# Section 1 – Introduction

In this assignment, we showed that we can work with different NoSQL systems for data persistence and understand the strength and weakness of each system with respect to certain query workload features. We worked with a Question and Answer data set and a list of target queries. The target queries include very basic OLTP type queries and analytic queries. We designed data schema based on the data set feature and the given query workload for MongoDB and Neo4j respectively. We showed that our design can support the target queries by loading the data in each system following schemas and by running queries against the data.

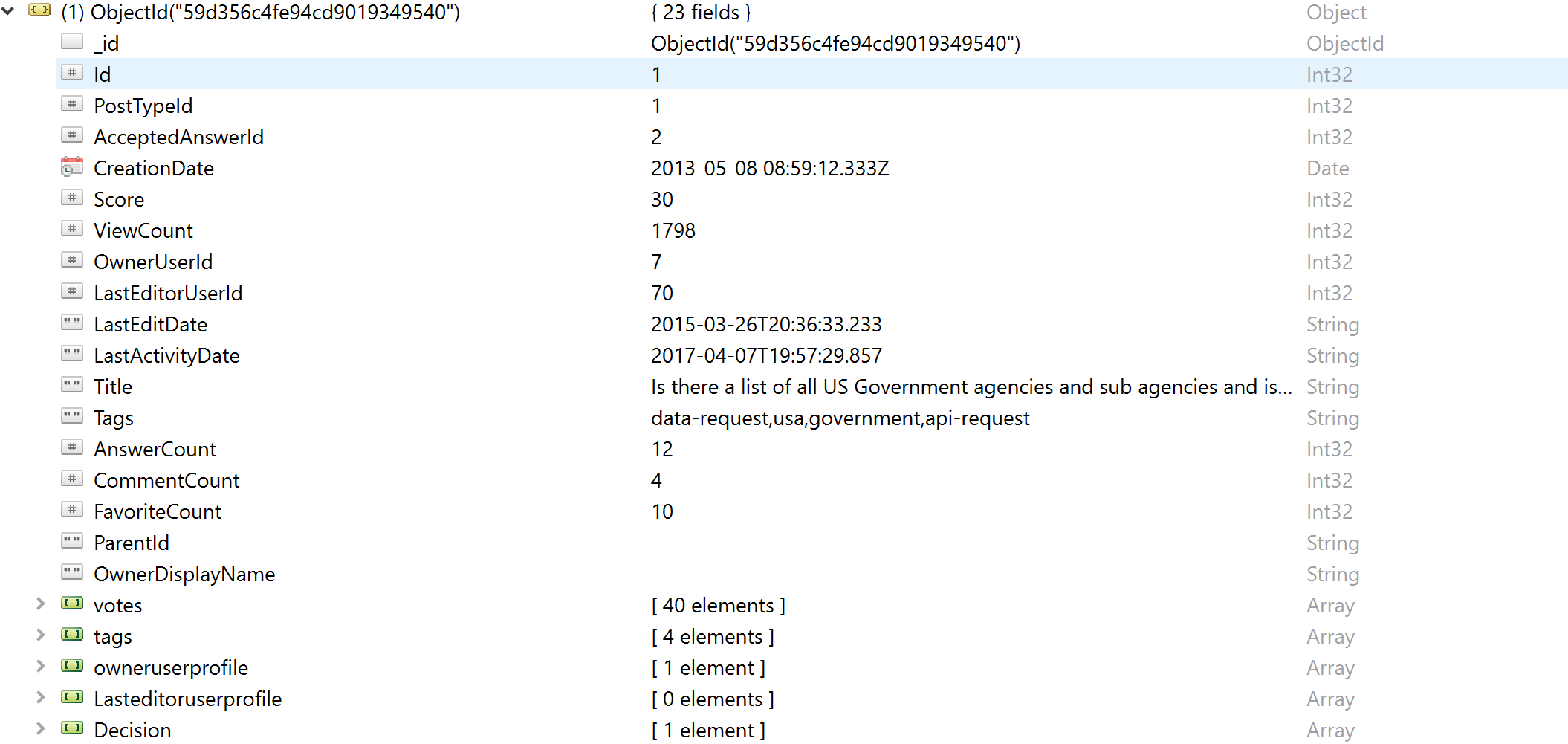
# Section 2 – MongoDB schema and query design

# **Data Pre-Processing:**

1. First we ensure that the date values in all the CSVs are in a common format “yyyy-mm-ddThh:mm:ss.000”.
2. The tags field in the posts csv was in the form “tag1,tag2,tag3”. We removed the quotes so that it would be easy for us to store the tags in an array form.

