Ashley Music

6/16/20

Lab 3

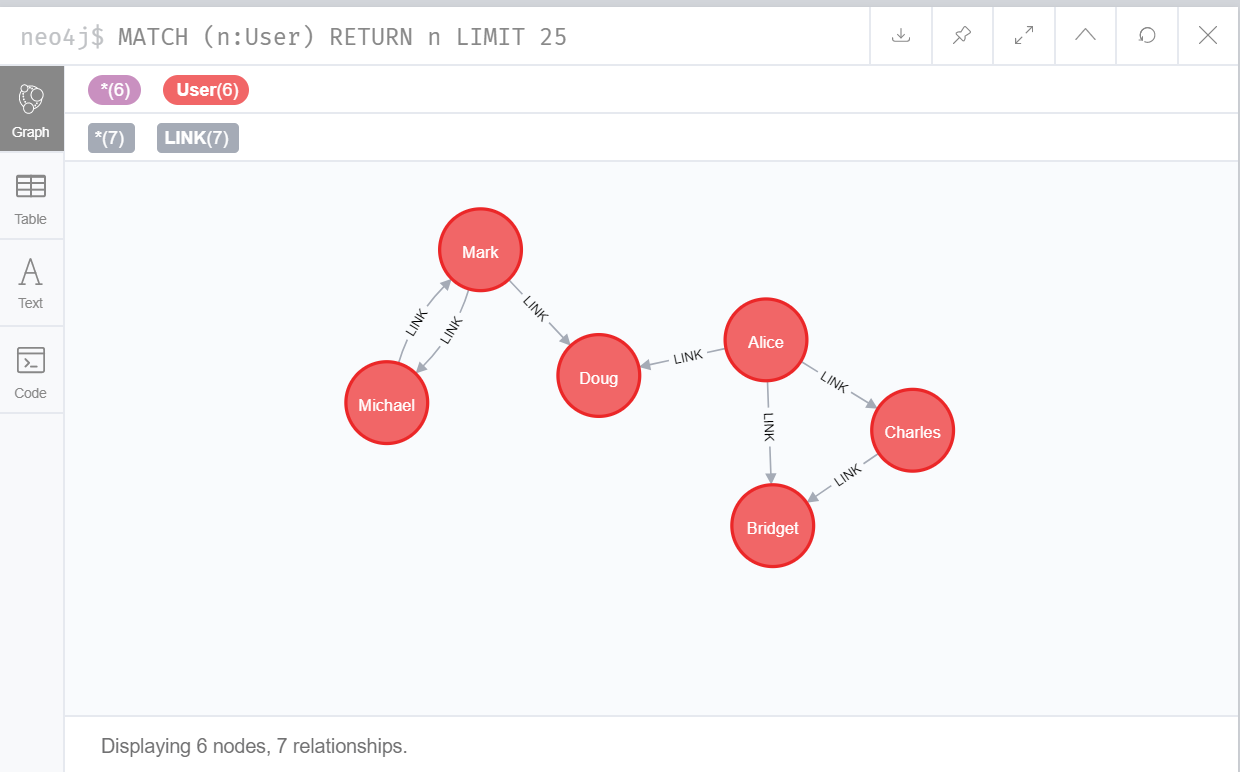
BDAT 635

**Part 1**

Choose at least one algorithm from the list and create a sample graph in Neo4j

**Louvain Example:**

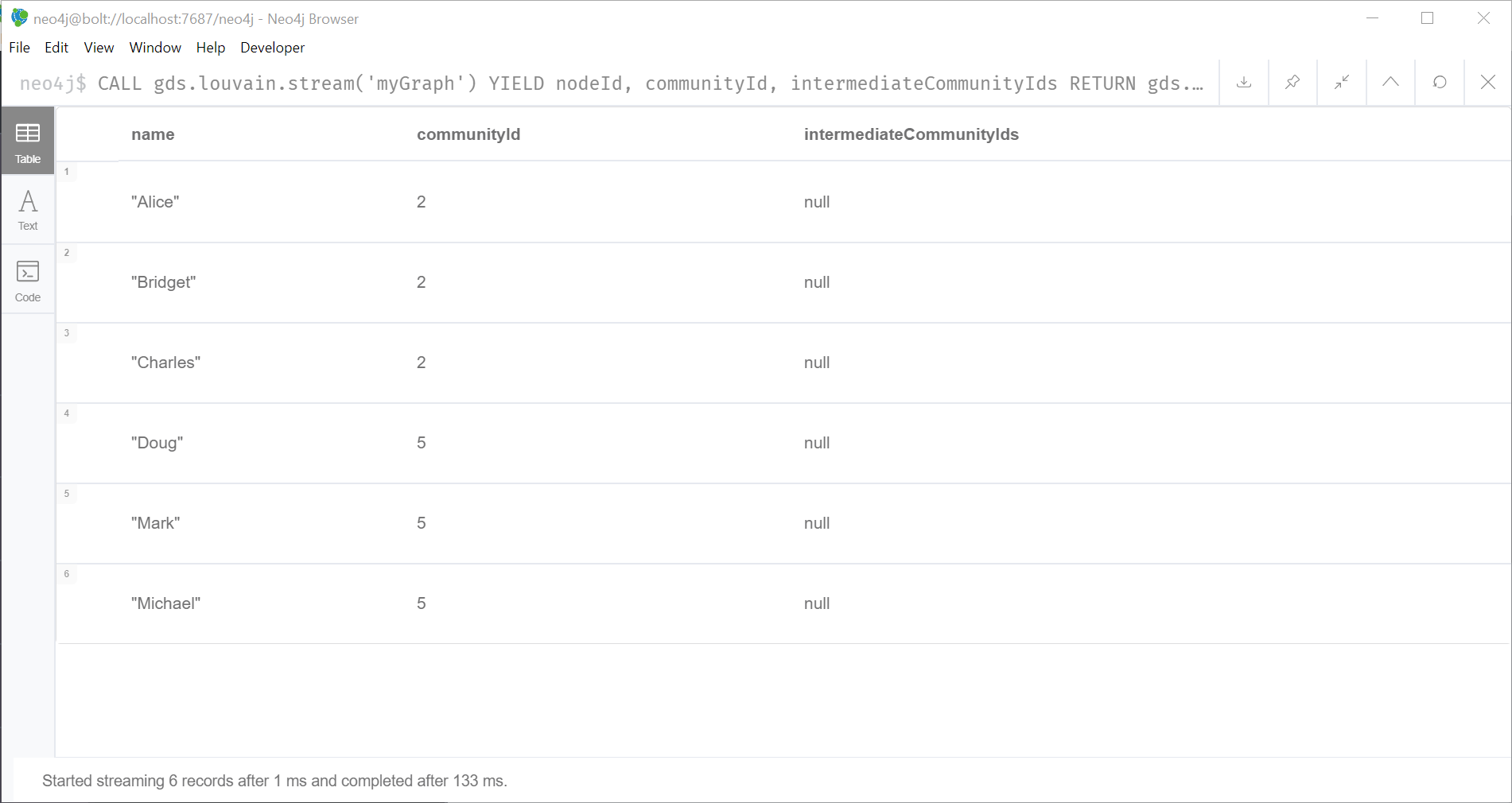
**Sample graph to run the Louvain algorithm on**



