Graph Query Optimization

Performance Analysis and Heuristics for Time Tree in Neo4j

Shiva Nalla
Dept. of Comp Sci.
Iowa State Univ.
shiva@iastate.edu

Reshma Kottamsetty
Dept. of Comp Eng.
Iowa State Univ.
reshmak@iastate.edu

Ahmed Shaik
Dept. of Comp Eng.
Iowa State Univ.
ahshaik@iastate.edu

Puneet Kaur
Dept. of Comp Eng.
Iowa State Univ.
puneetk@iastate.edu

***Abstract*— With the prevalence of graph databases in multiple domains and its vast applications, there is also an increased need for optimization in terms of representing the graph structure and querying in the best possible way. Most of the databases tend to have hierarchical and continuous data variables such as time. Ordering the events occurring over a period of time has huge applications in the area of time varying databases where each event can eventually be represented by a unique sequence of time ordered bits. However, in real time, performance time is what matters when the database is queried. This work tries to identify and evaluate the factors that can influence the performance of a time tree in the graph database. Experiments and trails are conducted on a database of Movies collected from the Neo4j repository. Analysing the observations and metrics from the experiment reveals that selectivity factor and number of nodes over a time range together have a profound effect on the performance of a time tree. A comparison of the metrics for both the Time Tree model and the original model with indexes is made for different types of queries which yielded interesting results. These results can be further utilized to come up with a heuristic factor to determine when to use a query with time tree modelling. The results also exhibit interesting findings about scenarios where nodes that are not connected to the time tree are fetched through queries using a time tree model.**

Keywords—Time Tree, Performance evaluation, Event ordering, Heuristic measure, Indexing

1. INTRODUCTION

Graph databases such as Neo4j are being widely used in many domains such as Semantic Web, Social Network Analysis, Biological networks and in many other applications. One of the main features of the graph database is its topology, which is the structure of the database. Unlike other database management systems where determining relationships need foreign keys or usage of map-reduce to maintain relationships between entities; they are treated as the first class citizens in Neo4j graph database. This topological structuring simplifies and lessens the number of operations that need to be performed when the database needs to run a query that includes different entities. This also allows the user/ application to identify relationships easily in highly connected networks.

Considering the capability and features of Neo4j, nodes that represent an ordering of events/ roles/ location hierarchy can also be represented as an ordered structure such as a tree connected by relationships that represent the hierarchy; thus providing a better ordering of events occurring over a duration of time or structuring employee nodes in a graph according to their roles. Significant related work in this area has been done by Graph Aware [3] who came up with a time tree model that allows user to represent events in an ordered structure over a period of time. However, does remodelling the graph structure to represent an ordered set of events/ people/ locations come at an additional cost? How does remodelling effects the two most important parameters for a graph database: a) size of the graph and b) Execution times over a frequent set of queries? On what factors does execution time in such a model depend on and also, when is it best to remodel the graph to represent the events as an ordered set. This research addresses some of the above problems and discusses the results hence obtained.

Existing works were mainly aimed at modelling the graph to represent an ordering; however, they have not discussed the impact on performance, and heuristics for a call to choose when to remodel the graph. Current research work focuses on the performance evaluation of models discussed above with specific consideration to the Time Tree model and also in determining the factors that influence queries on a Time Tree model. These results then can be used to investigate and identify a heuristic that can determine when to use such a model over a given graph structure.

Current contributions in this work include the performance evaluation of the time tree model with

the original graph database over a set of selected queries. Another significant result this work contributes are the factors on which the performance of both the original and the timetree model depend on which are a) Selectivity and b) Overall range/number of nodes being queried upon. The evaluation also illustrates how the timetree model performs in queries where multiple entities that are not directly associated to the timetree are queried upon.