# **Schema Design:**

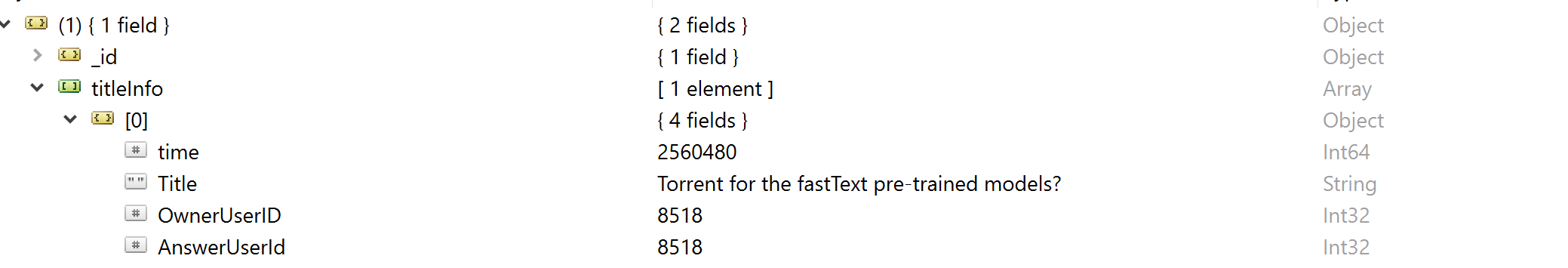
1. Collection Name: “Combined”



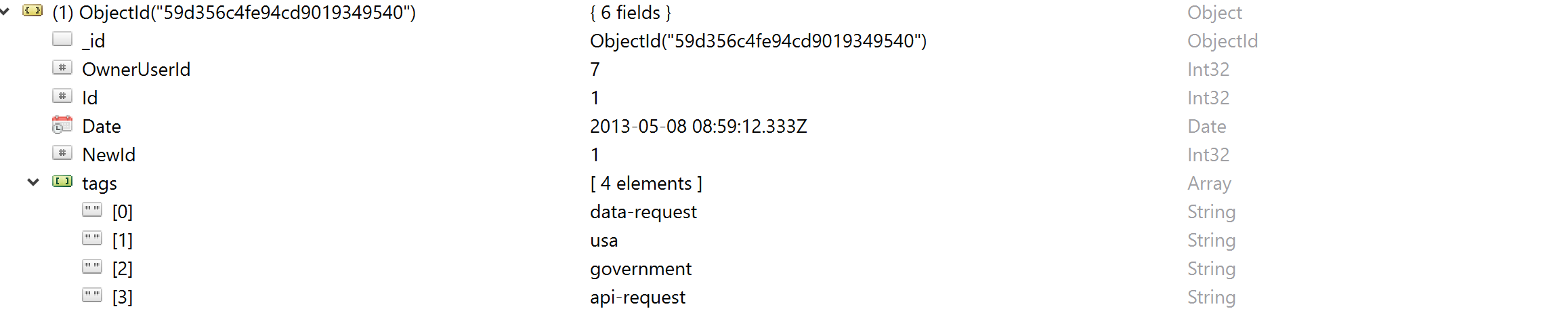
This collection has all the information for each post. In addition it also has the Votes(from Votes), owneruser(from User) details and Decision date(Calculated from Votes ) stored as embedded values. Indexing was applied to the Id field :

db.Combined.createIndex({Id:1})

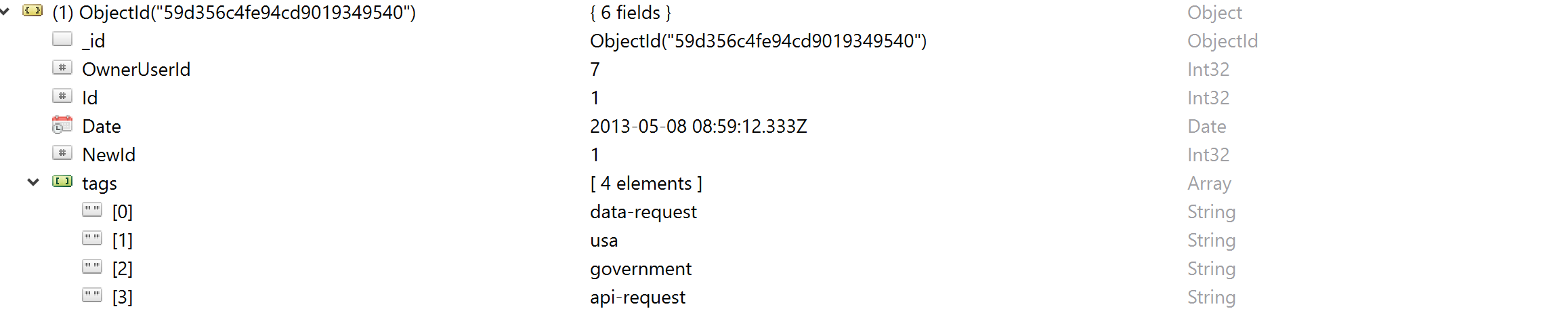
1. Collection Name: ”TagTime”



This collection stores the tag and the time taken for the fastest answer for each question.



