SOEN 363
Project Phase 2

Karim Hasbini—40053498
MHD Layth Awad— 26690394

a) Dataset is top 2 million upvoted reddit comments for may 2015 found at Kaggle

b)Neo4j was chosen. The data model will be a graph of nodes where each node is either a user, a comment or a subreddit.

code: importing the dataset

importing the users as nodes:

```
load csv with headers from "file:///unique_users.csv" as row
create(n:User{username:row.author})
```

importing the comments as nodes:

```
:auto using periodic commit 100000
load csv with headers from "file:///highest_clean4.csv" as row
create(n:Comment)
set n = row,
n.id = row.id,
n.parent_id = row.parent_id,
n.subbreddit_id = row.subbredit_id,
n.text = row.text,
n.score = row.score,
n.author = row.author,
n.controversiality = (row.controversiality = "0")
```

importing and creating subreddit nodes:

```
load csv with headers from “file:///subreddits.csv” as row
create (n:Subreddit{subreddit_id:row.subreddit_id})
```

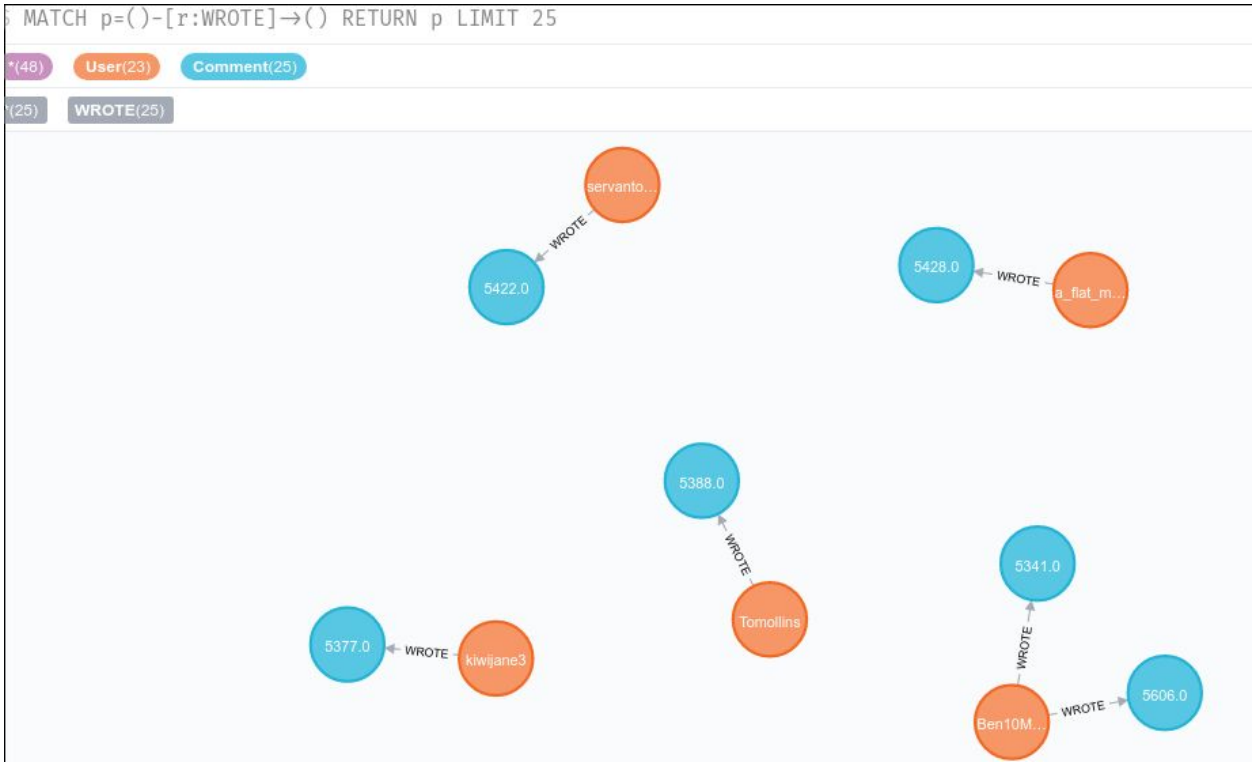
Creating an index for each node type:

```
create index for (n:User) on (n.username)

create index for (n:Subreddit) on (n.subreddit_id)

create index for (n:Comment) on (n.id)
```