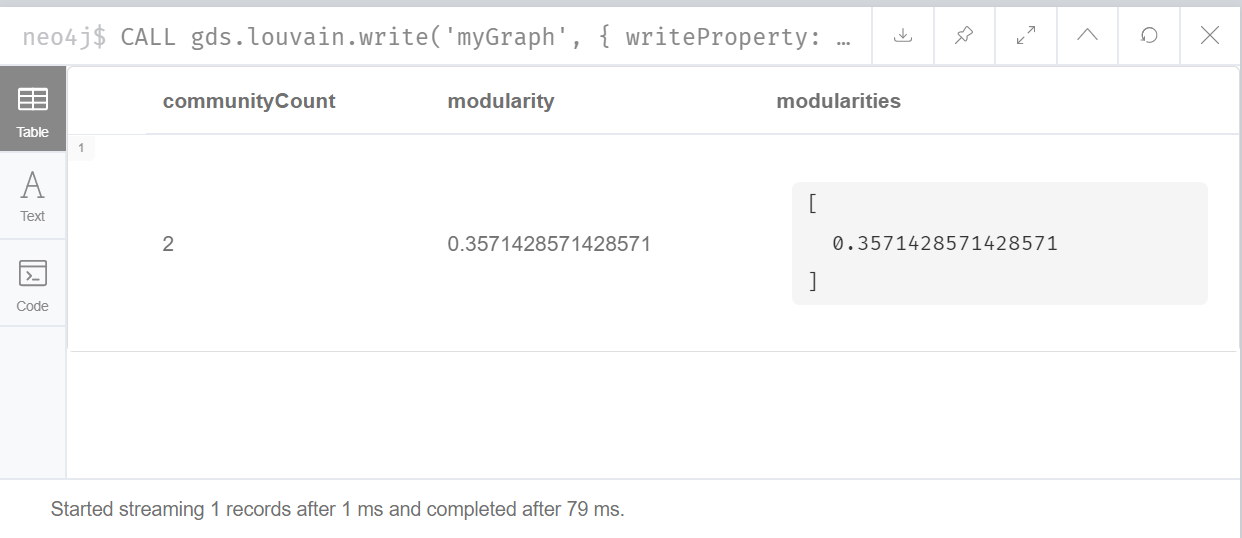
**Storing the graph**



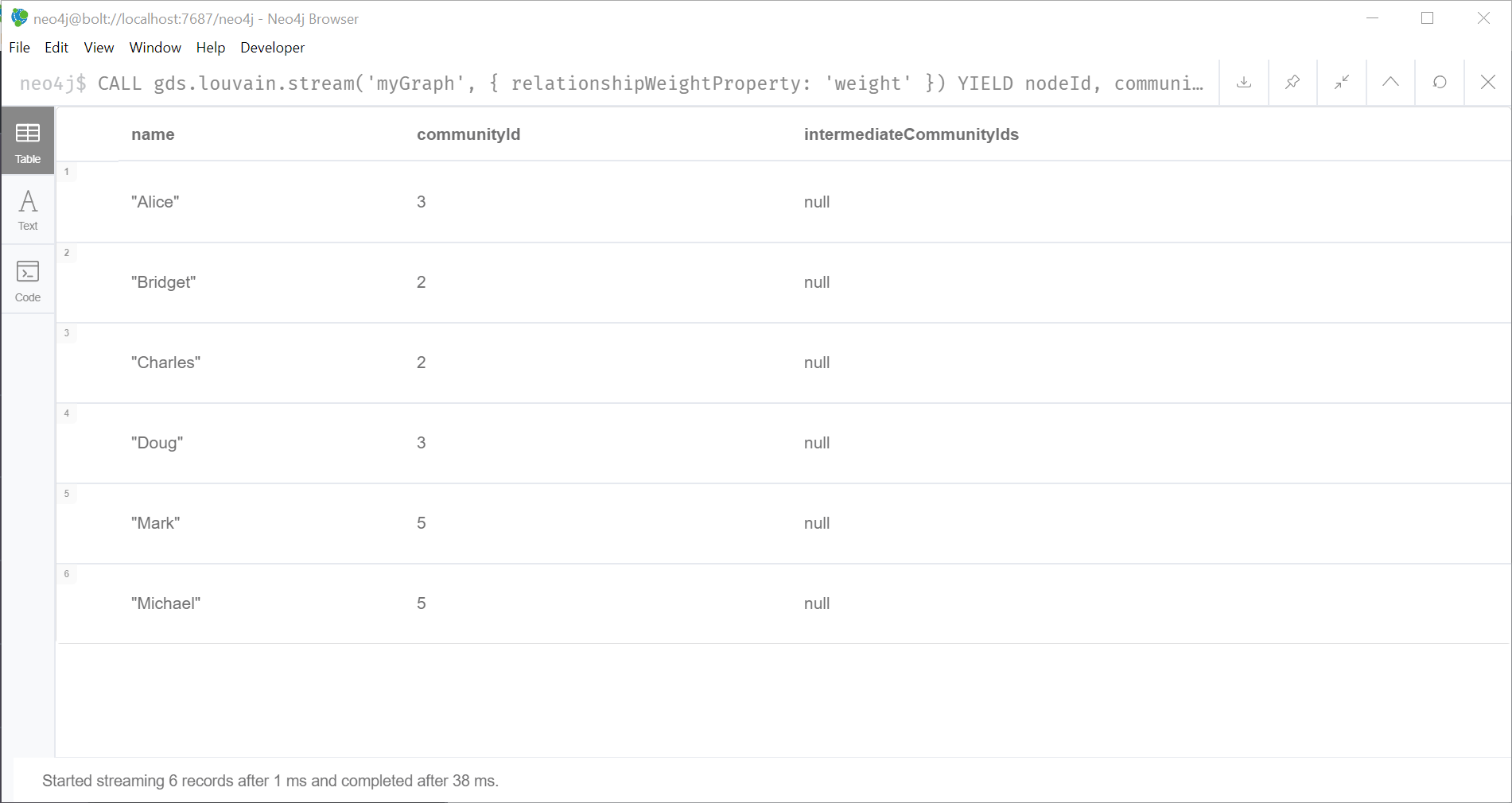
**Running the algorithm and streaming the results**



