Research Review of Algorithm Model in Graphic Database System

1st Tianrui Liu
The Grainger College of Engineering
University of Illinois at Urbana-Champaign
Champaign, USA
tl49@illinois.edu

2nd Tiannuo Yang
College of Liberal Arts and Sciences
University of Illinois at Urbana-Champaign
Champaign, USA
tiannuo2@illinois.edu

***Abstract*—Nowadays, with the establishment of social networks, graph data has played a critical role in everyday life. A graph database should be capable of dealing with the corresponding graph data. Every node in the graph is a data point, and the edges between nodes denote the relationship between data. The graph database is expected to reach the relationship between nodes rapidly and accurately, thus benefiting areas in need of vast computational resources, such as business and social networks. Querying and indexing the graph database is the most crucial part to reduce the computational resources, which naturally becomes the focus of this paper. In this review, we concluded the existing approaches and techniques of querying and indexing and summarized the pros and cons for each of them. Possible future research directions were also provided based on the analysis of existing ones.**

***Index Terms*—Graph-based database, Algorithm of database, Data management systems**

I. Introduction

Graph data has become a regular and crucial part of our daily life. We can represent data in various domains, including graphs, chemical compounds, social networks, biological compounds, and enzymes [1]. For example, taking social media sites, every account is stored as a node in the database with multiple attributes such as name, age, and gender. The relationship between nodes includes followers, liked posts, etc. The relationship of graph data is so complex that it would take too much time to comprehend and utilize any graph collection. To deal with this problem, if we can find a powerful query and index method, time can be saved during the data collection process [2].

Let $\mathcal{D} = \{g_1, g_2, \cdots, g_N\}$ be a graph database. The query in the graph database is described like this: given a query graph q, we want to retrieve all $g_i \in \mathcal{D}$ such that g_i is a supergraph of q. The graph structure is very complex, which requires us to match every part of the subgraph with the query in query processing, meaning that every attempt to get the subgraph of g_i is a NP-complete problem. Using a naive method to deal with this problem, we will find that the cost is unacceptably high: even a simple search would consume plenty of time. To reduce the computation complexity, people build up an effective index structure, the idea of which is based on trees and filtering sequences [3]. For instance, the closure-tree is an index structure for graph queries by using the tree, and FG-Index uses the filtering sequences to deal with the graph queries.

The structure of this review is demonstrated as follows: first, problems in graph queries are defined. Then several methods are introduced to improve the graph queries. Next, the performances of different methods are compared [4]. Finally, a conclusion on the above methods is made, and new research aspects are provided for future exploration.

II. Preliminaries

In this survey, we restrict our discussion to undirected labeled connected graphs even if some of the introduced methods can be used on the directed and unlabelled graphs.

A graph can be defined as (V, E, L, l) [5], where V represents the set of vertices in this graph and E is used to represent the edge in this graph, L is the set of labels and l should be the function which maps the vertex or edge to every label. And we define the size of graph g as $size(g) = |E(g)|$. The edge in this graph can be distinct represent by two nodes $e = (u, v)$ where $e \in E$ and $u, v \in V$.

Definition 1 (Subgraph Isomorphism): A subgraph isomorphism is an injective function for two graph $g = (V, E, L, l)$ and $g' = (V', E', L', l')$, $f : V(g) \rightarrow V(g')$, such that (1) $\forall u \in V, f(u) \in V'$ and $l(u) = l'(f(u))$, and (2) $\forall (u, v) \in E, (f(u), f(v)) \in E$ and $l(u, v) = l'(f(u), f(v))$.

Definition 2 (Graph Query Processing): Given a graph database $D = \{g_1, g_2, \cdots, g_N\}$ and a graph query q, it will return the query answer $D_q = \{g_i | q \subseteq g_i, g_i \in D\}$.

Many graph-related problems should be extremely difficult [6]. Moreover, when it comes to query processing, the majority of issues share these characteristics:

1) A single attribute index (vertex label or edge label) is insufficiently selective, but a random combination of numerous characteristics produces an enormous number of index entries [7].
2) The query is rather large, containing numerous edges.
3) In a large database, a sequential scan and test are costly.

The method mentioned in the next section can help us deal with these problems.

III. A Formal Representation of a Pattern for a General Method

There are several methods for graph indexing, and most of them are based on the tree structure or pruning method. In this part, we will roughly discuss the methods for indexing the graph, which will significantly reduce the time of graph indexing.

