Amine Mhedhbi

Email: `amine.mhedhbi@uwaterloo.ca` , *Web:* http://amine.io/

EDUCATION

University of Waterloo, Waterloo, ON
Doctor of Philosophy, Computer Science — Jan. 2017 - May. 2023

Concordia University, Montréal, QC
Bachelor of Engineering, Computer Engineering — Sep. 2012 - Apr. 2016

RESEARCH INTEREST

The overarching goal of my research is to ***build data systems capable of efficient processing of graph data at scale***. My Ph.D. research focuses on developing novel query processing, optimization, and storage techniques for querying graph-structured relations. To that end, I have been designing and implementing the GraphflowDB in-memory property graph DBMS which allows me to rethink core DBMS components. ***In the future, I aim to research building data systems for machine learning on graphs in additional to analytical query processing problems***. I have completed two internships at Microsoft Research leading to two on-going collaborations, both of which expanded my research interests into transactional processing and cloud infrastructure for data systems.

CONFERENCE & JOURNAL PUBLICATIONS

[9] **Optimizing One-time and Continuous Subgraph Queries using Worst-Case Optimal Joins. A. Mhedhbi**, C. Kankanamge, S. Salihoglu. ACM Transactions on Database Systems (TODS) 2021.

[8] **A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems. A. Mhedhbi**, P. Gupta, S. Khaliq, S. Salihoglu. IEEE International Conference on Data Engineering (ICDE) 2021.

[7] **Columnar Storage and List-Based Processing for Graph Database Management Systems.** P. Gupta, **A. Mhedhbi**, S. Salihoglu. Proceedings VLDB Endowment (PVLDB) 2021.

[6] **Optimizing Subgraph Queries by Combining Binary and Worst-case Optimal Joins. A. Mhedbhi**, S. Salihoglu. Proceedings VLDB Endowment (PVLDB) 2019.

[5] **The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey.** S. Sahu, **A. Mhedbhi**, S. Salihoglu, J. Lin, MT. Özsu. VLDB Journal (VLDBJ) 2019.

[4] **The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing.** S. Sahu, **A. Mhedbhi**, S. Salihoglu, J. Lin, MT. Özsu. Proceedings VLDB Endowment (PVLDB) 2018. **(Best Paper Award)**

WORKSHOP & DEMO PAPERS + TUTORIALS

[3] **Modern Techniques for Querying Graph-Structured Relations: Foundations, System Implementations, and Open Challenges. A. Mhedhbi**, S. Salihoglu. Proceedings VLDB Endowment (PVLDB), Tutorial 2022.

[2] **LSQB: A Large-Scale Subgraph Query Benchmark. A. Mhedhbi**, M. Lissandrini, L. Kuiper, J. Waudby, G. Szárnyas. ACM GRADES-NDA: Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2021.

[1] **Graphflow: An Active Graph Database.** C. Kankanamge, S. Sahu, **A. Mhedbhi**, J. Chen, and S. Salihoglu. ACM International Conference on Management of Data (SIGMOD), Demonstration, 2017.

RESEARCH IMPACT

- 2021: Our query processor and optimizer combining binary and worst-case optimal joins [**6, 9**] is integrated in Alibaba's graph processing engine GraphScope .
- 2021: Huawei's R&D group exploring the integration of A+ indexes [**8**] and our compressed columnar storage [**6**] for the design of a new graph DBMS.
- 2021: The **l**arge-scale **s**ubgraph **q**uery **b**enchmark (LSQB) [**2**] is used as part of the continuous integration performance pipelines for the commercial products at RelationalAI , MemGraph, and Neo4j .

INVITED TALKS

- Taming Large Intermediate Results using Factorization: A system perspective.
  Factorized Databases Workshop. Zurich, Switzerland. Aug. 2022.
- Taming Large Intermediate Results for Joins over Graph-Structured Relations.
  DATA lab at Northeastern University. Boston, MA, USA. Nov. 2022.
  Huawei-Edinburgh University Joint Lab. Edinburgh, Scotland. Apr. 2022.
  Centrum Wiskunde & Informatica. Amsterdam, Netherlands. May 2022.

GRADUATE RESEARCH EXPERIENCE

Microsoft Corporation Jun. 2022 - Sep. 2022
Research Intern, DMX Group (Mentor: Phil Bernstein)

- Researched the architecture and performance characteristics of data-sharing database management systems such as Azure SQL Hyperscale.
- Implemented a new lock manager and researched various optimizations to locking protocols under contention.

Microsoft Corporation Jun. 2021 - Sep. 2021
Research Intern, DMX Group (Mentors: Christian Konig & Vivek Narasayya)

- Researched orchestration of database as a service (DBaaS) tenants to minimize failovers during automatic cluster upgrades.
- Formulated the problem as an optimization problem and produced an approach that is capable of taking the specific cluster deployment instance into account when creating a cluster upgrade schedule.

TEACHING EXPERIENCE AS TEACHING ASSISTANT

(F=Fall, W=Winter, S=Spring)

Teaching Assistant at the University of Waterloo:
CS 338 - Computer Applications in Business: Databases. S17, F17, W18, S18 & F19.
CS 330 - Management Information Systems. W19, F19 S20, F20, W21 & F21.
CS 348 - Introduction to Database Management. F18 & W22.
CS 115 - Introduction to Computer Science 1. F16, W17 & F22.
Instructor at Concordia University:
ELEC/COEN 390 - Principles of Design and Development. F15 & W16.

AWARDS

Microsoft Research Ph.D. Fellowship, 2020-2022.
Facebook Research Ph.D. Fellowship, 2020-2022. (declined)
VLDB Best Paper Award, 2018.
David R. Cheriton Scholarship, University of Waterloo, 2017-2018.
Cheriton Symposium Poster Presentation - 1st Place, University of Waterloo, 2018.
SIGMOD Travel Award, 2017.
Graduate Entrance Scholarship, University of Waterloo, 2016-2017.

SERVICE

External Reviewer
IEEE Transactions on Knowledge and Data Engineering (TKDE) 2020 - Present.
Semantic Web journal (SWJ) 2021.
The Web Conference (WWW) 2020.

INTERN EXPERIENCES

Société Générale (Corporate & Investment Banking) May 2016 - Aug. 2016
Technology Analyst Intern

- Rearchitected a hedge fund lending app from monolithic to service-oriented.
- Helped reduce the length of the release cycle from 6 to 4 weeks.

Ericsson Sep. 2015 - Mar. 2016
Software Engineering Intern

- Researched virtualization for physical IP Multimediate Subsystem (IMS) nodes.
- Prepared a fully functional demo for World Mobile Congress 2016.

Concordia University, Montréal, QC May 2015 - Aug. 2015
Undergraduate Research Assistant (Advisor: Jelena Trajkovic)

- Verified empirically the theoretical foundations and guarantees of an existing Optical Network-on-Chip design.

InterDigital, Inc. Jan. 2015 - Mar. 2015
Software Developer Intern

- Revamped the UX of a smart network access manager mobile application.
- Prepared a demo to showcase the app's features in a congested network.

Thales Group (Academic Partnership) Nov. 2013 - Nov. 2014
Research Intern (Advisors: Prof. William Lynch & Prof. Glenn Cowan)

- Researched and designed various digital filters and amplitude detection algorithms for aerospace use cases.

Immersion Corporation Sep. 2013 - Dec. 2013, May 2014 - Aug. 2014
Software Engineering Intern

- Implementation of haptic effect lib APIs exposed in C and Java for mobile developers to abstract hardware details and remove parameterization.
- Upgraded and integrated a haptic effects lib for the AOSP kitkat release.
- Automated manual complex AOSP framework builds with our lib integration.

COMPUTER SKILLS

Technology: C++, Python, Java, Javascript, SQL, Shell Scripting.
Tools: Git, Numpy, Pandas, Scikit-learn, PyTorch.
Languages: English, French, Arabic.

MISC

Member of a research team studying "Fostering Experiential Partnerships with Theatre and Mental Health: Realistic Family Therapy Training for Psychology and Acting Students". Work presented at University of Waterloo Teaching and Learning (UWTL) Conference 2022.

INTERESTS

Improvisational theatre, cooking, football freestyle, running, boxing, and lifting weights.

Research Statement

Building Data Systems for Graph Data Management At Scale

Amine Mhedhbi, University of Waterloo

The overarching goal of my research is to ***build data systems capable of efficient processing of graph data at scale***. Graph data refers to datasets that can be modeled as a set of nodes and edges that represent the entities and their relationships in applications. My Ph.D. thesis argues that DBMSs that optimize for *graph analytical workloads* need to integrate a suite of modern techniques, most important of which are: (i) novel *worst-case-optimal join algorithms* that adopt column-at-a-time over the traditional table-at-a-time join paradigm; (ii) *factorized query processing*, which is a technique to compress the intermediate and final results of queries; and (iii) a flexible indexing sub-system to cache and reuse subqueries in a workload. I do systems-oriented research and have integrated each of these techniques into a modern graph DBMS (GDBMS) called GraphflowDB. Such an integration requires tackling a set of system and algorithmic research questions that necessitate rethinking and introducing novel techniques to core DBMS components such as the query executor, the query optimizer, and the storage layer.

Graph data is integral to a wide range of analytical applications, such as finding fraud patterns in financial transaction networks *e.g.*, fraud rings, or recommendation patterns in social networks *e.g.*, clique-like communities. The workloads of these applications, colloquially referred to as "graph analytical workloads", often require finding complex and large graph patterns on data that primarily consists of many-to-many relationships, *i.e.*, where node records can connect to many other node records through the same relationship. For example, a Twitter recommendation application searches for diamonds in their who-follows-whom graph, where a user can follow many other users. In database terminology, these workloads involve relational queries containing complex many-to-many joins that lead to an explosion in the size of intermediate relations. This poses serious challenges for traditional analytical DBMSs, which were not optimized for complex many-to-many joins.

My work on GraphflowDB have appeared in top data management venues such as the Proceedings of the VLDB Endowment (PVLDB) and the VLDB journal [1,2], ACM's SIGMOD conference [3], IEEE's ICDE conference [4], and ACM's TODS journal [5]. I have co-presented a tutorial on modern techniques for querying graph data at VLDB 2022 [6]. I have also co-authored a survey paper on the users of graph processing software, which won the VLDB 2018 Best Paper award [7]. The output of my research had industry impact. For instance, some of the techniques I developed for analytical DBMSs within the query executor and optimizer were adopted by GraphScope [8], the graph processing engine product of Alibaba. A benchmark that I co-desgined gained both academic and industrial interest and is used by database vendors such as RelationalAI, Memgraph, and Neo4j. In the remainder of this statement, I cover my research in the three techniques above and cover some other work I have done during my Ph.D.

1 Worst-case Optimal Join Algorithms

The de facto join algorithms used in most production DBMSs are known as *binary join (BJ) algorithms*, which adopt the table-at-a-time joining paradigm. The theory of worst-case optimal join (WCOJ) algorithms [9,10], demonstrates that BJ algorithms are provably sub-optimal on cyclic join queries under many-to-many relations in the following sense: the number of intermediate results they can generate can be polynomially larger than the maximum outputs for a query Q. For example, consider finding triangles in an input graph, whose edges are modeled as an `Edge(from,to)` relation. In relational algebra this is equivalent to the cyclic self-join query $\texttt{Q}_\Delta$`:= Edge(a,b),Edge(b,c),Edge(a,c)`. If the graph has m edges, then the worst-case output size of $\texttt{Q}_\Delta$ is $O(m^{3/2})$, while there are input graphs on which any BJ algorithm could generate $\Omega(m^2)$ many tuples because they compute open triangles as an intermediate step. Surprisingly, even a decade after this theory, many systems today can be either prohibitively slow on such queries if the input relations are many-to-many since they still rely on BJs. The WCOJ theory proposes join algorithms that adopt a novel column-at-a-time paradigm to mitigate this sub-optimality. These algorithms order the columns in the query in some order, *e.g.*, , a, b, c, and iteratively find all prefixes of (a)'s, then (a, b)'s and (a, b, c)'s etc. using multiway intersections as a core algorithmic primitive.

It is clear that these algorithms would find their best applications on graph workloads, which frequently contain cyclic join-heavy queries over datasets with many-to-many relations. However, existing implementations when I started to consider WCOJs for graph workloads were not practical for two reasons: (i) they proposed processors that completely ignored BJs, which are still very efficient on many queries and benefit from decades-long engineering; or (ii) they ignored core architectural design principles of modern systems, such as pipelined and vectorized query processors, or dynamic programming-based (DP) join optimizers, making them impractical to integrate into existing systems. The first research questions I undertook were: *How can WCOJ algorithms be integrated in modern query processors? When do WCOJs outperform BJs and vice versa? How can queries be optimized to generate efficient plans that mix WCOJ and BJ algorithms?*

1.1 End-to-end WCOJ DBMS Integration

I proposed an end-to-end practical system integration approach that can seamlessly use a mix of WCOJ and BJ algorithms in query plans. My approach enhances many components of the DBMS without deviating from the architectural principles of modern analytical query processors. Specifically, I designed and implemented a novel query operator called `E`xtend-Intersect that can do WCOJ-style multiway intersections, a new cost-model called *intersection-cost (i-cost)* to assign costs to plans with `E`xtend-Intersect and other BJ operators, a novel cardinality estimator based on summary statistics of small-size cyclic graph patterns, and an enhanced DP-based join optimizer.

One important observation I made during this project was that although the WCOJ theory advises system designers to use column-at-a-time joins, it gives no advice as to how to optimize the choice of the column ordering. My work is the first to: i) show that the choice of the column order could make orders of magnitude difference in query execution time; and ii) study how to pick a column order in sub-plans that executed WCOJ algorithms. I showed that using my novel i-cost metric, one can design a DP optimizer that picked efficient column orderings. My work was also the first to systematically study for which queries WCOJs outperformed BJs based on the structures of queries. I further showed how to seamlessly mix BJ and WCOJ algorithms in a DP-based optimizer to generate novel hybrid plans that were not considered in prior systems. I have integrated this solution in the GraphflowDB system and showed that GraphflowDB plans can be up to two orders of magnitude faster in execution time than others. This work was published in PVLDB and presented at VLDB 2019 [1]. Following my work, parts of my approach, have also been adopted by other commercial and academic systems. For example, Alibaba uses our hybrid plans, i-cost metric, and cardinality estimator in GraphScope [8], their graph processing engine.

1.2 Multi-query Optimization for Continuous Queries

The follow-up work to this is the study of query evaluation for queries that find patterns over dynamic datasets. I and colleagues, studied the principles of using WCOJ algorithms for *continuous query evaluation* at scale. Continuous queries are those that track the inserted or deleted graph patterns in a dynamic graph that is continuously updated. They are the backbone of several time-sensitive applications, such as recommendations or fraud detection. These applications register many graph patterns as queries in a DBMS, e.g., different size cliques or cycles and the challenge for the DBMS is to find *joint query plans* that can evaluate a set of queries that together share computation. This problem is known as *multi-query optimization (MQO)* in the DBMS literature. Based on a published technique, my colleagues and I showed that the problem of only detecting the inserted and deleted patterns can be formulated as computing a set of *delta queries*. We further showed that when evaluating delta queries using WCOJ algorithms and intersection-cost as an estimated cost metric, finding the optimal plan is NP-hard, so traditional DP-based approaches would not be suitable. I developed a greedy multi-query optimization algorithm to find efficient plans and integrated this approach into a new continuous query processor I designed and implemented in the context of GraphflowDB. We showed that using WCOJ algorithms on delta queries can be up to two orders of magnitude faster than prior solutions on single cyclic continuous queries. We further showed that the plans generated by my greedy optimizer can speedup system throughput by up to 4x over running queries independently. This work was published in ACM's TODS journal [5].

2 Factorized Query Processing

Acyclic join queries and acyclic components of queries, pose a different challenge for DBMSs under many-to-many relationships. Specifically, their results can be inherently very large, regardless of the algorithms used to compute them. Consider finding 2-hop paths in a graph with m edges, which in Datalog is expressed as `Q`$_{2H}$ `:= Edges(a,b),Edges(c,b).` If the graph is highly skewed, *e.g.*, there is a single b node b_1 that all edges target, there can be $\Omega(m^2)$ many output regardless of the algorithm used. The dominant relation representation system in existing DBMSs is to use flat tuples, e.g., using m^2 tuples, $\{(a_1, b_1, c_1), (a_2, b_1, c_1), ..., (a_m, b_1, c_m)\}$, to represent the output of `Q`$_{2H}$. The recent *theory of factorized databases [11]* has shown that outputs of (many-to-many) acyclic joins can often be more compactly represented as unions of Cartesian products and exploit conditional independence of variables in the query. For example, another factorized representation of the same output is $\cup_{i=1...m} a_i \times b_1 \times \cup_{j=1...m} c_j$, because given a fixed binding for b in the output, all of the a and c bindings are independent.

My work on factorizations was based on the following observation: the core query processing algorithms described in the theory of factorization [11] assumed a setting in which algorithms would take as input and output full materialized factorized relations. Following this, existing systems work on factorization designed fully materialized processors that directly processed large tries. This deviates significantly from the well understood principle adopted in modern analytical query processors that every primitive operator in the system should operate on a block of flat tuples (aka vectorized processing) to be efficient on modern CPUs. Existing approaches further abandon the well established pipelined architectures, making them impractical to integrate into existing DBMSs. My research questions challenged this assumption and asked: *How can modern vectorized and pipelined query processors adopt factorization without direct processing on factorized representations?*

2.1 Factorized Vector Execution

I proposed a novel query processor architecture that extends vectorized and pipelined processors to benefit from factorization. In this design, instead of passing a single vector of flat tuples between operators, operators pass *a set of factorized vectors*[1]. At any point in time, each vector can be *flat* and represent a single value, or *unflat* representing a set of values. The relation represented is the Cartesian product of the factorized vectors. Performing many-to-many joins inevitably computes Cartesian products, e.g., the join of tuple (a_1, b_1) with $\cup_{j=1...m}(b_1, c_j)$ leads to m tuples $\cup_{j=1...m}(a_1, b_1, c_j)$, with many repetitions of (a_1, b_1) values. Instead of performing this Cartesian products as in existing systems, the join operator in my design flattens the vector (a_1, b_1) into a single value, and appends a second vector of $\cup_{j=1...m}(b_1, c_j)$ to the relation, delaying the Cartesian product. Importantly, every single primitive operator in the system still performs computation on a single vector of tuples. For example, we can ensure that every binary operation, *e.g.*, comparison of values, either operates on two unflat vectors of the same size, or one flat and one unflat vector. I implemented my design into GraphflowDB. In our work, which was published in PVLDB and presented at VLDB 2021 [2], we showed that processing using factorized vectors can lead to speedups of up to three orders of magnitude over vanilla vectorized processing on acyclic join-heavy workloads.

3 Graph-based Materialized Views and Indexes

In addition to my work on the query processor and optimizer of GDBMSs, I worked on their indexing sub-system. Aside from their differences in their data models and query languages, the most pronounced difference between RDBMSs and GDBMSs is in their indexing sub-systems. Virtually all GDBMSs implement join indexes (a.k.a adjacency list indexes) that index the relationships of nodes to support fast joins of nodes with their neighbours. Such join indices are not common in RDBMSs. My work here was based on the observation that existing GDBMSs have system-specific and fixed adjacency list structures, making them efficient only on a fixed set of workloads. For example, some systems supported indexes in which all of the edges were indexed together, while others partitioned them based on edges types (e.g., `Follows` relationships would be indexed separately from `LivesIn` edges), while others on a combination of node and edge types. This meant each system was optimized for a specific workload and could not be tuned without remodeling application data. My research questioned this unflexibility and asked: *How should the indexing sub-systems of GDBMSs be designed to be tunable to support a wider range of workloads?*

3.1 A+ Indexes

I proposed a new indexing subsystem for GDBMSs called *A+ Indexes* that supported indexing edges based on a large set of neighborhood queries. As such, A+ indexes support a limited form of *materialized views*, which in RDBMSs refers to relations that are results of queries over base relations. An important property of adjacency list indices in GDMBSs is that they are accessible in constant time through positional node ID offsets. An important objective in my design is to identify a set of materialized views that can be indexed and still be accessible using such positional offsets. I identified two types of materialized views that satisfy this constraint: (i) node-partitioned "1-hop" indexes that index immediate neighborhoods of vertices and support predicates and partitioning on arbitrary node properties; and (ii) my design is the first to observe that by using edge IDs as positional offsets, certain 2-hop paths can also be indexed and partitioned using edge properties. My design further proposed mechanisms to support nested partitioning criteria, and compression techniques to improve the space-efficiency of these indices. Along with two Master's students, I implemented this design in GraphflowDB and described a complete end-to-end system solution that modified the query processor and optimizer of the system to effectively use A+ indexes as part of its core join operators, such as its WCOJ `Extend-Intersect` operator. In our paper, published in the ICDE 2021 Conference [2], we demonstrated how to optimize GraphflowDB to multiple workloads simply by tuning A+ indexes and that in some cases this had negligible storage and update overheads with significant performance gains. For example our 1-hop views lead to up to 20x speedups with an overhead of 1.1x on storage.

4 Other Work and Collaborations

I have worked on a number of other projects during my PhD. Early in my career, I participated in an extensive user survey on the use of graph processing software. The goal of our survey was to understand use cases, general trends, and primary challenges of current graph processing software with the goal of gaining insights into important problems that the community should focus on. The results of this survey and the regular conversations I have had with users of graph software heavily informed the focus of my Ph.D. research. Specifically, this work clarified the set of applications to optimize for when designing a GDMBS and identified scalability as the top challenge amongst users. The work was awarded the Best Paper Award in the VLDB 2018 conference [7] and was later extended with further analysis and published by the VLDB journal [12].

[1]In our original publication [2], this was referred to as *list-based processing*. We later adopted the terminology of *factorized vectors*.

I have also designed a new large-scale subgraph query benchmark (LSQB) that focused on queries ignored by prior benchmarks. LSQB tests the join performance as a choke-point on read-heavy subgraph queries. This work was published in the GRADES-NDA 2021 workshop co-located with SIGMOD conference [13]. Since its publication, LSQB has gained both academic and industrial interest and is used internally by vendors such as RelationalAI, Memgraph, and Neo4j.

In the summers of 2021 and 2022, I was a research intern at Microsoft Research which led to two currently on-going collaborations. Both of these collaborations expanded my research interests into transactional processing and cloud infrastructure for data systems. In my first internship, I researched orchestration of Database-as-a-Service systems. The goal is to ensure predictable performance during planned cluster upgrades by minimizing failovers. I formulated an optimization problem and produced a general approach that is capable of taking the specific cluster deployment instance into account when creating a cluster upgrade schedule. Our approach is currently under deployment internally by my collaborators to be tested against Microsoft's real workloads. In my second internship, I researched the performance characteristics of transactional processing in data-sharing systems such as Azure SQL Hyperscale. Specifically, I implemented a new lock manager and researched various optimizations to locking protocols under contention. My collaborators and I enumerated various scenarios that lead to contention and focused so far on the worst-case. We are currently making progress towards studying other contention cases and evaluating our solution on complex workloads, *e.g.*, YCSB and TPC-C.

5 Future Work

I have developed a research style that utilizes theoretical insights to inform the architectural designs of data processing systems. I further implement and test my designs in real systems to ensure the practicality of my ideas. I aim to continue adopting this style of research as I believe it gives my research the potential to do large real-world impact. I outline below a set of short-term and long-term future research projects that I am excited to start in the first few years of my faculty career.

5.1 Short Term Research

I plan to continue working on factorized query processing both in GDBMSs as well as analytical RDBMSs.

Factorized Processing Using Definitions: The factorized representation system I implemented inside GraphflowDB is called *f-representations* in the theory of factorization. An even more compressed representation system in the theory is known as *d-representations*, which extend f-representations with nested resuable expressions. For example, d-representations can represent a k-hop path query starting from a single node v in a graph using space commensurate with the number of nodes and edges in the k-hop subgraph around v, by nesting each j-hop path's result (for $j < k$) and reusing it. This can be polynomially more compact than *f-representations*, which cannot nest and reuse expressions. There is currently no practical proposal for integrating d-representations to pipelined vectorized modern processors, which I plan to address in my research.
Factorization to Optimize for Aggregation-Heavy Business Reporting In addition to graph workloads, factorization has the potential to be used for aggregation-heavy business analytical reporting, which analytical RDBMSs are optimized for, as long there are some many-to-many joins in these queries. As a simple example, count queries on factorized vectors can be computed simply by multiplying the sizes of the vectors in the intermediate tuples. *What would be the right processor architecture for performing fast arbitrary aggregations on factorized representations in RDBMSs?*

5.2 Long Term Research

I plan to position my future research in the space of ***data systems for machine learning on graphs***. Perhaps the most exciting recent developments in the graph data analytics space is the surge of graph-based learning models. These are machine learning techniques that embed nodes and edges in a graph-structured data to a vector space to perform predictive tasks in this space, e..g., predicting if a user, represented as a node, is likely to buy a product, represented as another node. This technology has led to many highly impactful applications and major scientific breakthroughs *e.g.*, protein folding prediction by DeepMind's AlphaFold and predicting side effects of drugs. Existing applications that use graph embeddings: (i) extract a graph out of a DBMS, e.g., a GDBMS, into raw disk files; (ii) use an in-memory graph learning library, such as Pytorch Geometric [14] or Deep Graph Library [15] to compute embeddings; and (iii) then use a separate (often custom built) system to use the embeddings for a predictive task (e.g., recommendations). As such, these pipelines consist of using three (or more) disintegrated systems. I am interested in developing graph data systems that can better integrate these components. *In particular, how could a better GDBMS-graph learning library integration help scale in-memory graph training? What features can a GDBMS be extended to natively store and process embeddings and perform hybrid "querying" of node/edge embeddings, e.g., to predict non-existing properties and edges between nodes, along with regular queries of existing nodes, edges and their properties?*

References

[1] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *PVLDB*, 2019.

[2] P. Gupta, A. Mhedhbi, and S. Salihoglu, "Columnar storage and list-based processing for graph database management systems," *PVLDB*, 2021.

[3] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," *SIGMOD*, 2017.

[4] A. Mhedhbi, P. Gupta, S. Khaliq, and S. Salihoglu, "A+ indexes: Tunable and space-efficient adjacency lists in graph database management systems," *ICDE*, 2021.

[5] A. Mhedhbi, C. Kankanamge, and S. Salihoglu, "Optimizing one-time and continuous subgraph queries using worst-case optimal joins," *TODS*, 2021.

[6] A. Mhedhbi and S. Salihoglu, "Modern techniques for querying graph-structured relations: Foundations, system implementations, and open challenges," *PVLDB*, 2022.

[7] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *PVLDB*, 2017.

[8] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu, Y. Yao, Q. Yin, W. Yu, K. Zeng, K. Zhao, J. Zhou, D. Zhu, and R. Zhu, "Graphscope: A unified engine for big graph processing," *PVLDB*, 2021.

[9] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," *SIAM J. Comput.*, 2013.

[10] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *JACM*, 2018.

[11] D. Olteanu and J. Závodný, "Size bounds for factorised representations of query results," *TODS*, 2015.

[12] 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," *VLDBJ*, 2020.

[13] A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby, and G. Szárnyas, "LSQB: a large-scale subgraph query benchmark," *GRADES-NDA*, 2021.

[14] "PyTorch Geometric. https://pytorch-geometric.readthedocs.io/en/latest/."

[15] "Deep Graph Library. https://www.dgl.ai/."

Teaching Statement, Amine Mhedhbi

My teaching philosophy is informed by the common characteristics of instructors whose courses I found memorable. When I reflect back on my education, I find that effective teachers were not those that only instructed the technical material in a good way but also those that could raise curiosity, both through their enthusiasm and by giving students a glimpse of the largeness of the field that the taught material is part of.

Over the past eight years, I have had a lot of teaching experience at my current graduate institution as well as my undergraduate Alma mater. At University of Waterloo, I served as a teaching assistant (TA) for 3 terms for introductory programming courses using Racket (200+ students) with two different versions for CS and non-CS students. I was also a TA for 13 terms for multiple database courses (40-200 students) ranging from learning SQL and usage of database management systems (DBMSs) to an introductory database implementation course. At Concordia university during my undergraduate studies, I was twice an instructor for a third year design engineering course (30-40 students), in which I lectured on mobile development and supported student projects through hands-on lab sessions. I have had experiences co-advising undergraduate research assistants and a master student with professor Semih Salihoglu. I was also a guest lecturer at a graduate seminar course on 'Knowledge Graphs'. Finally, I gave a 3-hour teaching tutorial on my Ph.D. research topics at the Very Large Data Base (VLDB) conference in 2022.

I have had prior teaching experience with a variety of class sizes ranging from 40 to 200+ students that come from different technical backgrounds. I have also experienced teaching and interacting with students both in the physical and remote settings. I believe that these experiences make me well prepared to teach classes and excel regardless of size, medium (physical vs. remote), and background or technical level of the students.

In the remainder of this statement, I will outline the set of undergraduate and graduate courses I am qualified to teach and then give my vision of database and data science education.

1 Undergraduate Courses

Regarding the topics that I can teach, I am able to teach introductory programming courses and a variety of undergraduate database courses, both introductory and advanced ones. Although not in my direct research area, I can also comfortably teach other systems courses, such as operating systems, and software engineering (SOEN) ones that are focused on software design, SOEN methodology, and are project based.

My aim for undergraduate courses is to ground them in real world examples and providing depth in terms of theory but also with hands-on programming assignments and projects. To make this concrete, an example of assignments and projects in early introductory database system implementation courses might require changes to a simple prototype DBMS. A more advanced database implementation course would require changes to much more complex systems such as PostgreSQL. I have had prior experiences with creating such projects and assignments in prior teaching roles.

While I was mainly an instructor and TA for CS classes, I can teach undergraduate courses that are targeted for non-CS students, which I have done as a TA at the University of Waterloo. Specifically, I have TAed, *"CS 115 - Introducing to computer programming"*, *"CS 330 - Management Information Systems"*, and *"CS 338 - Computer Applications in Business: Databases"*, all targeted for non-CS students with a range from students in mathematics to linguistics.

Due to my research area and having studied how this course is taught across many campuses, I believe that there is a need for a textbook on database management system (DBMS) implementation. Current textbooks present DBMS architectures that are decades old, and this makes these courses using them less relevant to current day students. My aim is to collect teaching material and the notes that I use to teach and transform this material into a new textbook after my tenure-track years.

2 Graduate Courses

Early on, I am interested in teaching different graduate courses, one on query processing and execution for analytical DBMSs that focuses on new algorithmic development and another on graph data management for different workloads, especially for ML/AI applications. Later on, I am also interested in developing a graduate course on the history of database system implementation. The course would present the evolution of DBMSs, covering covering the early days targeting transactional query processing, then business analytical workloads to the later 'no-size fits all' era where systems target newer workloads, *e.g.*, streaming, leading to re-architecting DBMSs. This would act as a buffet course introducing many database implementation topics. I see the course covering both: i) how research directions changed over time; and ii) the fundamentals of data system implementations through foundational papers.

3 Vision of Database and Data Science Education

My primary focus is teaching the next generation of CS students about data systems technology in terms of use and development. A limitation of exclusively teaching students purely technical material is that they may come to implicitly believe that technology is value neutral and will not see societal problems as problems within their domains. As such, I believe that both undergraduate and graduate courses relating to database and data science education require an interdisciplinary bent, to clarify the subtle but real impact technical choices can have on society. Additionally, I hope that an interdisciplinary perspective will spark an interest in the challenges that exist in other fields such as health, urban planning, and tech policy.

Conversely, students from business and management schools, life sciences, and the social sciences and humanities, *i.e.*, outside of CS, can always benefit from courses relating to database and data science. These courses would cover the topics needed to tackle what many refer to as golden age of data, in which there is a lot of interest in capturing and analyzing very large datasets for quantitative studies and building data-driven products within their fields.

Both of these goals, making CS classes more interdisciplinary as well as providing non-CS students with the necessary database and data science training might necessitate collaboration across departments. I foresee such collaboration would further lead to a cross-disciplinary research environment, due to faculty members from different departments interacting, which grounds CS research and makes it helpful to real-world users.

November 20, 2022

Dear Faculty Recruiting Committee:

I am writing this letter of reference for Amine Mhedhbi in my capacity as his PhD advisor. I want to begin by saying that Amine has *my strongest recommendation* for the Assistant Professor position at your university. As I will articulate throughout this letter, my recommendation is based on two opinions I have formed about Amine throughout six years of advising him: (1) Amine has produced very impactful and high quality work on his thesis topic of graph data management systems and I expect his work's impact to be more visible over time. (2) Amine has grown into a fiercely independent and ambitious researcher with a very practical and systems-oriented research style in his field of data management and processing. I believe this gives him a great potential to grow and makes him a very promising candidate for a junior faculty position. In the remainder of this letter I will first give a brief background on Amine's thesis topic and then summarize his work and accomplishments covering only the work he lead. I refer you to his research statement for the work that I co-authored with Amine but he did not lead or his work in which I was not involved at all.

Amine's research has been on novel query processor and storage techniques to improve the performances of graph database management systems (GDBMSs). GDBMSs are DBMSs that support a graph/network model where users, who are developers, model their application data as a set of nodes, representing entities, and edges, representing relationships between entities. This data model contrasts with the prevalent relational/tabular data model. The major commercial DBMSs in this space include Neo4j and TigerGraph. Despite their data models, the query languages of these systems are very similar to SQL except that joins of records are described through a graph-like syntax, where users "draw" a sub-graph pattern to match in the underlying node and edge records. At the same time, these systems are relational in the sense that: (i) they store node and edge records in node and edge relations; and (ii) they compile their queries to standard relational operators, such as joins, projections, and aggregations of node and edge relations. DBMSs based on graph models have been built throughout history driven by different waves of applications. The latest wave during the NoSQL movement of 2010s was driven by recommendation and fraud detection applications. These applications require finding complex patterns across entities in application data. Example workloads include finding cliques of users/nodes in social or phone networks, or long paths between bank accounts/nodes in financial transaction networks. These workloads are equivalent to complex joins of node and edge relations, where the joins are many-to-many, e.g., each user u's record may join with many other users' records whom u is connected to. As such, these workloads are very challenging to evaluate for DBMSs because they generate very large intermediate relations. Amine's thesis precisely tackles this challenge through a combination of novel join algorithms, intermediate relation representation systems, and storage optimizations. He based his work on a GDBMS called GraphflowDB, whose research and development he lead.

Worst-case Optimal Join Algorithms: Amine's first work in PhD was on tackling complex cyclic workloads[1] such as finding triangles or larger clique-like patterns in an input database. The standard join algorithms in DBMSs, called *binary join plans/algorithms*, iteratively join pairs of relations until all tables in the query are joined. In graph terms, this corresponds to iteratively joining sets of *query edges*. An important theoretical breakthrough in join processing in the last decade was the realization that binary join algorithms can generate provably large intermediate results. For example, when finding triangles, these plans can generate m^2 intermediate open triangles in a graph with m edges, while there can be at most $m^{1.5}$ many triangles on any input graph. This lead to invention of a new class join algorithms called *worst-case optimal join (WCOJ) algorithms*, which perform joins column-by-column, i.e., in graph terminology *query vertex-by-vertex*, joining parts of possibly more than 2 relations at a time. In the context of GDBMSs, these joins are done through multiway intersections of adjacency lists of nodes. I suggested that Amine look into these algorithms and

[1] Although query cyclicity has very formal definitions, cyclic queries on graph data is exactly equivalent to whether the undirected version of the pattern searched in the query has cycles or not.

study how to develop a query processor that uses WCOJs in GraphflowDB.

