Graph Database Query Optimization by Data Structure in Neo4j

Yinbei Tong
Iowa State University
ybtong@iastate.edu

Ying Wei
Iowa State University
yingw@iastate.edu

Tianxiang Gao
Iowa State University
gtx@iastate.edu

Ce Zhang
Iowa State University
cez@iastate.edu

Abstract

Several graph database system has emerged in the past few years. And due to the limitation of traditional database, a hot topic about graph database is arising in recent years.

Efficient query is very important in large database. So for the graph database, we want to do the analysis about query optimization. Because the efficient query is a time-critical task, so in this paper we will focus our analysis on the query execution time.

We get the idea from has-table and index on some attributes that can save the traversal time for whole graph database. In our research, the data structure also the crucial part for improving the query efficiency. According to the query, we use the indexed attribute idea to restructure the original data in different model.

From our analysis, we find that the index on the popular attributes can save the query execution time. The reduced time degree is based on the data quantity and the index layer for structure model.

Keywords

Graph Database, Index, Query Optimization, Performance, Neo4j, Data Structure

1 Introduction

In recent years, the restriction of relational databases has led to the development of new database technologies called NOSQL databases. One important type of NOSQL databases is graph database, which is used to store graph-like data. Graph database has gained much attention recently because in popular areas where a database is needed such as social network, it has proved to be more efficient than relational database since data in these areas are more relied on relations instead of entities.

Graph database is widely applied in many fields now including web mining, biology, data provenance and semantic web. When applying graph database to these fields which has very large data scales, an obvious question people face is how to reduce the average response time of querying graph database. One commonly used way to accomplish this goal is query optimization, in other words, enhance the performance of graph database by searching for better query structures.

Another possible approach to improve the performance is to initialize the graph database with a structure that is more efficiency for querying. However, few studies have investigated how to compute an effective structure of graph database. It's difficult to determine which data structure is better since the data could be different and queries are unknown. Logically, it's reasonable to expect that changing the data structures would affect the efficiency of graph databases. Thus, given a specific data and a set of queries, there must exist a best structure with lowest average response time. Apparently, there is a need for developing an algorithm on evaluating and selecting a good graph data structure.

This paper presents an experiment on measuring the performance of different data structure given graph-like data and a set of queries in graph database Neo4j. We first form the data into different structures that might have better performance, then evaluate the result by computing the average response times for each structure, make a comparison between different structures under each distinct queries set. Our analyses on the result displayed the best data structures for each type of queries, and could also be helpful in query optimization.

2 Related Work

The traditional database, in particular the relational mode shows a limitation for the requirements of current application, however, the graph-like structure of the data is needed, so the graph database has been grown up in recent years.

There are several graph database models be compared in the research. There is one paper[1] talks about the features, advantage and disadvantage of AllegroGraph, DEX, HyperGraphDB, InfinieGraph, Neo4j, Sones. And it founds that all of them provide an innate support for different graph structures, query facilities, but just a few of them has query languages, neo4j is one be included in this small group. The member of DAMA-UPC[2] also did the similar research. They evaluate the performance of four of most scalable native graph database: Neo4j, Jena, HyperGraphDB and DEX. After the comparison of these graph database by their performance for different typical

graph operations and graph size using the HPC(High Performance Computing) Scalable Graph Analysis Benchmark v1.0, they found that only DEX and Neo4j were able to load the largest benchmark sizes. And Among these two graph models, the Neo4j execute query faster than DEX for the large dataset.

Neo4j, an open-source graph database was released in 2007, and the first version was released in 2010[3]. Due to the popularity of the graph database and growing network operation, and the research we mentioned above, we want to make a research on the graph database structure and query optimization on Neo4j to improve the effectiveness of the graph database operation.

For recreate the dataset structure, we get idea from some research papers. Some researchers [4] found the indexing method by suffixing tree structure. In order to deal with graph searching efficiently, they come up the advanced method for indexing, which is indexing by suffix tree structure. This method aims to try to reduce the search space by filtering out the graphs that do not contain the query. First step, builds the suffix tree query, and load suffix tree graph database. Second step, select only candidate graphs by tree matching. Finally, run exhaustive subgraph matching on candidates. Even though the index paths may result in more preprocessing time and index space, paths require less filtering and querying time. So we get the idea from this paper to create the index by the frequent search in the query, and then assign the data node to their matching index. The index method also mentioned by Sherif Sakr and Ghazi Al-Naymat[5]. They find several graph indexing and querying techniques. The common process for all of these method be listed as follows: according to the graph query, they create the query processor which using graph index, then after the query processor choose out the candidate answer set, then get the final answer set by verification phase.

