# 1. Introduction

The project aims to solve problems and analysis on a set of Question and Answer data with basic queries with two specific NoSQL systems, MongoDB and Neo4j. Given four csv files represented four groups of data relating to posts, users, votes and tags respectively, we design appropriate schema to support the target queries, improve the system's’ efficiency, and work out the queries.

# 2. Data Preprocessing

Use python to create new csv files. For Posts.csv file, we added CreationTime column to it and update its Tags column. For Votes.csv file, we added CreationTime column to it.

## 2.1 Posts.csv

### CreationTime

For CreationDate column, we converted the original time format to Unix Timestamp format. Because we found that there are two time formats in CreationDate. For instance, one is “2013-12-05T10:10:00.000Z” and the other one is “05/12/2013 10:10”. If we directly convert type of CreationDate in mongodb, I think these two different formats will cause problems. In addition, we found that we cannot use date type unless we install a plugin. At last, we decide to use the Unix Timestamp, which is in purely number format and it’s convenient to compare in both Neo4j and MongoDB systems. The CreationTime column will be added after running the code as shown in Figure 1.



Figure 1. Convert CreationDate to CreationTime

### Tags

In original Posts.csv file, the data in Tags column is like this, “data-request,usa”. However, it will cause problem. After we imported the original data to mongodb, it changed to “\”data-request,usa\””. If we have another Tags as “\”usa,government\””, and we unwind the Tags. Then we will get two different tags “\”usa” and “usa”. But in fact, they should be the same. To solve this problem, we decided to delete the “” symbol in Tags column in original dataset. In this case, symbol / will not appear again. The python code is shown in Figure 2.

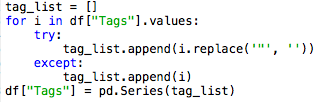


Figure 2. Replace “” in Tags

Until now, the processing of Posts.csv has been finished. We can check the result in mongodb, shown as Figure 3 and Figure 4.

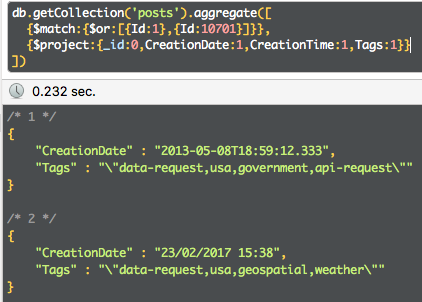


Figure 3. Original data

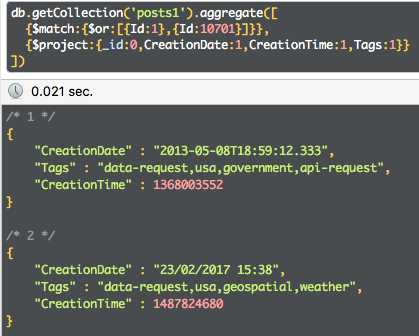


Figure 4. Processed data

## 2.2 Votes.csv

Similarly, we also added a CreationTime column according to the CreationDate in Votes.csv file. The python code is shown in Figure 5.

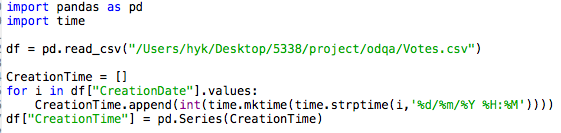


Figure 5. Convert CreationDate to CreationTime

Import the new csv file to mongodb and check the result by querying as Figure 6.

Figure 6. Processed data

# 3. Mongodb

## 3.1 Schema Design

### Indexing

An index on an attribute of a collection is a data structure that makes it efficient to find those required documents. An index consists of records (called index entries) each of which has a value for the attribute(s).

In this project, we created the following indexes to improve the query performance.



Figure 7. Indexing

### Decision day

In order to solve analytic query 5, firstly we used lookup to find questions’ accepted answers. Then lookup from vote to accepted answer and VoteTypeId=1. In this way we got decision day from the CreationTime of vote for questions. At last, generate a new collection posts2, which includes all questions that have decision day, by $out stage. The mongodb shell command is shown in Figure 8.



Figure 8. Generate posts2 collection

### Score

The score field in posts is very useful in analytic query 7. It indicates the total number of upvotes belong to the post. This feature is proved in schema design part of neo4j.

## 3.2 Query Design and Execution

### Simple query 1

Description: For each question (PostType=1), we identified their OwnerUserId and LastEditorUserId, these two are the direct users involved. As for answers to each question, they are picked up by their ParentId field, which declared they are answers to which question. By using $lookup stage, it connect the answers and the questions, and then link to the respective profile information from users collection by Id.

The query command is as Figure 9 shows.

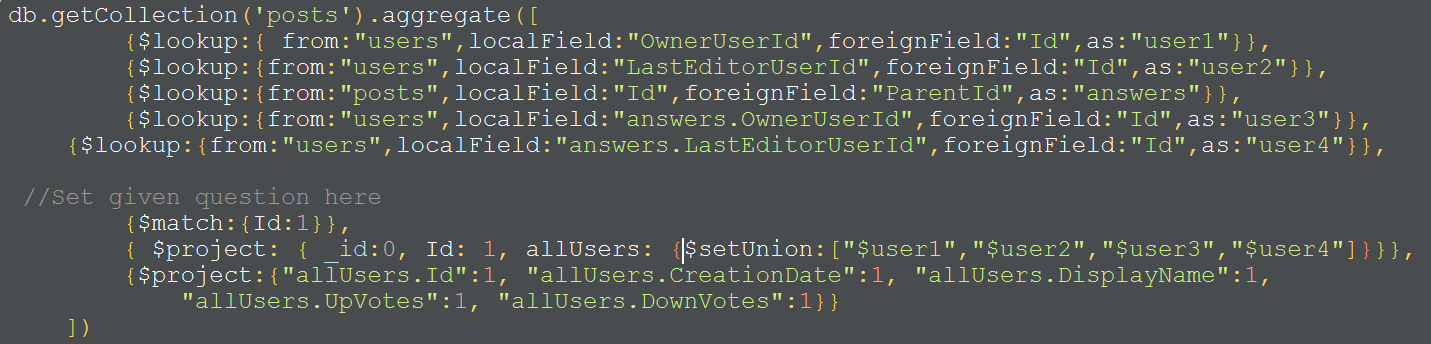


Figure 9. Simple query 1 command

And the result is shown as Figure 10.