A. Closure-Tree

The Closure-Tree [8] is a tree that satisfies the following properties:

1) Each node is a graph closure of its children. The inner nodes are the closure node, and the leaf nodes are the database graph.
2) Each node has at least m children unless it is root or leaf.
3) Each node has at most M children, $\frac{(M+1)}{2} \geq m$.

The subgraph query should be handled in two steps when employing the closure-tree. The C-tree is traversed in the first phase, and nodes are pruned based on pseudo subgraph isomorphism. After this stage, a candidate's answer can be returned. Subsequently, each possible response will be verified for exact subgraph isomorphism, and then their return, the results will be in the second step.

They prune by using a histogram-based strategy besides the subgraph isomorphism method. A graph's histogram is a vector that counts the number of vertices and edges with each property. Let FQ and FG be the histograms of Q and G, respectively, given a query Q and a graph G. If Q and G are subgraphs isomorphic, then $FQ[i] \leq FG[i]$ for all i. They will then use this property to put this node to the test. Subgraph isomorphism tests are slower than this method. The sketch line of this method is shown in Alg.1, where $visited$ should be obtaining the candidate answer via subgraph isomorphism and histogram trimming.

Here comes the cost of the subgraph queries' performance. Subgraph isomorphism is used to visit and test $\gamma|D|$ nodes and database graphs during the search phase. Then we will have database graphs with $|CS|$ (the number of candidate answers) that are tested using exact subgraph isomorphism. The total time should then be

$$T = \gamma \cdot |D| \cdot T_{visited} + |CS| \cdot T_{isom}$$

Let's assume that the closure-tree has a h-level. Let $x(i)$ be the number of children that survive the histogram test in the ith level and $y(i)$ be the number of children who survived after the subgraph isomorphism test in the ith level. $R(i)$ be the expected number of nodes and database graphs visited below a node at level i.

$$R(0) = \sum_{i=0}^{h-1} x(i) \prod_{j=0}^{i-1} y(j) + \prod_{i=0}^{h-1} y(i)$$

$$\gamma = \frac{1 + R(0)}{|D|}$$

Algorithm 1 SubgraphQuery(query, ctree)

$candidates \leftarrow \{\}, ANS \leftarrow \{\}$
$Visit(query, ctree.root, CS)$
for $G \in CS$ **do**
  if SubIsomorphic(query,G) **then** $ANS = ANS \cup \{G\}$
  end if
end for
return ANS

B. K-Adjacent Tree

Definition 3 (Adjacent Tree): The adjacent tree of a vertex $v(AT(v))$ in G is a breadth-first search tree of vertex v, the children of each node of $AT(v)$ are sorted by their labels in the graph.

Definition 4 (κ-Adjacent Tree [9]): The κ-Adjacent Tree of a vertex v $(\kappa - AT(v))$ in graph G is the top κ-level subtree of $AT(v)$.

After getting the definition of these, they firstly generate all the $\kappa - AT$s of each graph in the graph set and store them in a table. For a query Q, they generate its $\kappa - AT$s. Then they justify whether it holds.

$$|\kappa - ATS(Q) \cap \kappa - ATS(G)| \geq$$
$$|V(Q)| - \Delta_G(Q, G) \cdot 2(\delta(Q) - 1)^{\kappa - 1}$$

If this inequality holds, then G belongs to the candidate set of the query.

However, if κ increases, the space we need will increase exponentially, and even if we have enough space to store $\kappa - AT$s, the isomorphism test is also slow on a large tree.

To compact the index size, for each $\kappa - AT$, they use a unique ID to represent the κ-AT. Firstly, they give 1-AT a unique ID and get the frequency of 1-AT. Then they use the 1-AT to generate the 2-AT. Recursively, they can generate all of the κ-AT.

Despite the fact that searching on the κ-AT is significantly faster than using the ΔG matching approach, this method is still a sequence search method. In this graph, we need to access all of κ-AT, and the computation cost is very high. Then they figure out that they can reverse the κ-AT index and use their κ-AT as keywords in each graph in the graph set. This approach eliminates the need to search for sequences over the entire graph. The primary step of this approach is depicted in Alg.2.

Algorithm 2 Filter_inv(query,G, ϵ, κ)

generate inv_Q and inv_G
for $i = 0$ to $len(inv_Q)$ **do**
  pick up the inverted list of $inv_Q[2\kappa - 1]$ to match inv_G
end for
Check whether $|\kappa - ATS(Q) \cap \kappa - ATS(G)| \geq |V(Q)| - \Delta_G(Q, G) \cdot 2(\delta(Q) - 1)^{\kappa - 1}$ satisfied. If satisfied, add this list to result.
return result

From the paper, we naturally conclude that κ-AT has a better performance than FG-Index and closure-tree. Moreover, the

bigger the size of the database, the better the performance of κ-AT is.

