Efficient Property Projections of Graph Queries over Relational Data

Mikael Morales[1], Vlad Ioan Haprian[1], Srinivas Karthik[2], Danica Porobic[3]
Laurent Daynés[1], Anastasia Ailamaki[2,4]
1: Oracle Labs 2: EPFL 3: Oracle 4: RAW Labs

ABSTRACT

Specialized graph data management systems have made significant advances in storing and analyzing graph-structured data. However, a large fraction of the data of interest still resides in relational database systems (RDBMS) due to their maturity and security reasons. Recent studies, in view of composability, show that the execution of graph queries over relational databases, (i.e., a graph layer on top of RDBMS), can provide competitive performance compared to specialized graph databases.

While using the standard property graph model for graph querying, one of the main bottlenecks for efficient query processing, under memory constraints, is *property projections*, i.e., to project properties of nodes along paths matching a given pattern. This is because graph queries produce a large number of matching paths, resulting in a lot of requests to the data storage or a large memory footprint, to access their properties.

In this paper, we propose a set of novel techniques exploiting the inherent structure of the graph (aka, a *graph projection cache manager*) to provide efficient property projections. The controlled memory footprint of our solution makes it practical in multi-tenant database deployments. The empirical results on a social graph show that our solution reduce the number of accesses to the data storage by more than an order of magnitude, resulting in graph queries being up to 3.1X faster than the baseline.

1 INTRODUCTION

Graph data representing connected entities and their relationships appear in many application domains such as social media, finance, health, and sciences. With the constant increase in the amount of graph data that needs to be managed and processed, graph databases are becoming more and more popular. Graph databases offer more intuitive and efficient ways of extracting information about the data, through *graph analytics* and *graph querying*. This has led to a proliferation of systems being built over the last decade, including Neo4j [3], Amazon Neptune [1], Oracle PGX [12] and TigerGraph [4].

However, due to the reliability, performance, security, and maturity of relational databases, most of the data is still managed by RDBMS [16]. Specifically, the usage of specialized graph engines is not always possible, given that the data have to be exported from RDBMS. Furthermore, exporting data not only adds multiple overheads, such as time and memory, but also raises privacy and security concerns. When managing sensitive data, such as banking or medical information, many regulations need to be respected

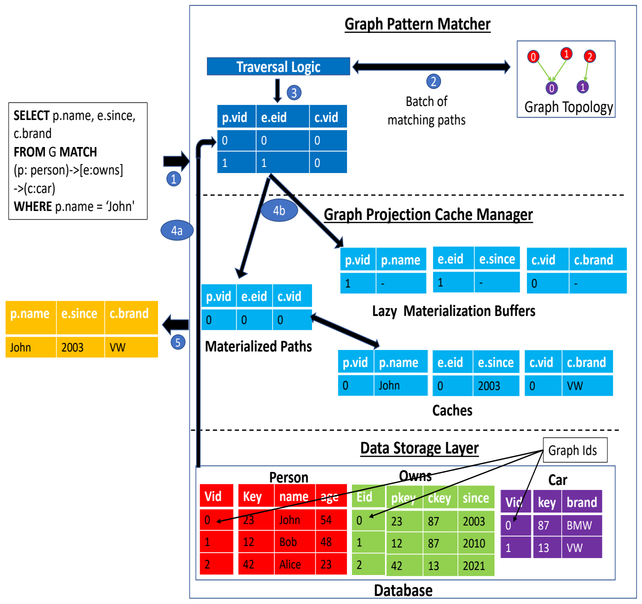


Figure 1: System Architecture Overview

which can prevent the usage of any external engine that does not comply with necessary certifications. Also synchronizing the two copies of data in face of updates would add to the overheads. This has led to the design of architectures for supporting graph querying and analytics directly over relational databases [9, 11, 12, 14–16].