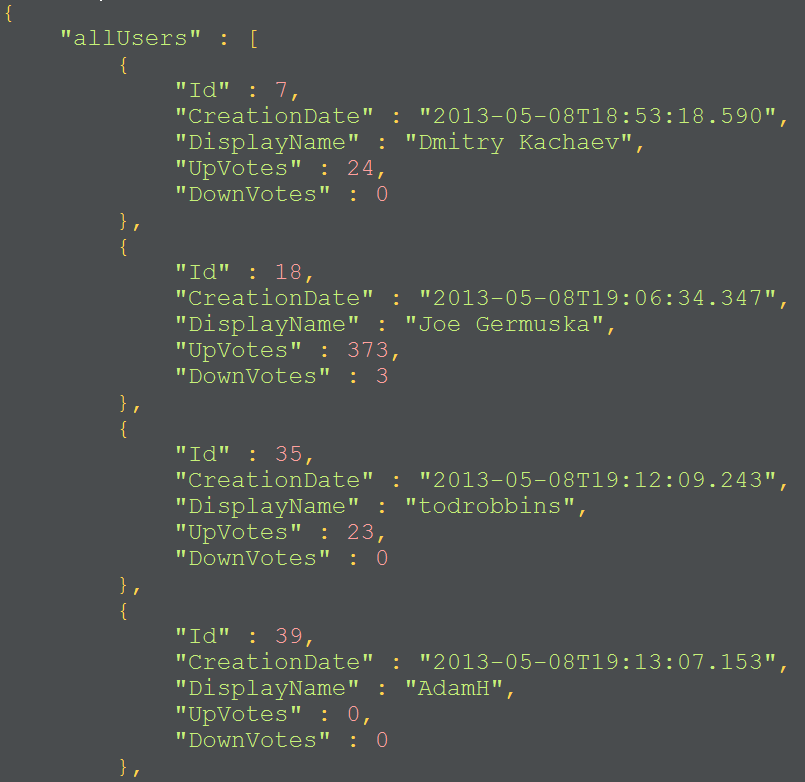


Figure 10. Simple query 1 result

### Simple query 2

In the posts1 collection, we split the tags and unwind them to individual ones(prepare all the topics). The filed ViewCount in each document represents how many times this post has been viewed, we solve the query by simply match the given topic, sort the ViewCount attribute by the descending order, the first shown document should be the post which has been viewd most. The query command is as Figure 11.

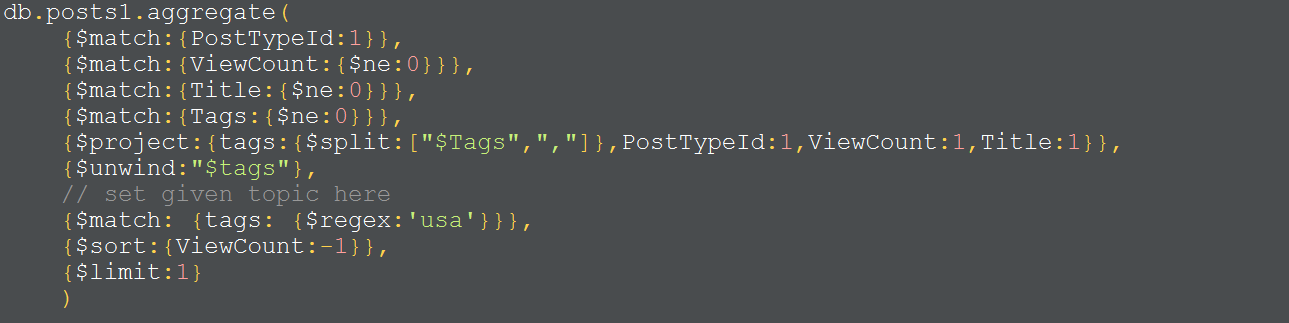


Figure 11. Simple query 2 command

And the result is shown like Figure 12.

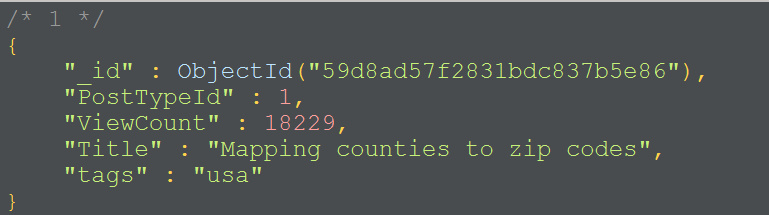


Figure 12. Simple query 2 result

### Analytic query 1

Query Design: The key part of this query, is to get the time difference of a question’s CreationTime and its corresponding answer’s CreationTime, after $lookup stage from posts1 collection, the time difference can be easily figured out by using the $subtract stage.

Execution: The query command is as Figure 13.