C. R-Tree

Like the B-Tree, the R-tree is also a balanced search tree in which the node contains its children's range and points to its children. R-tree, proposed by Guttman in 1984, is the most popular dynamic spatial index structure, widely used in prototype research and commercial applications. There are many different improvements in R-trees for different spatial operations on this basis.

R-tree is a completely dynamic spatial index data structure in which insert, delete, and query can be performed simultaneously without periodic index reorganization. It is composed of intermediate nodes and leaf nodes. Leaf nodes store the minimum boundary rectangle (MBR) [10] of the actual spatial object, not the actual spatial object.

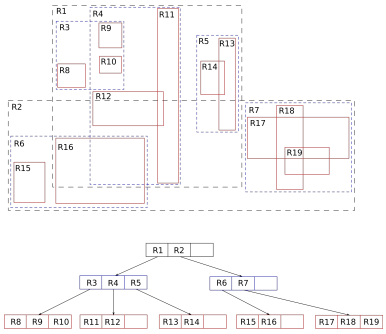

Fig. 1. An example of a simple R-tree on a two-dimensional rectangle

The R-tree allows sibling nodes to overlap each other. Therefore, the R-tree cannot guarantee the unique search path for the exact matching query. To avoid the multi-path query problem caused by the overlap of sibling nodes in the R-tree, in 1987, Sellis [11] designed the R+tree that adopts object segmentation technology to improve its retrieval performance. The overlapping of sibling nodes is avoided. It is required that objects spanning subspaces must be divided into two or more MBRs. R+tree solves the problem of multi-path search in the R-tree query, but it also brings other problems. For example, redundant storage increases the tree's height, reduces the performance of domain queries, and may cause deadlock in adverse circumstances.

The basic form of the R-tree algorithm is (I, tuple identifier). Where the ancestor identifier points to the corresponding data. I is an n-dimensional rectangle of a spatial object contained in a bounding box, expressed as:

$$I = (I_0, I_1, ...I_{n-1})$$

n refers to the number of dimensions, and I_i is a closed bounded interval [a, b], used to describe the range of spatial objects on dimension I. It can have one or two infinite boundaries, and the surface object is infinite. The non-leaf nodes of the R-tree contain entries in the form of (I, child pointer), where child pointer is the address of a low-level node in the R-tree, and the rectangles are covered in all low-level node entries. In short, each node contains multiple child nodes or data (when the node is a leaf), and the node contains a multi-dimensional rectangle. I represent the smallest bounding rectangle of all child nodes or data. We can assume solving the problem of an appropriate insertion path. Only the area parameter is considered in the description of the R-tree. Later the area, edge, and coverage will be considered together. The coverage of one item is defined as follows:

$$overlap(E_k) = \sum_{i=1, i\neq k}^{p} area(E_k Rectangle \bigcap E_i Rectangle)$$
$$1 \leq k \leq p$$

D. PIS

PIS [12] builds a fragment-based index on the graphics database. Graph database members are decomposed into overlapping fragments, in which fragments with the same topology are indexed by the R-tree data structure.

Definition 5 (PIS substructure): A fragment-based index database is constructed based on a graph. Each query graph is divided into highly selective fragments. Using the index can effectively identify the collection and validate each candidate to find all qualified answers.

Two advantages of the method are highlighted over other methods of violence retrieval as follows:

1) Candidate verification is performed only through the retrieval structure. In this way, subgraph by subgraph isomorphism calculation in the database is avoided, and the candidate atlas has a much smaller size.
2) The candidate set itself is built with the help of selective fragmentation and a lower distance limit.

