Comp5338 NoSQL Schema Design and Query Workload Implementation

Group: Lab-114-Tue 8
Group members: Zhizhi Chai(470259267), Xin Ning(460346559)

1.Introduction

·Brief Introduction

This report is used for “NoSQL schema design and query workload implementation” and which would be divided into five parts as the following sequence: 1. brief introduction of the report; 2.design and analysis: this part would contain three subsections which contains query workload, schema design, query design and execution. 3. Comparison of two system, including schema differences and query performances. 4. The last part would contain sample query result and respective arguments.

·Background

The inventor of MongoDB wanted to design a document storage database with high scalability, flexibility, ultimate consistency and lightning speed (Membrey, Hows, & Plugge, 2014).

Neo4J is designed to solve the main performance bottlenecks of RDBMS, containing the advantages of the flexibility of mode, the speed of the query (Jordan, 2014).

2. Design and analysis

·Background: We implemented all eight target queries for each storage system option and chose relatively suitable query workloads for each one.

2.1 Query workload

2.1.1 MongoDB workload

·Briefly analyze

The advantage of MongoDB is that it has a good document structure storage method, which makes it easier to obtain data. For AQ2, we have to implement it by MongoDB, or the data model would be so complex that contain cross-relationships of Posts, Tags and Users and it would cause low efficiency. We did not choose AQ3 and AQ5 for MongoDB because both have two lookups and they would affect the performance. And AQ6 has 3 graph lookup, which would lead to poor performance in time efficiency and code complexity.

·query workload

MongoDB query set :{SQ1, SQ2, AQ1, AQ2, AQ4}

2.1.2 Neo4j workload

·Briefly analyze

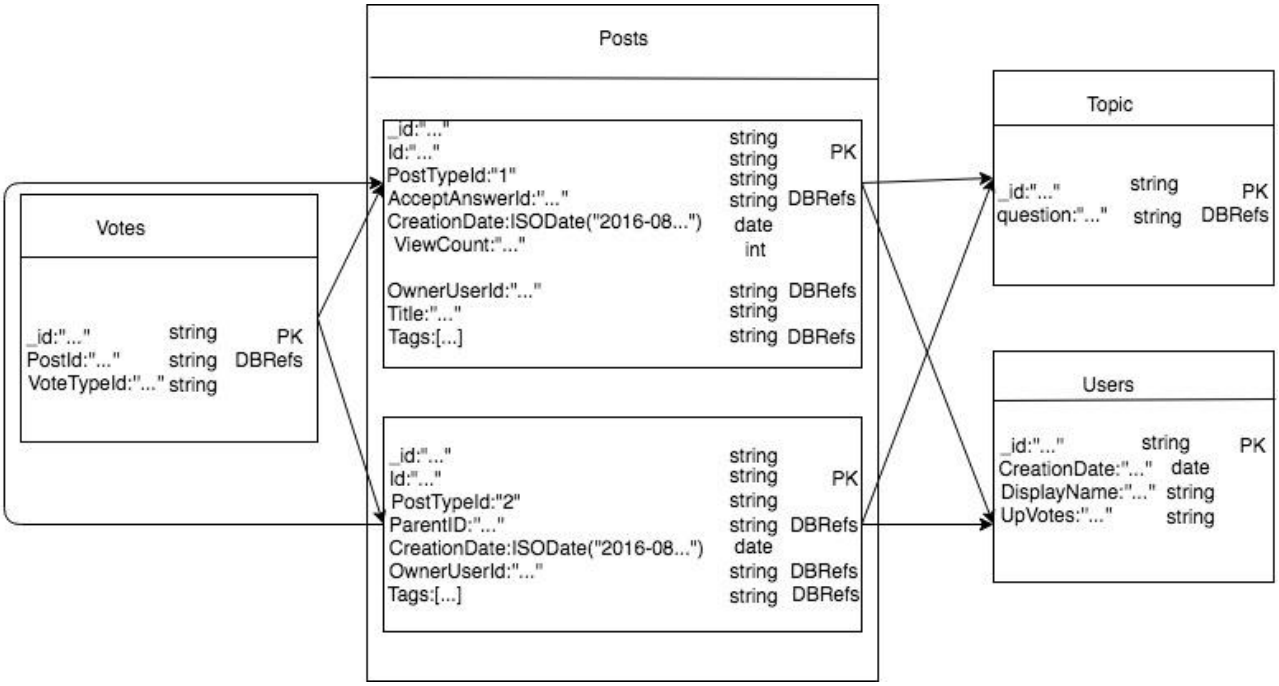
Neo4j is suitable for graphics type data such as social network, which is a significant difference from MongoDB. For this reason, we choose AQ6 because it describes a complex social network problem which would be much harder and less efficiency to solved by MongoDB. And we choose AQ4 and AQ5 with the similar reasons that they could be better solved as graphics data types.

·query workload

Neo4j query set: {SQ1, SQ2, AQ3, AQ5, AQ6}

2.2 Schema design

2.2.1 Mongodb schema design



2.2.1.1 Description

1)For Posts collection: It contains both question and answer which would be identified by PostTypeId: A, question collection: the dataset would be identified as question if PostTypeId is 1, which contains id(string), PostTypeId(string), AcceptAnswerId(string), CreationDate(date), ViewCount(int), OwnerUserId(string), Title(string), Tags(string). Id would be identified as Primary key and AcceptAnswerId would be DBRefs reference to answer collection; OwneruserId would be DBRefs reference to Users collection; Tags would be DBRefs reference to Tags collection. B, answer collection: the dataset would be identified as question if PostTypeId is 2, which contains id(string), PostTypeId(string), ParentId(string), CreationDate(date), OwnerUserId(string), Tags(string). Id would be identified as Primary key and ParentId would be DBRefs reference to question collection; OwneruserId would be DBRefs reference to Users collection; Tags would be f DBRefs reference to Tags collection. **2)**Topic collection is evolved from tags collection, containing _id and question. For "question": It contains Id of corresponding question. _id would be identified as Primary key and question would be DBRefs reference to question collection; **3)**Users collection contains _id(string), CreationDate(date), DisplayName(string), UpVotes(string). _id would be identified as Primary key. **4)** Votes collection contains _id(string), PostId(string), VoteTypeId(string). _id would be identified as Primary key and PostId would be DBRefs reference to question and answer collection.