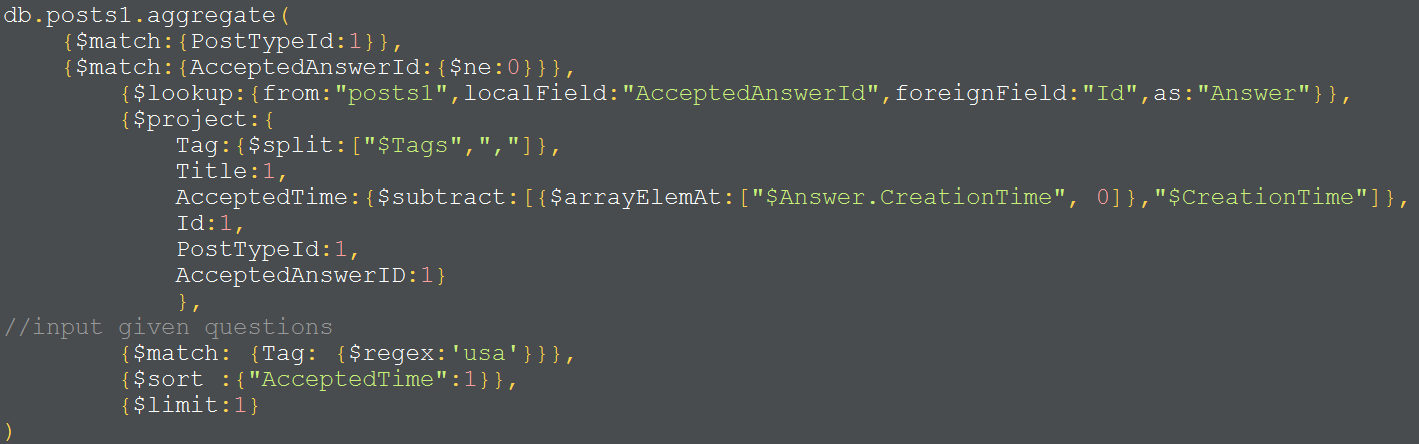


Figure 13. Analytic query 1 command

Performance: Without indexes, the operation time will be around 4.6 seconds. But after we created the indexes on PostTypeId, AcceptedAnswerId and Id in posts1 collection, the entire running time will decrease to 0.086 second. Indexes increase the efficiency evidently.

We used the command {explain:true}by adding it to the last line to observe the performance analysis. There is a “winnerPlan” attribute which contains a inputStage and filter stage. The inputStage applies index scan(IXSCAN) on the indexName “PostTypeId\_1”. The filter stage filtering documents based on the AcceptedAnswerId field, shown as Figure 14.



Figure 14. Analytic query 1 explain

### Analytic query 2

Query Design: Firstly, lookup from posts1 collection to get all the answers of each question. Secondly, unwind Tags and Answers to get each document contains one question, one answer and one Tag. Thirdly, match in a certain period and group by Tag. Finally, we got five hottest topics by using $sort and $limit.

Execution: query command in Figure 15.

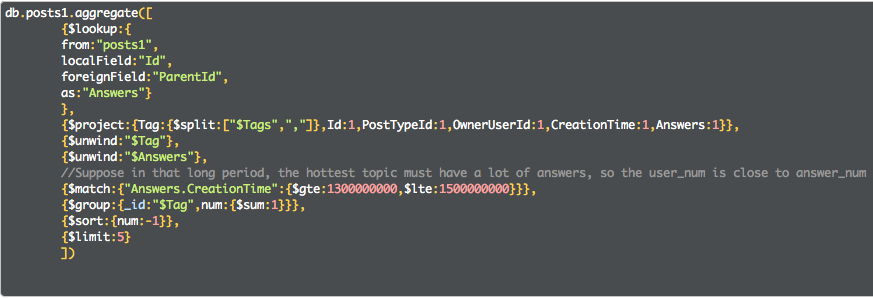


Figure 15. Analytic query 2 command

Performance: The result is shown in Figure 16. Using indexes, the running time is 0.72 second. Without index, the running time is much more than 0.72.

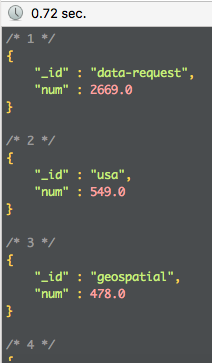


Figure 16. Analytic query 2 result

### Analytic query 3

Query Design: We divided the problem into two parts. First, find out the champion user.

Second, list questions in that topic which have accepted his answer.

Execution: In this query, we set the given topic to be “data-request” as an example. First part command as Figure 17.

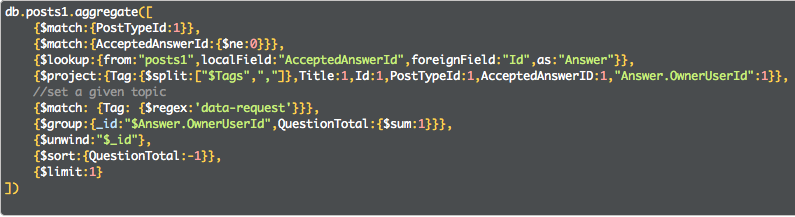


Figure 17. Analytic query 3(a) command

After running the command above, we found the champion user in topic “data-request” is the person whose user Id is 1511. As Figure 18 shows.

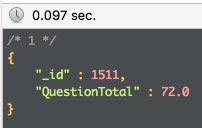


Figure 18. Analytic query 3(a) result

With the user Id, we can check all the questions in topic “data-request” which accepted his answer. Run the command as Figure 19.

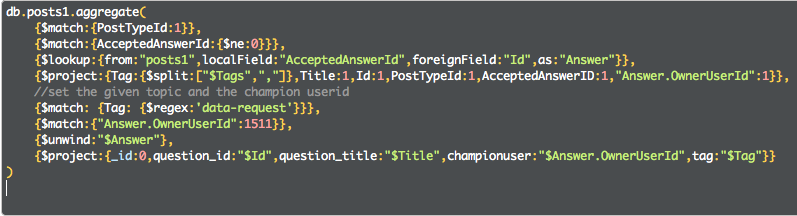


Figure 19. Analytic query 3(b) command

Result:

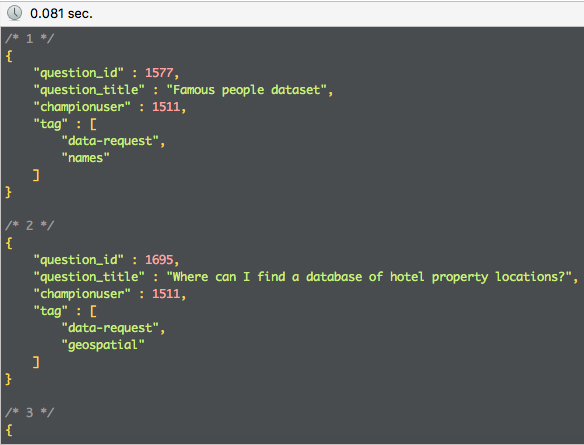


Figure 20. Analytic query final result

### Analytic query 4

Query Design: We divided the task into two steps. The first step is to find some potential users whose accepted answers number is larger than a threshold α.