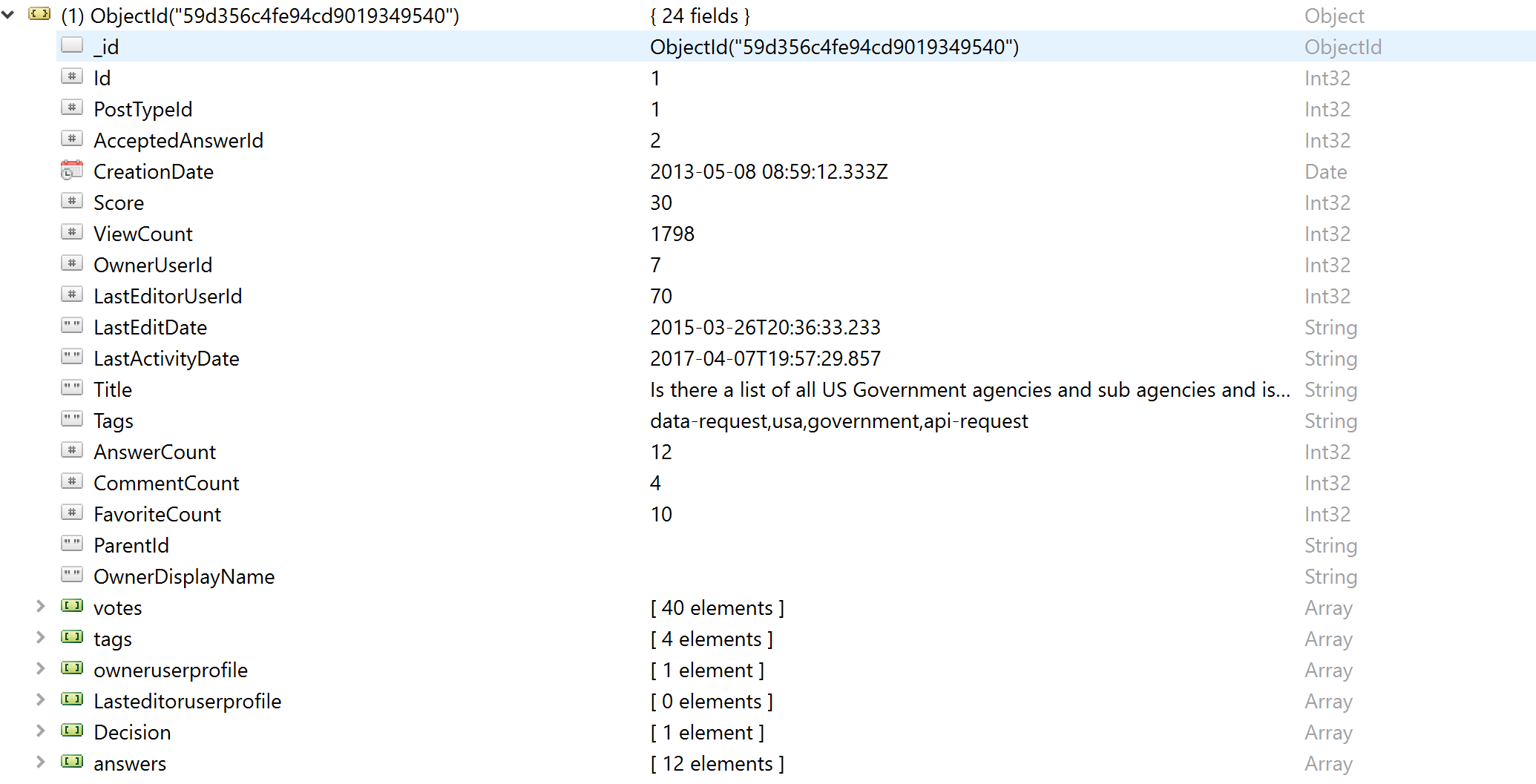
1. Collection Name: “PostAnlytical”



This Collection has a new field called NewId. All questions and the respective answers share the same newID. This is done so that we can perform group operations on questions and answers at the same time. Indexing was applied to the NewId field:

db.PostAnalytical.createIndex({NewId:1})

1. Collection Name: AuthorCollection:



This collection has all answers embedded for each question row.

1. Collection Name: Combined1



# **Simple Queries**:

1. DB.combined.aggregate([

{"$match":{"$or":[{"Id":id},{"ParentId":id}]}},

{"$unwind" : "$owneruserprofile"},

{"$group" : { "\_id" :

{"OwnerID":"$owneruserprofile.Id","OwnerName":"$owneruserprofile.DisplayName","CreationDate":"$owneruserprofile.CreationDate","Upvote":"$owneruserprofile.UpVotes","Downvote":"$owneruserprofile.DownVotes"}}},

{"$sort":{"\_id.OwnerID":1}},

])

Description:

In the collection Combined. Each Post will have the User details stored as an embedded data which is mapped with the owneruserid. After we filter using the userid we group the userdetails.

1. Db.combined([

{"$match":{"PostTypeId":1}},

{"$unwind" : "$tags"},

{"$match":{"tags":”open-access”}},

{"$group" :{"\_id" : {"Title":"$Title","Topics":"$tags","Postid":"$Id","Viewcount":{"$max":"$ViewCount"}}}},

{"$sort":{"\_id.Viewcount":-1}},

{$limit:1}

])

Description:

The tags are stored as an array in the collection, therefore we have to uunwind it after filtering out the questions using the PostTypeId.After filtering again with the user provided tag name, we will group the records on the title and select the field with the maximum view count.