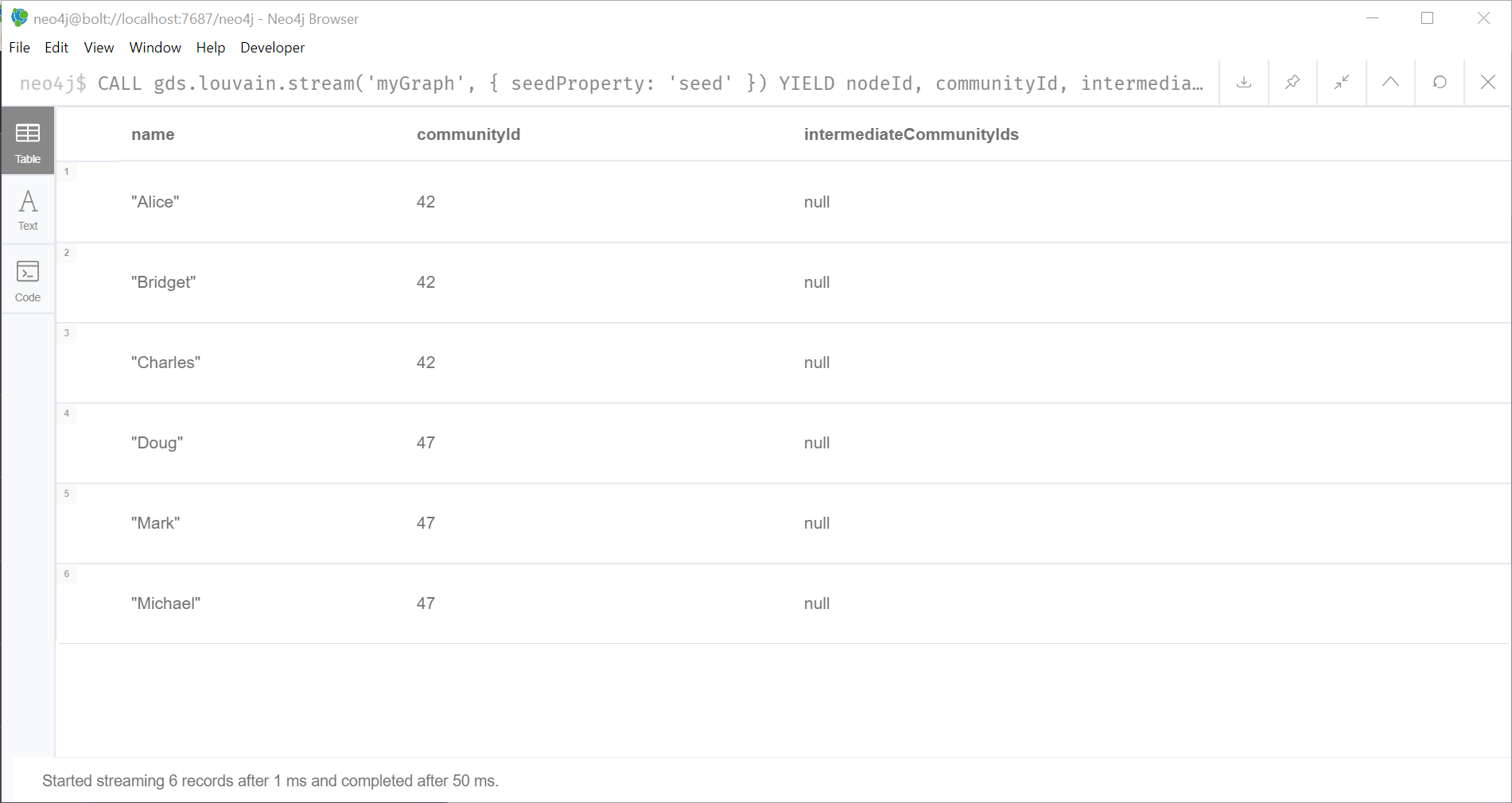
**Writing the results**



**Using a weighted graph to run the results**



**Using seed community to run the algorithm**

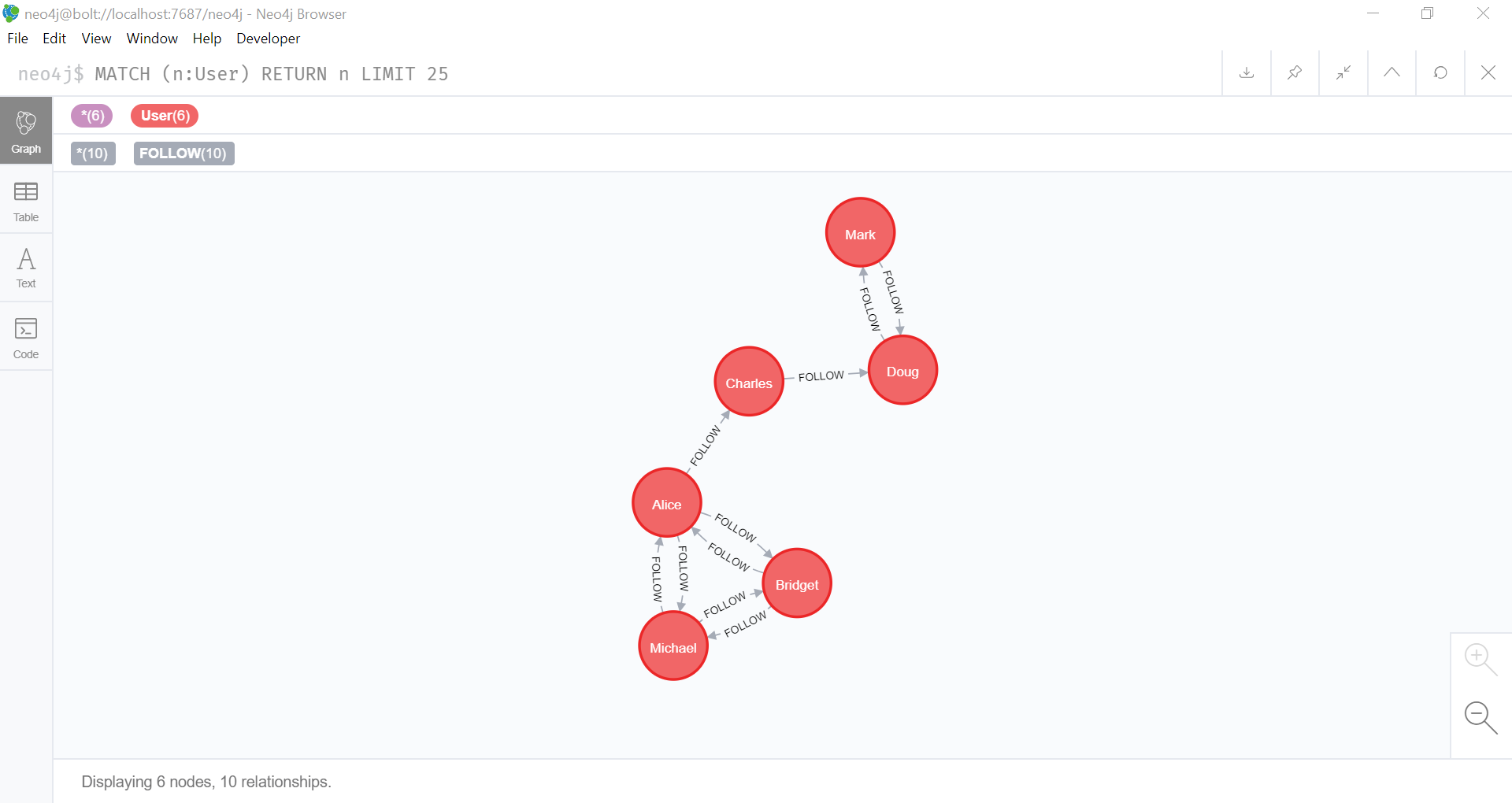


**Louvian Results:**

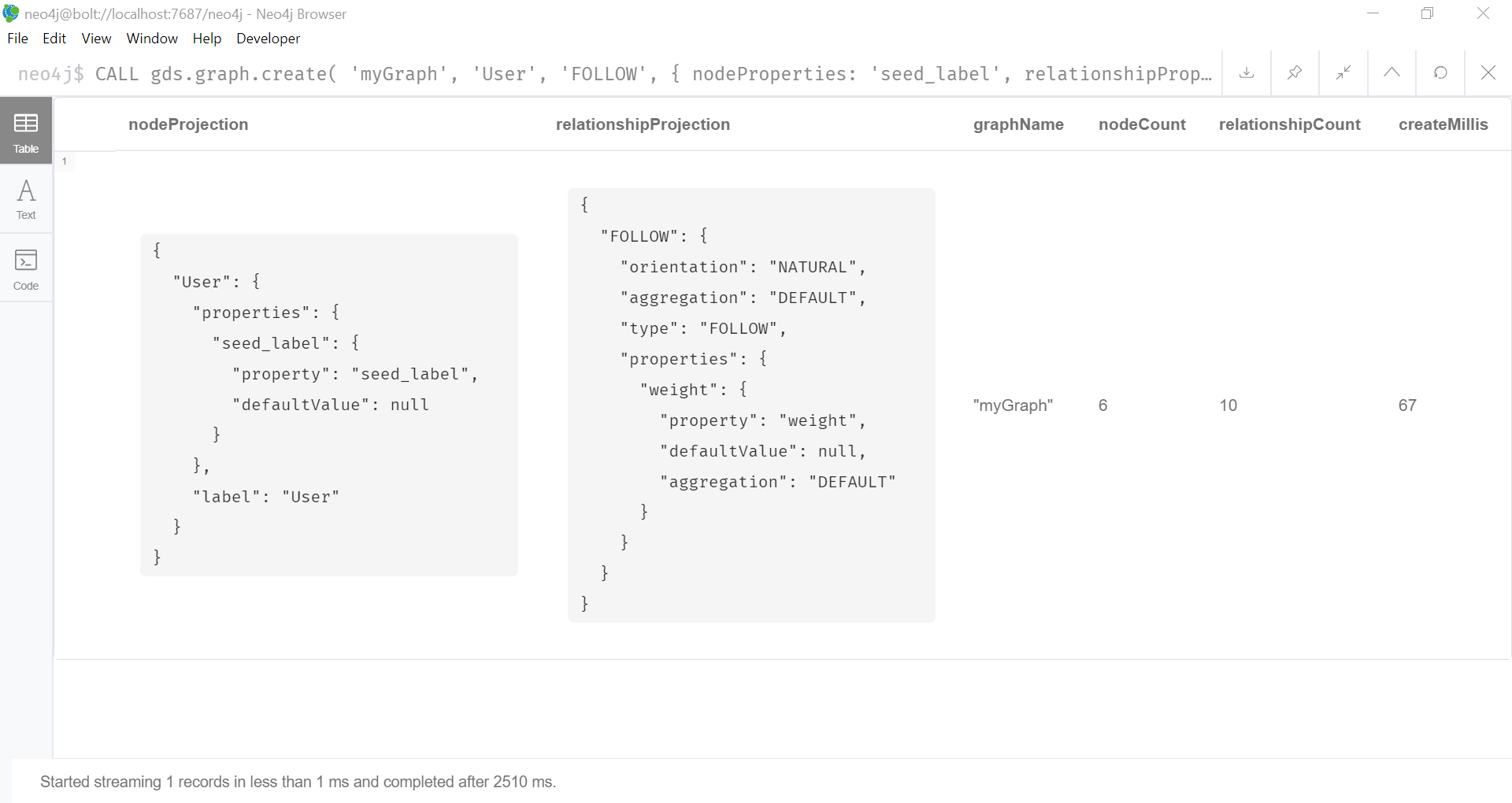
This algorithm is detecting how connected the nodes within a community compared to how they would be connected in a random network, according to the Neo4j guide. In this example we created a simple graph that has two closely connected clusters. The relationship between the nodes is an undirected link. It is weighted and the weight is determined by the strength of the relationship. For example there is a link between Michael and Mark, a link between Mark and Doug, Doug and Alice, etc. The results show us the two communities and labeled them community 2 and 5.