We also get the idea from another paper "CT-index: Fingerprint-based Graph Indexing Combining Cycles and Trees "[6]. Since the efficient subgraph queries are time-critical task in application area, the author proposed a new approach that is using a new hash-key fingerprint technique with a combination of tree. Another research paper from University of North Carolina[7] also mentioned this method, the second graph is a hash table which cross-indexes each subgraph for fast lookup. The core idea of these method is same with the previous one(index), and both of them try to using a key/index to save the quantities of the node traversal.

Benchmark is also an important part for optimize the query by restructure data. Traverse operations over graph database is a standard one. D.Dominguez-Sal etc. adopted HPC benchmark for graph processing [8]. Ciglan, Marek, Alex Averbuch, and Ladialav Hluchy[9] found an idea focus on graph traversal operations.

After studying these related works, we get some ideas and find some weak point of their papers.

What we want to do is making a research on the graph database structure and query optimization to improve the effectiveness of the graph database operation. There is no research that just talks about the query optimization on Neo4j. So we want to focus just on Neo4j to analyze this topic. The idea we get from the previous work is using index to restructure the dataset, and then create a benchmark that fairly show the performance of the different structure we created.

3 Proposed Work

There are 6 datasets were used in this paper. Each of them has 6 data structures. Each data structure is represented by a database. So each dataset has 6 databases. In the following section, we are going to explain the details of each database.

3.1 Databases Construction

The first database represented the original data structures, which is constructed by randomly generate a number of “Person” nodes in Neo4j. The “Person” node has three attributes: name, age and gender. The values of “name” attributes are generated by randomly choose first name and last name from a given string array. The values of “age” attributes are generated by choose a random number from 0 to 99 by using java’s Math package’s random method. The “gender” attribute has two possible values: “Male” and “Female” and is assigned also by using java’s Math package’s random method.

For each “Person” node, the probability to let “age” attribute has value of number n (0 <= n <= 99) is 1/100 because we are randomly pick number from 0 to 100. The probability to let “gender” attributes has value of “Male” or “Female” is 1/2 because we are randomly pick values between “Male” and “Female”.

The second database has 100 more number of nodes compared with first database. Each of new nodes has attribute “age”. The value of their “age” attribute are various from number 0 to number 99, which means each node will has its distinct “age” value. Then we setup edges between new nodes and “Person” nodes. When the new node with “age” attribute’s value is “x”, we will setup edge from this node to all the “Person” nodes that have “age” attributes’ value is “x”

The third database has two more nodes with attribute “name” than the first database. The “Person” nodes are generated in the same way as first database. The first new node has “name” attribute has value “Male” and the second new node’s “name” attribute has value “Female”. And the first new node has edges connected to every “Person” nodes that has “Male” in “gender” attribute. The second

new node has edges connected to every "Person" nodes that has "Female" in "gender" attribute.

The forth database has 102 more number of nodes compared with the first database. The "Person" nodes are generated in the same way as first database. It not only have two new nodes have "gender" attribute, but also has 100 new nodes have "age" attribute. The two new nodes' "gender" attribute has value "Male" and "Female". The 100 new nodes with "age" attribute has value various from 0 to 99, same as the second database. Then we setup edges from "age" nodes to its corresponding "Person" nodes that has a same value in "age" attribute and setup edges from "gender" nodes to its corresponding "Person" nodes that has a same value in "gender" attribute.

The fifth database has 300 more nodes compared with first database. The "Person" nodes are generated in the same way as first database. Besides "Person" nodes, it has 100 "age" nodes, which are generated the same way as the second database. Each "age" node has two edges. One edge connect to a "gender" node with value "Male", the other edge connect to another "gender" node with value "Female". Every "age" nodes has two "gender" nodes. So there are totally 200 number of "gender" nodes. Then we setup edge from "gender" nodes to "Person" nodes. Each "gender" nodes will setup edge with all the "Person" nodes that has same value in "gender" attribute and same value in "age" attribute of the "age" node connected with it. For example, an "age" node has "age" attribute's value equals to 1, and this node has two edges connected to two gender nodes with "gender" attributes' value "Male" or "Female". In this two gender nodes, the first node with "Male" value will setup edge to all the "Person" nodes that have "age" attributes equals to 1 and "gender" attribute equals to "Male". And the second node with "Female" value will setup edge to all the "Person" nodes that have "age" attributes equals to 1 and "gender" attribute equals to "Female".

The sixth database has 202 more nodes compared with the first database. The "Person" nodes are generated in the same way as first database. Moreover, it has two more "gender" nodes with "gender" attributes' value are "Male" or "Female". Every "gender" nodes has 100 number of "age" nodes respectively and each "gender" node will connect to 100 "age" nodes with "age" attributes' value various from 0 to 99. Then each "age" node has edge connect to "Person" with same "age" attribute value of itself and same "gender" attribute value of its connected "gender" node.

