Exploiting Fusion Opportunities in Linear Algebraic Graph Query Engines

Yuttapichai (Guide) Kerdcharoen, Upasana Sridhar, Tze Meng Low
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
ykerdcha@andrew.cmu.edu, {upasanas,lowt}@cmu.edu

***Abstract*—Queries in a graph database are often converted into a sequence of graph operations by a graph query engine. In recent years, it has been recognized that the query engine benefits from using high-performance graph libraries via the GraphBLAS interface to implement time-consuming operations such as graph traversal. However, using GraphBLAS requires explicitly casting data into linear algebra objects and decomposing the query into multiple operations, some of which are expressible by the GraphBLAS. The combination of these two requirements translates into increased memory footprints and additional execution times. In this paper, we show that fusing different stages of the query engines into GraphBLAS calls can reduce the size of the intermediate data generated during the query. Furthermore, by relaxing the semi-ring constraints imposed by GraphBLAS, more aggressive fusions of the stages can be performed. We show a speedup of up to 1235.89x (8.82x on geometric average) relative to an open-source graph query engine using GraphBLAS (i.e. RedisGraph) for processing undirected subgraph enumeration queries.**

***Index Terms*—linear algebra, graph algorithms, high performance, graph database systems**

I. Introduction

Research in graph database systems, for example, [1], [8], has largely been independent of the field of graph algorithms. However, graph algorithms such as subgraph enumeration and graph traversal that are commonly used in graph database systems [4], [15], are the focus of much research in graph algorithms [3], [10]. More recently, there has been interest in leveraging the community-led GraphBLAS interface to improve the performance of graph databases. The GraphBLAS provides standard building blocks for graph algorithms in the language of linear algebra [12], [13]. RedisGraph [5] and MAGIQ [11] are two example graph databases that leverage the GraphBLAS at various stages in their graph query engines. The graph query engines benefit from high-performance implementations by linking to high-performance GraphBLAS backends such as SuiteSparse [6], [7]. We term query engines that utilize the linear algebraic approach (e.g., GraphBLAS) as *linear algebraic graph query engines*.

We make the observation that many graph query engines are implemented to carry out an execution plan comprising many stages, and selected stages in these query engines are implemented as a sequence of GraphBLAS routines. This approach introduces numerous inefficiencies in the use of GraphBLAS for the implementation of query engines. Firstly, there is a difference in the data format used by GraphBLAS and the more traditional query engine processes. Hence, there is a need to translate between the linear algebraic (matrices/vector) data format required by GraphBLAS and the "list of data" format in more traditional database systems. Second, each stage produced by the query engine may involve multiple GraphBLAS calls. This means (with current library-based implementations of GraphBLAS) that there is a need to materialize a large amount of intermediate data after every GraphBLAS call.

In this work, we focus on effectively using the GraphBLAS within graph query engines. Specifically, we identify opportunities to fuse different operations in a typical graph database query engine with more efficient use of GraphBLAS routines. Given that graph algorithms are generally memory-bound operations, fusing multiple operators will allow us to increase the arithmetic intensity while also reducing the amount of intermediate data that needs to be written out after each stage of the query engine. Specifically, we focus on 1) fusing across multiple linear-algebraic stages of the query engines using the masked-matrix multiplication operator in GraphBLAS, 2) introducing the use of *common neighbor identification* [10], [20] to fuse across different stages of the query engine, and to reduce the number of GraphBLAS calls required, and 3) relaxing the semi-ring constraints imposed by GraphBLAS while relying on the data access pattern of typical graph algorithms to expose even greater amount of fusion.

We demonstrate the feasibility of our approach by reimplementing the query engine in RedisGraph [5], demonstrating that these insights can provide performance improvements of more than 3 orders of magnitude (1235.89x) over the baseline RedisGraph implementation. Furthermore, these insights allow us to compute on large graphs and queries than the baseline implementation, demonstrating a reduction in the resource required to execute a query.

II. Anatomy of a Graph Query Engine

In this section, we describe the typical tasks performed by a graph query engine when servicing a graph query. Specifically, we use triangle enumeration, i.e., finding all triangles in a graph, as our running example throughout this paper.