**Part 1 - Label Propagation:**

**Create a sample graph to run label propagation on**



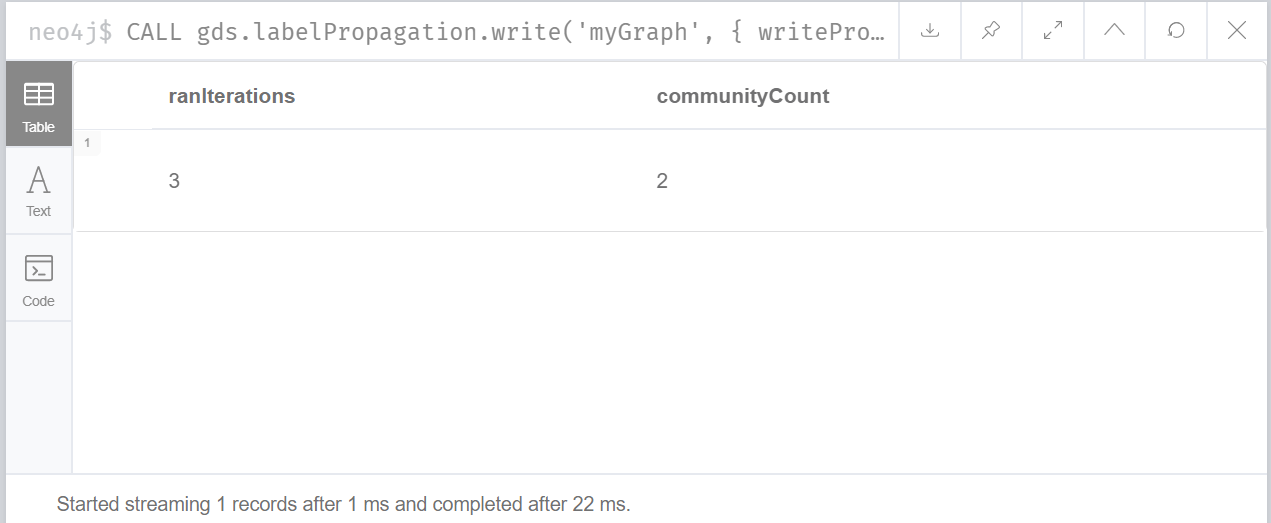
**Creating a graph**



**Running the results unweighted**



**Running the algorithm and writing the results**



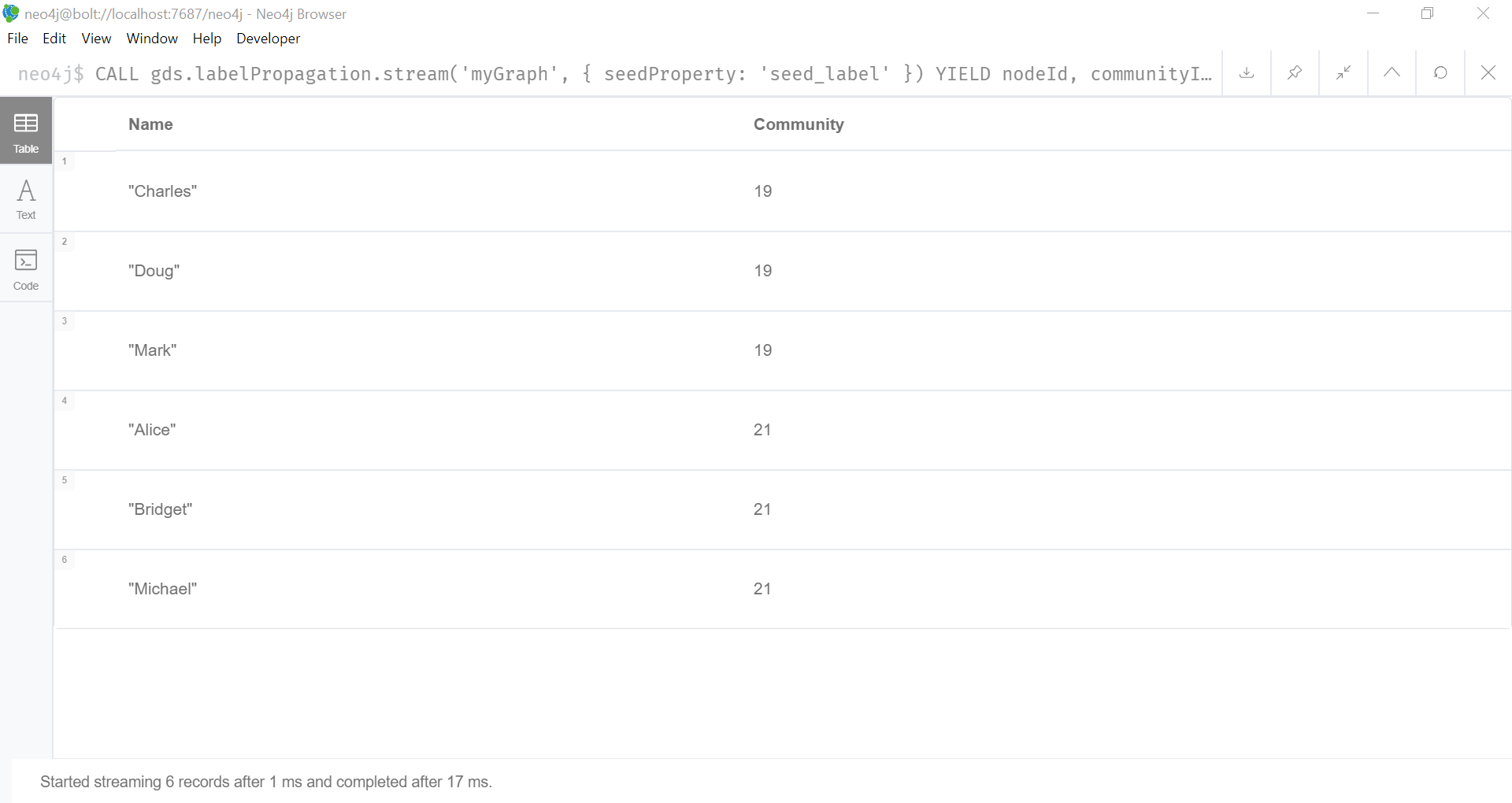
**Running the results on a weighted graph**