In this paper, we focus on *graph querying* [10, 17] where the objective is to identify sub-graphs matching a given pattern (e.g., diamond shape). Further, we use *property graph* data model which is one of the most prevalent graph models in industry (e.g., Neo4j [3]). From a data modelling point of view, a node represents an entity, an edge represents a relationship between entities, and a property represent attributes of an entity or a relationship [7].

Architecture: Figure 1 represents the overall design of our system. When a query request is received (Arrow: 1), it goes through the Graph Pattern Matcher component, specifically the traversal logic, that traverses the topology (built over the relational data) to compute all the paths (sequence of vertices and edges) matching a path pattern on a given graph (Arrow: 2). The Graph Pattern Matcher assumes that an optimal matching order of the path pattern variables is given by a query optimizer. The order of a variable in the matching order constitutes the level of the query (i.e., variable p is at level 0 if matching order starts with p, etc). The graph topology, capturing graph relationships, is built over the relational rows using an additional fixed size row identifier column, i.e., Vertex id's (Vid) and Edge id (EiD) for each row in the relational tables. Note that the identifier-based topology consumes much less memory as opposed to topology containing the actual data (for example, variable length primary keys). Also, the storage layer is loosely coupled with the matching layer - thanks to the identifier columns - which works off the shelf while being independent of the storage layout. In addition to this, the fixed size vertex and edge identifiers opens opportunities for graph topology compression and improves cache locality during pattern matching.

Once the matching patterns are identified, for each pattern the *properties* of nodes or edges along the paths are projected (Arrow: 3). For instance, when traversing a social network graph, one might be interested in extracting the age, the photos, etc. of all the people (nodes) who are friends with each other (pattern). Under memory constraints, projecting the properties, hence referred to as *property projection*, is challenging. This is because the properties stored in relational data storage need to be accessed for all paths and at each level of every path (as described in detail in Section 2.1) (Arrow: 4a/4b). For instance, one needs to do a table access on the person table searching for p.vid =0. Similarly, for other matched id's.

The objective of this paper is to minimize the number of accesses to the data storage (to fetch properties) under memory constraints.

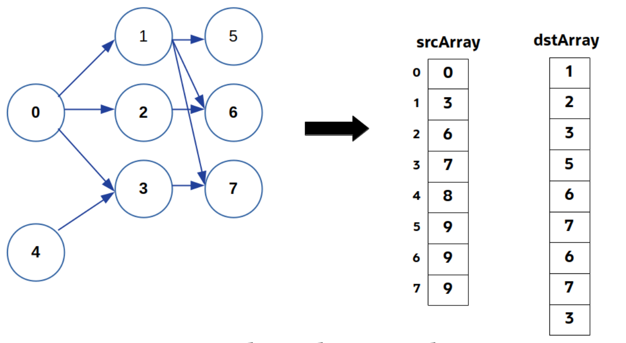
Existing systems can be classified as follows:

- Systems that do eager projections, meaning that before the pattern matching starts they access tables to fetch properties of vertices that can potentially appear in the final result. The properties are then carried along with the vertex throughout the whole matching pipeline. As graph queries can easily generate a large number of intermediate results this approach will suffer from a huge memory footprint [2, 14, 15].
- Systems that do lazy projections, meaning that they first match all the paths according to the given path pattern and then access tables to fetch the properties needed in the final result. In order to minimize the cost of the table access requests those systems either store the data into fast random access in-memory storage offering a quick look-up interface, or they materialize huge number of paths before accessing tables in order to amortize its cost. Both approaches are very memory intensive [11, 16].

Since existing solutions requires large amount of memory, they do not work well in multi-user deployments where memory capacity is limited. In this paper, to mitigate the above issue, we propose a novel *graph projection cache manager* that does lazy projections in small batches of paths. In order to further reduce the number of accesses to the data storage, we use techniques such as caching, prefetching and lazy materialization. The distinctive feature of our solution is that all of these mechanisms are tailored to leverage the graph topology resulting in significant performance benefits.