3.2 Data Set

We designed 5 dataset with different size in this paper. They have five datasets have 1000, 2000, 3000, 4000, 5000 number of "Person" nodes. Each of them will generate 6 database as explained in the last section.

3.3 Query

For each database, we designed 37 number of query to test its performance. Each query will performed 10 times and we only take the last 5 times' query time's average, because the query time have fluctuate values during the first 5 times queries.

Query 1 only query for the "Person" nodes whose "age" attributes' value is in an age range. We designed 10 age range with the properties of increasing selection factors.

Query 1

No.	Query Properties	Selection Factor
	age	
1	0 - 9	10 %
2	0 - 19	20 %
3	0 - 29	30 %
4	0 - 39	40 %
5	0 - 49	50 %
6	0 - 59	60 %
7	0 - 69	70 %
8	0 - 79	80 %
9	0 - 89	90 %
10	0 - 99	100 %

Query 2 has two queries properties requirements. The first one is query for all the "Person" nodes whose "gender" attribute's value is "Male". The second one is query for all the "Person" nodes whose "gender" attribute's value is "Male" or "Female". Because we assign "Male" and "Female" value to "gender" attribute evenly, so we don't need to query for only "Female" value's nodes, which will have the same result as query by "Male".

Query 2

No.	Query Properties	Selection Factor
	gender	
1	Male	50 %
2	Male & Female	100 %

Query 3 need to consider both gender and age. It query for "Persons" nodes whose "gender" is "Male" and "age" is in an age range. Same reason as query 2, we only need to get the query time for one gender.

Query 3

No.	Query Properties		Selection Factor
	gender	age	
1	Male	0 - 9	5 %
2	Male	0 - 19	10 %
3	Male	0 - 29	15 %
4	Male	0 - 39	20 %
5	Male	0 - 49	25 %
6	Male	0 - 59	30 %
7	Male	0 - 69	35 %

8	Male	0 - 79	40 %
9	Male	0 - 89	45 %
10	Male	0 - 99	50 %

Query 4 expand the query properties to a mix of gender and age queries properties. We designed 15 queries to test the databases' performance under those queries. As explained before, we assign the value to "gender" attributes of "Person" nodes with even probability. So theoretically, the following two queries should have same query performance:

```
match(p:person)
where (p.gender='Male' and p.age>=0 and p.age<20)
and (p.gender='Female' and p.age>=0 and p.age<60)
return p;
```

```
match(p:person)
where (p.gender='Male' and p.age>=0 and p.age<60)
and (p.gender='Female' and p.age>=0 and p.age<20)
return p;
```

Query 4

No.	Query Properties		Selection Factor
	gender=Male age range	gender=Female age range	
1	0 - 19	0 - 19	20 %
2	0 - 19	0 - 39	30 %
3	0 - 19	0 - 59	40 %
4	0 - 19	0 - 79	50 %
5	0 - 19	0 - 99	60 %
6	0 - 39	0 - 39	40 %
7	0 - 39	0 - 59	50 %
8	0 - 39	0 - 79	60 %
9	0 - 39	0 - 99	70 %
10	0 - 59	0 - 59	60 %
11	0 - 59	0 - 79	70 %
12	0 - 59	0 - 99	80 %
13	0 - 79	0 - 79	80 %
14	0 - 79	0 - 99	90 %
15	0 - 99	0 - 99	100 %

3.4 Experiment Environment

The running machine is on one PC with

CPU	Intel i7 870 2.93GHz
RAM	4 GB
Operating System	Windows 8.1 Enterprise 64-bit
Neo4j	2.3.1

We closed all the other software during our experiment. The database construction takes several minutes, so we use timer to count the construction time.

3.5 Storage Size

Database / Dataset Size	D1	D2	D3	D4	D5	D6
1000	729KB	0.99MB	1.08MB	1.62MB	1.15MB	1.14MB
2000	0.99MB	1.74MB	1.87MB	2.96MB	1.89MB	1.89MB
3000	1.41MB	2.52MB	2.72MB	4.34MB	2.67MB	2.65MB
4000	1.83MB	3.31MB	3.57MB	5.75MB	3.43MB	3.42MB
5000	2.28MB	4.09MB	4.60MB	7.31MB	4.21MB	4.20MB

3.6 Construction Time

Database / Dataset Size	D1	D2	D3	D4	D5	D6
1000	1:03	1:06	1:39	2:09	1:57	1:50
2000	2:13	2:18	3:31	4:30	3:50	3:45
3000	3:09	3:23	5:12	7:39	5:40	5:12
4000	4:29	4:41	7:17	9:45	7:42	7:38
5000	5:15	5:45	9:07	12:12	10:11	9:51
10000	10:57	14:05	18:33	25:00	20:05	19:00

We build database of dataset size of 10000 to see its build time.

