A Unified Narrative for Query Processing in Graph Databases

Yue Pang
Peking University
Beijing, China
michelle.py@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

M. Tamer Özsu
University of Waterloo
Waterloo, Canada
tamer.ozsu@uwaterloo.ca

***Abstract*—With the advent of graph data, graph databases have garnered significant research interest and efforts in recent years, especially with respect to graph query processing. There have been a vast suite of methods for efficient graph query processing, especially for the core graph query constructs, regular path queries (RPQs) and subgraph matching queries (SMQs). In the meantime, there is an observable divide among these methods as well as confusion between them and their relational counterparts. We thus propose this tutorial to provide a unified narrative for graph query processing, so as to bridge the gap between existent lines of work and offer a comprehensive view of the query processing workflow in graph databases.**

***Index Terms*—graph query, graph database, query processing, query planning**

I. Introduction

Graphs are a natural abstraction for modeling interconnected data, which represent entities as vertices and the relationships between them as edges. A graph database system uses graphs as its data model, which brings about opportunities of exploring interconnected data with high efficiency. Thus, significant effort has been spent designing and optimizing graph databases in the last decades, especially query processing.

Graph queries typically exhibit greater diversity and complexity than their relational counterparts, incorporating regular path queries (RPQs), which pose no limit on the path length, and subgraph matching queries (SMQs), which translate to a large number of joins. These unique features have prompted the development of a vast suite of methods for efficient graph query processing. Though there have been excellent tutorials [1]–[3] and surveys [4], [5] dedicated at least partly to graph query processing in recent years, we are still motivated to come up with a unified narrative that 1) bridges the gap between the two core graph query constructs, RPQs and SMQs; 2) crystallizes the relationship between graph and relational query processing, including the overlapping and non-overlapping aspects; and 3) offers an easily comprehensible bird's eye view of the query processing workflow in graph databases. To our knowledge, no previous survey or tutorial has achieved all three goals in unison.

In this tutorial, we will start by introducing the preliminary knowledge regarding graph databases (Sec. III) and defining the core query constructs, RPQs and SMQs (Sec. IV). Then, we will put forward the unified set of operators for these constructs and discuss in detail the relationship between them and their relational counterparts (Sec. V). On this basis, we will discuss the classic and state-of-the-art methods used in the query processing pipeline, including query evaluation (Sec. VI) and query planning (Sec. VII). We will conclude with the open challenges of existing techniques and possible ways to address them in the future (Sec. VIII).

II. Organizational information

- **Duration:** This 90-minute tutorial covers: (1) Graph database preliminaries (Sec. III): 10 minutes; (2) Core graph query constructs (Sec. IV): 10 minutes; (3) The unified set of graph query operators (Sec. V): 20 minutes; (4) Graph query evaluation (Sec. VI): 20 minutes; (5) Graph query planning (Sec. VII): 20 minutes; (6) Open Challenges (Sec. VIII): 10 minutes.
- **Target Audience and Assumed Background:** This tutorial is intended for database researchers and practitioners, especially those with an interest in graphs and query processing. We do not require any prior knowledge and will provide any necessary background.
- This tutorial will not be hands-on.

III. Preliminaries

Graph data & query models. The two major types of graph data models adopted by graph DBMS are property graphs and RDF (Resource Description Framework) graphs. SPARQL is the standard query language for RDF graphs, while there are numerous query languages over property graphs, including but not limited to GQL and Cypher. The key difference between the property graph and the RDF graph model is that the former allows attaching key-value pairs denoting property names and values to vertices and edges. Such a difference leads to different graph database storage layouts, which affects the cost of query operators (Sec. VII). The queries on these data models can also be different, since queries on property graphs can involve property values. In this tutorial, we primarily focus on the common core constructs of property graph and RDF graph queries (Sec. IV).

Graph storage. Conceptually, the edges in a graph are stored as a two-column table, each row of which denotes the source and target vertices of an edge. We omit the storage of vertex and edge properties, since we do not focus on query constructs concerning properties in this tutorial. Physically, there are two distinct types of storage schemes for materializing the aforementioned structures. The hybrid storage scheme, used by systems such as Virtuoso [6], leverages relational tables as the back-end storage engine, storing the edges as table rows. Contrarily, the native storage scheme, employed by systems such as Neo4j [7], assigns dense integer IDs to vertices and stores each vertex's neighbors as adjacency lists, ensuring that a vertex's neighbors can be accessed in constant time. Native storage schemes are either implemented from scratch or based on key-value stores. Whether the storage is native or hybrid has a prominent effect on query evaluation (Sec. VI) and planning (Sec. VII).