To improve the performance and efficiency of code execution, there is four indexes need to be created: 1)Id、

2）ParentId、3)CreationDate、4)OwnerUserId

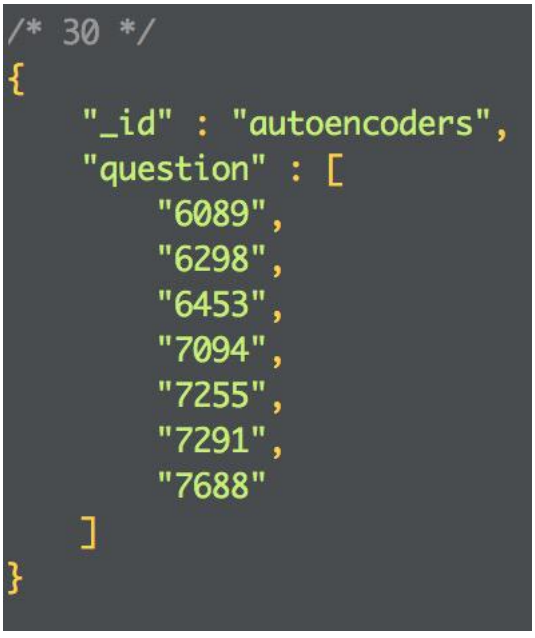
2.2.1.2 Sample document of each collections

1.Posts collection (Question:PostTypeId=1; Answer: PostTypeId=2)

```
/* 1 */
{
    "_id" : ObjectId("5bb84030a936994cbeef7c1c"),
    "Id" : "1",
    "PostTypeId" : "1",
    "AcceptedAnswerId" : "3",
    "CreationDate" : ISODate("2016-08-02T15:39:14.947Z"),
    "ViewCount" : 306.0,
    "OwnerUserId" : "8",
    "Title" : "What is \"backprop\"?",
    "Tags" : [
        "neural-networks",
        "definitions",
        "terminology"
    ]
}

/* 2 */
{
    "_id" : ObjectId("5bb84030a936994cbeef7c1d"),
    "Id" : "3",
    "PostTypeId" : "2",
    "CreationDate" : ISODate("2016-08-02T15:40:24.820Z"),
    "OwnerUserId" : "4",
    "Tags" : [
        "neural-networks",
        "definitions",
        "terminology"
    ],
    "ParentId" : "1"
}
```

2.Topic collection



```
/* 30 */
{
    "_id" : "autoencoders",
    "question" : [
        "6089",
        "6298",
        "6453",
        "7094",
        "7255",
        "7291",
        "7688"
    ]
}
```

3.Vote collection

4.Users collection

2.2.1.3 Data preprocessing

1)For Posts collection

Data selection: _id, Id, PostTypeId, AcceptedAnswerId, CreationDate, ViewCount, OwnerUserId, Title, Tags, ParentId

Data duplicated: Score, Commentcount, Favoritecount ,CloseDate, OwnerDisplayName

Data preprocessing: Delete the ParentID of Question document; Delete AcceptedAnswerID, ViewCount, Title of Answer document.

2)For Votes collection:

Data selection:Id, PostId, VoteTypeId

Data Duplicated: CreationDate, UserId, BountyAmount

3)For Users collection:

Data selection:_id, CreationDate , DisplayName , UpVotes, DownVotes

Data Duplicated: Reputation, LastAccessDate, Location , Views , AccountID

4)For Topic collection:

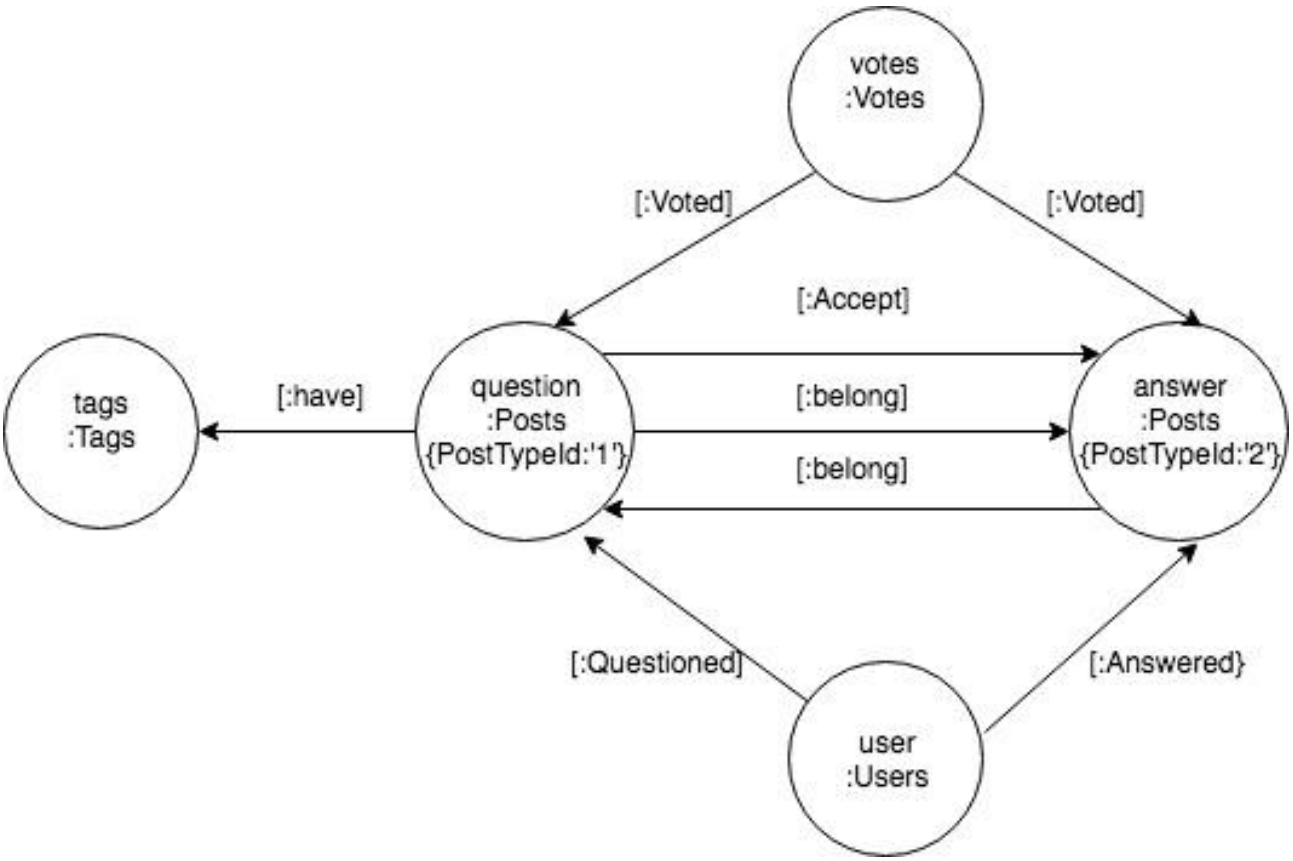
Data selection: _id, question

Data Duplicated: Count,ExceptPostId, WikipostId

Preprocessing :_id inTopic collection is extracted from TagName like “Svm”; question is representing the Id of corresponding question which contains the topic.