Queries:

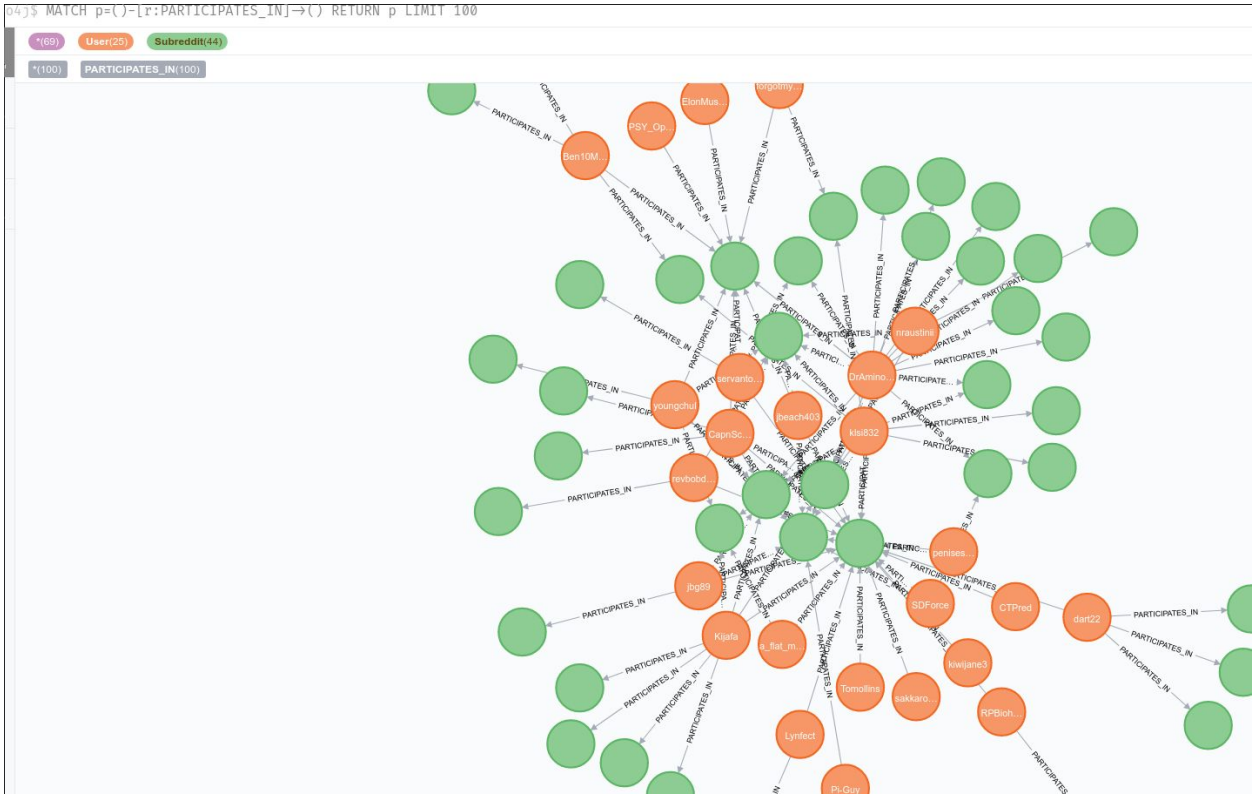
Creating a relationship between the user and the comment the user wrote:

call apoc.periodic.iterate(
"match (u:User), (c:Comment) where u.username = c.author return u,c",
"create (u)-[:WROTE]->(c)",
{batchSize:10000, iterateList:true})



Create a relationship between users and the subreddits they commented in:

call apoc.periodic.iterate(
"match (u:User)--(c:Comment) return u, c",
"match (c), (s:Subreddit) where c.subreddit_id = s.username merge
(u)-[:PARTICIPATES_IN]->(s)",
{batchSize:10000, iterateList:true})