Amine's first insight into the problem was that the important thing to study was how to order the query vertices in a query when using WCOJ algorithms. This is akin to optimizing the order of relations to join in binary join algorithms. We referred to this problem as the problem of query vertex ordering (QVO). For example, consider the $(a){-}{>}(b){-}{>}(c)$, $(c){-}{>}(a)$ triangle pattern. Should the pattern be matched in a, b, c or c, b, a order? Amine had important insights about the factors to consider when picking an order, such as the directions of adjacency lists and the amount of pairwise intersections that can be cached when intersecting more than 2 adjacency lists. He then realized that he needed a good cardinality estimator to pick good QVOs. Cardinality estimator is the component of a DBMS that predicts the sizes of sub-queries. He designed a specialized cardinality estimator based on keeping statistics on small-size patterns. Soon he had developed: (i) a join optimizer with a new cost metric; (ii) a new cardinality estimator that picked QVOs in a cost-based manner; and (iii) a query processor that demonstrated the benefits of using WCOJs on highly cyclic queries.

Amine was however very unsatisfied with his solution because he realized that there were a lot queries on which binary join algorithms were orders of magnitude more performant than WCOJ plans. He then extended his optimizer to generate plans that used a mix of binary join algorithms and WCOJ-like multiway intersections seamlessly. Although his approach was based on the most traditional query optimization paradigm—a dynamic programming-based (DP) approach of extending optimal plans for sub-queries to larger queries–this, in my opinion, is so far the most practical and general solution in literature to generate plans that use a mix of WCOJ and binary join algorithms. I was quite impressed with Amine's approach in this phase of the project because several earlier papers had proposed novel optimization paradigms to generate plans with WCOJs and binary joins, such as using generalized hypertree decompositions of queries. Despite my advise to study these techniques, Amine vehemently rejected them because he thought they ignored common wisdom of using DP optimization and as such they were not practical. Amine' solution also generated the most general plan space of any system that implemented WCOJs in literature, in the sense that it can generate plans no other system can and subsumes other systems' plans.

Amine and I published this work in a PVLDB 2019 paper titled "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins". This work focused on finding all patterns in a particular snapshot of a graph database (called a one-time query in DBMS literature). I was concurrently advising another Master's student to look into using WCOJ algorithms for *continuous queries*. In this setting, applications register a set of patterns to a GDBMS whose input graph is changing through updates. The problem is to find emergence and deletions of each registered pattern after each update. DBMSs that support continuous queries produce *joint plans* for multiple queries, i.e., a single plan that concurrently evaluates multiple queries. This can be thought of as generating multiple query plans for each query and then overlapping them in a single plan. In our context, our goal was to find QVOs for each query so that after overlapping, the joint plan shares as much computation as possible. Although we had made some progress with my Master's student to understand the core algorithms to use in a continuous query processor, we were unable to optimize our implementation and choice of plans to get good results. Amine was very familiar with my Master's student's work. Upon my student's graduation, Amine offered to take a stab at the problem because he had ideas on sharing more intersections across queries that could deliver promising results. His idea was again very practical and simple. We had so far viewed the problem as maximizing homomorphic sub-queries that are generated by different QVOs of queries. Instead, he observed that we can share *partial intersections* even if two sub-queries are not homomorphic but subsets of intersections that are implied by their QVOs are identical. Indeed, when Amine next implemented a new continuous query processor from scratch, we finally got important performance optimizations that we could publish on. Amine and I then wrote these results (my Master's student had already moved on) as an extension to his PVDLB paper and published in TODS in 2021.

The work on the first WCOJ algorithms were published by Hung Ngo et. al in PODS 2012, which is the primary venue to publish theoretical database research. In 2022, Ngo et al.'s work was given the PODS Test of Time Award. In his acceptance presentation, Hung listed Amine's work as one of first to integrate WCOJ into actual systems. I know many graduate seminars offered by colleagues on GDBMSs assign his PVLDB paper as a primary reading. I expect this work's impact to be more visible over time because WCOJs are critical to evaluate workloads with cyclic patterns in GDBMSs, which are frequent in the primary applications supported by GDBMSs, e.g., in fraud detection and recommendation applications.

A+ Indexes:: Around his third year, Amine started working closely with another Master's student on the adjacency list indexing sub-system of GraphflowDB. This is the sub-system that indexes the edges of each node. Nodes and edges in GDBMSs have labels, e.g., a social network application can model its data as User and City nodes and Knows and $\mathrm{LivesIn}$ edges. We had observed that different systems make ad-hoc decisions in how to design these indexes. For example, in the most basic design all of the edges of a node would be indexed in a single adjacency list. Other designs would partition based on edge labels, i.e., there would be separate Knows and $\mathrm{LivesIn}$ edges. This had important performance implications because the design required different workloads to run different predicates when finding patterns in workloads. As a simple example, if all edges are stored in a single list, a system needs to run additional predicates when finding only Knows edges of a node, which could be avoided if edges are partitioned by edge labels.

Amine quickly realized that the right way to understand existing adjacency list designs were as different but fixed *materialized views*, which is the standard technique in DBMSs to store results of ad-hoc queries in relations. For example, in the design that partitions by edge labels, when a system accesses the Knows adjacency list of node, the system effectively implicitly evaluates the $\mathrm{edgeLabel}$=Knows predicate/query. The first thing Amine realized was that there was no principled reason to limit the design space of indexes to partitioning on edge labels. One could also consider partitioning based on values of other edge properties (e.g., a year property storing in which year a User got to Know another User). Then, Amine and my Master's student studied which materialized views should be supported in adjacency list indexes of GDBMSs and how we can use these views in the system's query plans. Lead by Amine, they designed an end-to-end solution that consisted of a new indexing sub-system which could store edges in nested partitions, support arbitrary predicates, and an optimizer that could choose which views to use in query plans. Importantly this work blended very nicely with Amine's work on WCOJ algorithms, as we were able to show plans that used WCOJ-like multiway intersections of values that came from arbitrary views, that were indexed as adjacency lists. This work was published in the ICDE 2021 Conference in a paper titled "A+ indexes: Tunable and Space-efficient Adjacency Lists in GDBMSs". Although the design Amine lead was not new, it was the right way to understand what adjacency list indexes are in GDBMS: *they are materialized views over the edge records*. This is very characteristic of Amine's research: instead of attempting to produce a novel approach for the sake of producing novel solutions, his instinct is to diligently understand prior literature and building on, modifying and reinterpreting a prior solution, which itself requires producing novel ideas.

Factorized Query Processing:. Amine's third important piece of work focused on workloads that contained acyclic patterns, such as when finding long paths or large star-shaped patterns in graph databases. In many cyclic queries, such patterns are computed frequently as sub-patterns. We had realized in our work on WCOJs that for many queries, including cyclic ones, often the bottleneck was the large intermediate results generated for those acyclic sub-patterns in the query. Unlike WCOJs, which can reduce the number of intermediate results, for acyclic queries, such a reduction is often not possible: it is in the nature of the databases with many-to-many relations that they can contain many acyclic patterns. For example, on a k-regular graph, there are k^n stars for an n-star pattern. I had heard of a new theory on *factorized databases* from the theory community that argued that the results of acyclic sub-queries should be represented in a compressed format as unions of Cartesian products instead of standard flat tuples. For example, consider a node v^* that has k outgoing edges to nodes $\{v_1, ..., v_k\}$. The result of a 2-star pattern $(a){<}{-}(b){-}{>}(c)$ where b matches v^* can be represented as (a=$\{v_1, ..., v_k\}$) × ($b = v^*$) × (c=$\{v_1, ..., v_k\}$), instead of k^2 many flat tuples. I pointed Amine to this literature and said the answer to better handling acyclic intermediate results may lie there.

Amine quickly mastered this literature and understood the prior attempts to represent relations in factorized formats. He was deeply unsatisfied with existing systems work as they were dropping all common wisdom on how to architect query processors. For example, they were all based on full intermediate relation materialization instead of pipelining tuples between operators. Similarly, they dropped vectorization—the standard query processor architecture for analytics-oriented columnar DBMSs—in which operators pass vectors, e.g., 1024, off flat tuples to each other. Instead, prior solutions would pass tries between operators, which required re-writing the entire query processor. He immediately wanted to fill this gap and architect a modern factorized query processor: one that adopts pipelining and vectorization. Amine's solution was a very practical design that was based on two simple insights: (1) most acyclic queries GDMBSs evaluate are long paths and stars; and (2) if we give up the goal of producing arbitrary Cartesian product representations, we can develop a practical processor that gives us the benefits of factorization without major changes to the rest the operators. Amine's design was very elegant: instead of passing a single vector of tuples between operators, as done in traditional analytics-oriented columnar DBMSs, operators would pass multiple vectors of tuples, which would represent the sets in the Cartesian product. His design ensured that operators would continue to work on a single vector of tuples so the operators never had to take Cartesian products. As I told him myself, the meeting when he proposed the design was my favorite meeting of the entire GraphflowDB project. It was an elegant and a very practical design and very easy to explain in writing. This work was published in PVLDB 2021 in a paper titled "Columnar Storage and List-based Processing for GDBMSs". Amine appears as a second author on this paper because I gave the advice to merge his work with another Master's student's thesis work, which was on columnar storage techniques. This is because what Amine did was in fact a novel columnar query processing technique, as it extended vanilla vectorized query processors of columnar DBMSs. Once we decided to write a single paper, we thought that the query processor part of the paper was relatively shorter than the work on the columnar storage designs, which was lead by my Master's student.

Summary: I joined University of Waterloo in 2016 and am affiliated with the Data Systems Group (DSG) there. During my tenure here, DSG has contained between 4-6 different faculty who publish in core data management venues, such as VLDB, SIGMOD, and ICDE conferences. Over that period of time, I have met or observed close to 20 PhD students who completed their theses in these fields. Along with Xu Chu (Georgia Tech), I rank Amine in the top two PhDs from our group in that time frame in terms of their abilities as researchers. Focusing on our students who have taken faculty jobs: I rank both Xu and Amine above Michael Mior (Rochester Institute of Technology) and Chang Ge (University of Minnesota), and much above Ahmed El-Roby (Carleton Unviersity, Canada). Amine and Xu have very different styles, so they are difficult to rank. Xu produces core algorithmic work or focuses on algorithmic optimizations in systems, instead of implementation-heavy systems optimizations. In contrast, Amine does and I believe will continue doing implementation-heavy systems research. Xu is much stronger in his algorithmic skills and is more prolific than Amine but Amine is the clear top systems-oriented researcher to graduate from DSG in the last 6 years.

I next want to clarify a point that may be relevant for interpreting Amine's file. The above work I have outlined have also formed an important part of my tenure case. Although I had set the long term vision for the GraphflowDB project before Amine joined my group, GraphflowDB was not the only such project I had started early in my Waterloo years. I had done the majority of my PhD work on theoretical data management topics and advised several theory students in the beginning of my Waterloo years. I also started several concurrent graph systems projects with other graduate students, such as GraphSurge, GraphWrangler and a failed attempt to develop a distributed GDBMS. In the end, I ended up focusing around GraphflowDB because Amine's work was more productive compared to other systems or theoretical projects I was doing. Therefore, my focus organically followed my most productive graduate student and project.

Amine is an outstanding and very ambitious researcher who has done a very high quality work during his graduate school. He compares favorably to the other students I have seen graduate from DSG. His work has appeared in top data management venues of PVLDB, ICDE, TODS, and SIGMOD. He was given both a Microsoft and a Facebook PhD Fellowships (had to choose one of them and chose Microsoft). He knew from day one in my group what type of researcher he wanted to be: a researcher who works on novel architectures for core components of large-scale data management and processing systems. He was inspired by the long-lived research projects that developed successful open-source systems like Spark from Berkeley, MonetDB from CWI, and C-Store from MIT, each of which became very impactful work that changed how we develop DBMS for different workloads. Amine is completing a thesis that is similar in style to these theses for graph workloads with many-to-many joins. His thesis has already made visible impact in the field: Amine's techniques are integrated into a GDBMS/graph platform at Alibaba and Amine has given several seminars in different research groups, such as Northeastern University, University of Edinburgh, CWI, and a workshop on factorized databases organized by Dan Olteanu, the pioneer of factorized databases. When he forms his group, you can expect that Amine will start similar long-term systems projects of his own, and aim to produce impactful and high quality publications instead of aiming to produce a large quantity of publications. As he articulates in his research statement, he will be pursuing an agenda in the "systems for machine learning" space, studying the challenges that data systems can tackle for graph machine learning workloads. I think this area is ripe with many interesting problems and will give Amine a lot of space to grow.

Amine is further an excellent lecturer and fully prepared to teach courses on data management and more broadly systems. I have seen countless presentations by him, co-presented with him a tutorial in VLDB 2022, and invited him to give a lecture in my graduate seminar on knowledge graphs. Finally, Amine has a very warm, positive, and a very optimistic personality. I have very fond memories of watching him do freestyle football (see his video links at the bottom of his website) or being surprised to find him performing on stage in an improv comedy theater I happened to drop by in Waterloo. I have no doubt he will be known as a fun-loving and positive friend to his colleagues throughout his career.

Please do not hesitate to contact me if you have further questions on my opinions about Amine.

Sincerely,

Semih Salihoğlu
Associate Professor and David R. Cheriton Faculty Fellow
David Cheriton School of Computer Science
University of Waterloo, Canada

[illegible]

UNIVERSITY OF WATERLOO

FACULTY OF MATHEMATICS | David R. Cheriton School of Computer Science
519-888-4567, ext. 38693 | fax 519-885-1208
cs.uwaterloo.ca

22 November 2022

Dear Faculty Hiring Committee,

I am writing to strongly support Amine Mhedhbi's application to your institution for a faculty position. Amine is a very good young researcher who has done some very interesting work and has many ideas that should lead to more high-quality results.

I have followed Amine's work since he started his PhD work at Waterloo under the supervision of Semih Salihoglu. I am a member of his PhD examining committee, so I know most of his work well.

Early in his PhD, Amine and another PhD student were the lead authors of a survey paper on graph use and challenges; I was also a co-author. The objective of the survey was to get a full understanding of the uses of graph databases and compare them with the research focus that we find in published literature. This involved designing a survey, recruiting responders and analyzing the results. The research results were published in Proceedings of VLDB Endowment (PVDB) in 2018, and the paper was selected as the best paper of the VLDB 2018 Conference. We then expanded the paper with interviews by a number of large companies and a fuller study of literature. The extended paper appeared in VLDB Journal. The findings of this survey were instrumental in informing Amine's work.

Amine's PhD work involves processing of large-scale graph data. Graph data has become important as part of the "big data," because in many modern applications, the relationships among entities have become as important as entities themselves, and graphs capture these relationships naturally. Work in graph data management falls into two categories based on their workloads. The first workload class consists of analytical queries (or analytical workloads) whose evaluation typically requires processing each vertex in the graph over multiple iterations until a fixpoint is reached. Examples of analytical workloads include PageRank computation, clustering, finding motifs (patterns) in graphs, and graph machine learning. The second class of workloads is called online queries (or online workloads), which are not iterative and usually require access to a portion of the graph and whose execution can be assisted by properly designed auxiliary data structures. Examples of online workloads are reachability queries, and single-source shortest path. Amine's work mostly falls into the first category – graph analytics. Many of the graph analytics algorithms are difficult to scale to the large graph datasets that are in use today. In many cases, the use of theoretically appealing techniques requires careful redesign and novel architecting to be practically useful. This requires deep algorithmic considerations, systems architecting, and very careful and extensive analysis. Amine's research does this very well and demonstrably achieves superior results. His work demonstrates a deep understanding of the theoretical underpinnings, algorithmic development, (prototype) system building and very careful experimental analysis.

Amine's research results have been integrated into GraphflowDB, which is a system that focuses on efficient subgraph computation over dynamic graphs. Although the project involves other students, Amine is the primary graduate student. The project involves considerable algorithmic and systems research and has led to a prototype that was demonstrated at the 2017 SIGMOD Conference. GraphflowDB incorporates three of Amine's contributions that I summarize below.

One of Amine's contributions is efficient joins on partial results of graph match queries. Existing techniques typically follow binary join whose intermediate results can be very large, and therefore, may not be suitable for large graphs. Amine investigated how to architect and build a technique, called worst-case optimal join (WOJ), into a graph database system. This required developing algorithms around WOJ and how to architect the system to fully utilize this technique. He went further and demonstrated that worst-case optimal join is not always the preferred technique, which was contrary to what we thought at the time. His PVLDB 2019 paper demonstrated that WOJ performs well under certain circumstances but not others. Amine studied the problem of expanding the query optimizer search space by considering both binary and WOJ, giving the optimizer more options in finding an efficient execution plan. The paper demonstrates both the feasibility and the usefulness of this approach. This work was subsequently extended by developing a cost-based optimizer for graph DBMSs in support of both one-time and continuous queries. The optimizer can take a set of continuous queries, decompose them, find common decompositions across the set of these queries and pick an efficient execution plan. This extended paper was published in ACM Transactions on Database Systems (TODS) in 2021.

In a second line of research, Amine investigated storage structures for graph DBMSs, which is important for the GraphflowDB system that is disk-based. He worked on two problems that are important for storage system design. One line of work involves designing a columnar storage structure. This is not a direct adaptation of columnar storage in relational systems; he developed a unique list-based query processing technique and data structures for optimizing the edge and vertex lists. This work was published in PVLDB in 2021. The second line of work he has conducted involves improvements to the well-known adjacency list indexes. This data structure is efficient and works well, but its construction is based on a given workload composition and the structure is static. His research has led to the development of a tunable adjacency list index structure that allows adaptation to workload changes. This work is published in ICDE Conference in 2021.

The third line of work he has done is along factorized query processing. He published his initial work in this area in his PVLDB 2021 paper I mentioned above. I know that part well, but there is more to his work in that area, and I do not fully know that work – I have not yet seen his dissertation where the full scope is described. I find the approach very interesting and novel, but I will leave deeper comments on that work to his supervisor.

Amine's research results to date demonstrate his ability to dig deep into problems to fully understand and to find efficient ways of solving them. The different topics he tackled show that he can maneuver in his research space easily. He finds unique problems and develops novel solutions – even when he adapts previously developed techniques. He has been very well trained in finding the right problems and figuring out how to attack them.

Amine is an excellent graduate student with a very good background. He has a very good research sense, and he is articulate in discussing his research ideas. He has done very good work in his PhD research and has built a very respectable publication list. He is a serious researcher who does not mind doing the necessary work (reading, implementing, experimenting) to find a solution. I strongly and enthusiastically support his application for a faculty position.

Sincerely,

[illegible]

M. Tamer Özsu
University Professor
tamer.ozsu@uwaterloo.ca

M. Tamer Özsu is a University Professor in the David R. Cheriton School of Computer Science at the University of Waterloo. He also holds a Distinguished Visiting Professorship at Tsinghua University, China. He is the Founding Director of Waterloo-Huawei Joint Innovation Laboratory since 2018; he was the Director of the Cheriton School from January 2007 to June 2010, and the Associate Dean of Research of the Faculty of Mathematics from January 2014 to June 2016. Previously he was with the Department of Computing Science of the University of Alberta (1984–2000) where he served as Acting Department Chair in 1994–95. He has held visiting scientist/professor positions in US, France, Germany, Singapore, Italy, and Switzerland during his sabbatical leaves. His PhD degree is from the Ohio State University (1983). His research is on data management focusing on large-scale data distribution and management of non-traditional data. He is the co-author (with Patrick Valduriez) of the book *Principles of Distributed Database Systems*, which is considered the classic textbook on the topic and is now in its fourth edition with translations into Chinese and Portuguese. He has also edited, with Ling Liu, the *Encyclopedia of Database Systems*, whose second edition was released in 2019. He is a Fellow of the Royal Society of Canada, American Association for the Advancement of Science (AAAS), Association for Computing Machinery (ACM), The Institute of Electrical and Electronics Engineers (IEEE), Asia-Pacific Artificial Intelligence Association (AAIA), and Fellow of the Balsille School of International Affairs (BSIA). He is an elected member of the Science Academy, Turkey, and a member of Sigma Xi. He is the current and past holder of Cheriton Faculty Fellowship (2013–2016 and 2018–2024), a University Research Chair (2004–2011), and a Faculty Research Fellowship (2000–2003) at the University of Waterloo, and a McCalla Research Professorship (1993–1994) at the University of Alberta. He is the recipient of the IEEE Innovation in Societal Infrastructure Award (2022), CS-Can/Info-Can Lifetime Achievement Award (2018), ACM SIGMOD Test-of-Time Award (2015), the ACM SIGMOD Contributions Award (2006), The Ohio State University College of Engineering Distinguished Alumnus Award (2008), and multiple Outstanding Performance Awards at the University of Waterloo (2004, 2009, 2014, 2019). He was the Founding Editor-in-Chief of ACM Books (2013—2019) and Synthesis Lectures in Data Management (2009–2013). He has served as the Program Chair of all three major database conferences: VLDB (2004), ICDE (2007), and SIGMOD (2014), and has served on many Technical Program Committees. He is on the editorial boards of three journals, and two book series. He chairs the Advisory /Steering Committee of Hong Kong Polytechnic University's Research Centre on Data Science and Artificial Intelligence and serves on the Technical Advisory Committee of National Engineering Laboratory for Big Data Software of Tsinghua University, on the Advisory Boards of Hong Kong University of Science and Technology School of Engineering, and Hong Kong University of Science and Technology Big Data Institute. He previously served as Chair of ACM SIGMOD (2001–2005), Member and Chair of Natural Sciences and Engineering Research Council (NSERC) of Canada's Computer Science Grant Selection Committee (1991–1994), Member of Computer Research Association (CRA) Board (2009–2013), EIC of VLDB Journal and on the Scientific Advisory Board of National Institute of Informatics of Japan (2011–2017), Technical Expert Advisory Committee of City University of Hong Kong Multimedia Software Engineering Research Center (2014–2019), ACM Publications Board (2002–2017), the Board of VLDB Endowment (1996–2002).

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Tel 425 882 8080
Fax 425 936 7329
http://www.microsoft.com/

Microsoft

November 28, 2022

Dear Faculty Recruiting Committee,

This letter is to recommend Amine Mhedhbi for a position in your department. I have known Amine since last spring when I hired him as an intern. He worked for me from June through September this year on a transaction processing project, and we have continued to collaborate since he returned to Waterloo.

When considering Amine for a position, I was impressed by his publication record, recommendations, and interview. But I hesitated to hire him because his background wasn't an ideal fit for my project. Since he volunteered to do a lot of reading about transaction processing before starting his internship, I decided to take a risk and hire him. He delivered far beyond my expectations. He arrived with a better understanding of transactions in a data sharing system than many graduate students who specialize in transaction processing.

Our project is on distributed lock management in a data-sharing database system. In this type of system, the database system runs on multiple nodes of a data center network, and the instances share a single copy of the database on network-attached storage. Examples are Azure SQL Hyperscale, AWS Aurora, Google AlloyDB, Oracle RAC, and IBM DB2 Data Sharing. The project involves developing a high-performance lock manager, locking protocols to synchronize distributed caches, and a simulator to evaluate those protocols. Amine dove into detailed design and coding almost immediately and got the job done. We're now in the evaluation phase and continue to innovate with new protocols.

The project is all about performance. Amine persevered in diagnosing and fixing several subtle sources of inefficiency we encountered along the way. He demonstrated an excellent understanding of CPU cache architecture, multicore, and hardware synchronization. And he contributed several ideas to optimize the cache synchronization protocol, which is the core of our research.

Amine gave an excellent end-of-internship talk to our group. Several researchers commented that they found the work very interesting. I'm confident we'll publish a top-tier conference paper about it. Beyond that I cannot share details since it is unpublished work.

Amine writes beautiful, well-documented code. Perhaps this isn't the most valuable skill for a professor. However, it reflects the deep thinking he applies to his work and his desire for elegant solutions.

Amine learns quickly, thinks deeply, is articulate, and works very hard. During his internship we talked daily. He was a great foil to brainstorm and critically analyze new ideas. He's always cheerful and a pleasure to work with. I recommend him very strongly and without any reservations for a position in your department.

Sincerely,

[illegible]

Philip A. Bernstein
Distinguished Scientist, Microsoft Research
Affiliate Professor, University of Washington
Member, National Academy of Engineering

Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins

AMINE MHEDHBI, CHATHURA KANKANAMGE, and SEMIH SALIHOGLU,
University of Waterloo

We study the problem of optimizing one-time and continuous subgraph queries using the new worst-case optimal join plans. Worst-case optimal plans evaluate queries by matching one query vertex at a time using multiway intersections. The core problem in optimizing worst-case optimal plans is to pick an ordering of the query vertices to match. We make two main contributions:

1. A cost-based dynamic programming optimizer for one-time queries that (i) picks efficient query vertex orderings for worst-case optimal plans and (ii) generates *hybrid plans* that mix traditional binary joins with worst-case optimal style multiway intersections. In addition to our optimizer, we describe an *adaptive technique* that changes the query vertex orderings of the worst-case optimal subplans during query execution for more efficient query evaluation. The plan space of our one-time optimizer contains plans that are not in the plan spaces based on tree decompositions from prior work.
2. A cost-based greedy optimizer for continuous queries that builds on the delta subgraph query framework. Given a set of continuous queries, our optimizer decomposes these queries into multiple delta subgraph queries, picks a plan for each delta query, and generates a single combined plan that evaluates all of the queries. Our combined plans share computations across operators of the plans for the delta queries if the operators perform the same intersections. To increase the amount of computation shared, we describe an additional optimization that shares partial intersections across operators.

Our optimizers use a new cost metric for worst-case optimal plans called *intersection-cost*. When generating hybrid plans, our dynamic programming optimizer for one-time queries combines intersection-cost with the cost of binary joins. We demonstrate the effectiveness of our plans, adaptive technique, and partial intersection sharing optimization through extensive experiments. Our optimizers are integrated into GraphflowDB.

CCS Concepts: • **Information Systems → DBMS engine architectures**; **Query Planning**; **Join Algorithms**;

Additional Key Words and Phrases: Subgraph queries, worst-case optimal joins, generic join

ACM Reference format:
Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. *ACM Trans. Database Syst.* 46, 2, Article 6 (May 2021), 45 pages.
https://doi.org/10.1145/3446980

Authors' addresses: A. Mhedhbi, C. Kankanamge, and S. Salihoglu, University of Waterloo, emails: {amine.mhedhbi, c2kankan, semih.salihoglu}@uwaterloo.ca.

0362-5915/2021/05-ART6 $15.00
https://doi.org/10.1145/3446980

1 INTRODUCTION

Subgraph queries, which find instances of a query subgraph $Q(V_Q, E_Q)$ in an input graph $G(V, E)$, are a fundamental class of queries supported by graph databases. We refer to finding subgraphs in a static graph as *one-time subgraph queries* and monitoring subgraphs in a dynamic graph as *continuous subgraph queries*. Subgraph queries appear in many applications where graph patterns reveal valuable information [61]. For example, Twitter searches for diamonds in their follower network for recommendations [23], cliquelike structures in social networks indicate communities [48], and cyclic patterns in transaction networks indicate fraudulent activities [12, 44].

As observed in prior work [2, 6], a subgraph query Q is equivalent to a multiway self-join query that contains one $E(a_i, a_j)$ (for **E**dge) relation for each $a_i \rightarrow a_j \in E_Q$. The top box in Figure 1(a) shows an example query, which we refer to as *diamond-X*. This query can be represented as:

$$Q_{DX} = E_1 \bowtie E_2 \bowtie E_3 \bowtie E_4 \bowtie E_5 \text{ where}$$
$$E_1(a_1, a_2), E_2(a_1, a_3), E_3(a_2, a_3), E_4(a_2, a_4), \text{and } E_5(a_3, a_4) \text{ are copies of } E(a_i, a_j).$$

We study evaluating a general class of subgraph queries where V_Q and E_Q can have labels. For labelled queries, the edge table corresponding to the query edge $a_i \rightarrow a_j$ contains only the edges in G that are consistent with the labels on a_i, a_j, and $a_i \rightarrow a_j$. Subgraph queries are evaluated with two main approaches:

- *Query-edge(s)-at-a-time* approach executes a sequence of binary joins to evaluate Q. Each binary join effectively matches a larger subset of the query edges of Q in G until Q is matched.
- *Query-vertex-at-a-time* approach picks a *query vertex ordering* σ of V_Q and matches Q one query vertex at a time according to σ. Query vertex matching uses a multiway join operator that performs multiway intersections. This is the computation performed by the recent worst-case optimal join algorithms [49, 50, 66]. In graph terms, this computation intersects one or more adjacency lists of vertices to extend partial matches by one query vertex.

We refer to plans with only binary joins as *BJ* plans, with only intersections as *WCO* (for **w**orst-**c**ase **o**ptimal) plans, and with both operations as *hybrid* plans. Figure 1(a), (b), and (c) show an example of each plan for the diamond-X query.

Recent theoretical results [8, 50] showed that BJ plans can be suboptimal on cyclic queries. Specifically, the size of the intermediate results of BJ plans, on cyclic queries, can be asymptotically larger than the maximum possible final output size of the query. This maximum output size is now known as a query's *AGM bound*. Given the sizes of a set of relations $|R_1|, \ldots, |R_n|$ and a join query Q on these relations, the AGM bound is the maximum output size of Q under all possible database instances with these relation sizes. These results also showed that WCO plans correct for this sub-optimality. However, this theory has two shortcomings. First, the theory does not give advice as to *how to pick a good* ***query vertex ordering (QVO)*** for WCO plans. Specifically, the theory demonstrates any query vertex ordering achieves worst-case optimality. In practice however, different query vertex orderings have very different performances. Second, the theory does not capture plans with binary joins, which have been shown to be efficient on many queries by decades-long research in databases as well as several recent work in the context of subgraph queries [2, 38].

In this work, we study how to generate efficient plans that use WCO join-style multiway intersections and use them to evaluate one-time and continuous subgraph queries in graph database management systems. We describe two cost-based optimizers that we developed for GraphflowDB: (i) a dynamic programming optimizer that generates efficient plans for one-time subgraph queries using a mix of worst-case optimal join-style multiway intersections and binary joins and (ii) a greedy optimizer that generates WCO plans for continuous queries that share computation across

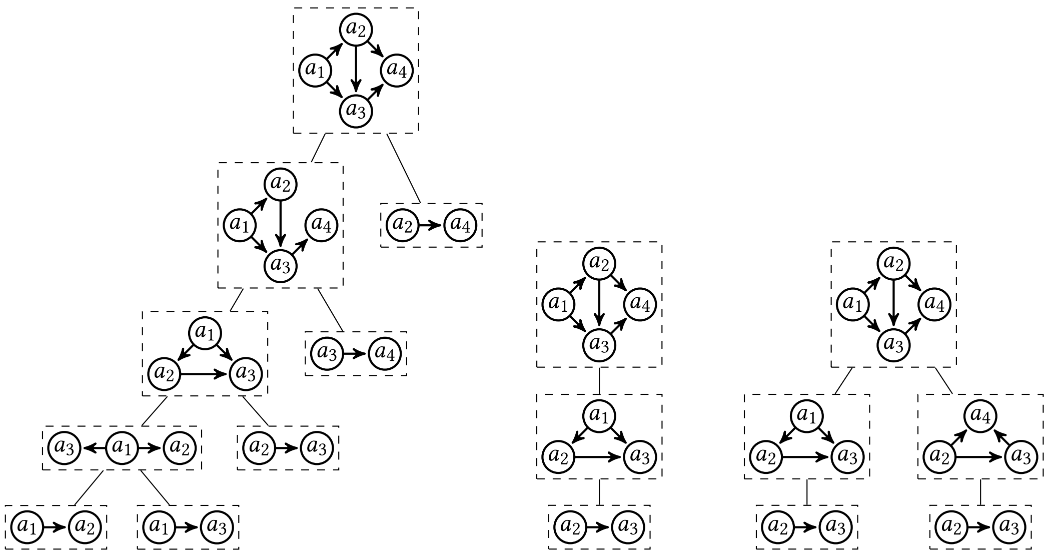

(a) Binary join plan. (b) WCO plan. (c) Hybrid plan.

Fig. 1. Example plans. The subgraph on the top box of each plan is the actual query.

Table 1. Abbreviations Used Throughout the Paper

Abbrv.	**Explanation**	**Abbrv.**	**Explanation**
BJ	Binary Join	E/I	Extend/Intersect
CP	Combined Plan	GHD	Generalized Hypertree Decompositions
DSQ	Delta Subgraph Query	QVO	Query Vertex Ordering
EH	EmptyHeaded	WCO	Worst-case Optimal

queries. Our cost metric for WCO plans captures the various runtime effects of query vertex orderings we have identified. Our cost-based optimizers' plans are significantly more efficient than the plans generated by prior solutions using WCO plans that are either based on heuristics or have limited plan spaces. The optimizers of both native graph databases, such as Neo4j [43], as well as those that are developed on top of RDBMSs, such as SAP's graph database [60], are often cost-based. As such, our work gives insights into how to integrate the new worst-case optimal join algorithms into cost-based optimizers of existing systems.

In the remainder of this section, we give an overview of existing solutions for one-time and continuous subgraph queries, our approach, and contributions. Table 1 summarizes the abbreviations used throughout the article.

1.1 Single One-time Subgraph Query Optimization

1.1.1 Existing Approaches. Perhaps the most common approach adopted by graph databases (e.g., Neo4j), RDBMSs, and RDF systems [47, 70] is to evaluate subgraph queries with BJ plans. As observed in prior work [49], BJ plans are inefficient in cliquelike queries, such as cliques. Several prior solutions, such as BiGJoin [6], and the LogicBlox system have studied evaluating queries with only WCO plans, which, as we demonstrate in this article, are not efficient for large cycle queries. In addition, these solutions either use simple heuristics to select query vertex orderings or arbitrarily select them.

Table 2. Comparisons against Solutions for One-time Queries Using WCO Joins

	QVO	Binary Joins?
BiGJoin	Arbitrarily	No
LogicBlox	Heuristics or Cost-based[1]	No
EmptyHeaded	Arbitrarily	Cost-based: depends on Q
CTJ	Heuristic + Cost-Based (uses caching)	No
GraphflowDB	Cost-based & Adaptive	Cost-based: depends on Q and G

The EmptyHeaded system [2], which is the closest to our work, is the only system we are aware of that mixes worst-case optimal joins with binary joins. **EmptyHeaded** (EH) plans are **generalized hypertree decompositions (GHDs)** of the input query Q. A GHD is effectively a join tree T of Q, where each node of T contains a sub-query of Q. EmptyHeaded evaluates each sub-query using a WCO plan, i.e., using only multiway intersections, and then uses a sequence of binary joins to join the results of these sub-queries. As a cost metric, EmptyHeaded uses the *generalized hypertree widths* of GHDs and picks a minimum-width GHD. This approach has three shortcomings: (i) If the GHD contains a single sub-query, then EmptyHeaded arbitrarily picks the query vertex ordering for that query; otherwise, it picks the orderings for the sub-queries using a simple heuristic; (ii) the width cost metric depends only the input query Q, so when running Q on different graphs, EmptyHeaded always picks the same plan; and (iii) the GHD plan space does not allow plans that can perform multiway intersections after binary joins. As we demonstrate, there are efficient plans for some queries that *seamlessly mix binary joins and intersections* and do not correspond to any GHD-based plan of EmptyHeaded.