# **Analytical** **Queries**:

1. Db.TagTime.aggregate([

{"$match":{"\_id.Topic":{"$in":list(tags)}}},

{"$unwind":'$titleInfo'},

{"$sort":{"titleInfo.time":1}},

{"$group":{

"\_id":"$\_id.Topic","titleDetails":{"$first":"$titleInfo"}

}}

])

Description:

In this query, we filter based on a list of topics. After which we unwind the titleInfo field as it is an array. Then we sort by the time taken for the answer and then group by the tag and display the first question.

1. db.PostAnalytic.aggregate(

[

{$unwind : "$tags"},

{

"$match":

{

"Date":

{

$gt: ISODate("2012-04-29T00:00:00.000Z"),

$lt: ISODate("2014-05-01T00:00:00.000Z")

}

}

},

{$group: {\_id:"$NewId",Owners:{$addToSet:"$OwnerUserId"},tags:{$addToSet:"$tags"}

}

},{$unwind : "$tags"},

{$unwind:"$Owners"},

{$match:{tags:{$ne:""}}},

{$group:{\_id:"$tags",Owners:{$addToSet:"$Owners"}}},

{$project:{id:1,countofusers:{$size:"$Owners"}}},

{$sort:{countofusers:-1}}

])

Description:

Since we require only distinct owners and tags we add them to a set, after which we unwind the tags. Now we will have the owners for each tag. Since we need the count of users, we group by the tags and add the size of the owner list.

1. db.TagTime.aggregate([

{$unwind:"$titleInfo"},

{$match:{"\_id.Topic":"usa"}},

{

   $group:{

       \_id:{"Owner":"$titleInfo.AnswerUserId"},count:{$sum:1},questions:{"$push":"$titleInfo.Title"}

   }

},

{$sort:{"count":-1}},

{$limit:1}

])

Description:

Initially, we filter based on the tag then we group by the AnswerId , push the titles and get the count of rows. This will give us the Champion user for the given tag and the titles which the user has his/her answers accepted.

1. Query 1:

db.TagTime.aggregate([

{$unwind:"$titleInfo"},

{$match:{"titleInfo.AnswerUserId":1511}},

{

   $group:{

       \_id:{"Tag":"$\_id.Topic"},count:{$sum:1}

   }

   },

{$match:{"count":{$gte:5}}},

{$sort:{"count":-1}}

])

Query 2:

db.Combined.aggregate([

{$match:{AcceptedAnswerId:""}},

{$match:{PostTypeId:1}},

{$unwind:"$tags"},

{$match:{"tags":{$in:["data-request","usa","open-access"]}}},

{$sort:{CreationDate:-1}},

{$limit:5}

])

Description:

To solve this answer, we have split it into two queries. The first query finds the tags where the given user has a count of questions contributed to a tag that is greater than the threshold value. Then we pass the tags as the input for the next query which finds the latest 5 unanswered questions that belong to the tags that were obtained in the previous query.

1. db.Combined1.aggregate([

{$unwind:"$votes"},

{$unwind:"$Decision1"},

{$match:{"PostTypeId":2}},

{$match:{"votes.VoteTypeId":2}},

{$project:{\_id:"$ParentId",time:{$subtract:["$votes.CreationDate","$Decision1.decisiondate"]}}},

{$match:{"time":{"$gte":0}}},

{$group:{\_id:"$\_id",count:{$sum:1}}},

{$sort:{"count":-1}}

])

Description:

We are assuming that the VoteTypeId:2 refers to the upvote. Since we have to calculate the number of upvotes for answers, we are filtering for objects with VoteTypeId 2 and PostTypeId 2. After Which, we subtract the Creation Date and Decision date to find out which is greater. We then count the votes where the Creation Date is greater than the Decision date.

1. db.coauth.aggregate([

{$unwind:"$answers"},

{$match:{"OwnerUserId":1}},

{$group:{\_id:{author:"$OwnerUserId",coauthor:"$answers.OwnerUserId"},count:{$sum:1}}},