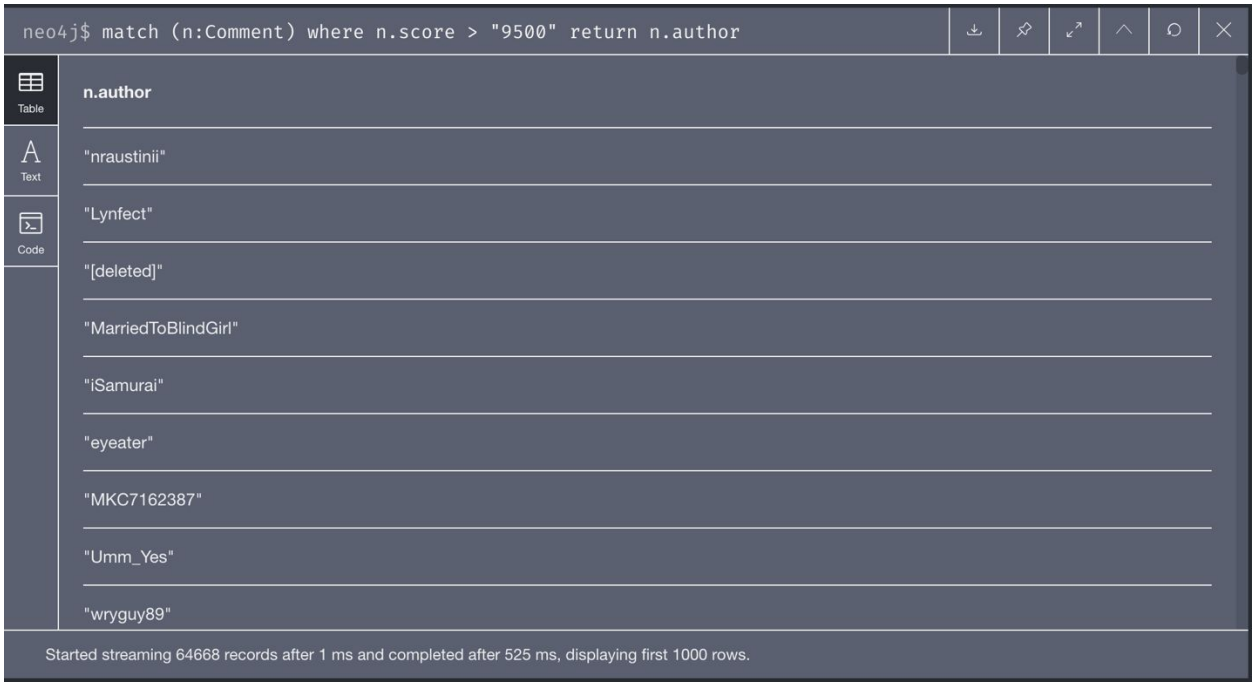


Get distinct subreddits (this query, while simple, was exported to a csv which allowed us to create the subreddit nodes):

match(n:Comment)
return distinct (n.subreddit.id)

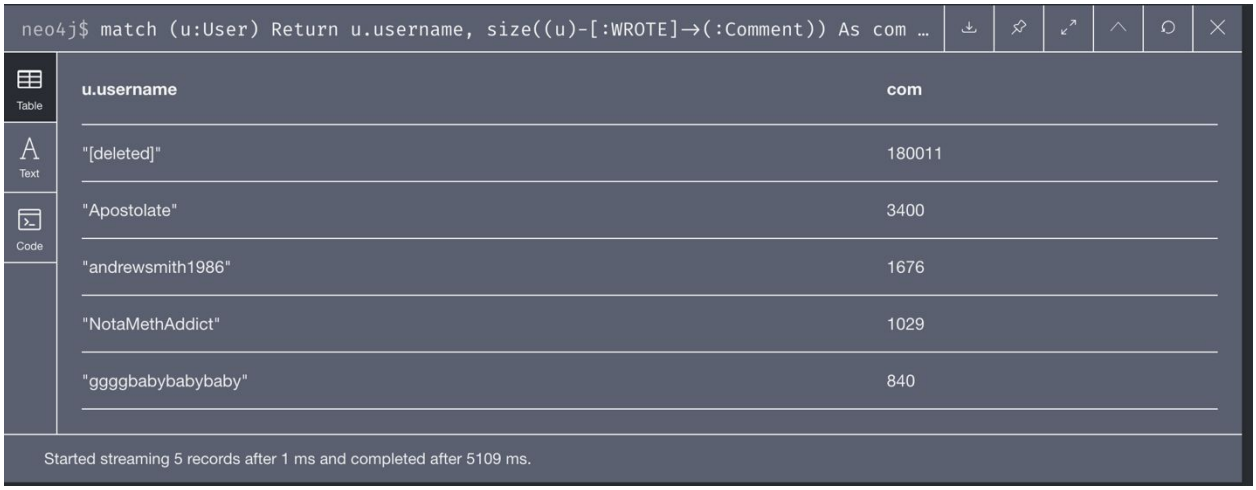
Find users with comments whose score is above 9500

```
match (n:Comment)
where n.score > "9500"
return n.author
```