Cache Trie Join (CTJ) [28] is another system using a WCO join algorithm. An important advantage of WCO join algorithms is their small memory footprints. For example, when executed in a purely pipelined fashion, such algorithms do not require memory to keep large intermediate results. CTJ observes that by keeping a cache of certain intermediate results and reusing these results, the performance of WCO join algorithms can be improved.

1.1.2 Our Contributions. Table 2 summarizes how our approach compares against prior solutions. Our first main contribution is a dynamic programming optimizer that generates plans with both binary joins and an EXTEND/INTERSECT operator that extends partial matches with one query vertex. Let Q contain m query vertices. Our optimizer enumerates plans for evaluating each k-vertex sub-query Q_k of Q, for k=2, ..., m, with two alternatives: (i) a binary join of two smaller sub-queries Q_{c1} and Q_{c2} or (ii) by extending a sub-query $Q_{k\text{-}1}$ by one query vertex with an intersection. This generates all possible WCO plans for the query as well as a large space of hybrid plans that are not in EmptyHeaded's plan space. Figure 2 shows an example hybrid plan for the 6-cycle query that is not in EmptyHeaded's plan space.

For ranking WCO plans, our optimizer uses a new cost metric called *intersection cost* (i-cost). I-cost represents the amount of intersection work that a plan P will do using information about the sizes of the adjacency lists that will be intersected throughout P. For ranking hybrid plans, we combine i-cost with the cost of binary joins. Our cost metrics account for the properties of the input graph, such as the distributions of the forward and backward adjacency lists sizes and the number of matches of different subgraphs that will be computed as part of a plan. Unlike EmptyHeaded, this allows our optimizer to pick different plans for the same query on different input graphs. Our optimizer uses a *subgraph catalogue* to estimate i-cost, the cost of binary joins, and the number of partial matches a plan will generate. The catalogue contains information about: (i) the adjacency list size distributions of input graphs; and (ii) selectivity of different intersections on small subgraphs.

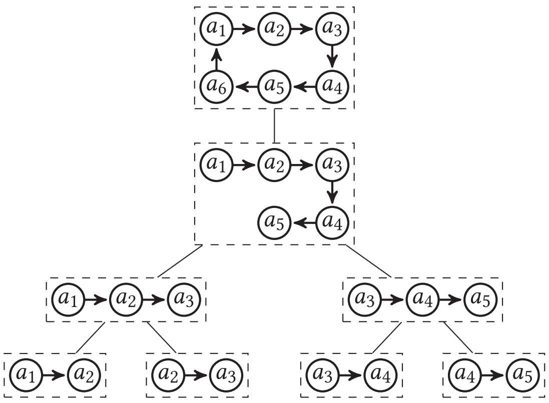


Fig. 2. Example plan not in EmptyHeaded's GHD-based plan space. Top box is the actual query.

Our second contribution is an *adaptive technique* for picking the query vertex orderings of WCO parts of plans during query execution. Consider a WCO part of a plan that extends matches of sub-query Q_i into a larger sub-query Q_k. Suppose there are r possible query vertex orderings, $\sigma_1, \ldots, \sigma_r$, to perform these extensions. Our optimizer tries to pick the ordering σ^* with the lowest cumulative i-cost when extending all partial matches of Q_i in G. However, for any specific match t of Q_i, there may be another σ_j that is more efficient than σ^*. Our adaptive executor re-evaluates the cost of each σ_j for t based on the actual sizes of the adjacency lists of the vertices in t, and picks a new ordering.

We incorporate our optimizer into GraphflowDB and evaluate it across a large class of subgraph queries and input graphs. We show that our optimizer is able to pick close to optimal plans across a large suite of queries and our plans, including some plans that are not in EmptyHeaded's plan space, are up to 68× more efficient than EmptyHeaded's plans. We show that adaptively picking query vertex orderings improves the runtime of some plans by up to 4.3×, in some queries improving the runtime of every plan and makes our optimizer more robust against picking bad orderings.

1.2 Multiple Continuous Subgraph Queries Optimization

Continuous subgraph query evaluation is the problem of detecting the emergence and deletions of a set of (often a small number of) queries that are registered in a system, as the system gets updates to the input graph it manages. Specifically, the problem is to produce a set of newly added and deleted matches of a query after each update to the graph, as a set of tuples with + and - tags, respectively. In this work, we consider the updates to be only edge insertions and deletions. Traditionally, this functionality is the core of triggers in active database management systems [29, 68].

1.2.1 Existing Approaches. Prior work on continuous subgraph queries has two main shortcomings: (i) They are either designed for a single query instead of multiple queries [6, 14, 30], so do not benefit from optimization possibilities across queries and/or (ii) require large auxiliary data structures [14, 30, 34, 55, 58]. We build on the *Delta Generic Join* **incremental view maintenance (IVM)** algorithm from Reference [6]. Delta Generic Join is an IVM algorithm based on an algebraic IVM technique called *delta join query decompositions* [11] of queries. Using graph terminology, we refer to these queries as *delta subgraph queries*. Figure 3 shows the five delta subgraph queries of the diamond-X query. Suppose a set E_δ of updates arrive at G. Let E_o be the **o**ld edges of G and E_n the **n**ew edges of G after the update, i.e., $E_n = E_o \cup E_\delta$. Each query edge of a delta subgraph query is labelled with δ, o, or n, indicating, upon an update to G, whether the edge should match an edge

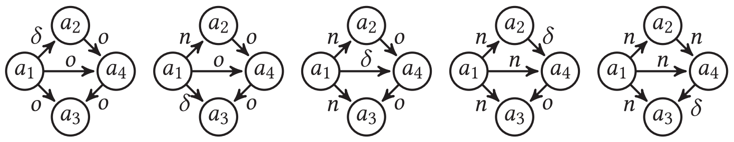


Fig. 3. Example delta subgraph queries for the diamond-X continuous subgraph query.

Table 3. Comparisons against Solutions for Continuous Queries Using WCO Joins

	QVO	Multi-Query Optimization?	Auxiliary Data Structures?
DeltaBiGJoin [6]	Arbitrary	No	No
GraphflowDB$_{old}$ [29]	Heuristics	No	No
TurboFlux [30]	Cost-based	No	Yes
GraphflowDB	Cost-based	Yes	No

in E_δ, E_o, or E_n, respectively. Delta Generic Join evaluates each delta subgraph query using a WCO plan. References [6] and [29] demonstrated the runtime, memory, and theoretical benefits of this approach. For example, one advantage of the delta subgraph query framework is that it does not require auxiliary data structures and that under insertion-only workloads, the cumulative computation performed by Delta Generic Join is worst-case optimal [6]. However, these works focused on the case of evaluating a single query and assumed the query vertex orderings were given or picked them using simple heuristics.

1.2.2 Our Contributions. Table 3 summarizes how our approach compares against prior solutions. Our contribution is a greedy optimizer for multiple continuous subgraph queries that builds upon the delta subgraph query framework. Our optimizer takes as input the delta subgraph query decompositions of a set of queries $\bar{Q}$ and outputs a low i-cost *combined plan* that cumulatively evaluates all of the delta subgraph queries. We first prove that unlike one-time subgraph queries, the optimization problem of finding the lowest i-cost combined plan is NP-complete. As a result, instead of a dynamic programming optimizer, we describe a greedy optimization algorithm that picks a plan, i.e., a query vertex ordering for each delta subgraph query, and generates a combined plan that shares common operators across the plans of delta subgraph queries. The sharing opportunity arises when delta subgraph queries share isomorphic components. An important observation we make is that in absence of perfect symmetry between delta subgraph queries, it is not possible to share computations at the last operators of each plan, that is often where the majority of work is done in the plans. We describe an optimization we call *partial intersection sharing* that allows partial computation sharing in the operators to increase the amount of computation sharing across plans. We show that on small sets of queries, our optimizer is able to find close to optimal plans in terms of wall-clock time in our experimental analysis. On larger queries, we demonstrate that our optimizer can generate combined plans that are up to 3.51× more efficient than optimizing and evaluating each delta subgraph query separately. For completeness, for single continuous subgraph queries, we also provide comparisons against the most efficient continuous subgraph query algorithm we are aware of called TurboFlux [30].

1.3 Outline

Section 2 provides necessary background. Section 3 and 4.2 describe, respectively, our one-time and continuous subgraph query optimizers. Section 5 provides several details on the implementation of our optimizers and GraphflowDB. Section 6 provides extensive experiments studying the performances of our one-time and continuous plans. Finally, Sections 7 and 8 cover related work and conclude, respectively.

2 PRELIMINARIES

We assume a subgraph query $Q(V_Q, E_Q)$ is directed, connected, and has m query vertices $a_1, \ldots, a_m$ and n query edges. To indicate the directions of query edges clearly, we use the $a_i \rightarrow a_j$ and $a_i \leftarrow a_j$ notation. We assume that all of the vertices and edges in Q have labels on them, which we indicate with $l(a_i)$, and $l(a_i \rightarrow a_j)$, respectively. Similar notations are used for the directed edges in the input graph $G(V, E)$. Unlabelled queries can be thought of as labelled queries on a version of G with a single edge and single vertex label. The outgoing and incoming neighbours of each $v \in V$ are indexed in forward and backward adjacency lists and sorted by their IDs, which allows for fast intersections.

2.1 Generic Join: A WCO Join Algorithm

Generic Join [49] is a *WCO* join algorithm that evaluates queries one attribute at a time. We describe the algorithm in graph terms; Reference [49] gives an equivalent relational description. In graph terms, the algorithm evaluates queries one query vertex at a time with two main steps:

- **Query Vertex Ordering (QVO):** Generic Join first picks an order σ of query vertices to match. For simplicity, we assume $\sigma = a_1 \ldots a_m$. The projection of a query $Q(V, E)$ on a set of vertices $X \subseteq V$, denoted by $\Pi_X Q$, is a query $Q_x(X, E_X)$ such that for any pair a_i, $a_j \in X$, $\mathrm{a}_i \rightarrow \mathrm{a}_j \in E_x$ if only if $a_i \rightarrow a_j \in E$. We assume the projection of Q onto any prefix of k query vertices in σ for $k = 1, \ldots, m$ to be connected.[2]
- **Iterative Partial Match Extensions:** Generic Join iteratively computes matches for $Q_1, \ldots, Q_m$, where $Q_k = \Pi_{\{a_1,\ldots,a_k\}} Q$ is a subquery that consists of Q's projection on the first k query vertices in σ: $a_1 \ldots a_k$. Each iteration k produces a set of k-matches for Q_k, where a k-match is a tuple t of size k and $t[i]$, the ith element in t, is the vertex in G matching a_i in Q_k. The first iteration is produced by matching all vertices in G to a_1. To compute Q_k for $k > 1$, for each (k-1)-match t of $Q_{k\text{-}1}$, Generic Join performs the following computation. First, the algorithm takes the forward adjacency list of $t[i]$ for each $a_i \rightarrow a_k \in E_Q$ and the backward adjacency list of $t[i]$ for each $a_i \leftarrow a_k \in E_Q$, where $i \leq k$-1 and intersects these lists. The result of the intersection is the set S=$\{s_1, \ldots, s_\ell\}$ of possible vertex matches for a_k. Then, for each $s_i \in S$, one output k-match $(t[1], \ldots, t[k-1], s_i)$ is produced by appending s_i to t. If $S = \{\}$, then no output tuples are produced.
  Consider for example, the diamond-X query $Q_{DX}(V_{DX}, E_{DX})$ from Section 1 with a QVO $\sigma = a_1 a_2 a_3 a_4$. The fourth iteration takes as input the set of 3-matches for Q_3=$\Pi_{\{a_1,a_2,a_3\}} Q_{DX}$ and produces a set of 4-matches for Q_{DX}. Let $t = (v_1, v_4, v_5)$ be a 3-match, where v_1, v_4, and v_5 match a_1, a_2, and a_3, respectively. To compute the set S, i.e., vertex matches for a_4, Generic Join intersects the forward adjacency lists of v_4 (matching a_2) and v_5 (matching a_3). Note that Generic Join uses the forward adjacency lists, because a_2 and a_3 are already matched in the pattern and both a_2 and a_3 have forward edges to a_4 in the query, i.e., $a_2 \rightarrow a_4 \in E_{DX}$ and $a_3 \rightarrow a_4 \in E_{DX}$. Assume $S = \{v_3, v_{11}\}$ then the set of output tuples is $\{(v_1, v_4, v_5, v_3), (v_1, v_4, v_5, v_{11})\}$.

[2]Otherwise, Generic Join would need to compute expensive Cartesian products to produce intermediate results that match these prefix k query vertices.

2.2 Delta Generic Join: A WCO IVM Algorithm

Recall from Section 1.2 that evaluating a continuous subgraph query Q is the problem of producing a set of newly added and deleted matches of Q after each batch of updates to a dynamic graph. We consider only edge insertions and deletions as updates and assume that the newly added and deleted tuples should be produced as a set of tuples with + and - tags, respectively, after each batch. In graph terms, a continuous subgraph query Q is equivalent to the IVM of the join query that is equivalent to Q, that produces the changes in the output of Q after each update. We adopt and optimize the Delta Generic Join framework [6] as an IVM algorithm to evaluate continuous subgraph queries. Let E_δ be a set of updates, E_o be the `old` edges in G before E_δ, and E_n the `new` edges, i.e., $E_n = E_o \cup E_\delta$. We assume added and deleted edges in E_δ are identified by +/− labels, respectively. Emerged and deleted outputs are identified similarly. We will use $a_i \xrightarrow{\delta/\text{o}/\text{n}} a_j$ notation to refer to the query edges that should match edges in E_δ, E_o, or E_n. Delta Generic Join uses an algebraic IVM technique called delta join query decomposition of queries [11], which decomposes Q, of n query edges, into n **delta subgraph queries (DSQ)s**, and upon an update evaluates each delta subgraph query and unions their results to find the emerged and deleted instances of Q:

$$DSQ_1 = a_{11} \xrightarrow{\delta} a_{12}, a_{21} \xrightarrow{o} a_{22}, a_{31} \xrightarrow{o} a_{32}, \ldots, a_{n1} \xrightarrow{o} a_{n2}$$

$$DSQ_2 = a_{11} \xrightarrow{n} a_{12}, a_{21} \xrightarrow{\delta} a_{22}, a_{31} \xrightarrow{o} a_{32}, \ldots, a_{n1} \xrightarrow{o} a_{n2}$$

$$\ldots$$

$$DSQ_n = a_{11} \xrightarrow{n} a_{12}, a_{21} \xrightarrow{n} a_{22}, a_{31} \xrightarrow{n} a_{32}, \ldots, a_{n1} \xrightarrow{\delta} a_{n2}$$

For example, the delta subgraph queries of the asymmetric triangle query are as follows:

$$DSQ_1 = a_1 \xrightarrow{\delta} a_2, a_2 \xrightarrow{o} a_3, a_1 \xrightarrow{o} a_3$$

$$DSQ_2 = a_1 \xrightarrow{n} a_2, a_2 \xrightarrow{\delta} a_3, a_1 \xrightarrow{o} a_3$$

$$DSQ_3 = a_1 \xrightarrow{n} a_2, a_2 \xrightarrow{n} a_3, a_1 \xrightarrow{\delta} a_3$$

We represent delta subgraph queries visually as labelled graphs as shown in Figure 3. We assume the updates that arrive, i.e., $|E_\delta|$, are small, say, several edges, compared to existing edges in G. Delta Generic Join runs each delta subgraph query using Generic Join with a QVO that starts with the two query vertices in a δ query edge. This ensures running Generic Join leads to a very small number of 2-matches.

3 OPTIMIZING ONE-TIME QUERIES

In this section, we describe our end-to-end solution to optimizing one-time queries using a mix of WCO join-style intersections and binary joins. The outline of the section is as follows:

- Section 3.1 describes how we optimize the QVOs of WCO plans, which constitute a subset of our plan space. We describe our WCO plan space, three performance effects of different QVOs that we identify and demonstrate through experiments, and the i-cost metric we designed to capture these effects.
- Section 3.2 describes our full plan space, which includes plans with binary joins, and our dynamic programming optimizer that generates plans that can seemlessly mix WCO join-style multiway intersections with binary joins.
- Section 3.4 describes our cost and cardinality estimation technique, which uses a subgraph catalogue that contains statistics about small size subgraphs.

- Section 3.5 describes our adaptive technique that changes the QVOs of WCO sub-plans as actual adjacency list sizes are observed during query execution.

3.1 Optimizing WCO Plans

This section demonstrates our WCO plans, the effects of different query vertex orderings we have identified, and our i-cost metric for WCO plans. Throughout this section, we present several experiments on unlabelled queries for demonstration purposes. The datasets we use in these experiments are described in Table 9.

3.1.1 WCO Plans and E/I Operator. Each query vertex ordering σ of Q is effectively a different WCO plan for Q. Figure 1(b) shows an example σ, which we represent as a chain of m−1 nodes, where the (k−1)th node from the bottom contains a sub-query Q_k, which is the projection of Q onto the first k query vertices of σ. We use two pipelined operators to evaluate WCO plans:
Scan: Leaf nodes of plans, which match a single query edge, are evaluated with a Scan operator. The operator scans the forward adjacency lists in G that match the labels on the query edge, and its source and destination query vertices, and outputs each matched edge $u \rightarrow v \in E$ as a 2-match.
Extend/Intersect (E/I): Internal nodes labelled $Q_k(V_k, E_k)$ that have a child labelled $Q_{k-1}(V_{k-1}, E_{k-1})$ are evaluated with an E/I operator. The E/I operator takes as input (k−1)-matches and extends each tuple t to a set of k-matches. The operator is configured with one or more *adjacency list descriptors* (descriptors for short) and a label l_k for the destination vertex, which indicate the adjacency lists that the operator needs to use when performing intersections when extending each input k-1-match t it receives. Each descriptor is an (`i`, `dir`, l_e) triple, where `i` is the index of a vertex in t, `dir` is `forward` or `backward`, and l_e is the label on the query edge the descriptor represents. For each (k−1)-match t, the operator first computes the extension set $S = \{s_1, \ldots, s_\ell\}$ of t by intersecting the adjacency lists described by its descriptors, ensuring they match the specified edge and destination vertex labels, and then produces one k-match, $(t[1], \ldots, t[k-1], s_i)$, for each $s_i \in S$. When there is a single descriptor, S is the vertices in the adjacency list described by the descriptor. Otherwise, we intersect the adjacency lists using iterative 2-way in tandem intersections.

When extending a single (k−1)-match t to ℓ many k-matches, all of these k matches are identical on the first k-1 query vertices of σ (which is equal to t). Therefore later E/I operators, which might use the adjacency lists of these k-1 vertices may perform repeated intersections. In such cases, our E/I operators cache and reuse all or a subset of the intersections they make for the last tuple they extend. We next explain this optimization through two examples. Consider the diamond-X query Q_{DX} from Section 1 with a QVO $\sigma = a_2a_3a_1a_4$. Let o_3 and o_4 be the E/I operators extending the 2-matches to 3-matches and 3-matches to 4-matches, respectively. Let $t = (v_1, v_2)$ be a 2-match, where v_1 and v_2 match a_2 and a_3, respectively. o_3, when taking t as input, computes an extension set $S = \{s_1, \ldots, s_\ell\}$ and passes each output 3-match (v_1, v_2, s_i) to o_4 consecutively. Therefore, o_4 would intersect the forward adjacency lists of v_1 and v_2 ℓ consecutive times. Instead, o_4 can compute this intersection for (v_1, v_2, s_1) once and reuse it for the following ℓ−1 tuples it receives. Similarly, consider a 4-clique query, which is the same as Q_{DX} with an added edge $a_1 \rightarrow a_4$. o_4, given the same input, would now intersect the forward adjacency lists of v_1, v_2, and s_i. In this example, the intersections that o_4 needs to perform to extend each of the ℓ tuples is different. However, if we order our 2-way in tandem intersections to start with the forward adjacency lists of v_1 and v_2, they would all perform this *partial intersection*, which we can cache and reuse in each of the ℓ extensions, i.e., in each extension, we intersect this partial intersection's result with the forward adjacency list of s_i.

Caching and reusing the last full or partial intersection overall improves the performance of WCO plans as it reduces the amount of repetitive work at the E/I operators. This optimization also

Table 4. Experiment on Intersection Cache Utility for Diamond-X

	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8
Cache On	2.4	2.9	3.2	3.3	3.3	3.4	4.4	6.5
Cache Off	3.8	3.2	3.2	3.3	3.3	3.4	8.5	10.7

Table 5. Runtime (seconds), Intermediate Partial Matches (part. m.), and i-cost of Different QVOs for the Asymmetric Triangle Query

	BerkStan			Live Journal		
QVO	time	part. m.	i-cost	time	part. m.	i-cost
$a_1a_2a_3$	2.6	8M	490M	64.4	69M	13.1B
$a_2a_3a_1$	15.2	8M	55,8B	75.2	69M	15.9B
$a_1a_3a_2$	31.6	8M	55,9B	79.1	69M	17.3B

has a very small memory footprint, since we only store at most one full or one partial intersection at each E/I operator at any point in time during query execution. As a demonstrative example, Table 4 shows the runtime of all WCO plans for the diamond-X query with caching enabled and disabled on the Amazon graph. The orderings in the table are omitted. 4 of the 8 plans utilize the intersection cache and improve their runtime. One of the plans improves by 1.9x.

3.1.2 Effects of QVOs. The work done by a WCO plan is commensurate with the "amount of intersections" it performs. Three main factors affect intersection work and therefore the runtime of a WCO plan σ: (1) directions of the adjacency lists σ intersects, (2) the amount of intermediate partial matches σ generates, and (3) how much σ utilizes the intersection cache. We discuss each effect next.

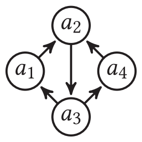
Directions of Intersected Adjacency Lists: Perhaps surprisingly, there are WCO plans that have very different runtimes *only because* they compute their intersections using different directions of the adjacency lists. The simplest example of this is the asymmetric triangle query $a_1{\rightarrow}a_2$, $a_2{\rightarrow}a_3$, $a_1{\rightarrow}a_3$. This query has three QVOs, all of which have the same SCAN operator, which scans each $u{\rightarrow}v$ edge in G, followed by three different intersections (without utilizing the intersection cache):

- σ_1:$a_1a_2a_3$: intersects both u and v's forward lists.
- σ_2:$a_2a_3a_1$: intersects both u and v's backward lists.
- σ_3:$a_1a_3a_2$: intersects u's forward and v's backward lists.

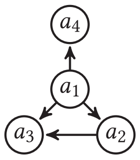
Table 5 shows a demonstrative experiment studying the performance of each plan on the BerkStan and LiveJournal graphs (the i-cost column in the table will be discussed in Section 3.1.3 momentarily). For example, σ_1 is 12.1× faster than σ_2 on the BerkStan graph. Which combination of adjacency list directions is more efficient depends on the structural properties of the input graph, e.g., forward and backward adjacency list distributions.

Different WCO plans generate different partial matches leading to different amount of intersection work. Consider the *tailed triangle* query in Figure 4(b), which can be evaluated by two broad categories of WCO plans:

- EDGE-2PATH: Some plans, such as QVO $a_1a_2a_4a_3$, extend scanned edges $u{\rightarrow}v$ to 2-edge paths ($u{\rightarrow}v{\leftarrow}w$), and then close a triangle from one of 2 edges in the path.
- EDGE-TRIANGLE: Another group of plans, such as QVO $a_1a_2a_3a_4$, extend scanned edges to triangles and then extend the triangles by one edge.



(a) Diamond-X with symmetric triangle.



(b) Tailed triangle.

Fig. 4. Queries used to demonstrate the effects of QVOs.

Table 6. Runtime (seconds), Intermediate Partial Matches (part. m.), and i-cost of Different QVOs for the Tailed Triangle Query

	Amazon			Epinions		
QVO	time	part. m.	i-cost	time	part. m.	i-cost
$a_1a_2a_3a_4$	0.9	15M	176M	0.9	4M	0.9B
$a_1a_3a_2a_4$	1.4	15M	267M	1.0	4M	0.9B
$a_2a_3a_1a_4$	2.4	15M	267M	1.7	4M	1.0B
$a_1a_4a_2a_3$	4.3	35M	640M	56.5	55M	32.5B
$a_1a_4a_3a_2$	4.6	35M	1.4B	72.0	55M	36.5B

Let $|E|$, $|2Path|$, and $|\triangle|$ denote the number of edges, 2-edge paths, and triangles. Ignoring the directions of extensions and intersections, the Edge-2Path plans do $|E|$ many extensions plus $|2Path|$ many intersections, whereas the Edge-Triangle plans do $|E|$ many intersections and $|\triangle|$ many extensions. Table 6 shows the runtimes of the different plans on Amazon and Epinions graphs with intersection caching disabled (again the i-cost column will be discussed momentarily). The first 3 rows are the Edge-Triangle plans. Edge-Triangle plans are significantly faster than Edge-2Path plans, because in unlabelled queries $|2Path|$ is always at least $|\triangle|$ and often much larger. Which QVOs will generate fewer intermediate matches depend on several factors: (i) the structure of the query; (ii) for labelled queries, on the selectivity of the labels on the query; and (3) the structural properties of the input graph, e.g., graphs with low clustering coefficient generate fewer intermediate triangles than those with a high clustering coefficient.

Intersection Cache Hits: The intersection cache of our E/I operator is utilized more if the QVO extends $(k-1)$-matches to a_k using adjacency lists with indices from $a_1 \ldots a_{k-2}$. Intersections that access the $(k-1)^{\text{th}}$ index cannot be reused, because a_{k-1} is the result of an intersection performed in a previous E/I operator and will match to different vertex IDs. Instead, those accessing indices $a_1 \ldots a_{k-2}$ can potentially be reused. We demonstrate that some plans perform significantly better than others only because they can utilize the intersection cache. Consider a variant of the diamond-X query in Figure 4(a). One type of WCO plans for this query extend $u \rightarrow v$ edges to (u, v, w) symmetric triangles by intersecting u's backward and v's forward adjacency lists. Then each triangle is extended to complete the query, intersecting again the forward and backward adjacency lists of one of the edges of the triangle. There are two sub-groups of QVOs that fall under this type of plans: (i) $a_2a_3a_1a_4$ and $a_2a_3a_4a_1$, which are equivalent plans due to symmetries in the query, so will perform exactly the same operations, and (ii) $a_1a_2a_3a_4$, $a_3a_1a_2a_4$, $a_3a_4a_2a_1$, and $a_4a_2a_3a_1$, which are also equivalent plans. Importantly, all of these plans cumulatively perform exactly the same intersections but those in group (i) and (ii) have different orders in which these intersections are performed, which lead to different intersection cache utilizations.

Table 7 shows the performance of one representative plan from each sub-group: $a_2a_3a_1a_4$ and $a_1a_2a_3a_4$, on several graphs. The $a_2a_3a_1a_4$ plan is 4.4× faster on Epinions and 3× faster on Amazon.

Table 7. Runtime (seconds), Intermediate Partial Matches (part. m.), and i-cost of Some QVOs for the Symmetric Diamond-X Query

	Amazon			Epinions		
QVO	time	part. m.	i-cost	time	part. m.	i-cost
$a_2a_3a_1a_4$	1.0	11M	0.1B	0.9	2M	0.1B
$a_1a_2a_3a_4$	3.0	11M	0.3B	4.0	2M	1.0B

This is because when $a_2a_3a_1a_4$ extends $a_2a_3a_1$ triangles to complete the query, it will be accessing a_2 and a_3, so the first two indices in the triangles. For example if ($a_2 = v_0$, $a_3 = v_1$) extended to t triangles $(v_0, v_1, v_2), \ldots, (v_0, v_1, v_{t+2})$, these partial matches will be fed into the next E/I operator consecutively, and their extensions to a_4 will all require intersecting v_0 and v_1's backward adjacency lists, so the cache would avoid $t-1$ intersections. Instead, the cache will not be utilized in the $a_1a_2a_3a_4$ plan. Our cache gives benefits similar to *factorization* [52]. In factorized processing, the results of a query are represented as Cartesian products of independent components of the query. In this case, matches of a_1 and a_4 are independent and can be done once for each match of a_2a_3. A study of factorized processing is an interesting topic for future work.

3.1.3 Cost Metric for WCO Plans. We introduce a new cost metric called *intersection cost* (i-cost), which we define as the size of adjacency lists that will be accessed and intersected by different WCO plans. Consider a WCO plan σ that evaluates sub-queries $Q_2, \ldots, Q_m$, respectively, where $Q = Q_m$. Let t be a $(k-1)$-match of Q_{k-1} and suppose t is extended to instances of Q_k by intersecting a set of adjacency lists, described with adjacency list descriptors A_{k-1}. Formally, i-cost of σ is as follows:

$$\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{\substack{(i, dir) \in A_{k-1} \\ \text{s.t. (i, dir) is accessed}}} |t[i].dir|. \tag{1}$$

We discuss how we estimate i-costs of plans in Section 3.4. For now, note that Equation (1) captures the three effects of QVOs we identified: (i) the $|t.dir|$ quantity captures the sizes of the adjacency lists in different directions; (ii) the second summation is over all intermediate matches, capturing the size of intermediate partial matches; and (iii) the last summation is over all adjacency lists that are accessed, so ignores the lists in the intersections that are cached. For the demonstrative experiments we presented in the previous section, we also report the *actual* i-costs of different plans in Tables 5, 6, and 7. The actual i-costs are measured in a profiled run of each query. Notice that in each experiment, i-costs of plans rank in the correct order of runtimes of plans.

There are alternative cost metrics from literature, such as the C_{out} [16] and C_{mm} [35] metrics, that would also do reasonably well in differentiating good and bad WCO plans. However, these metrics capture only the effect of the number of intermediate matches. For example, they would not differentiate the plans in the asymmetric triangle query or the symmetric diamond-X query, i.e., the plans in Tables 5 and 7 have the same actual C_{out} and C_{mm} costs.

3.2 Full Plan Space and Dynamic Programming Optimizer

In this section, we describe our full plan space, which contain plans that include binary joins in addition to the E/I operator, the costs of these plans, and our dynamic programming optimizer.

3.2.1 Hybrid Plans and HashJoin Operator. In Section 3.1, we represented a WCO plan σ as a chain, where each internal node o_k had a single child labelled with Q_k, which was the projection of Q onto the first k query vertices in σ. A plan in our full plan space is a rooted tree as follows. Below, Q_k refers to a projection of Q onto an arbitrary set of k query vertices.

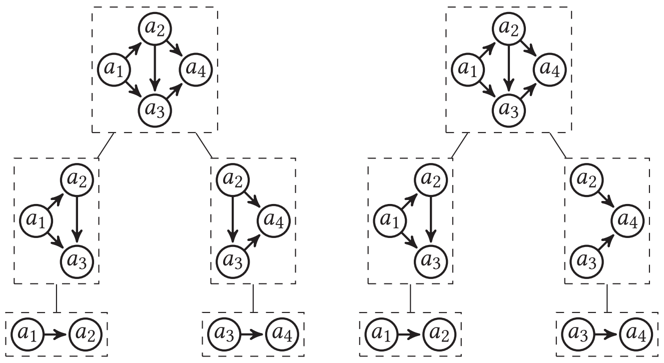
- Leaf nodes are labeled with a single query edge of Q.
- Root is labeled with Q.
- Each internal node o_k is labeled with $Q_k = \{V_k, E_k\}$, with the *projection constraint* that Q_k is a projection of Q onto a subset of query vertices. o_k has either one child or two children. If o_k has one child o_{k-1} with label $Q_{k-1} = \{V_{k-1}, E_{k-1}\}$, then Q_{k-1} is a subgraph of Q_k with one query vertex $q_v \in V_k$ and q_v's incident edges in E_k missing. This represents a WCO-style extension of partial matches of Q_{k-1} by one query vertex to Q_k. If o_k has two children o_{c1} and o_{c2} with labels Q_{c1} and Q_{c2}, respectively, then $Q_k = Q_{c1} \cup Q_{c2}$ and $Q_k \neq Q_{c1}$ and $Q_k \neq Q_{c2}$. This represents a binary join of matches Q_{c1} and Q_{c2} to compute Q_k.

As before, leaves map to Scan operators, an internal node o_k with a single child maps to an E/I operator. If o_k has two children, then it maps to a Hash-Join operator:
Hash-Join: We use the classic hash join operator, which first creates a hash table of all of the tuples of Q_{c1} on the common query vertices between Q_{c1} and Q_{c2}. The table is then probed for each tuple of Q_{c2}.

Our plans are highly expressive and contain several classes of plans: (1) WCO plans from the previous section, in which each internal node has one child; (2) BJ plans, in which each node has two children and satisfy the projection constraint; and (3) hybrid plans that satisfy the projection constraint. We show in our supplementary Appendix C that our hybrid plans contain EmptyHeaded's minimum-width GHD-based hybrid plans that satisfy the projection constraint. For example the hybrid plan in Figure 1(c) corresponds to a GHD for the diamond-X query with width 3/2. In addition, our plan space also contains hybrid plans that do not correspond to a GHD-based plan. Figure 2 shows an example hybrid plan for the 6-cycle query that is not in EmptyHeaded's plan space. As we show in our evaluations, such plans can be very efficient for some queries.

The projection constraint prunes two classes of plans:

1. Our plan space does not contain BJ plans that first compute open triangles and then close them. Consider a triangle Q_T that is a subquery of a larger query Q. Suppose Q_T is $a_1{\rightarrow}a_2{\rightarrow}a_3$, $a_1{\rightarrow}a_3$. Then due to the projection constraint, we do not enumerate any plan that contains an open triangle Q_{OT}, e.g., $a_1{\rightarrow}a_2{\rightarrow} a_3$, of Q_T, with, say, a later binary join to close the $a_1{\rightarrow}a_3$ edge. This is because Q_{OT} is not a projection of Q, as it does not contain the $a_1{\rightarrow}a_3$ edge. Such BJ plans are in the plan spaces of existing optimizers, e.g., Postgres, MySQL, and Neo4j. This is not a disadvantage, because for each such plan, there is a more efficient WCO plan that computes triangles directly with an intersection of two already-sorted adjacency lists. Specifically, we force the triangles to be computed by extending edges (which are projections of Q) directly to Q_T using WCO-style intersections.
2. More generally, some of our hybrid plans contain the same query edge $a_i{\rightarrow}a_j$ in multiple parts of the join tree, which may look redundant, because $a_i{\rightarrow}a_j$ is effectively joined multiple times. There can be alternative plans that remove $a_i{\rightarrow}a_j$ from all but one of the sub-trees. For example, consider the two hybrid plans P_1 and P_2 for the diamond-X query in Figure 5(a) and (b), respectively. P_2 is not in our plan space, because it does not satisfy the projection constraint, because $a_2{\rightarrow}a_3$ is not in the right sub-tree. Omitting such plans is also not a disadvantage, because we duplicate $a_i{\rightarrow}a_j$ only if it closes cycles in a sub-tree, which effectively is an additional filter that reduces the partial matches of the sub-tree. For example, on the Amazon graph dataset, P_1 takes 14.2 s and P_2 takes 56.4 s so P_1 is 3.97× faster than P_2.



(a) Plan $P1$. (b) Plan $P2$.