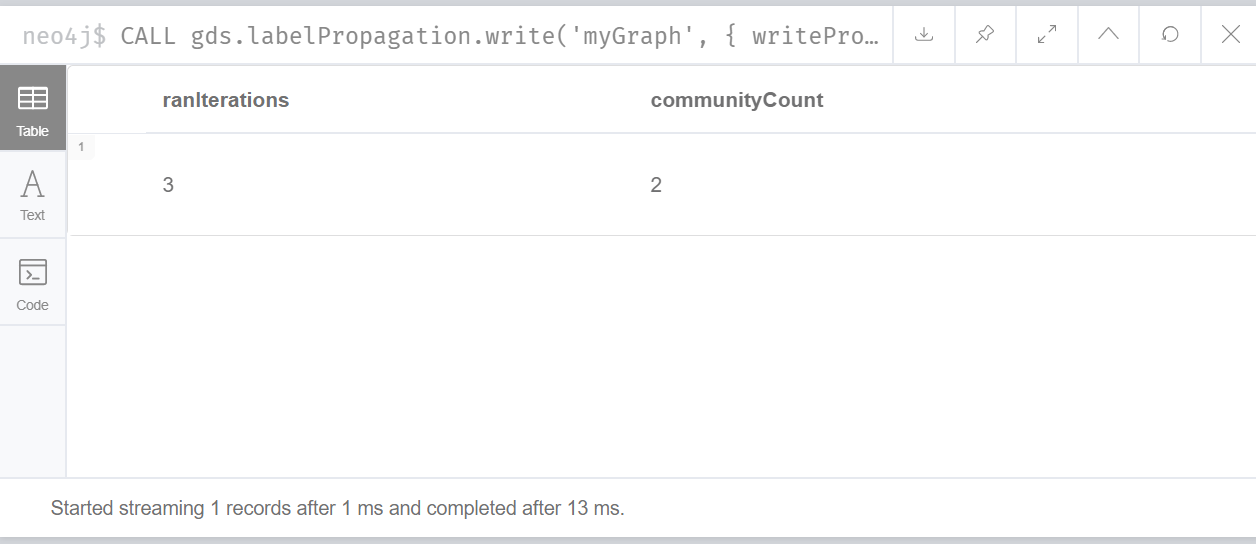
**Writing the results**



**Running the algorithm with pre-defined labels:**



**Run the algorithm and write back results**



**Label Propagation results:**

This example graph that represents a social network of people who follow each other. In this example we were given a seed label for each user. The example tells us this might have come from a previous run of the algorithm. There is also a weight added based on the relationship between the users. The unweighted graph shows 2 communities and it took 3 iterations to run. When we look at this graph based upon the provided weight we see that there are 2 communities and it took 4 iterations to run. The community looks different as well and we see that there are 4 members in one community and two in the other. When we looked at it unweighted there were three members in each community. We provided a seed on the users in the beginning and this algorithm used these set of labels it will go through all the seed labels and look at the most frequent seed label among its neighbors and update the label at each iteration. We ended up with 2 communities after 4 iterations and used the seed from Mark and Doug (19 and 21).

**Part 2 – Load Data from Lab 3 Resources**

**I selected a dataset from** [The Network Repository](http://networkrepository.com/fb-pages-government.php)  that represents a community of Government Facebook Pages and the mutual likes between them. The nodes represent government Facebook pages and the edges represent the mutual likes between those Facebook pages. You even see some recursive relationships in this graph – the Facebook page likes itself.

**Creating the nodes code:**

LOAD CSV WITH HEADERS FROM "file:///GovtNodes.csv" AS nodes

CREATE (p:Pages { name:nodes.name, id: toInteger(nodes.new\_id)})

**Creating the relationship between the nodes code:**

LOAD CSV WITH HEADERS FROM "[file:///GovtEdges.csv](file:///\\GovtEdges.csv)" AS edges

MATCH (p1:Pages {id: toInteger(edges.FromNode)}), (p2:Pages {id: toInteger(edges.ToNode)})

CREATE (p1)- [:LIKES]-> (p2);

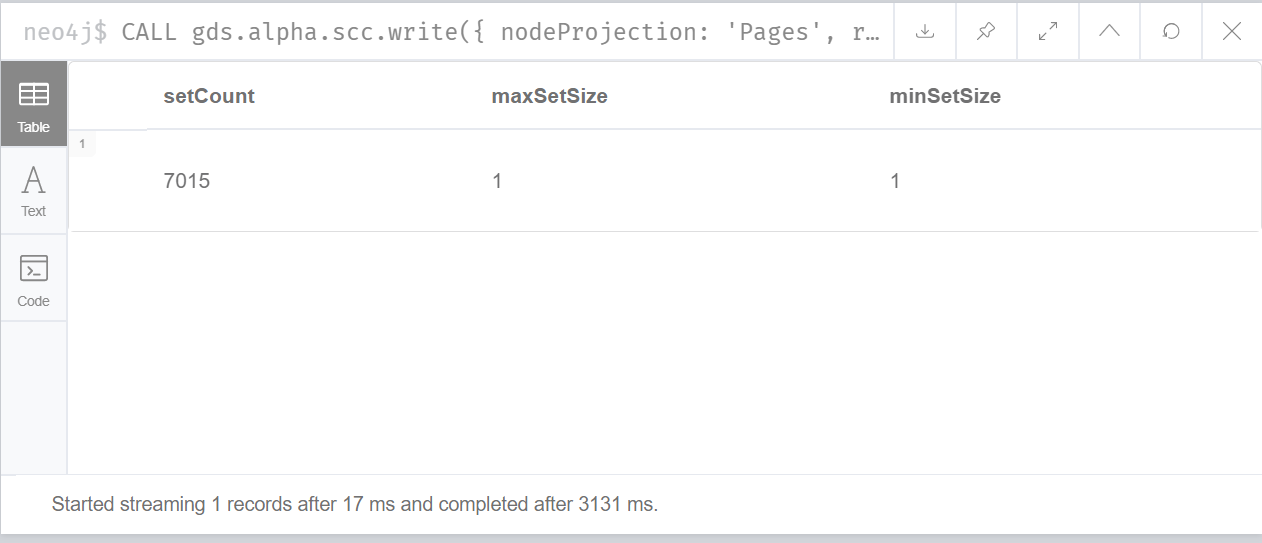
**Resulting Graph:**

****

**Part 2: SCC Algorithm**

I like the SCC algorithm because it’s simple and helps you get a feel for your graph.