The findings indicate that for the same selectivity factor, timetree will take more time to execute over a larger range of time values than a smaller range. This can be attributed to the increase in the number of relationships in the time tree that the database engine has to traverse to fetch the results. Also as the selectivity factor increases, a steady rise in the execution times has been observed. However, in the case of the original graph model, the results are not very consistent across the selectivity factor/range of values and differed significantly in certain cases. This can be ascribed to the caching scheme and indexing methods that Neo4j uses internally to manage data during its operation sequence, the specifics of which are not provided by Neo4j at this point of time. Furthermore, it is observed that in case of queries that include retrieval of entities that are not directly associated to the time tree, the original model and its queries has a better performance when compared to its corresponding time tree model. This indicates a fine tuning to the time tree model to identify such cases before hand, and build either a multi-mapped tree model, i.e. identifying dependency of such entities to the timetree and associating them accordingly or creating a multi-tree structure that has a tree ordering for all such identified entities. Such optimizations can enhance the performance of the queries in the timetree model which is planned for future work.

2. RELATED WORK

In [1], a prototype of a new query engine for the Sparksee graph database based on algebra of operations on sets of key-value pairs of nodes is presented. The new engine combines some of the regular relational database operations including recursive programs with extensions to collection processing and complex graph queries. The findings of the experiment show that most graph query operations can be efficiently expressed as semi-join programs that can optimize a query in terms of execution time and query operations. Semi-joins can be used in most of the traversal queries as they reduce significantly the size of the intermediate results, in contrast to joins, and allow optimizations based on the swapping of operations. Interactive queries of SNB workload generator and DBGEN datasets were used for analysis and testing. Results show that more than 50% of the operations are scan and semi-join, which is a potential optimization in the access methods to the KVP (key-value pairs) and in their interactions with semi-join. 90% of the semi-joins have a key as the left operand in the comparison, which is an object identifier from a persistent KVP; another optimization hint based on indexed access methods to the Sparksee graph storage. Some semi-join programs are large with lengths of up to 6 or 7 consecutive semi-joins which is another optimization area based on the reorganization of semi-join programs by applying the properties of semi-joins. While most of the graph traversal queries can be expressed as semi-join programs, there is considerable memory especially with large graphs that is used to store the data in the form of key value pairs and in maintaining indexes for faster retrieval. However, this opens up more research into providing best of both the RDBMS and graph database features as a hybrid. The future work of this proposal involves implementing the prototype of the query engine with full support of the algebra; and building a query plan optimization for semi-join programs combined with the non-relational extensions for the resolution of complex queries over large graphs.

In [2], the authors used a back-end based on the graph database Neo4j and compared the alternatives for querying data over versus traditional JPA implementation. They have analysed performance and programming effort for data back-ends for Apache Shindig. They analysed why the different approaches may often yield such diverging results concerning throughput. Comparing the existing open source JPA back-end using MySQL to their own implementation with Neo4j and Cypher, they were able to achieve performance improvements of one order of magnitude for queries that span multiple tables. They compared Cypher and Gremlin, two popular query languages and found out: Gremlin has very good performance benefits in FOAF queries than Cypher. Compared to native object access, Cypher is about two times slower, however, it is cleaner in terms of code maintainability as well as some efficiency in development time. Overall, Neo4j can be used as a high performance replacement for relational databases, especially when handling highly interconnected data. Using an embedded Neo4j instance can yield a very good performance when using data sets of limited size. For production environments with higher scalability requirements,

Neo4j's advanced versions promise high availability and scalability if needed. Their results show that the graph-based back- end can match and even outperform the traditional JPA implementation and that Cypher is a promising candidate for a standard graph query language, but still leaves room for improvements.7

In [3] the GraphAware TimeTree was proposed, which is a library that builds the time tree on-demand. It provides 2 options for creating a time tree. One by using REST API and the other by defining the respective attributes in the properties file. It supports resolutions of one year down to one millisecond and has time zone support as well as full support for attaching event nodes to on-demand time instants. They also had a GraphAware Framework that speeds up development with Neo4j by providing a platform for building useful generic as well as domain-specific functionality, analytical capabilities, (iterative) graph algorithms, etc. On a high level, there are two key pieces of functionality, GraphAware Server and GraphAware Runtime. GraphAware Server is a Neo4j server extension that allows developers to build (REST) APIs on top of Neo4j using Spring MVC, rather than JAX-RS. GraphAware Runtime is a runtime environment for both embedded and server deployments, which allows the use of pre-built as well as custom modules called "GraphAware Runtime Modules". These modules typically extend the core functionality of the database by transparently enriching/ modifying/ preventing ongoing transactions in real-time.