Fig. 5. Two plans: $P1$ shares a query edge and $P2$ does not.

3.2.2 Cost Metric for General Plans. A Hash-Join operator performs a very different computation than E/I operators, so the cost of Hash-Join needs to be normalized with i-cost. This is an approach taken by DBMSs to merge costs of multiple operators, e.g., a scan and a group-by, into a single cost metric. Consider a Hash-Join operator o_k that will join matches of Q_{c1} and Q_{c2} to compute Q_k. Suppose there are n_1 and n_2 instances of Q_{c1} and Q_{c2}, respectively. Then o_k will hash n_1 number of tuples into a table and probe this table n_2 times. We compute two weight constants w_1 and w_2 and calculate the cost of o_k as $w_1 n_1 + w_2 n_2$ i-cost units. These weights can be hardcoded as done in the C_{mm} cost metric [35], but we pick them empirically.

3.2.3 Dynamic Programming Optimizer. Algorithm 1 shows the pseudocode of our optimizer. Our optimizer takes as input a query $Q(V_Q, E_Q)$. We start by enumerating and computing the cost of all WCO plans (line 1). We discuss this step momentarily. We then initialize the cost of computing 2-vertex sub-queries of Q, so each query edge, to the selectivity of the label on the query edge (line 2). Then starting from $k = 3$ up to $|V_Q|$, for each k-vertex sub-query Q_k of Q, we find the lowest cost plan $P^*_{Q_k}$ to compute Q_k in three different ways:

(i) $P^*_{Q_k}$ is the lowest cost WCO plan that we enumerated (line 5).
(ii) $P^*_{Q_k}$ extends the best plan $P^*_{Q_{k-1}}$ for a Q_{k-1} by an E/I operator (Q_{k-1} contains one fewer query vertex than Q_k) (lines 7–10).
(iii) $P^*_{Q_k}$ merges two best plans $P^*_{Q_{c1}}$ and $P^*_{Q_{c2}}$ for Q_{c1} and Q_{c2}, respectively, with a Hash-Join (lines 12–15).

The best plan for each Q_k is stored in a *sub-query map.* We enumerate all WCO plans, because the best WCO plan $P^*_{Q_k}$ for Q_k is not necessarily an extension of the best WCO plan $P^*_{Q_{k-1}}$ for a Q_{k-1} by one query vertex. That is because $P^*_{Q_k}$ may be extending a worse plan $P^{bad}_{Q_{k-1}}$ for Q_{k-1} if the last extension has a good intersection cache utilization. Strictly speaking, this problem can arise when enumerating hybrid plans, too, if an E/I operator in case (ii) above follows a Hash-Join. A full plan space enumeration would avoid this problem completely but we adopt dynamic programming to make our optimization time efficient, i.e., to make our optimizer efficient, we are potentially sacrificing picking the plan with the lowest estimated-cost. However, we verified that our optimizer returns the same plan as a full enumeration optimizer in all of our experiments in Section 6. So at least for our experiments there, we have not sacrificed optimality.

Finally, our optimizer omits plans that contain a Hash-Join that can be converted to an E/I. Consider the $a_1 \rightarrow a_2 \rightarrow a_3$ query. Instead of using a Hash-Join to materialize the $a_2 \rightarrow a_3$ edges and

ALGORITHM 1: DP Optimization Algorithm

```
Require: $Q(V_Q, E_Q)$
1: WCOP = enumerateAllWCOPlans($Q$) // WCO plans
2: QPMap: init each $a_i \xrightarrow{l_e} a_j$'s cost to the $\mu(l_e)$
3: for $k = 3, \ldots, |V_Q|$ do
4:     for $V_k \subseteq V$ s.t. $|V_k|$=k do
5:         $Q_k(V_k, E_k)=\Pi_{V_k} Q$; bestP = WCOP($Q_k$); minC = $\infty$
6:         // Find best plan that extends to $Q_k$ by one query vertex
7:         for $v_j \in V_k$ let $Q_{k-1}(V_{k-1}, E_{k-1}) = \Pi_{V_k - v_j} Q_k$ do
8:             P = QPMap($Q_{k-1}$).extend($Q_k$);
9:             if cost(P) < minC then
10:                bestPlan = P;
11:        // Find best plan that generates $Q_i$ with a binary join
12:        for $V_{c1}, V_{c2} \subset V_k$: $Q_{c1}=\Pi_{V_{c1}} Q_k, Q_{c2}=\Pi_{V_{c2}} Q_k$ do
13:            P = join(QPMap($Q_{c1}$), QPMap($Q_{c2}$));
14:            if cost(P) < minC then
15:                bestPlan = P;
16:    QPMap($Q_k$) = bestPlan;
17: return QPMap(Q);
```

then probe a scan of $a_1 \rightarrow a_2$ edges, it is more efficient to use an E/I to extend $a_1 \rightarrow a_2$ edges to a_3 using a_2's forward adjacency list.

3.3 Plan Generation for Very Large Queries

Our optimizer can take a very long time to generate a plan for large queries. For example, enumerating only the best WCO plan for a 20-clique requires inspecting 20! different QVOs, which would be prohibitive. To overcome this, we further prune plans for queries with more than 10 query vertices as follows:

- We avoid enumerating all WCO plans. Instead, WCO plans get enumerated in the DP part of the optimizer. Therefore, we possibly ignore good WCO plans that benefit from the intersection cache.
- At each iteration k, out of the t_k many plans that evaluate a k-vertex sub-query of Q we only keep the r lowest cost plans (5 by default). At iteration $k + 1$, we will extend these r plans to t_{k+1} many plans that evaluate $(k + 1)$-vertex sub-queries but we will again keep on the top r, so on and so forth.

3.4 Cost and Cardinality Estimation

To assign costs to the plans we enumerate, we need to estimate: (1) the cardinalities of the partial matches different plans generate; (2) the i-costs of extending a sub-query Q_{k-1} to Q_k by intersecting a set of adjacency lists in an E/I operator; and (3) the costs of Hash-Join operators. We focus on the setting where each subquery Q_k has labels on the edges and the vertices. In the remainder of this section, we describe how we make these estimations using a data structure called the *subgraph catalogue*. However, we emphasize that our optimizer can be used with any estimation technique that can estimate i-cost and cardinalities of partial matches of sub-queries and a detailed study of advanced cost and cardinality techniques is beyond this article's scope and is left for future work.

Table 8 shows an example catalogue. Each entry contains a key (Q_{k-1}, A, $a_k^{l_k}$), where A is a set of (labelled) query edges and $a_k^{l_k}$ is a query vertex with label l_k. Let Q_k be the subgraph that extends

Table 8. A Subgraph Catalogue

$(Q_{k-1}$	A	$l_k)$	$\lvert A\rvert$	$\mu(Q_k)$
$(1^{l_a}\xrightarrow{l_x}2^{l_b};$	L_1:$2\xrightarrow{l_x};$	$3^{l_a})$	$\lvert L_1\rvert$:4.5	3.8
$(1^{l_a}\xrightarrow{l_x}2^{l_b};$	L_1:$2\xrightarrow{l_x};$	$3^{l_b})$	$\lvert L_1\rvert$:4.5	2.4
$(1^{l_a}\xrightarrow{l_x}2^{l_b};$	L_1:$2\xrightarrow{l_y};$	$3^{l_a})$	$\lvert L_1\rvert$:8.0	3.2
$(1^{l_a}\xrightarrow{l_x}2^{l_a};$	L_1:$1\xrightarrow{l_x}$, L_2:$2\xrightarrow{l_x};$	$3^{l_a})$	$\lvert L_1\rvert$:4.2, $\lvert L_2\rvert$:5.1	1.5
$(1^{l_a}\xrightarrow{l_x}2^{l_a};$	L_1:$1\xleftarrow{l_x}$, L_2:$2\xleftarrow{l_x};$	$3^{l_a})$	$\lvert L_1\rvert$:9.8, $\lvert L_2\rvert$:8.4	2.5
(...;	...;	...)	...	...

A is a set of adjacency list descriptors; μ is selectivity.

Q_{k-1} with a query vertex labelled with $a_k^{l_k}$ and query edges in A. Each entry contains two estimates for extending a match of a sub-query Q_{k-1} to Q_k by intersecting the adjacency lists A describes:

1. |A|: Average sizes of the lists in A that are intersected.
2. $\mu(Q_k)$: Average number of Q_k that will extend from one Q_{k-1}, i.e., the average number of vertices that: (i) are in the extension set of intersecting the adjacency lists A; and (ii) have label l_k.

In Table 8, the query vertices of the input subgraph Q_{k-1} are shown with canonicalized integers, e.g., 0, 1 or 2, instead of the non-canonicalized a_i notation we used before. Note that Q_{k-1} can be extended to Q_k using different adjacency lists A with different i-costs. The fourth and fifth entries of Table 8, which extend a single edge to an asymmetric triangle, demonstrate this possibility.

3.4.1 Catalogue Construction. For each input G, we construct a catalogue containing all entries that extend an at most h-vertex subgraph to an $(h+1)$-vertex subgraph. By default, we set h to 3. When generating a catalogue entry for extending Q_{k-1} to Q_k, we do not find all instances of Q_{k-1} and extend them to Q_k. Instead, we first sample Q_{k-1}. We take a WCO plan that extends Q_{k-1} to Q_k. We then sample z random edges (1,000 by default) uniformly at random from G in the Scan operator. The last E/I operator of the plan extends each partial match t it receives to Q_k by intersecting the adjacency lists in A. The operator measures the size of the adjacency lists in A and the number of Q_k's this computation produced. These measurements are averaged and stored in the catalogue as |A| and $\mu(Q_k)$ columns.

3.4.2 Cost Estimations. We use the catalogue to do three estimations as follows:
1. Cardinality of Q_k: To estimate the cardinality of Q_k, we pick a WCO plan P that computes Q_k through a sequence of (Q_{j-1}, A_j, l_j) extensions. The estimated cardinality of Q_k is the product of the $\mu(A_j)$ of the (Q_{j-1}, A_j, l_j) entries in the catalogue. If the catalogue contains entries with up to h-vertex subgraphs and Q_k contains more than h nodes, then some of the entries we need for estimating the cardinality of Q_k will be missing. Suppose for calculating the cardinality of Q_k, we need the $\mu(A_x)$ of an entry (Q_{x-1}, A_x, l_x) that is missing, because Q_{x-1} contains $x-1 > h$ query vertices. Let $z = (x-h-1)$. In this case, we remove each z-size set of query vertices $a_1, \ldots a_z$ from Q_{x-1} and Q_x, and the adjacency list descriptors from A_x that include $1, \ldots, z$ in their indices. Let (Q_{y-1}, A_y, l_y) be the entry we get after a removal. We look at the $\mu(A_y)$ of (Q_{y-1}, A_y, l_y) in the catalogue. Out of all such z set removals, we use the minimum $\mu(A_y)$ we find.

As an example, consider a missing entry for extending Q_{k-1}= 1→2→3 by one query vertex to 4 by intersecting three adjacency lists all pointing to 4 from 1, 2, and 3. For simplicity, let us ignore the labels on query vertices and edges. The resulting sub-query Q_k will have two triangles: (i) an asymmetric triangle touching edge 1→2 and (ii) a symmetric triangle touching 2→3. Suppose entries in the catalogue indicate that an edge on average extends to 10 asymmetric triangles but to 0 symmetric triangles. We estimate that Q_{k-1} will extend to zero Q_k taking the minimum of our two estimates.

2. I-cost of E/I operator: Consider an E/I operator o_k extending Q_{k-1} to Q_k using adjacency lists A. We have two cases:

- No intersection cache: When o_k does not utilize the intersection cache, we estimate its i-cost as:

$$\text{i-cost}(o_k) = \mu(Q_{k-1}) \times \sum_{L_i \in A} |L_i|. \tag{2}$$

  Here, $\mu(Q_{k-1})$ is the estimated cardinality of Q_{k-1}, and $|L_i|$ is the average size of the adjacency list $L_i \in A$ that are logged in the catalogue for entry $(Q_{k-1}, A, a_k^{l_k})$ (i.e., the $|A|$ column).
- Intersection cache utilization: If two or more of the adjacency list in A, say, L_i and L_j, access the vertices in a partial match Q_j that is smaller than Q_{k-1}, then we multiply the estimated sizes of L_i and L_j with the estimated cardinality of Q_j instead of Q_{k-1}. This is because we infer that o_k will utilize the intersection cache for intersecting L_i and L_j.

Reasoning about utilization of intersection cache is critical in picking good plans. For example, recall our experiment from Table 4 to demonstrate that the intersection cache broadly improves all plans for the diamond-X query. Our optimizer, which is "cache-conscious" picks σ_2 ($a_2a_3a_4a_1$). Instead, if we ignore the cache and make our optimizer "cache-oblivious" by always estimating i-cost with Equation (2), it picks the slower σ_4 ($a_1a_2a_3a_4$) plan. Similarly, our cache-conscious optimizer picks $a_2a_3a_1a_4$ in our experiment from Table 7. Instead, the cache-oblivious optimizer assigns the same estimated i-cost to plans $a_2a_3a_1a_4$ and $a_1a_2a_3a_4$, so cannot differentiate between these two plans and picks one arbitrarily.

3. Cost of HASH-JOINoperator: Consider a HASH-JOIN operator joining Q_{c1} and Q_{c2}. The estimated cost of this operator is simply $w_1n_1 + w_2n_2$ (recall Section 3.2.2), where n_1 and n_2 are now the estimated cardinalities of Q_{c1} and Q_{c2}, respectively.

3.4.3 Limitations. Similarly to Markov tables [3] and MD- and Pattern-tree summaries [39], our catalogue is an estimation technique that is based on storing information about small size subgraphs and extending them to make estimates about larger subgraphs. We review these techniques in detail and discuss our differences in Section 7. Here, we discuss several limitations that are inherent in such techniques. We emphasize again that our optimizer can be used with more advanced cardinality estimation techniques and studying such techniques is beyond the scope of this article.

First, as expected our estimates (both for i-cost and cardinalities) get worse as the size of the subgraphs for which we make estimates increase beyond h. Equivalently, as h increases, our estimates for fixed-size large queries get better. At the same time, the size of the catalogue increases significantly as h increases. Similarly, the size of the catalogue increases as graphs get more heterogenous, i.e., contain more labels. Second, using larger sample sizes, i.e., larger z values, increase the accuracy of our estimates but require more time to construct the catalogue. Therefore h and z respectively trade off catalogue size and creation time with the accuracy of estimates. We provide demonstrative experiments of these tradeoffs in our supplementary Appendix B for cardinality estimates.

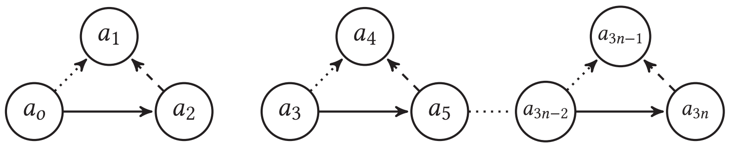


Fig. 6. Input graph for adaptive QVO example.

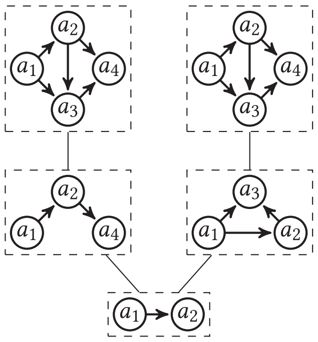


Fig. 7. Example adaptive WCO plan.

3.5 Adaptive WCO Plan Evaluation

Recall that the $|A|$ and μ statistics stored in a catalogue entry (Q_{k-1}, A, $a_k^{l_k}$), are estimates of the adjacency list sizes (and selectivities) for matches of Q_{k-1}. These are *estimates* based on *averages* over many sampled matches of Q_{k-1}. In practice, *actual* adjacency list sizes and selectivities of *individual* matches of Q_{k-1} can be very different. Let us refer to parts of plans that are chains of one or more E/I operators as *WCO subplans.* Consider a WCO subplan of a *fixed* plan P that has a QVO σ^* and extends partial matches of a sub-query Q_i to matches of Q_k. Our optimizer picks σ^* based on the estimates of the average statistics in the catalogue. Our adaptive evaluator updates our estimates for individual matches of Q_i (and other sub-queries in this part of the plan) based on actual statistics observed during evaluation and possibly changes σ^* to another QVO for each individual match of Q_i.

Example 3.1. Consider the input graph G shown in Figure 6. G contains 3n edges. Consider the diamond-X query and the WCO plan P with $\sigma = a_2a_3a_4a_1$. Readers can verify that this plan will have an i-cost of 3n: 2n from extending solid edges, n from extending dotted edges, and 0 from extending dashed edges. Now consider the following *adaptive* plan that picks σ for the dotted and dashed edges as before but $\sigma' = a_2a_3a_1a_4$ for the solid edges. For the solid edges, σ' incurs an i-cost of 0, reducing the i-cost to n.

3.5.1 Adaptive Plans. We optimize subgraph queries as follows. First, we get a *fixed* plan P from our dynamic programming optimizer. If P contains a chain of two or more E/I operators $o_i, o_{i+1} \ldots, o_k$, then we replace it with an *adaptive* WCO plan. The adaptive plan extends the first partial matches Q_i that o_i takes as input in all possible (connected) ways to Q_k. In WCO plans o_i is Scan and Q_i is one query edge. Therefore in WCO plans, we fix the first two query vertices in a QVO and pick the rest adaptively. Figure 7 shows the adaptive version of the fixed plan for the diamond-X query from Figure 1(b). In addition, we adapt hybrid plans if they have a chain of two or more E/I operators.

3.5.2 Adaptive Operators. Unlike the operators in fixed plans, our adaptive operators can feed their outputs to multiple operators. An adaptive operator o_i is configured with a function f that takes a partial match t of Q_i and decides which of the next operators should be given t. f consists of two high-level steps: (1) For each possible σ_j that can extend Q_i to Q_k, f re-evaluates the estimated i-cost of σ_j by re-calculating the cost of plans using *updated cost estimates* (explained momentarily). o_i gives t to the next E/I operator of σ_j^* that has the lowest re-calculated cost. The cost of σ_j is re-evaluated by changing the estimated adjacency list sizes that were used in cardinality and i-cost estimations with actual adjacency list sizes we obtain from t.

Example 3.2. Consider the diamond-X query from Figure 1(a) and suppose we have an adaptive plan in which the Scan operator matches edges to a_2a_3, so for each edge needs to decide whether to pick the ordering $\sigma_1 : a_2a_3a_4a_1$ or $\sigma_2 : a_2a_3a_1a_4$. Suppose the catalogue estimates the sizes of $|a_2{\rightarrow}|$ and $|a_3{\rightarrow}|$ as 100 and 2000, respectively. So we estimate the i-cost of extending an a_2a_3 edge to $a_2a_3a_4$ as 2100. Suppose the selectivity μ_j of the number of triangles this intersection will generate is 10. Suppose Scan reads an edge $u{\rightarrow}v$ where u's forward adjacency list size is 50 and v's backward adjacency list size is 200. Then we update our i-cost estimate directly to 250 and μ_j to $10 \times (50/100) \times 200/2000 = 0.5$.

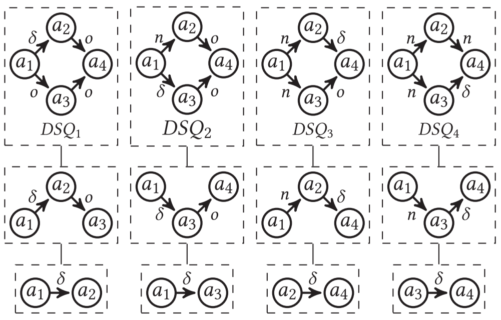
As we show in our evaluations, adaptive QVO selection improves the performance of many WCO plans but more importantly guards our optimizer from picking bad QVOs.

4 OPTIMIZING CONTINUOUS QUERIES

We next consider evaluating continuous subgraph queries that are registered in a GDBMS and maintaining their outputs as updates arrive to the graphs. Continuous queries provide trigger functionality to developers and are used to develop applications that require detecting the emergence and/or deletion of subgraph patterns in a graph, e.g., the MagicRecs recommendation application from Twitter [23] that continuously monitors diamonds in the Twitter social network. We consider the setting where a set of subgraph queries $\bar{Q}$ are registered in a system and a series of updates E_{δ_1}, E_{δ_2}\ldots arrive at G and our goal is to detect the emergence and deletions of subgraphs that match any of the $Q \in \bar{Q}$. In this section, we describe our end-to-end solution to optimizing these queries using WCO plans.

Our approach is based on the *Delta Generic Join* incremental view maintenance algorithm that we reviewed in Section 2. References [6] and [29] used this framework for evaluating single subgraph queries, respectively in a distributed and single node settings, where QVOs were picked arbitrarily or using simple heuristics. We build upon this framework and study how to evaluate multiple continuous queries and select the QVOs in a cost-based optimizer using the i-cost metric we introduced in Section 3. Our optimizer generates a single low i-cost *combined plan*, which combines the individual plans generated for each delta subgraph query and shares common computations and evaluates all of the queries in $\bar{Q}$. The outline of this section is as follows:

- Section 4.1 describes our WCO plan space for delta subgraph queries and our combined plans for sets of delta subgraph queries.
- Section 4.2 describes our greedy optimizer that picks QVOs for each delta subgraph query and shares subplans to generate a combined plan.
- Section 4.3 describes our partial intersection sharing technique that allows sharing of computations across the E/I operators that perform different intersections but have partial overlaps in the intersections. We motivate this optimization by an important empirical observation we make about the limitation of computation sharing in combined plans.


(a) Individual plans.

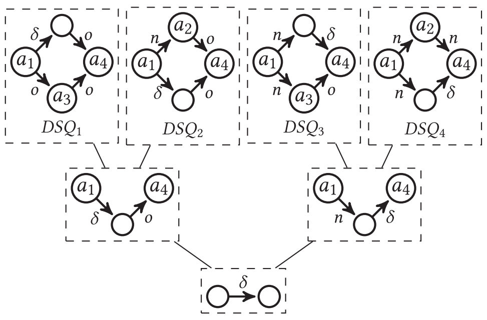

(b) Combined plan of the individual plans.

Fig. 8. Individual and combined plans for delta subgraph queries of a diamond query example.

In the remainder of the section, we assume that given a batch of updates E_δ, there are three types of adjacency lists in memory for each vertex v in G:

- delta contains v's neighbours in E_δ.
- old contains v's neighbours in G before the update.
- new contains v's neighbours in G after the update.

4.1 Optimizing Plans for Delta Subgraph Queries and Combined Plans

Recall from Section 2.2 that Delta Generic Join decomposes each continuous subgraph query Q into n delta subgraph queries, where n is the number of query edges in Q. Then, Delta Generic Join picks a QVO for each delta subgraph query starting from the two query vertices that form the δ query edge and evaluates it one query vertex at a time, and then unions the results.

The QVO for each delta subgraph query is essentially a logical plan. For example, consider the *diamond* query $a_1{\rightarrow}a_2$, $a_1{\rightarrow}a_3$, $a_2{\rightarrow}a_4$, $a_3{\rightarrow}a_4$ and assume for simplicity that the query has a single query edge label on each query edge. Consider the following delta decomposition of this query:

$$DSQ_1 = a_1 \xrightarrow{\delta} a_2, a_1 \xrightarrow{o} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4$$

$$DSQ_2 = a_1 \xrightarrow{n} a_2, a_1 \xrightarrow{\delta} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4$$

$$DSQ_3 = a_1 \xrightarrow{n} a_2, a_1 \xrightarrow{n} a_3, a_2 \xrightarrow{\delta} a_4, a_3 \xrightarrow{o} a_4$$

$$DSQ_4 = a_1 \xrightarrow{n} a_2, a_1 \xrightarrow{n} a_3, a_2 \xrightarrow{n} a_4, a_3 \xrightarrow{\delta} a_4$$

Figure 8(a) shows four query plans respectively corresponding to the following four QVOs: $a_1a_2a_4a_3$, $a_1a_3a_4a_2$, $a_2a_4a_1a_3$ and $a_3a_4a_1a_2$. There are two differences between these logical plans and the ones for one-time subgraph queries from Section 3.1: (i) Each operator o_i is a sub-query S_i whose query edges are labeled with δ, n, or o to indicate whether they match E_δ, E_n, or E_o, respectively; and (ii) each internal node has only one child. So the plans do not contain binary joins. We avoid binary joins, because one branch of a binary join would match sub-queries with only old or new query edges, which we assume are very large compared to the delta edges (recall from Section 2.2 that delta subgraph queries have only one δ query edge). Therefore joins in such branches would lead to very large intermediate results.

We evaluate these plans with Scan and E/I operators, which slightly differ from the ones in Section 3.1.1:

Scan: Scans only the edges in E_δ, so the edges in the delta adjacency lists and appends a +/− label to them indicating a deletion or an addition of an edge.

Extend/Intersect (E/I): Each adjacency list descriptor is now an (`i`, `dir`, `version`) triple. `version` can be `old` or `new` indicating whether the adjacency list should come from the `new` or `old` adjacency lists of vertices (E/I's do not access E_δ).

When evaluating multiple delta subgraph queries at the same time, there might be opportunities to share computation between the plans. This opportunity arises when plans of two or more delta subgraph queries contain operators whose outputs are both isomorphic sub-queries, so we do not need to compute the same sub-queries over and over again. Instead, we can just compute these sub-queries once and give their results to possibly multiple operators. Due to these opportunities, instead of evaluating each delta subgraph query separately, we evaluate all of them together using a *combined plan*, which we define next.

Definition 4.1 (Combined Plan). Let $\bar{Q}$ be a set of queries and $\bar{Q}_{DSQ}$ be a set of DSQs corresponding to the union of a delta decomposition for each query in $\bar{Q}$. We assume that no two DSQs in $\bar{Q}_{DSQ}$ are isomorphic as that would imply that $\bar{Q}$ contains two isomorphic queries (note that two DSQs from the same query cannot be isomorphic, because according to the decomposition the number of n- and o-labeled edges in each DSQ is different). A combined plan *(CP)* for $\bar{Q}_{DSQ}$ is a directed acyclic graph of Scan and E/I operators that contain: (1) one source Scan operator that scan δ edges, i.e., updates to the graph; (2) a set of E/I operators that take input from exactly one other operator (Scan or E/I) but can give outputs to any number of E/I operators; and (3) exactly $|\bar{Q}_{DSQ}|$ many sink E/I operator, where there is a one to one mapping between the outputs of sink E/I operators and DSQs in $\bar{Q}_{DSQ}$, i.e., the output of each DSQ in $\bar{Q}_{DSQ}$ is produced by exactly one sink E/I operator. An E/I operator produces the output of a DSQ if the subgraph query that it evaluates is isomorphic to the DSQ considering the edge labels δ, n, and o.

For example, Figure 8(b) shows an example combined plan for the 4 DSQs above. In Figure 8(b), we omit the a_i labels on the query vertices that take different labels for different delta subgraph queries. Tracing back from each sink operator back to the source (Scan) operator effectively gives a QVO for one DSQ. For example, the left most sink operator evaluates DSQ_1 above and uses exactly the same QVO as the leftmost DSQ. In fact, the combined plan in Figure 8(b) evaluates the 4 DSQs in our example using exactly the 4 individual plans from Figure 8(b) but shares some duplicate operators whose inputs and outputs are isomorphic subgraphs, such as the level 2 E/I operators of DSQ_1 and DSQ_2 as well as DSQ_3 and DSQ_4.

Our continuous subgraph query optimizer aims to find an efficient combined plan evaluating all of the DSQs of a delta decomposition of a set of registered queries $\bar{Q}$ in GraphflowDB. Similarly to one-time queries, we adopt a cost-based approach using the i-cost metric and compute the estimated cost of a combined plan as the sum of the estimated i-costs of its E/I operators (we take the cost of Scan operator as 1). When estimating the i-costs of an E/I operator, we use the same catalogue-based cost estimation formulas described for one-time queries in Section 3.4.2 (recall the two bullet points under item 2). In particular, we do not differentiate between `delta`, `old`, and `new` query edges that are used in the operators of the combined plan. There are two reasons for this: (1) the `delta` query edges only appear in the source nodes in combined plans, which map to Scan operator and get a uniform cost of 1; and (2) the differences between the lengths of the `old` and `new` adjacency lists are minor, because we assume there are a small number of edges in each update to the graph.

We can formally state the optimization problem our optimizer solves for continuous queries. For simplicity of the formal definition, we assume that a full catalogue is available to the optimizer, i.e., information about every possible Q_{k-1} to Q_k extension exists in the catalogue. As we explain momentarily, this assumption holds in our implementation as well, i.e., for continuous queries GraphflowDB generates a catalogue that contains all the relevant entries for the registered queries.

Definition 4.2 (Multiple Continuous Subgraph Query Optimization Problem). Given a set of queries $\bar{Q}$ and a delta decomposition of these queries $\bar{Q}_{DSQ}$ and an arbitrary full catalogue C, find the lowest estimated cost combined plan CP evaluating $\bar{Q}_{DSQ}$.

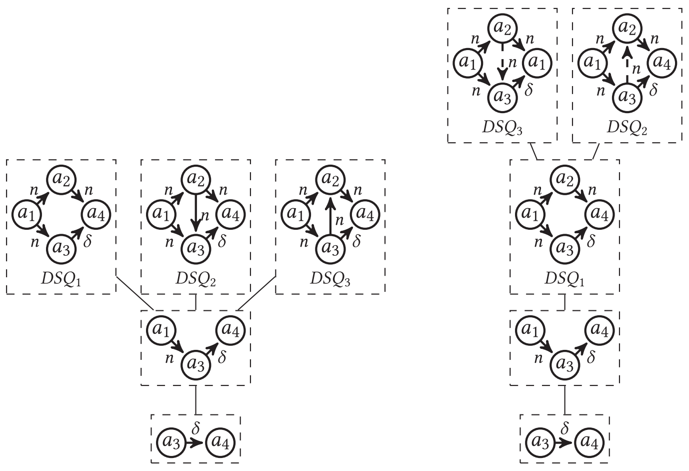
We do not establish the hardness of this formal problem in this article and leave this to future work. However, in our supplementary Appendix A.1, we show that the natural decision version of a generalized version of this problem, in which we assume that $\bar{Q}_{DSQ}$ can contain arbitrary DSQs and do not necessarily have to be the set of DSQs from delta decompositions of a set of queries, is NP-hard. Our reduction is from the maximum common induced subgraph problem [40]. Resolving if the problem is easier when the DSQs are delta decompositions of a set of queries, which is a property that holds in our setting, is left for future work.

We end this section with two notes. First, each query Q can be decomposed in multiple ways. For example, an alternative decomposition for our example diamond query could have started with $DSQ_1 : a_1 \xrightarrow{o} a_2, a_1 \xrightarrow{o} a_3, a_2 \xrightarrow{\delta} a_4, a_3 \xrightarrow{o} a_4$ instead of $a_1 \xrightarrow{\delta} a_2, a_1 \xrightarrow{o} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4$. These decompositions are not identical. We studied the effects of different decompositions but found that they make little difference in performance. So we take an arbitrary decomposition of each query Q and do not consider and optimize alternative decompositions. Second, each delta query that contains a `new` query edge can be further decomposed into smaller delta subgraph queries algebraically. For example, $DSQ_2 = a_1 \xrightarrow{n} a_2, a_1 \xrightarrow{\delta} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4$ above is algebraically equivalent to $DSQ_{21} = a_1 \xrightarrow{\delta} a_2, a_1 \xrightarrow{\delta} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4 \cup DSQ_{22} = a_1 \xrightarrow{o} a_2, a_1 \xrightarrow{\delta} a_3, a_2 \xrightarrow{o} a_4, a_3 \xrightarrow{o} a_4$, because `new` edges are unions of `delta` and `old` edges. These further decompositions, which we call *expanded delta query decompositions* can allow for more sharing opportunities, because the query edges in delta queries have only two labels (`delta` and `old`) instead of three (`delta`, `old`, and `new`). This can lead to more isomorphic sub-queries but this expansion leads to many more delta queries, and we observed in practice that this does not lead to significant performance improvements in our query sets.

4.2 Greedy Optimizer

One approach to optimizing this problem is to find the lowest i-cost QVO for each delta subgraph query in $\bar{Q}_{DSQ}$ and then merge these individual plans in a combined plan. This approach often generates reasonably good plans and will form one of our baseline optimizers in our evaluations. However, this approach does not directly search for sharing opportunities or optimize for the total i-cost of the combined plan. To do so, we adopt a greedy approach. We start with an empty combined plan CP. In each iteration, the algorithm goes through each QVO of each delta subgraph query in $\bar{Q}_{DSQ}$ and finds the pair $<qvo^*, dsq^*>$ with the minimum *additional cost* to CP (ties are broken randomly). This fixes the QVO of dsq^* to be qvo^*, and we merge the plan P^* induced by $<qvo^*, dsq^*>$ to CP. The minimum additional cost is the extra i-cost introduced by the new operators added to CP. We remove dsq^* from $\bar{Q}_{DSQ}$ and repeat this greedy step until $\bar{Q}_{DSQ}$ is empty. The additional cost of a $<qvo, dsq>$ pair is computed as follows. Let the logical plan induced by $<qvo, dsq>$ be P. Recall that P is a linear plan that starts with Scan followed by a series of E/I operators. Then starting from the last operator in P and going to previous operators, we find the first operator $o_i \in P$, that produces matches isomorphic to an operator o'_i in CP. If such o'_i exists, then the suffix operators after o_i in P are added to CP as suffix operators to o'_i. The additional cost of $<qvo, dsq>$ is the sum of the costs of these suffix operators.

Catalogue Generation: Recall from above that we adopt the same catalogue-based cost and cardinality estimation technique we use for one-time queries and do not differentiate between `delta`, `old`, and `new` query edges that are used in the operators of the combined plan.



(a) Combined plan sharing delta plan operators.

(b) Combined plan sharing partial intersection work.

Fig. 9. Part of a combined plan example shows the transformation from sharing delta plan operators to also sharing partial intersection work.

However, unlike one-time subgraph queries, where the catalogue entries are limited to entries with at most h query vertices, when optimizing continuous queries, we generate all necessary catalogue entries for a given query set $\bar{Q}$. This is because continuous queries are long running and the number of added queries is often small, so even if some of the queries are large in size, the number of necessary entries for a fixed small number of queries is not large. In contrast, a system needs a catalogue that can be used to answer arbitrary one-time queries. For example, if a query $Q \in \bar{Q}$ has six query vertices, we generate a catalogue entry for extending each five-vertex sub-query Q' of Q to Q. This is because our greedy optimizer computes the cost of each possible QVO for each delta subgraph query, so each of these entries is necessary. As before, each entry is generated based on sampling.