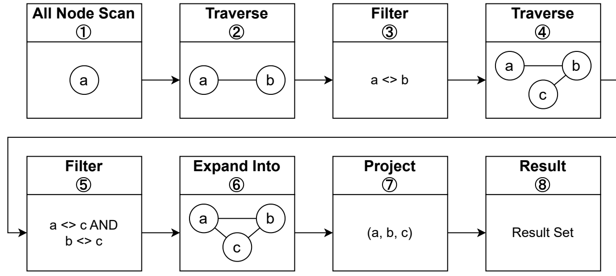

Fig. 1. A sequence of operations for a triangle enumeration query

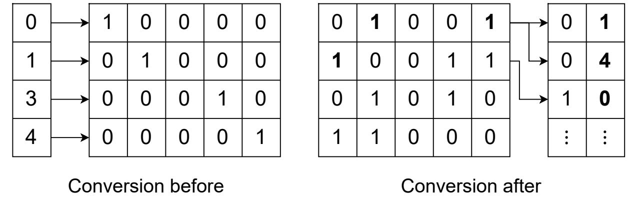

Fig. 2. Conversions between lists and matrices occurring before and after traverse in RedisGraph

A. Triangle Enumeration as a graph query

Triangle enumeration is a well-studied problem in the graph algorithm domain, where there are closed-form solutions/algorithms for counting and/or enumerating the different triangles formed in a graph [2], [16], [17], [19], [21], [22]. However, for a graph query engine, the query itself is first converted into a query execution plan, and the plan is executed in multiple stages shown in Figure 1. We provide a brief overview of the different stages in a graph query plan[1].

Specifically, triangles are enumerated via eight stages (① to ⑧). The sequence starts by (①) reading all nodes from a node list in graph database systems. These nodes form the starting node of a possible triangle. Starting from each node (②), a single step of traversal is performed from the starting node to each of its neighbors. These identified nodes are then considered to be possible second nodes of the triangle. This step creates a list of two-node walks. Then, the list is filtered (③) to eliminate walks where the first and second nodes are the same node. Three-node walks are then identified through a repeated process of traversal (④) and filtering (⑤). Having found a list of three node walks, we need to ensure that the third node must be connected to the first node. This is performed via another traversal (⑥) from the third node to the first node. This turns the list of three-node walks into a list of three-node cycles (triangles). The last two stages of the query process are to format the intermediate data representation into a user-friendly format (⑦) and then pack the result data for presentation to the user (⑧).

B. Graph query as linear algebra-like operations

The implementation of graph query engines with a linear algebraic approach has been proposed. The essence of MAGIQ [11] and RedisGraph [5] leverages the GraphBLAS APIs by casting various stages (e.g., traverse and filter) of the query as matrix-matrix multiplication of appropriately created matrices and the adjacency matrix representing the data graph. To illustrate: consider an S matrix that selects 4 out of the 5 nodes in a data graph. A is an adjacency matrix of the data graph. Multiplying S with A gives us a result matrix M whose different rows represent different neighborhoods for different nodes in the list as shown below:

$$M = SA = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Selector Matrix}} \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}}_{\text{Adjacency Matrix}}$$

Note that we assume all matrices are boolean matrices.

The result of the matrix-matrix multiplication is either used as inputs to other matrix-multiplication operations or will be converted to other data formats (e.g., lists of nodes/walks) where other database operations (e.g., relational join, filter) are employed in stages of the query process that have not been or cannot be written using the GraphBLAS API.

C. Translation to/from GraphBLAS and database operations

While the graph query engine benefits from high-performance GraphBLAS libraries, there is a need to convert between the matrix/vector view of GraphBLAS and the "list of data" view favored by traditional databases and expected by users. As such, there is a need to perform these conversions. Examples of such conversions are shown in Figure 2, where at the start, there is a need to convert from the "list of nodes" into a selector matrix. After the results have been obtained, there is a need to separate multiple results into individual records. In the above example, multiple identified neighbors of node 0 need to be separated into multiple separate records to create a list of two-node walks.

Furthermore, any additional data operations that are not implemented in terms of linear algebraic (specifically GraphBLAS) operations will require data representation conversion.