IV. Core Graph Query Constructs

Regardless of the data model, regular path (i.e., navigation) and subgraph matching (i.e., conjunctive graph) queries are the core query constructs in graph databases, defined as follows. (We refer to graph queries consisting of only these two types of constructs as graph queries henceforth without ambiguity.)

Regular path query (RPQ). Given a regular expression using the set of graph edge labels as the alphabet, an RPQ returns either all the graph vertex pairs between which at least one path has an edge label sequence that conforms to the regular expression, or the conforming paths. The distinctive characteristic of RPQs is that they allow Kleene closures ($*$ or $+$ in the regular expressions), which can match paths of unlimited length.

Subgraph matching query (SMQ). Given a subgraph pattern, a subgraph where the vertices and edges can either be bound to concrete vertices and edge labels in the graph or be variables, an SMQ tries to find matches of the subgraph pattern in the graph by subgraph isomorphism and returns the variables' bindings if there are valid matches.

Distinction of graph queries from relational queries. Lossless translation can be conducted from these graph query constructs to relational queries, which means that they have the same basic algebraic operators. However, typical graph queries are distinct from their relational counterparts in terms of the algebraic operators' distributions. Specifically, RPQs with Kleene closures translate into relational queries with recursion, and typical SMQs translate into conjunctive relational queries with many joins. While these features are relatively rare in relational queries, they are prevalent in graph queries [8], which necessitate specialized query processing techniques discussed in the rest of the tutorial.

V. Graph Query Operators: A Unified Narrative

As discussed in Sec. IV, graph queries and relational queries share a common set of algebraic operators. These algebraic operators dictate the semantics of a query. However, these algebraic operators can be further split or reordered according to certain rules, so that multiple semantically equivalent query plans are generated. This process is called query plan enumeration, the first step in query planning (Sec. VII).

Though RPQs and SMQs require seemingly different operators, we present two operators at a higher level of abstraction

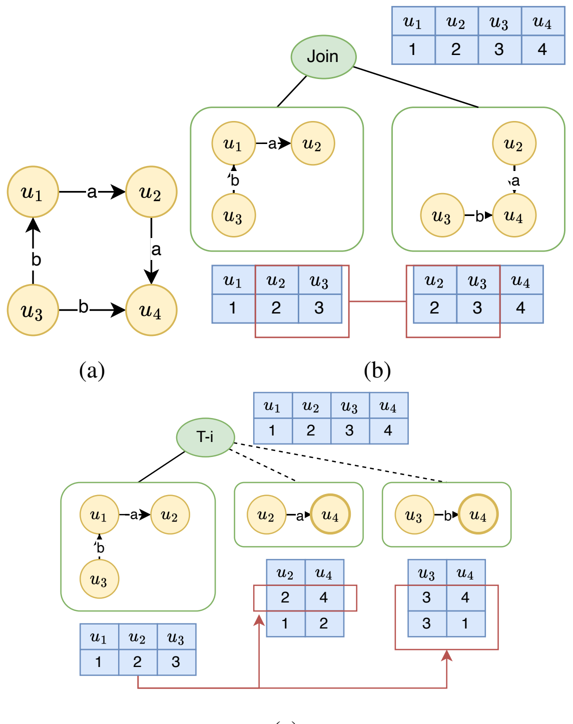

Fig. 1: Join and traverse-intersect (T-i) operators on an example SMQ.

that can implement both types of queries, which are called *join* and *traverse-intersect*, respectively. The reason why we would like to introduce such a unification is twofold. Firstly, it allows for a holistic understanding of the core graph query constructs' evaluation. Secondly, it could open doors to new opportunities for graph query optimization (Sec. VIII).

Join. In relational contexts, a join happens between two tables and produces a new table. In the graph query context, a join happens between the intermediate results of two query subgraphs (or paths, excluding single edges) that share at least one common vertex, and produces the results of the merged subgraph (or path). Such a join is always binary, i.e., it always has exactly two operands.