Specifically, **the following are the contributions** in this paper:

- A caching mechanism tuned to the way paths are produced that maximizes efficient reuse of previously retrieved properties and minimize the number of accesses to the underlying data storage.
- A prefetching technique that can prefetch properties of vertices and edges that are likely to appear in the next paths produced by the Graph Pattern Matcher. The technique leverages the graph topology to predict the next element (vertex or edge) that will appear in a path and prefetches the property value for this element.
- Evaluation of the improvements brought by our techniques on a social graph from Linkbench graph benchmark. The result suggests that this solution reduces the total number of storage accesses by up to an order of magnitude, thus, resulting in up to 3.1X speed up in query running time. Furthermore, the memory footprint of our solution is a small fraction compared to the full-materialization scenario.



	srcArray
0	0
1	3
2	6
3	7
4	8
5	9
6	9
7	9

dstArray
1
2
3
5
6
7
6
7
3

Figure 2: A graph topology stored as a CSR

2 EFFICIENT PROPERTY PROJECTIONS

In this section, we describe our techniques for efficient projection of properties for paths produced by the Graph Pattern Matcher. Without loss of generality, we only discuss how our approach works for projecting vertex properties. Projecting edge properties is supported using the same techniques. We use Compressed Sparse Rows (CSR) format to represent graphs. CSR can be seen as a compact representation of the vertex neighbourhood information, enabling the efficient exploration of the graph topology by following in memory pointers/offsets. The Graph Pattern Matcher is in charge of computing all the paths (sequence of vertices and edges) matching a path pattern on a given graph. In order to speed up the processing of paths, it leverages an in-memory representation of the graph i.e. the CSR. The paths are generated by orderly navigation of the graph in depth-first-search manner.

Figure 2 represent an example CSR of a graph. The neighbourhood information for each vertex is captured using two arrays namely - *srcArray* and *dstArray*. For vertex index *i*, its neighbours are indexed in the range [dstArray[srcArray[i]], dstArray[srcArray[i+1]]). Note that the range value is non-inclusive at the end. For example, neighbours of 0 are {1,2,3} which are captured in the range [dstArray[0],dstArray[3]).

2.1 Baseline implementation

For ease of presentation, we leverage Figure 1 to present the baseline control flow. The primary difference from our solution is that the baseline does not go through the middle cache manager layer, rather goes directly go data storage layer (Arrow 4A) after Step 3.

(1) Given the input query, the Graph Pattern Matcher traverses the graph and identify the paths that match the path pattern.
(2) We fetch paths from the matcher batch by batch.
(3) Once a path is extracted, we can then fetch its properties from the underlying tables. To do so, we identify the levels that need to project properties. In the example in Figure 1, the

query is requesting ($p.name$), hence we request the properties from the relational table bounded to the vertex identified by $person(p)$. A table access is performed to fetch only the properties of a single vertex. In general, N table accesses are executed, where N is the number of levels of the path pattern doing projections.
(4) (4A) The table access operator fetches the requested properties from the data storage.
(5) Once the properties are retrieved, we can safely output the projection result of the path.

Steps (2) to (5) are repeated for each path produced in Step (1).

The main bottleneck in the baseline implementation is that it requires $N*M/BATCH\_SIZE$ table accesses in total, where M is the number of paths matching the path pattern and $BATCH\_SIZE$ is the number of paths produced by the Graph Pattern Matcher at once. This makes this baseline implementation potentially inefficient.

2.2 Graph Projection Cache Manager

The baseline property projection pipeline offers multiple opportunities for improving performance and avoiding redundant work. Namely, the same vertex can often appear in paths belonging to different batches requiring separate table accesses to retrieve the same property value. Since table accesses are not free, a prudent use of the working memory would be caching the values that are fetched by a given table access as those values are likely to be needed also for subsequent batches. In addition to this, graph connectivity information can be used to predict, i.e., which properties will be required in the near future in order to be prefetched and boost the efficiency of the cache. Since all of these techniques trade-off space for time by using memory to store properties and avoid table accesses, it is necessary to distribute the available memory judiciously between the different data structures.