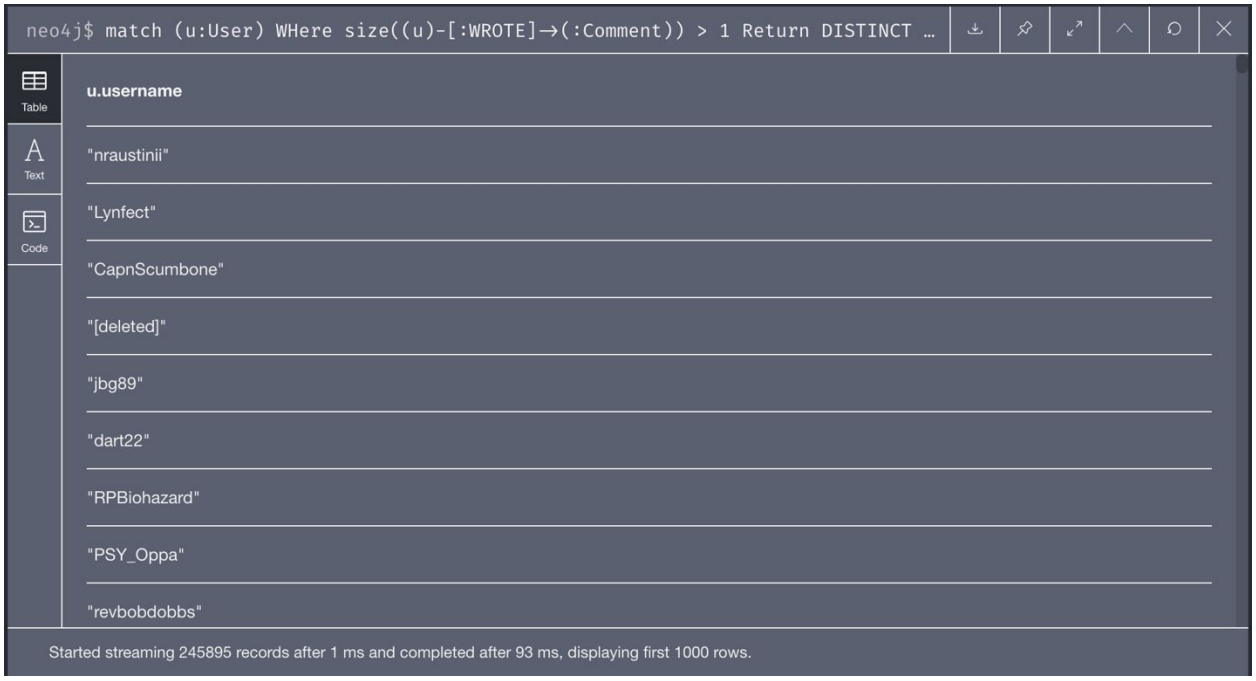
Find the 5 users with the most comments in the top 2 million

```
match (u:User)
Return u.username, size((u)-[:WROTE]->(:Comment)) As com
ORDER BY com DESC
LIMIT 5
```



