Abstract

Graph Databases are rapidly increasing in popularity, size and application. Currently, graph query processing involves some form of isomorphism test, which results in very high response times. Indexing is the most popular way to optimize query processing times. In this paper, we compare some of the existing work on subgraph query processing including cIndex, gIndex and FG-Index. There is a precision performance trade-off involved in subgraph queries on graph databases. There is a need to distinguish efficient querying methods tailored to certain applications. By analyzing the state of the art and comparing the methods in use, we can identify the key aspects in each and build a new indexing mechanism that can be adjusted according to the application and boost performance. This new index will map graphs in the dataset onto a plane and borrow some properties of similarity search techniques to greatly reduce the size of the candidate set of graphs on which the isomorphism test is performed.

1 Introduction

A graph database is a result of application of graph theoretic principles to data management and uses nodes, edges, and properties to represent and store data. There are several commercial graph databases available today including Neo4j, Titan, OrientDB, etc. General graph databases that can store any graph are distinct from specialized graph databases such as triplestores and network databases. Queries encountered on a graph database can be broadly categorized into subgraph, supergraph and similarity queries. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. However, query processing without indexing will require an enormous amount of subgraph isomorphism checks, which makes it the most inefficient method. Competent indices are necessary to bring graph queries on par with relational database queries even for moderately sized graph datasets.

One of the most common query on a graph database is the subgraph query; where, given a graph, it is required to determine a subgraph that satisfies certain properties. Subgraph queries include both exact querying, where we need to find exactly the same subgraph as the query q, and approximate solutions, where a subgraph close enough to q will be deemed satisfactory. Algorithms for queries of this type can be feature, closure or coding based, or verification based approaches. The first works in this area concentrated on filtering and candidate verification. The first process filters the dataset according to some query parameters. This filtered set of graphs is much smaller than the original dataset. The second step, candidate verification, is a subgraph isomorphism problem. Even if the number of graphs is greatly reduced, performing this operation every time the database is queried proves to be inexcusably expensive. Once it was accepted that the filter-verify approach is not producing results as expected, especially given the way datasets were expanding, different indexing approaches were proposed.

Through the course of this paper, we examine a few prevalent indexing techniques put forth for graph databases. They include the
cIndex, the GIndex and FG-Index. We also examine some supergraph indexes to see if they can be applied to subgraph queries in a bottom-up approach. The experimental results put forth in each paper can be used, albeit cautiously, as a yardstick to generally determine if some we can design an index structure that will yield better query performance even if it might take a significantly larger time to construct. With this goal in mind, we propose the Similarity Hash Index or SHIndex which is a 2 step Index process. Here, the first step is to map the graphs on a plane and then group them based on a similarity property into buckets. When a query comes in, we perform the same map and hash function on the query graph. This will severely limit the number of graphs in the candidate set and thus ensure that less time is spent on subgraph isomorphism. The progress on this area has been limited due to time constraints and the idea is still in its infancy. We have a long way to go before we can successfully implement this part of our project.

1.1 Paper layout

The layout of the paper is as follows. In section 2, we summarize some of the related works. In section 3 and 4, we describe the technical details including the algorithms, analysis and implementation details; and our evaluation and inference from the experiments, respectively. In section 5, we summarize our findings and in section 6, we describe some of the work to be carried out in the future. Section 7 contains the details of the work done by the team members.

2 Related Work

2.1 Exact Subgraph Queries

The most common type of queries on graph databases are exact subgraph queries. Given a Graph Database \( D = \{g_1, g_2, ..., g_N\} \), and a smaller graph \( q \), we retrieve all graphs in \( D \) that are supergraphs of \( q \). The result set here is more restricted than in the case of approximate matching, but the downside is the extra computation involved in locating the exact matches. Once again the naïve approach of filter and verify is proved ineffective. The common approaches used in this area include a path based or frequent structure based mechanisms that grow very quickly. One example here is the gIndex which involves too many candidate verifications for one’s comfort. The bottleneck in this method is the candidate verification, and, following Amdahl, we have to try and minimize, if not outright eliminate that expensive step. The idea proposed by Cheng et. al. is a step in this direction.