2.2.1 Neo4j schema design

2.2.1.1 Schema design diagram



· Description

1) Posts nodes are divided into two parts as question and answer, judging by PostTypeId, which is designed to be suitable for the questions. The relationships between two nodes contain: 1.belong: question and answer are belonged to each other, which is matched by ParentId of answer and Id of the question; 2.accept: it represents the answer which is accepted and this would be matched by AcceptAnswerId. 2) Users nodes mean the end users of this website who may have pointed out questions or gave answers to questions. In that case, the relationship between users and posts should correspond as Questioned or Answered. 3)For Votes node, it means the vote to question or answer. The relationship is Voted which matched to two types of posts, identifying by PostId.4)Tags means the tag of the question and the relationships between question、answer and tags is have. (ps: Additionally, tags are connected into the answer node manually, for comparing neo4j with MongoDB in Aq2, as performance comparing. However, this relationship would not be shown in schema designed for the query workload)

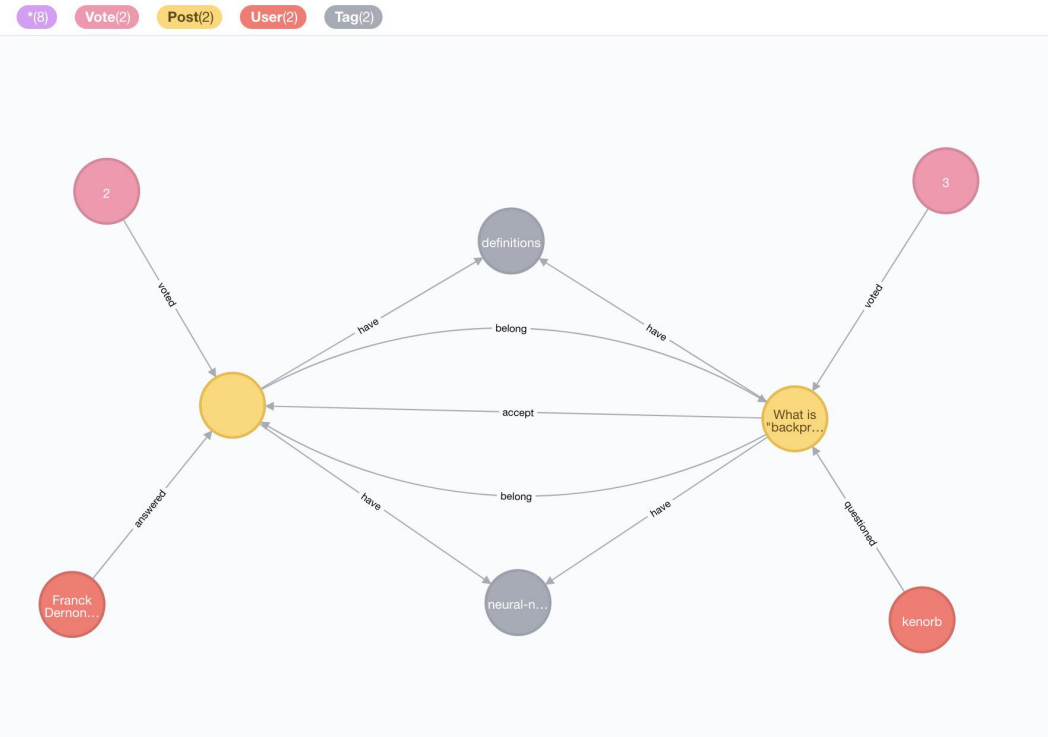
To improve the performance and efficiency of code execution, there is four indexes need to be created: 1)PostTypeId、2)VoteTypeId、3)VoteVoteTypeId

2.2.1.2 sample property graph

Node types			
1. Posts	2.Users	3.Votes	4.Tags

Note：For post that PostTypeId=1, we set a variable name as Question; For For post that PostTypeId=2, we set a variable name as Answer

Relationships
1. (Question:Post{PostTypeId:1})-[:belong]->(Answer:Post{PostTypeId:2})
2. (Question:Post{PostTypeId:1})<-[:belong]-(Answer:Post{PostTypeId:2})
3. (Question:Post{PostTypeId:1})-[:Accept]->(Answer:Post{PostTypeId:2})
4. (Users)-[:Questioned]->(Question:Post{PostTypeId:1})
5. (Users)-[:Answered]->(Answer:Post{PostTypeId:2})
6. (Votes)-[:Voted]->(Question:Post{PostTypeId:1})
7. (Votes)-[:Voted]->(Answer:Post{PostTypeId:2})
8. (Question:Post{PostTypeId:1})-[:have]->(Tags)



(Example)

2.2.1.3 Data preprocessing

1)For Posts

Data selection: Id, PostTypeId, AcceptedAnswerId, CreationDate, ViewCount, OwnerUserId, Title, Tags, ParentId

Data duplicated: Score, Commentcount, Favoritecount ,CloseDate, OwnerDisplayName

2)For Votes:

Data selection:Id, PostId, VoteTypeId

Data Duplicated: CreationDate, UserId, BountyAmount

3)For Users:

Data selection:Id, CreationDate , DisplayName , UpVotes, DownVotes

Data Duplicated: Reputation, LastAccessDate, Location , Views , AccountID

4)For Tags:

Data selection: Id, TagName

Data Duplicated: Count,ExceptPostId, WikipostId

2.3Query design and execution

2.3.1 MongoDB query design

1.SQ[1]

·requirement and design

Simple query 1 requires to find all users engaged in a given question and return detailed information about these users. We should not ignore the user post the question and also we need to avoid the duplicate users information because some user may answer their own question.

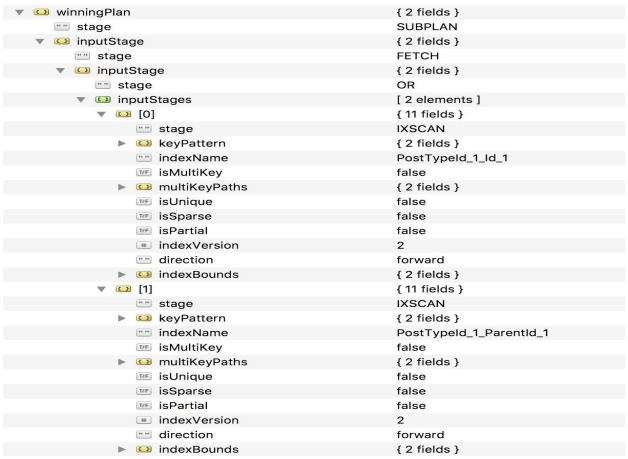
To implement the query, we execute the process as below:

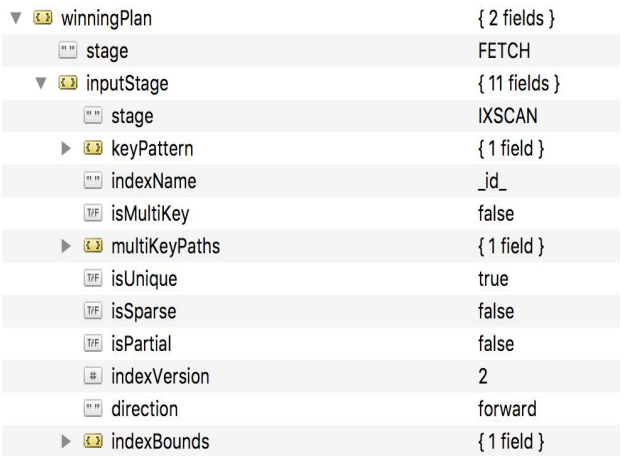
1. We use match function in aggregation pipeline to find the posts whose id or parent id equals the given question id
2. We group the result from the first step by their 'OwnerUserId' field. This can help to get all user id and remove duplicate user id
3. By using the user id from step 2, we output all the document with four fields required.

