Why Queries?

Abstract—Any user that queries any type of database expects at least one record to be returned. If the results are not what is expected, altering the query is the next logical step, maybe changing the way an item is spelled or choosing a different department to query. This approach is similar to trial and error. These alterations to the query do not help the user to understand a database better.

This capstone aims at analyzing and implementing at least one state-of-the-art algorithm on ‘Why? Queries’ to provide insight to users about unexpected answers.

Index Terms—Why Queries?; Why Empty?; Why so many?

I. INTRODUCTION

The human brain has always been curious, if something doesn't happen the way we want it to, it starts to think of reasons why to try and get to the bottom of the problem. Questions like "Why?" and "How?" have formed the basis for scientific exploration.

With this in mind, consider the following problems:

- A user is searching for some item on Amazon, he remembers seeing the item on sale somewhere, but now he can’t seem to find it.
- A football talent scout wants to know which new player would be a good fit for the team. So tries to figure out which players have already played with the players in his team before. But when he queries the database he has too many results.

Let us break down the two problems. For the first problem, maybe the item in question isn’t available in the same country that the user is in, maybe the seller stopped making the item, these could be some of the possible reasons. Similarly for the second, maybe the search the scout made has had some error. Observe that in both these cases the user can never know for sure why he faced this issue, mainly because he is not an experienced database user. So it would be helpful if Amazon or whatever service a user is using could answer these questions for us and the user can have a better understanding of where he went wrong.

Let us now take a look at this problem from a graphical perspective. Medical researches often make use of Graph databases to store the information of protein-coding genes. There are an estimated 19,000-20,000 human protein-coding genes[4]. Similarly there are such protein-coding gene sequences in different animals. So suppose that a researcher is trying to find if a human and lets say a giraffe have any sequences in common. The researcher chooses one human sequence and tries to find a match in the giraffe sequence database. If there is no match then the researcher would want to know why there was no match. Specifically, what part of our query sequence caused the answer to be empty. This is a key piece of information for the researcher and since there are millions of such sequences, it would be impossible to do it manually. We can see an example of this in figure 1, where only a part of the query graph matches with the data. This part is shown in blue in the result section, whereas the red represent the part of the query that didn’t match in the data.

This work is thus inspired by such Why? Queries.

II. RELATED WORK

Why Not? This question was first asked by Adriane Chapman and H.V. Jagadish in their paper titled 'Why Not?'[1] in Relational Database Management Systems. Their aim was to provide answer to the user when they encounter an unexpectedly missing result. Since the structure of the database is hidden from the user and/or the application limit the type of query the user can submit, they proposed a novel algorithm which would answer the question why not an item of interest to the user was included in the result set. After running the query and obtaining a result set, the user can enter what they were seeking from the query in the form of key or attribute values and the algorithm returns the series of manipulations which filtered out that result from the final set. A manipulation is one operation performed on the dataset. So a selection or a join is considered to be one manipulation.

The one and only papers researching Why? Queries in Graph Databases by Vasilyeva et al[2] expands on their previous work to answer Why Empty?, Why so many?, Why not? and Why so few? questions. For answering Why Empty?, they first detect the maximum common connected subgraph (MCCS) by using a modified version of McGregor's algorithm and then calculate the differential graph which tells
the user which part of the query graph failed to generate the required result. For answering Why so many?, the process remains the same except instead of finding the MCCS, we detect cardinality-bound MCCS and then the differential graph answers the required question.

III. PROBLEMS

Let’s now understand the type of problem we are aiming to solve.

A. Why Empty?

If the answer to a query is empty, meaning that there is no data fulfilling the structure of our query graph, but we were expecting at least some output then we ask the question ‘Why Empty?’

IV. STATE-OF-THE-ART ALGORITHMS

The first ever approach for answering Why Queries in a RDBMS was proposed in 2009 and since then there has been a lot of research in this field and there have been multiple different approaches taken to solve this problem. But the same cannot be said for Graph Databases. The only approach for answering Why Queries has been proposed by Vasilyeva et al in 2016. So this is the state-of-the-art approach and our main algorithm which we will be implementing in this Capstone. We judge the algorithm by performing by performing various tests on it to ensure that different types of queries can be executed.

V. RDBMS VS GRAPH DATABASES

For answering Why Queries in RDBMS, a lineage or a successor approach is followed. What this essentially means is that we either start at the end and trace the execution all the way to the start, a Bottom-Up approach[1], or we start at the beginning and run it all the way to the end, a Top-Down approach[1]. Top-Down approach is shown to produce better results by Chapman et al.

Since RDBMS has a uniform structure it is possible to perform such linear approaches to solve the required problem. For us, graph databases do not have a uniform structure a linear approach cannot be followed. And hence a novel approach of differential graphs is proposed by Vasilyeva et al. Simply put, differential graphs is that part of the query graphs which lead to the unexpected result.

VI. ALGORITHM

A. Foundation

To understand the solutions, first we need to understand a few basic concepts.