III. Opportunities for Fusion

Fusing multiple stages in the query engine allows us to utilize more efficient implementations in GraphBLAS. In this section, we illustrate different fusion opportunities to perform fusion within the query engine.

A. Filter with masked matrix-matrix multiply

Recall that in a typical graph query engine, the filtering of the intermediate result is performed as a separate stage from the traversal process. Furthermore, traversal in a linear algebraic query engine is implemented as a matrix-matrix

[1]It should be noted that while different graph database systems may produce different execution plans, the plans are largely similar in the stages and tasks performed per stage.

multiplication, and filtering can be implemented as an element-wise multiply. This implies that these two stages can often be implemented as a masked matrix-matrix multiplication as follows:

$$M = (SA \odot \neg F)$$

Here, F selects all the nodes that must not appear in M, where F often is the previous stage's M. This leverages the capabilities of GraphBLAS more efficiently, and it allows the fusion of the filtering as part of the write mask in the computation of the traversal step. This is also a common approach in many graph algorithm implementations using GraphBLAS [9]. Applying this to our triangle enumeration would mean that stages ② and ③ are fused.

B. Common neighbor identification

Finding common nodes that are in the neighborhoods of two or more other nodes is also known as common neighbor identification. Note that in the triangle enumeration example, this is performed as 3 separate stages (④ through ⑥). Coupled with the masked-matrix multiplication, these 3 stages can be combined into 2 steps using GraphBLAS as follows:

1) Finding all neighborhoods of interest, by performing a matrix-matrix multiplication of the adjacency node against a selector matrix where a 1 along the diagonal represents a node whose neighborhood is of interest[2].
2) All identified neighborhoods are intersected to find the nodes common across all neighborhoods.

To illustrate common neighbor identification, consider the case where we have identified two lists of candidates for 2 of the 3 nodes in a triangle enumeration query. This list of candidates can be implemented as 2 separate matrices as follows:

$$S_a = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ & & \vdots & & \end{bmatrix} \quad S_b = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ & & \vdots & & \end{bmatrix},$$

where S_a and S_b represent the two separate lists of candidates. Note that there are duplicated rows in S_a because each row could have multiple neighbors. In this case, the first node has 2 neighbors, and so 2 separate intersection operations (one with each neighbor of the first node) have to be performed to identify common neighbors[3].

After separately multiplying S_a and S_b with the adjacency matrix of the data graph, A, different intermediate matrices (M_a and M_b) where each row representing the neighborhoods of each node from their respective candidate lists are created.

[2]It should be noted that this step is already a fusion of multiple traversals to find all neighborhoods of interest.

[3]A way to identify how many rows need to be duplicated is to see how many times a node is represented in the list view as shown in Figure 2 (after conversion).

The intersection of the neighborhoods can then be performed via an element-wise multiplication of both intermediate matrices as follows:

$$M_c = M_a \odot M_b = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ & & \vdots & & \end{bmatrix}$$

which, along with the original candidate lists S_a and S_b, can be interpreted as the following three-node cycles: cycles $(0, 1, 4)$, $(0, 4, 1)$, $(1, 0, 4)$, and so on. Note that there may be multiple common nodes in a single intersection.

1) Fused implementation: Note that M_a and M_b can be obtained by performing masked-matrix multiplication, and they are element-wise multiplied together to obtain the list of common neighbors. The number of GraphBLAS calls can be further reduced by recognizing that the element-wise multiplication can only return a result if an element M_b is already present in M_a. This means we can use M_a as the write-mask for the masked-matrix multiplication to compute M_b, i.e.,

$$\begin{aligned} M_c &= M_a \odot M_b \\ &= M_a \odot S_b A \end{aligned}$$

This reduces the number of GraphBLAS operations from 3 to 2 and also reduces the need for producing results that may be eliminated by the intersection (element-wise) operation.

C. Fusing multiple common neighbor operations

Notice that computing common neighbors identification operation requires multiple passes through the adjacency matrices. As the number of neighborhoods that need to be intersected increases, this increases the number of matrix multiply that has to be performed. An alternative is to keep the previously computed results, but this is only an option for small graphs and queries.