FG-Index uses the data mining technique of Frequent Subgraph Mining to collect a set of subgraphs that occur very frequently in the dataset and so might be queried very frequently as well. This set of graphs is called the frequent graph set. The size of this frequent graph set can be determined by a user-defined threshold \( \sigma \). An FG is a subgraph of at least \((\sigma |D|)\) graphs in \( D \). The paper also uses the concept of \( \delta \)-tolerance to reduce the overall cardinality of the set of graphs in the index. This set is called the \( \delta \)-Tolerant Closed Frequent Subgraph set. A graph \( g \) is a \( \delta \)-TCFG \( \iff \) \( g \in F \) and there is no \( g' \in F \) such that \( g' \supseteq g \) and \( \text{freq}(g') \geq (1-\delta)\text{freq}(g) \), where \((0 \leq \delta \leq 1)\). Figure 1 shows an example of a \( \delta \)-TCFG.

An FG-index consists of the core FG-index and Edge-index. The core FGIndex consists of two parts. First, a memory-resident inverted-index is constructed on the set \( \tau \) of \( \delta \)-TCFG’s. These consist of the most frequent graphs in the database. Many smaller subgraphs might be contained in these set of graphs. They are indexed separately in a disk-resident inverted-index, built on the FGS in the closure of each \( \delta \)-TCFG. This index structure will allow us to query the most frequent subgraphs without performing candidate verification. To allow for queries on the graphs that have been left out, another index, called the Edge-index, is built on the set of infrequent distinct edges in \( D \).
index allows us to store the differences between a frequent and infrequent graph. Figure 2 shows an example of the FG-Index.

Given a query graph \( q \), we first search for it in the core FG-index. If \( q \) is a \( \delta \)-TCFG, \( q \) and \( D_q \) are retrieved from the memory-resident index. Otherwise, we need to find \( q_s \) closest \( \delta \)-TCFG supergraph \( g \)'s disk-resident index and retrieve \( D_q \). However, if \( q \) happens to be an infrequent subgraph, and is thus not found after both of the previous steps, we get a set of subgraphs \( S_{\text{core}} \) of \( q \) from the core index and a set \( S_{\text{edge}} \) of infrequent edges from the edge index; and compute the intersections to get a candidate set \( C_q \). Candidate verification is now performed on \( C_q \) to weed out false positives. Thu, FG-Index is an efficient way to minimize the number of candidate verifications required for an exact subgraph search. However, the efficiency of this method is dictated by choices for \( \sigma \) and \( \delta \).

Correlation Search proposes a new concept of mining data from graph databases called the Correlated Graph Search. However this concept is challenged by the problem of efficiency since every subgraph of a graph in the database is a candidate and the number of subgraphs is exponential in nature. This is addressed by an algorithm which operates by mining a much smaller database, thus obtaining a significantly smaller set of candidates. We optimize this further by applying some heuristic rules to further reduce the search space. Correlation mining is mainly used for identifying the underlying dependency between objects. It is used in a variety of applications and has been researched extensively on market basket databases.

Given a graph database \( D \) consisting of \( N \) Graphs, a query \( q \) and a minimum correlation threshold \( \nu \), the correlated graph search finds all graphs that have Pearson coefficient with respect to \( q \) no less than \( \nu \). The Pearson coefficient is used to describe the strength of the correlation among Boolean variables in a transaction database. The paper addresses the problem of reducing the large search space of CGS and avoids as many expensive graph operations as possible. This is done by deriving the lower and upper bound of the occurrence probability of a candidate graph \( g \). This reduces the search space to be the set of Frequent subgraphs in the database \( D \) with the support values between the lower and upper bounds of \( \text{supp}(g) \). The Pearson correlation coefficient for two given graphs \( g_1 \) and \( g_2 \) is defined as follows.

\[
\rho(g_1, g_2) = \frac{\text{supp}(g_1, g_2) - \text{supp}(g_1)\text{supp}(g_2)}{\sqrt{\text{supp}(g_1)\text{supp}(g_2)(1 - \text{supp}(g_1))(1 - \text{supp}(g_2))}}
\]

With this in mind, we move on to the next phase of obtaining the set of candidate graphs for a given graph database. This is done by generating the candidates by mining frequent graphs by using the value of the lower(\( q, g \)) divided by the \( \text{supp}(q) \) as the minimum support threshold. A generated candidate set, \( C \), is said to be complete with respect to \( q \) if \( g \) belongs to \( \text{base}(Aq) \) and \( g \) belongs to \( C \). This implies that we can mine the set of candidate graphs from a much smaller projected database with a greater minimum support threshold. We then go on to apply the heuristic rules, obtain the much smaller projected database and check the correlation condition of \( g \) with respect to \( q \) to produce the answer set. We give a small of explanation of the Heuristic rules used below. Heuristic 1 says that if we find a graph \( g \) in the candidate set to be a supergraph of \( q \), we can add \( (g, D_g) \) into the answer set without checking the correlation condition. Heuristic 2 says that if a graph \( g \) is uncorrelated with \( q \), we can prune all the subgraphs of \( g \) that have the same support as \( g \) in \( D_q \). Heuristic 3 says that if we can find a graph \( g \) to be uncorrelated with \( q \), we can prune all the subgraphs of \( g \) that has support values less than the value of \( f(\text{supp}(g)) \) divided by \( \text{supp}(q) \) in \( D_q \). Performance evaluation prove and justify the efficiency and effectiveness of the candidate generation and the heuristic rules. Compared with the approach that generates the candidates directly from the database by a support range,
this solution is orders of magnitude faster and consumes much less memory.