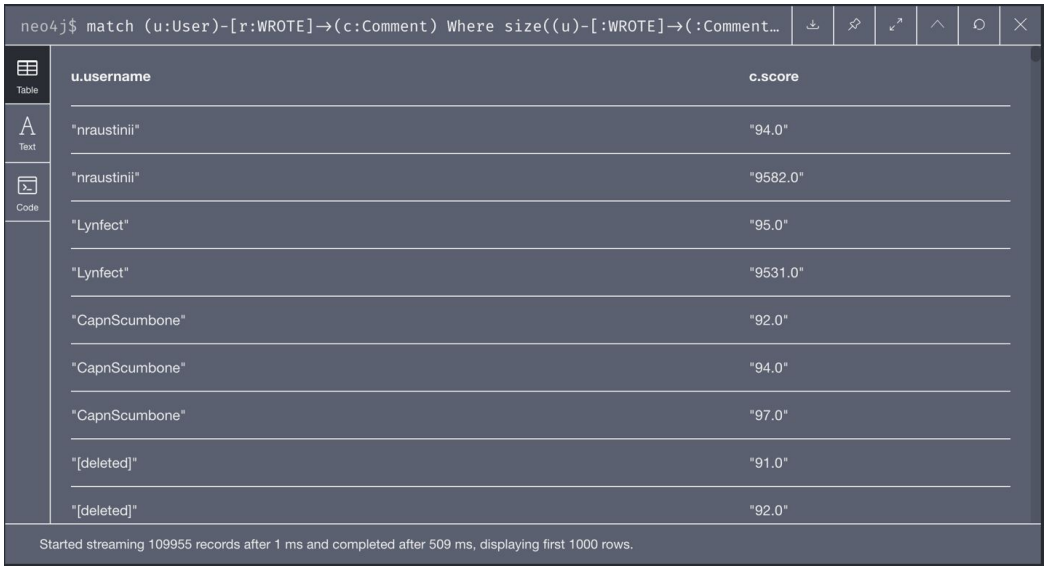
Get usernames of those who wrote more than 1 comment:

```
match (u:User)
WHere size((u)-[:WROTE]->(:Comment)) > 1
Return DISTINCT u.username
```

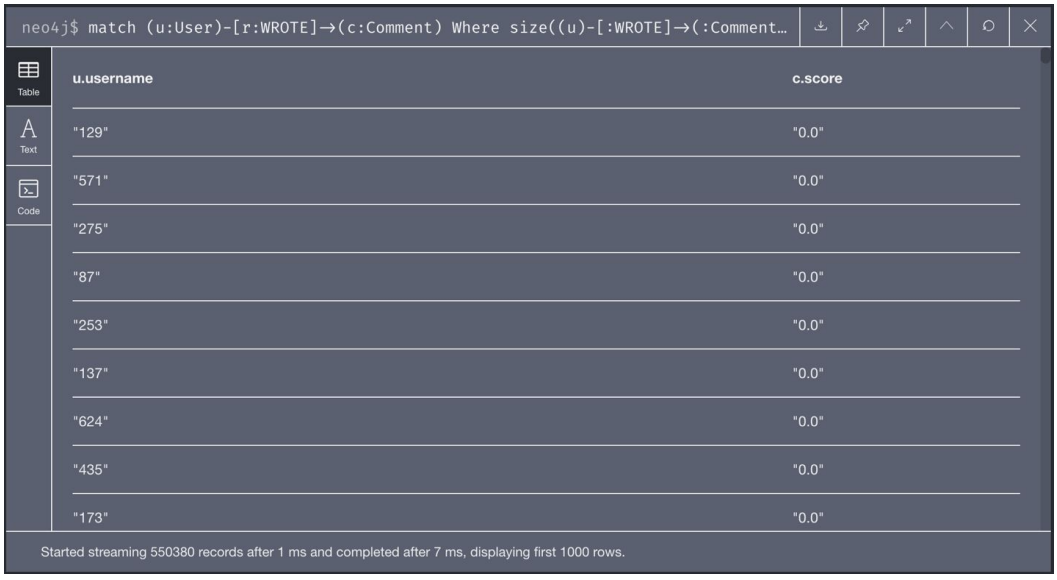


The following two queries are to show the difference between the number of users with score over 9000 and below 2000, and we can see that the users with score in the top part are more than double the people in the bottom.

Find all unique users with more than 1 comment and score over 9000
match (u:User)-[r:WROTE]->(c:Comment)
Where size((u)-[:WROTE]->(:Comment)) > 1 and c.score > "9000"
Return DISTINCT u.username, c.score