Execution: In this query, we set α=30 as an example. Command as Figure 21.

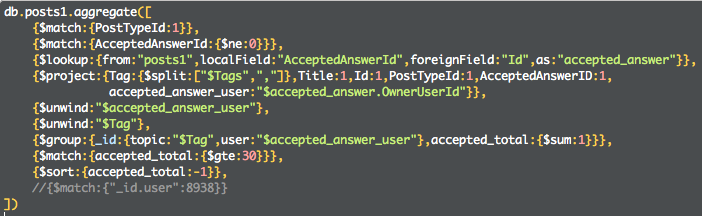


Figure 21. Analytic query 4(a) command

Result: I think each document in output should contain topic, user Id, and his/her total number of accepted answers. Example result as Figure 22 shows.



Figure 22. Analytic query 4(a) result

Step 2, select a user in the list. With the user Id and the topic, we can easily recommend 5 most recent unanswered questions where him/her are expert in. We matched all the questions which AcceptedAnswerId is null, matched the topic, sorted them by the descending order of CreationTime, limit 5. Query command as Figure 23.



Figure 23. Analytic query 4(b) command

Result:

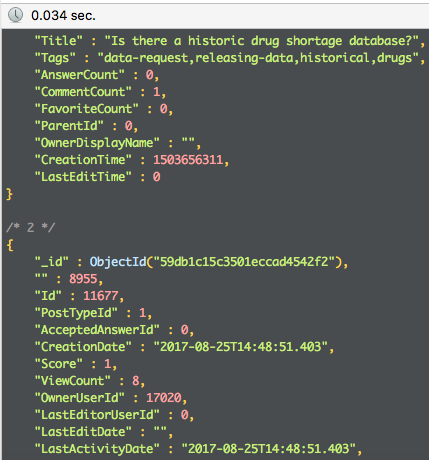


Figure 24. Analytic query 4 final result

### Analytic query 5

* Only consider the accepted answer

Firstly, as requested, match questions whoes total number of upVote is greater than or equal to a certain threshold value α. We set α to be 30.

Secondly, lookup from votes to get all votes to the accepted answer.

Thirdly, unwind votes and match VoteTypeId=2 and votes whose creationtime is later than decisionday because we only care about the upVotes after decision day.

Then we grouped by question\_id and AcceptedAnswer\_id, get percentage through dividing number of upVotes by score.

At last, we used $sort and $limit stages to get the highest percentage accepted answer.

Execution:

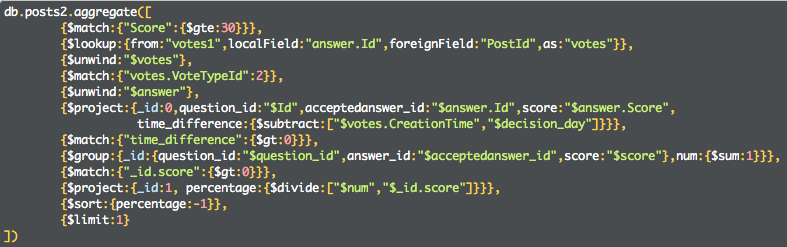


Figure 25. Analytic query 5(i) command

Result:

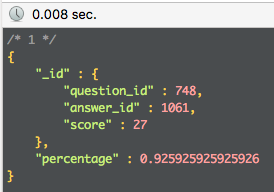


Figure 26. Analytic query 5(i) result

* Consider all other answers

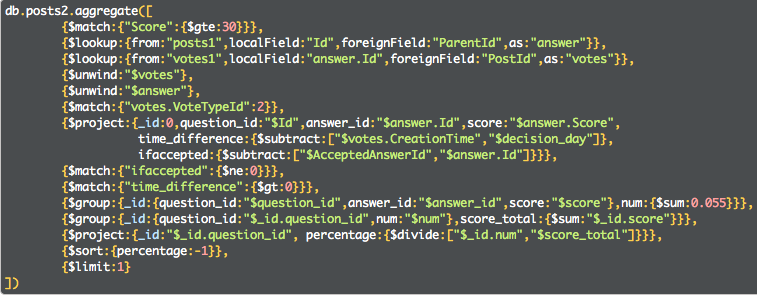
Execution: Similar to above command, run as Figure 27 shows. 

Figure 27. Analytic query 5(ii) command

Result:

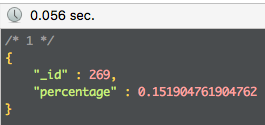


Figure 28. Analytic query 5(ii) result

### Analytic query6

Query Design: We combine a post’s owner and editor and its answer’s owners and editors together, as one set of related users. $lookup stage helps to link all the related posts as well as the users involved, then we can use $setUnion to hold them together as filed “involved\_users”. The match the given userID, unwind the field “involved\_users” to calculate the times involved with one user. Filter the pairs of itself and Id=0, sort them by the descending order and limit just 5 result.

Execution: The query command is as Figure 29 shows.

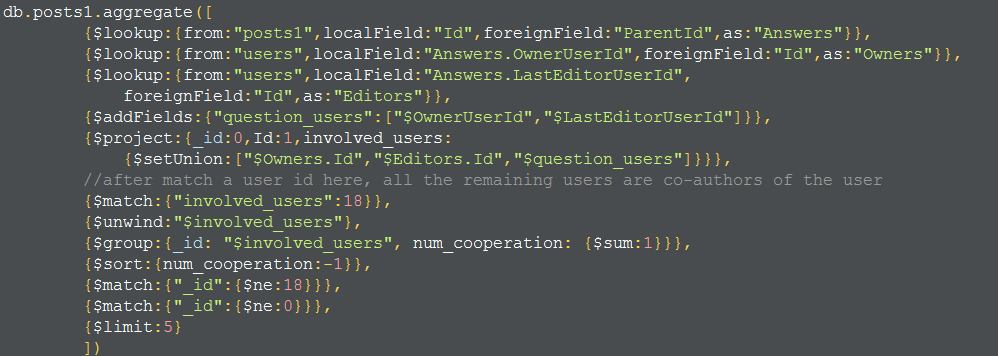


Figure 29. Analytic query 6 command

Result:

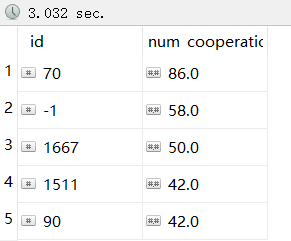


Figure 30. Analytic query 6 result

Performance: In this query, without using index will taking much more time to process the query, for more times all the documents need to be scanned. With indexes created in advance, all the query will be down in 3 seconds. It is much more efficient.