Recall that each row in S_x represents a selector that picks out a particular row in the adjacency matrix. Multiple non-zero elements in the same row can serve to pick out the appropriate rows in the adjacency matrix via the same matrix multiplication routine. Therefore, to reduce the number of passes, one possibility is to merge both the candidate lists S_a, S_b into a single paired candidate list $S_{a,b}$, i.e.,

$$S_{a,b} = S_a + S_b$$

where each row in $S_{a,b}$ represents all nodes whose neighborhood needs to be intersected.

However, while the combined selector matrix $S_{a,b}$ can pick out the appropriate rows of the adjacency matrix, implementations using the GraphBLAS yield incorrect results because this formulation violates the properties of the semi-ring.

Specifically, the properties of a semiring require that:

- *Additive Identity* ($Id_\oplus$). The addition must have an element that, when added with any element x, will yield x as the result, i.e.

$$Id_\oplus \oplus x = x \oplus Id_\oplus = x$$

.

- *Multiplicative Annihilator*. When multiplying $Id_{\oplus}$ with any element, the result will always be $Id_{\oplus}$, i.e.,

$$Id_{\oplus} \otimes x = x \otimes Id_{\oplus} = Id_{\oplus}.$$

To illustrate the problem, let us consider the problem of performing the intersection as part of the reduction/additive operator in GraphBLAS. Here, the multiplicative operator is used to select the approprate rows of the adjacency matrix.

In order for the intersection to be correct, the reduction operator has to be the logical `AND` operator. This means that additive identity has to be `true` or 1. However, `true` cannot be the multiplicative annihilator as multiplying 1 with any input x with return x. Similarly, 0 or logical `false` cannot be the additive identity, as performing intersection with 0 will always return 0.

To solve this problem, we implement a custom masked matrix-matrix multiply routine that utilizes a custom multiplicative operator that does not have the semi-ring constraints. Our routine has a similar data access pattern as a regular sparse matrix-matrix multiply. Specifically, our custom operator has to have the following properties:

1) When the element in a row of the selector matrix is 1, an intersection is performed using the selector row and the resulting vector. This means that the selected row must be passed unmodified to the intersection operator.
2) When the element in a row of the selector matrix is 0, the result vector must not be changed, since that row does not contribute to the result. To preserve the results after the intersection, the selected row must be all 1.

This means that the custom operator changes the value of the selected rows, as shown in the truth table below:

p	q	$p \otimes q$
0	0	1
0	1	1
1	0	0
1	1	1

The astute reader will recognize that the truth table above is the same as the boolean *implication* operator. This expected, since the value of the row used for intersection is dependent on whether that particular row is selected or not.

Implementing this operator using GraphBLAS matrix-matrix multiply routine yields incorrect results as its specification is only defined where both input elements are common in the index sets of the input matrices, i.e. $k \in \mathbf{Ind(S)} \cap \mathbf{Ind(A)}$. However, this operator needs to be performed for all values in the selector matrix S, including those containing implied zeros, i.e. $k \in \mathbf{Ind(S)} \cup \mathbf{Ind(S)}'$.

IV. Experimental Results

We compared the performance of our optimization against the original RedisGraph (Commit No. *ddcbcf4*) and Neo4j (v.5.2.0, community edition).

Fifteen datasets from the Stanford Network Analysis Project (SNAP) [14] were selected from various categories. A total of 9 different queries were used. These queries were undirected versions of motifs commonly found in biochemistry, neurobiology, ecology, and engineering [18]. The undirected versions of the motifs are visualized in Figure 3.

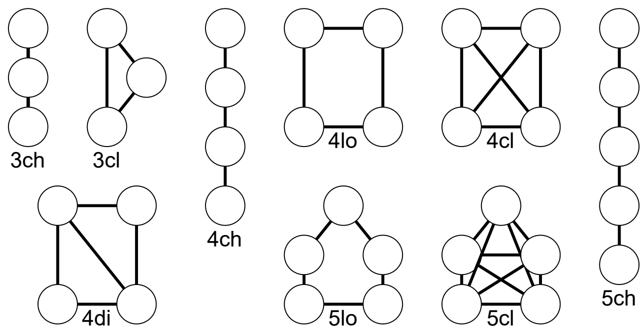

Fig. 3. The nine motifs used as input queries. with their name as a label for the experiments.