4 Performance Evaluation

4.1 Theoretical Result

Ideally, for Query 1, we guess that database 2 and 4 should return the best result (lowest average response time) since they have the node Age as index. The worst case should be database 1 and 3, which basically just need to scan every node in database.

For Query 2, similarly, database 3 and 4 would be the optimal ones and database 1 and 2 have the highest average response time.

For Query 3, we search for people with gender=Male and age within the certain range. In this case, database 5 and 6 are expected to have the best performance theoretically, that's because node Age and Gender are actually working as indexes thus could possibly reduce much running time. And database 1(the original one) then becomes the worst again.

The condition of Query 4 is more or less the same with Query 3, but instead of let the gender has fixed value "Male", we made the queries more complex, querying both Males and Females with different age range. So, we expect that the average response time to be longer than the case in Query 3 while the performance of each database remains the same, that's to say, database 5 and 6 are still the optimal ones.

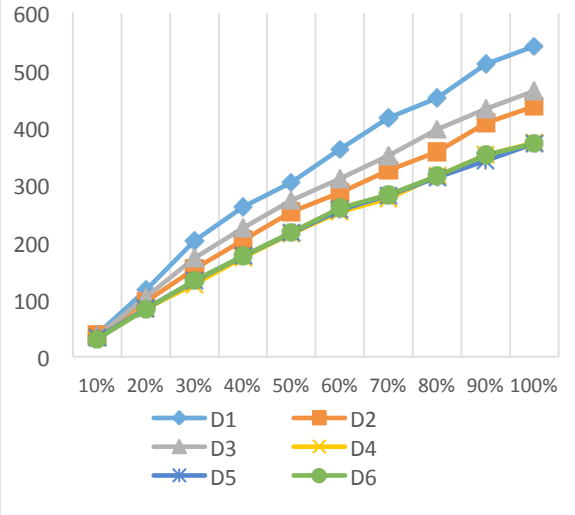
Another reasonable prediction is, as the size of dataset grows, the average response time increases proportionally as well. That's to say, since our data set grows linearly, the response times are linearly increasing.

4.2 Experiment Result and Analysis

This is the description for what the structure we create, and which line stands for which structure. D1 stands for the original structure, which has all of the person as a node in scatter status. D2 stands for the data structure that we use for gender as the index, and the person that match this attribute be connected by edge to the gender node. D3 stands for the data structure that we use for age as the index, and the "Person" nodes that match this attribute be connected by edge to the age node. D4 stands for the structure which we use two index, one is gender, and the other is age, the person who matches this two kinds of attributes connected by two edge to the gender and age node. D5 stands for the structure which we use age as the first layer index and gender as the second layer index, then the person who match these two kinds of attributes connect the second layer index. D6 stands for the structure which we use gender as the first layer index and age as the second layer index, then the person who match these two kinds of attributes connect the second layer index.

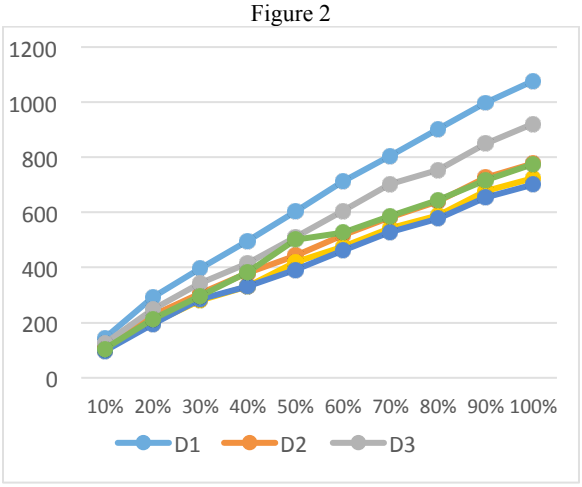
4.2.1 Query 1

Figure 1



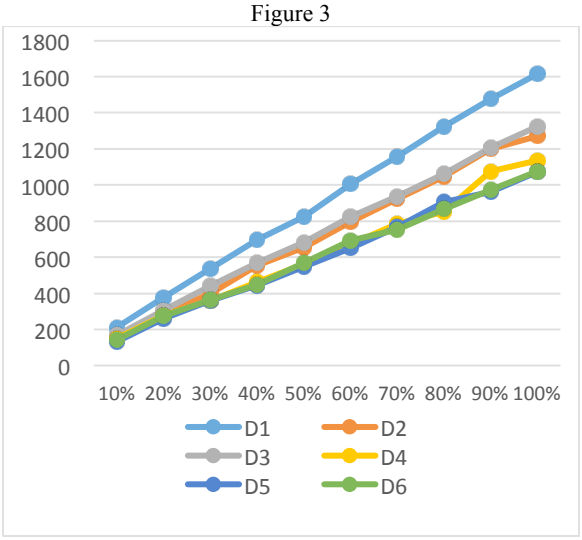
The dataset size of the original structure of this Figure is 1000. For other different structures we create, we get different number of nodes, edge and attributes. Each line stands for one structure. From this graph, we get query execution time for the query 1 using different structures. The line's trend meet our assumptions, with the growth of the selection factor, the line show the nearly linear increase, and the difference between lines also increase. For the structure 4, 5, 6, the line looks overlap by each other.