One of the major drawbacks is the addition of new nodes and edges to the graph structure for creating the time tree, which can be argued against the advantages it has such as ordering of nodes. However, the work done doesn't indicate when is it best to use a time tree model and the impact on the performance by modelling the graph structure into a time tree.

In [4], the author provides a holistic study of problems of querying graph databases. The emphasis is laid on expressiveness, complexity of evaluation and containment problems for various navigational query languages. The expressiveness deals with an expressive power of the language while containment is a problem of deciding whether answer set of first query is contained in the second one if two queries are present in a graph query language. The paper also provides overview of the path semantics of the graph database primarily simple and arbitrary path. The major contribution of this research is comprehensive study of complexity of evaluation and containment conducted for three groups of languages. First group is composed of traditional query languages for graph database such as G which is based on simple path semantics. On the other hand, there are number of navigational languages which are based on arbitrary paths like Regular path queries (2RPQs), conjunctive regular path queries (CRPQs), inverse (CRPQs) and unions (UC2RPQs) that enables the computation of data complexity. The class of acyclic C2RPQs and nested regular expressions (NREs) is also included in this group because combined complexity of C2RPQ is intractable. The containment is defined for each member in this group except that of G which is uncertain. Second group comprises of expressive languages for path queries such as extended conjunctive regular query language (ECRPs). It is particularly extension to CRP (conjunctive regular path queries) that enables the ability to output and compare paths. The containment is not retained for ECRPQs. The first two groups of graph database deal with the queries linked with graph topology only, while the third group involves languages that link graph topology with data. There is an expressive language called regular expressions with memory (REMs) where data is stored in registers. Thus the data is always available for comparison at any point. The REMs are tractable in complexity and containment is undecidable. Overall, the paper presents lucid vista of the evaluation of complexity and containment problems of various categories of the navigational query languages for graph databases.

3. PROPOSED WORK

Below is an idea of an optimization technique that can be applied to a graph database whose graph structure is known beforehand. Queries based on time or any attribute that can be represented in a hierarchical manner can be represented in the graph as a tree hierarchy and related nodes can be associated to the tree. The experimentation focuses primarily on timetree modelling. Modelling a graph structure as a time tree allows representing events in order of time; hence opening up many applications that can utilize the unique time ordered representation values obtained from the structure. However, a time tree may not always provide a better performance when compared to the original model because sometimes, a huge increase in nodes over the traversed path may take more time to execute.

A. Main Idea:

To find out the factors that influence the Time Tree performance in a Neo4j graph database, we

have re-model the graph structure as a Time Tree that allows us to represent events as nodes and link them to nodes representing instants of time. The idea is to develop a time tree and do a comparison of this time tree versus original graph model with indexes created on the time attributes. The proposed work does a performance evaluation of both using a set of queries that we developed. This is done to address the following issues:

i. What is the additional cost, if any, involved in building this time tree?
ii. What is the impact of time tree on execution time over a frequent set of queries?
iii. What are the factors that the execution time in this time tree depend on?
iv. Deciding when to use a time tree vs indexed graph?

The queries that we created are all time based and were executed in both indexed and time tree models. Each of these queries was executed several times by varying the selectivity factor and changing the time range involved in the query and taking several runs of the same query for better analysis. The results thus obtained can be used to determine what factors affect the execution time by an in-depth analysis of various factors such as Selectivity factor, Range of the query, Execution Time. The results can be further applied to finding a heuristic measure on when will a time tree perform better and when is it better to use indexed graph structure instead of time tree. The queries are also chosen in such as way so as to analyze the results when we use the timetree to query nodes that are not directly associated to the time tree.