{$sort:{"count":-1}},

{$limit:5}

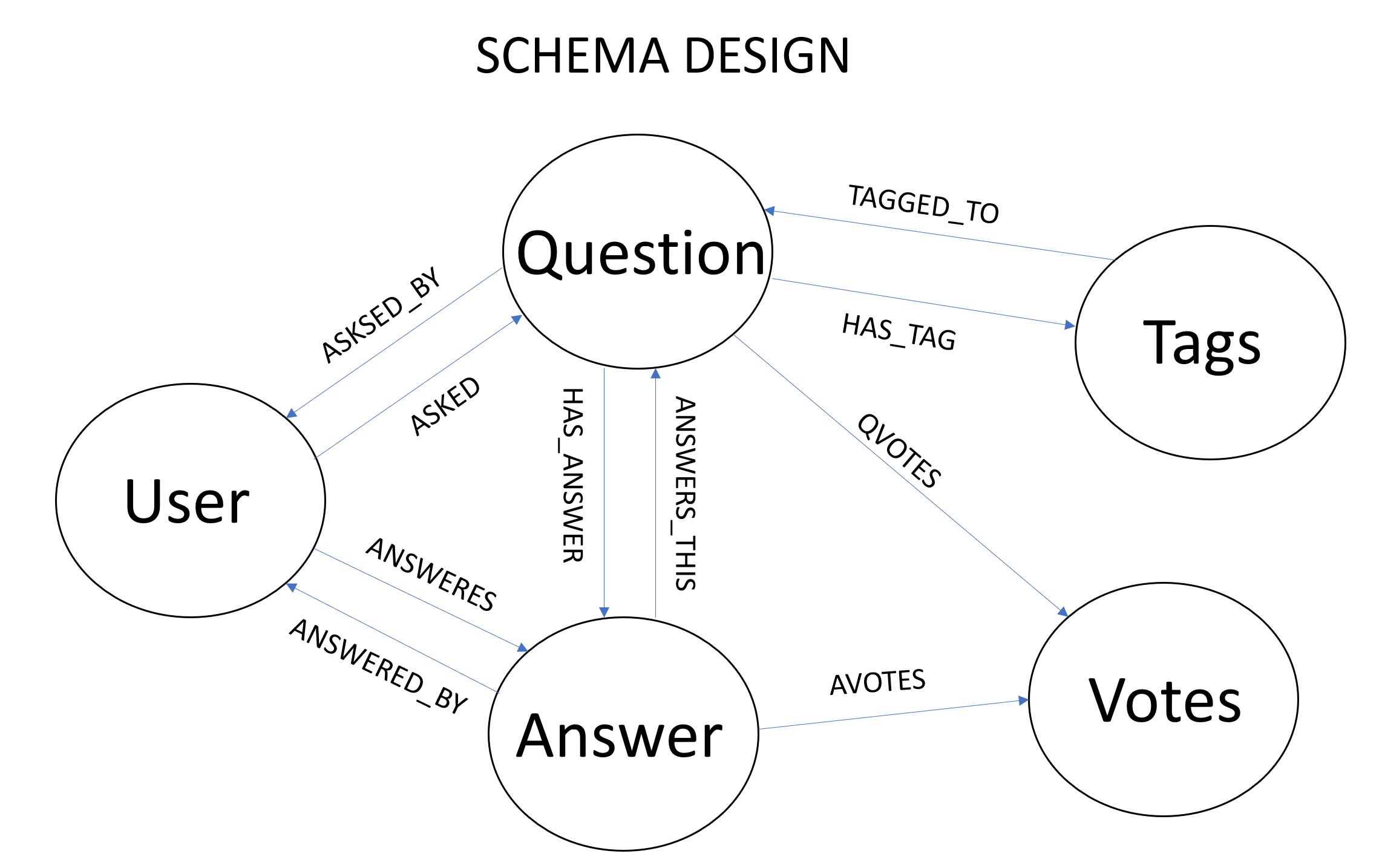
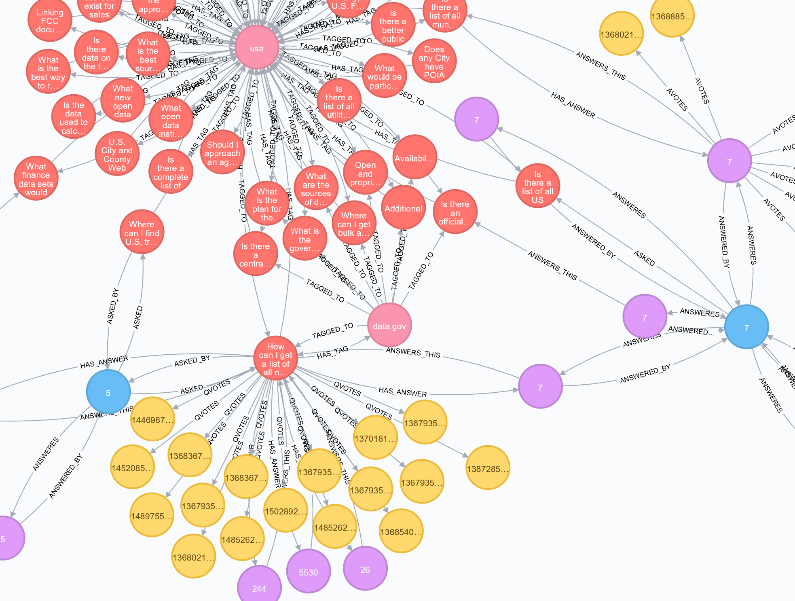
])

Description:

We first unwind the “Answers” array. We then pass the owneruserId as a parameter. After which, we group by a combination of OwnerUserId and AnswererUserID, and provide a count. We then sort the count in descending order and limit it by 5 to get the top 5 Co-authors.