Figure 2



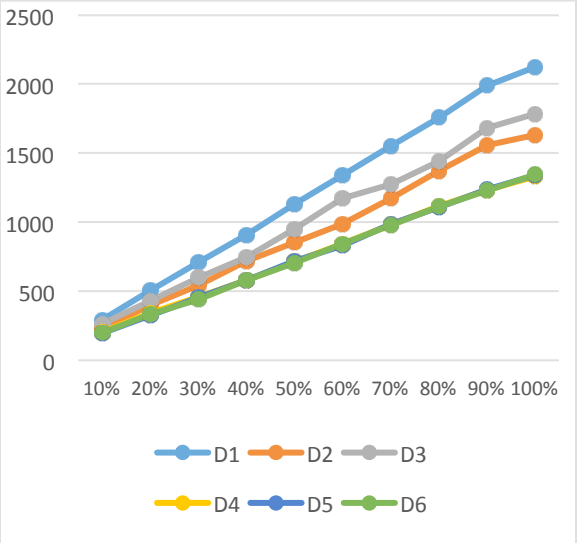
The dataset size of the original structure of this graph is 2000. From this graph, we get query execution time for the query 1 using different structures. What we find are same with the first graph, except that the line for the structure 4, 5, 6 are not looks overlap to each other.

Figure 3



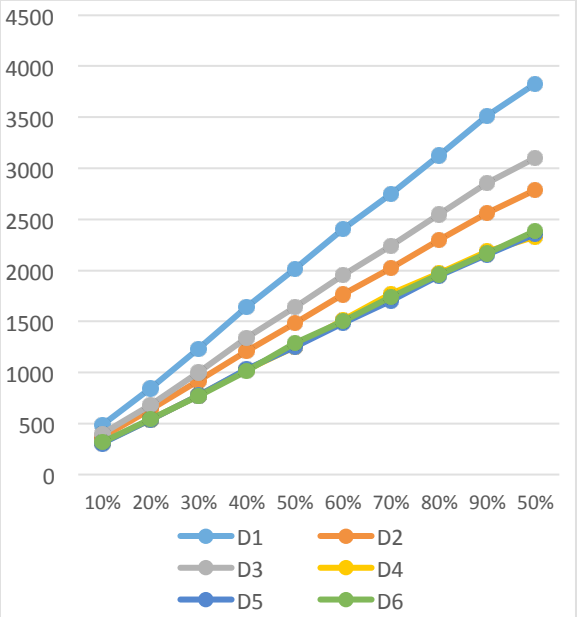
The dataset size of the original structure of Figure 3 is 3000. From this graph, we get query execution time for the query 1 using different structures. What we find are same with the first graph. But compared the coordinate axis with the first and second graph, we find that the coordinate axis in this graph is much bigger than the previous two, this means that gap between each line in this graph also bigger than that from previous two graphs.

Figure 4



The dataset size of the original structure of for Figure 4 is 4000. From this graph, we get query execution time for the query 1 using different structures. What we find are same with the third graph, and the gap between each line in this graph also bigger than that from previous three graphs.

Figure 5



Comparing these four graphs, we find that the bigger dataset size, the bigger gap between these lines that stands for the different structure. We also find that, for all graphs, with the growth of the selection factors, the gap between all of line become bigger. The reason we think is since there are 4 attributes in each person, if we extract the same attribute out as the index, the traversal will decrease a lot by the growth of the selection factors.