·Performance and analysis

The execution time is 1ms;

We use the time function in python and explain function in MongoDB to record the running time





and query planner to analyze the execution statistics.

The result shows that the query planner uses the two index to speed up the query in the sub-plan when implementing the matching cursor. And use the id index when traversing the user information in the user collection.

2.SQ[2]

·requirement and design

Simple query 2 requires us to find the most popular question with the highest value of field 'ViewCount' in any given topic. We need to convert the type of field 'ViewCount' since the sort function may return the wrong result when the type of field 'ViewCount' is a string.

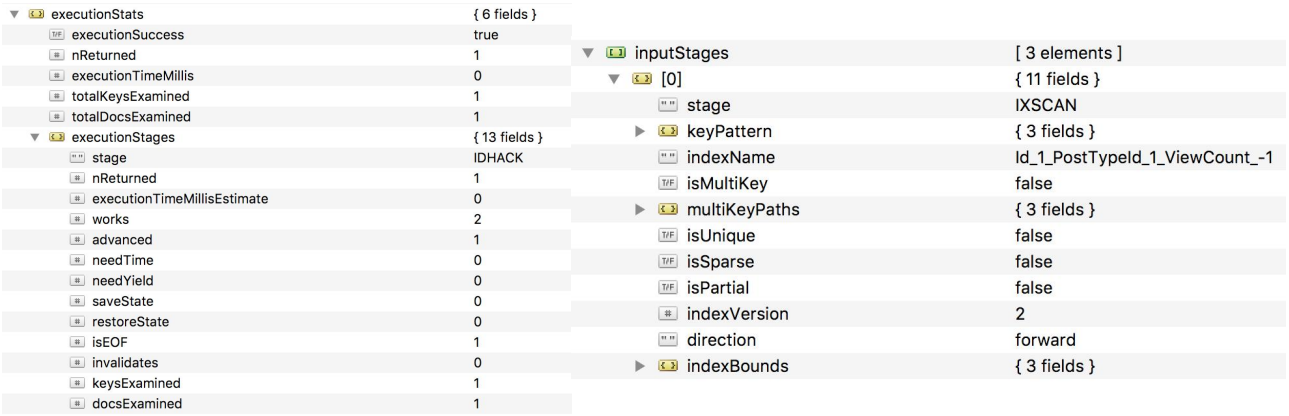
The execution process is shown below:

1. We first find all the question id by using a topic collection which was generated in the preprocessing stage
2. We find all document by using $match and $in function and sort them according to their 'ViewCount'

Performance and analysis

The execution time is 7ms;

By using the explain function, we find that it uses _id index when finding the topic name and the related question id in the first step. Then, the index with multiple fields that we designed before



specifically for this query was used in the second step to speed up the query results.

3.AQ[1]

·requirement and design

The analytical query 1 ask us to find the easiest question for each topic. Therefore, this query requires us to convert the 'CreationDate' field from string type to date type at first, because the field in string format cannot be calculated for date interval.

We followed the steps below to execute the entire query:

1. We used a temporary variable in python to store a topic in a given topic collection.
2. By querying this temporary variable, MongoDB returns all the question ids in the topic collection.

3. We find all the question ids in the post collection, then we use the lookup function to let it connect to the parent id of the post-collection based on the id. So all the answers in it are embedded in each question.

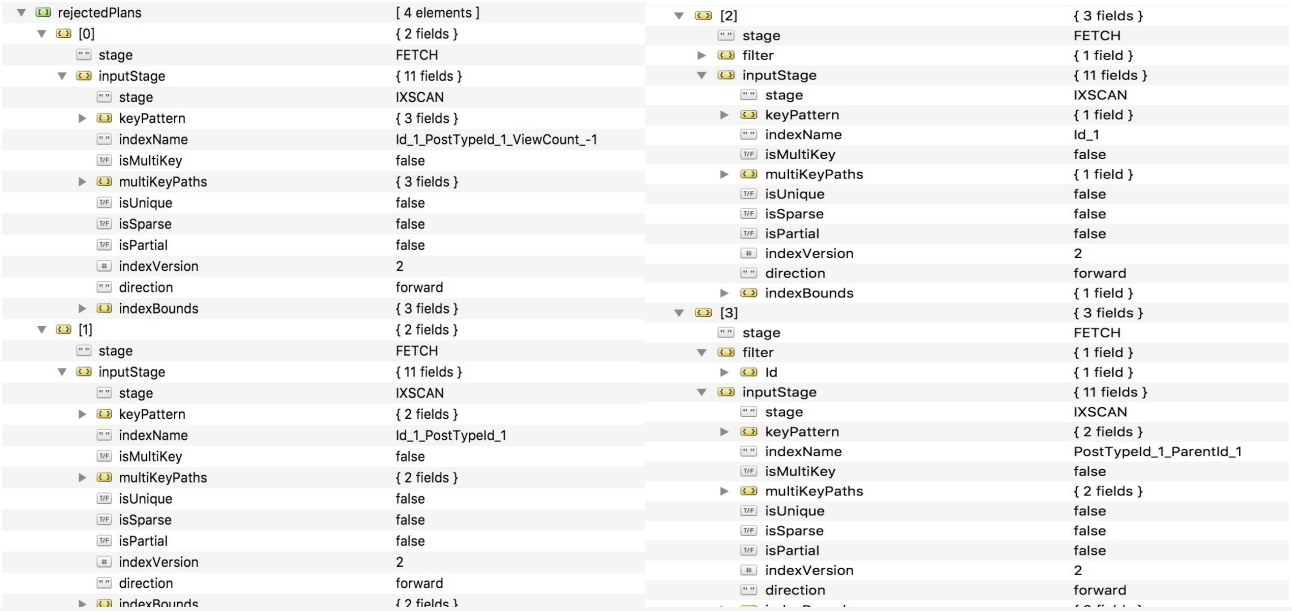
4. After picking the ‘CreationDate’ filed as ‘AnswerTime’ for these answers, we expand the ‘AnswerTime’ field for each question and then calculate the interval between ‘AnswerTime’ field and ‘CreationDate’ field