So, given a graph structure and a set of queries dealing with time that are being planned to be executed, we can modify the graph schema by creating a time tree in the database and associating the related nodes to the time tree. A heuristic can be computed for each query depending upon the factors affecting its performance in a time tree model. The heuristic value will suggest if the time tree can give a better performance for the given query than the usual query. Based on the heuristic score, the query can be modified to make use of the time tree model. This is done for each of the queries and at the end, the new graph structure and the updated query set will be returned.

B. Description:

To study the time tree model, we used a graph database (Movies) with more than 12000 Movie nodes, each with a timestamp attribute 'Release Date' in milliseconds. Since most of the realistic time-related queries on such a database would be only to a level of a month and not beyond that, we created a time tree that supports a resolution of one year down to one month (this, however, can be drilled down to a level of one millisecond if so desired). The time tree constitutes a root node, years on the first level and months on the second level. Then we have attached the Movie nodes to their corresponding month nodes, based on their release dates. The figure below depicts the time tree structure:

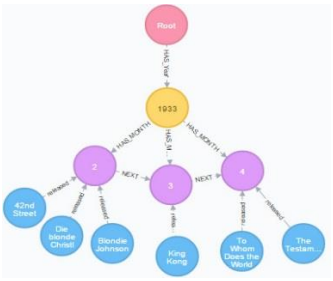

Fig.1: Time Tree Structure

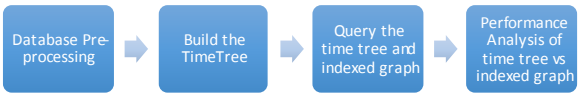

Fig.2: Block Diagram of Proposed Work

C. Database Pre-processing:

The database was initially checked for any invalid and outlier values for the "ReleaseDate" attribute. For example, there were certain movies with releaseDate indicating movies released in 1200's when the movie was actually released in 1800's. Such records were modified to reflect the actual values. Also, the releaseDate attribute for the Movie nodes in the Movie database is represented in milliseconds. However, to build the time tree we will need to have its corresponding year and month of release. As there are no time related functions in Neo4j that can extract the required values, we built a Java program that connects to the Neo4j movie database through a Jdbc connection and extracts the Movie nodes and creates the attributes "releaseYear" and "releaseMonth" on each Movie node. This completes the pre-processing step.

D. Build the TimeTree:

The time tree starts with a single root as we are

considering a single rooted timetree. The root will be connected by a "HAS_Year" relationship to all the year nodes ranging from the earliest "releaseYear" to the recent "releaseYear" value for the Movie nodes. Each year node has 12 child month nodes through a "HAS_MONTH" relationship. Each month node under a given year node has 'NEXT' relationship with the adjacent next higher month node. Also, the last month node of each year (12) has a 'NEXT' relationship with the first month (1) node of next higher year node. The purpose of this 'Next' relationship is to traverse through the entire time-tree. We have implemented two variations of time-tree that differ mainly in the way Movie nodes are attached to the time tree. They are: a) Pre-built time tree b) On-demand time tree.

In the Pre-built version, using a Java program, we have extracted the 'Release Date' timestamp attribute of all the Movie nodes that contains release date of the movie in milliseconds, to define two new attributes 'Release Year' and 'Release Month' for each Movie node. We have determined the range of years over which our Movie database spans, i.e., the lower bound and upper bound on movie release year attribute. Then the time tree is created that contains all the year nodes within this release year range. Finally, we have attached all the Movie nodes by a "released" relationship to their corresponding release month nodes of their release years. In the On-demand version, the time tree contains only those year and month nodes for which there is a movie to be attached (i.e., if there's at least one movie released in that year/month). Otherwise, that year won't be represented as a node in the time tree. This gives a timetree structure that is built on demand.