Motivated by these insights, we devise the following mechanisms:

(1) A **caching** mechanism that can efficiently reuse the existing results, and avoids the need to retrieve the same properties multiple times. This mechanism reduces the total number of table accesses required. In order to amortize the cost of table accesses needed to resolve cache misses, we accumulate them across batches and lazily resolve them when enough are available.
(2) A **data prefetching** technique that leverages the information provided by the Graph Pattern Matcher and the in-memory graph index (CSR) to determine vertices likely to appear in upcoming paths. Those vertices will be prefetched to further reduce cache misses.

The next subsections will describe the design of the data structures and mechanisms presented above in detail.

2.3 Caching

The intuition behind caching is that, if a vertex i is connected to multiple vertices then it is likely to appear in multiple paths returned by the Graph Pattern Matcher. In other words, multiple paths returned by the Graph Pattern Matcher belonging to different batches will contain vertex i. Therefore, caching the properties of vertex i can help. Highly connected vertices are expected to be present in many graph topologies, e.g., in social graphs.

When processing a graph query, i.e. (a)->(b)->(c), each level and each relational table involved at each level has an independent, in-memory, cache associated with it. This cache is row-oriented and has a fixed size. The cache data structure contains a mapping from vertex identifiers to the properties projected at this level. The caches are filled after getting the results from table accesses.

To illustrate how caches are allocated, using the input graph query will generate an instantiation tree, closely following the graph schema. Each node in the tree includes the information on whether or not it is doing projection. If it does, it will have a pointer to a cache data structure.

Having cache per level also facilitates using different cache size per level, or skipping a level altogether. Caching efficiency, i.e., the percentage of projections that can be satisfied from the cache, depends on the graph connectivity and degree of overlap between the subsequent paths. Having the versatility of a cache data structure allows us to configure caches differently based on the topology of the queried graph, resulting in a much more efficient usage of memory.

2.3.1 *Handling cache misses.*

Resolving cache misses for batch-at-a-time system can potentially be very expensive. For example, if only properties of a few vertices are missing in the batch, then the whole table will be accessed unnecessarily. In order to overcome this issue, we design a *lazy materialization* technique that allows us to accumulate the cache misses across multiple batches and only resolve them when enough are accumulated. By this, we amortize the cost of table accesses needed to resolve the cache misses.

Materialized paths: The materialized paths data structure stores the paths fetched from the Graph Pattern Matcher that hit the cache entirely. If the properties of all the node levels of the path requiring projections are already in the cache, the result can be written out without needing any table accesses. To materialize a path we need to store the vertex identifier (vid) of each level. As caches are indexed by vids, we can use them to access the required properties.

Lazy materialization buffer: The lazy materialization buffer is used to store paths fetched from the Graph Pattern Matcher. However, **the paths stored in the lazy materialization buffer may not hit the caches at all levels**.

Unlike the materialized paths data structure, the data stored in the lazy materialization buffer contains more than just vertex identifiers (vids). Considering that a subset of levels of these paths could potentially have hit the cache, the lazy materialization buffer also contain results that can store these properties.

Copying the available projected value from the cache(s) to the materialized buffer makes it possible to decouple the management of the cache from that of the materialization buffer. In particular, the cache doesn't have to interact with the materialization buffer to decide what value to evict. The lazy materialization buffer size is tunable based on the query path and graph topology.

2.4 Data prefetching

Our topology-aware prefetching algorithm is based on the level at which we are prefetching the data. Let us assume, without loss of generality, that we are executing a graph query with the following path pattern: $(a) \rightarrow (b) \rightarrow (c)$. Depending on the level, we do the following:

- **Source level** (a): Our Graph Pattern Matcher guarantees the order in which matching paths are returned, with respect to the source of the paths. Using this invariant, given a set of matching paths and a maximum value x, where x is the source vertex with the biggest identifier of the set. We know that for every vertex y that we will prefetch, we should ensure that $y > x$. This means that the vertices which are going to be prefetched have not yet been considered as a potential source of paths by the Graph Pattern Matcher, but are likely to be in the near future. Thus, by prefetching them, we will get their properties in advance and avoid potential table access operations later.
- **Other levels** (b), (c): we **leverage the information given by the graph topology**. When prefetching at level i, we can look-up the graph index for the next neighbors of the current parent (siblings of the current vertex). Those neighbors will appear in the next paths explored by the Graph Pattern Matcher and depending on the graph pattern and the input graph, have a high likelihood to be part of the final paths returned by the Graph Pattern Matcher.