A. Experimental setup

We experimented on a dual-socket machine with 128 GB DRAM memory and 24 threads. Each CPU is an Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz. We modified the original RedisGraph by replacing the default query engine with different optimizations applied. Suitesparse 7.3.0 was used as the GraphBLAS implementation. The following implementations were created:

- *Baseline*. This is the default/unmodified RedisGraph implementation.
- *Fused T&F*. RedisGraph with a fused traverse and filter implementation using GraphBLAS in Section III-A.
- *Fused CNI&F*. RedisGraph with a fused common neighbor identification and filter implementation using GraphBLAS in Section III-B. We also replaced all traverse operations with appropriate common neighbor identification operations.
- *Beyond GB*. RedisGraph with our fused common neighbor identification and filter implementation in Section III-C.
- *Beyond GB W/O Conversion*. The Beyond GraphBLAS implementation but conversions between successive common neighbor identification operations are removed.

For all RedisGraph implementations (including the baseline), we adjusted `BATCH_SIZE` in the traverse operation (as well as the fused operation) to `INT_MAX` to reduce the number of GraphBLAS operation calls. As a side effect, it used more space. For Neo4j, we set the maximum heap size to 120 GB, close to our DRAM capacity.

B. Performance breakdown

To identify the performance improvements yielded by the different techniques, we compared the runtimes and the breakdown for all 4 implementations running against the largest graph in our experiment, *cit-Patents* (3774768 nodes, 33037894 edges), and the triangle enumeration query. The

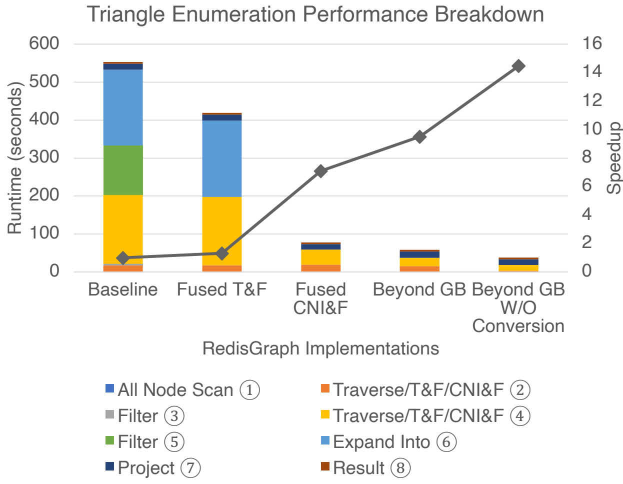

Fig. 4. The runtime breakdown plot for the triangle enumeration query with the cit-Patents graph. The plot shows the trend of incremental speedups starting from the baseline RedisGraph and implementations we created, which provide upto 14.47x speedup.

results are shown in Figure 4. The built-in RedisGraph profiler was used to obtain the breakdown of runtime.

A modest 1.32x speedup is gained from fusing the traverse and filter operations using GraphBLAS using masked-matrix multiplication. The largest gain of 7.1x speedup over the baseline is attained by switching to a common neighbor implementation in GraphBLAS. Using our custom common neighbor implementation, we gain a further speedup, resulting in a gain of 9.51x speedup over the baseline. Removing data conversions between stages ② and ④, a 14.47x speedup over the baseline is attained.

C. Applicability to other motifs and graphs

We picked Neo4j, the baseline RedisGraph, and Beyond GB W/O Conversion implementations to evaluate the applicability of the approaches to more graphs and queries. Figures 5 and 6 report runtimes for all graphs and queries, except for timeout (1 hour) or out-of-memory cases. We illustrate the timed-out workloads as a bar hitting 3600 seconds. The out-of-memory workloads are shown as no bar. Note that timeout and out-of-memory status are not mutually exclusive; we show the first status each workload encountered. Plots are ordered by the size of the original graph, starting with the smaller graphs from left to right and from top to bottom.

