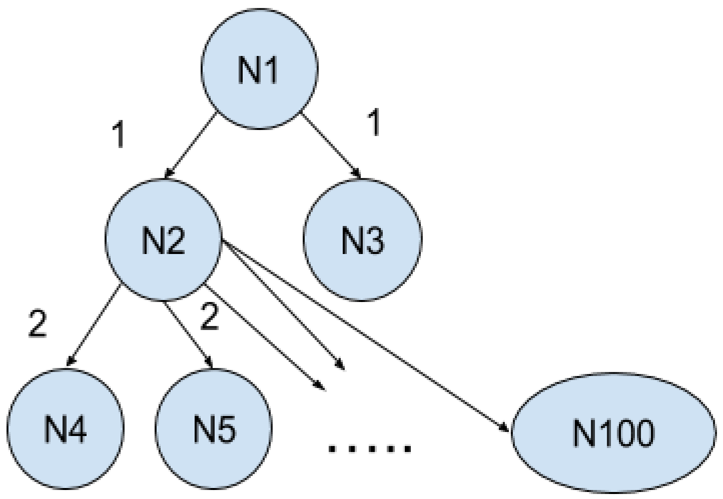
3.3 Important Nodes Based on Closeness Centrality

In order to enable the users to have an alternative way to decide a starting node, we found out the nodes with maximum closeness centralities. As explained earlier that the closeness centrality of a node k is 1 divided by the sum of k’s distance from all the other nodes.

The definition gives us a hint that the most central nodes in the graph will be the one which are mostly connected throughout the graph. Suppose we have the following two graphs i.e graph 1 and graph 2.



Graph 1



Graph 2

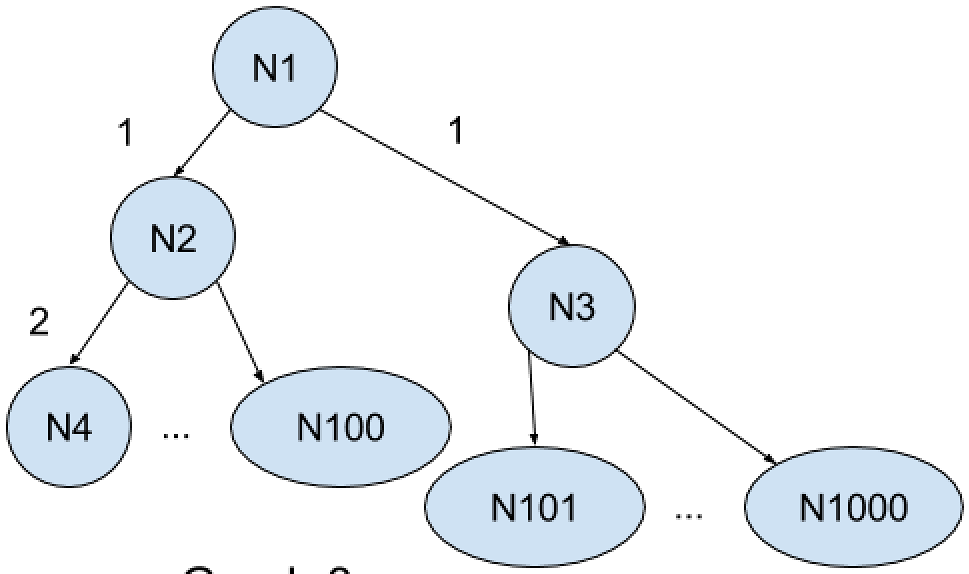
If we take a look at graph 1. The sum for the closeness centrality for N1 will be calculated in the following manner i.e the shortest path from N1 to all other nodes.
N1 to N2 = 1
N1 to N3 = 1
N1 to N4 = 1 + 2 = 3
N1 to N5 = 1 + 2 = 3

This gives us a total of 8. Hence the centrality for N1 in graph 1 will be 1/8 = 0.25.

In Graph 2, we'll get the same sum from until N5 but this time we've 95 more Nodes that needs to be added. This means we'll calculate the distance of N1 to N4 = 3, N1 to N5 = 3 …. N1 to N100 = 3. This will give us a total sum of 201 and a centrality of 1/201 = 0.005.

What actually causes the centrality for N1 in graph 2 to increase is node N2 which has a huge number of outdegree nodes.

Now suppose if we have a case where node N3 have 1000 out degree nodes.