# 4. Neo4j

## 4.1 Schema design

### Nodes

* Create Post nodes (with 11 out of 19 properties we need to use in all queries)

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS

FROM "file:///Posts1.csv" AS line

CREATE(p:Post{Id:toInteger(line.Id),

Score:toInteger(line.Score),

PostTypeId:toInteger(line.PostTypeId),

AcceptedAnswerId:toInteger(line.AcceptedAnswerId),

CreationTime:toInteger(line.CreationTime),

OwnerUserId:toInteger(line.OwnerUserId),

LastEditorUserId:toInteger(line.LastEditorUserId),

Title:line.Title,

Tags:line.Tags,

ParentId:toInteger(line.ParentId),

ViewCount:toInteger(line.ViewCount)})

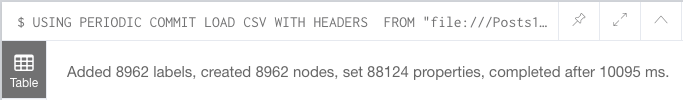


Figure 31. Create Post Label

* Tags need to be splitted so that we can unwind tags later

MATCH (p:Post)

SET p.Tags=split(p.Tags,',')

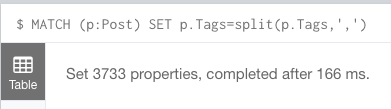


Figure 32. Set Post property

* Create User nodes (with 5 out of 11 properties we need to use in all queries)

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS

FROM "file:///Users.csv" AS row

CREATE(u:User{id:toInteger(row.Id),

creationdate:row.CreationDate,

displayname:row.DisplayName,

upvotes:toInteger(row.UpVotes),

downvotes:toInteger(row.DownVotes)})

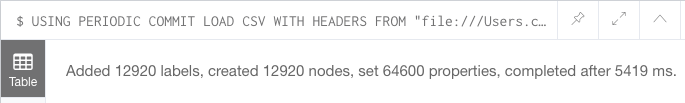


Figure 33. Create User Label

* Create Vote nodes (with 5 out of 7 properties we need to use in all queries)

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS

FROM "file:///Votes1.csv" AS line

CREATE(v:Vote{Id:toInteger(line.Id),

PostId:toInteger(line.PostId),

VoteTypeId:toInteger(line.VoteTypeId),

CreationTime:toInteger(line.CreationTime),

CreationDate:line.CreationDate})

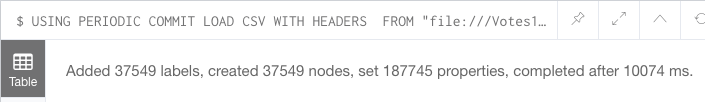


Figure 34. Create Vote Label

### Indexing

CREATE INDEX ON :Post(OwnerUserId)

CREATE INDEX ON :Post(LastEditorUserId)

CREATE INDEX ON :Post(Id)

CREATE INDEX ON :Post(ParentId)

CREATE INDEX ON :Post(AcceptedAnswerId)

CREATE INDEX ON :Post(DecisionDay)

CREATE INDEX ON :User(id)

CREATE INDEX ON :Vote(PostId)

CREATE INDEX ON :Vote(VoteTypeId)

CREATE INDEX ON :Vote(CreationTime)

### Create relationships

* Create OWNED relationship between Post and User

MATCH (p:Post),(u:User)

WHERE p.OwnerUserId = u.id

CREATE (p)<-[:OWNED]-(u)

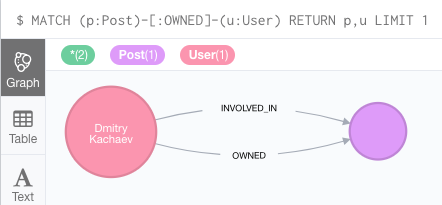


Figure 35. Create OWNED relationship

* Create EDITED relationship between Post and User

MATCH (p:Post),(u:User)

WHERE p.LastEditorUserId = u.id

CREATE (p)<-[:EDITED]-(u)

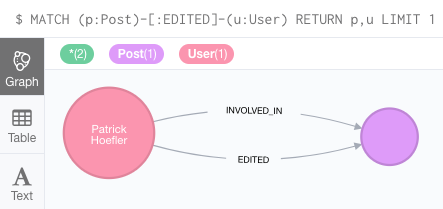


Figure 36. Create EDITED relationship

* Create ANSWERED relationship between Posts

MATCH (p1:Post),(p2:Post)

WHERE p2.ParentId=p1.Id

CREATE (p2)-[a:ANSWERED]->(p1)

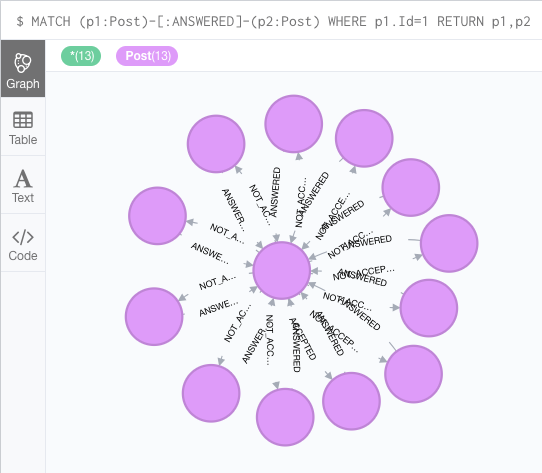


Figure 37. Create ANSWERED relationship

* Create ACCEPTED relationship with time property, which indicate the time it took to receive an accepted answer.