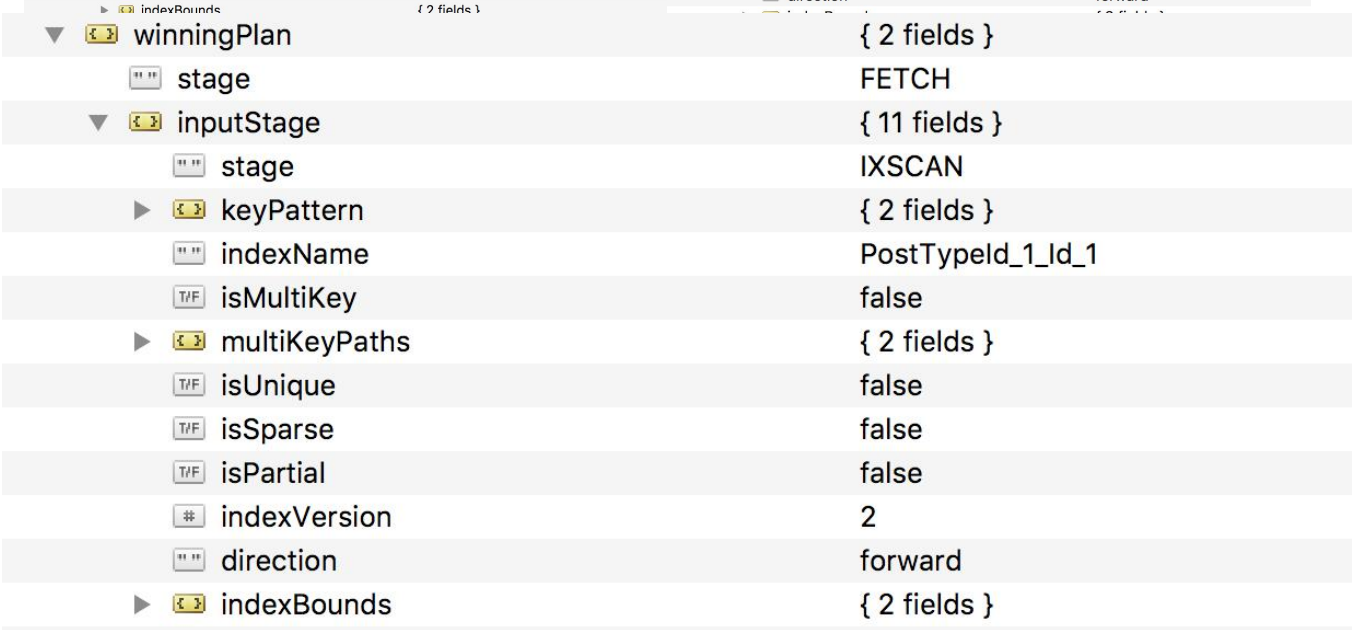
5. Finally, we sort the result to get the first one, which represents the easiest question under the topic with its corresponding shortest answer interval.

·**Performance and analysis**

The execution time is 15ms;

The query planner in the second step is similar to the simple question 1. It uses the id index to speed up the query speed. Then for the third to the fifth step, the query planner chooses the index with ‘PostTypeId’ and ‘Id’ as the index and reject another four query plan.





4.AQ[2]

·requirement and design

The analytic query 2 is to find the topic that has the largest number of participating users during a period of time. The time range here needs to use the DateTime module in python to store the data in date format, and store it in the dictionary to specify the time period. Also, we need to remove duplicate users to avoid erroneous results when counting the number of participating users.

We performed the following steps to execute the analytic query 2:

1. We find all posts in the time range
2. We expand the 'tags' field of all posts. Since the corresponding tags have been added to answer according to their parentId field, the statistics become more convenient here.
3. The $group function with the $addToSet function can add 'OwnerUserId' field in a array and remove duplicates for each.
4. Use $size to calculate the number of participating users for each tag and take the top five

·Performance and analysis

The execution time is 4ms;

We created an index on the 'CreationDate' field for this question and it was also used by the query planner to complete the first step using the $match to find the eligible document. The explain result shows below.

5.AQ[4]

·requirement and design

The analytic query needs to count the accepted answer of a given user under various topics, and use the result to recommend the latest questions that have not yet been answered to this given user. Moreover, there will be a threshold to limit the topics associated with the user.

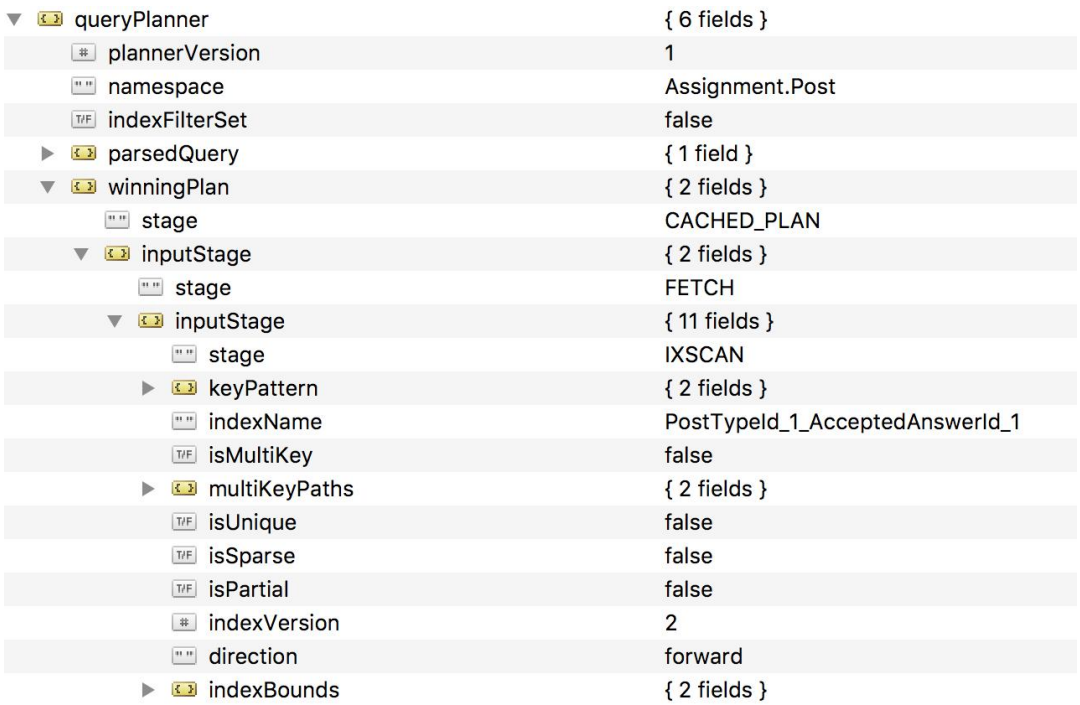
In order to execute this query, we performed the following steps:

1. We use the 'accepted answer' field and 'PostTypeId' field to find all the questions that already have an accepted answer.
2. After that, the result left join from Post collection by using its id field and the 'ParentId' field of the Post collection. We reduced the consumption of left join operations by initially match the question in the first step. Then, the corresponding accepted answer has been embedded in the questions.
3. Next, we take the 'OwnerUserId' field in the embedded field and match the given username,
4. Then, expanding the 'tags' field and use $group function to count the accepted answer this user has in each topic
5. We use the $match function and the value of threshold to get the all the tags that meet the criteria
6. Then, we expand tags for all questions which do not have accepted the answer yet and then match them using the topics from step 5
7. Finally, after grouping them by tags and sort them by using the 'CreationDate' field, we can output the final result.