Graph 3

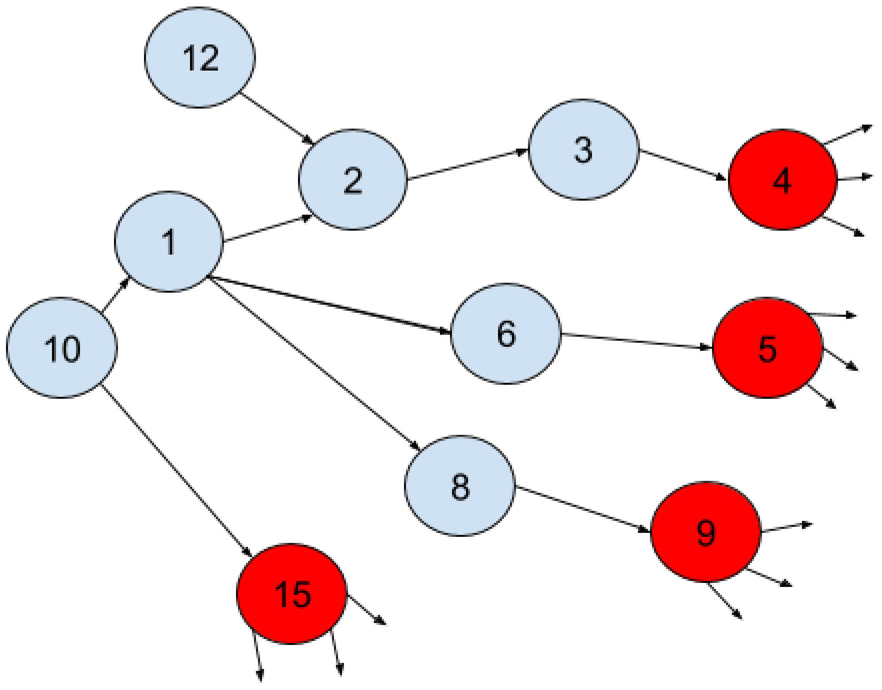
In this case the centrality for N1 will increase a lot. This means that the more a node is connected to other nodes with higher out degree the more central it is. Hence we take the same approach and implemented an algorithm that performs the following steps.

Step 1
We select the top 10 nodes in the graph with respect to the highest outdegree. We use the following query to return the nodes.

```
SELECT subject,COUNT(object) as OutDegreeCount
FROM Graph Group BY subject ORDER BY OutDegreeCount DESC LIMIT
10
```

The nodes are then passed to an algorithm that calculates the distance of these nodes from the farthest nodes to which they are connected to as objects.

The algorithm was focused and written using DataFrames purely. Let’s take an example graph and show how the algorithm works.



Graph 4

If you see graph 4, the nodes indicated in red are the ones with highest out degrees. Now we'll go backwards step by step to find the distances of the red nodes to the other nodes. We do this using the following steps.

Step 1:
Get subjects of objects. Initially, our objects are the red nodes.

subject	object	distance
3	4	1
6	5	1
8	9	1
10	15	1

Step 2:
Get subjects of subjects retrieved in step 1 and increment the distance by 1.

subject	object	distance
2	3	2
1	6	2
1	8	2

Step 3:

1. Join subject of Step 1 DataFrame with Object of Step 2 DataFrame.
2. Keep the distances of the step 2 DataFrame.
3. Then union the Step 1 DataFrame to the final result.

DataFrame Step 1

subject	object	distance
3	4	1
6	5	1
8	9	1
10	15	1

DataFrame Step 2

subject	object	distance
2	3	2
1	6	2
1	8	2

Joined DataFrame

object	subject	distance
4	2	2
5	1	2
9	1	2

Unioned DataFrame

object	subject	distance
4	2	2
5	1	2
9	1	2
4	3	1
5	6	1
9	8	1
15	10	1

Now we'll go back to step 2 and find the subjects of node 1,2 or in other words find the subject of nodes with max distance in the Unioned DataFrame. Than we repeat step 3 with the result and the last Unioned DataFrame giving us a DataFrame containing nodes whose distance is 3 from the red nodes.

Cycles:
There will be cycles in graph which could cause this algorithm to work in an infinite loop. We take care of the cycles in the graph by checking if subject and object column of the Unioned DataFrame has multiple entries of the same values. If yes we just discard the duplicate entries.

Break Condition:
The algorithm stops when the number of rows in the unioned DataFrame from the last step and current step remains the same. This means that there are no further nodes to expand as subjects.

Final Result:
The final result returned by this algorithm will look something like this.

object	subject	distance
4	10	4
9	10	3
5	10	3
4	12	3
4	1	3
4	2	2
5	1	2
9	1	2
4	3	1
5	6	1
9	8	1
15	10	1

Once we receive the result, we can simply group the nodes by object to find out the number of intersections. For example in this table, the node with highest intersection will be 10, in other words, the node which has reach to maximum number of red nodes is 10. While the second node which has reach to maximum number of nodes is 1.