1) Three-node queries: Figure 5 shows the performance for all three graph databases on three-node queries. As these three-node queries often require only a few stages, most workloads can be completed within the time limit and available memory. Only the three-node chain (3ch) query could not be completed within the allocated time frame (Neo4j) or the memory constraints (RedisGraph-based implementations). This is expected since both graph sizes and characteristics of query subgraphs often affect the number of query results, which affects both time and space.

Across the board, the linear algebraic (RedisGraph-based) implementations were often significantly faster than the Neo4j (non-linear algebraic) implementation. For the largest graph, Neo4j did not complete within the time limit, but the linear algebraic approaches ran out of memory well before the time limit. This could suggest that either better memory management is required for linear algebraic approaches, or the Neo4j query could have needed more time to run out of memory. This is something we intend to investigate further.

More importantly, the optimizations introduced in this paper are often more beneficial for the three-node cliques (3cl) than three-node chains (3ch). This is because the three-node clique query enabled the use of a common neighbor approach to find the third node that is in the neighborhood of multiple nodes. This is consistent with the performance breakdown shown in Figure 4, where most of the performance gained in our implementations comes from the common neighbor optimization.

2) Queries with larger node count: Figure 6 reports the performance of the same implementations on larger query graphs. Generally, our custom implementation returned the query results faster than the baseline RedisGraph, and Neo4j implementations.

We also find that our trend in terms of speedup significantly improves from, at most, 25x speedup over the baseline RedisGraph implementation to more than 1200x speedup. The better speedup happens in the workload whose graph contains a small number of results relative to the graph size, such as p2p-Gnutella04 (1235.89x). Since such workloads spend most of the time on the operation, we are capable of improving like traverse and filter, attaining higher speedup can be expected as we gain in this workload.

One of the interesting insights is that there are workloads that the baseline RedisGraph cannot outperform a non-linear algebra implementation like Neo4j. However, our implementation in RedisGraph can outperform Neo4j in all workloads all implementations can complete. This highlights that there is an opportunity to improve the linear algebra approaches in the domain of graph query engines, as shown in this work.

V. Conclusion

This paper highlighted multiple fusion opportunities for using linear algebra within graph query engines. We argued that by leveraging the identified fusion opportunities, we can reduce the number of GraphBLAS calls, which in turn reduces both the intermediate data that needs to be written out, and the number of times the data format has to be switched between the matrix/vector view required by GraphBLAS, and the ”list of data” view typical of databases.

We demonstrated the performance improvement from exploiting these identified fusion opportunities by replacing the query engine within RedisGraph. Our modified database attained over three orders of magnitude speedup relative to the original RedisGraph implementation. We also showed that our implementation is more efficient in terms of compute and resource utilization as compared to the baseline implementation, as our modified implementation is capable of performing queries on larger data graphs and more complex queries.