# Section 3 – NeoGraphDB schema and query design

## Schema Design:

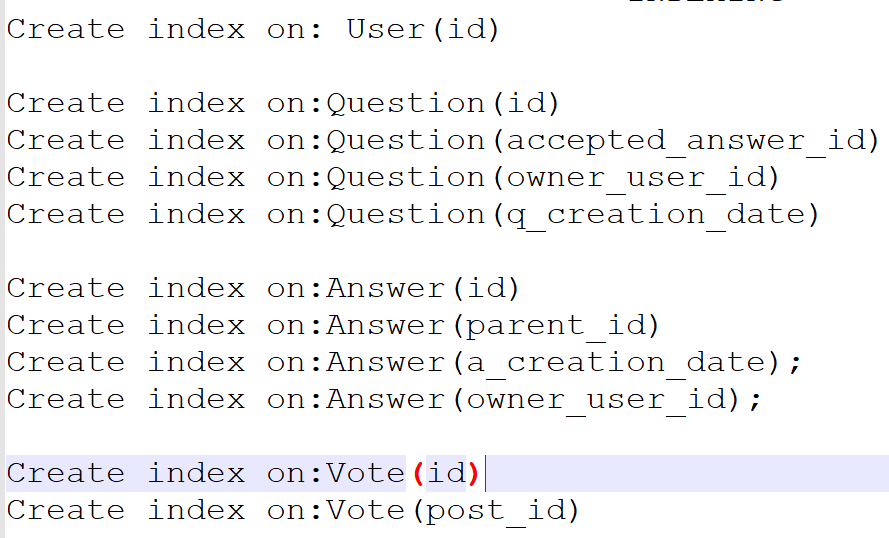


The above diagram represents our schema design and sample data for neo4j. To support this schema, we had to preprocess the input data. We split the Posts.csv into Questions.csv and Answer.csv. The reason behind this is that, in a graph DB it will be more easier and efficient to traverse between question and answer nodes instead of having both in the same node. Also, we can easily map the users, votes to answers and questions making it a loop so cursor can traverse in any direction making query processing easier.

Also, we converted the Date format to unix seconds to make operations on time field easier. The detailed steps of preprocessing are given in Neo4j Readme document attached in the ZIP.

We have made the the ‘Id’ fields as primary\_keys in each of the respective Question, Answers, User, Votes and Tag Nodes.

After creating the nodes, we have put an index on the following:

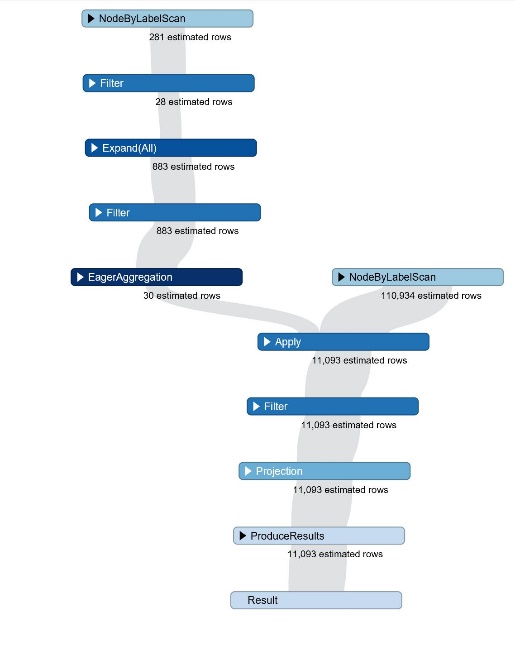


## C:\Users\lohit\AppData\Local\Microsoft\Windows\INetCache\Content.Word\sim1.jpgQuery Design and Execution:

Simple Query:

1. Match (ques:Question{id:"7"})-[:ASKED\_BY]->(n:User),(ans:Answer{parent\_id:"7"})-[:ANSWERED\_BY]-(u:User)

return distinct n,u

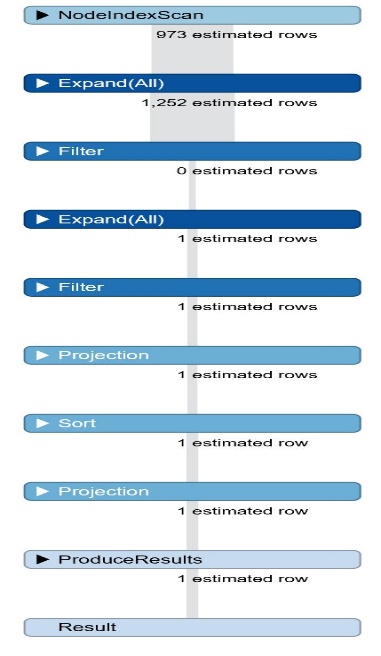
1. Match p=(t:Tag{tag\_name:'data-request'})-[:TAGGED\_TO]->(q:Question) with max(toInteger(q.view\_count)) as max\_views

match x=(ques:Question)

where toInteger(ques.view\_count)=max\_views

return toInteger(ques.view\_count), ques.title;

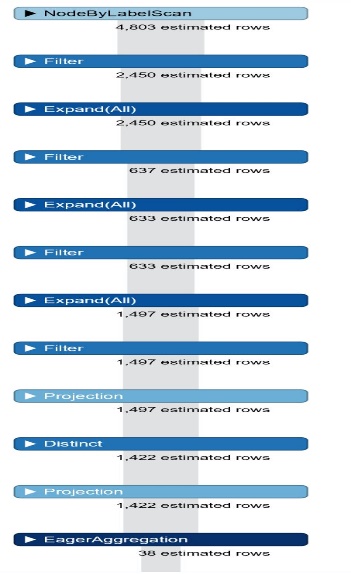
### Analytical Query

1. Match p=(t:Tag)-[:TAGGED\_TO]->(q:Question)-[:HAS\_ANSWER]->(a:Answer)

where t.tag\_name IN['usa','fcc','api','city']