The amount of data to prefetch is determined as follows: 1) The size of the cache is set to always be greater than the BATCH_SIZE. 2) Fetching missing properties from the lazy materialization buffer will refill the cache with N properties. 3) The remaining space due to duplicated vertices in lazy materialization buffer (the difference between the cache size and N) is available for prefetching data.

2.5 Overview of control flow

In this section we give an end to end view of the control flow used in our algorithm (see Figure 3). We have focused on popular OLAP scenarios, however, extending to OLTP setting is an interesting future work. We start by extracting paths produced by the Graph Pattern Matcher. For every path, we do the following:

(1) Check if all the properties of the vertices in the path are already cached. If yes, then we save it into the materialized paths data structure.
(2) Otherwise, we save it in the lazy materialization buffer. We also copy the properties of the vertices in the path, that are in the caches (if any) to the result set. The goal of this step is to delay the full materialization of these paths as much as possible.
(3) Once any of these two data structures are full, we stop extracting paths from the Graph Pattern Matcher and start outputting rows.
(4) For all paths in the materialized paths data structure, we consume them directly by fetching the properties from the caches and output them.
(5) Then, once the lazy materialization buffer is full, we access tables in order to fetch properties of the vertices present in the buffer. We also use the data prefetching algorithm to prefetch properties for the upcoming paths.
(6) When the table access results are ready, we cache them. Then, we start consuming the paths from the lazy materialization buffer by fetching the properties from the caches and outputting them. This operation will be done until the buffer is empty.

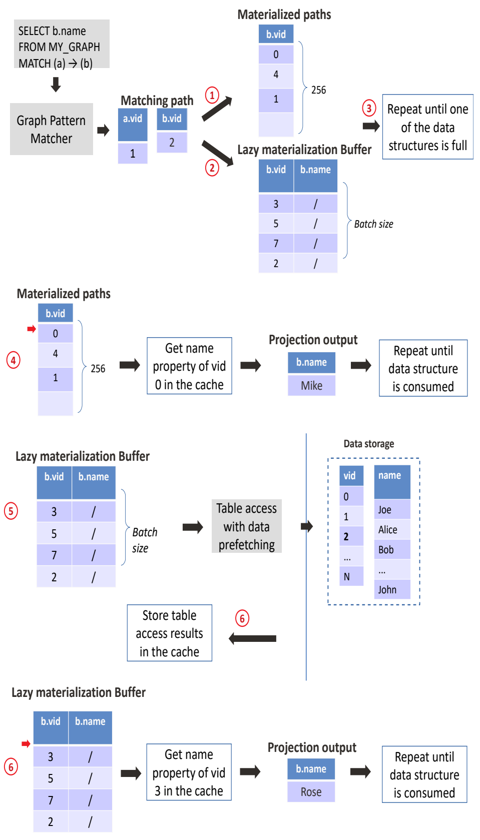

SELECT b.name
FROM MY_GRAPH
MATCH (a) → (b)
Graph Pattern Matcher
Matching path
a.vid
b.vid
1
2
Materialized paths
b.vid
0
4
1
256
Lazy materialization Buffer
b.name
3
5
7
2
/
Batch size
Repeat until one of the data structures is full
Get name property of vid 0 in the cache
Projection output
Mike
Repeat until data structure is consumed
Table access with data prefetching
Data storage
vid
name
Joe
Alice
Bob
John
N
Store table access results in the cache
Get name property of vid 3 in the cache
Rose
1
2
3
4
5
6

Figure 3: Visual representation of the control flow

3 EXPERIMENTAL EVALUATION

In this section, we evaluate the techniques described in the previous section and quantify their impact on the graph query execution time based on the number of table access saved. We use the batch-at-a-time, described in Section 2.1, as our baseline. We also assess the slow down in execution time of our solution compared to the ideal scenario of full materialization.