4.2.2 Query 3

Figure 6

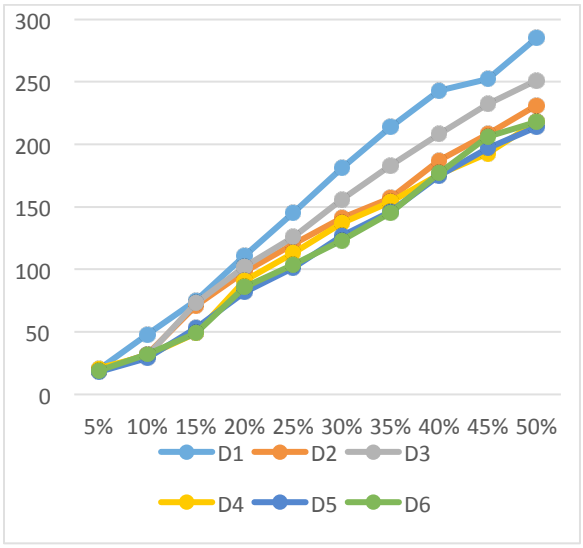


Figure 7

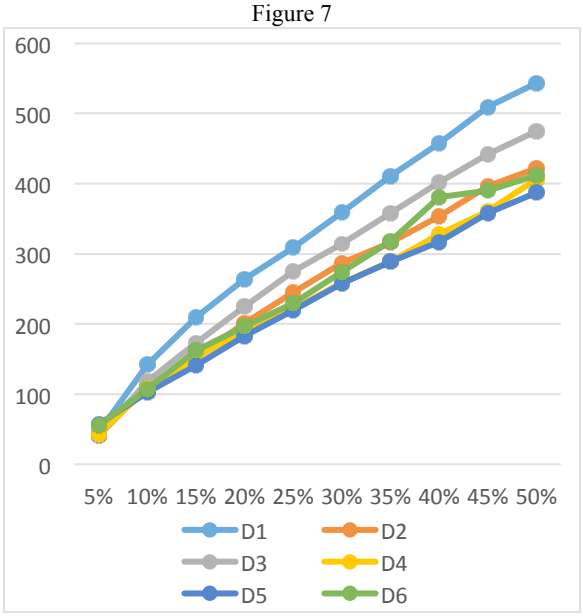


Figure 8

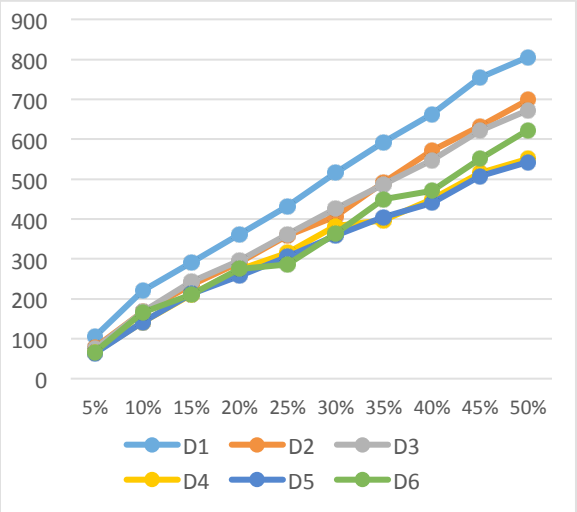


Figure 9

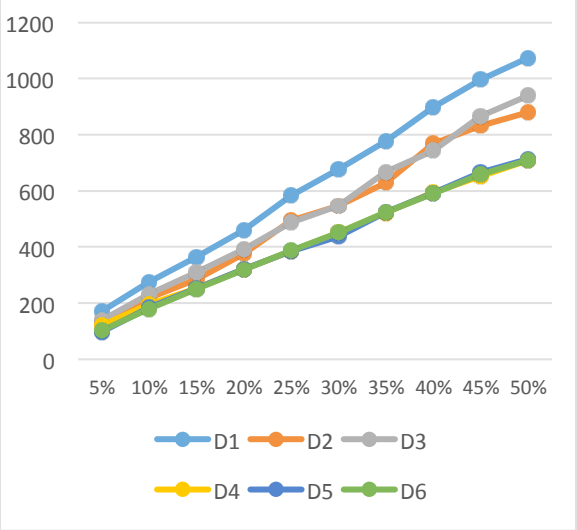
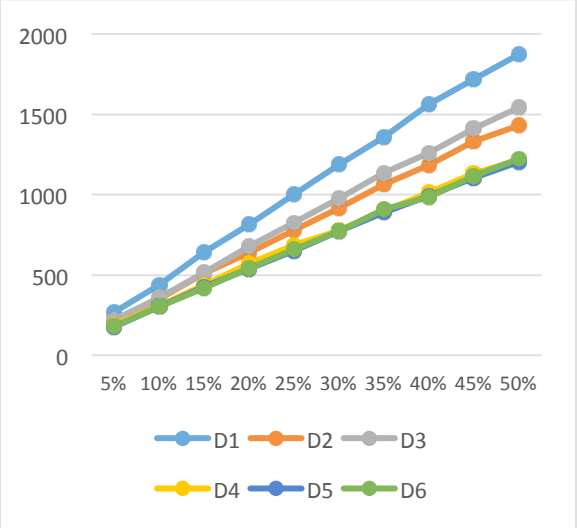


Figure 10



The graph 6 – 10 shows the query 3 execution time of the dataset size from 1000 to 5000. The query time trend still maintain its linear property. All trends and finds we get from these graphs are same with the first set of 5 graphs.