·Performance and analysis

The execution time is 73ms;

To speed up the query, we design an index based on the 'AcceptedAnswerId' field and 'PostTypeId' field:



2.3.2 Neo4j query design

1.SQ[1]

·Requirement

Find the users and their respective profiles in a given question.

·Query design

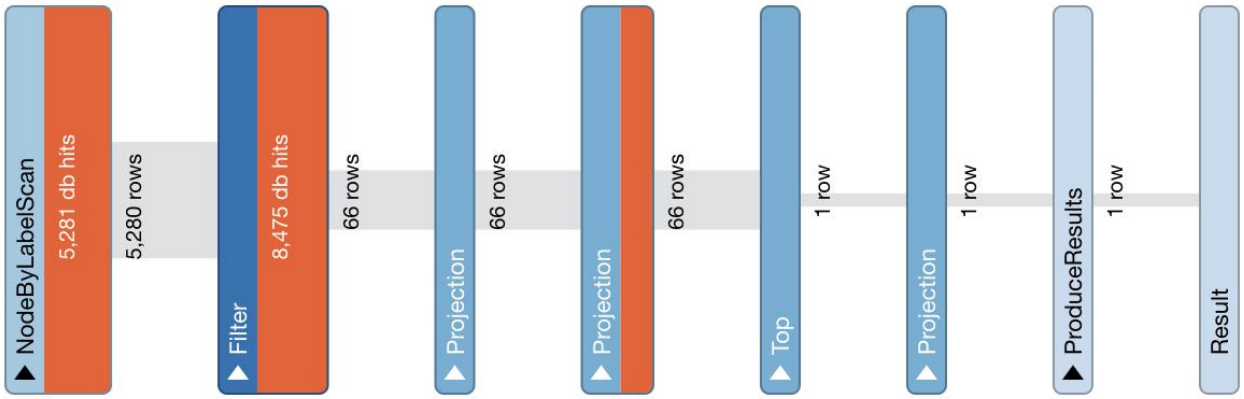
To find users in a given question, we can use the given question and its answers to find the corresponding users who asked this question or gave answers for it. The query is designed in two directions:

1.find users who asked this question by using the relationship [:questioned]: " node(given question)->node(users) "; meanwhile

2.find users who answered this question by using the relationship [:answered]: "node(given question)->node(answer of the given question)->node(users)"

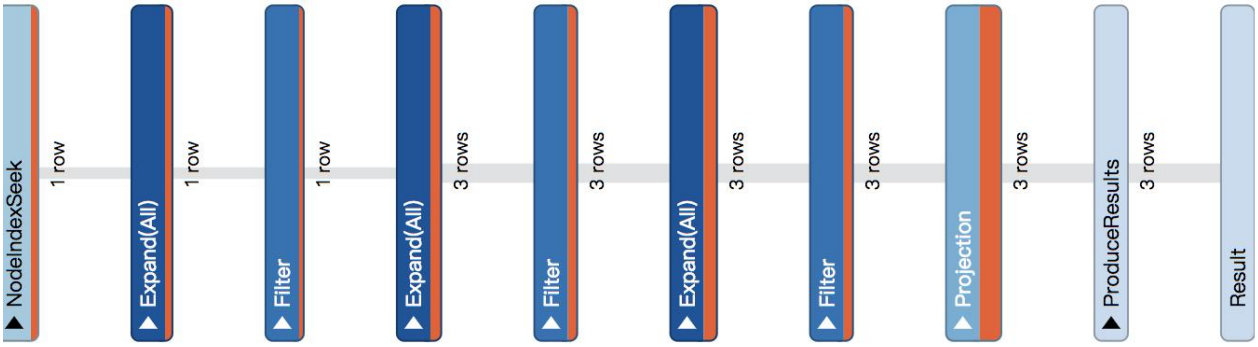
·Performance and analysis

The execution time is 5.23ms;



1. Before index:

Rather than performing an AllNodeScan, this search performs a NodeByLabelScan followed by a Filter, which gives us 5280 database hits instead of the whole database. Filter lookup 8475 nodes and return 66 rows, which is considered as the performance bottleneck of this query.



2. After index:

With the “create index on”, we are setting an index and optimize the search, which would help us to find the start point for queries in the graph database. And this start point is “given the question”. NodeIndexSeek gives us the specific 1 database hits instead of 5280(before the index), which would greatly improve the performance of this search

2.SQ[2]

·Requirement

Find the most viewed question in a given topic。

·Query design

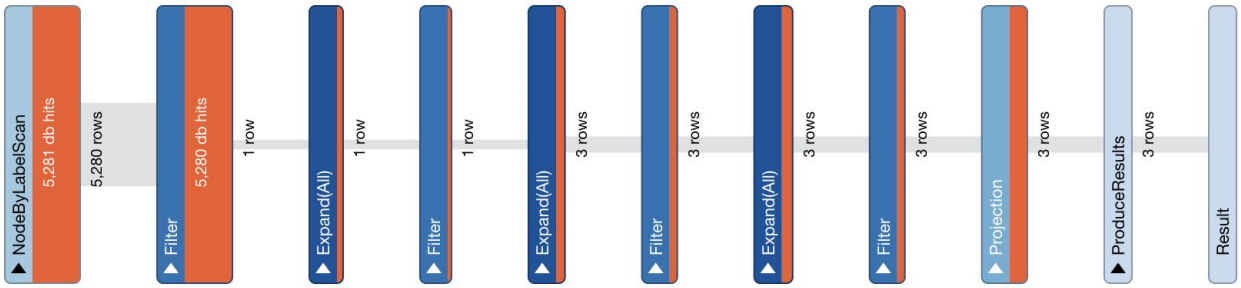
This question could be solved by identifying the post which contains the specific given tags and sort by the viewpoint of each post. So the query is designed as two steps:

1.first, use function “in” to find all questions whose attributes tags has the specific given tags;

2.second, order by the viewcount to find the most viewed questions.

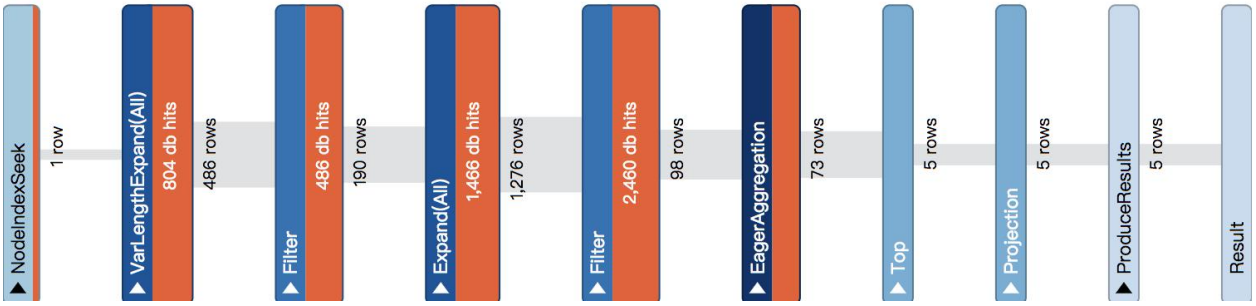
·Performance and analysis

The execution time is 55 ms;



1. Before index:
NodeBYLabelScan lookup 5280 nodes in the database, applies a filter and return one row. With the combined database hits from both functions, this search has performed 15060 pieces of work. This query relatively improves the performance by using label and classifier at front of the query.