Traverse-intersect (T-i). A T-i operator extends a query subgraph (or path) by a vertex by traversing the edges extending from the previous step's query subgraph (or path) and potentially pointing to the newly added query vertex. Only if there is exactly one traversed edge or the edges intersect at a vertex is the target data vertex bound to the added query vertex.

Example. Fig. 1 shows an example SMQ whose last evaluation step is implemented with a join or T-i operator, respectively. The join operator merges two query subgraphs with the common vertices u_2 and u_3, while the T-i operator extends the previous step's subgraph with the vertex u_4 by traversing all the edges pointing to it. The intermediate result tables associated with each query subgraph are processed accordingly to produce the operator's results.

Discussion: Relation with binary joins (BJs) and multi-way joins (MWJs). The fundamental difference between BJs and MWJs is in their arity, while the join and the T-i operators fundamentally differ in that T-i uses sideways information passing [9], while join treats its operands independently. Therefore, the join and T-i operators are not one-to-one mapped to BJs and MWJs. Instead, a BJ can be mapped to either a join or a T-i operator depending on its operands, while an MWJ is mapped to a T-i operator. Such a distinction is particularly important on native graph storage, which enables pointer-based joins for sideways information passing (Sec. VII). Sideways information passing is not employed by the join operator due to the difficulty in utilizing the passed intermediate results beyond scanning for single edges.

VI. Graph Query Evaluation

The two basic operators introduced in Sec. V, join and T-i, are sufficient for evaluating the core graph query constructs, RPQs and SMQs. In this section, we will introduce the classic and state-of-the-art graph query evaluation methods and how they can be mapped to the basic operators.

A. SMQs

There are mainly two lines of work on optimizing SMQ evaluation: one takes the perspective of join algorithms, while the other follows the filtering-ordering-enumeration framework [10].

Join algorithms. Since an SMQ can be viewed as joining the respective edge tables on the intersecting query vertices, it can be evaluated with join algorithms, including BJs and MWJs. MWJs, when used across the entire query, achieves worst-case-optimality, which guarantees that the intermediate result sizes do not exceed the final result size in the worst case (i.e., the AGM bound [11]). However, BJs can still be more efficient in practice when the query subgraph pattern is acyclic. These two types of join can be used in conjunction to evaluate a SMQ, which requires novel planning techniques [12]. As discussed in Sec. V, both BJs and MWJs can be implemented with the aforementioned basic operators.

Filtering-ordering-enumeration framework. Commonly seen in works that treat SMQ evaluation as an algorithmic problem as opposed to a query optimization problem, this framework generates candidate result sets for each query vertex, selects an order for them that expectedly maximizes efficiency, and finally enumerates the valid results according to the order. In fact, the third stage in the framework, i.e., enumeration, is equivalent to evaluating an SMQ using MWJs only.

B. RPQs

The two main RPQ evaluation methods in the literature are automaton-based and extended-relational-algebra-based (e-RA-based), respectively.

Automaton-based evaluation. These methods convert the given regular expression into a finite automaton, compute the product automaton between it and the data graph, search for paths in the product automaton, and map the paths back to the data graph to produce the results.

e-RA-based evaluation. These methods extend relational algebra with operators for computing Kleene closures. There are multiple ways of extension, including α-RA [13] and μ-RA [14]. The given regular expression is compiled into an e-RA tree, and the graph is treated as a set of edge tables. All the RA operators except for the extended operator are executed according to their standard definitions, while the extended operator performs a fix-point operation, which self-joins the intermediate results until no new results are produced.

Hybrid method. Since automaton-based and e-RA-based evaluation methods are not entirely overlapping, [13] proposes a hybrid evaluation method based on extended automata that allows bidirectional transitions and using views (i.e., cached intermediate results) as transition labels.

Implementing RPQs with the basic operators. RPQs without Kleene closures are a special case of SMQs where the query subgraph is a path, and thus can be implemented with the basic operators. Kleene closures make up the essential difference between RPQs and SMQs. They are evaluated using automata by traversing the same edge label sequence repeatedly until the path cannot be further extended, or using e-RA with the extended fix-point operator, which self-joins the intermediate results repeatedly until no new results are produced. The first operation can be implemented with T-i with a self-loop that symbolizes recursion, while the second can be implemented with a fix-point operator with a join or T-i operator as its only child operator.