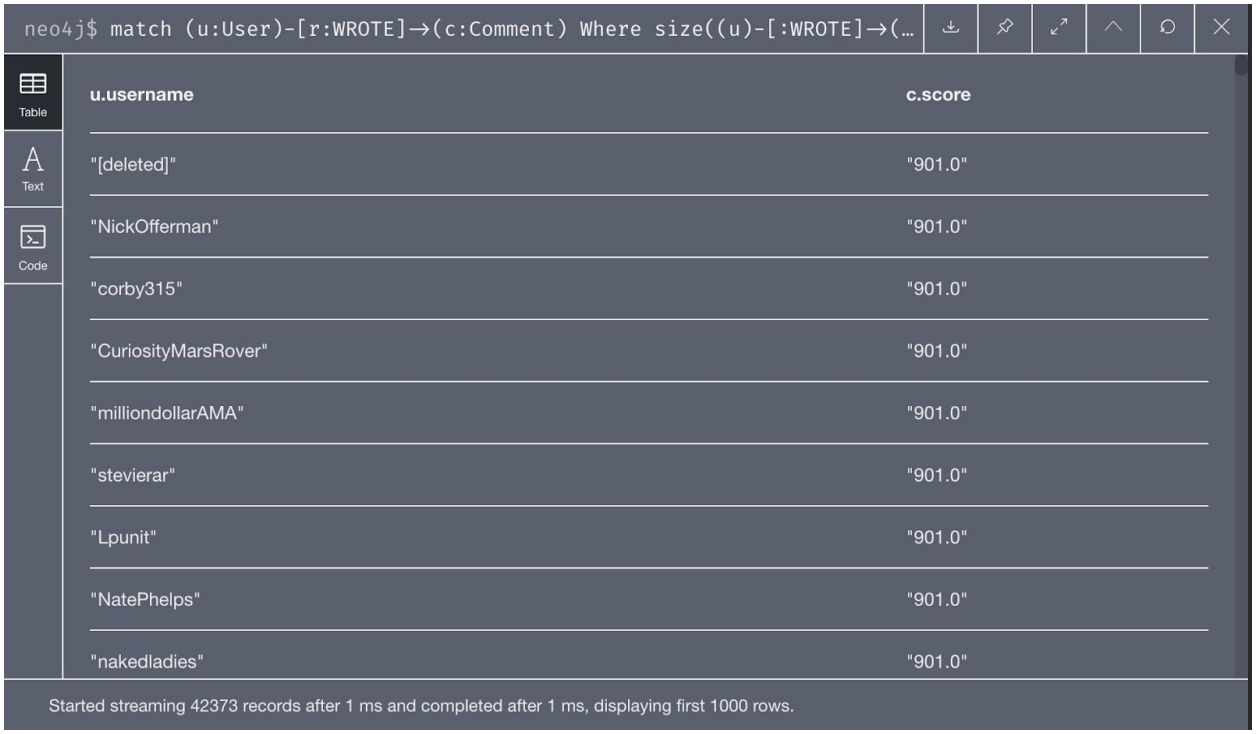


Find all users with more than 1 comment and a score lower than 2000
match (u:User)-[r:WROTE]->(c:Comment)
Where size((u)-[:WROTE]->(:Comment)) > 1 and c.score < "2000"
Return DISTINCT u.username, c.score
ORDER BY c.score