2.2 Study of Supergraph Indices

There are two types of queries that can be performed on graph databases. The first one, the subgraph query returns results from a set of subgraphs of a database $D$, given a query $q$. The supergraph query on the other hand returns results from a supergraph of a graph database $D$, given a query $q$. In this section, we examine some supergraph indexing practices with the aim of applying some of their basic ideas towards subgraphs. Though the queries are essentially different, the way the search is performed in a graph is fundamentally similar and we might be able to gain some inspiration from this area.

Supergraph queries are difficult to process and the usual methods of filter and verification are inefficient. The filter and verification technique is a method used to process subgraphs quickly, since processing subgraphs are said to be an NP Complete problem. Typically supergraphs are processed by extracting subgraphs out of a graph database and processing them with the cIndex algorithm. The cIndex algorithm constructs a feature index based on historical data from query logs. Though cIndex is efficient, one big disadvantage of cIndex is that the query logs are variant resulting in outdated feature indices.

Zhang et al propose a new way to process supergraph queries without relying on the query logs for historical data. This approach first starts with compressing the graph database and then constructing feature indices and supplementing it with a new query processing technique. Addressing the storage of graph databases problem, a new method called the GPTree is proposed. The GPTree works by taking all graph databases as subgraphs and putting them into one graph database structure. All graphs in the database is first encoded by an encoding algorithm and entities called GVCodes are created. Subgraphs that are commonly occurring are then stored only once in the new GPTree structure. The GPTree algorithm is performed recursively till the final compressed GPTree is created. After compression is done, we move on to the concept of feature generation. Feature generation is done in two different methods. One selects all significant frequent subgraphs while the other selects a subset of significant frequent subgraphs. Frequent subgraphs are said to expose the intrinsic characteristics of a graph database. After all the features are extracted using the above techniques, it is filtered to save the cost of unnecessary subgraph isomorphism testing. This reduces the overall cost of query processing. After the feature set is also obtained, we construct a optimized GP Tree called the FGPTree that contains common GVCodes and ordered collection of feature sets.

This process is then supplemented with some query processing techniques. There are three methods used here for query processing. The first one is the subgraph isomorphism testing from one to many. Here a subgraph isomorphism is represented by a set of ordered pairs of matched vertices and these pairs are used to create a growing induced subgraph of the graph database $g$. The graph corresponding to the path from the root to any node in a GPTree is a subgraphs of graphs from the root to the descendant nodes. The second method online redundant features shedding is used to filter out features from sub-iso testing. This is used to shed feature set from subgraph iso testing algorithms. The third method, Integrated Query processing method integrates the above two methods and produces a result set after a GPTree test. The paper concludes by stating that proposed methods of creating the GPTree and the GPTreeTest creates efficiency by reducing the number of subgraph isomorphism testing to be performed in the filter and verification method.

3 Technical Details

The three algorithms we implemented as a part of the comparative study have already been described in the survey of related work. The three algorithms: Cindex, Gindex and FGIndex were implemented using the Neo4j Java API for the Neo4j graph database. The Neo4j also has an API for the Lucene indexing engine which made our job easier. The source code for the Neo4j Lucene index is available online and we made modifications to the IndexImplementation
class in the Lucene jar file to change the way Lucene built the index. Once this modification was done, the Lucene engine took care of the construction and maintenance of the index. So, only the algorithm for index construction was implemented 3 different ways. Once this was done, the 3 indexing methodologies were then pitted against each other and also compared with a state of No Index. We observed that the FG-Index performs best, and g-Index comes a close second. The C-Index, contrary to expectations performed worse than G-Index, but this may have to do with this particular dataset.

The next stage was the development of the SHIndex algorithm. This was a multi-stage process. We had to first figure out an appropriate mapping function to plot graphs onto a 2 dimensional space. Then, we had to ascertain the frequent graphs in the dataset based on some support and threshold values. These values can be fixed by a database administrator. The next step is to use an appropriate similarity hash function to classify the graphs into buckets based on their similarity to the frequent graphs. This is the final step in the index construction phase. When a query is sent to the system, the query graph is also mapped and hashed. It will now fall into one (or more) of the buckets. Only the graphs in the chosen bucket(s) need to undergo candidate verification. It is our belief that this will speed up the query despite taking a long time to construct the index.