4.3 Partial Intersection Sharing

Consider two plans, P_1 and P_2 for two delta subgraph queries corresponding to two QVOs. Suppose that P_1 and P_2 compute isomorphic sub-queries until their level i operators but their level $i+1$ operators' outputs are not isomorphic. Let o_i^1 and o_i^2 be the level i operators of P_1 and P_2, respectively, and o_{i+1}^1 and o_{i+1}^2 be their level $i+1$ operators. Our greedy optimizer will share computation across o_i^1 and o_i^2 but not o_{i+1}^1 and o_{i+1}^2. Even though the $(i+2)$-matches produced by o_1^{i+1} o_2^{i+1} may not be isomorphic, so the full intersections performed by these operators are not identical, part of the intersections performed by these operators might be common. Consider the combined plan shown in Figure 9(a), which represents the combination of three plans for three delta subgraph queries. The third-level operators of these plans perform different intersections that partially match. Specifically, all of these three operators take as input 2-edge paths, say, $u \rightarrow v \rightarrow w$, and intersect u's `new` forward and w's `new` backward adjacency lists but the middle and right operators also intersect a third adjacency list. This gives an opportunity to partially share the common two-way intersection in one operator and complete the remaining intersections in other operators. Partial intersection sharing is especially important for increasing the amount of computation shared at the last-level operators, because unless two delta subgraph queries are completely isomorphic, or one is contained in another, the last-level operators of individual plans for delta subgraph queries cannot be shared. As we show in our evaluations, in some workloads, the majority of the work done can be in the last-level and sharing partial intersection work yields significant benefits.

We implement partial intersection sharing through two variants of the E/I operator:

E/I-Partial: Similarly to E/I, intersects two or more adjacency lists and outputs the result either as an adjacency list to the next E/I-Remaining operator and as regular output tuples to next E/I and E/I-Partial operators.

E/I-Remaining: Given an intersection result from an E/I-Partial operator, intersects it further with one or more adjacency lists.

Figure 9(b) shows an example plan that partially shares computation in the last-level operators of the plan in Figure 9(a).

Our optimizer searches for partial intersections as a post processing step once a combined plan CP with full operator sharing is generated.[3] Specifically, starting from the lowest-level Scan operator, we iterate over each operator in CP at levels 1, 2, so on and so forth, up to the last level. For each operator o_j, we iterate over, we inspect the next-level operators in CP that extend the outputs of o_j. Let S_j be that set of successor operators. We enumerate all partial intersections ALD_{PI} that at least two operators in S_j share and calculate how much i-cost reduction sharing ALD_{PI} would yield. The amount of i-cost reduction is the multiplication of: (1) partial matches of o_j; (ii) sum of the estimated lengths of the adjacency lists in ALD_{PI}; and (iii) the number of operators that share ALD_{PI} minus 1. Let ALD^*_{PI} be the partial intersection that reduces the most i-cost. We add an E/I-Partial operator o_{part} that takes as input the outputs of o_j[4] and intersects ALD^*_{PI}. Then we remove ALD^*_{PI} from each operator o' in S_j that contain ALD^*_{PI} and replace o' with an E/I-Remaining operator.

5 SYSTEM IMPLEMENTATION

We implemented our new techniques inside GraphflowDB. GraphflowDB is a single machine, multi-threaded, main memory graph DBMS implemented in Java. The system supports a subset of the Cypher language [53]. We extended the Cypher language with a `CONTINUOUS` clause to allow registering continuous subgraph queries. One-time queries and continuous queries are optimized respectively by our dynamic programming and greedy optimizers. Our optimizers share significant code. In particular, they use a single plan enumerator that can be configured to either generate one-time plans that contain both E/I and HashJoin operators or only WCO plans that start by scanning `delta` edges. Our optimizers also use the same catalogue for cost and cardinality estimations. In the rest, we give implementation details about several other components of the system.

Storage: We index both the forward and backward adjacency lists and store them in sorted vertex ID order. Adjacency lists are by default partitioned by the edge labels or by the labels of neighbour vertices if a single edge label exists. With this partitioning, we can quickly access the edges of nodes matching a particular edge label and destination vertex label, allowing us to perform filters on labels very efficiently. Upon an update to the graph, we create the *new* adjacency lists of the vertices whose adjacency lists are changing. These are reused from a pool of existing lists to avoid Java object creations. Once all delta queries are executed, we copy the data of the `new` adjacency

[3]Alternatively, partial intersection overlaps can be searched as part of our greedy optimizer. We implemented the post-processing approach because of its simplicity.

[4]Note that o_j may have been replaced with an E/I-Remaining operator o'_j in the previous iteration, in which case o_{part} takes as input o'_j's output.

Table 9. Datasets Used

Domain	Name	Nodes	Edges
Social	Epinions (Ep)	76K	509K
	LiveJournal (LJ)	4.8M	69M
	Twitter (Tw)	41.6M	1.46B
Web	BerkStan (BS)	685K	7.6M
	Google (Go)	876K	5.1M
Product	Amazon (Am)	403K	3.5M
Citation	Patent (Pa)	3.7M	16.5M

lists to the `old` adjacency lists to update them and reset the adjacency lists in the pool for the next batch update. All `delta` edges are kept in a fixed forward array that is also reused across batches.
Query Executor: Our query plans follow a Volcano-style plan execution [22]. Each plan P has one final Sink operator, which connects to the final operators of all branches in P. The execution starts from the Sink operator and each operator asks for a tuple from one of its children until a Scan starts matching an edge. In adaptive parts of one-time plans, an operator o_i may be called upon to provide a tuple from one of its parents, but due to adaptation, provide tuples to a different parent. We note that our executor can be improved using query compilation techniques [45] or SIMD instructions for intersecting sorted neighbour ID lists [2, 36]. These techniques are complementary to our work.
Parallelization: We implemented a work-stealing-based parallelization technique. Let w be the number of threads in the system. We give a copy of a plan P to each worker and workers steal work from a single queue to start scanning ranges of edges in the Scan operators. Threads can perform extensions in the E/I operators without any coordination. Hash tables used in Hash-Join operators are partitioned into $d{>}{>}w$ many hash table ranges. When constructing a hash table, workers grab locks to access each partition but setting $d{>}{>}w$ decreases the possibility of contention. Probing does not require coordination and is done independently.

6 EVALUATION

In this section, we demonstrate the efficiency of the plans that our one-time and continuous query optimizers generate. We begin in Section 6.1 by describing the hardware and the datasets we use in our experiments. Sections 6.2 and 6.3 then present our experiments for one-time and continuous queries, respectively. We refer readers to our supplementary appendix for several additional experiments throughout the section.

6.1 Setup

6.1.1 Hardware. We use a single machine that has two Intel E5-2670 @2.6 GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical cores. Except for our scalability experiments in Section 6.2.5, we use only one physical core. We set the maximum size of the JVM heap to 500 GB and keep the default minimum heap size of the JVM. We ran each experiment twice, one to warm-up the system and recorded measurements for the second run.

6.1.2 Datasets. The datasets we use are in Table 9.[5] Our datasets differ in several structural properties: (i) size; (2) how skewed their forward and backward adjacency lists distribution is; and (3) average clustering coefficients, which is a measure of the cyclicity of the graph, specifically

[5]We obtained the graphs from Reference [37] except for the Twitter graph, obtained from Reference [33].

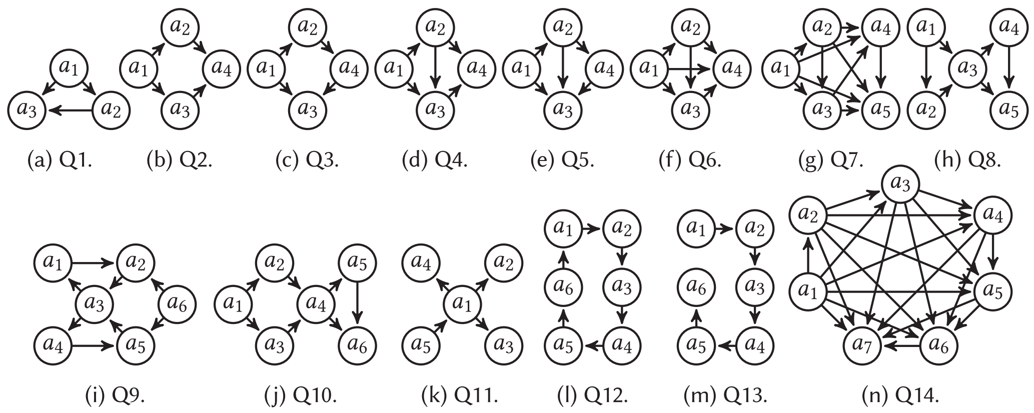


Fig. 10. Subgraph queries used for evaluations.

the amount of cliques in it. The datasets also come from a variety of application domains: social networks, the web, and product co-purchasing. For our one-time query experiments, each dataset catalogue was generated with z = 1,000 and h = 3 except for Twitter, where we set h = 2. For our continuous query experiments, as we discussed in Section 4.2, we generate all relevant catalogue entries.

6.1.3 Queries Notation. Our datasets and queries are not labelled by default, and we label them randomly as done in prior work [9, 24]. For a subgraph query Q or a query set QS, we use the notation Q_i and QS_i, respectively, to refer to evaluating Q and QS on a dataset for which we randomly generate a label l on each edge, where $l \in \{l_1, l_2, \ldots, l_i\}$. For example, evaluating Q_2 on Amazon indicates randomly adding one of two possible labels to each data edge in Amazon and query edge on Q. If a query is unlabelled, then we simply refer to it as Q.

6.2 One-time Query Optimizer Evaluations

In these experiments, we aim to answer five questions relating to one-time subgraph query optimization: (1) How good are the plans our optimizer picks? (2) Which type of plans work better for which queries? (3) How much benefit do we get from adapting QVOs at runtime? (4) How do our plans and processing engine compare against EmptyHeaded (EH), which is the closest to our work and the most performant baseline we are aware of? (5) How do our plans compare against prior work titled "Flexible Caching in Trie Joins" [28], which is another algorithm that extends the worst-case optimal **Leapfrog TrieJoin (LFTJ)** algorithm with caching [66]? We also tested the scalability of our single-threaded and parallel implementation on our largest graphs LiveJournal and Twitter. Finally, for the completeness of our study, in Appendix D, we compare our plans on big queries against the subgraph matching algorithm CFL [10].

For the experiments in this section, we used the 14 queries shown in Figure 10, which contain both acyclic and cyclic queries with dense and sparse connectivity with up to 7 query vertices and 21 query edges. To give a sense of the scale, we report the number of output tuples of the unlabeled versions of these queries in Table 10. We use both unlabeled and labeled versions of these queries. When we put labels, these numbers will naturally decrease depending the number of query edges each query has and the number of labels we use. The majority of these queries are obtained from real applications and from the literature. For example, queries Q1 and Q2 are used in Reference [2] and Reference [6], queries Q2–Q7 are used in Reference [38] and Q12 is used in Reference [56].

Table 10. The Number of Output Tuples for Unlabeled Versions of the Queries in Figure 10 on Amazon (Am), Epinions (Ep), and Google (Go)

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
Am	11.6M	118.2M	110.0M	59.0M	64.8M	37.8M	118.9M
Go	28.2M	375.3M	358.4M	239.9M	295.8M	217.0M	2.0B
Ep	3.6M	326.3M	305.0M	87.0M	100.3M	32.0M	320.6M
	Q8	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
Am	558.7M	3.1B	5.8B	3.1B	4.5B	30.2B	907.3M
Go	3.9B	42.5B	61.5B	173.1B	34.2B	266.4B	256.6B
Ep	12.5B	262.1B	1125.2B	7865.1B	1544.5B	39502.0B	32.9B

6.2.1 Plan Suitability For Different Queries and Optimizer Evaluation. To evaluate how good are the plans our optimizer generates, we compare the runtime of plans we pick against a query's plan spectrum, i.e., the set of all plans enumerated by GraphflowDB for the query. This also allows us to study which types of plans are suitable for which queries. We generated plan spectrums of queries $Q1$–$Q8$ and $Q11$–$Q13$ on Amazon without labels, Epinions with 3 labels, and Google with 5 labels. The spectrums of $Q12$ and $Q13$ on Epinions took a prohibitively long time to generate and are omitted. Figure 11 presents our spectrums for $Q1$–$Q8$ and $Q11$. Figure 12 presents our spectrums for $Q12$ and $Q13$. Each circle in the figures is the runtime of a plan and $\times$ is the plan our optimizer picks. Throughout these experiments, we use the term "optimal plan" to refer to the executed plan with the lowest runtime, i.e., the plan corresponding to the lowest circle in our plan spectrum charts.

We first observe that different types of plans are more suitable for different queries. The main structural properties of a query that govern which types of plans will perform well are how large and how cyclic the query is. For cliquelike queries, such as $Q5$, and small cycle queries, such as $Q3$, best plans are WCO. On acyclic queries, such as $Q11$ and $Q13$, BJ plans are best on some datasets and WCO plans on others. On acyclic queries WCO plans are equivalent to left deep BJ plans, which are worse than bushy BJ plans on some datasets. Finally, hybrid plans are best plans for queries that contain small cyclic structure that do no share edges, such as $Q8$.

Our most interesting query is $Q12$, which is a 6-cycle query. $Q12$ can be evaluated efficiently with both WCO and hybrid plans (and reasonably well with some BJ plans). The hybrid plans first perform binary joins to compute 4-paths, and then extend 4-paths into 6-cycles with an intersection. Figure 2 from Section 1 shows an example of such hybrid plans. These plans do not correspond to the GHDs in EH's plan space. On the Amazon graph, one of these hybrid plans is optimal and our optimizer picks that plan. On the Google graph our optimizer picks an efficient BJ plan although the optimal plan is WCO.

Our optimizer's plans were broadly close to optimal across our experiments. Specifically, our optimizer's plan was optimal in 15 of our 31 spectrums, was within 1.4× of the optimal in 21 spectrum and within 2× in 28 spectrums. In two of the three cases we were more than 2× of the optimal, the absolute runtime difference was in sub-seconds. Ignoring queries whose plans generally ran in sub-second latency, there was only one experiment in which our plan was not close to the optimal plan, which is shown in Figure 11(z). Observe that our optimizer picks different types of plans across different types of queries. In addition, as we demonstrated with $Q12$ above, we can pick different plans for the same query on different datasets ($Q8$ and $Q13$ are other examples).

Although we do not study query optimization time in this article, our optimizer generated a plan within 331 ms in all of our experiments except for $Q7_5$ on Google, which took 1.4 s.

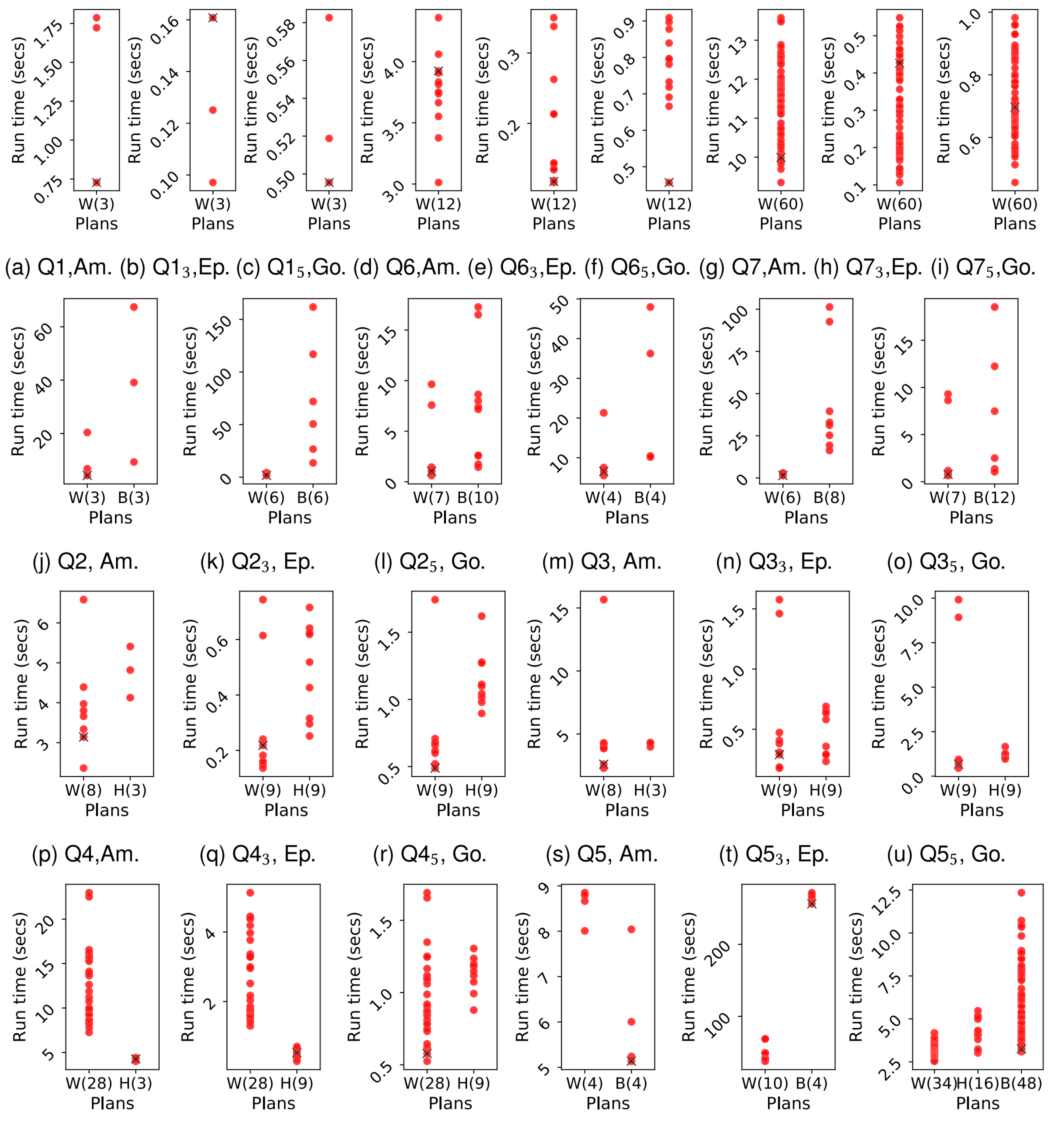
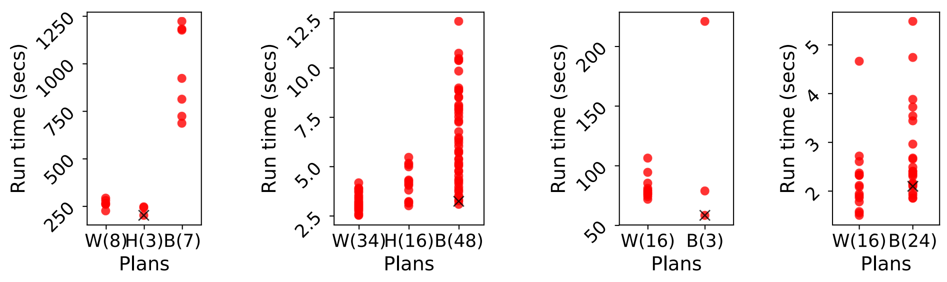


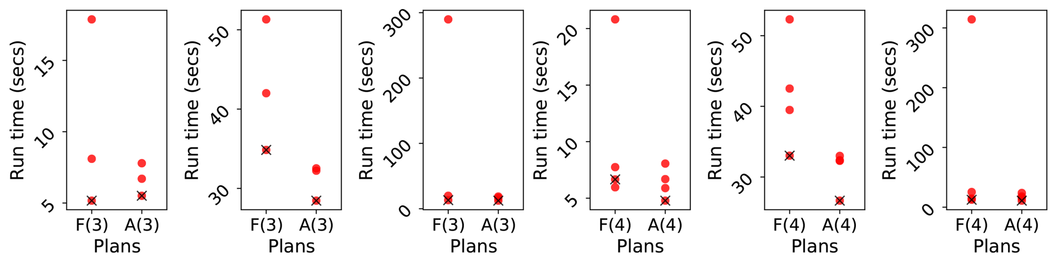
Fig. 11. Runtime (seconds) of the set of all plans enumerated by GraphflowDB for queries Q1–Q8 and Q11. “x” specifies the plan picked by GraphflowDB.

6.2.2 *Adaptive WCO Plan Evaluation.* To understand the benefits we get by adaptively picking QVOs, we studied the spectrums of WCO plans of $Q2$, $Q3$, $Q4$, $Q5$, and $Q6$, and hybrid plans for $Q10$ on Epinions, Amazon and Google graphs. These are the queries in which our DP optimizer's fixed plans contained a chain of two or more E/I operators (so we could adapt them). The spectrum of $Q10$ on Epinions took a prohibitively long time to generate and is omitted. Figure 13 shows the 17 spectrums we generated. In the case of $Q2$, $Q3$, and $Q4$, selecting QVOs adaptively overall improves the performance of every fixed plan. For example, the fixed plan our DP optimizer picks for $Q3$ on Epinions improves by 1.2× but other plans improve by up to 1.6×. $Q10$'s spectrum for

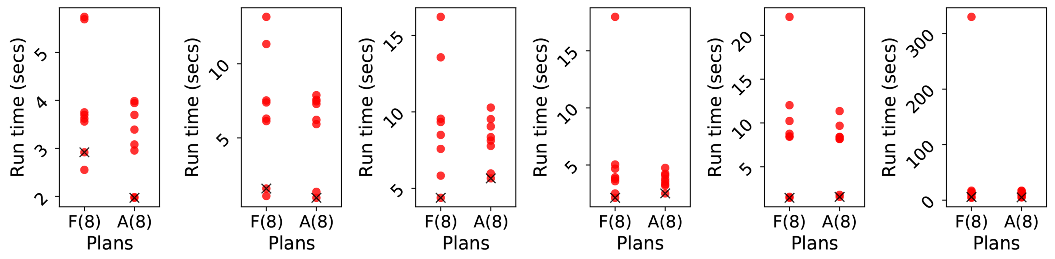


(a) Q12, Am. (b) Q12$_5$, Go. (c) Q13, Am. (d) Q13$_5$, Go.

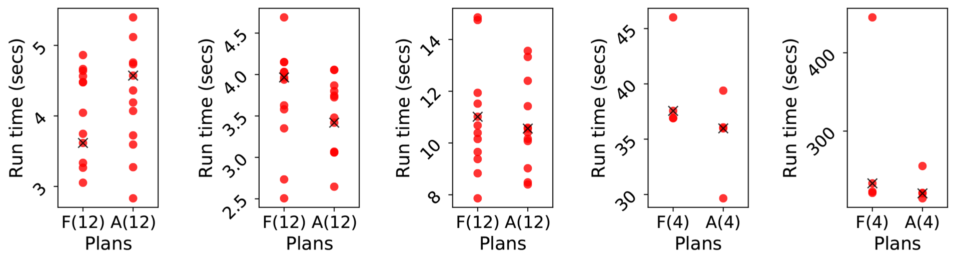
Fig. 12. Runtime (seconds) of the set of all plans enumerated by GraphflowDB for queries Q12 and Q13. “x” specifies the plan picked by GraphflowDB.



(a) Q2, Am. (b) Q2, Ep. (c) Q2, Go. (d) Q3, Am. (e) Q3, Ep. (f) Q3, Go.



(g) Q4, Am. (h) Q4, Ep. (i) Q4, Go. (j) Q5, Am. (k) Q5, Ep. (l) Q5, Go.



(m) Q6, Am. (n) Q6, Ep. (o) Q6, Go. (p) Q10, Am. (q) Q10, Go.

Fig. 13. Runtime (seconds) of the set of adaptive plans enumerated by GraphflowDB for queries Q2–Q6 and Q10. “x” specifies the plan picked by GraphflowDB.

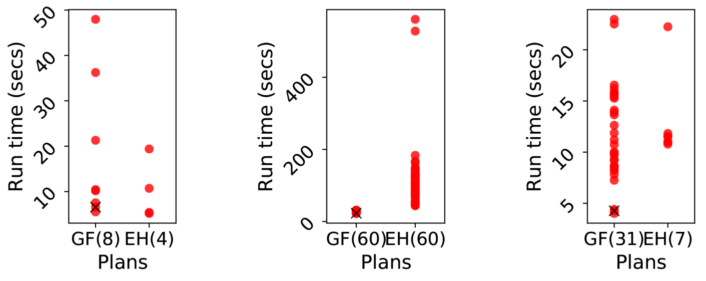
hybrid plans are similar to $Q3$ and $Q4$'s. Each hybrid plan of $Q10$ computes the diamonds on the left and triangles on the right and joins on a_4. Here, we can adaptively compute the diamonds (but not the triangles). Each fixed hybrid plan improves by adapting and some improve by up to 2.1×. On $Q5$ most plans' runtimes remain similar but one WCO plan improves by 4.3×. The main benefit of adapting is that it makes our optimizer more robust against picking bad QVOs. Specifically, the deviation between the best and worst plans are smaller in adaptive plans than fixed plans.

The only exception to these observations is $Q6$, where several plans' performances get worse, although the deviation between good and bad plans still become smaller. We observed that for cliques, the overheads of adaptively picking QVOs is higher than other queries. This is because: (i) cost re-evaluation accesses many actual adjacency list sizes, so the overheads are high; and (ii) the QVOs of cliques have similar behaviors: each one extends edges to triangles, then four cliques, etc.), so the benefits are low.

6.2.3 EmptyHeaded (EH) Comparisons. EH is one of the most efficient systems for one-time subgraph queries and its plans are the closest to ours. Recall from Section 1 that EH has a cost-based optimizer that picks a GHD with the minimum width, i.e., EH picks a GHD with the lowest AGM bound across all of its sub-queries. This allows EH to often (but not always) pick good decompositions. However: (1) EH does not optimize the choice of QVOs for computing its sub-queries; and (2) EH cannot pick plans that have intersections after a binary join, as such plans do not correspond to GHDs. In particular, the QVO that EH picks for a query Q is the lexicographic order of the variables used for query vertices when a user issues the query. EH's only heuristic is that QVOs of two sub-queries that are joined start with query vertices on which the join will happen. Therefore by issuing the same query with different variables, users can make EH pick a good or a bad ordering. This shortcoming has the advantage that by making EH pick good QVOs, we can show that our orderings also improve EH. The important point is that EH does not optimize for QVOs. We therefore report EH's performance with both "bad" orderings (EH_b) and "good" orderings (EH_g). For good orderings, we use the ordering that GraphflowDB picks. For bad orderings, we generated the spectrum of plans in EH (explained momentarily) and picked the worst-performing ordering for the GHD EH picks. For our experiments, we ran $Q3$, $Q5$, $Q7$, $Q8$, $Q9$, $Q12$, and $Q13$ on Amazon, Google, and Epinions. We first explain how we generated EH spectrums and then present our results.

EH Spectrums: Given a query, EH's query planner enumerates a set of minimum width GHDs and picks one of these GHDs. To define the plan spectrum of EH, we took all of these GHDs, and by rewriting the query with all possible different variables, we generate all possible QVOs of the sub-queries of the GHD that EH considers. Figure 14 shows a sample of the spectrums for Q3 and Q7 on Amazon and for Q8 on Epinions along with GraphflowDB's plan spectrum (including WCO, BJ, and hybrid plans) for comparison. For Q9, Q12, and Q13, we could not generate spectrums as every EH plan took more than our 30-minute time limit. For Q7, both GraphflowDB and EH generate only WCO plans. For Q8, EH generates two GHDs (two triangles joined on a_3) whose different QVOs give four different plans for a total of eight. One of the plans in the spectrum is omitted as it had memory issues. We note that out of these queries, Q9 was the only query for which EH generated two different decompositions (ignoring the QVOs of sub-queries) but neither decomposition under any QVO ran within our time limit on our datasets.

GraphflowDB vs. EmptyHeaded Comparisons: We ran our queries on GraphflowDB with adapting off. To compare, we ran EH's plan with good and bad QVOs for $Q3$, $Q5$, $Q7$, $Q8$ (recall no EH plan ran within our time limit for $Q9$, $Q12$, and $Q13$). We repeated the experiments once with no labels and once with two labels. Table 11 shows our results. Except for $Q1$ on Google and $Q8_2$ on Amazon where the difference is only 500 ms and 200 ms, respectively. GraphflowDB is always



(a) $Q3$, Amazon. (b) $Q7$, Epinions. (c) $Q8$, Amazon.

Fig. 14. Runtime (seconds) of the set of all plans enumerated by EmptyHeaded (EH) compared with those enumerated by GraphflowDB (GF). "x" specifies the plan picked by GraphflowDB.

Table 11. Runtime (seconds) of GraphflowDB (GF) and EmptyHeaded with Good Orderings (EH$_g$) and Bad Orderings (EH$_b$)

		Q1	Q3	Q3$_2$	Q5	Q5$_2$	Q7	Q7$_2$	Q8	Q8$_2$	Q9	Q9$_2$	Q12	Q12$_2$	Q13	Q13$_2$
Am	EH$_b$	1.0	19.0	3.4	47.1	9.2	91.4	11.6	22.2	1.8	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	EH$_g$	0.6	5.4	1.3	3.3	1.5	21.2	1.7	10.6	1.4	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	GF	0.6	5.5	2.1	1.9	0.8	9.5	0.9	5.1	2.0	24.7	2.4	209.2	14.8	48.0	11.3
Go	EH$_b$	1.9	444.5	42.6	401.1	77.6	1.04K	23.4	66.6	16.0	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	EH$_g$	1.4	12.0	2.1	11.3	2.3	107.3	4.8	35.8	3.0	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	GF	2.6	14.0	4.0	5.9	2.1	48.8	3.3	17.0	4.5	236.2	6.9	510.6	73.8	1.44K	70.1
Ep	EH$_b$	0.5	42.7	6.5	64.5	11.4	560.7	2.9	1.01K	22.0	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	EH$_g$	0.2	26.6	1.7	3.5	0.9	45.7	0.8	117.2	7.0	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*	*Mm*
	GF	0.4	28.1	4.6	1.5	0.6	23.7	1.2	37.5	5.4	865.3	26.1	*TL*	*TL*	95.0k	2.35k

TL indicates the query did not finish in 48 hrs. *Mm* indicates running out of memory.

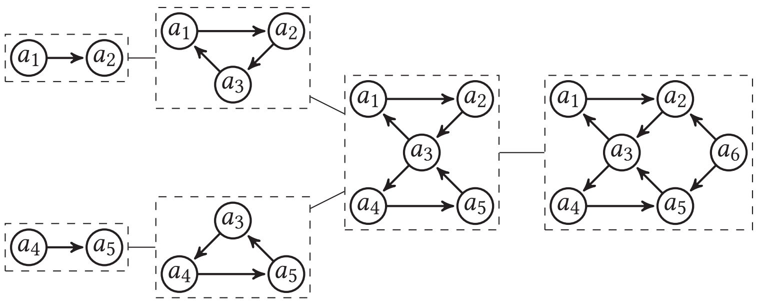
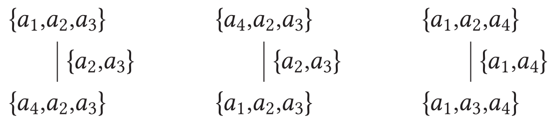


Fig. 15. Plan (drawn horizontally) with seamless mixing of intersections and binary joins on Q9.

faster than EH$_b$, where the runtime is as high as 68× in one instance. The most performance difference is on $Q5$ and Google, for which both our system and EH use a WCO plan. When we force EH to pick our good QVOs, on smaller size queries EH can be more efficient than our plans. For example, although GraphflowDB is 32× faster than EH$_b$ on $Q3$ Google, it is 1.2× slower than EH$_g$. Importantly EH$_g$ is always faster than EH$_b$, showing that our QVOs improve runtimes consistently in a completely independent system that implements WCO join-style processing.

We next discuss $Q9$, which demonstrates again the benefit we get by seamlessly mixing intersections with binary joins. Figure 15 shows the plan our optimizer picks on $Q9$ on all of our datasets. Our plan separately computes two triangles, joins them, and finally performs a 2-way intersection.



(a) TD_{2_1} for Q2. (b) TD_{2_2} for Q2. (c) TD_{2_3} for Q2.

Fig. 16. Example of CTJ's tree decompositions (TDs) for Q2.

This execution does not correspond to the GHD-based plans of EH, so is not in the plan space of EH. Instead, EH considers two GHDs for this query but neither of them finished within our time limit.

6.2.4 CTJ Comparisons. Similarly to Generic Join, LFTJ [66] is a WCO join algorithm that evaluates join queries one attribute at a time, so evaluates subgraph queries one query vertex at a time. Therefore the same optimization problem of picking a good QVO arises when using LFTJ. An important advantage of these algorithms is their small memory footprints. For example, when executed in a purely pipelined fashion, LFTJ does not require memory to keep large intermediate results. Reference [28] observes that by keeping a cache of certain intermediate results and reusing these results, LFTJ's performance can be improved. For example, consider evaluating the "two-triangle" query $Q8$ and using the QVO $a_1a_2a_3a_4a_5$. Note that for each a_3 value, irrespective of the previous a_1 and a_2 values, the same a_4a_5 values would be matched. Therefore, if LFTJ keeps a cache of a_3 to a_4a_5 matches as it extends a_3's, it can save and reuse computation. The algorithm from Reference [28] called CTJ extends LFTJ with caching. This is a more advanced cache than our simple intersection cache and in some queries, gives LFTJ benefits that are similar to using the HashJoin operator in binary or hybrid join plans. For example, consider a hybrid plan for $Q8$ that uses a HashJoin to evaluate $a_3a_4a_5$ triangles on the one side, hashes these on a_3, and then probes this hash table with $a_1a_2a_3$ triangles. The hash table here is similar to CTJ's cache and reuses the computation that was done to compute $a_3a_4a_5$ triangles for different a_3 values.

CTJ generates plans as follows. First CTJ enumerates a set of *ordered tree decompositions* (TDs), which are rooted TDs, whose bags have a particular *preorder* [28]. The adhesion of two parent-child bags is the number of common attributes they have. Then using a set of heuristics, CTJ picks one of these TDs. Specifically, CTJ picks a TD with the minimum value for its largest adhesions, breaking ties with maximizing the number of bags, and then minimizing the sum of adhesions. One of these TDs is picked arbitrarily (say, TD T). Then for T, CTJ defines: (1) a set of *compatible QVOs*; and (2) a caching scheme. Finally, from the compatible QVOs, one is picked using heuristics from another reference, Tributary Join [15]. We explain with an example.

Example 6.1. Consider the Q2 diamond query from Figure 10(b). For this query, there are several TDs that CTJ can pick according to its heuristics. Three of these are shown in Figure 16 (a), (b), and (c) as they have the same adhesion sizes (which is minimum) and the other tie-break metrics. Suppose CTJ picks TD_{1_2}. A preorder traversal on TD_{1_2} orders the bags as follows: (1) $\{a_1, a_2, a_3\}$; and (2) $\{a_4, a_2, a_3\}$. Next, CTJ removes from each bag the query vertices in the adhesions found in the root-to-bag path, which yields the ordering: (1) $\{a_1, a_2, a_3\}$; and (2) $\{a_4\}$. These are the variables *owned* by each bag. The compatible QVOs are those that order the QVO for each bag and concatenating these orderings from root to the leaf. Of these, CTJ uses another cost called Tributary Join's [15] cost model to choose the QVO in each bag. We explain Tributary Join momentarily. Suppose the algorithm picks the QVO $a_1a_2a_3a_4$. For each non-root bag B in TD, CTJ adds a cache to LFTJ. Suppose the parent of B is p in TD. The cache has (i) as key the query vertices in the adhesion of p and B and (ii) as value the query vertices "owned" by B. For example, the cache for the QVO $a_1a_2a_3a_4$ for TD_{1_2} will be from key:$\{a_2, a_3\}$ to value:a_4.