Since our Movie database is very huge, we found that there was at least one movie node for every month of most of the years in the ReleaseYear range and hence the advantage of building an on-demand time tree is still being met with the prebuilt version i.e., we realized that the on demand time tree diverges to a pre-built version when the database is very huge such that there is a movie attached to every leaf node of the time tree. So, we implemented queries on a pre-built time tree.

E. Query the TimeTree and Indexed graph models:

To study and analyse the behaviour of the time tree model, 5 types of queries that query over time are chosen. While some query over the Movie nodes which are directly associated with the Timetree by "released" relationship, others query over the Actors, Directors and Ratings which are not directly related to the time tree in the structure but are only connected to the Movie nodes. Also, it is ensured that all kinds of time range equality and inequality queries are covered in the queries.

F. Execution Performance and Analysis:

Each of these above queries is executed several times by varying the selectivity factor and changing the time range involved in the query and taking several runs of the same query for better analysis. The results are summarized and average performance times are computed in each case. Graphs are plotted with Selectivity (vs) Execution time and Time range factor (vs) Execution Time to study the impact of these factors on the performance of the Time Tree model and how they vary in each of the cases for both the TimeTree model and the original indexed graph model.

Software Components Used:

Neo4j, Java, Jdbc connection to Neo4j

4. PERFORMANCE EVALUATION

To evaluate the performance of the time tree, a comparative study and analysis has been done on the performance of 5 selected query types in both the models, original graph model with index on time attribute and time tree model respectively. For each type of query, the query is executed with different selectivity factors over a varied range of time values for 10 runs, excluding the first 5 runs each time to let the execution times stabilize. The average of the 10 execution times values is considered for each selectivity factor and range. The results obtained are plotted graphically with the selectivity factor against the execution time.

The "Movies" dataset consisting of 12862 movies along with its associated directors, actors and ratings obtained from Neo4j repository is used during evaluation. The dataset is initially manually pre-processed to check for outliers and invalid time values and nodes with such values are modified to reflect correct values. Then, attributes for releaseYear, releaseMonth are added through a Java program by establishing a Jdbc connection to Neo4j Movies database. Below are the 5 different query types selected for the performance evaluation of the time tree.

Query 1: All movies in a given month of a year (say May 1984)
Query 2: All movies in a given time range (say between 1950 and 1975 both inclusive)
Query 3: Names of the actors and the number of

movies they acted in a given time range (say between 1950 and 1975 both inclusive)
Query 4: List all directors for movies released before 1950
Query 5: List all 5 star rated movies after 2000

A. Cypher Queries in original model (with Index on "releaseDate" attribute)

*Query1***:** MATCH (mov: Movie)
where toInt(mov.releaseDate)>452235600000 and toInt(mov.releaseDate)<454913999000
return mov

Query2: MATCH (mov:Movie)
where toInt(mov.releaseDate)>-631130400000 and toInt(mov.releaseDate)<189323999000
return mov

*Query3:*MATCH(mov:Movie)<-[:ACTS_IN]-(actor:Actor)
where toInt(mov.releaseDate)>-631130400000 and toInt(mov.releaseDate)<189323999000
with actor.name as Name, count(*) as movieCount
return distinct Name,movieCount

*Query4:*MATCH (mov:Movie)<-[:DIRECTED]-(dir:Director)
where toInt(mov.releaseDate)<=-599594401000
return distinct dir.name

Query5: MATCH (mov:Movie)<-[rat:RATED]-(user:User) where toInt(mov.releaseDate) >= 978328799000
with mov as MovieName, avg(rat.stars) as avgRating where avgRating=5
return MovieName

B. Cypher Queries in TimeTree model

Query1: MATCH
(root)-[:HAS_Year]->(year1:Year {year:1984})-[:HAS_MONTH]->(start:Month{ month:1})<-[:released]-(movie:Movie)
return distinct movie