C. Physical Implementation Issues

Apart from sideways information passing as discussed in Sec. V, there are other important physical implementation issues that are not covered in the operators' definition but can impact query efficiency.

Depth-first- and breadth-first-style (DFS-style and BFS-style) implementations of T-i. A DFS-style implementation operates on each row of the intermediate results, carrying it over to the subsequent operator after it is successfully extended with the current query vertex; while a BFS-style implementation operates on the entire intermediate result table, only proceeding to the next operator after extending all the rows. DFS-style implementations are more space-efficient, while BFS-style implementations can take the opportunity of observing all the intermediate results to detect and reduce redundant computation.

Pointer- and value-based joins. Native graph storage, which assigns dense integer IDs to vertices, enables pointer-based joins as opposed to value-based joins in relational DBMSs. Pointer-based joins use the integer ID of each vertex as a pointer to access its attributes or neighbors in constant time. Value-based joins compare the values in the specified table columns to sift out matching rows. Pointer-based joins are typically more efficient, unless there are indexes on the join conditions built for value-based joins. The proposed join operator can only use value-based joins, since the intermediate results are stored as tables. Whether the traverse step in T-i uses pointer- or value-based joins depends on the graph storage. Native graph storage may favor using T-i as opposed to the join operator, since it can leverage pointer-based joins.

VII. Graph Query Planning

As mentioned in Sec. V, though the semantics of a query is fixed, its operators can be split or reordered in a semantics-preserving manner. Any organization of a query is referred to as one of its query plans. Different plans of the same query can lead to drastically different execution efficiency. Therefore, it is crucial to select an efficient query plan for execution, the process of which is called query planning.

Graph query planning differs from its relational counterpart due to the difference in their query workloads' characteristics (Sec. IV), i.e., graph queries typically have recursion and a large number of joins. Such differences pose challenges to graph query planning. We will introduce the concept of plan spaces that is integral to planning, the main query planning methods, how they address these challenges, and what challenges remain unaddressed.

Plan space. The plan space of a given query is the space consisting of all its semantically equivalent query plans. The operators allows in query plans as well as the constraint on their order define the shape, size, and contents of the plan space. For example, referring back to the proposed set of basic graph query operators, the plan space will be different depending on which one or both of them are allowed. On one hand, a larger plan space that subsumes a smaller one means that the globally optimal plan is at least as good. On the other hand, a larger plan space also means more time needs to be spent exploring it for a potentially optimal plan, which increases the planning overhead. Therefore, whether more operators should be allowed and more constraints should be removed is a delicate tradeoff. Currently, many distinct plan spaces exist for both RPQs and SMQs. It is preferable to devise a unification so as to simultaneously leverage various existing query evaluation techniques, such as using the set of operators proposed in this tutorial.

Cost-based planning. Cost-based planning methods first enumerate semantically equivalent query plans, then use the cost estimator to assess the cost of these plans, and finally choose the plan with the lowest cost estimate for evaluation. Cost estimation necessitates a cost model that accurately assesses each operator's cost according to its operands' cardinalities as well as the operands' cardinality estimates. Cost models for SMQs are well-established, but not so much for RPQs. Specifically, how to accurately model the cost of evaluating a Kleene closure is still an open problem. As for cardinality estimation, the cardinality of SMQs is arguably more challenging to estimate accurately than relational join queries due to the typically larger number of joins. To address the challenge, existing works diverge from their relational counterparts by building synopses based on subgraph patterns, devising sampling strategies that reduce the sampling space [15], using graph neural networks (GNN) to capture the graph structure [16], etc. Estimating the cardinality of RPQs with Kleene closures is also open, since the existing works all manually enforce a maximum path length.

Heuristics-based planning. Unlike cost-based methods, heuristics-based planning methods do not estimate a query plan's execution cost, but rely on other, typically less computation-intensive metrics such as the number of edges with a certain label or the portion of the search space that is guaranteed to be reduced [17]. They are thus typically used to control the planning overhead when the number of joins is particularly large. It is possible to combine cost- and heuristics-based planning, e.g., planning a subquery's execution with a cost-based method and deciding the remaining evaluation steps using heuristics.