and exists(q.accepted\_answer\_id) and q.accepted\_answer\_id=a.id

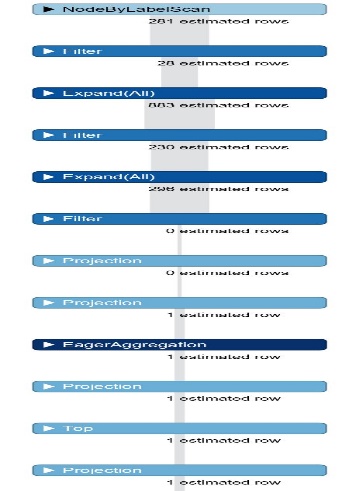
return toInteger(a.a\_creation\_date)-toInteger(q.q\_creation\_date) as diff,q.title,q.id,t.tag\_name order by diff

1. Match (t:Tag)-[:TAGGED\_TO]->(q:Question)-[:HAS\_ANSWER]->(a:Answer)-[:ANSWERED\_BY]->(u:User)

where (toInteger(q.q\_creation\_date)>= 1357027578) and (toInteger(q.q\_creation\_date)<= 1388563578) and (toInteger(a.a\_creation\_date)>= 1357027578) or (toInteger(a.a\_creation\_date)<= 1388563578)

with distinct u.id as person,(t.tag\_name) as topic

return topic, count(topic) as no\_users order by no\_users desc limit 5

1. match (t:Tag{tag\_name:"api"} )-[:TAGGED\_TO]->(q:Question),(q:Question)-[:HAS\_ANSWER]->(a:Answer)

where exists(q.accepted\_answer\_id) and (q.accepted\_answer\_id= a.id)

//match (a:Answer)-[:ANSWEREDBY]->(u:User)

with (a.owner\_user\_id) as User, q.title as ques

return User,count(User) as total, collect(distinct ques) [0..] as Allques

order by total desc limit 1

1. MATCH p=(u:User)-[r:ANSWERES]->(a:Answer)-[:ANSWERS\_THIS]->(q:Question)-[:HAS\_TAG]->(t:Tag)

where u.id="1511" and u.id=a.owner\_user\_id and a.parent\_id=q.id and a.id=q.accepted\_answer\_id

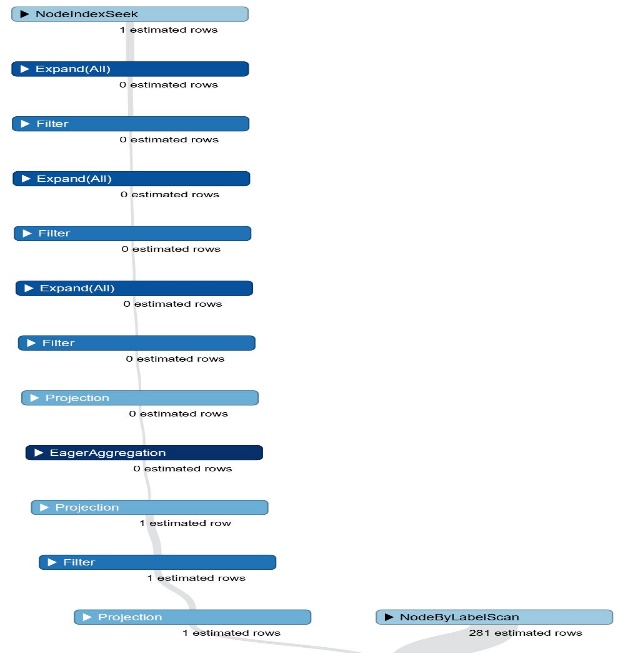
with t.tag\_name as tag, count(t.tag\_name) as tag\_count

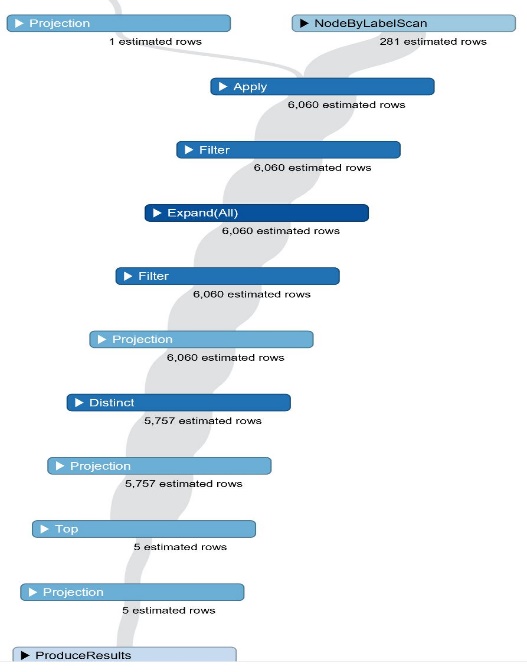
where tag\_count >= 5

with tag as top\_tags

match o=(t1:Tag)-[:TAGGED\_TO]->(q1:Question)

where t1.tag\_name in top\_tags and not exists(q1.accepted\_answer\_id)

with distinct q1.title as quest, q1.q\_creation\_date as created\_date

return quest, created\_date order by created\_date desc limit 5;