Query2: MATCH
(root)-[:HAS_Year]->(year1:Year {year:1950})-[:HAS_MONTH]->(start:Month {month:1}),
(root)-[:HAS_Year]->(year2:Year {year:1975})-[:HAS_MONTH]->(end:Month {month:12}),
p=(start)-[:NEXT*0..]->(end)
WITH NODES(p) AS months
UNWIND months AS month
OPTIONAL MATCH (month)<-[:released]-(movie:Movie)
return distinct movie

Query 3: MATCH
(root)-[:HAS_Year]->(year1:Year {year:1950})-[:HAS_MONTH]->(start:Month {month:1}),
(root)-[:HAS_Year]->(year2:Year {year:1975})-[:HAS_MONTH]->(end:Month {month:12}),
p=(start)-[:NEXT*0..]->(end)
WITH NODES(p) AS months
UNWIND months AS month
OPTIONAL MATCH (month)<-[:released]-(movie:Movie)<-[:ACTS_IN]-(actor:Actor)
with count(movie) as cnt, actor.name as name
return name,cnt order by cnt desc

Query 4: Match (yy:Year) with min(yy.year) as minYear
OPTIONAL MATCH
(root)-[:HAS_Year]->(year1:Year {year:minYear})-[:HAS_MONTH]->(start:Month {month:1}),
(root)-[:HAS_Year]->(year2:Year {year:1950})-[:HAS_MONTH]->(end:Month {month:12}),
p=(start)-[:NEXT*0..]->(end)
WITH NODES(p) AS months
UNWIND months AS month
OPTIONAL MATCH (month)<-[:released]-(movie:Movie)<-[:DIRECTED]-(dir:Director)
return distinct dir.name

Query 5: Match (yy:Year) with max(yy.year) as maxYear
OPTIONAL MATCH
(root)-[:HAS_Year]->(year1:Year {year:2000})-[:HAS_MONTH]->(start:Month {month:1}),
(root)-[:HAS_Year]-
>(year2:Year{year:maxYear})-[:HAS_MONTH]-
>(end:Month{month:12}), p=(start)-[:NEXT*0..]-
>(end)
WITH NODES(p) AS months
UNWIND months AS month
OPTIONAL MATCH (month)<-[:released]-(movie:Movie)<-[rate:RATED]-(user:User) with avg(rate.stars) as rateAvg,movie.title as movieName
where rateAvg=5
return distinct movieName,rateAvg