**Running the algorithm and writing back results:**

****

**Code:**

CALL gds.alpha.scc.write({

nodeProjection: 'Pages',

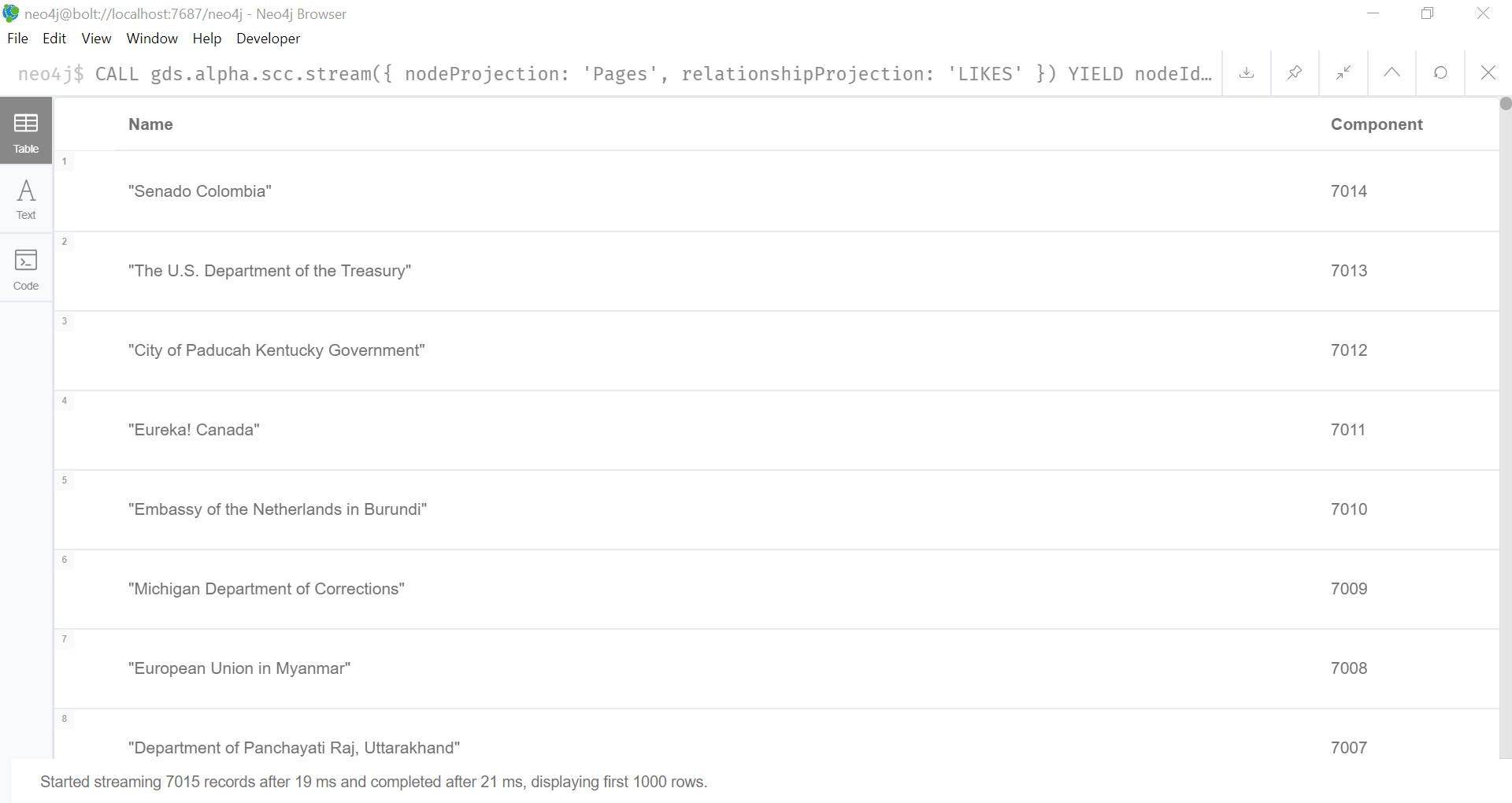
relationshipProjection: 'LIKES',

writeProperty: 'componentId'

})

YIELD setCount, maxSetSize, minSetSize;

**Running the algorithm and stream back results:**



**Code:**

CALL gds.alpha.scc.stream({

nodeProjection: 'Pages',

relationshipProjection: 'LIKES'

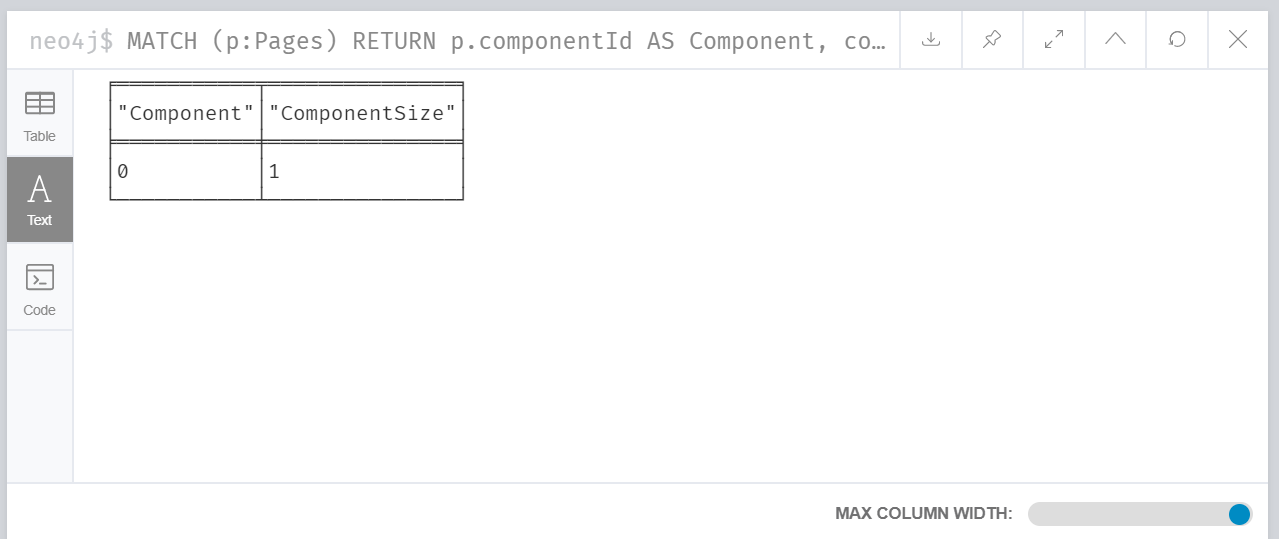
})

YIELD nodeId, componentId

RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Component

ORDER BY Component DESC

**Finding largest partition:**

****

**Code:**

MATCH (p:Pages)

RETURN p.componentId AS Component, count(\*) AS ComponentSize

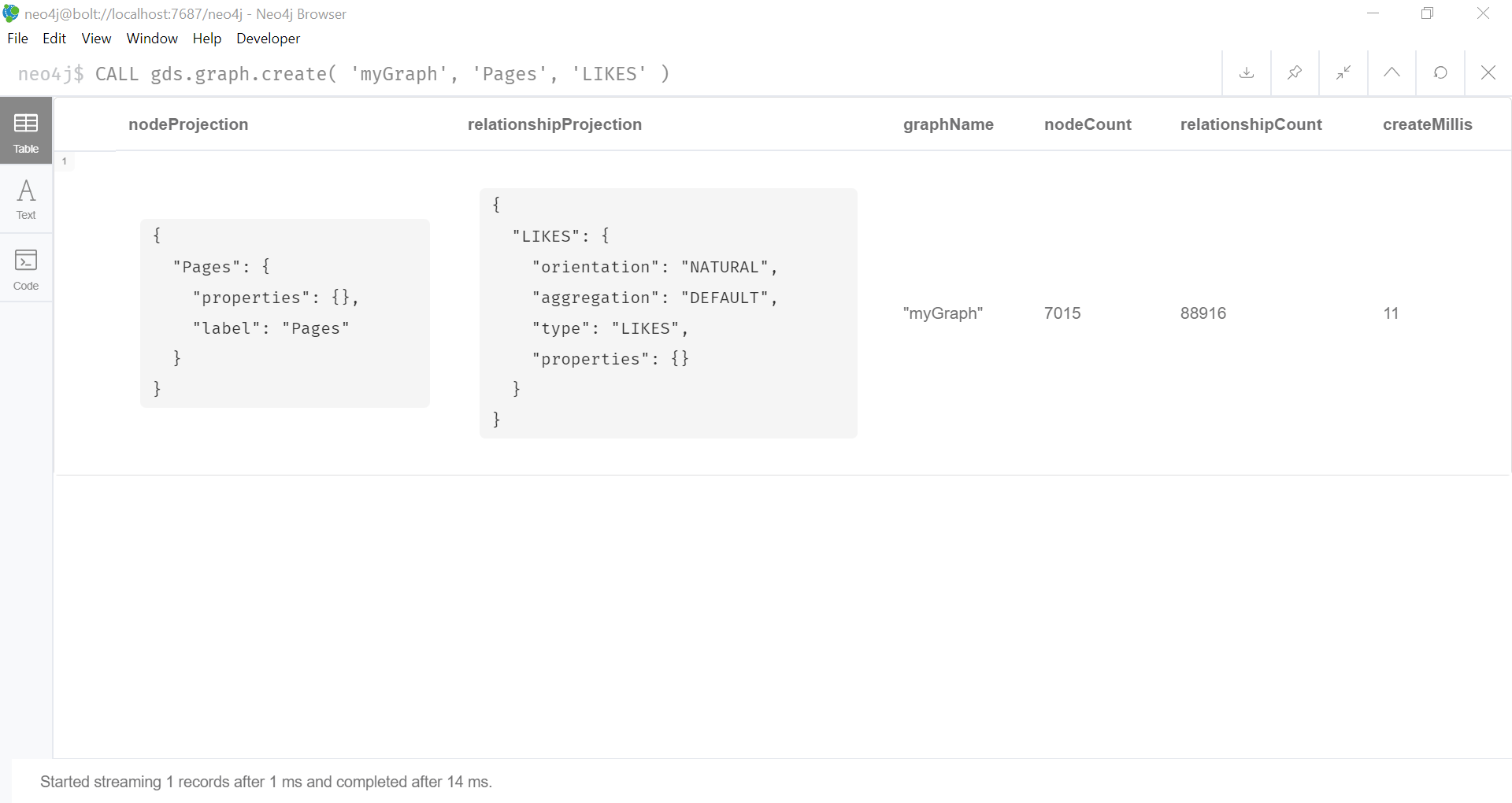
ORDER BY ComponentSize DESC

LIMIT 1

**Analysis:**

I might have done something incorrect, but it appears there are no strongly connected components in this graph. I decided to run the WCC algorithm to see if I could find any connected components and I did find 4 when I ran the WCC.