Compare different dataset scale

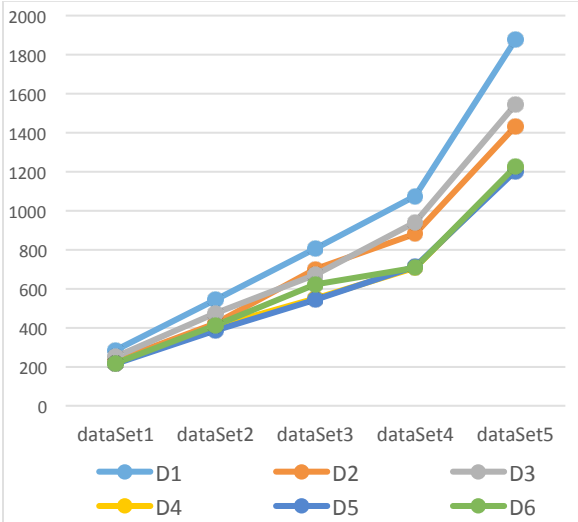


Figure 11: compare the average response time with different dataset using a query of 50% selection factor.

4.2.3 Query 2

For query 2, we query for all the "Person" nodes' "gender" attribute. Figure 12 shows the result when we only query for "Male" gender nodes. Figure 13 shows the result when we query for both "Male" and "Female" gender nodes.

Figure 12

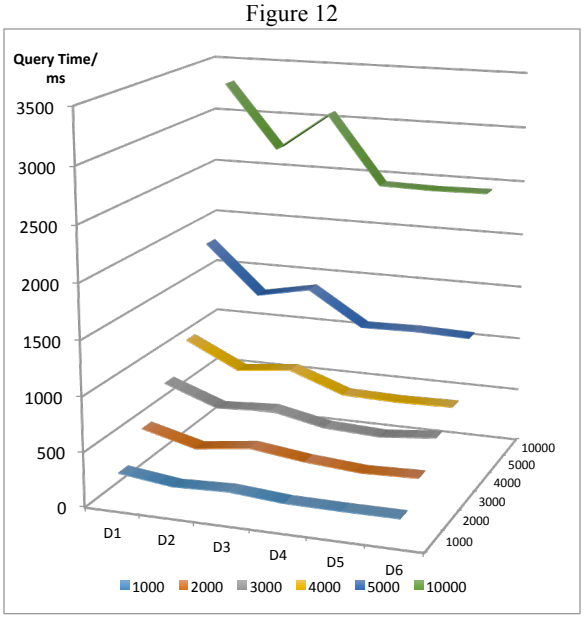
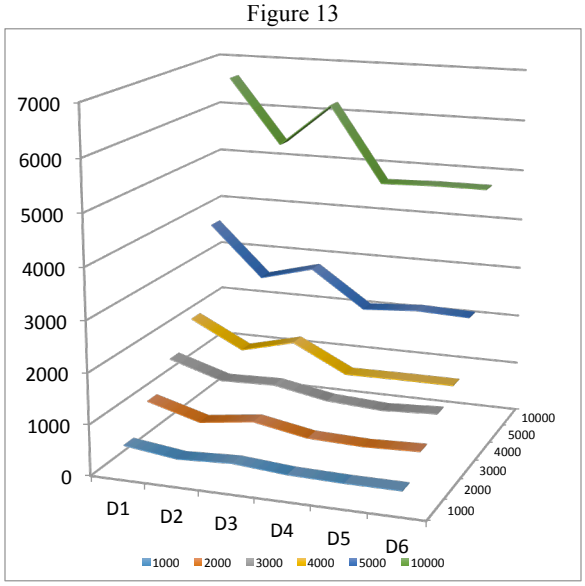


Figure 13



Both of two figures show the increasing trend when the dataset size increasing. Both the two figures show the result that the databases have worse performance on third data structure. Recall that 3rd data structure are constructed by extract only age attributes to build new nodes, it is obvious why this query will have worse performance on this data structure.

4.2.4 Query 4

Figure 14

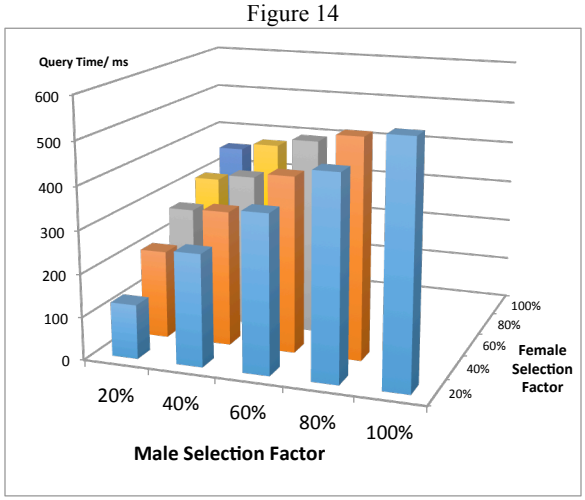


Figure 14 shows when query both gender and age, how the original graph performance. When mixed query both gender and age, the trends are still increasing with increasing selection factors. This only performed under original data structure, the other data structures also show the same trend.