C. Graphical Plots:

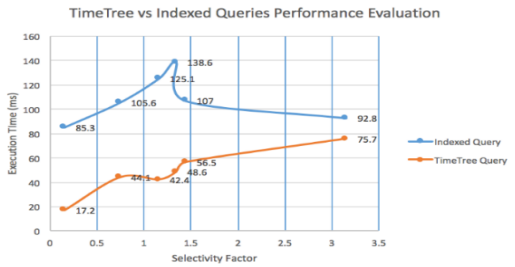

Fig.3: Query 1 – Selectivity vs Performance Time

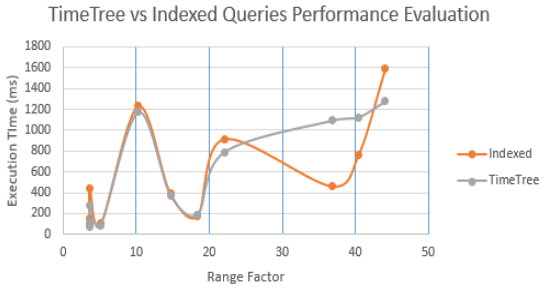

Fig.4: Query 2 - Range vs Performance Time

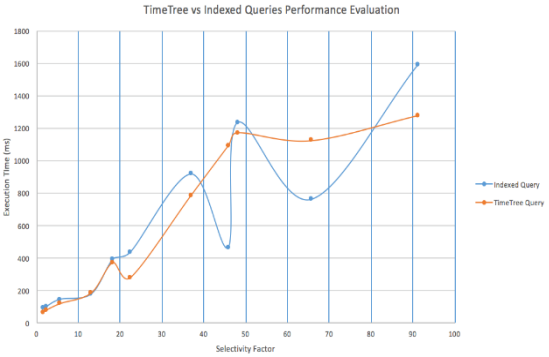

Fig.5: Query 2 - Selectivity vs Performance Time

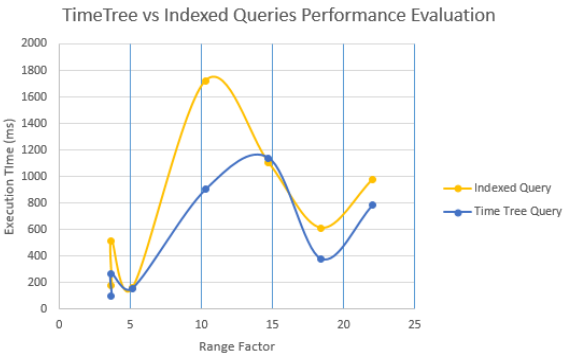

Fig.6: Query 3 - Range vs Performance Time

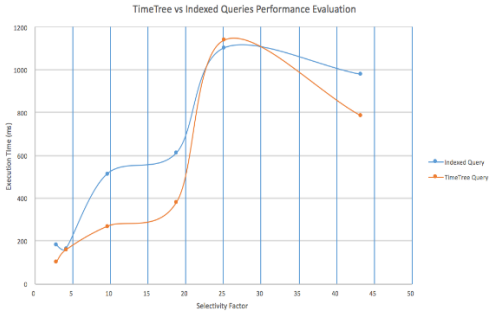

Fig.7: Query 3 - Selectivity vs Performance Time

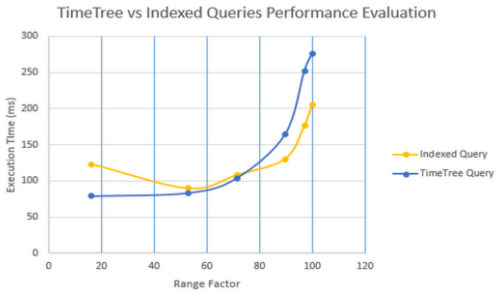

Fig.8: Query 4 – Range vs Performance Time

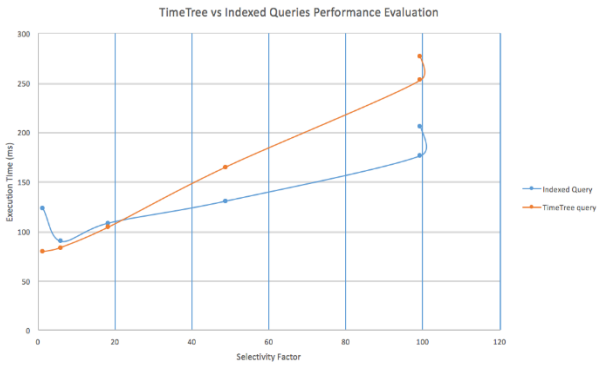

Fig.9: Query 4 - Selectivity vs Performance Time

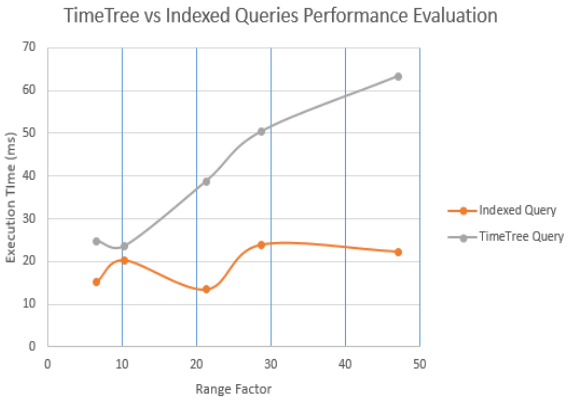

Fig.10: Query 5 – Range vs Performance Time

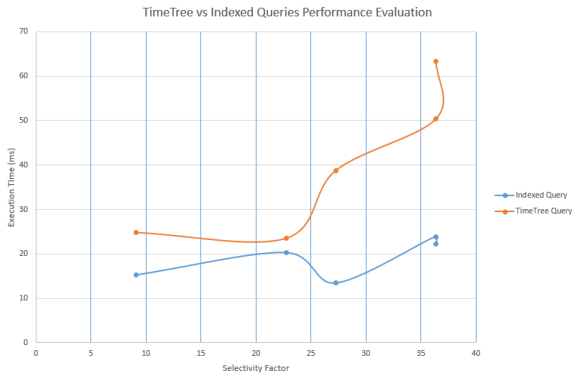

Fig.11: Query 5 - Selectivity vs Performance Time

The graphs plotted are with Selectivity factor (vs) Execution Time and Range factor (vs) Execution Time. Selectivity factor is the percentage of nodes selected from the available nodes whereas range factor is defined as the percentage of range being considered out of the available range of values. From the above graphical results, it can be clearly seen that the performance of the query depends on both Selectivity factor and Range factor. The rationale behind choosing this metrics is because Selectivity is one of the major factor for a database query in terms of query planning as it determines the number of nodes retrieved at each point of database filtering and finally retrieved by the query. Hence, checking for variation of performance time with Selectivity gives promising results. An analysis of the results revealed that performance has also been varying significantly based on the range of values/ nodes being queried upon and this is evident from the above graphical plots which explains the behaviour of variation of performance times relative to range of values searched upon.

The graphs for queries 1, 2 and 3 indicate that the performance of the time tree model is better than the original model when the range is less and the selectivity factor is varied. However, when range is increased significantly, the time tree shows equal or slightly lower performance compared to the original model. Also, results to the queries 4 and 5 show that the performance of queries fetching nodes related to the primary nodes that are linked to the timetree needs improvement and further optimization of the graph structure shall be done in those cases.

5. CONCLUSION AND FUTURE WORK