Definition 1 Connected subgraph - A connected subgraph is a non-empty graph whose every vertex and edge are contained in the respective graph. A connected graph is a graph where there exists a path from each vertex to every other vertex.

Definition 2 Common connected subgraph - Given a data graph and a query graph, a graph $G_d$ is a common connected subgraph of the data and the query graph iff $G_d$ is a connected subgraph of both the data and the query graph. It is also important to note that there may be multiple common connected subgraphs.

Definition 3 Maximum common connected subgraph (MCCS) - A maximum common connected graph is the common connected subgraph of the data and the query graph but it is not a subset of any other common connected subgraph.

The discovery of the MCCS is crucial to the algorithms. For this reason Vasilyeva et al have experimented with Ulmann’s and Macgregor’s algorithm and found that Macgregor’s algorithm performs better and saves space.

Figure 2 shows an example of computing an MCCS. After trying to match the query(a) on the data graph(b), we see that the whole query cannot be matched in the data graph. But we can find the common connected subgraphs shown in (c). Out of these only the last one is our required MCCS.

Algorithm 1 GraphMCS.

1: function mcsSize(circuit(query graph $G_q$))
2: result graphs $G_d$
3: maximum subgraph result
4: for all edge $e$ in $G_d$
5: if source() $==$ get sources for edge
6: for all source vertex $e$ source() do
7: $G_d$ $=$ $\varnothing$
8: $G_d$ $==$ DFS(sourceVertex, edge, true graph)
9: $G_d$ $=$ graph
10: for all graph in $G_d$
11: if size(graph) $\geq$ size(result) then result $==$ graph
12: return result
13: if isStart then edge $==$ get next edge for edge
14: if no further edge then return graph
15: target() $=$ traverse edge from source
16: $G_d$ $=$ filter target()
17: for all target vertex $\in$ target() do
18: extend graph with edge (source, target vertex)
19: $G_d$ $==$ DFS(targetVertex, edge, false graph)
20: return $G_d$

Fig. 2. Finding a MCCS

B. Why Empty?

Algorithm 1 [2] shown in Figure 3 developed by Vasilyeva et al, considers each vertex of the query graph as the starting
point of a DFS search, lines 4-5, and tries to find the MCCS. To do this it first computes the search space by mapping the elements of the query graph onto the data graph, line 5. It runs a DFS search for each of those mappings as shown in line 6-9, and then updating the current MCCS if the size of the result graph is greater, lines 10-11.

Let’s take a look at the actual DFS to find the MCCS. The algorithm first starts by trying to fetch the next edge in the query graph if the current node is not the starting point, line 14. The algorithm takes one step from the source vertex with the required edge and fetches all the target nodes associated with it, line 16. An important thing to keep in mind here is that the data graph is considered to be undirected for the purposes of getting an MCCS while the query graph is considered to be directed. Once we have the target nodes they are filtered with the predicates in our query graphs, line 17. It continues to run the DFS for all the target nodes that have matched our predicates by choosing the target node as the source vertex for the next iteration of the DFS. The algorithm ends when there is no more edges left in the query graph to traverse.

C. Differential Graph

After getting the resulting graph from MCCSSearch, we now need to compute the differential graph which is our required output. To do this we simply, subtract the parts of the query graph with our result graph. Once all the vertices and edges are removed the query graph remaining is our answer. So if subtract the MCCS from Figure 2 from the query graph, we compute figure 4 as our differential graph.

D. Restart

There are some problems with the base version of the algorithm as highlighted by Problems section. Particularly problems C and D are the ones which lead to a MCCS which is smaller than the actual MCCS. These problems arise because the current algorithm only looks in the forward direction. To solve this problem we will maintain a list of traversed nodes while performing the DFS to obtain our MCCS result. Now at the end we will restart the search using only the vertices which are not in the list of traversed nodes. So in the case of fig. 8, we will obtain the white part as our output in the first try and A, B and C query nodes will be the ones which are still unvisited. We will restart the query again from these nodes to obtain a secondary Common Connected Subgraph which will be appended to our current MCCS to obtain the final result.

E. Implementation

For the implementation and testing of the above algorithm we have chosen to use Neo4j. It is a graph database management system developed by Neo4j Inc. We will connect to our database using the Neo4j JAVA driver api which is officially supported by Neo4j Inc.

VII. Problems

While and after implementing the above algorithm we have found many problems. While we managed to fix some of those, there are still significant ones which prevent us from deploying this approach on a live dataset. Problems A and B arise because Vasilyeva et al. make an underlying assumption that the nodes in the database will be unique. This assumption is not valid in the case of a proteins sequence database, where there are multiple repetitions of a node. Problem A can be fixed by modifying the algorithm a bit so that when the DFS reaches an end node in its path instead of returning it will store the current graph as a new graph in a list of graphs. So instead of just one MCCS, a list of graphs will be returned from the DFS. The rest of the process remains the same. Due to problem B, all the nodes in the data graph needs to have a unique value. Hence we are unable to apply this algorithm to a real life protein synthesis dataset as of now.