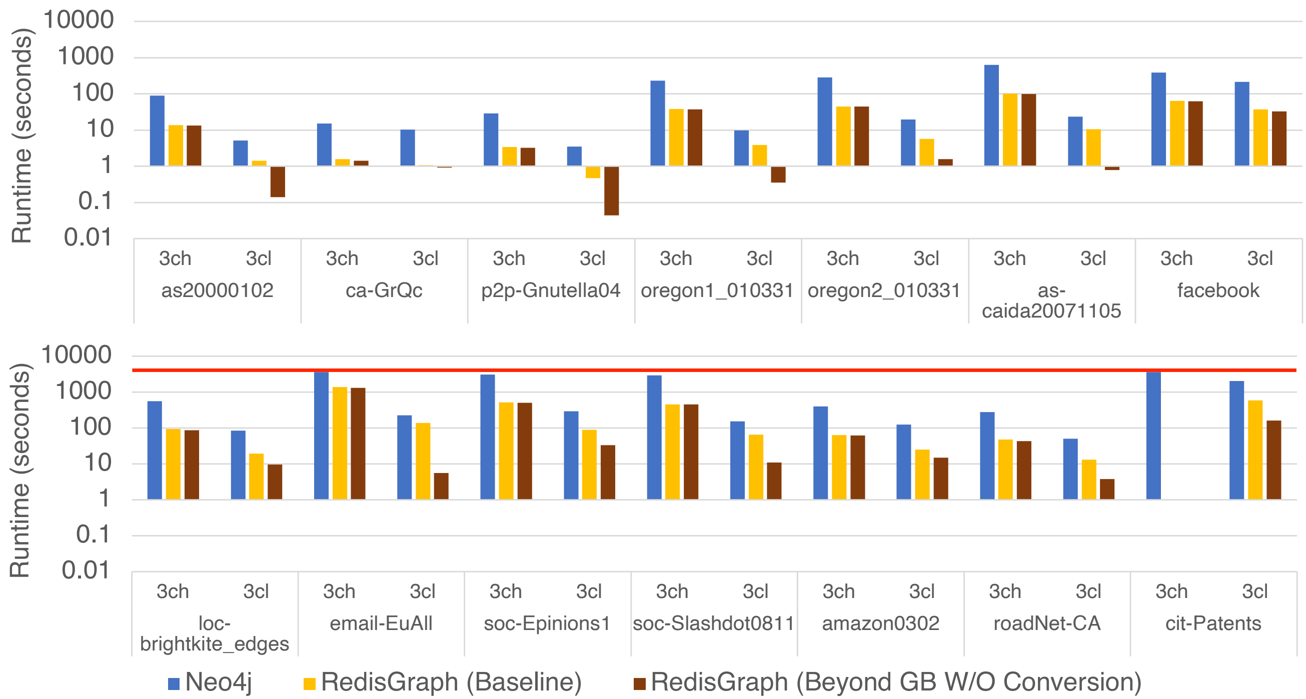

Fig. 5. On geometric average, a 2.22x speedup improvement is attained by running our implementation using our operation fusion techniques against the baseline RedisGraph for three-node subgraphs. Against Neo4j, our implementation can gain an 11.1x speedup on the geometric average.