We simply pick the top 10 nodes with highest intersections and feed them into the SSSP algorithm (explained above) to find the centralities of the collected nodes. This gives us the nodes with the highest closeness centralities in the graph. The result is saved in a parquet file as preprocessed data and shown as a recommended starting node in the system.

4. PageRank

Pagerank was another method used to calculate the importance of nodes in the RDF graph based on the incoming and outgoing links from the node. Pagerank specifically considers a node more important if it has incoming links from many other nodes. The computation of pagerank is done according to the following formula [7]:

PR(A) = (1-d) + d (PR(B1)/C(B1) + ... + PR(Bn)/C(Bn))

Where:
PR(A) is PageRank of node A
PR(Bi) is PageRank of nodes Bi which link to node A
C(Bi) is number of outgoing links from node Bi
d is damping factor with range of [0,1] and set to 0.85 in this case

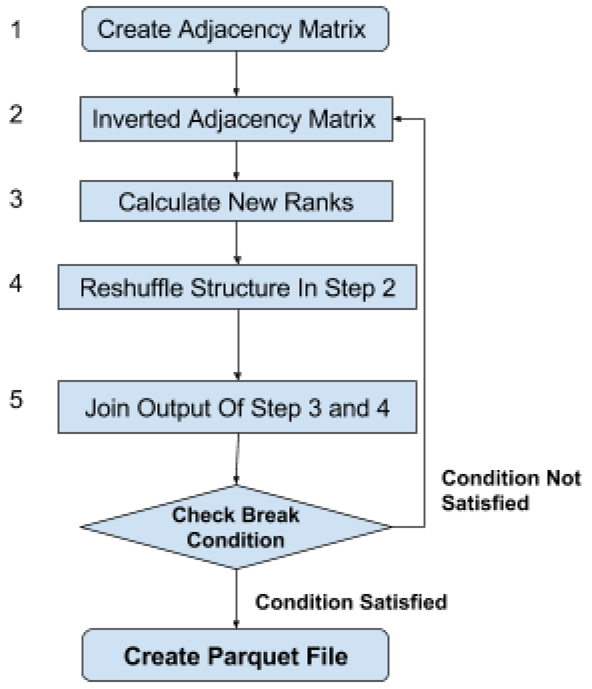
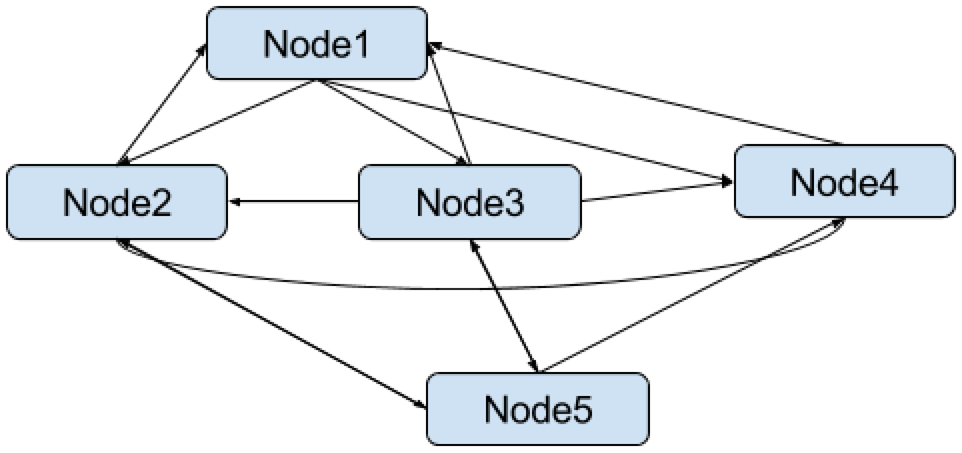


Figure: Flowchart showing the sequence of steps to calculate pageranks

Implementation

We'll use the following graph to explain the algorithm.



Step 1: Creation of adjacency matrix

The structure is in the following format:

Node, [list of nodes being pointed to], 1/no. of outgoing links, rank

In the initial step, the rank of each node is set as the number of outgoing links of that particular node. This is shown in the following figure:

key	neighbors	Pj or 1/n	rank
1	[2,3,4]	⅓ = 0.3333	3
2	[1,5]	½ = 0.5	2
3	[1,2,4,5]	¼ = 0.25	4
4	[1,2]	½ = 0.5	2
5	[2,3,4]	⅓ = 0.3333	3

Step 2: Calculate Inverted Adjacency Matrix
Since pageRank calculates the nodes importance by considering all nodes providing their importances as in-degrees to a particular node. We calculate the inverted adjacency matrix to figure out which nodes a pointing to a particular node.