Figure 15

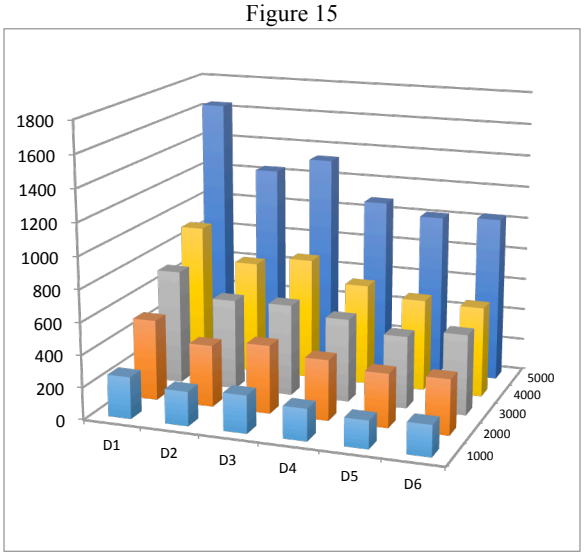


Figure 15 shows the query time on data structure from D1 to D6. We can see that the third data structure also shows a drawback on mix query properties. We also see that on larger dataset, D5 and D6 have almost same performance level.

5 Conclusion and Future Work

5.1 Conclusion

In this paper we designed experiments to test if the mechanism to extract attributes out will have better performance. Based on the data we obtained, we have following conclusion:

1. As the selection factor grows linearly, the average response time also increases proportionally. Similarly, if the size of dataset grows linearly, the average response querying time increases linearly as well.

2. Those structures using popular attributes as index performed obviously better than the original structure. The structure (database 5 and 6) with 2-layer index returns lowest average response time thus has best performance.

3. When the selection factor reaches nearly 100 %, the data structure 5 and 6 will have much more edges than the original discrete nodes. However, query 5 and 6 still maintain a better performance than the original data structure. An assumption is that Neo4j will table scan all the attributes in the nodes to match the search properties. In query 5 and 6, we extract the attributes out and form a better tree structure. This action avoids many void matching in the table scan.

4. When query for only one attributes, using both of two attributes to construct database will also return a better result than only extract one kind of attribute. The reason should be the same as last result. Neo4j's internal table

scan mechanism is always worse than we extract the attributes out and build indexing nodes.

5.2 Future work

We have found that Neo4j's internal nodes traverse algorithm will have performance worse than building new index for the original graph. But the dataset size is less than 110k. Based on Marek Ciglan, Alex Averbuch and Ladialav Hluchy's [9] work, traversal will become worse when the insert edges exceed 110k. Next step, the performance analysis on larger size can be researched.

6 References

[1] Angles, Renzo. "A Comparison of Current Graph Database Models." 2012 IEEE 28th International Conference on Data Engineering Workshops.

[2] Dominguez-Sal, D., P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. "Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark." Web-Age Information Management Lecture Notes in Computer Science: 37-48.

[3] Wikipedia. https://en.wikipedia.org/wiki/Neo4j

[4] Bonnici, Vincenzo, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. "Enhancing Graph Database Indexing by Suffix Tree Structure." Pattern Recognition in Bioinformatics Lecture Notes in Computer Science: 195-203.

[5] Sakr, Sherif, and Ghazi Al‐Naymat. "Graph Indexing and Querying: A Review." Int J of Web Info Systems International Journal of Web Information Systems, 2010, 101-20.

[6] Klein, Karsten, Nils Kriege, and Petra Mutzel. "CT-index: Fingerprint-based Graph Indexing Combining Cycles and Trees." 2011 IEEE 27th International Conference on Data Engineering.

[7] Williams, David W., Jun Huan, and Wei Wang. "Graph Database Indexing Using Structured Graph Decomposition." 2007 IEEE 23rd International Conference on Data Engineering.

[8] Dominguez-Sal, D., P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. "Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark." Web-Age Information Management Lecture Notes in Computer Science: 37-48.

[9] Ciglan, Marek, Alex Averbuch, and Ladialav Hluchy. "Benchmarking Traversal Operations over Graph Databases." 2012 IEEE 28th International Conference on Data Engineering Workshops.