2. After index:



As the same reason in SQ[1], NodeIndexSeek optimizes the search and improve the performance, which would give one database hit instead of 5280(before the index). Additionally, two filters would give back 190 rows and 98 rows.

3.AQ[3]

·Requirement

Given a topic, find the champion user and all questions the user has answers accepted in that topic

·Query design

This question could be easily solved with graphics data type by analyzing the relationships in the network.

1. First, find the given tag and its questions and the corresponding accepted answers of each question.
2. Second, find all users who are related to these accepted answer. Finally, sort users by the number of their accept answers in this field and select the first one as champion answer
3. Third, use the result from step 2 to find all accepted answer in this tag

·Performance and analysis

1. The execution time is 96 ms;

For this query workload, NodeByLabelScan would return 288 database hits and apply a filter to return 1 row. The performance bottleneck is the second filter which would search 1382 database hits and return 999 rows. However, the performance of time efficiency is acceptable in total.

4.AQ[5]

·Requirement

Find all the questions whose accepted answer did not receive the highest up mod votes.

·Query design

We first count up_mod votes for all answers and make them as subgroup according to the question id and then we sort the results for each subgroup according to the number of up_mode votes.

1. We find answer id which achieves the highest number of up_mode votes.
2. Then we count the up_mod votes for accepted answer with question
2. And compare the up_mod vote of accepted answer with corresponding up_mod vote of highest answer. Finally, we output these question id whose accept are not equal to the highest up_mode id.

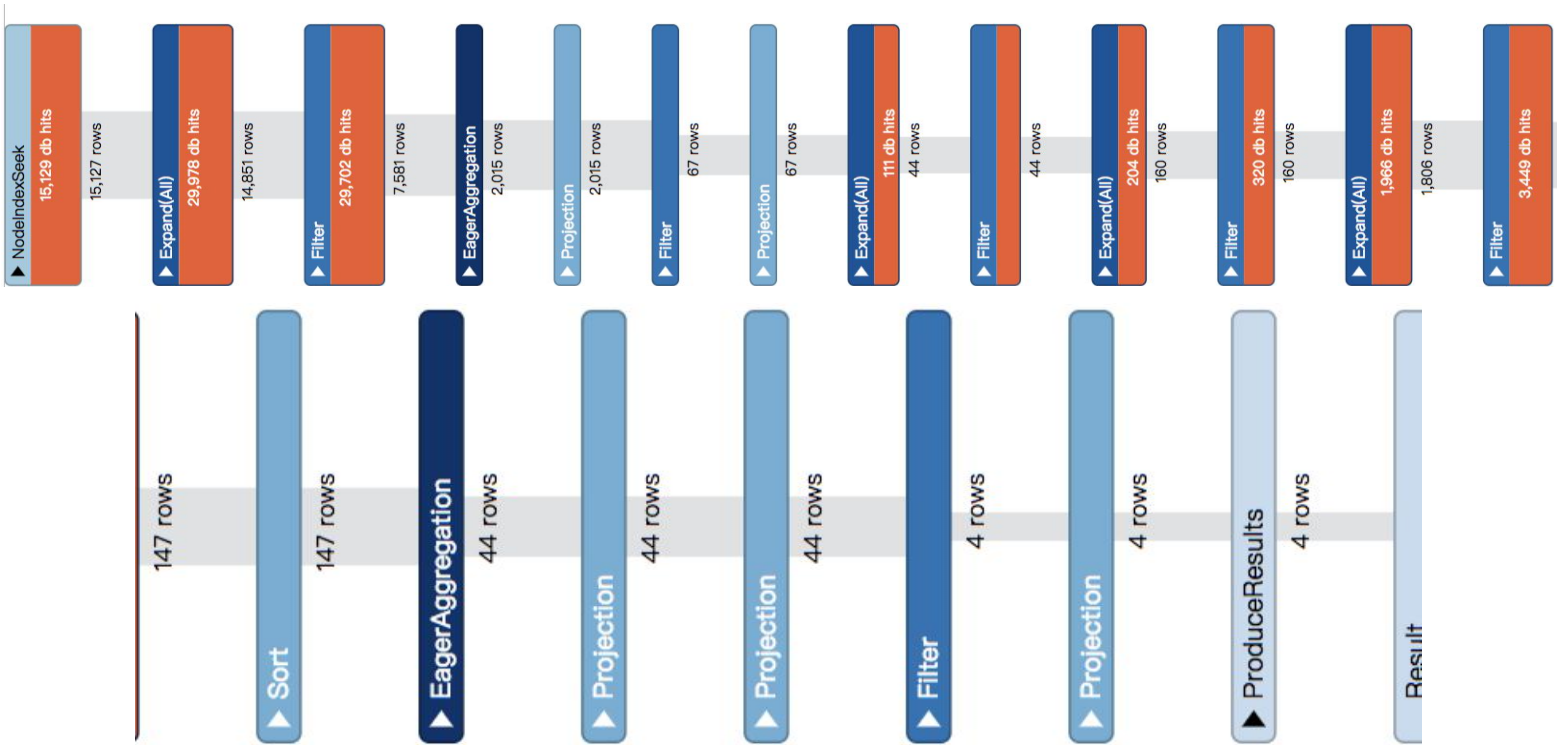
·Performance and analysis

The execution time is 108 ms;

1. Before Index:

This search is complex with lots of steps. First, NodeByLabelScan would be efficiency to keep from the extra work and several filters is acting the function of classifiers which is necessary for this query, which would affect the performance.

2. After index:



After using an index, the search's performance would be improved greatly, comparing with the previous expensive search.

5.AQ[6]

·Requirement

Find co_author of a given user who could be found in same questions and corresponding answers.

·Query design

We could solve this question by utilizing relationships between the given user and his co_authors which is that their posts(questions or answers) are connected or indirectly connected by each other.