MATCH (p1:Post),(p2:Post)

WHERE p1.AcceptedAnswerId=p2.Id

CREATE (p1)-[a:ACCEPTED{time:p2.CreationTime-p1.CreationTime}]->(p2)

## 

Figure 38. Create ACCEPTED relationship

* Create NOT\_ACCEPTED relationship

MATCH (p1:Post)-[:ANSWERED]->(p2:Post)

WHERE NOT EXISTS(p2.AcceptedAnswerId) OR p1.Id<>p2.AcceptedAnswerId

CREATE (p1)<-[:NOT\_ACCEPTED]-(p2)

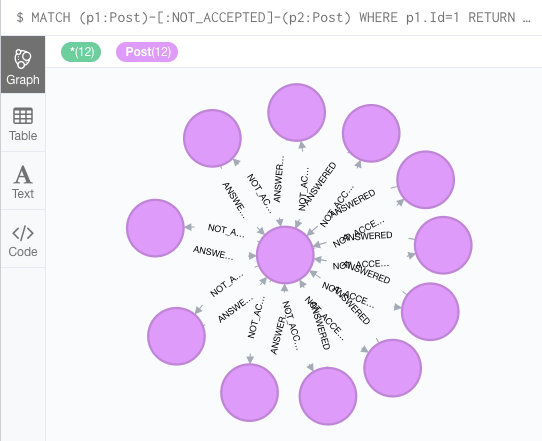


Figure 39. Create NOT\_ACCEPTED relationship

* Create UPVOTED relationship, with CreationTime property

MATCH (p:Post),(v:Vote)

WHERE v.PostId=p.Id AND v.VoteTypeId=2

CREATE (v)-[:UPVOTED{CreationTime:v.CreationTime}]->(p)

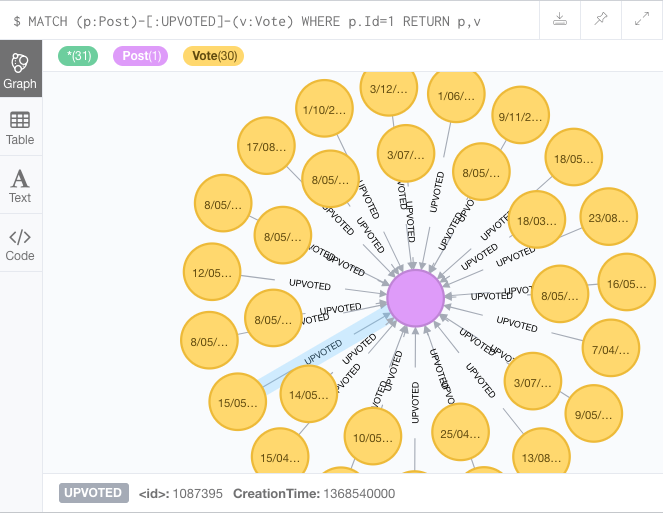


Figure 40. Create UPVOTED relationship

* Create DECIDED relationship, which is related to decision day

MATCH (p:Post),(v:Vote)

WHERE v.PostId=p.Id AND v.VoteTypeId=1

CREATE (v)<-[:DECIDED]-(p)

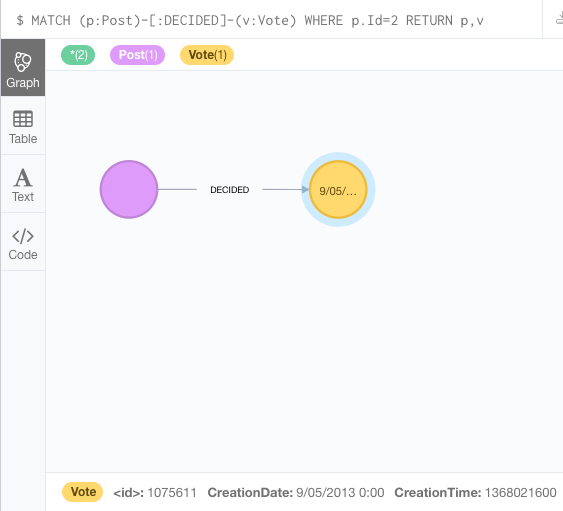


Figure 41. Create DECIDED relationship

* According to DECIDED relationship, we can add decision day property to Post.

MATCH (p:Post)-[d:DECIDED]-(v:Vote)

SET p.DecisionDay=v.CreationTime



Figure 42. Set DECIDED property

* Create INVOLVED\_IN relationship, which indicate all users related to a question.

MATCH (p1:Post)<-[:ANSWERED]-(p2:Post)

OPTIONAL MATCH (u1:User)-[:EDITED]-(p1)

OPTIONAL MATCH (u2:User)-[:OWNED]-(p1)

OPTIONAL MATCH (u3:User)-[:EDITED]-(p2)

OPTIONAL MATCH (u4:User)-[:OWNED]-(p2)

WHERE EXISTS(u1.id) OR EXISTS(u2.id) OR EXISTS(u3.id) OR EXISTS(u4.id)

WITH p1,(collect(u1)+collect(u2)+collect(u3)+collect(u4)) as users

UNWIND users as user

WITH p1,user

MERGE (p1)<-[:INVOLVED\_IN]-(user)

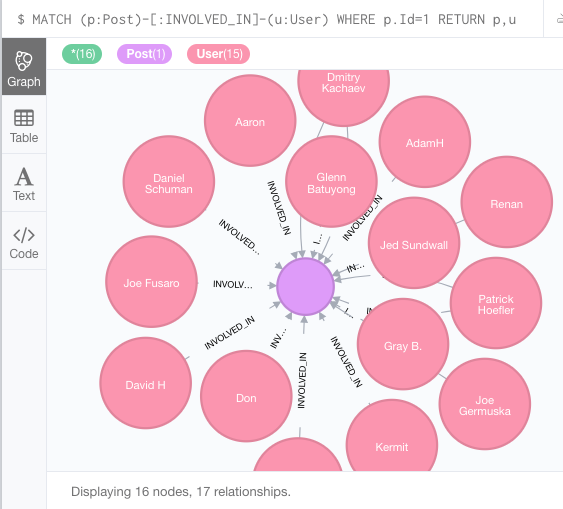


Figure 43. Create INVOLVED\_IN relationship

* In addition, I found that in most cases, the score property of Post indicates the total number of its upvotes. This feature will simplify the query. Proof of this feature:

MATCH (p:Post)-[:UPVOTED]-(v:Vote)

RETURN p.Id,p.Score,count(v) as num\_upvotes ORDER BY num\_upvotes DESC

LIMIT 20

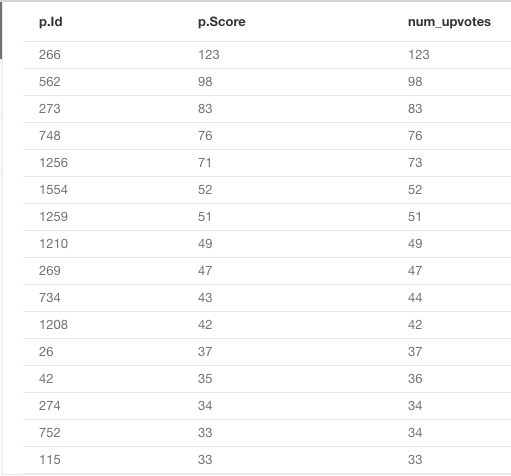


Figure 44. Proof of Score feature

## 4.2 Query design and execution

### Simple query 1

MATCH (p:Post)-[:INVOLVED\_IN]-(u:User)

WHERE p.Id=1

RETURN p,u

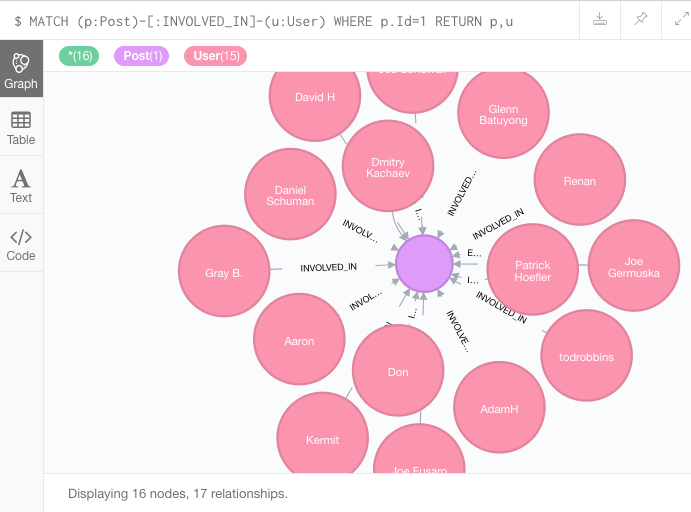


Figure 45. Simple query 1

### Simple query 2

MATCH (p:Post)

WHERE 'usa' in p.Tags

RETURN p.Title,p.ViewCount,p.Tags

ORDER BY p.ViewCount DESC

LIMIT 1

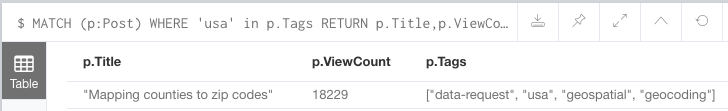


Figure 46. Simple query 2

### Analytic query 1

MATCH (p1:Post)-[a:ACCEPTED]->(p2:Post)

UNWIND p1.Tags AS Tag

WITH Tag,p1,a

WHERE Tag in ['usa','data-request']

RETURN DISTINCT Tag,p1.Title,a.time ORDER BY a.time LIMIT 2

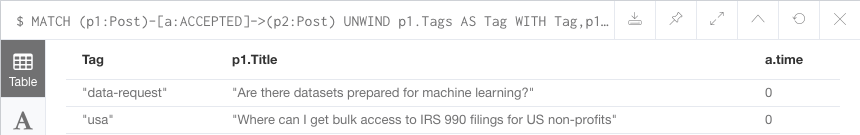


Figure 47. Analytic query 1

### Analytic query 2

MATCH (p1:Post)-[a:ANSWERED]-(p2:Post)

MATCH (p1)-[:OWNED]-(o1)

MATCH (p2)-[:OWNED]-(o2)

WHERE 1368004750 < p1.CreationTime < 1368008000

OR 1368004750 < p2.CreationTime < 1368008000

UNWIND p1.Tags as Tag

WITH Tag,p1,o1,o2

RETURN DISTINCT Tag,(count(o1)+count(o2)) as num\_users

ORDER BY num\_users DESC LIMIT 5

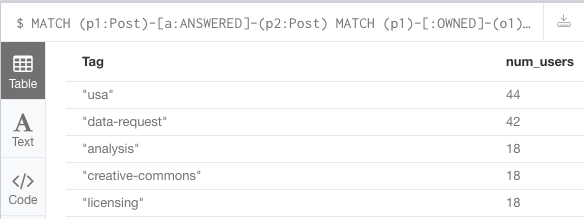


Figure 48. Analytic query 2

### Analytic query 3

Firstly, find the champion user in a certain topic. For instance, ‘data-request’.

MATCH (p1:Post)-[a:ACCEPTED]->(p2:Post)-[:OWNED]-(u:User)

WHERE 'data-request' in p1.Tags

RETURN u.id,count(u) as num\_accepted

ORDER BY num\_accepted DESC LIMIT 1

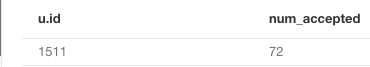


Figure 49. Analytic query 3a

Then use the user id to find all questions that his answer has been accepted by in that topic.