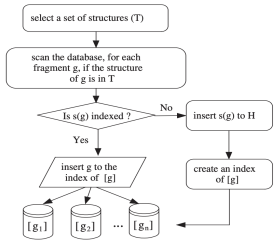

Fig. 2. Detailed process of PIS index

Firstly, the basic logical framework is to scan the entire database and check whether the target graph has a superposition with a distance less than the threshold. The graphs that do not contain query structures can be eliminated. Then Enumerate the overlay candidate graphs of query graphs in an extensive collection. This structure is based on two main indexes, fragment and partition. The fragment index is used as the index feature according to the proposed standard, and a new index is constructed, in which the range query marks the graph and skeleton. The partition search divides a given

query graph into a group of non-overlapping data fragments of the index, finds its equivalent class in the index, and submits the range query to find the database that conforms to the superposition distance all fragment databases.

$$\sum_{i=1}^{n} d(g_i, G) = \sum_{i=1}^{n} mind(g_i, g')$$

The algorithm of this method is based on NP-hard and index-based partitioning. Set the instance (I, q) of index partition as index structure I and query graph Q. In a given instance (G, w), an index-based partition instance (I, q) is constructed, I self-loops are added to each vertex on this ring, and each edge v_i and v_j is replaced. In this way, it can obtain the query graph Q, and each ring and all its adjacent edges now form the unique topology in Q. It must be a set of subgraphs described. Namely, each subgraph is a ring, and its vertices have the same number of self cycles, and each vertex has an adjacent edge.

Algorithm 3 PIS structure in Graph search

```
for each fragment g and [g] is indexed do
    F = F union g, remove fragments g from F
end for
for each fragment g in F do
    calculate g's canonical label, locate the index structure
I pointed, submit a range query
    for each pair g, G do
        if G in T then
            d(g,G) = min(d(g,G), d(g, g')
        else
            d(g,G) = d(g, g'), T = T union G
        end if
    end for
end for
construct an overlapping relation graph for Q
select a partition P according to Greedy()
```

E. Tree-Pi

There is a new structure TreePi [13] in tree-based mining, which is a method of using frequent subtrees as graph structure index units. Frequent subtrees are used as index structures because the tree is a more compact form to store structural information in graphical databases, especially the structure and data set in the graph database. This makes indexing and searching easier. More importantly, the symmetry of trees makes the retrieval structure more complete.

1) Tree data structures are more complex patterns than paths and trees as they can preserve an almost equivalent amount of structural information as arbitrary subgraph patterns.
2) The frequent subtree mining process is relatively easier than the general frequent subgraph mining process.

The main structure rules of TreePi are through the index method based on frequent trees. It selects a set of frequent trees in the graph database as the index mode. In query processing, for query graph Q, the frequent subtrees in Q are listed respectively. Furthermore, we determine the graphics that contain these subtrees in the database. The standard form of any tree can be calculated in polynomial time, and it can rapidly perform the first filtering operation. Moreover, by applying the center distance constraint, TreePi reduces the graph database and query graph, which dramatically reduces the search space. In addition, the location information tree stored in the feature part is used in the verification phase.

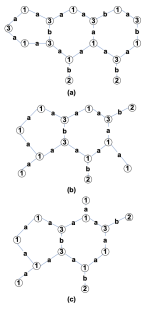

Fig. 3. TreePi Processing structure

TreePi can be divided into two main steps for graphic query processing: construction and query processing of database preprocessing. In the first step, the subtree in the frequent graphics database is enumerated and selected as the feature tree. Next, query any subtree existing in the feature tree. Query processing is divided into three steps: the query graph is divided into a group of function trees and then filtered and trimmed respectively to project the graph database. In the end, it is applied to the center distance constraint.

For TreePi, because the number of different trees will increase exponentially over time, unique methods are used in the pretreatment process σ function in order to ensure the integrity of the index and the worst case.

$$\sigma(s) = \begin{cases} 1 & \text{if } s \leq \alpha \\ 1 + \beta s & \text{if } \alpha < s \leq \eta \\ +\infty & \text{if } s > \eta \end{cases}$$

For tree filtering and center processing, the concepts of distance chart and center distance constraint based on the center of query subtree are adopted. It uses the position information of the number and adopts any substructure mode without the unique center. On the candidate graph, the algorithm can filter out most of the unqualified candidate graphs.

F. GD-Index

In this method, Directed Acyclic Graph is utilized to represent graph decomposition which can get the graph G's structure [14].

1) Each node is a subgraph P of G.
2) For any two nodes P and P', there is a directed link from P to P' if $P \subset P'$ and there do not exists any other graph P'' where $P \subset P'' \subset P'$

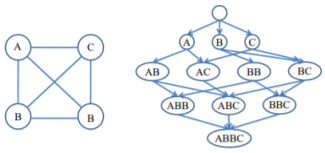

Fig. 4. Decomposition of a complete graph

Since the amount of the graph data is so huge, some fast search methods are needed to search the query. In this paper, a hash table is used based on the canonical code. Every different entry G will be made for a different canonical code $\phi(G)$. Then canonical code is used as a key to store the graph so that the search function can quickly locate the node in DAG if this subgraph is isomorphic to the query graph.