A. Multiple MCCSs

If we look at the example from Figure 5, the query graph has two possible MCCS matches in the data graph. These are B->C->D and B->C->F. Both of these have length of 3 and should be considered as candidates for the final MCCS but the algorithm only returns the last MCCS which it finds, which in this case is B->C->F.

B. Wrong MCCS

Figure 6 shows the second problem with the algorithm. We see that there are multiple matches for node B which are added to the resulting graph.

C. Special Case

Figure 7 shows a unique case of a query graph where there the algorithm is not able to figure out that A, B and C are actually connected since the path from nodes A and C end at B. This problem occurs in the base version of our algorithm and can be fixed by using a restart strategy.
D. Bridge Structure

Suppose if our query and the data graph are the same and have the structure shown in fig.8. The only difference being that the data graph doesn’t have the edge ‘e4’. The base version of our current approach will only be able to find the white part as the MCCS and it will miss out on the gray part of graph. To ensure that it is able to find the gray part of the structure as well, we will use a restart strategy mentioned in Restart subsection of our Algorithm.

VIII. Evaluation Setup

As mentioned in the problems section, because of the restrictions we have to impose on the data graphs, we need to have a dataset which also complements these restrictions. As such we have decided to generate a custom database to suit our needs.

For the first phase, we create a dataset where each node has an attribute called ‘name’ which has a value from A to Z after which it repeats the letters with numbers (A1 to Z1, A2 to Z2...). Each edge has a value which is the combination of the name attribute of the two vertices it is connecting. So an edge between nodes ‘A’ and ‘B’ will have the edge as AB.

we have evaluated the base and restart version of the algorithm against the different configurations shown in fig.10.
we create a query which connects all nodes present in the
database, such that if there are \( n \) nodes query graph there
there will be at least \( n \) edges, such that it creates a cycle.

IX. RESULTS

Let’s take a look at the results from the first phase. Figure 12 shows the graph of time and the size of MCS i.e. the number of query nodes it discovered for our baseline approach. This is where we discover problem C mentioned in the problem section when we run query configuration 4. Even simple query configurations like 3 take a long time to be discovered in the data graph. So, bigger configurations like 1 take a lot more time. The only fast computation is for queries of type 2 which are simple straight path queries. DFS performs well in those situations. Typically query 2 configurations will occur in an airline connections database. So supposing you want to find flights which connect to multiple destinations, that would generate a query of type 2 and this is where, if there is an empty result, this approach will be able to do well.

Figure 13 shows the graph of time and the size of MCS i.e. the number of query nodes it discovered for our baseline coupled with the restart approach. We can see that the time required for computing the actual MCCS has increased, because of another heavy DFS operation being added to the original approach. But the plus side to this is that we now can discover bigger MCCS. The results follow similar trend as our baseline approach.

Finally let’s take a look at the results from the second phase. The goal here is to find which node from the data graph was deleted. We create a query graph which has the same structure as the data graph.

We can see from the above two figures 14 and 15 the executions times for our two data graphs D1 and D2. The K value on the X axis are the number of nodes that were removed from the data base to carry out the computation. As we can see that the execution times for computing MCCS increase by a big margin when going from \( K = 1 \) to \( K = 2 \). This phenomenon is mostly occurring due to the fact that the data graph might have been broken into two when removing two nodes and hence the restart strategy takes up more time. We can also see that there is a huge increase in the execution times for D2 when compared to D1.

X. CONCLUSION

The authors Vasilyeva et al. use a graph database know as GRATIN [4] which is not open source. This is an in-memory column oriented graph database designed for efficient traversals. Hence the fetching time is significantly faster than using Neo4j. Also, the authors perform many more optimizations like the restart strategy, one of which is computing a spanning tree for the query graph to find the optimal node to start the MCCS search on. The main component which has a heavy

![Fig. 13. First phase evaluations for Base line + Restart approach](image)

![Fig. 14. Second Phase evaluations for data graph D1](image)

![Fig. 15. Second Phase evaluations for data graph D2](image)
impact on the execution time for us is the computation of MCCS. We run the search for each of the query nodes. Even if we end up finding the MCCS from the first node, we still go through all the extra computations, because there is no sure way to figure out if we have a MCCS or not.

The only downside to the authors approach is that, we need to first convert the data base into GRATIN format since GRATIN is not a traditional graph database, which is freely available.

After performing the above experiments, we have come to the conclusion that Neo4j is not an ideal graph database to perform Why Queries on. The main reason being that Neo4j is not an in-memory database. The denser the graph gets the more the execution time increases with Neo4j. Also as the size of the query becomes bigger and bigger, the execution time will increase.

ACKNOWLEDGMENT

I would like to thank my advisor, Dr. Carols Rivero for the guidance and advice during the entire duration of the implementation of the project. He has been really helpful throughout the semester and has been providing consistent feedback which made the work easy.

REFERENCES