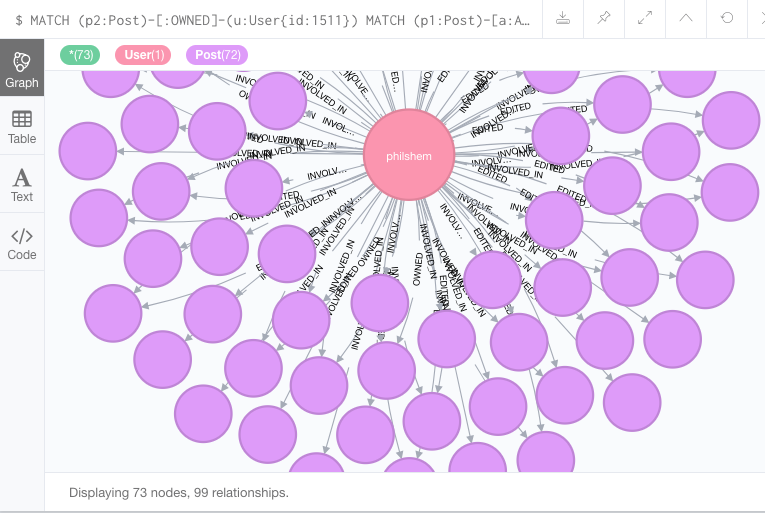


Figure 50. Analytic query 3b

### Analytic query 4

Firstly, find the potential users whose accepted answers are more than a threshold α=10.

MATCH (p1:Post)-[a:ACCEPTED]->(p2:Post)-[:OWNED]-(u:User)

UNWIND p1.Tags as Tag

WITH Tag,u,count(u) as num\_accepted

WHERE num\_accepted>10

RETURN DISTINCT Tag,u.id,num\_accepted ORDER BY num\_accepted DESC

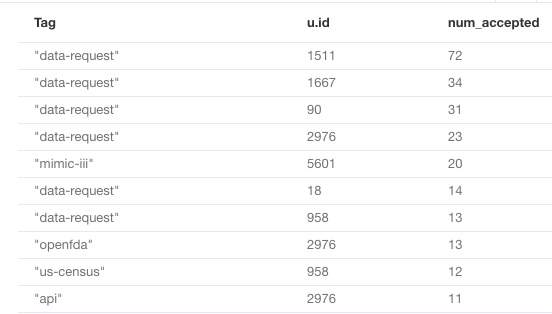


Figure 51. Analytic query 4a

Then choose a user and a certain topic to recommend 5 unanswered questions to him.

MATCH (p1:Post)

WHERE NOT EXISTS(p1.AcceptedAnswerId) AND 'data-request' in p1.Tags

RETURN p1 ORDER BY p1.CreationTime DESC LIMIT 5

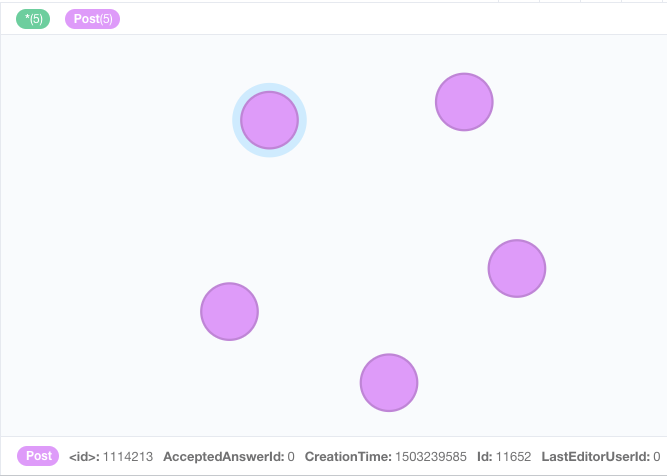


Figure 52. Analytic query 4b

### Analytic query 5

* Only consider the accepted answer

MATCH (p1:Post)-[:ACCEPTED]-(p2:Post)-[up:UPVOTED]-(v:Vote)

WHERE p1.Score >= 30 AND up.CreationTime > p2.DecisionDay

RETURN DISTINCT p1.Id as question\_id,p2.Id as accepted\_answer\_id, count(v)\*100/p2.Score as percentage ORDER BY percentage DESC LIMIT 1

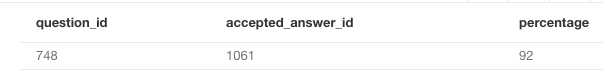


Figure 53. Analytic query 5a

* Consider all the other answers

MATCH(p3:Post)-[:ACCEPTED]-(p1:Post)-[:NOT\_ACCEPTED]-(p2:Post)-[up:UPVOTED]-(v:Vote)

WHERE p1.Score>30 AND up.CreationTime > p3.DecisionDay AND p2.Score>0

RETURN DISTINCT p1.Id,count(v) as upvotes\_after,sum(p2.Score) as upvotes\_total,count(v)\*100/sum(p2.Score) as percentage ORDER BY percentage DESC LIMIT 1

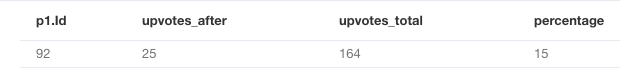


Figure 54. Analytic query 5b

### Analytic query 6

MATCH (u1:User{id:1511})-[:INVOLVED\_IN]->(p1:Post)

MATCH (u2:User)-[:INVOLVED\_IN]->(p1)

WHERE u2.id<>u1.id AND u2.id<>0

RETURN u2.id,count(u2) as num\_cooperation ORDER BY num\_cooperation DESC LIMIT 5

### 

Figure 55. Analytic query 6

# 5. Comparison and Summary

Through this project, we found both mongodb and neo4j have their own advantages and weakness.

For mongodb: it stores data in collection, it is easy to generate or drop a collection. And we can use $lookup to connect different collections. Since it is based on javascript, sometimes we could update collections using js. For example, write js function to change data type. I think it is convinient. But this database require more complex logic in each query, in my opinion.

For neo4j: it is a graph database so the data is visible, I like it. And the best thing is that after we build good relationships, the queries will become much more simple.

For indexing: with creating indexes, both two databases become more efficient.

In conclusion, in the process of completing this project, we faced many difficulties such as data preprocessing by python, schema design. Because logic in mongodb and neo4j is quite different sometimes. We spent a lot of time on this project. At last, we solved most of the problems. Feel good.