Motivated by the need to reduce time and cost of query plans in the graph database for query optimization, we have done a comparative study of performance of a Time Tree Model based on prebuilt approach to that of the original model with indexing approach. The performance evaluation results depict that the performance of the query is largely affected by the selectivity factor in both the techniques. The graph obtained through experiment presents a finding that the execution time of the query not only depends on the selectivity factor, but also depends on the number of nodes over the time range. Our analysis shows that the number of nodes in a particular range that a query needs to fetch plays a pivotal role in making a decision on which approach would be optimal depending upon how large the range is.

Though one of the major drawbacks of the time tree model is the increase in the number of nodes and the edges in the given graph structure, considering the performance gain and ability to produce an ordering of nodes based on time, it makes the worth. Future work in this area shall include coming up with a heuristic measure that can dynamically decide when to use a time tree so that given a set of queries, queries that have a better heuristic measure can make use of the time tree model. Another interesting finding is that in case of queries that fetch the nodes which are not directly associated with the time tree, the original indexed model performed better in most of the cases. This lays a direction to optimization of the time tree model by identifying such classes of nodes and building a multi-mapped time tree. The development of this multi-mapped tree model will lead to efficient query optimization due to the linking of the identified classes of nodes to the time tree and the association of multiple time trees with each other through joins. It could also be quite interesting to build advanced tree model dependent on location or combination of location and time attributes.

REFERENCES

[1] Martinez-Bazan Norbert and David Dominguez-Sal: "Using semi-join programs to solve traversal queries in graph databases" Proceedings of Workshop on Graph Data Management Experiences and Systems. ACM, 2014.

[2] Holzschuher, Florian, and René Peinl: "Performance of Graph query languages: Comparison of Cypher, Gremlin and Native Access in Neo4j" Proceedings of the Joint EDBT/ ICDT 2013 Workshops. ACM, 2013.

[3] GraphAware Neo4j TimeTree – Library for representing time in Neo4j as a tree of time instants, 2014.

[4] Barceló Baeza, Pablo: "Querying Graph databases" Proceedings of the 32nd symposium on Principles of database systems. ACM, 2013.