The focus of CTJ and Reference [28] is to control the memory consumption of LFTJ to increase its performance and not on how to pick TDs or optimize the QVO selection. For example, while CTJ can avoid storing the complete joins of subqueries, our binary join and hybrid plans do not have mechanisms to control for memory. In our setting, we assume that the HashJoin operators have enough memory to create their hash tables. Instead, our work focuses primarily on efficient plan selection for queries. There are several important differences between our optimized plans and the plans CTJ uses:

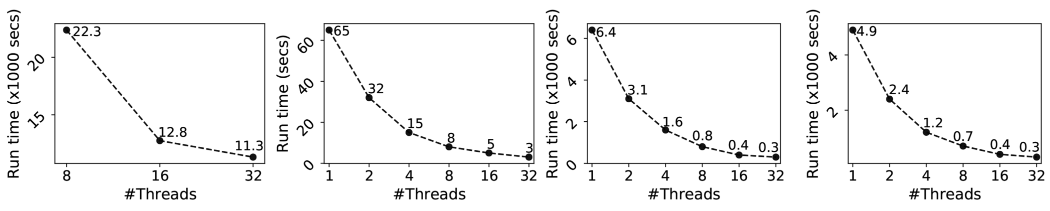
1. On some queries, the heuristics that CTJ uses to pick a TD cannot distinguish between efficient TDs from inefficient ones. For example, consider the diamond query Q2 from Figure 10(b). CTJ's heuristics will not be able to tie break between TD_{2_1}, TD_{2_2}, and TD_{2_3} in Figure 16(a), (b), and (c), respectively, and pick one of these arbitrarily, which yield different QVOs (and caching schemes). In fact, in the code provided by the users, we noticed that similar to EH, we can make CTJ pick different TDs and final QVOs, with very different runtimes. For instance, TD_{2_1} and TD_{2_2} on Google lead to runtimes 86.6 s and 806.2 s, respectively. The difference in runtime is mainly due to a difference in the number of intermediate results, which are 72.94M for TD_{2_1} and 1.38B for TD_{2_2}. Yet CTJ's optimizer does not differentiate between these two TDs and their final QVOs. Instead, our i-cost-based model can differentiate between these QVOs.
2. Once a TD has been picked, CTJ uses Tributary Join's [15] technique to pick a QVO within each bag. Tributary Join studies picking the QVO for LFTJ algorithm in the context of joining multiple relational tables and picks the QVO based on the distinct values in the attributes of the relations. This heuristic however is not designed for self-join queries as in our subgraph queries, where attributes will have the same number of distinct values, specifically $|V|$ (assuming every vertex in an input graph has an incoming and outgoing edge). Recall that in subgraph queries, each binary $\mathsf{E}(a_i, a_j)$ relation is a replica of the edges of the input graph $G(V, E)$. Note that in our evaluations we either use unlabeled queries or add random edge labels to the edges of our datasets and queries. This effectively partitions the edge table E into multiple tables, but any differences in the distinct values across these partitions would be due to random assignment.
3. On some queries CTJ's plans do not benefit from caching results of sub-queries larger than a single query edge, due to the heuristics CTJ uses to pick TDs. For example, for a path query, say, $Q13$, CTJ considers TD's in which each bag consists of a single query, edge. Since CTJ caches the results of a single bag, only results of a single query edge, so adjacency lists can be cached. This contrasts with traditional binary join plans that can cache sub-paths.

We compared GraphflowDB to CTJ on default versions of queries Q1 to Q14 on Amazon, Google, and Epinions. We obtained the CTJ code from the authors of Reference [28]. Recall that CTJ's main focus is in controlling the cache size. We observed that we obtain the best runtime numbers when we run CTJ with an unbounded cache size, which implies that CTJ caches every key-value between each bag. Table 12 shows our results. As we explained above, CTJ can pick between multiple different TDs and QVOs depending on how the query is written. In Table 12, we report the best runtime for CTJ for each query after writing the attributes of the query in every lexicographic order. As shown in the table, GraphflowDB outperforms the implementation we obtained for CTJ across these queries, varying from competitive performances to differences that are two orders of magnitude in runtime. We note that it is not possible to a very controlled comparison here, because the

Table 12. Runtime (Seconds) of GraphflowDB (GF) and CTJ

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
Am	CTJ	5.1	38.9	41.5	22.6	21.0	18.6	61.1
	GF	0.6(**8.5x**)	4.7(**8.3x**)	5.5(**7.6x**)	2.0(**11.3x**)	1.9(**11.1x**)	3.3(**5.6x**)	9.5(**6.4x**)
Go	CTJ	15.3	82.7	86.6	59.4	55.7	64.0	464.1
	GF	2.6(**5.9x**)	12.3(**6.7x**)	12.0(**7.2x**)	4.9(**12.1x**)	5.9(**9.4x**)	8.6(**7.4x**)	48.8(**9.5x**)
Ep	CTJ	2.3	88.4	94.7	10.5	9.5	27.5	329.2
	GF	0.4(**5.8x**)	31.5(**2.8x**)	26.6(**3.6x**)	1.5(**7.2x**)	1.5(**6.3x**)	3.3(**8.3x**)	23.7(**13.9x**)
		Q8	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
Am	CTJ	94.8	142.1	2256.5	184.2	878.5	456.0	639.6
	GF	5.1(**18.6x**)	56.3(**2.5x**)	20.8(**108.5x**)	6.8(**27.1x**)	209.2(**4.2x**)	48.0(**9.5x**)	125.0(**5.12x**)
Go	CTJ	606.3	574.4	94084.1	8055.1	3048.5	2165.4	67049.9
	GF	17.0(**35.7x**)	303.9(**1.9x**)	135.9(**692.3x**)	214.6(**37.5x**)	510.6(**6.0x**)	1440.0(**1.5x**)	5348.7(**1.5x**)
EP	CTJ	3251.1	1618.8(**1.5x**)	158274.2	TL	TL	145K	95027.4
	GF	37.5(**86.7x**)	2384.8	1908.7(**82.9x**)	12852.5	TL	95027.4(**1.5x**)	3373.1(**13.0**)

TL indicates the query did not finish in 48 hrs.



(a) Q1, Twitter. (b) Q1, LiveJournal. (c) Q2, LiveJournal. (d) Q14, Google.

Fig. 17. Scalability experiments.

implementations of GraphflowDB plans and CTJ are very different, e.g., the implementations use different programming languages and data organization. However, the differences we discussed above contribute to these runtime differences. For example, the plan that CTJ uses for Q13, which is a path query and where CTJ does not benefit from caching, generates 3.43B many intermediate tuples on Amazon. In contrast, GraphflowDB's plan hashes on a_4 and generates only 0.39B many intermediate tuples.

6.2.5 Scalability Experiments. We next demonstrate the scalability of GraphflowDB on larger datasets and across a larger number of physical cores. The goal of our experiments is to demonstrate that when more cores are available, our approach can utilize them efficiently. We evaluated $Q1$ on LiveJournal and Twitter graphs, $Q2$ on LiveJournal, and $Q14$, which is a very difficult 7-clique query, on Google. We repeated each query with 1, 2, 4, 8, 16, and 32 cores, except we use 8, 16, and 32 cores on the Twitter graph. Figure 17 shows our results. Our plans scale linearly until 16 cores with a slight slow down when moving 32 cores, which is the maximum number of cores in our hardware. For example, going from 1 core to 16 cores, our runtime is reduced by 13× for $Q1$ on LiveJournal, 16× for $Q2$ on LiveJournal, and 12.3× for $Q14$ on Google.

6.3 Continuous Query Optimizer Evaluations

We next evaluate our optimizer for continuous queries. We aim to answer three main questions: (1) How much benefit do combined plans get from sharing operators and why? (Section 6.3.2), (2) How much benefit do combined plans get from sharing partial intersections and why? (Section 6.3.2), and (3) How good are the plans our optimizer picks? (Section 6.3.3). In Section 6.3.4, we test the

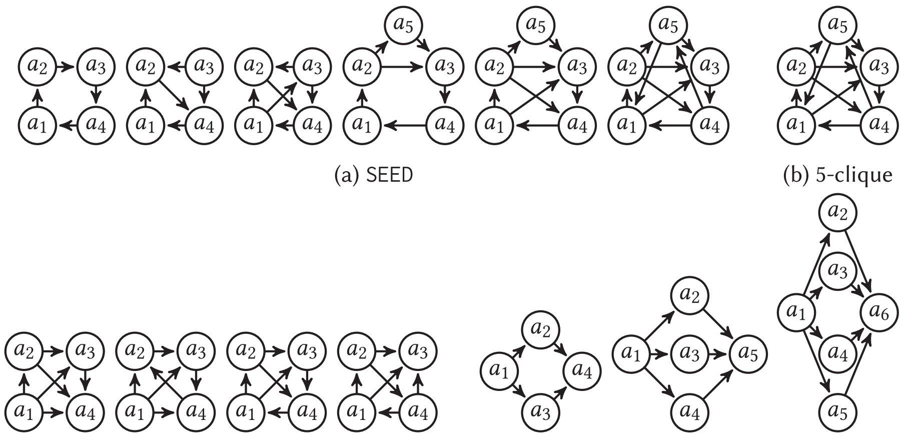

(a) SEED (b) 5-clique (c) 4-cliques (d) MagicRecs

Fig. 18. Queries used for continuous subgraph query evaluations.

scalability of our implementation on a billion-scale Twitter graph. Finally, for completeness, in Appendix E, we compare our approach on single queries against TurboFlux, a recent work on continuous subgraph query evaluation. Although our approaches and implementations are very different, we found GraphflowDB outperforms TurboFlux by at least 3.3× and by up to 117.5× on our queries.

6.3.1 Query and Datasets. We used four different query sets:

- `SEED`: Directed versions of 6 queries from Reference [38] shown in Figure 18(a).
- `MagicRecs`: Diamond queries of Twitter's MagicRecs recommender [23] shown in Figure 18(d).
- `4Cs`: All four unique directed 4-cliques as shown in Figure 18(c).
- `4Cs5C`: `4Cs` set and a 5-clique shown in Figure 18(c) and (b).

Our query sets have structural overlaps across their queries, which is necessary to have sharing opportunities. We use four datasets: Amazon (Am), Google (Go), Epinions (Ep), and Patents (Pt) from Table 9. In our scalability experiments, we will also use the Twitter (Tw) dataset. Note that adding labels on query edges decreases sharing opportunities as delta subgraph queries need to be both structurally isomorphic and have the same query edge labels. To allow for more sharing opportunity across queries in our query sets, we keep the edge labels in query sets homogeneous by labelling them with a single label. Interestingly, as we discuss in Section 6.3.2, adding more labels on datasets, i.e., making the datasets more heterogeneous, increases benefits of sharing.

6.3.2 Benefits of Combined Plans and Partial Intersection Sharing. To evaluate (i) how much benefit is gained from sharing computation across plans (both non-greedily and greedily) and (ii) the benefits of partial intersection sharing, we compared the performance of the plans generated by four optimizers:

- $\mathtt{B}_{ns}$ (for **n**o **s**haring): Picks the lowest i-cost QVO for each delta subgraph query and runs each one separately.[6]

[6]We also adaptively evaluated the delta subgraph queries in B_{ns} but our query sets contain cliquelike cyclic and benefits of adapting these queries were minor (see Section 6.2.2).

Table 13. Runtime (Seconds) of $\mathtt{B}_{ns}$, $\mathtt{B}_s$, $\mathtt{G}_r$, and $\mathtt{G}_{opis}$ on 4Cs, 4Cs5C, SEED, and MagicRec Query Sets

		Runtime	**I-cost**	**Runtime**	**I-cost**	**Runtime**	**I-cost**
		4Cs$_1$		**4Cs$_2$**		**4Cs$_3$**	
Am	$\mathtt{B}_{ns}$	6.29	0.602B (**65%**)	1.33	0.094B (**48%**)	0.72	0.033B (**32%**)
	$\mathtt{B}_s$	5.56 (**1.13x**)	0.478B (**1.26x**)	1.07 (**1.24x**)	0.067B (**1.40x**)	0.60 (**1.20x**)	0.021B (**1.57x**)
	Gr	5.46 (**1.15x**)	0.442B (**1.36x**)	1.01 (**1.32x**)	0.060B (**1.57x**)	0.47 (**1.53x**)	0.017B (**1.94x**)
	$\mathtt{Gr}_p$	5.46 (**1.15x**)	0.442B (**1.36x**)	0.92 (**1.45x**)	0.053B (**1.77x**)	0.46 (**1.57x**)	0.015B (**2.20x**)
Pa	$\mathtt{B}_{ns}$	8.61	0.858B (**29%**)	3.28	0.178B (**12%**)	2.12	0.076B (**7%**)
	$\mathtt{B}_s$	5.23 (**1.65x**)	0.479B (**1.79x**)	1.76 (**1.86x**)	0.081B (**2.20x**)	1.17 (**1.81x**)	0.032B (**2.38x**)
	Gr	4.63 (**1.86x**)	0.423B (**2.03x**)	1.67 (**1.96x**)	0.073B (**2.44x**)	0.97 (**2.19x**)	0.025B (**3.04x**)
	$\mathtt{Gr}_p$	4.28 (**2.01x**)	0.383B (**2.24x**)	1.67 (**1.96x**)	0.073B (**2.44x**)	0.99 (**2.14x**)	0.024B (**3.17x**)
		4Cs5C$_1$		**4Cs5C$_2$**		**4Cs5C$_3$**	
Am	$\mathtt{B}_{ns}$	13.4	1.226B (**35%**)	1.87	0.129B (**6%**)	0.90	0.043B (**1%**)
	$\mathtt{B}_s$	11.8 (**1.14x**)	1.015B (**1.21x**)	1.32 (**1.42x**)	0.084B (**1.54x**)	0.50 (**1.80x**)	0.026B (**1.65x**)
	Gr	11.4 (**1.18x**)	0.979B (**1.25x**)	1.20 (**1.56x**)	0.075B (**1.72x**)	0.50 (**1.80x**)	0.021B (**2.05x**)
	$\mathtt{Gr}_p$	11.4 (**1.18x**)	0.979B (**1.25x**)	1.11 (**1.68x**)	0.071B (**1.82x**)	0.45 (**2.00x**)	0.020B (**2.15x**)
Pa	$\mathtt{B}_{ns}$	9.73	1.015B (**0%**)	4.50	0.217B (**0%**)	2.84	0.094B (**0%**)
	$\mathtt{B}_s$	5.34 (**1.82x**)	0.479B (**2.12x**)	1.66 (**2.71x**)	0.081B (**2.68x**)	0.97 (**2.93x**)	0.032B (**2.94x**)
	Gr	4.43 (**2.20x**)	0.423B (**2.40x**)	1.43 (**3.15x**)	0.065B (**3.34x**)	0.81 (**3.51x**)	0.024B (**3.92x**)
	$\mathtt{Gr}_p$	4.50 (**2.16x**)	0.383B (**2.65x**)	1.43 (**3.15x**)	0.065B (**3.34x**)	0.81 (**3.51x**)	0.024B (**3.92x**)
		SEED$_1$		**SEED$_2$**		**SEED$_3$**	
Am	$\mathtt{B}_{ns}$	28.8	2.675B (**59%**)	3.60	0.201B (**23%**)	1.45	0.060B (**10%**)
	$\mathtt{B}_s$	27.3 (**1.05x**)	2.322B (**1.15x**)	2.44 (**1.48x**)	0.129B (**1.56x**)	0.85 (**1.71x**)	0.033B (**1.82x**)
	Gr	26.1 (**1.10x**)	2.299B (**1.16x**)	2.44 (**1.48x**)	0.125B (**1.61x**)	0.82 (**1.77x**)	0.031B (**1.94x**)
	$\mathtt{Gr}_p$	26.0 (**1.11x**)	2.226B (**1.20x**)	2.21 (**1.63x**)	0.117B (**1.72x**)	0.84 (**1.73x**)	0.030B (**2.00x**)
Pa	$\mathtt{B}_{ns}$	15.5	1.343B (**6%**)	7.02	0.261B (**4%**)	4.01	0.101B (**0.3%**)
	$\mathtt{B}_s$	9.26 (**1.67x**)	0.639B (**2.10x**)	4.38 (**1.60x**)	0.116B (**2.25x**)	2.28 (**1.76x**)	0.044B (**2.30x**)
	Gr	9.26 (**1.67x**)	0.639B (**2.10x**)	3.29 (**2.13x**)	0.105B (**2.49x**)	1.77 (**2.27x**)	0.034B (**2.97x**)
	$\mathtt{Gr}_p$	9.26 (**1.67x**)	0.639B (**2.10x**)	3.29 (**2.13x**)	0.105B (**2.49x**)	1.77 (**2.27x**)	0.034B (**2.97x**)
		MagicRec$_1$		**MagicRec$_2$**		**MagicRec$_3$**	
Am	$\mathtt{B}_{ns}$	36.6	5.238B (**18%**)	5.41	0.659B (**20%**)	2.03	0.180B (**17%**)
	$\mathtt{B}_s$	21.1 (**1.73x**)	2.288B (**2.29x**)	2.73 (**1.98x**)	0.301B (**2.19x**)	1.10 (**1.85x**)	0.076B (**2.37x**)
	Gr	20.9 (**1.75x**)	2.188B (**2.39x**)	2.73 (**1.98x**)	0.265B (**2.49x**)	1.05 (**1.93x**)	0.066B (**2.73x**)
Pa	$\mathtt{B}_{ns}$	87.6	11.16B (**16%**)	11.7	1.406B (**14%**)	6.88	0.435B (**13%**)
	$\mathtt{B}_s$	44.1 (**1.99x**)	4.484B (**2.49x**)	6.48 (**1.81x**)	0.537B (**1.95x**)	3.22 (**2.14x**)	0.159B (**2.74x**)
	Gr	43.7 (**2.00x**)	4.114B (**2.71x**)	6.22 (**1.88x**)	0.473B (**2.21x**)	2.66 (**2.59x**)	0.138B (**3.15x**)

The percentage value next to $\mathtt{B}_{ns}$ total i-cost shows the percentage of work done in the last level.
Values in parentheses show the factor of improvement of the runtime over $\mathtt{B}_{ns}$.

- $\mathtt{B}_s$ (for **s**haring): Puts the plans of $\mathtt{B}_{ns}$ into a combined plan.
- Gr: The combined plan generated by our greedy optimizer.
- $\mathtt{Gr}_p$ (for **p**artial intersection sharing): The combined plan from Gr sharing partial intersections.

We measured the performances of these plans on Amazon, Google, Epinions, and Patents with one, two, and three labels. In each experiment, we pick 90% of the edges of the input graph G randomly and pre-load them to GraphflowDB. We then insert the remaining 10% edges in batches of 5. Table 13 shows our experiments on Patent and Amazon. Appendix F shows our results on Epinions and Google. The table shows the total runtime and i-cost of $\mathtt{B}_{ns}$, $\mathtt{B}_s$, Gr, and $\mathtt{Gr}_p$. We show the i-cost numbers to explain an important pattern we discuss momentarily. The numbers in the parantheses next to $\mathtt{B}_s$, Gr, and $\mathtt{Gr}_p$ report the relative performance improvements of Gr and $\mathtt{Gr}_p$ over $\mathtt{B}_{ns}$. We explain the percentage value next to the i-cost value of $\mathtt{B}_{ns}$ momentarily. In the remainder of this section, we make several observations on the experiments reported in

Table 13 and readers can verify that these observations also hold in our experiments reported in Appendix F.

We start by analyzing the benefits of sharing computations. For this, we compare the $\mathtt{B}_{ns}$ and $\mathtt{B}_s$ rows, which use exactly the same QVOs for each delta subgraph query and only differ on whether or not they share computation. We can also compare $\mathtt{B}_{ns}$ and Gr rows but in addition to sharing vs. not sharing, these plans also differ in the QVOs they use for each delta subgraph query. So, $\mathtt{B}_{ns}$ and $\mathtt{B}_s$ comparison is more controlled. First, observe that sharing always improves performance. Specifically, $\mathtt{B}_s$ outperforms $\mathtt{B}_{ns}$ by up to 2.93×. However observe also that there are significant variations in the relative runtime improvements across experiments. We next explain what governs these differences.

Fundamentally, the runtime improvements of sharing depends on what fraction of $\mathtt{B}_{ns}$'s work $\mathtt{B}_s$ shares. Equivalently, this fraction depends on how much of the work is done at the operators where sharing happens in $\mathtt{B}_s$. In our query sets, one good proxy for this is to study the amount of work that is done in the last-level operators. The last-level operators consist of the operators of the delta subgraph queries with the largest number of query vertices. Unless two delta subgraph queries are completely symmetric, which does not happen in our query sets, there can be no computation sharing in the last-level operators. Therefore, the amount of work done in this level is a good proxy for how much benefits sharing can give. We report the percentage of i-cost in the last-level operators in the parentheses next to the i-cost column of $\mathtt{B}_{ns}$. The lower this number, the more benefits we expect to get from sharing. For example, on Amazon, $\mathtt{4Cs}_1$ this percentage is 65% and the runtime benefits of $\mathtt{B}_s$ is 1.13×, while on $\mathtt{MagicRecs}_1$, this percentage is 18% and the runtime difference is 1.73×. A controlled comparison can be made between $\mathtt{4Cs}_1$ and $\mathtt{4Cs5C}_1$ on the Patents dataset. On Patents, even though there are matches for the 4Cs query set, there are no matches of the 4-cliques that are subsets of the 5-clique in 4Cs5C. That is why we see percentage of 0% in the Patents row of $\mathtt{4Cs5C}_1$, because the last-level operators have no inputs. So when evaluating $\mathtt{4Cs5C}_1$ with $\mathtt{B}_{ns}$, each of 10 delta subgraph queries of the 5-clique query needs to search for matches for 4-cliques over and over again. However, $\mathtt{B}_s$ shares the computation of these 10 delta subgraph queries with the delta subgraph queries from the 4-clique queries, so incurs no additional i-cost (observe the 0.479B i-cost of $\mathtt{B}_s$ both in $\mathtt{4Cs}_1$ and $\mathtt{4Cs5C}_1$ on Patents). So we expect $\mathtt{B}_s$ to outperform $\mathtt{B}_{ns}$ by a larger fraction in $\mathtt{4Cs5C}_1$ than in $\mathtt{4Cs}_1$. This is indeed what we observe on Patents: 1.65× vs. 1.82× in runtime and 1.79× vs. 2.12× in i-cost.

How much work is done at the last-level operators also depends on structural properties of the input datasets. We focus on two structural properties that give us controlled ways to test their effects:

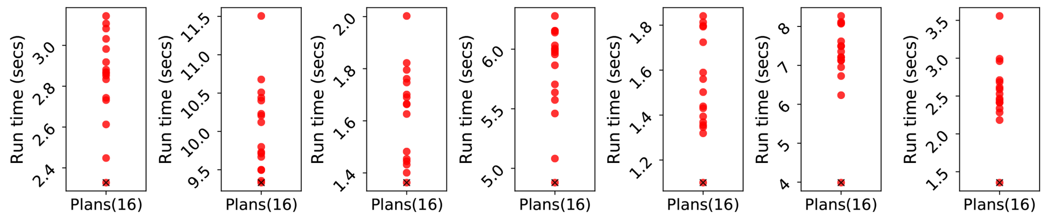
(i) Clustering coefficient: This is a measure of how cyclic a graph is and for the number of cliques there are in a graph. Because all of the queries in 4Cs and 4Cs5C query sets are cliques, the clustering coefficients of the input graphs allow us to control for how much of the work is done in the last levels. When the clustering coefficient is low there will be less cliques in the graph, so the last-level operators, which produce outputs, will do less work. Let us take as an example the benefits of sharing on $\mathtt{4Cs5C}_1$ on Amazon and Patents, which respectively have clustering coefficients of 0.42 and 0.08. So we expect more benefits on Patents than on Amazon. Indeed, this is what we observe. $\mathtt{B}_s$ outperforms $\mathtt{B}_{ns}$ on Amazon and Patents, respectively, by a factor of 1.14× and 1.82× in runtime and 1.21× and 2.12× in i-cost. A similar pattern holds on 4Cs and in fact the rest of our query sets, which are also cyclic.

(ii) Dataset heterogeneity: The number of labels in the datasets gives us another parameter we can use to control for the amount of work that's done at the last levels. Increasing the number of labels in a dataset decreases the number of matches of the queries in our query sets, which have a single label, so less work would be done in the last-level operators. Indeed readers can observe that the fraction of work done in the last level decreases when the data heterogeneity increases or we go right on any row in Table 13. Therefore, we expect $\mathtt{B}_s$ to outperform $\mathtt{B}_{ns}$ by a larger factor as we go right in the table. For example, on Amazon SEED row, performance increases from 1.05× to 1.48×, to 1.71× as labels increase from 1 to 2 to 3. This pattern broadly holds in our experiments but there are exceptions. For example, moving from 1 to 2 labels on Patents and running the SEED query set, we see lower relative benefits of sharing, a reduction from 1.67× to 1.60×. This is because as we observed above on Patents there are no 4-cliques so 0% of the work is done in the last-level operator even when there is a single label on the dataset. So we cannot use this metric as a proxy to predict the benefits of sharing as labels increase.

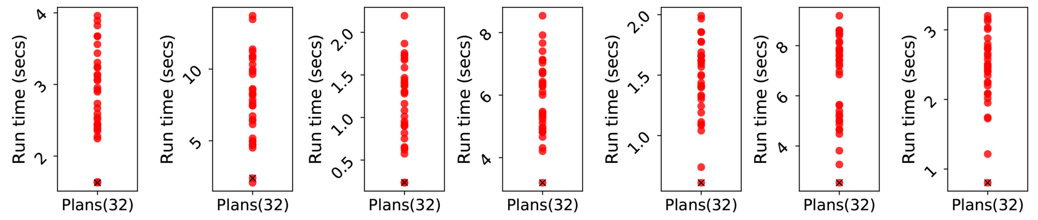
We next compare $\mathtt{B}_s$ and Gr to answer whether or not our greedy optimizer, which directly optimizes for a combined low i-cost plan, can find more computation sharing opportunities than $\mathtt{B}_s$. Observe that across all of our experiments Gr is able to find a plan with better i-cost and runtime. For example on Patents $\mathtt{4Cs5C}_2$, Gr improves performance over $\mathtt{B}_{ns}$ by 3.51× while $\mathtt{B}_s$ improves by 2.93× (so an additional 1.21x improvement). Similarly on $4Cs_3$ Amazon, Gr improves performance over $\mathtt{B}_{ns}$ by 1.53× while $\mathtt{B}_s$ improves by 1.20× (so an additional 1.28x improvement). There are very few exceptions to this pattern (all in Appendix F and in all of them the absolute difference is less than 140 ms and relative slow down at most 1.04×).

Finally, we compare $\mathtt{Gr}_p$ with Gr to understand how much benefits we get from our partial intersection sharing optimization. We omit $\mathtt{Gr}_p$ numbers for MagicRecs. This is because to apply partial intersection sharing, we need three-way intersections and on MagicRecs our Gr plan only performs two way intersections. Observe that in all of our experiments, $\mathtt{Gr}_p$ either performs equally or better than Gr (except 3 cases of a total of 48). For example, on Amazon dataset and $\mathtt{SEED}_2$ query set, we see that $\mathtt{Gr}_p$ improves performance over $\mathtt{B}_{ns}$ by 1.68× while Gr improves over $\mathtt{B}_{ns}$ by 1.56× (So an improvement of 1.08×). There is no simple answer to when $\mathtt{Gr}_p$ outperforms Gr more. For example, we do not observe a clear pattern that increasing the number of labels in input labels increases the benefits of partial sharing. This is because we perform partial intersection sharing at all levels, so shifting the amount of work done to lower-level operators does not necessarily imply that we should expect to benefit less from partial sharing. Importantly, our experiments demonstrate that partial intersection sharing is robust and improves performance broadly in our experiments. Finally, putting our greedy optimizer's plan and the partial intersection sharing, we observe up to 3.51× runtime improvements over $\mathtt{B}_{ns}$ and up to 3.92× reduction in i-cost in our experiments.

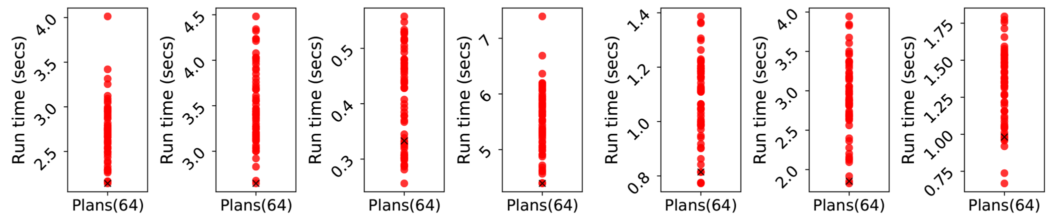
6.3.3 Goodness of Greedy Optimizer. We next study how good are our greedy optimizer's combined plans, compared to the space of all combined plans. We compare the plans we pick against all other possible plans in a query set's plan spectrum using the same setup as Section 6.3.2. For this analysis, we pick query sets that consist of one or two queries to ensure that the number of possible combined plans is small. The queries we evaluate on are the diamond query (Q_D), the diamond-X query (Q_{DX}), and the 4-Clique query (Q_{4C}) on four datasets. We also evaluate on two query sets with two queries: one contains two 4-Cliques (Q_{4Cs}) and the other contains a diamond and a 4-Clique (Q_{D-4C}). We use all datasets with one and two labels. Except we omit Amazon with two labels as the runtimes of all plans were less than 1 s. Our greedy optimizer's plans were broadly optimal or very close to optimal across our experiments. Figure 19 shows our spectrum charts. Our optimizer's plans were optimal in 16 of our 25 spectrums and within 1.15× of the optimal in 7 spectrums. In the 2 left cases, we were 1.30× and 1.47× of the optimal and the absolute runtime difference was 77 ms and 313 ms, respectively.



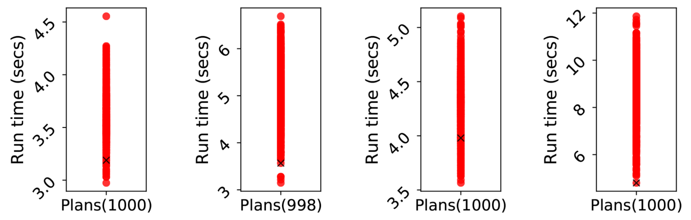
(a) Q_{D_1}, Am. (b) Q_{D_1}, Ep. (c) Q_{D_2}, Ep. (d) Q_D,Go$_1$. (e) Q_D,Go$_2$. (f) Q_D,Pa$_1$. (g) Q_D,Pa$_2$.



(h) $_{DX}$,Am$_1$. (i) Q_{DX},Ep$_1$. (j) Q_{DX},Ep$_2$. (k) Q_{DX},Go$_1$. (l) Q_{DX},Go$_2$. (m) Q_{DX},Pa$_1$. (n) Q_{DX},Pa$_2$.



(o) Q_{4C},Am$_1$. (p) Q_{4C},Ep$_1$. (q) Q_{4C},Ep$_2$. (r) Q_{4C},Go$_1$. (s) Q_{4C},Go$_2$. (t) Q_{4C},Pa$_1$. (u) Q_{4C},Pa$_2$.



(v) Q_{4Cs},Am$_1$. (w) Q_{4Cs},Pa$_1$. (x) Q_{D-4C},Am$_1$. (y) Q_{D-4C},Pa$_1$.

Fig. 19. Runtime (seconds) of a sample of the plans enumerated by the continuous query optimizer for Diamond (Q_D), Diamond-X (Q_{DX}), and 4-Clique (Q_{4C}) and two query sets, one of two 4-Cliques (Q_{4Cs}) and the other of a Diamond and a 4-Clique (Q_{D-4C}) on datasets Am, Ep, Go, and Pa with one and two labels. "x" specifies the plan picked by GraphflowDB.

6.3.4 Scalability. Finally, for completeness of our work, we tested the scalability of our combined plans on our largest graph Twitter dataset and loaded 90% of it to GraphflowDB. We evaluated the system on the 4Cs and 4Cs5C query sets. For 4Cs, we inserted 500K random updates in batches of 5. For 4Cs5C, we inserted 25K updates. Table 14 shows the runtime and output throughput, i.e., number of cliques output. We are able to output 23.8M cliques per second on the 4Cs5C. These numbers look competitive with distributed implementation of Delta Generic Join from Reference [6], which reports outputting 46.5M 4-cliques on a larger graph using 224 cores. A direct comparison is not possible, since the work from Reference [6] considers a single query at at time,

Table 14. Continuous Subgraph Queries Scalability Evaluations

	Runtime (seconds)	Output matches throughput/second
4Cs	1,323	14.5M
4Cs5C	6,627	23.8M

does not contain an optimizer, is designed and implemented for the distributed setting, and is written in a different programming language.

7 RELATED WORK

Our current work substantially expands a previous conference publication [41], which studied optimizing one-time subgraph queries using WCO join algorithms. We expand on this work by studying how to optimize continuous subgraph queries using WCO join algorithms. This includes the entire Section 4.2 and parts of every section related to continuous subgraph query evaluation. We also expand our experimental evaluation in Section 6 for one-time queries by providing spectrum analyses for all of our queries.

In the rest of this section, we review related work in WCO join and IVM algorithms, one-time and continuous subgraph query evaluation algorithms, and cardinality estimation techniques related to our catalogue. We focus on serial algorithms and single node systems. For join and subgraph query evaluation, several distributed solutions have been developed in the context of graph data processing [38, 64], RDF engines [1, 70], or multiway joins of relational tables [4, 6, 54]. We do not review this literature here in detail. Reference [34] and Reference [58] evaluate multiple one-time subgraph queries with selective predicates. We omit their detailed review here. There is a rich body of work on adaptive query processing in relational systems and multiple query processing in stream processing, for which we refer readers to References [13, 20, 26, 62].

WCO Join Algorithms and IVM: Prior to Generic Join, there were two other WCO join algorithms introduced called NPRR [50] and LFTJ [66]. Similarly to Generic Join, these algorithms also perform attribute-at-a-time join processing using intersections. We covered EH [2], CTJ [28], and Tributory Join [15], which are systems and algorithms that use these algorithms for one-time natural join or subgraph queries in Section 6.

Our continuous subgraph query evaluation is based on the Delta Generic Join [6], an IVM algorithm for join queries. Numerous works exist on IVM of relational queries. A survey of this literature can be found in Reference [57]. The closest to Delta Generic Join is an IVM algorithm based on LFTJ from Reference [67], which maintains an index that can be as large as the AGM bound of the query. Instead, our approach, as also observed in Reference [6], does not maintain any auxiliary indexes.

DBToaster [5, 31] is another IVM system that generates delta queries to maintain continuous queries. To incrementally maintain a query Q, DBToaster relies on *higher-order IVM*. Although this technique can be used to maintain our join-only continuous subgraph queries, it is primarily designed for and is efficient on queries with aggregations. In particular, DBToaster maintains all of the higher order delta queries of Q, and upon an update to the relations, uses ith degree delta queries to update $(i-1)$th degree views. These update computations do not involve any joins and perform only selections (and other operations such as arithmetic and unions). However, to avoid joins, DBToaster maintains views that are sub-queries of Q, which can be prohibitively expensive for the queries we target. For example, to maintain a triangle query $Q : R(a,b) \bowtie S(b,c) \bowtie T(c,a)$, DBToaster would use three delta queries, e.g., $\Delta_R(Q) : \Delta R \bowtie S \bowtie T$, which is similar to our delta queries, with the following important difference. To compute $\Delta_R(Q)$ without computing joins, DBToaster maintains the view $V_{ST} = S \bowtie T$. Notice that this computes the “open triangles” (in graph setting). Instead, our processor does not maintain any views other than the original tables, and executes $\Delta R \bowtie S \bowtie T$ from scratch. Note also that because we adopt worst-case optimal joins in our evaluator, even when evaluating delta queries from scratch, we do not generate open triangles.