Step 2.1 : Reshuffler operation
The reshuffler maps the keys to the neighbors column while flatMapping the neighbors column to become keys. This helps us identify which nodes are being pointed out by which nodes. This is shown in the following figure:

Note that this step is only shown for nodes 1, 4 for the sake of simplicity.

key	neighbors	Pj or 1/n	rank
2	1	⅓ = 0.3333	3
3	1	⅓ = 0.3333	3
4	1	⅓ = 0.3333	3
1	4	½ = 0.5	2
2	4	½ = 0.5	2

Step 2.2: Reduce phase
The reduce phase is performed on the output of step 2.1 which gives us the final result in the following format.

key	neighbors	Pj or 1/n	rank
2	[1,4]	[0.3333,0.5]	[3,2]
3	[1]	[0.3333]	[3]
4	[1]	[0.3333]	[3]
1	[2]	[0.5]	[2]

Note in the above diagram. It means node 1,4 are pointing to node 2.

Step 3: Calculate new ranks
Use the pagerank formula to calculate the new rank for each node for example the calculation is applied to values for node 1 in step 2 as follows:

NewRank = 0.85* ((0.50*2) + (0.25*4) + (0.50*2) + 0.15

The node and its respective rank are then combined in the following structure:
Node, NewRank

Step 4: Reshuffle the structure in step 2
The following structure was obtained as output of step 2:
node1,([node2,node3,node4],[0.50,0.25,0.50],[2,4,2])

This is now reshuffled the same way as in step 2.1 by making the node's neighbours as the key against the actual node and shown as follows:

Node2 , Node1 , 0.5, 2
Node3 , Node1 , 0.25, 4
Node4 , Node1, 0.5, 2

Step 5: Join output of step 3 and 4

Node2 , Node1 , 0.5	***Node2, NewRank1***
Node3 , Node1 , 0.25	***Node3, NewRank2***
Node4 , Node1, 0.5	***Node4, NewRank3***

Node2 , Node1, 0.5, NewRank1
Node3 , Node1, 0.25, NewRank2
Node4 , Node1, 0.5, NewRank3

Step 6: Goto Step 2
Reduce step is performed on the output of step 7 and the following structure is obtained:

Node1 , [Node2,Node3,Node4], [0.5,0.25,0.5], [NewRank1, NewRank2, NewRank3]

Now the new ranks are calculated again as demonstrated before in step 3 and the procedure is continued.

When this algorithm is run for a particular graph, the results which include the nodes and their importance in form of the pagerank value, at the end are stored in a parquet file.

Break Condition:
In order for the algorithm to decide when to stop. We implemented a break condition which calculates the percentage change in the total sum of the ranks from the previous step. The threshold can be set manually by the user. By default we set the threshold to 5%. This means if there is a less than 5% change in the ranks from the previous step the algorithm stops, organizes the data and provides us with a pageRank parquet file.

The break condition is calculated using the following formula:

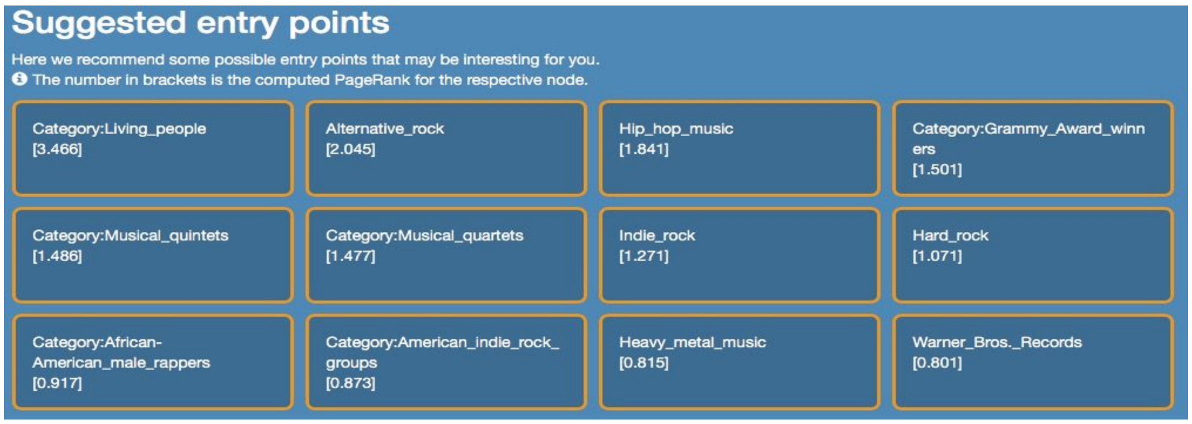
```
double decrease = lastscore - score;
double percentageDecrease = (decrease/lastscore) * 100;
```

Where the ‘percentageDecrease’ value is checked against our defined threshold to decide whether to continue running or stop.