**Part 2 – WCC**

****

**Code:**

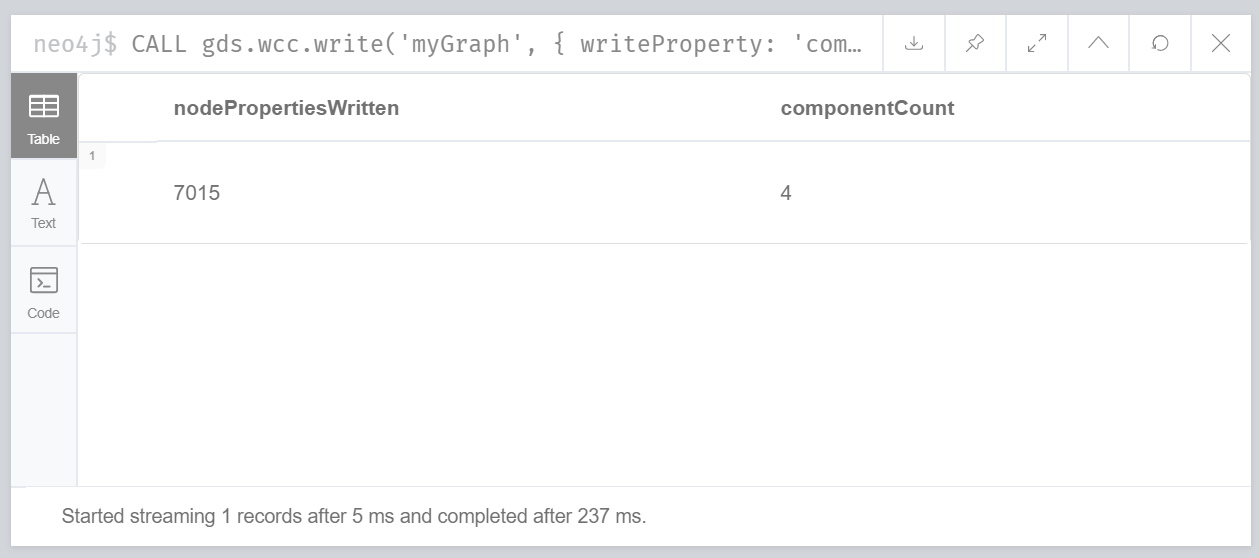
CALL gds.graph.create(

'myGraph',

'Pages',

'LIKES'

)



Code:

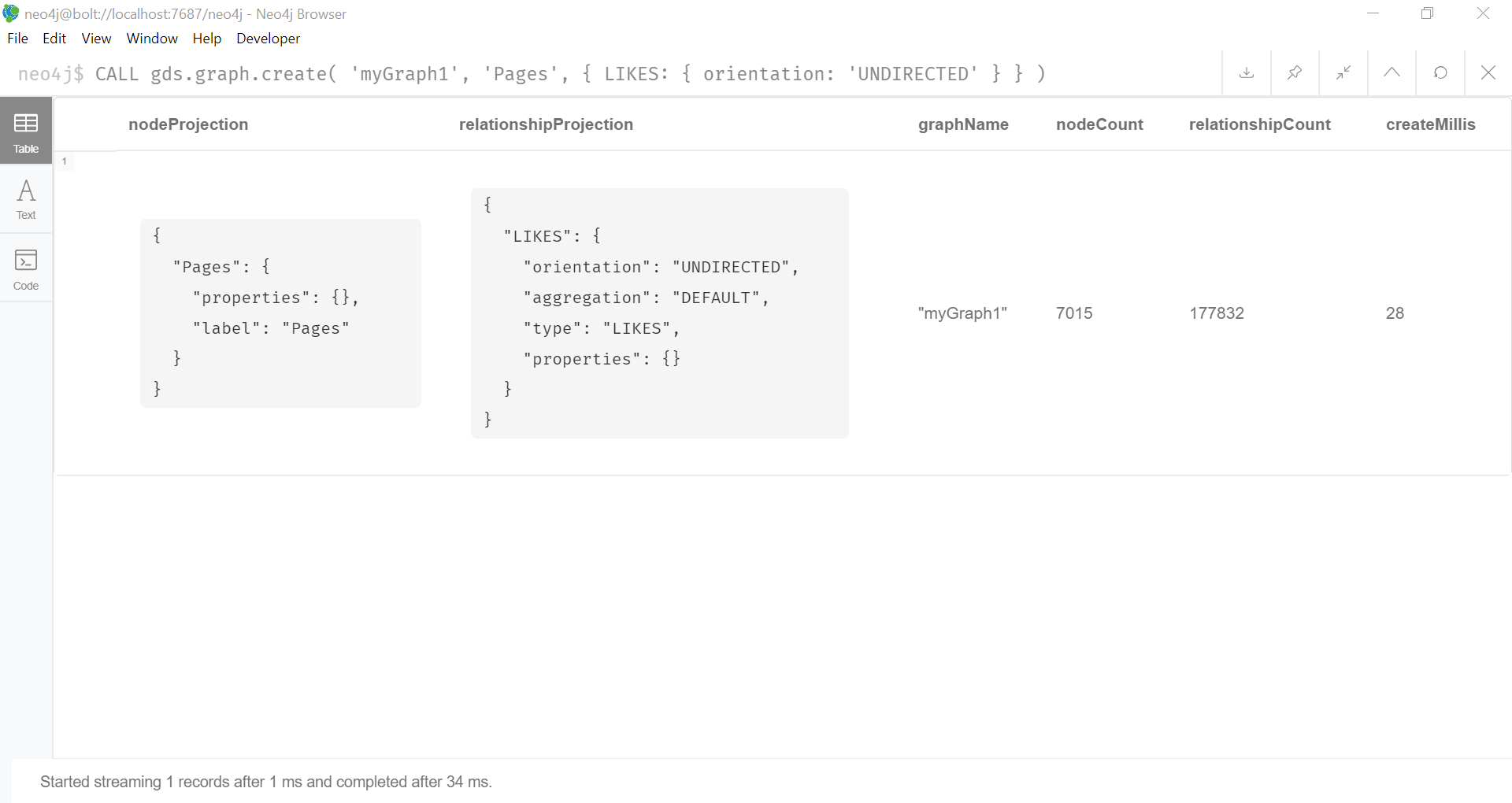
CALL gds.wcc.write('myGraph', { writeProperty: 'componentId' })

YIELD nodePropertiesWritten, componentCount;

**Part 2: Triangle Count:**

Based on my data set I wanted to see if I could find any triangles in the graph – or a “set of three nodes where each node has a relationship to the other”.

**Creating the Graph:**

****

**Code:**

CALL gds.graph.create(

'myGraph1',

'Pages',

{

LIKES: {

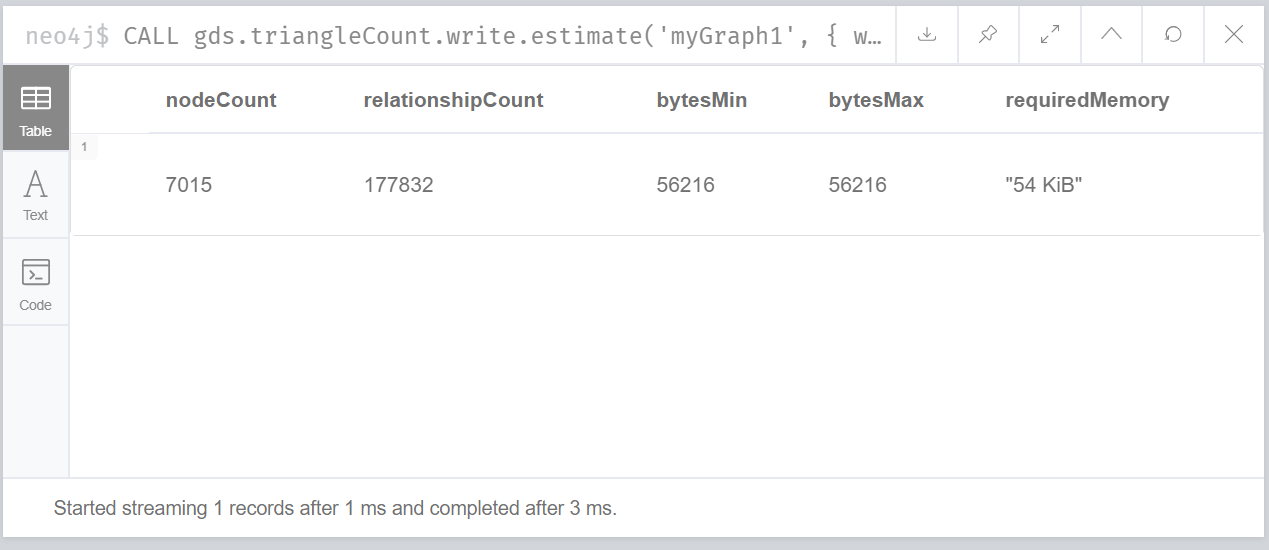
orientation: 'UNDIRECTED'