Algorithm 4 GD-index(G)

$ans = \emptyset, visited = \emptyset$
$v = \mathcal{H}(\phi(G))$
if v exists **then**
  $Visit(v, ans.visited)$
end if
return ans

By using this method, there is no need to compare two graphs sequentially. What is necessary to do is to compare the canonical code, which reduces a good chunk of time to compare on the large graph. However, this method also has some disadvantages in that it performs well on the large graph, but it may consume more time when the graph database contains many small graphs. The Alg.4 shows the primary step of the GD-index.

From the experiment, it can be concluded that the GD-Index has a better performance than the closure-tree, and when the size of the query gets larger, the average query time decreases.

IV. Conclusion

Query processing often includes a cost-based optimization step in which query optimizers use cost models to select the best query plan from a set of options. The cardinality estimate of the intermediate and final query results is a critical issue in the cost model. Most of existing indexing methods still have significant disadvantages. A framework which can estimate the selectivity of more complicated graph patterns accurately and speed up the processing of various graph queries is thus needed.

Graph database enables individuals and organizations to better understand the large amount of collected data, thus contributing to the development of business and society. It helps determine the relationships of data, which are difficult and even impossible to be clarified using other techniques, and beneficial for data professionals at all levels to release the potential of their data relationships, instead of just a single data point. The imagination of database users becomes the only limitation on how to use these relationships. Driven by 5G, the Internet of things, artificial intelligence, and other digital technology innovations, the complexity of the association among data also increases drastically. The traditional relational database, therefore, has low operation efficiency when dealing with complex associated data and is no longer able to help people further explore the value behind the massive relational data. That is why new technology of graph database has come to life. However, it is not perfect since it tends to be optimized for speed and structure, meaning that the data would be represented as a table without too many missing values, which is often not a good choice. In conclusion, a graph database is essentially a solution to solidify and query graph data structure. However, in real practice, people still need to be cautious when using it and remember to be flexible according to the actual situation.

References

[1] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Feature-based similarity search in graph structures," *ACM Trans. Database Syst.*, vol. 31, no. 4, p. 1418–1453, dec 2006. [Online]. Available: https://doi.org/10.1145/1189769.1189777

[2] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Min. Knowl. Discov.*, vol. 15, no. 1, p. 55–86, aug 2007. [Online]. Available: https://doi.org/10.1007/s10618-006-0059-1

[3] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "Igraph: A framework for comparisons of disk-based graph indexing techniques," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 449–459, sep 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920901

[4] R. kumar Kaliyar, "Graph databases: A survey," in *International Conference on Computing, Communication Automation*. Greater Noida, India: IEEE, May 2015, pp. 785–790.

[5] A. Bhattacharyya and D. Chakravarty, "(graph database: A survey)," in *2020 International Conference on Computer, Electrical Communication Engineering (ICCECE)*. Kolkata, India: IEEE, 2020, pp. 1–8.

[6] X. Wang, X. Ding, A. K. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *2012 IEEE 28th International Conference on Data Engineering*. Arlington, VA, USA: IEEE, April 2012, pp. 210–221.

[7] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 335–346. [Online]. Available: https://doi.org/10.1145/1007568.1007607

[8] H. He and A. Singh, "Closure-tree: An index structure for graph queries," in *22nd International Conference on Data Engineering (ICDE'06)*. Atlanta, GA, USA: IEEE, April 2006, pp. 38–38.

[9] G. Wang, B. Wang, X. Yang, and G. Yu, "Efficiently indexing large sparse graphs for similarity search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 3, pp. 440–451, March 2012.

[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, p. 47–57, jun 1984. [Online]. Available: https://doi.org/10.1145/971697.602266

[11] Y. Theodoridis and T. Sellis, "Optimization issues in r-tree construction," in *IGIS '94: Geographic Information Systems*, J. Nievergelt, T. Roos, H.-J. Schek, and P. Widmayer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 270–273.

[12] X. Yan, F. Zhu, J. Han, and P. S. Yu, "Searching substructures with superimposed distance," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. USA: IEEE Computer Society, 2006, p. 88. [Online]. Available: https://doi.org/10.1109/ICDE.2006.136

[13] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," in *2007 IEEE 23rd International Conference on Data Engineering*. Istanbul, Turkey: IEEE, 2007, pp. 966–975.

[14] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *2007 IEEE 23rd International Conference on Data Engineering*. Istanbul, Turkey: IEEE, 2007, pp. 976–985.