More recent work improves over DBToaster and higher-order IVM techniques [27]. Since Higher-Order IVM materializes not only the result of Q but also the results of the higher-order delta query, it struggles to compute Q when it is output size is much larger than that of the database especially for the in-memory setting. Reference [27] introduces an algorithm to maintain results of acyclic queries under updates relying instead of materialization on a data structure called **Dynamic Constant-delay Linear Representation (DCLR)**. DCLR and the Dynamic Yannakakis Algorithm introduced guarantee linear time maintenance under updates while using only linear space in the size of the database. The technique is reminiscent of factorized database representation and processing [52]. In contrast to DCLR, our delta query IVM technique that we adopted does not require any space (not even linear space) and can maintain both cyclic and acyclic queries.

Single One-time Subgraph Query Evaluation Algorithms: Many of the earlier subgraph matching algorithms are based on Ullmann’s branch and bound or backtracking method [65]. The algorithm conceptually performs a query-vertex-at-a-time matching using an arbitrary QVO. This algorithm has been improved with different techniques to pick better QVOs and filter partial matches, often focusing on queries with labels [17, 18, 63]. Several recent algorithms perform preprocessing to find *candidate vertex sets* (the set of possible data vertices for each query vertex), build an auxiliary data structure for these sets and finally pick a QVO for the evaluation. Such algorithms include Turbo$_{ISO}$ [25], CFL [10], CECI [9], and DP-iso [24]. Each of these algorithms include optimizations on the auxiliary data structure as well as query processing. Turbo$_{ISO}$, for example, proposes to merge similar query vertices (same label and neighbours) to minimize the number of partial matches and once the merged and smaller query is evaluated, perform a Cartesian product to enumerate the final outputs. CFL decomposes the query into a dense subgraph and a forest, and processes the dense subgraph first to reduce the number of partial matches. CFL also uses an index called compact path index, which estimates the number of matches for each root-to-leaf query path in the query and is used to enumerate the matches as well. We compare our approach to CFL in our supplementary Appendix D as its code is available. CECI and DP-iso rely on an auxiliary data structure that maintains edges between candidates and also rely on multiway intersections when finding candidate sets. Each of the algorithms has its own optimization, e.g., CECI divides the data graph into multiple embedding clusters for parallel processing while DP-iso relies on an adaptive QVO selection and a pruning technique called *pruning by failing sets*, which are partial matches with no possible extensions in the data graph. A systematic comparison of our approach against these approaches is beyond the scope of this article. Our approach is specifically designed to be decomposable into operator-based query plans that the query processors of existing GDBMSs generate and implementable on GDBMSs that adopts a cost-based optimizer.

Another group of algorithms index different structures in input graphs, such as frequent paths, trees, or triangles, to speed up query evaluation [69, 71]. Such approaches can be complementary to our approach. For example, Reference [6] in the distributed setting demonstrated how to speed up Generic Join based WCO plans by indexing triangles in the graph.

Multi-Query Optimization: Computation sharing by identifying common computations arises in many query processing settings, such as when running a single complex query that contains repeated sub-queries, running a batch of queries with common expressions [59, 72], data streaming systems that perform on-line queries with common aggregations [32], or in systems that maintain multiple materialized views [5, 42]. Our continuous subgraph query evaluation setting is an example of maintaining multiple views. When a query processor needs to evaluate multiple queries, instead of using separate individual plans for each query, the task is to construct a consolidated plan to evaluate all of these queries, ensuring that common subexpressions are evaluated once and consumed by multiple upstream operators. There as been many prior work that have studied different aspects of this problem, such as how to detect common sub-expressions, e.g., using exhaustive [62] or heuristic algorithms [59], or whether to share computation through materialization [42, 59] or pipelining [19]. Our approach is based upon these same foundations. Specifically, our combined plans fall under a heuristic method that finds common expressions greedily, similar to Reference [59], and performs the entire computation in a pipelined manner. Building upon these methods, our work studies how to optimize the delta decompositions of multiple subgraph queries when using the new intersection-based worst-case optimal join algorithms, for which we use a new i-cost metric, and a partial intersection sharing technique to improve performance.

Multiple Continuous Subgraph Query Algorithms: EMVM [55] evaluates multiple subgraph queries under single-edge insertion workloads. Given a set of queries $\bar{Q}$, EMVM partitions the queries in $\bar{Q}$ into separate query sets $\bar{Q}_{l_1}, \ldots, \bar{Q}_{l_k}$, one for each separate edge predicate l_i (called labels) in the queries. Each query set $\bar{Q}_{l_i}$ contains as many **edge-annotated views (EAVs)** of the same query Q as there are edges with predicate l_i in Q. EAVs are similar to our delta subgraph queries. For each $\bar{Q}_{l_i}$, EMVM constructs a larger "merged view" that is similar to our combined plans. EMVM assumes a query set with highly selective predicates, which is reflected in two main differences between EMVM and our approach: (1) Merged views are constructed to share as many edges as possible between the queries on M, ignoring their cyclic structures, and (2) queries are evaluated one query edge at a time.

Single Continuous Subgraph Query Algorithms: TurboFlux [30] evaluates a query using a **data centric graph (DCG)**, which is a compressed representation of partial matches of the query in $G(E_G, V_G)$. Upon updates to G, TurboFlux runs a subgraph matching algorithm on the DCG to detect instances of Q. Such processing on compressed data structures is very different from out flat tuple-based processing and, unlike our approach, seems harder to decompose into existing graph databases. Our supplementary Appendix E gives a more detailed overview of TurboFlux and its performance comparison against our approach.

Reference [21] describes a general search localization technique called *IncIsoMat* that, given an update $e(u, v)$ to G computes a region of G called the *affected area* that may include an emergence or deletion of instances of a query Q. Matching instances of Q is found by using any subgraph matching algorithm on the affected area and is left unspecified. Our delta subgraph query framework automatically localizes its search to the same and sometimes smaller area around e. Finally, Reference [14] describes a technique called *SJ-Tree*, which constructs a left-deep query plan P for a query Q, where each leaf is either a 1-edge or 2-edge path of Q. Upon updates to the graph, SJ-Tree maintains partial matches to each intermediate node of P using a hash join algorithm. SJ-Tree is designed for queries with highly selective predicates, e.g., the reference assumes that the number of matches for 2-edge paths are expected to be significantly fewer than 1-edge paths, which does not hold for many queries in practice. As a result, for many queries, this technique can materialize prohibitively large intermediate results.

Cardinality Estimation Using Small-size Graph Patterns: Our catalogue is closely related to Markov tables [3], and MD- and Pattern-tree summaries from Reference [39]. Similarly to our catalogue, both of these techniques store information about small-size subgraphs to make cardinality estimates for larger subgraphs. Markov tables were introduced to estimate cardinalities of paths in XML trees and store exact cardinalities of small size paths to estimate longer paths. MD- and Pattern-tree techniques store exact cardinalities of small-size acyclic patterns, and are used to estimate the cardinalities of larger subgraphs (acyclic and cyclic) in general graphs. These techniques are limited to cardinality estimation and store only acyclic patterns. In contrast, our catalogue stores information about acyclic and cyclic patterns and is used for both cardinality and i-cost estimation. In addition to selectivity (μ) estimates that are used for cardinality estimation, we store information about the sizes of the adjacency lists (the $|A|$ values), which allows our optimizer to differentiate between WCO plans that generate the same number of intermediate results, so have same cardinality estimates, but incur different i-costs. Storing cyclic patterns in the catalogue allow us to make accurate estimates for cyclic queries.

8 CONCLUSION

We described two cost-based optimizers: (i) a cost-based dynamic programming optimizer for one-time subgraph queries that enumerates a plan space that contains WCO plans, BJ plans, and a large class of hybrid plans and (ii) a cost-based greedy optimizer for continuous subgraph queries, which builds on top of the delta subgraph query framework. Our one-time optimizer generates novel hybrid plans that seamlessly mix intersections with binary joins, which are not in the plan space of prior optimizers for subgraph queries. Our continuous optimizer relies on multi-query optimization using computation sharing to lower the costs of plans. Within both optimizers, WCO plans are assigned a cost based on our i-cost metric, which captures the several runtime effects of QVOs we identified through extensive experiments.

Our approaches in this article have several limitations, which give us directions for future work. First, our optimizer can benefit from more advanced cardinality and i-cost estimators, such as those based on sampling outputs or machine learning. Second, for very large one-time queries, currently our one-time optimizer enumerates a limited part of our plan space. Studying faster plan enumeration methods, similar to those discussed in Reference [46], is an important future work direction. Finally, existing literature on subgraph matching, both in the one-time and continuous settings, contain several optimizations for identifying and evaluating independent components of a query separately. Example optimizations include factorization [52] or postponing the Cartesian product optimization from Reference [10].

REFERENCES

[1] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. 2015. SPARTex: A vertex-centric framework for RDF data analytics. *Proc. VLDB* (2015).

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Empty-Headed: A relational engine for graph processing. *Trans. Database Syst.* 42, 4, Article 20 (2017), 44 pages.

[3] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. 2001. Estimating the selectivity of XML path expressions for internet scale applications. *Proc. VLDB* (2001).

[4] F. N. Afrati and J. D. Ullman. 2011. Optimizing multiway joins in a map-reduce environment. *TKDE* 23, 9 (2011), 1282–1298.

[5] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB* (2012).

[6] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *Proc. VLDB* (2018).

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *SIGMOD'15*.

[8] A. Atserias, M. Grohe, and D. Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.

[9] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD'19*.

[10] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing Cartesian products. In *SIGMOD'16*.

[11] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *SIGMOD Rec.* 15, 2 (1986), 61–71.

[12] Arezo Bodaghi and Babak Teimourpour. 2018. *Automobile Insurance Fraud Detection Using Social Network Analysis*.

[13] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD'00*.

[14] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT'15*.

[15] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD'15*.

[16] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products.

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. 1999. Performance evaluation of the VF graph matching algorithm. In *ICIAP'99*.

[18] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *Trans. Pattern Anal. Mach. Intell.* (2004).

[19] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. 2001. Pipelining in multi-query optimization. In *PODS'01*.

[20] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Found. Trends Databases* (2007).

[21] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *SIGMOD'11*.

[22] Goetz Graefe. 1994. Volcano—An extensible and parallel query evaluation system. *Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.

[23] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proc. VLDB* (2014).

[24] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD'19*.

[25] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD'13*.

[26] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-based multi-query optimization. In *EDBT'09*.

[27] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD'17*.

[28] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible caching in Trie Joins. In *EDBT'17*.

[29] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD'17*.

[30] Kim, Kyoungmin and Seo, In and Han, Wook-Shin and Lee, Jeong-Hoon and Hong, Sungpack and Chafi, Hassan and Shin, Hyungyu and Jeong, Geonhwa. 2018. TurboFlux: A fast continuous subgraph matching system for streaming graph data. In *SIGMOD'18*.

[31] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: Higher-order delta processing fordynamic, frequently fresh views. *VLDB J.* 23 (2014), 253–278.

[32] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD'06*.

[33] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? In *WWW'10*.

[34] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *ICDE'12*.

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB* (2015).

[36] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exper.* 46, 6 (2016), 723–749.

[37] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.

[38] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. 2016. Scalable distributed subgraph enumeration. In *VLDB'16*.

[39] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. 2008. Graph summaries for subgraph frequency estimation. *The Semantic Web: Research and Applications* (2008).
[40] Maximum Common Induced Subgraph [n.d.]. Maximum Common Induced Subgraph. Retrieved from https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph.
[41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB* (2019).
[42] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD'01*.
[43] neo4j [n.d.]. Retrieved from Neo4j. https://neo4j.com/.
[44] neo4j:fraud [n.d.]. Fraud Detection: Discovering Connections with Graph Databases. Retrieved from https://neo4j.com/use-cases/fraud-detection.
[45] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB* (2011).
[46] Thomas Neumann and Bernhard Radke. 2018. Adaptive optimization of very large join queries. In *SIGMOD'18*.
[47] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDBJ* 19 (2010), 91–113.
[48] M. E. J. Newman. 2004. Detecting community structure in networks. *Eur. Phys. J. B* (2004).
[49] H. Ngo, C. Ré, and A. Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2014), 5–16.
[50] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms. In *PODS'12*.
[51] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. CoRR/1503.04169 (2015)
[52] Dan Olteanu and Jakub Závodný. 2015. Size bounds for factorised representations of query results. *Trans. Database Syst.* 40, 1, Article 2 (2015).
[53] opencypher [n.d.]. Retrieved from openCypher. http://www.opencypher.org.
[54] Paraschos Koutris and Semih Salihoglu and Dan Suciu. 2018. Algorithmic aspects of parallel data processing. *Found. Trends Databases* (2018).
[55] Andrea Pugliese, Matthias Bröcheler, V. S. Subrahmanian, and Michael Ovelgönne. 2014. Efficient multiview maintenance under insertion in huge social networks. *ACM Trans. Web* 8, 2, Article 10 (2014).
[56] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB* (2018).
[57] Rada Chirkova and Jun Yang. 2012. Materialized views. *Found. Trends Databases* (2012).
[58] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proc. VLDB* (2016).
[59] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *SIGMOD'00*.
[60] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The graph story of the SAP HANA database. In *BTW'13*.
[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: Extended survey. *VLDB J.* 29 (2020), 595–618.
[62] Timos K. Sellis. 1988. Multiple-query optimization. *Trans. Database Syst.* 13, 1 (1988), 23–52.
[63] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB* (2008).
[64] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *SIGMOD'14*.
[65] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.
[66] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: A worst-case optimal join algorithm. CoRR/1210.0481 (2012).
[67] Todd L. Veldhuizen. 2013. Incremental maintenance for Leapfrog Triejoin. CoRR/1303.5313 (2013).
[68] Umeshwar Dayal, Jennifer Widom, and Stefano Ceri. 1994. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc.
[69] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph indexing: A frequent structure-based approach. In *SIGMOD'04*.
[70] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. *Proc. VLDB* (2013).
[71] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. 2007. Graph indexing: Tree + delta <= graph. In *Proc. VLDB'07*.
[72] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD'07*.

Received January 2020; revised December 2020; accepted January 2021

Online Appendix to: Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins

AMINE MHEDHBI, CHATHURA KANKANAMGE, and SEMIH SALIHOGLU,
University of Waterloo

This online appendix contains: (1) the complexity result of the multiple continuous subgraph query optimization problem; (2) catalogue experimental results; (3) an explanation for how our plans subsume EmptyHeaded plans; (4) comparisons against CFL and TurboFlux; and (5) further continuous subgraph query evaluation experiments.

A COMPLEXITY OF MULTIPLE CONTINUOUS SUBGRAPH QUERY OPTIMIZATION

A.1. Formal Optimization Problem and Its Computational Complexity

Recall that we do not know the exact complexity of the actual computational problem that our optimizer solves. However, we can show that the natural decision version of a slightly more general version of the problem, in which we drop the assumption that the set of DSQs in $\bar{Q}_{DSQ}$ are delta decompositions of a set of subgraph queries is NP-hard. Note that combined plans effectively use common operators across DSQs if the DSQs compute isomorphic sub-queries. Our goal in providing this proof is to make the connection between our optimization problem and the maximum common induced subgraph problem, which is NP-hard. We first define the more general optimization problem we consider:

Definition (Generalized Multiple DSQ Optimization Problem (GDOP)): Given a set of arbitrary delta subgraph queries $\bar{Q}_{DSQ}$, i.e., a set of subgraph queries where one edge is labeled with δ and the other edges with o or n, and an arbitrary full catalogue C, and a target cost k, find whether or not there is a combined plan with cost at most k.

Theorem A.1. *GDOP is NP-hard.*

Proof. We show that the GDOP is NP-hard on instances in which the given a catalogue that has a value of 1 for each cost and selectivity. That is the entries in the catalogue are such that any Q_{k-1} to Q_k extension entry has μ value 1 and the $|A|$ value equal to lists that sum to 1. We call this the *uniform catalogue*. Note that when this is the input catalogue the optimal combined plan is the plan that contains the smallest number of operators. We next show that the **maximum common induced subgraph problem (MCISP)** [40] , which is NP-hard, reduces to GDOP. Given two graphs G_1 and G_2 and a target value t, MCISP is the problem of finding whether or not there is a subgraph H with at least t vertices that is an induced subgraph of both G_1 and G_2, i.e., the projection of G_1 and G_2 onto the vertices in H gives H.

The reduction is as follows. Take an instance of MCISP with graphs G_1 and G_2 and a target induced subgraph of size t. Assume the nodes in G_1 are labeled with $a_1, a_2, \ldots, a_{m_1}$ and each node in G_2 is labeled with $b_1, b_2, \ldots, b_{m_2}$. We first construct an instance of GDOP as follows. Label each edge of G_1 and G_2 with o, and extend both G_1 and G_2 with a new edge $x \rightarrow y$ with label δ and connect

© 2021 Association for Computing Machinery.
0362-5915/2021/05-ART6 $15.00
https://doi.org/10.1145/3446980

both x and y to each node in G_1 and G_2. These labeled and extended G_1 and G_2 graphs are now delta subgraph queries, and we refer to them as DSQ_1 and DSQ_2. Now consider solving GDOP on DSQ_1 and DSQ_2 using a uniform catalogue and a target cost of $m_1 + m_2 + 1 - t$, which, due to the structure of the catalogue, is the problem of finding a combined plan with at most $m_1 + m_2 + 1 - t$ operators.

First, observe that any combined plan needs to have exactly two sink E/I operators because there are two DSQs, i.e., in the DAG of any correct combined plan for this GDOP instances there will be two final "branches" leading to sink operators. Second, observe that we only need to consider combined plans whose DAGs have the following structure: (1) start with a source Scan that matches $\xrightarrow{\delta}$ as usual; (2) a chain of z E/I operators each giving its output to one output E/I operator, which compute a common sub-query for both DSQ_1 and DSQ_2, where the last E/I operator gives its output to two operators (to start the final two "branches"); (3) two branches, one with $r_1 = m_1 - z$ many and the other with $r_2 = m_2 - z$ many E/I operators, each giving its output to one output E/I operator. Any combined plan that branches and merges multiple times is suboptimal, because, we can always keep the last two branches and then only keep one chain back to the source Scan operator, so remove all but one of the previous branches that eventually merge, which strictly decreases the number of operators in the combined plan. Therefore any combined plan effectively starts with a Scan operator that evaluates the extra $x \rightarrow y$ edge we added to G_1 and G_2 and then a z-size common induced subgraph of G_1 and G_2 and then in two separate branches of E/I operators evaluates the rest of the vertices in G_1 and G_2, with a cost of $m_1 + m_2 + 1 - z$ (+1 is for the initial scan operator). Therefore, there is an induced subgraph of size at least t if and only if there is a combined plan in the GDOP instance with at most $m_1 + m_2 + 1 - t$ cost, completing the proof. □

B CATALOGUE EXPERIMENTS

We present preliminary experiments to show two tradeoffs: (1) the space vs. estimation quality tradeoff that parameter h determines; and (2) construction time vs. estimation quality tradeoff that parameter z determines. For estimation quality we evaluate cardinality estimation and omit the estimation of adjacency list sizes, i.e., the $|A|$ column, that we use in our i-cost estimates. We first generated all 5-vertex size unlabeled queries. This gives us 535 queries. For each query, we assign labels at random given the number of labels in the dataset (we consider Amazon with 1 label, Google with 3 labels). Then for each dataset, we construct two sets of catalogues: (1) we fix z to 1,000, and construct a catalogue with $h = 2$, $h = 3$, and $h = 4$ and record the number of entries in the catalogue; (2) we fix h to 3 and construct a catalogue with $z = 100$, $z = 500$, $z = 1{,}000$, and $z = 5{,}000$ and record the construction time. Then, for each labeled query Q, we first compute its actual cardinality, $|Q_{true}|$, and record the estimated cardinality of Q, Q_{est} for each catalogue we constructed. Using these estimation we record the q-error of the estimation, which is max($|Q_{est}|$ / $|Q_{true}|$, $|Q_{true}|$ / $|Q_{est}|$). This is an error metric used in prior cardinality estimation work [35] that is at least 1, where 1 indicates completely accurate estimation. As a very basic baseline, we also compared our catalogues to the cardinality estimator of PostgreSQL. For each dataset, we created an Edge relation E(from, to). We create two composite indexes on the table on (from, to) and (to, from) which are equivalent to our forward and backward adjacency lists. We collected stats on each table through the ANALYZE command. We obtain PostgreSQL's estimate by writing each query in an equivalent SQL select-join query and running EXPLAIN on the SQL query.

Our results are shown in Tables 1 and 2 as cumulative distributions as follows: for different q-error bounds τ, we show the number of queries that a particular catalogue estimated with q-error at most τ. As expected, larger h and larger z values lead to less q-error, while respectively yielding

Table 1. Q-error and Catalogue Creation Time (CT) in Secs for GraphflowDB for Different z Values

	z	CT	≤2	≤3	≤3	≤5	≤10	>20
Am	100	0.1	318	445	510	526	529	535
	500	0.3	384	486	520	527	530	535
	1,000	0.5	383	481	519	529	532	535
	5,000	1.5	384	475	518	529	532	535
Go$_3$	100	3.1	166	276	356	415	561	535
	500	9.3	214	310	371	430	477	535
	1,000	17.0	222	315	371	430	475	535
	5,000	66.1	219	322	373	432	473	535

Table 2. Postgres (PG) and GraphflowDB (GF) Q-error and Number of Catalogue Entries (|R|) for GF for Different h Values

		h	|R|	≤2	≤3	≤3	≤5	≤10	>20
Am	GF	2	8	348	464	512	523	527	535
		3	138	381	482	512	524	527	535
		4	2858	498	510	518	524	527	535
	PG	–	–	15	15	23	23	25	535
Go$_3$	GF	2	144	181	289	375	447	492	535
		3	20.3K	222	315	371	430	475	535
		4	11.9M	441	497	515	524	529	535
	PG	–	–	0	0	0	0	0	535

larger catalogue sizes and longer construction times. The biggest q-error differences are obtained when moving from $h = 3$ to $h = 4$ and $z = 100$ to $z = 500$. There are a few exception τ values when the larger h or z values lead to very minor decreases in the number of queries within the τ bound but the trend holds broadly.

C SUBSUMED EMPTYHEADED PLANS

We show that our plan space contains EmptyHeaded's GHD-based plans that satisfy the projection constraint. For details on GHDs and how EmptyHeaded picks GHDs we refer the reader to Reference [2]. Briefly, a GHD D of Q is a decomposition of Q where each node i is labelled with a subquery Q_i of Q. The interpretation of a GHD D as a join plan is as follows: each subquery is evaluated using Generic Join first and materialized into an intermediate table. Then, starting from the leaves, each table is joined into its parent in an arbitrary order. So a GHD can easily be turned into a join plan T in our notation (from Section 3.2) by "expanding" each sub-query Q_i into a WCO subplan according to the chosen QVO that EmptyHeaded picks for Q_i and adding intermediate nodes in T that are the results of the joins that EmptyHeaded performs. Given Q, EmptyHeaded picks the GHD D^* for Q as follows. First, EmptyHeaded loops over each GHD D of Q, and computes the worst-case size of the subqueries, which are computed by the AGM bounds of these queries (i.e., the minimum *fractional edge covers* of sub-queries; see Reference [8]). The maximum size of the subqueries is the width of GHD and the GHD with the minimum width is picked. This effectively implies that one of these GHDs satisfy our projection constraint. This is because adding a missing query edge to $Q_i(V_i, E_i)$ can only decrease its fractional edge cover. To see this consider $Q'_i(V_i, E'_i)$, which contains V' but also any missing query edge in E_i. Any fractional edge cover for Q_i is a fractional edge cover for Q'_i (by giving weight 0 to $E'_i - E_i$ in the cover), so the minimum fractional edge cover of Q'_i is at most that for Q_i, proving that D^* is in our plan space.

We verified that for every query from Figure 10, the plans EmptyHeaded picks satisfy the projection constraint. However, there are minimum-width GHDs that do not satisfy this constraint. For example, for Q10, EmptyHeaded finds two minimum-width GHDs: (i) one that joins a diamond and a triangle (width 2) and (ii) one that joins a three path ($a_2a_1a_3a_4$) joined with a triangle with an extended edge (also width 2). The first GHD satisfies the projection constraint, while the second one does not. EmptyHeaded (arbitrarily) picks the first GHD. As we argued in Section 3.2.1 , satisfying the projection constraint is not a disadvantage, as it makes the plans generate fewer intermediate tuples. For example, on a Gnutella peer-to-peer graph [37] (neither GHD finished in a reasonable amount of time on our datasets from Table 9), the first GHD for Q10 takes around 150ms, while the second one does not finish within 30 minutes.

D CFL COMPARISON

CFL [10] is one of the state-of-the-art subgraph matching algorithms whose code is available. The algorithm can evaluate labelled subgraph queries as in our setting. The main optimization of CFL is what is referred to as "postponing Cartesian products" in the query. These are conditionally independent parts of the query that can be matched separately and appear as Cartesian products in the output. CFL decomposes a query into a dense *core* and a *forest*. Broadly, the algorithm first matches the core, where fewer matches are expected and there is less chance of independence between the parts. Then the forest is matched. In both parts, any detected Cartesian products are postponed and evaluated independently. This reduces the number of intermediate results the algorithm generates. CFL also builds an index called CPI, which is used to quickly enumerate matches of paths in the query during evaluation. We follow the setting from the evaluation section of Reference [10]. We obtained the CFL code and 6 different query sets used in Reference [10] from the authors. Each query set contains 100 randomly generated queries that are either sparse (average query vertex degree ≤ 3) or dense (average query vertex degree > 3). We used three sparse query sets Q10s, Q15, and Q20s containing queries with 10, 15, and 20 query vertices, respectively. Similarly, we used three dense query sets Q10d, Q15d, and Q20d. To be close to their setup, we use the human dataset from the original CFL paper. The dataset contains 86,282 edges, 4,674 vertices, 44 distinct labels. We report the average runtime per query for each query set when we limit the output to 10^5 and 10^8 matches as done in Reference [10]. Table 3 compares the runtime of GraphflowDB and CFL on the 6 query sets. Except for one of our experiments, on Q10d with 10^5 output size limit, GraphflowDB's runtimes are faster (between 1.2× to 12.2×) than CFL. We note that although our runtime results are faster than CFL on average, readers should not interpret these results as one approach being superior to another. For example, we think the postponing

Table 3. Average Runtime (seconds) of GraphflowDB (GF) and CFL on Large Queries

|T|		**Q10s**	**Q15s**	**Q20s**	**Q10d**	**Q15d**	**Q20d**
10^5	GF	7.3	6.0	5.5	29.2(**2.2x**)	99.8	142.0
	CFL	9.3(**1.2x**)	17.5(**2.9x**)	40.5(**7.3x**)	13.2	389.9(**3.9x**)	1,140.7(**8.0x**)
10^8	GF	625.6	665.5	797.2	1,159.6	1,906.2	1,556.9
	CFL	4,818.9(**7.7x**)	5,898.1(**8.8x**)	7,104.1(**8.9x**)	7,974.3(**6.8x**)	11,656.2(**6.1x**)	19,135.7(**12.2x**)

Q*i(s/d)* is a query set of 100 randomly generated queries where *i* is the number of vertices and *s* and *d* specify sparse and dense queries, respectively as specified in Appendix D.

Table 4. Turboflux (T_f) vs. GraphflowDB (GF) on Continuous Subgraph Queries Diamond (Q_D), Diamond-X (Q_{DX}), 4-Clique (Q_{4C}), and 5-Clique (Q_{5C}) from the Query Set SEED

		Q_D	Q_{4C}	Q_{5C}
Am	GF	3.5	2.1	9.1
	T_f	11.4 (**3.3x**)	13.2 (**6.3x**)	1069 (**117.5x**)
Go	GF	9.1	3.8	50.4
	T_f	29.9 (**3.3x**)	41.6 (**10.9x**)	3090 (**61.3x**)

of Cartesian products optimization and a CPI index are good techniques and can improve our approach. However, one major advantage of our approach is that we do flat tuple-based processing using standard database operators, so our techniques can easily be integrated into existing graph databases. It is less clear how to decompose CFL-style processing into database operators.

E TURBOFLUX COMPARISON

TurboFlux [30] evaluates a query using a *data centric graph* (DCG), which is a compressed representation of partial matches of the query in $G(E_G, V_G)$. Briefly, the DCG is a multigraph $G_{DCG}(V_{DCG}, E_{DCG})$ where $V_{DCG} = V_G$ and E_{DCG} contains $|V_Q| - 1$ parallel edges for each $e \in E_G$. Each parallel edge has a *state* of *null* (N), *implicit* (IM), or *explicit* (EX) and a *label* which is one of the query vertices in Q. An EX edge $u \rightarrow v$ with label $a_i \in V_Q$ indicate a set of successful matches of vertices (according to an order) to query vertices where v matches a_i. Updates to G, TurboFlux transitions the states of the edges and then runs a subgraph matching algorithm on the DCG.

We obtained the code from the original authors. We used four different continuous subgraph queries from the SEED query set: (i) a Diamond (Q_D); (ii) a Diamond-X (Q_{DX}); (iii) a 4-Clique (Q_{4C}); and (iv) a 5-Clique (Q_{5C}). As in previous experiments we pre-loaded both TurboFlux and GraphflowDB with a random 90% of the dataset. We streamed in the remaining 10% of edges one edge at a time because TurboFlux does not support batching of updates. We show the results of this experiment in Table 4. Across all datasets and queries, GraphflowDB outperforms TurboFlux by at least 3.3× and by up to 117.5×. As we emphasized in our one-time query CFL comparisons, readers should not conclude from these experiments that our approach is superior to TurboFlux's approach. The compared approaches and the actual implementations of these approaches are very different (and we were only provided the binary). We provide these experiments merely for completeness of our work and sanity checks to verify that our implementation is competitive with existing recent solutions from literature. We believe approaches such as DCG that allow compressed representations are good techniques. One important distinction to note is that our approach was specifically designed to be easily integrated into a GDBMS and our implementation is part of a GDBMS architecture. In contrast it is less clear how to decompose TurboFlux-style processing into actual database implementations. This is an interesting research direction.

F RUNTIME AND EXECUTION METRICS FOR CONTINUOUS SUBGRAPH QUERIES

Table 5 reports the rest of our experiments from Section 6.3 on Epinions and Google datasets.

Table 5. Runtime (Seconds) of B_{ns}, B_s, Gr, and G_{opis} on 4Cs, 4Cs5C, SEED, and MagicRec Query Sets

		Runtime	**I-cost**	**Runtime**	**I-cost**	**Runtime**	**I-cost**
		4Cs$_1$		**4Cs$_2$**		**4Cs$_3$**	
Ep	B_{ns}	5.51	1.488B (**84%**)	0.56	0.163B (**65%**)	0.33	0.052B (**50%**)
	B_s	5.15 (**1.07x**)	1.347B (**1.10x**)	0.43 (**1.30x**)	0.125B (**1.30x**)	0.21 (**1.57x**)	0.035B (**1.49x**)
	Gr	4.95 (**1.11x**)	1.328B (**1.20x**)	0.49 (**1.14x**)	0.126B (**1.30x**)	0.22 (**1.50x**)	0.037B (**1.41x**)
	Gr_p	4.78 (**1.15x**)	1.244B (**1.20x**)	0.49 (**1.14x**)	0.126B (**1.30x**)	0.18 (**1.83x**)	0.030B (**1.73x**)
Go	B_{ns}	14.61	3.032B (**58%**)	2.70	0.515B (**38%**)	1.31	0.188B (**39%**)
	B_s	13.03 (**1.12x**)	2.226B (**1.36x**)	2.14 (**1.26x**)	0.314B (**1.64x**)	1.12 (**1.17x**)	0.126B (**1.49x**)
	Gr	13.10 (**1.12x**)	2.073B (**1.46x**)	2.23 (**1.21x**)	0.285B (**1.81x**)	1.00 (**1.31x**)	0.118B (**1.59x**)
	Gr_p	13.16 (**1.11x**)	1.864B (**1.63x**)	1.99 (**1.36x**)	0.278B (**1.85x**)	0.94 (**1.39x**)	0.106B (**1.77x**)
		4Cs5C$_1$		**4Cs5C$_2$**		**4Cs5C$_3$**	
Ep	B_{ns}	16.71	5.607B (**64%**)	0.97	0.278B (**22%**)	0.42	0.074B (**7%**)
	B_s	16.35 (**1.02x**)	5.339B (**1.05x**)	0.95 (**1.02x**)	0.218B (**1.28x**)	0.29 (**1.45x**)	0.048B (**1.54x**)
	Gr	15.78 (**1.06x**)	5.320B (**1.05x**)	0.82 (**1.18x**)	0.215B (**1.29x**)	0.36 (**1.17x**)	0.048B (**1.54x**)
	Gr_p	15.80 (**1.06x**)	5.235B (**1.07x**)	0.66 (**1.47x**)	0.209B (**1.33x**)	0.27 (**1.56x**)	0.041B (**1.80x**)
Go	B_{ns}	33.96	5.422B (**36%**)	3.54	0.607B (**6%**)	1.63	0.211B (**1%**)
	B_s	33.1 (**1.03x**)	4.436B (**1.22x**)	2.88 (**1.23x**)	0.374B (**1.63x**)	0.99 (**1.65x**)	0.136B (**1.55x**)
	Gr	32.3 (**1.05x**)	4.283B (**1.27x**)	2.64 (**1.34x**)	0.343B (**1.77x**)	0.97 (**1.68x**)	0.126B (**1.67x**)
	Gr_p	32.8 (**1.04x**)	4.073B (**1.33x**)	2.71 (**1.31x**)	0.332B (**1.83x**)	0.97 (**1.68x**)	0.113B (**1.87x**)
		SEED$_1$		**SEED$_2$**		**SEED$_3$**	
Ep	B_{ns}	205.9	47.76B (**86%**)	8.45	2.071B (**61%**)	2.43	0.430B (**43%**)
	B_s	197.8 (**1.04x**)	46.03B (**1.04x**)	7.68 (**1.10x**)	1.740B (**1.19x**)	1.80 (**1.35x**)	0.323B (**1.33x**)
	Gr	197.8 (**1.04x**)	46.03B (**1.04x**)	7.68 (**1.10x**)	1.732B (**1.20x**)	1.81 (**1.34x**)	0.317B (**1.36x**)
	Gr_p	192.8 (**1.07x**)	44.75B (**1.07x**)	7.45 (**1.13x**)	1.691B (**1.22x**)	1.81 (**1.34x**)	0.311B (**1.38x**)
Go	B_{ns}	97.9	13.04B (**74%**)	7.25	0.715B (**34%**)	3.31	0.208B (**28%**)
	B_s	97.4 (**1.01**)	11.98B (**1.09x**)	5.50 (**1.32x**)	0.499B (**1.43x**)	2.37 (**1.40x**)	0.146B (**1.42x**)
	Gr	81.6 (**1.20x**)	11.51B (**1.13x**)	5.64 (**1.29x**)	0.500B (**1.43x**)	1.77 (**1.87x**)	0.114B (**1.82x**)
	Gr_p	81.6 (**1.20x**)	11.51B (**1.13x**)	5.31 (**1.37x**)	0.474B (**1.51x**)	1.61 (**2.06x**)	0.109B (**1.91x**)
		MagicRec$_1$		MagicRec$_2$		MagicRec$_3$	
Ep	B_{ns}	1524	40.64B (**19%**)	40.2	9.999B (**28%**)	7.66	2.048B (**24%**)
	B_s	1416 (**1.08x**)	18.06B (**2.25x**)	24.6 (**1.63x**)	5.414B (**1.85x**)	3.73 (**2.05x**)	1.014B (**2.02x**)
	Gr	1414 (**1.08x**)	17.65B (**2.30x**)	24.4 (**1.65x**)	5.121B (**1.95x**)	3.66 (**2.09x**)	0.929B (**2.20x**)
Go	B_{ns}	475.3	43.77B (**20%**)	18.2	7.345B (**26%**)	5.48	1.600B (**23%**)
	B_s	430.4 (**1.10x**)	20.19B (**2.17x**)	10.3 (**1.77x**)	3.922B (**1.87x**)	2.78 (**1.97x**)	0.786B (**2.04x**)
	Gr	427.2 (**1.11x**)	19.88B (**2.20x**)	9.9 (**1.84x**)	3.607B (**2.04x**)	2.62 (**2.09x**)	0.702B (**2.23x**)

The percentage value next to B_{ns} total i-cost shows the percentage of work done in the last level. Values in parentheses show the factor of improvement of the runtime over B_{ns}.