3.1 Experimental setup

All our experiments are carried out on a machine with Intel Xeon E5-2699 CPU with 512GB DDR4 RAM while running Linux kernel 4.1.12. We use System X as the relational backend. Recall that our goal is to minimize the number of table access requests for accessing the properties under memory constraints. The savings achieved, in terms of table access, by our techniques would remain the same independent of the underlying RDBMS (or data storage system). Thus our evaluation is performed only on X in this work by building the Graph Pattern Matcher and graph projection cache manager on top it (as captured in Figure 1). Finally, to avoid noise, we report the average values over five executions.

Datasets: We conduct experiments on a social graph using the data from Linkbench [8] graph benchmark. We primarily use a

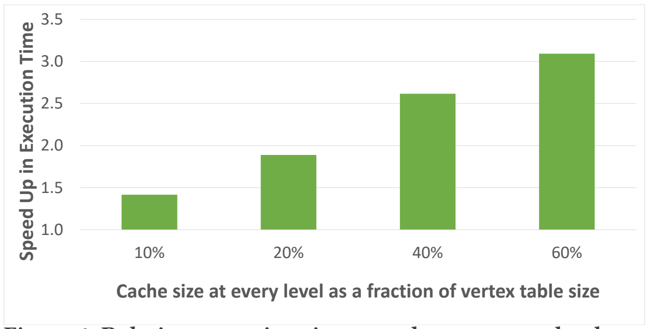


Figure 4: Relative execution time speed up compared to baseline with caching

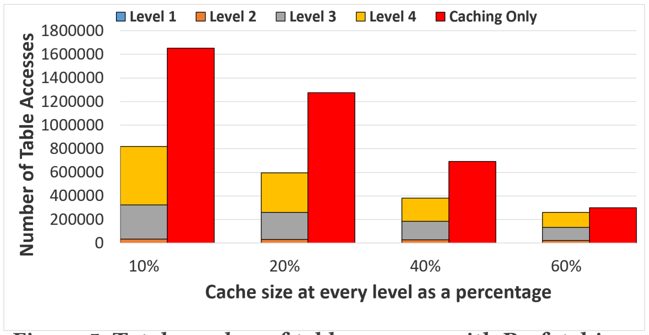


Figure 5: Total number of table accesses with Prefetching.

scale factor 10 (SF10) of this benchmark which creates a graph of 65′645 vertices and 1′938′516 edges. In our experiments, we use the *Person* and *personKnowsPerson* tables.

We use a PGQL query that asks for all paths of length four and projects the age property at every level without any filters. The query pattern does not matter in our context, as we are prefetching only at first level. Even though we are working on a relatively smaller graph size, the number of paths returned for the query is **5'782'122'733**. Since our goal is efficient property projection through minimal number of table accesses, our chosen dataset/query is large enough to evaluate the benefits of our proposed techniques.

3.2 Caching

We now evaluate the efficacy of caching. We empirically fix the optimal batch size based on the data size. Further, we enable caching at every level equally by defining the cache size value as a percentage of the total size of the properties to be projected in the vertex table. Note that level corresponds to an order of a variable in the matching order. Overall, as captured in Figure 4, the results show that even while using a cache size of just 10% we get a speed up of 1.4X compared to the baseline as we are able to execute 2 times less table accesses. The speedup factor increase further to around 3.1X with around **10 times less** table accesses, if we increase the cache size to 60%.

Caching helps since the properties of a vertex that are projected and cached will end up being reused as the graph is well connected. More connected the graph is, more the reuse and hence more the benefits of caching.