5. Merging with RDF browser

After running the pagerank and closeness algorithms on the graph of a particular dataset, the final output was in the form of a number of top nodes based on their importance values. This data then formed the base for a functionality of the RDF browser which was developed side by side along with our implementation. The purpose of the RDF browser was to enable the user to browse a RDF graph representing a large dataset by traversing it through a click-based interface. One of the challenges of this implementation was to provide a starting/entry point for the user to start traversing the graph from.

This is where our work was used in that the most important nodes computed by each algorithm are used to provide starting points for a particular graph. The computation is not done at runtime instead the nodes and their importance are precomputed and the top results are then stored in a parquet file in the Hadoop Distributed File System (HDFS). This data is then accessed by the application and the results are shown below in figure a, where the top twelve entry points for the graph based on the pagerank values are displayed to the user.



Suggested entry points

Here we recommend some possible entry points that may be interesting for you.
The number in brackets is the computed PageRank for the respective node.

Category:Living_people [3.466]	Alternative_rock [2.045]	Hip_hop_music [1.841]	Category:Grammy_Award_winners [1.501]
Category:Musical_quintets [1.486]	Category:Musical_quartets [1.477]	Indie_rock [1.271]	Hard_rock [1.071]
Category:African-American_male_rappers [0.917]	Category:American_indie_rock_groups [0.873]	Heavy_metal_music [0.815]	Warner_Bros._Records [0.801]

Figure a

6. Summary and Conclusion

The main motivation behind this project was to determine a series of starting nodes for an RDF graph based on their importance. We have achieved this by implementing algorithms based on the concepts of centrality and pagerank. For centrality, the two measures used were degree centrality and closeness centrality with closeness the more intuitive of the two. To calculate closeness centrality, the approach was to calculate the single source shortest path using breadth first search with spark map and reduce functions.

The second approach was based on the knowledge of a node connected to other nodes with higher out degree being more central hence those nodes could be pinpointed as have more importance. The other main algorithm was pagerank which was also implemented using mapreduce.

Some previous work regarding calculating centrality metrics was done by Guillaume Ereteo and co. [1][2] by defining SPARQL extensions. The was done mainly for analysis of social networks. The metrics were computed for specific predicates meaning that the knowledge of the dataset beforehand was needed whereas in our case, we calculated the centralities irrespective of the predicate so our implementation was graph independent.

Some future work regarding this would be the computation of shortest path using the all pair shortest path algorithm. But as discussed earlier, the running time for this algorithm is a factor which has to be considered. This algorithm can then be used to calculate another measure of centrality called betweenness centrality. Betweenness centrality highlights an important node by focusing on the capacity of the node to be an intermediary between any two other nodes. Just like closeness centrality, it requires the computation of shortest paths. The major difference is that the shortest paths have to determined for each pair of nodes in the graph and the number of those paths which pass through a particular node in order to calculate its betweenness.

References

1) Guillaume Ereteo, Fabien Gandon, Olivier Corby, Michel Buffa. Semantic Social Network Analysis. Web Science, Mar 2009, Athenes, Greece. 2009.

2) Guillaume Ereteo, Michel Buffa, Fabien Gandon, Olivier Corby. Analysis of a Real Online Social Network using Semantic Web Frameworks. ISWC 2009, Oct 2009, Washington, United States. 5823/2009, pp.180-195, 2009.

3) Alvaro Graves, Sibel Adalı, James Hendler. A method to rank nodes in an RDF graph, Rensselaer Polytechnic Inst.,Troy, New York, 2008.

4) Ulrik Brandes, A Faster Algorithm for Betweenness Centrality, University of Konstanz, Germany, 2001

5) Teknomo, Kardi (2012) PageRank Tutorial, http://people.revoledu.com/kardi/tutorial/PageRank/

6) Spark SQL, DataFrames and Datasets Guide http://spark.apache.org/docs/latest/sql-programming-guide.html

7) Sergey Brin and Lawrence Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine, Stanford University, Stanford, 1998