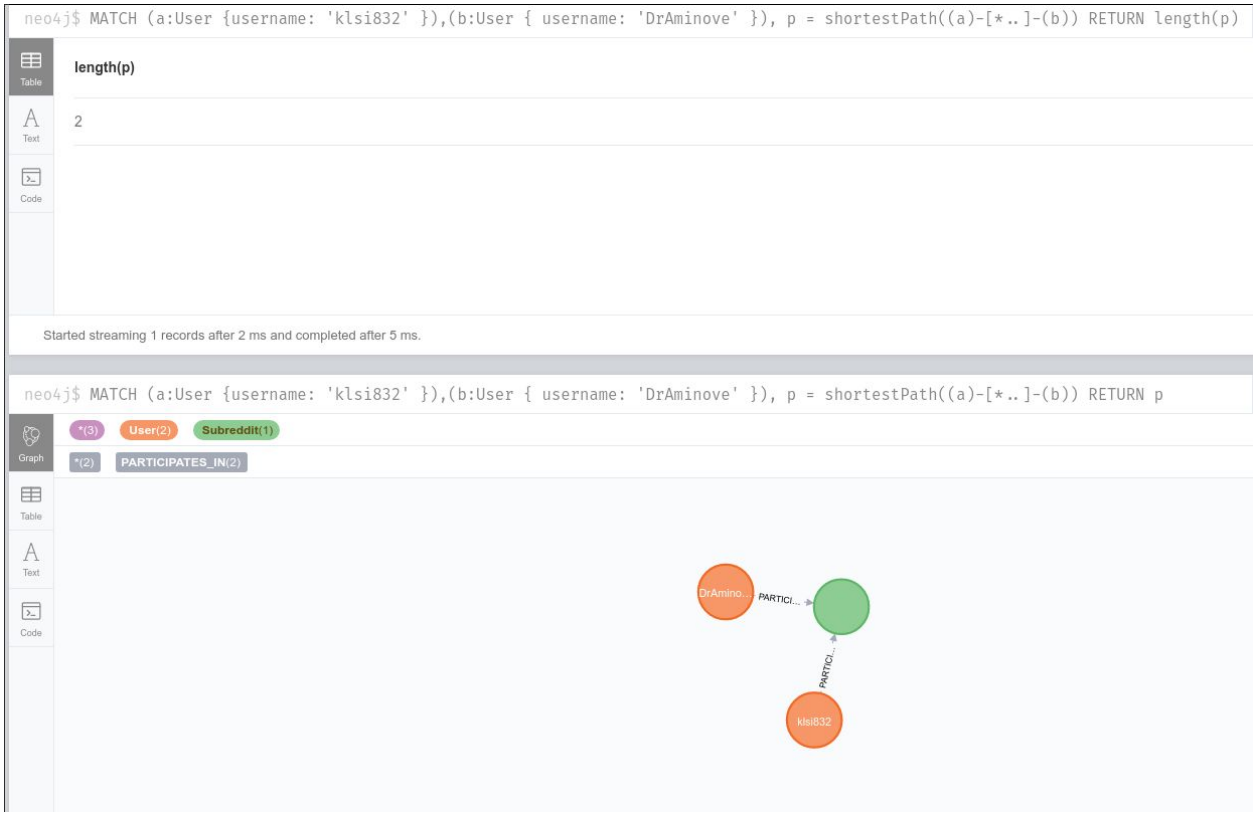
Find all unique users with more than 1 comment, score over 9000 and a big enough comment text(we chose size 100)

```
match (u:User)-[r:WROTE]->(c:Comment)
Where size((u)-[:WROTE]->(:Comment)) > 1 and c.score > "9000" and size(c.text) > 100
Return DISTINCT u.username, c.score
ORDER BY c.score
```



Find the degrees of separation (shortest path) between 2 users:

MATCH (a:User {username: 'klsi832' }),(b:User { username: 'DrAminove' }), p =
shortestPath((a)-[*..]-(b))
RETURN p



f) Neo4j supports a consistency checker for the database. However, in order to take advantage of this tool, the database must be taken offline while the tool is analyzing it, thus incurring a significant availability cost. Until very recently, Neo4j was centralized, being able to only be run on one server. Taking this into account, using this tool was quite costly. As of version 4.0, Neo4j upgraded its scalability to allow for horizontal scaling[1]. In our system, we did not need to run the consistency checker. Neo4j was run on only one local server, however interestingly, in terms of consistency, the database allows running simultaneous read queries in most cases. Care should be taken, as concurrent write queries may allow for inconsistent results. This happens while running 2 queries with a merge clause in each (it creates a node only if it does not already exist), and ending up with duplicate values.

g) Indexing:

Indexing in Neo4j is used to make searches more efficient, but like we learned in class this will come with a price paid in memory. It is a way to tell the DBMS that this will be used a lot later for searches and queries, so it will be managed by the DBMS which makes it an important task to choose what to index.

Cypher in Neo4j has two indexes:
single-property index and composite index.
Both types support all types of predicates(like equality, list, range,contains, etc…)
But the main difference is that composite indexes need all properties used to be indexed, which is not needed in the single-property index.

We have used a part of the single-property index called schema index in our project, which creates an index for the property chosen on all the nodes we specify. This will make it easier for the DBMS to search.Normally the DBMS has to search all the nodes for the property we need, but with indexing, we will have a list of the property we need to search it directly without going through unneeded results.

Since schema index can’t support multiple properties now, a special way of going around this is using multiple properties, which is combining two properties into one and using that to search.

Reference:
[1]:https://neo4j.com/whats-new-in-neo4j/scalability/

[2]:https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-search-performance/#administration-indexes-single-vs-composite-index