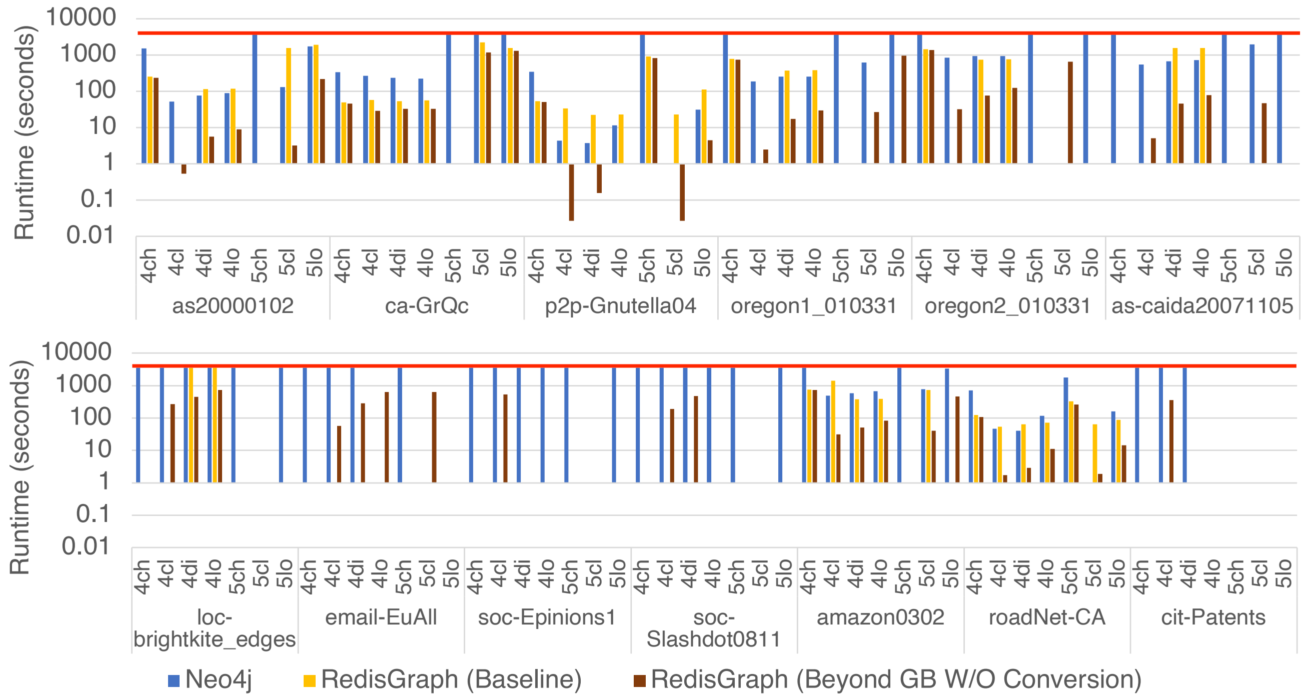

Fig. 6. Our implementation outperforms the baseline RedisGraph and Neo4j by 8.82x and 14.9x speedup on geometric average, respectively. The results also show that our implementation can complete most of the workloads than the other implementations.

While our work highlights the benefits of linear algebraic graph query, it also shows a need to robustly bridge the gap between the high-performance library standards for graph algorithms and the use cases of interest to graph database system communities. Future work could explore how node properties can be included in the search for common neighbors, performing graph algorithms on dynamic graphs, and using linear algebra equivalence as rule-based optimizations for graph queries.

A key work we intend to explore is to identify "good" linear algebraic formulations from a given graph query. Our measure of "goodness" would be one that translates to an execution plan that requires either lesser resources and/or execution time overall.

ACKNOWLEDGMENT

Yuttapichai Kerdcharoen is sponsored by a Ph.D. scholarship from the CMKL University.

REFERENCES

[1] DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS. 2023.

[2] Ariful Azad, Aydin Buluc, and John Gilbert. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, Hyderabad, India, May 2015. IEEE.

[3] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing Breadth-First Search. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Salt Lake City, UT, November 2012. IEEE.

[4] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Computing Surveys*, page 3604932, June 2023.

[5] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 285–286, Rio de Janeiro, Brazil, May 2019. IEEE.

[6] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Transactions on Mathematical Software*, 45(4):1–25, December 2019.

[7] Timothy A. Davis. Algorithm 10xx: SuiteSparse:GraphBLAS: parallel graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software*, page 3577195, January 2023.

[8] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. KÙZUˆ* Graph Database Management System.

[9] Vitaliy Gleyzer, Andrew J. Soszynski, and Edward K. Kao. Leveraging Linear Algebra to Count and Enumerate Simple Subgraphs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, Waltham, MA, USA, September 2020. IEEE.

[10] Wentian Guo, Yuchen Li, and Kian-Lee Tan. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[11] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, Dresden Germany, March 2019. ACM.

[12] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, USA, 2011.

[13] Jeremy Kepner, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, Jose Moreira, Peter Aaltonen, David Bader, Aydin Buluc, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, and Andrew Lumsdaine. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, Waltham, MA, September 2016. IEEE.

[14] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1), jul 2016.

[15] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment*, 12(4):390–403, December 2018.

[16] Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru Popovici, Franz Franchetti, and Scott McMillan. First look: Linear algebra-based triangle counting without matrix multiplication. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Waltham, MA, September 2017. IEEE.

[17] Tze Meng Low, Daniele G. Spampinato, Anurag Kutuluru, Upasana Sridhar, Doru Thom Popovici, Franz Franchetti, and Scott McMillan. Linear Algebraic Formulation of Edge-centric K-truss Algorithms with Adjacency Matrices. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, Waltham, MA, September 2018. IEEE.

[18] R. Milo. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, October 2002.

[19] Daniele G. Spampinato, Upasana Sridhar, and Tze Meng Low. Linear algebraic depth-first search. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 93–104, Phoenix AZ USA, June 2019. ACM.

[20] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. Efficient Parallel Subgraph Enumeration on a Single Machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 232–243, Macao, Macao, April 2019. IEEE.

[21] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Waltham, MA, September 2017. IEEE.

[22] Abdurrahman Yasar, Sivasankaran Rajamanickam, Jonathan Berry, Michael Wolf, Jeffrey S. Young, and Umit V. CatalyUrek. Linear Algebra-Based Triangle Counting via Fine-Grained Tasking on Heterogeneous Environments : (Update on Static Graph Challenge). In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–4, Waltham, MA, USA, September 2019. IEEE.