Due to time constraints and the scope of the project, we were unable to go beyond implementing the 3 existing index structures. Graph databases represented a complete shift in the paradigm to us as we were completely used to the relational model. It took us some time to get used to the way the database worked and the way queries were written and this took time away from implementation. The proposed algorithm will be implemented in the future and we will then ascertain if this is a venture worthy of pursuit.

4 Evaluation

Several open source graph databases were considered and Neo4j was found to be a suitable prospective graph database for our research work. The Neo4j database is a highly robust, open source property graph database that is fully ACID compliant. It has the ability to scale up to a billion nodes and is very efficient for storing highly connected entities. The basic reason for choosing Neo4j is that it is open source and comes with a Java API to change its internal algorithms. Neo4j was successfully installed in our systems and we proceeded to test queries in it. The environment that Neo4j is installed on is a 6 GB RAM machine with a 500 GB of disk space, which runs on the Linux Ubuntu 12.04 operating system. Neo4j comes with a web interface which runs on the systems localhost apache server and oers a wide range of querying and data presentation tools. A dataset called the Campaign 2012 Election Data was downloaded and was used as an example data set for initial research. The data set comes with 525,769 nodes, 4614311 properties, 527,767 relationships and 6 relationship types.

We also had to learn Neo4js NoSQL querying language Cypher, which is simple and easy to learn., though initially, some time is required to unlearn the SQL approach. Different Cypher queries were tried out and computation times for each query is recorded and graphed. For evaluation, we developed 5 standard queries. Query1 looks for a particular node by property. Query 2 looks for top 100 results, ordered by a property. Query 3 is based on matching a relationship, which is essentially a join. Query 4 matches a set of properties across 2 relationships. Query 5 is an aggregate query based on 2 matches.
When we ran the queries, the runtimes we got were quite varied. So, each query was run 50 times for each index structure and then the runtimes were averaged out. The following figure graphs the performance of the 3 index algorithms with a No Index approach for comparison. As we can see, FG-Index gives the best performance, followed by the G-Index.

5 Future Work

This project is only halfway done. At present, we have studied the behavior of the 3 prominent index structures in use for graph databases. Future work will include ironing out the details of the SHIndex algorithm, implementing it and comparing its performance against the state of the art. We suspect that the index construction phase would perform worse than the state of the art due to the number of steps involved but the query performance will be better as it will greatly reduce the number of isomorphism tests. The details that remain to be worked out include the mapping function to reduce the complexity if the graph and the similarity hash function (or a similar function) used to sort the graphs into buckets based on their similarity. The algorithm should also prove to be correct. We believe that it might not be able to pull out 100% exact matches with a response time comparable to the state of the art, but it could perform as well, if not better, if the tolerance threshold is increased and we only look for a good enough match. We look forward to continuing the project and seeing it through.

6 Conclusion

There are a host of methods and suggestions in subgraph query processing, all of which try to reduce the query response time in one way or other. A study of these methods provides a keen insight into the critical aspects involved in query processing and the way the index is constructed to ease the burden of subgraph isomorphism. We learned that indexes used in graph databases can also grow very quickly and prove inefficient if certain factors used while building the index are not chosen with utmost care. All of these ideas try to work with generic graph databases, and while quite simple and efficient in the overall sense; might actually display a wide range of performance characteristics when compared on a case by case basis. An effort is needed to recognize which type of index might prove to be useful in a graph database for a particular application. An overall hybrid index structure that can be tweaked and adjusted by a DBA to perform with the maximal efficiency for that case might just be a step forward in the right direction. To this end, we propose the SHIndex which allows for enough flexibility to tweak certain factors, like the hash function, the mapping function and the number of buckets to yield the best results in subgraph query processing.

7 Work Distribution

The workload was pretty evenly distributed between the team members. The study of existing systems and choosing the algorithms for study was done by both team members. Srinath was responsible for choosing and studying the tools used and Sharanya was responsible for a detailed study of the algorithms for implementation and incorporating graphs in a LSH inspired model. Otherwise, all the workload was shared.

References

[1] Cheng, James; Ke, Yiping and Ng, Wilfred
Efficient query processing on graph databases

[2] Cheng, James; Ke, Yiping; Ng, Wilfred and Lu, An
Fg-index: towards verification-free