VIII. Open Challenges

At the end of the tutorial, we will sum up the open challenges in graph query processing as follows and shed some light on potential solutions:

- **Unified query plan representation.** We have proposed a unified set of operators that bridge the gap between RPQ and SMQ evaluation. The apparent next step is to derive a query plan representation and the corresponding planning methods so as to synthesize the existing lines of work in graph query processing and expose new optimization opportunities. It is yet unclear how such a unified representation should be defined.
- **Query planning for RPQs.** As discussed in Sec. VII, the cost and cardinality estimation for RPQs remain open problems due to the recursive nature of Kleene closures. Providing a solution that eliminates the need to manually enforce a maximum path length will benefit both the graph theory and database systems community.
- **Processing conjunctive RPQs (CRPQs).** This tutorial and most of the past literature focus on RPQs and SMQs in isolation, while a large portion of real graph queries incorporate both constructs, i.e., CRPQs. It is imperative to consider synthesizing RPQ and SMQ processing techniques to devise a principled framework for optimizing CRPQ processing.

IX. Author information

Yue Pang is a Ph.D. student at Peking University. Her research focuses on efficient graph algorithms, especially path algorithms and query optimization in graph DBMSs.

Lei Zou is a Professor at Peking University. His work focuses on graph DBMSs, knowledge graphs, and hardware-accelerated graph computing systems and their applications in big data analysis.

M. Tamer Özsu is a University Professor at the University of Waterloo. His work focuses on the data engineering aspects of data science, including the application of database technology to non-traditional data types such as graphs, and distributed & parallel data management.

REFERENCES

[1] A. Mhedhbi and S. Salihoğlu, "Modern techniques for querying graph-structured relations: Foundations, system implementations, and open challenges," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3762–3765, Aug. 2022.

[2] M. Arenas, C. Gutierrez, and J. F. Sequeda, "Querying in the Age of Graph Databases and Knowledge Graphs," in *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event China: ACM, Jun. 2021, pp. 2821–2828.

[3] G. H. Fletcher, H. Voigt, and N. Yakovets, "Declarative graph querying in practice and theory," in *EDBT/ICDT 2017 Joint Conference 20th International Conference on Extending Database Technology*, 2017, pp. 598–601.

[4] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.

[5] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB journal*, vol. 29, pp. 595–618, 2020.

[6] O. Erling and I. Mikhailov, "RDF Support in the Virtuoso DBMS," in *Networked Knowledge - Networked Media*, J. Kacprzyk, T. Pellegrini, S. Auer, K. Tochtermann, and S. Schaffert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 221, pp. 7–24.

[7] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, no. 36, 2013, pp. 141–147.

[8] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large SPARQL query logs," *The VLDB Journal*, vol. 29, no. 2-3, pp. 655–679, May 2020.

[9] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets, *Querying Graphs*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, Oct. 2018, vol. 10, no. 3. [Online]. Available: https://inria.hal.science/hal-01974379

[10] Z. Zhang, Y. Lu, W. Zheng, and X. Lin, "A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction," *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–29, Mar. 2024.

[11] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013. [Online]. Available: https://doi.org/10.1137/110859440

[12] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1692–1704, jul 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342643

[13] N. Yakovets, P. Godfrey, and J. Gryz, "Query Planning for Evaluating SPARQL Property Paths," in *Proceedings of the 2016 International Conference on Management of Data*. San Francisco California USA: ACM, Jun. 2016, pp. 1875–1889.

[14] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaida, "On the Optimization of Recursive Relational Queries: Application to Graph Queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, Jun. 2020, pp. 681–697.

[15] K. Kim, H. Kim, G. Fletcher, and W.-S. Han, "Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 964–976. [Online]. Available: https://doi.org/10.1145/3448016.3457246

[16] T. Schwabe and M. Acosta, "Cardinality estimation over knowledge graphs with embeddings and graph neural networks," *Proc. ACM Manag. Data*, vol. 2, no. 1, mar 2024. [Online]. Available: https://doi.org/10.1145/3639299

[17] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, and A.-C. Ngonga Ngomo, "Tentris–a tensor-based triple store," in *International Semantic Web Conference*. Springer, 2020, pp. 56–73.