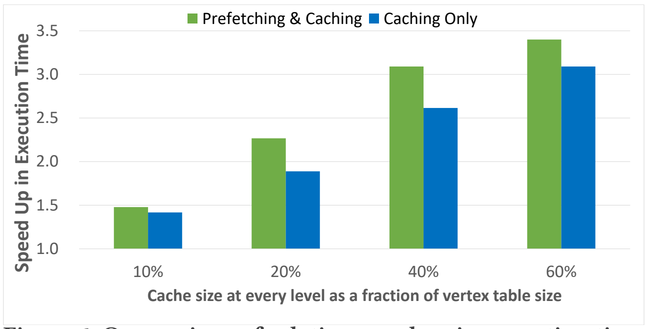


Figure 6: Comparison of relative speed up in execution time over baseline for with and without data prefetching.

3.3 Data prefetching

Figure 5 shows that by aggressively prefetching unprocessed neighbors, it meaningfully reduces the number of table accesses required to fetch projected properties. Thus leading up to **2 times fewer** table accesses with prefetching than without it (i.e. caching only). Data prefetching helps in reducing the number of table accesses at every level, not only at the source. The more neighbors a vertex has, the more likely it will still have unprocessed neighbors when the lazy materialization buffer is full, thus, triggering table accesses. Hence, our prefetching algorithm will be able to prefetch the unprocessed neighbors and ensure a cache hit at the next iterations, leading to fewer table accesses.

It is important to see that the impact of data prefetching is more pronounced with smaller cache sizes. This is because, as we increase our cache size, the likelihood of having the data available in the cache, without needing any other mechanism, increases. Thus, the impact of data prefetching is reduced. Given that our techniques are aimed at deployments with memory constraints, having efficient processing with a small memory footprint is particularly important. Figure 6 presents a comparison of the relative speedup over baseline of graph query time with and without prefetching (i.e. caching only). We are able to achieve up to **20% faster** response times using data prefetching compared to caching only strategy. In summary, the total number of table accesses to the storage layer reduces by 4X to 12.7X for cache sizes ranging from 10% to 60%.

3.4 Sensitivity analysis

In this experiment, properties needed for projection are materialized into in-memory arrays providing fast random access. This is the scenario wherein we have enough memory to materialize the entire relational data. We compare the performance of our techniques to this ideal case for various cache sizes. We use a batch size of 1000 paths with caching and data prefetching enabled. We use LDBC SF1 dataset for this evaluation for which the graph query yields 198'136'160 matching paths.

Figure 7 shows the relative slow down of the execution time compared to the full materialization setting. We see that the slow down ranges from 2.4X for 10% cache size configuration to as low as 1.4X with 60% cache size. The takeaway is that our memory saving techniques allow us to execute graph queries with projections without severely impacting the performance as compared to full-materialization case.

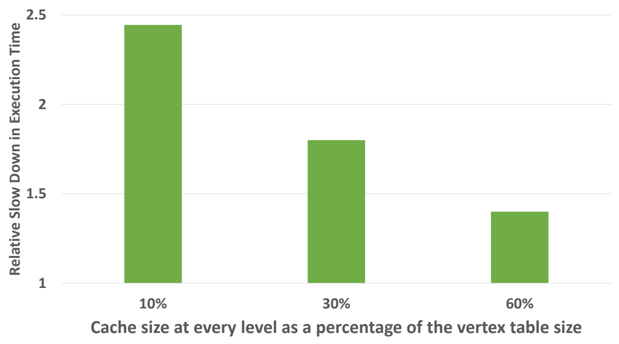


Figure 7: Relative execution time slow down over full materialization

4 RELATED WORK

Constant increase in graph data has led to design of several specialized graph databases such as Neo4j [3] and TigerGraph [4]. Supporting graph queries directly over relational data has the advantage of avoiding ETL overheads, data duplication and potential security issues. Systems using this approach can be categorized as follows:

Graph-Core: They typically extract graphs from the relational data and materialize it, and answer graph queries from the in-memory graph view. Apart from the data duplication issue, another downside is that they operate on stale data. Examples of such systems are GraphGen [18] and Oracle PGX [12].