1. match(u:User)-[:ANSWERES]->(a:Answer)-[:ANSWERS\_THIS]->(q:Question)-[:HAS\_ANSWER]->(a1:Answer)-[:ANSWERED\_BY]->(u1:User)

where (a.owner\_user\_id="1511")

with collect({uid:u1.id}) as rows

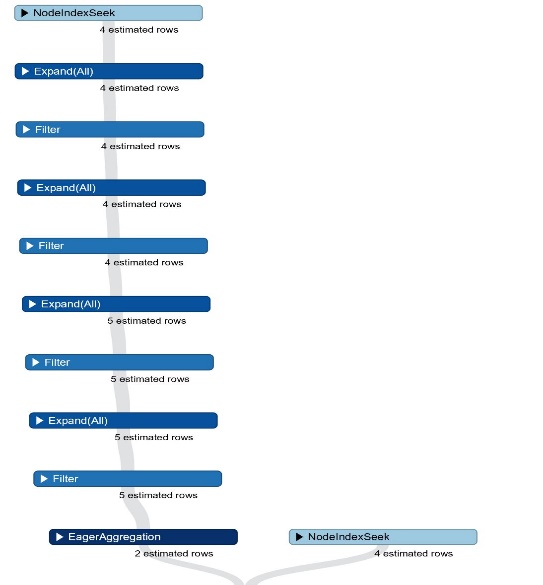
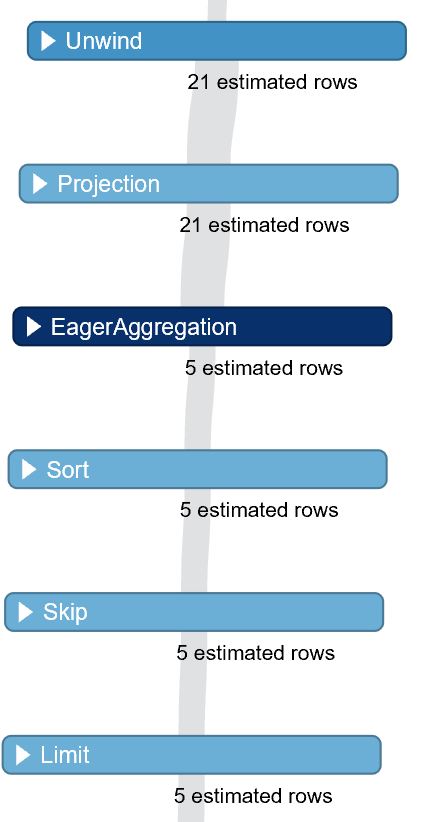
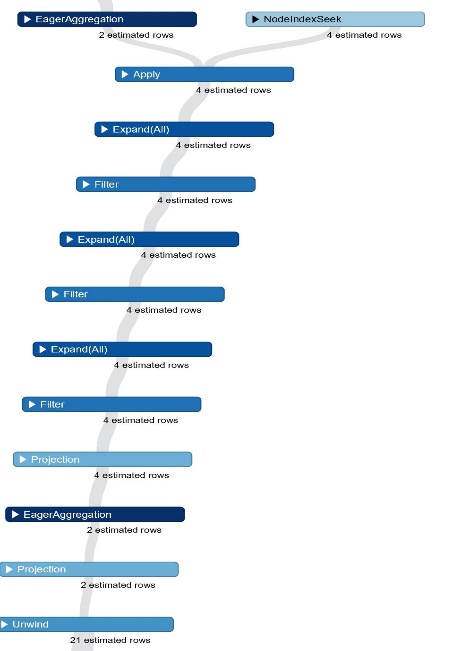
match(u2:User)-[:ASKED]->(q1:Question)-[:HAS\_ANSWER]->(a2:Answer)-[:ANSWERED\_BY]->(u3:User)

where (q1.owner\_user\_id="1511")

with rows + collect({uid:u3.id}) as all\_rows

unwind all\_rows as row

return row.uid, count(row.uid) as couser\_count order by couser\_count desc skip 1 limit 5;



# Section 4-Comparison and Evaluation

# Execution Times

|  |  |  |
| --- | --- | --- |
|  | MongoDB Execution Times in secs | Neo4J Execution Times secs |
| Query 1 | 0.057 | 0.001 |
| Query 2 | 0.027 | 0.015 |
| Query 3 | 0.005 | 0.047 |
| Query 4 | 0.009 | 0.088 |
| Query 5 | 0.001 | 0.005 |
| Query 6 | 0.062 | 0.052 |
| Query 7 | 0.009 |  |
| Query 8 | 0.047 | 0.035 |

**Challenges:**

Mapping between rows in a collection is hard. We had to create an embedded data every time we wanted to insert question details in for answers or Answer details for questions..It is also hard to compare two field values from a collection at the same time. When doing group operations including non-accumulator fields was not possible.

**Ease of Use:**

MongoDB Queries was easier to implement when we had to use multiple group statements. Handling Dates was much easier in MongoDB with the use of the ISODate.

**Conclusion:**

While Running Complex queries Neo4j was faster and easier to use, Especially While working with data that can be structured as a tree.