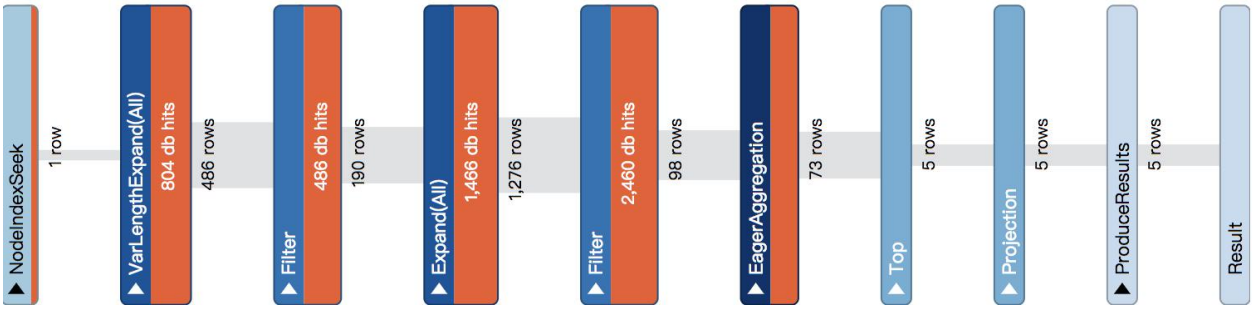
1. In that case, we decided to use -[*0..n]- to find the relative post of the questions or answers from the given user.
2. We could also find the relative users to the above posts by using the same way.
3. And then we can count the number of appearing in these posts for each co_authors and distinct the number in repeat posts (some user may give several answers in one question)

·Performance and analysis

The execution time is 65 ms;

1.Before Index

NodeBYLabelScan lookup 5280 nodes in the database, applies two filters and return 98 rows. We split the query into two MATCH statements and start with the point: a post with a specific id, which act as a classifier and prevents the database from extra work. However, in the phase of finding co_author by using relationships, query have to expand all the relative nodes and that would be considered as a performance bottleneck for this query.



2.After index

If we apply NodeIndex to this query to tighten up the searches, we only get one database hits and expand 804 DB hits instead of 5280db hits and 293250 DB hits. After compare and analysis, we find the three indexes work well and make a great improvement in the performance of our workload queries.

3.Comparison of two system

Overview of the comparison for MongoDB and neo4j is summarized in the following graph:

	MongoDB	Neo4j
Description	One of the most popular document stores	Open source graph database
Primary Database model	Document store	Graph DBMS
Data Schema	Schema-free	Schema-free and schema-optional

3.1Performance comparison

For the performance comparison, we compared MongoDB and Neo4j in several aspects: time efficiency and complexity of the code.

1. **time efficiency:** For the simple queries, both of their execution time in the shell is around 2.00ms to 5.00ms, the difference is not very considerable. However, the gaps of that in analytic queries are relative considerable and MongoDB has a better performance. We analyze and compare two methods in query logic and data structure and find two reasons for explaining this phenomenon: 1. MongoDB create index on Id,ParentId,CreationDate,OwnerUserId. However, considering space efficiency and other considerations including data amount, we choose not to create an index for the Neo4j system at first; the 2.query logic of Neo4j could be evolved by using functions such as “distinct” in “with” step to reduce multiple inquiries on repeat nodes.

However, we made a evolution to create an index on PostTypeId,VoteTypeId, VoteVoteTypeId and optimize the query design. After that, in order to compare time efficiency in the same standard, we use the function “%timeit” to measure, which is the average time of 100 times execution.

	MongoDB	Neo4j
Sq1	1.66	2.06
Sq2	3.25	4.75
Aq1	6.51	27.7
Aq2	2.29	19
Aq3	33.4	25
Aq4	70.8	19.8
Aq5	N/A	33.1
Aq6	15.6	3.19

2. **the complexity of code:** For some queries such as analytic query [6], the code complexity of Neo4j is much lower than that of MongoDB. Because Neo4j is more suitable for processing graphics data type such as network related problem.

```
{"$match":{"OwnerUserId":user_id}},   #given user
{"$project":{"_id":"$Id","PostTypeId":1,"OwnerUserId":1,"ParentId":1}},
 {"$lookup":{         # embedd the questioner whose question are answered by this given user
            "from":"Post",
            "localField":"ParentId",
            "foreignField":"Id",
            "as":"HelpedQuestioner"
}},
 {"$graphLookup":{     # embedd the answerer who answer the same question with this given user
            "from":"Post",
            "startWith":"$ParentId",
            "connectFromField":"Id",
            "connectToField":"ParentId",
            "as":"SameLabelAnswer"
}},
{"$graphLookup":{         # embedd the answerer who answer the question this user asked
            "from":"Post",
            "startWith":"$_id",
            "connectFromField":"_id",
            "connectToField":"ParentId",
            "as":"CooperatedAnswer"
}},
 {"$project":{"_id":"$Id","PostTypeId":1,"OwnerUserId":1,"ParentId":1,
             "CooperatedAuthor":"$SameLabelAnswer.OwnerUserId",
                "cooperatedAnswerer":"$CooperatedAnswer.OwnerUserId",
                "helpedQuestioner":"$HelpedQuestioner.OwnerUserId"
                }},
 {"$project":{"_id":"$Id","PostTypeId":1,"OwnerUserId":1,"ParentId":1,
             "CooperatedAuthor":{"$setUnion":["$CooperatedAuthor",
                                    "$cooperatedAnswerer",
                                    "$helpedQuestioner"
                                     ]}
            }},
{"$unwind":{"path":"$CooperatedAuthor","preserveNullAndEmptyArrays":True}},
{"$group":{"_id":"$CooperatedAuthor","CoopTime":{"$sum":1}}},
{"$match":{"_id":{"$ne":user_id}}},
{"$match":{"_id":{"$ne":None}}},
{"$sort":{"CoopTime":-1}},
{"$limit":5}
```

[AQ6]-MongoDB:

[AQ6]-Neo4j:

```
###AQ6
#13ms   #Given person: id = 8
pd.DataFrame(graph.run("match (u:User{id:'8'})-->(:Post)--(:Post)<--(u1:User) \
where u1.id<>'8' \
return u1.DisplayName as name,count(u1.id) as coop_Times order by coop_Times DESC limit 5"
```

3.2 Evolution

At first, for Neo4j we consider the space efficiency and small data amount and choose not to create indexes. However, in the performance analysis, we used profile to expand the data flow in the search process and find the performance is limited and poor. In that case, we try to create indexes on: 1)PostTypeId、2)VoteTypeId、3)VoteVoteTypeId. After that we compare the performance before and after the index creating, we found the three indexes work well and make a great improvement in the performance of our workload queries.

5.User Manual

To run the query on MongoDB, you need to install the Jupiter notebook, python and PyMongo at first. Then after using the 1_convertTSV.ipynb create three data file in Jason format, you can follow the code in another three txt file to modify the data, structure and create the index. At last, you can run QueryCode.ipynb file to execute the query. There are some sample results already contained in the QueryCode.ipynb file. The execution source code of Neo4j is saved in the Neo4j file. Please run the code in convertTsv.ipynb, Neo4j_schema_design. Then copy the code in query_shell_code and run it on a Neo4j browser. There is some sample answer contained in the queryExecution.ipynb.

Reference

1. Jordan, G. (2014). Practical Neo4j. New York City, NY: APress.
2. Membrey, P., Hows, D., & Plugge, E. (2014). MongoDB Basics. New York City, NY: APress.

·contribution

	Chichi Zhi	Xin NIng				
1.MongoDB						
Data process	60%	40%				
Schema design	60%	40%				
Query design	60%	40%				
2.Neo4j						
Data process	50%	50%				
Schema design	50%	50%				
Query design	50%	50%				
3.Query optimization	55%	45%				
4.Report	50%	50%				
Total	55%	45%				