}

}

)

**Estimating the memory requirements:**

****

**Code:**

CALL gds.triangleCount.write.estimate('myGraph1', { writeProperty: 'triangleCount' })

YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory

**Analysis**:

This part was cool to see because I didn’t realize the relationship count doubles because of the undirected orientation.

**Running the algorithm in stream mode to return the number of triangles in each node:**

****

**Code:**

CALL gds.triangleCount.stream('myGraph1')

YIELD nodeId, triangleCount

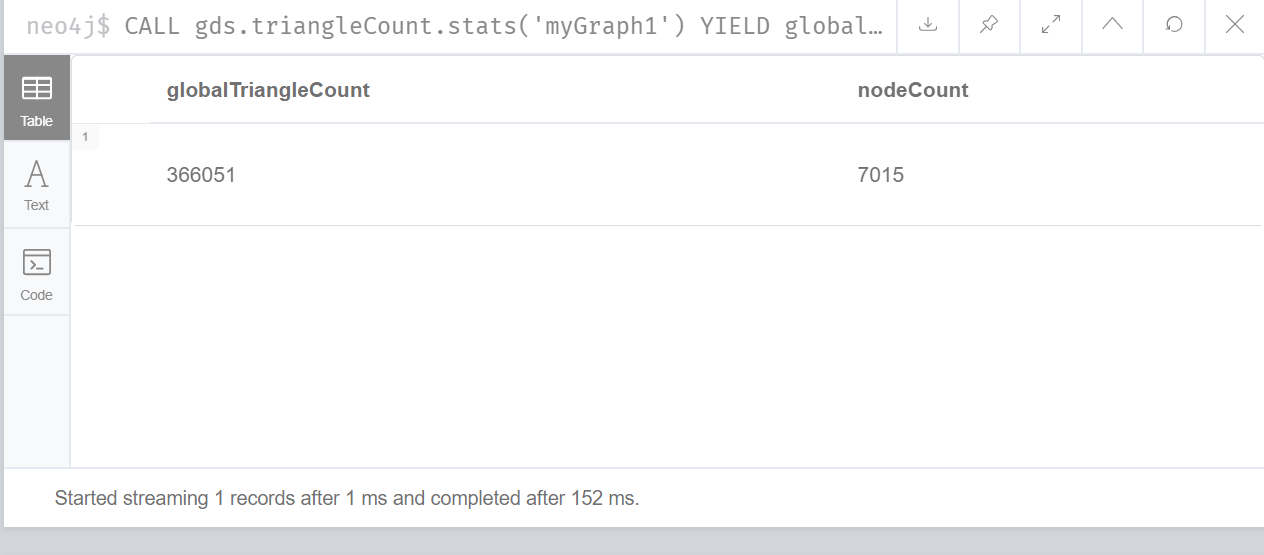
RETURN gds.util.asNode(nodeId).name AS name, triangleCount

ORDER BY triangleCount DESC

**Analysis:**

I did find 3-cliques in this community. The U.S. Army Chaplain Chorps has 8863 triangles!

**Stats:**

****

**Code:**

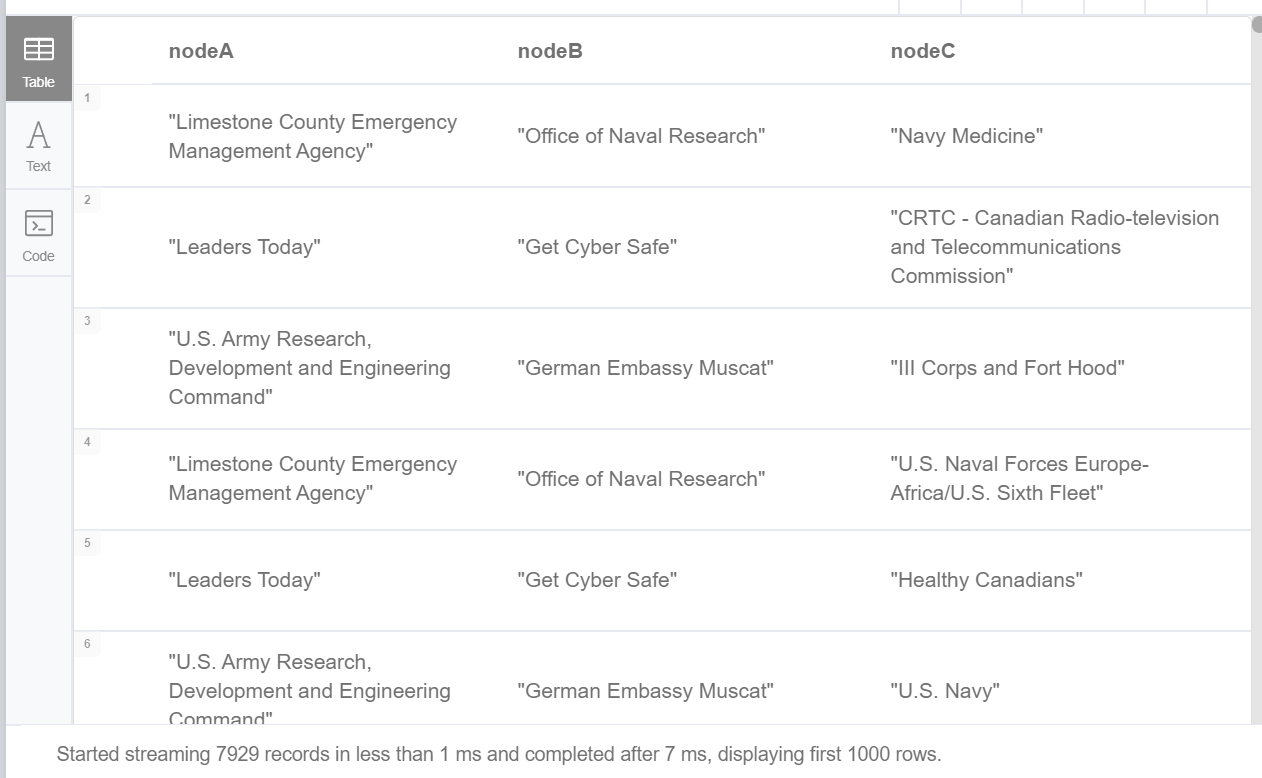
CALL gds.triangleCount.stats('myGraph1')

YIELD globalTriangleCount, nodeCount

**Analysis:**

There were a lot of triangles found in this data which I think you might expect. Out of 7015 nodes there were 366051 triangles found.

**Listing all Triangles:**

****

**Code:**

CALL gds.alpha.triangles('myGraph1')

YIELD nodeA, nodeB, nodeC

RETURN

gds.util.asNode(nodeA).name AS nodeA,

gds.util.asNode(nodeB).name AS nodeB,

gds.util.asNode(nodeC).name AS nodeC

**Analysis:**

It was interesting to see who liked who in this community. I saw the German Embasy Muscat, the Office of Naval Research and U.S. Army Research in there a lot.

Source and Acknowledgements:

@inproceedings{nr,  
     title={The Network Data Repository with Interactive Graph Analytics and Visualization},  
     author={Ryan A. Rossi and Nesreen K. Ahmed},  
     booktitle={AAAI},  
     url={<http://networkrepository.com>},  
     year={2015}  
}