Relational-Core: In this line of work, the graph data is stored in relational databases. Once the graph data is persisted into a relational database, given a graph query, a translation layer is used to convert it into SQL equivalents for querying over relational tables. Often this translation is non-intuitive and cumbersome, as well as less performant than native graph querying. SQLGraph [15], Vertexica [14] and Microsoft SQL Server Graph [2] fall into this category.

Graph-Relational-Core: In this line of work, the boundary between the graph layer and the relational data is blurred by building a graph topology or index over it. DB2 Graph [16] and GraphFusion [11] are examples of such an approach (including our system). The main difference between our system and GraphFusion is that GraphFusion requires an in-memory copy of the relational data as the graph topology has tuple pointers to the in-memory relational tables. This allows them to have (assume) fast random access to the underlying properties at the expense of additional memory usage, which we overcome in this paper. Neither GraphFusion nor DB2 Graph address the problem of efficient property projection in face of memory constraints.

In short, existing solutions requires large amount of memory, they do not work well in multi-user deployments where memory capacity is limited. On the other hand, property projection for graph queries resembles tuple construction in column stores. Chapter 4 of the excellent tutorial on column store system design [5] presents trade-offs in evaluating join operators and projecting attributes that are not necessary for evaluation. Lazy materialization is very advantageous in column stores as it improves hardware utilization by increasing cache locality, amortizes decompression cost and offers vectorization opportunities [6], however, additional bookkeeping makes early materialization preferable for queries that are not selective. Caching and prefetching techniques are fundamental design building blocks in database buffer manager components [13]. We take inspiration from extensive work done in the context of relational query processing and adapt it in the graph querying context by focusing on leveraging the topology information such as connectivity. We expect our solution to be beneficial in any of the above-mentioned graph-relational composable systems.

5 CONCLUSIONS

Composing relational and graph query processing has seen a lot of attention in recent times. In this paper, starting from a batch processing technique as a baseline, we propose mechanisms such as caching, prefetching and lazy materialization by utilizing the graph structure to minimize the number of table accesses. We also introduce a control flow that encapsulates all these techniques to provide a memory-efficient solution. Overall, our proposed techniques can reduce the number of table accesses by nearly an order of magnitude compared to baseline, and making the graph queries run over relational data significantly faster.

REFERENCES

[1] [n.d.]. Amazon Neptune. https://aws.amazon.com/neptune/

[2] [n.d.]. Microsoft SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017

[3] [n.d.]. Neo4j. https://neo4j.com/

[4] [n.d.]. Tigergraph. https://www.tigergraph.com/

[5] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[6] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *Proc. of IEEE ICDE*. 466–475.

[7] Renzo Angles. 2018. The Property Graph Database Model. In *Proc. of AMW*.

[8] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. of the ACM SIGMOD*. 619–630.

[9] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[10] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *Proc. of EDBT*. 520–523.

[11] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. Extending In-Memory Relational Database Engines with Native Graph Support. In *Proc. of EBDT*. 25–36.

[12] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A Fast Distributed Graph Processing Engine. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 58:1–58:12.

[13] R Jauhari, Michael J Carey, and Miron Livny. 1990. Priority-hints: an algorithm for priority-based buffer management. In *VLDB Conf.* 708–721.

[14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: your relational friend for graph analytics! *Proc. VLDB Endow.* 7, 13 (2014), 1669–1672.

[15] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proc. of ACM SIGMOD*. 1887–1901.

[16] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proc. of the ACM SIGMOD*. 345–359.

[17] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proc. of ACM GRADES Workshop*. 1–6.

[18] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. GraphGen: Adaptive Graph Processing Using Relational Databases. In *Proc. of the ACM GRADES Workshop*. 9:1–9:7.