A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems

Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, Semih Salihoglu
Cheriton School of Computer Science, University of Waterloo
{amine.mhedhbi, pranjal.gupta, shahid.khaliq, semih.salihoglu}@uwaterloo.ca

Abstract— **Graph database management systems (GDBMSs) are highly optimized to perform fast traversals, i.e., joins of vertices with their neighbours, by indexing the neighbourhoods of vertices in adjacency lists. However, existing GDBMSs have system-specific and fixed adjacency list structures, which makes each system efficient on only a fixed set of workloads. We describe a new tunable indexing subsystem for GDBMSs, we call A+ indexes, with materialized view support. The subsystem consists of two types of indexes: (i) vertex-partitioned indexes that partition 1-hop materialized views into adjacency lists on either the source or destination vertex IDs; and (ii) edge-partitioned indexes that partition 2-hop views into adjacency lists on one of the edge IDs. As in existing GDBMSs, a system by default requires one forward and one backward vertex-partitioned index, which we call the primary A+ index. Users can tune the primary index or secondary indexes by adding nested partitioning and sorting criteria. Our secondary indexes are space-efficient and use a technique we call *offset lists*. Our indexing subsystem allows a wider range of applications to benefit from GDBMSs' fast join capabilities. We demonstrate the tunability and space efficiency of A+ indexes through extensive experiments on three workloads.**

I. Introduction

The term *graph database management system* (GDBMS) in its contemporary usage refers to data management software such as Neo4j [1], JanusGraph [2], TigerGraph [3], and GraphflowDB [4], [5] that adopt the property graph data model [6]. In this model, entities are represented by vertices, relationships are represented by edges, and attributes by arbitrary key-value properties on vertices and edges. GDBMSs have lately gained popularity among a wide range of applications from fraud detection and risk assessment in financial services to recommendations in e-commerce [7]. One reason GDBMSs appeal to users is that they are highly optimized to perform very fast joins of vertices with their neighbours. This is primarily achieved by using *adjacency list indexes* [8], which are join indexes that are used by GDBMSs' join operators.

Adjacency list indexes are often implemented using constant-depth data structures, such as the compressed sparse-row (CSR) structure, that partition the edge records into lists by source or destination vertex IDs. Some systems adopt a second level partitioning in these structures by edge labels. These partitionings provide constant time access to neighbourhoods of vertices and contrasts with tree-based indexes, such as B+ trees, which have logarithmic depth in the size of the data they index. Some systems further sort these lists according to some properties, which allows them to use fast intersection-based join algorithms, such as the novel intersection-based worst-case optimal (WCO) join algorithms [9]. However, a major shortcoming of existing GDBMSs is that systems make different but fixed choices about the partitioning and sorting criteria of their adjacency list indexes, which makes each system highly efficient on only a fixed set of workloads. This creates physical data dependence, as users have to model their data, e.g., pick their edge labels, according to the fixed partitioning and sorting criteria of their systems.

We address the following question: *How can the fast join capabilities of GDBMSs be expanded to a much wider set of workloads?* We are primarily interested in solutions designed for read-optimized GDBMSs. This is informed by a recent survey of users and applications of GDBMSs that we conducted [7], that indicated that GDBMSs are often used in practice to support read-heavy applications, instead of primary transactional stores. As our solution, we describe a tunable and space-efficient indexing subsystem for GDBMSs that we call *A+ indexes*. Our indexing subsystem consists of a *primary* index and optional *secondary* indexes that users can build. This is similar to relational systems that index relations in a primary B+ tree index on the primary key columns as well as optional secondary indexes on other columns. Primary A+ indexes are the default indexes that store all of the edge records in a database. Unlike existing GDMBSs, users can tune the primary A+ index of the system by adding arbitrary nested partitioning of lists into sublists and providing a sorting criterion per sublist. We store these lists in a nested CSR data structure, which provides constant time access to vertex neighborhoods that can benefit a variety of workloads.

We next observe that partitioning edges into adjacency lists is equivalent to creating multiple materialized views where each view is represented by a list or a sublist within a list. Similarly, the union of all adjacency lists can be seen as the coarsest view, which we refer to as the *global view*. In existing systems and primary A+ indexes, the global view is a trivial view that contains all of the edges in the graph. Therefore, one way a GDBMS can support an even wider range of workloads is by indexing other views inside adjacency lists. However storing and indexing views in secondary indexes results in data duplication and consumes extra space, which can be prohibitive for some views.

Instead of extending our system with general view functionality, our next contribution carefully identifies two sets of global views that can be stored in a highly space-efficient manner when partitioned appropriately into lists: (i) 1-hop views that

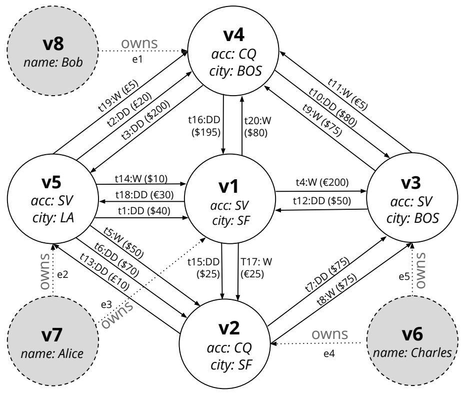


Fig. 1: Example financial graph.

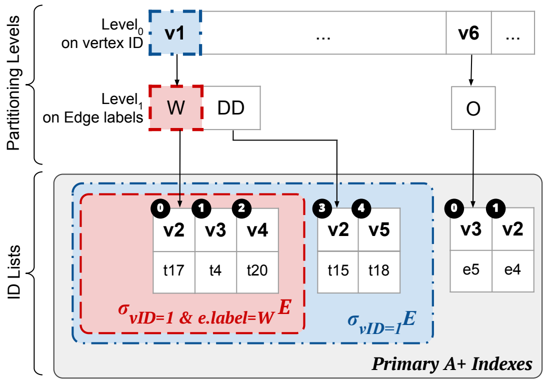


Fig. 2: Neo4j Adjacnecy List Indexes.

satisfy arbitrary predicates that are stored in *secondary vertex-partitioned A+ indexes*; and (ii) 2- hop views that are stored in *secondary edge-partitioned A+ indexes*, which extend the notion of neighborhood from vertices to edges, i.e., each list stores a set of edges that are adjacent to a particular edge. These two sets of views and their accompanying partitioning methods guarantee that the final lists that are stored in secondary A+ indexes are subsets of lists in the primary A+ index. Based on this property, we implement secondary A+ indexes by a technique we call *offset lists*, which identify each indexed edge by an offset into a list in the primary A+ index. Due to the sparsity, i.e., small average degrees, of real-world graphs, each list in the primary A+ index often contains a very small number of edges. This makes offset lists highly space-efficient, taking a few bytes per indexed edge instead of the ID lists in the primary index that store globally identifiable IDs of edges and neighbor vertices, each of which are often 8 bytes in existing systems. Similar to the primary A+ index, secondary indexes are implemented in a CSR structure that support nested partitioning, where the lower level is the offset lists. To further improve the space-efficiency of secondary A+ indexes, we identify cases when the secondary A+ indexes can share the partitioning levels of the primary A+ index.

We implemented A+ indexes inside the GraphflowDB in-memory GDBMS [5]. We describe the modifications we made to the optimizer and query processor of the system to use our indexes in query plans. We present examples of highly efficient plans that our system is able to generate using our indexing subsystem that do not exist in the plan spaces of existing systems. We demonstrate the tunability and space efficiency of A+ indexes by showing how to tune GraphflowDB to be highly efficient on three different workloads using either primary index reconfigurations or building secondary indexes with very small memory overhead. GraphflowDB is a read-optimized system that does not support transactions but allows non-transactional updates. Although update performance is not our focus, for completeness of our work, we report the update performance of A+ indexes in the longer version of our paper [10].

Figure 1 shows an example financial graph that we use as a running example throughout this paper. The graph contains vertices with `Customer` and `Account` labels. `Customer` vertices have `name` properties and `Account` vertices have `city` and `accountType(acc)` properties. From customers to accounts are edges with `Owns(O)` labels and between accounts are transfer edges with `Dir-Deposit(DD)` and `Wire(W)` labels with `amount(amt)`, `currency`, and `date` properties. We omit dates in the figure and give each transfer edge an ID such that $t_i.date < t_j.date$ if $i < j$.

II. Overview of Existing Adjacency List Indexes

Adjacency lists are accessed by GDBMS's join operators e.g., EXPAND in Neo4j or EXTEND/INTERSECT in GraphflowDB, that join vertices with neighbours. GDBMSs employ two broad techniques to provide fast access to adjacency lists while performing these joins:

(1) Partitioning: GDBMSs often partition their edges first by their source or destination vertex IDs, respectively in *forward* and *backward* indexes; this is the primary partitioning criterion.

Example 1: Consider the following 2-hop query, written in openCypher [11], that starts from a vertex with name "Alice". Below, a_i, c_j, and r_k are variables for the `Account` and `Customer` query vertices and query edges, respectively.

$$\textit{MATCH}\ \ c_1{-}[r_1]{-}{>}a_1{-}[r_2]{-}{>}a_2$$
$$\textit{WHERE}\ \ c_1.name = \text{`}Alice\text{'}$$

In every GDBMS we know of, this query is evaluated in three steps: (1) scan the vertices and find a vertex with `name` "Alice" and match a_1, possibly using an index on the `name` property. In our example graph, `v7` would match c_1; (2) access `v7`'s forward adjacency list, often with one lookup, to match $c_1 \rightarrow a_1$ edges; and (3) access the forward lists of matched a_1's to match $c_1 \rightarrow a_1 \rightarrow a_2$ paths.

Some GDBMSs employ further partitioning on each adjacency list, e.g., Neo4j [1] partitions edges on vertices and then by edge labels. Figure 2 showcases a high-level view of Neo4's paritioning levels and adjacency list index. Given the ID of a vertex v, this allows constant time access to: (i) all edges of v; and (ii) all edges of v with a particular label through the lower level lists e.g., all edges of v with label `Owns`.

Example 2: Consider the following query that returns all `Wire` transfers made from the accounts Alice `Owns`:

$$\textit{MATCH}\ \ c_1{-}[r_1{:}O]{-}{>}a_1{-}[r_2{:}W]{-}{>}a_2$$
$$\textit{WHERE}\ \ c_1.name = \text{`}Alice\text{'}$$

The "r_1:O" is syntactic sugar in Cypher for the `r1.label=`

Owns predicate. A system with lists partitioned by vertex IDs and edge labels can evaluate this query as follows. First, find v7, with name "Alice", and then access v7's Owns edges, often with a constant number of lookups and without running any predicates, and match a_1's. Finally access the Wire edges of each a_1 to match the a_2's.

(2) Sorting: Some systems further sort their most granular lists according to an edge property [2] or the IDs of the neighbours in the lists [4], [12]. Sorting enables systems to access parts of lists in time logarithmic in the size of lists. Similar to major and minor sorts in traditional indexes, partitioning and sorting keeps the edges in a sorted order, allowing systems to use fast intersection-based join algorithms, such as WCOJs [9] or sort-merge joins.

Example 3: Consider the following query that finds all 3-edge cyclical wire transfers involving Alice's account $v1$.

MATCH $a_1{-}[r_1{:}W]{-}{>}a_2{-}[r_2{:}W]{-}{>}a_3,\ a_3{-}[r_3{:}W]{-}{>}a_1$
WHERE $a_1.ID{=}v1$

In systems that implement worst-case optimal join (WCOJ) algorithms, such as EmptyHeaded [12] or GraphflowDB [4], this query is evaluated by scanning each v1$\rightarrow$$a_2$ Wire edge and intersecting the pre-sorted Wire lists of v1 and a_2 to match the a_3 vertices.

To provide very fast access to each list, lists are often accessed through data structures that have constant depth, such as a CSR instead of logarithmic depths of traditional tree-based indexes. This is achieved by having one level in the index for each partitioning criterion, so levels in the index are not constrained to a fixed size unlike traditional indexes, e.g., k-ary trees. Some systems choose alternative implementations. For example Neo4j has a linked list-based implementation where edges in a list are not stored consecutively but have pointers to each other, or JanusGraph uses a pure adjacency list design where there is constant time access to all edges of a vertex. In our implementation of A+ indexes (explained in Section III), we use CSR as our core data structure to store adjacency lists because it is more compact than a pure adjacency list design and achieves better locality than a linked list one. Finally, we note that the primary shortcoming of adjacency list indexes in existing systems is that GDBMSs adopt fixed system-specific partitioning and possibly sorting criteria, which limits the workloads that can benefit from their fast join capabilities.

III. A+ Indexes

There are three types of indexes in our indexing subsystem: (i) primary A+ indexes; (ii) secondary vertex-partitioned A+ indexes; and (iii) secondary edge-partitioned A+ indexes. Each index, both in our solution and existing systems, stores a set of adjacency lists, each of which stores a set of edges. We refer to the edges that are stored in the lists as *adjacent* edges, and the vertices that adjacent edges point to as *neighbour* vertices.

A. Primary A+ Indexes

The primary A+ indexes are by default the only available indexes. Similar to primary B+ tree indexes of relations in relational systems, these indexes are required to contain each edge in the graph, otherwise the system will not be able to answer some queries. Similar to the adjacency lists of existing GDBMSs, there are two primary indexes, one forward and one backward, and we use a nested CSR data structure partitioned first by the source and destination vertex IDs of the edges, respectively. In our implementation, by default we adopt a second level partitioning by edge labels and sort the most granular lists according to the IDs of the neighbours, which optimizes the system for queries with edge labels and matching cyclic subgraphs using multiway joins computed through intersections of lists. However, unlike existing systems, users can reconfigure the secondary partitioning and sorting criteria of primary A+ indexes to tailor the system to variety of workloads, with no or very minimal memory overhead.

1) Tunable Nested Partitioning

A+ indexes can contain nested secondary partitioning criteria on any categorical property of adjacent edges as well as neighbour vertices, such as edge or neighbour vertex labels, or the currency property on the edges in our running example. In our implementation we allow integers or enums that are mapped to small number of integers as categorical values. Edges with null property values form a special partition.

Example 4: Consider querying all wire transfers made in USD currency from Alice's account and the destination accounts of these transfers:

MATCH $c_1{-}[r_1{:}O]{-}{>}a_1{-}[r_2{:}W]{-}{>}a_2$
WHERE $c_1.name$ = '*Alice*', $r_2.currency{=}USD$

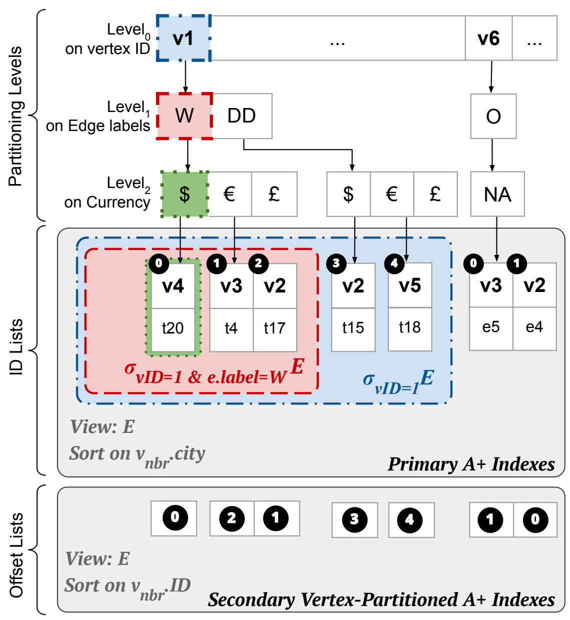
Here the query plans of existing systems that partition by edge labels will read all Wire edges from Alice's account and, for each edge, read its currency property and run a predicate to verify whether or not it is in USD.

Instead, if queries with equality predicates on the currency property are important and frequent for an application, users can reconfigure their primary A+ indexes to provide a secondary partitioning based on currency.

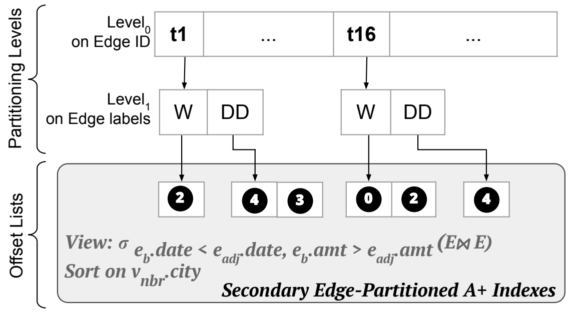
RECONFIGURE PRIMARY INDEXES
PARTITON BY $e_{adj}.label,\ e_{adj}.currency$
SORT BY $v_{nbr}.city$

In index creation and modification commands, we use reserved keywords e_{adj} and v_{nbr} to refer to adjacent edges and neighbours, respectively. The above command (ignore the sorting for now) will reconfigure the primary adjacency indexes to have two levels of partitioning after partitioning by vertex IDs: first by the edge labels and then by the currency property of these edges. For the query in Example 4, the system's join operator can now first directly access the lowest level partitioned lists of Alice's list, first by Wire and then by USD, without running any predicates.

Figure 3a shows the final physical design this generates as an example on our running example. We store primary indexes in nested CSR structures. Each provided nested partitioning adds a new partitioning level to the CSR, storing offsets to a particular slice of the next layer. After the partitioning levels, at the lowest level of the index are *ID lists*, which store the IDs of



(a) Example primary adjacency lists and secondary vertex-partitioned adjacency lists.


Partitioning Levels
Level$_0$ on Edge ID
t1
...
t16
...
Level$_1$ on Edge labels
W
DD
W
DD
Offset Lists
View: $\sigma_{e_b.date < e_{adj}.date,\ e_b.amt > e_{adj}.amt}(E \bowtie E)$
Sort on v_{nbr}.city
Secondary Edge-Partitioned A+ Indexes

(b) Example secondary edge-partitioned A+ Index.

Fig. 3: Example A+ indexes on our running example.

the edges and neighbour vertices. The ID lists are a consecutive array in memory that contains a set of nested sublists. For example, consider the second level partitions of the primary index in Figure 3a. Let L_W, L_{DD}, and L be the list of Wire, Dir-Deposit, and all edges of a vertex v, respectively. Then within L, which is the list between indices 0-4, are sub-lists L_W (0-2) and L_{DD} (3-4), i.e., $L = L_W \cup L_{DD}$.

2) Tunable ID List Sorting

The most granular sublists can be sorted according to one or more arbitrary properties of the adjacent edges or neighbour vertices, e.g., the date property of Transfer edges and the city property of the Account vertices of our running example. Similar to partitioning, edges with null values on the sorting property are ordered last. Secondary partitioning and sorting criteria together store the neighbourhoods of vertices in a particular sort order, allowing a system to generate WCOJ intersection-based plans for a wider set of queries.

Example 5: Consider the following query that searches for a three-branched money transfer tree, consisting of wire and direct deposit transfers, emanating from an account with vID v5 and ending in three sink accounts in the same city.

MATCH a_1−[:W]−>a_2−[:W]−>a_3 , a_1−[:W]−>a_4
a_1−[:DD]−>a_5−[:DD]−>a_6
WHERE $a_1.ID{=}v5$, $a_3.city{=}a_4.city{=}a_6.city$

If Wire and Dir-Deposit lists are partitioned or sorted by city, as in the above reconfiguration command, after matching $a_1 \rightarrow a_2$ and $a_1 \rightarrow a_5$, a plan can directly intersect two Wire lists of a_1 and a_2 and one Dir-Deposit list of a_5 in a single operation to find the flows that end up in accounts in the same city. Such plans are not possible with the adjacency list indexes of existing systems.

Observe that the ability to reconfigure the system's primary A+ indexes provides more physical data independence. Users do not have to model their datasets according to the system's default physical design and changes in the workloads can be addressed simply with index reconfigurations.

B. Secondary A+ Indexes

Many indexes in DBMSs can be thought of as data structures that give fast access to *views*. In our context, each sublist in the primary indexes is effectively a *view* over edges. For example, the red dashed list in Figure 3a is the $\sigma_{\text{srcID=v1 \&e.label=Wire}}$Edge view while the green dotted box encloses a more selective view corresponding to $\sigma_{\text{srcID=1 \& e.label=wire \& curr=USD}}$Edge. Each nested sublist in the lowest-level ID lists is a view with one additional equality predicate. One can also think of the entire index as indexing a *global view*, which for primary indexes is simply the Edge table. Therefore the views that can be obtained through the system's primary A+ index are constrained to views over the edges that contain an equality predicate on the source or destination ID (due to vertex ID partitioning) and one equality predicate for each secondary partitioning criterion.

To provide access to even wider set of views, a system should support more general materialized views and index these in adjacency list indexes. However, supporting additional views and materializing them inside additional adjacency list indexes requires data duplication and storage. We next identify two classes of global views and ways to partition these views that are conducive to a space-efficient implementation: (i) 1-hop views that are stored in secondary vertex-partitioned A+ indexes; and (ii) 2-hop views that are stored in secondary edge-partitioned A+ indexes. These views and partitioning techniques generate lists that are subsets of the lists in the primary index, which allows us to store them in space-efficient *offset lists* that exploit the small average-degree of real-world graphs and use a few bytes per indexed edge. In Sections III-B1 and III-B2 we first describe our logical views and how these views are partitioned into lists. Similar to the primary A+ index, these lists are stored in CSR-based structures. Section III-B3 describes our offset list-based storage and how we can further increase the space efficiency of secondary A+ indexes by avoiding the partitioning levels of the CSR structure when possible.

1) Secondary Vertex-Partitioned A+ Indexes: 1-hop Views

Secondary vertex-partitioned indexes store 1-hop views, i.e., 1-hop queries, that contain arbitrary selection predicates on the edges and/or source or destination vertices of edges. These views cannot contain other operators, such as group by's, aggregations, or projections, so their outputs are a subset of the original edges. Secondary vertex-partitioned A+ indexes store these 1-hop views first by partitioning on vertex IDs (source or destination) and then by the further partitioning and sorting options provided by the primary A+ indexes. In order to use secondary vertex-partitioned A+ indexes, users need to first define the 1-hop view, and then define the partitioning structure and sorting criterion of the index.

Example 6: Consider a fraud detection application that searches money flow patterns with high amount of transfers, say over 10000 USDs. We can create a secondary vertex-partitioned index to store those edges in lists, partitioned first by vertices and then possibly by other properties and in a sorted manner as before.

CREATE 1–HOP VIEW $LargeUSDTrnx$
MATCH $v_s\text{−}[e_{adj}]\text{−}>v_d$
WHERE $e_{adj}.currency{=}USD,\ e_{adj}.amt{>}10000$
INDEX AS FW–BW
PARTITION BY $e_{adj}.label$ *SORT BY* $v_{nbr}.ID$

Above, v_s and v_d are keywords to refer to the source and destination vertices, whose properties can be accessed in the WHERE clause. FW and BW are keywords to build the index in the forward or backward direction, a partitioning option given to users. FW-BW indicates indexing in both directions. The inner-most (i.e., most nested) sublists of the resulting index materializes a view of the form $\sigma_{\text{srcID=* \& elabel=* \& curr=USD \& amount > 10000}}$`Edge`. If such views or views that correspond to other levels of the index appear as part of queries, the system can directly access these views in constant time and avoid evaluating the predicates in these views.

2) Secondary Edge-Partitioned A+ Indexes: 2-hop Views

Secondary edge-partitioned indexes store 2-hop views, i.e., results of 2-hop queries. As before, these views cannot contain other operators, such as group by's, aggregations, or projections, so their outputs are a subset of 2-paths. The view has to specify a predicate and that predicate has to access properties of both edges in 2-paths (as we momentarily explain, otherwise the index is redundant). Secondary edge-partitioned indexes store these 2-hop views first by partitioning *on edge IDs* and then, as before, by the same partitioning and sorting options provided by the primary A+ indexes. Vertex-partitioned indexes in A+ indexes and existing systems provide fast access to the adjacency of a vertex given the ID of that vertex. Instead, our edge-partitioned indexes provide fast access to the *adjacency of an edge* given the ID of that edge. This can benefit applications in which the searched patterns concern relations between two adjacent, i.e., consecutive, edges. We give an example:

Example 7: Consider the following query, which is the core of an important class of queries in financial fraud detection.

MATCH $a_1\text{−}[r_1{:}]\text{−}>a_2\text{−}[r_2{:}]\text{−}>a_3\text{−}[r_3{:}]\text{−}>a_4$
WHERE $r_1.eID{=}t13,$
$r_1.date{<}r_2.date\ \&\ r_2.amt{<}r_1.amt{<}r_2.amt{+}\alpha\ \&$
$r_2.date{<}r_3.date\ \&\ r_3.amt{<}r_2.amt{<}r_3.amt{+}\alpha$

The query searches a three-step money flow path from a transfer edge with `eID t13` where each additional transfer (`Wire` or `Dir-Deposit`) happens at a later date and for a smaller amount of at most α, simulating some money flowing through the network with intermediate hops taking cuts.
The predicates of this query compare properties of an edge on a path with the previous edge on the same path. Consider a system that matches r_1 to `t13`, which is from vertex `v2` to `v5`. Existing systems have to read transfer edges from `v5` and filter those that have a later `date` value than `t13` and also have the appropriate `amount` value. Instead, when the next query edge to match r_2 has predicates depending on the query edge r_1, these queries can be evaluated much faster if adjacency lists are partitioned by edge IDs: a system can directly access the *destination-forward adjacency list of* `t13` in constant time, i.e., edges whose `srcID` are `v5`, that satisfy the predicate on the `amount` and `date` properties that depend on `t13`, and perform the extension. Our edge-partitioned indexes allow the system to generate plans that perform this much faster processing. Note that in an alternative design we can partition the same set of 2-hop paths by vertices instead of edges. However, this would store the same number of edges but would be less efficient during query processing. To see this, suppose a system first matches r_1 to the edge $(v_2)\text{−}[\texttt{t13}]\text{−}>(v_5)$ and consider extending this edge. The system can either extend this edge by one more edge to r_2, which would require looking up the 2-hop edges of v_2 and find those that have `t13` as the first edge. This is slower than directly looking up the same edges using `t13` in an edge-partitioned list. Alternatively the system can extend `t13` by two more edges to $[r_2]\text{−}>a_3\text{−}[r_3]\text{−}>a_4$ by accessing the 2-hop edges of v_5 but would need to run additional predicates to check if the edge matching r_2 satisfy the necessary predicates with `t13`, so effectively processing all 2-paths of v_5 and running additional predicates, which are also avoided in an edge-partitioned list.

There are three possible 2-paths, $\rightarrow\rightarrow$, $\rightarrow\leftarrow$, and $\leftarrow\leftarrow$. Partitioning these paths by different edges gives four unique possible ways in which an edge's adjacency can be defined:

1) Destination-FW: $v_s\text{−}[e_b]{\rightarrow}v_d\text{−}[e_{adj}]{\rightarrow}v_{nbr}$
2) Destination-BW: $v_s\text{−}[e_b]{\rightarrow}v_d{\leftarrow}[e_{adj}]\text{−}v_{nbr}$
3) Source-FW: $v_{nbr}\text{−}[e_{adj}]{\rightarrow}v_s\text{−}[e_b]{\rightarrow}v_d$
4) Source-BW: $v_{nbr}{\leftarrow}[e_{adj}]\text{−}v_s\text{−}[e_b]{\rightarrow}v_d$

e_b, for "bound", is the edge that the adjacency lists will be partitioned by, and v_s and v_d refer to the source and destination vertices of e_b, respectively. For example, the Destination-FW adjacency lists of edge $e(s,d)$ stores the forward edges of d. To facilitate the fast processing described above for the money flow queries in Example 7, we can create the following index:

CREATE 2–HOP VIEW MoneyFlow
MATCH $v_s\text{−}[e_b]{\rightarrow}v_d\text{−}[e_{adj}]{\rightarrow}v_{nbr}$
WHERE $e_b.date{<}e_{adj}.date,\ e_{adj}.amt{<}e_b.amt$
INDEX AS PARTITION BY $e_{adj}.label$ SORT BY $v_{nbr}.city$

The location of the variable e_b in the query implicitly defines the type of partitioning, which in this example is Destination-FW. This query creates an index that, for each edge t_i, stores the forward edges from t_i's destination vertex which have a later `date` and a smaller `amount` than t_i, partitioned by the labels of their adjacent edges and sorted by the `city` property of the neighbouring vertices, i.e., the vertex that is not shared with t_i. Figure 3b shows the lists this index stores on our running example. The inner-most lists in the index correspond to the view: $\sigma_{\text{e}_b\text{.ID=* \& e}_{adj}\text{.label=* \& e}_b\text{.date} < \text{e}_{adj}\text{.date \& e}_b\text{.amt} > \text{e}_{adj}\text{.amt}}(\rho_{e_b}(E) \bowtie \rho_{e_{adj}}(E))$. `E` abbreviates `Edge` and the omitted join predicate is e$_b$.dstID=e$_{adj}$.srcID. Readers can verify that, in presence of this index, a GDBMS can evaluate the money flow query from Example 4 (ignoring the predicate with α) by scanning only one edge. It only scans `t13`'s list which contains a single edge `t19`. In contrast, even if all `Transfer` edges are accessible using a vertex-partitioned A+ index, a system would access 9 edges after scanning `t13`.

Observe that unlike vertex-partitioned A+ indexes, an edge e in the graph can appear in multiple adjacency lists in an edge-partitioned index. For example, in Figure 3b, edge `t17` (having offset 2) appears both in the adjacency list for `t1` as well as `t16`. As a consequence, when defining edge-partitioned indexes, users have to specify a predicate that accesses properties of both edges in the 2-hop query. This is because if all the predicates are only applied to a single query edge, say $v_s{-}[e_b]{\rightarrow}v_d$, then we would redundantly generate duplicate adjacency lists. Instead, defining a secondary vertex-partitioned A+ index would give the same access path to the same lists without this redundancy.

Consider the following example:

```
CREATE 2–HOP VIEW Redundant
MATCH $v_s-[e_b]\rightarrow v_d-[e_{adj}]\rightarrow v_{nbr}$
WHERE $e_{adj}.amt<10000$
```

In absence of an INDEX AS command, views are only partitioned by edge IDs. Consider the account `v2` in our running example graph in Figure 1. For each of the four incoming edges of `v2`, namely `t5`, `t6`, `t15`, and `t17`, this index would contain the same adjacency list that consists of all outgoing edges of `v2`: {`t7`,`t8`, `t13`}, because the predicate is only on a single edge. Instead, a user can define a vertex-partitioned A+ index with the same predicate and `v2`'s list would provide an access path to the same edges {`t7`,`t8`,`t13`}.

We further note that although we will describe a space-efficient physical implementation of these indexes momentarily, the total number of edges in edge-partitioned indexes can be as large as the sum of the squares of degrees unless a selective predicate is used, which can be prohibitive for an in-memory system. In our evaluations, we will assume a setting where a selective enough predicate is used. For 2-hop views that do not have selective predicates, a system should resort to partial materialization of these views to reduce the memory consumption under user-specified levels. Partial materialized views is a technique from relational systems that has been introduced in reference [13], where parts of the view is materialized and others are evaluated during runtime. We have left the integration of this technique to future work.

3) *Offset List-based Storage of Secondary A+ Indexes*

The predominant memory cost of primary indexes is the storage of the IDs of the adjacent edges and neighbour vertices. Because the IDs in these lists globally identify vertices and edges, their sizes need to be logarithmic in the number of edges and vertices in the graph, and are often stored as 4 to 8 byte integers in systems. For example, in our implementation, edge IDs take 8 and neighbour IDs take 4 bytes.

In contrast, the lists in both secondary vertex- and edge-partitioned indexes have an important property, which can be exploited to reduce their memory overheads: they are subsets of some ID list in the primary indexes. Specifically, a list L_v that is bound to v_i in a secondary vertex-partitioned index is a subset of one of v_i's ID lists. A list L_e that is bound to $e = (v_s, v_d)$ in a secondary edge-partitioned index is a subset of either v_s's or v_d's primary list, depending on the direction of the index, e.g., v_d's list for a Destination-FW list. Recall that in our CSR-based implementation, the ID lists of each vertex are contiguous. Therefore, instead of storing an (edge ID, neighbour ID) pair for each edge, we can store a single offset to an appropriate ID list. We call these lists *offset lists*. The average size of the ID lists is proportional to the average degree in the graph, which is often very small, in the order of tens or hundreds, in many real world graph data sets. This important property of real world graphs has two advantages:

1) Offsets only need to be list-level identifiable and can take a small number of bytes which is much smaller than a globally identifiable (edge ID, neighbour ID) pair.
2) Reading the original (edge ID, neighbour ID) pairs through offset lists require an indirection and lead to reading not-necessarily consecutive locations in memory. However, because the ID list sizes are small, we still get very good CPU cache locality.

An alternative implementation design here is to use a bitmap instead of offset lists. A bitmap can identify whether each edge in the lists of the primary A+ index is a secondary A+ index. This design has the shortcoming that it cannot support the cases when the sorting criterion of secondary A+ indexes is different than the primary index. However when the sorting criteria are the same, this is also a reasonable design point. This has the advantage that when the predicates in the lists are not very selective, bitmaps can be even more compact than offset lists, as they require a single bit for each edge. However reading the edges would now require additional bitmask operations. In particular, irrespective of the actual number of edges stored in a secondary index, the system would need to perform as many bitmask operations as the number of edges in the lists of the primary index. Therefore as predicates in secondary indexes get more selective, bitmaps would progressively lose their storage advantage over offset lists and at the same time progressively perform worse in terms of access time.

We implement each secondary index in one of two possible ways, depending on whether the index contains any predicates and whether its partitioning structure matches the secondary structure of the primary A+ indexes.

- *With no predicates and same partitioning structure*: In this case, the only difference between the primary and the secondary index is the final sorting of the edges. Specifically, both indexes have identical partitioning levels, with identical CSR offsets, and the same set of edges in each inner-most ID/offset sublists, but they sort these sublists in a different order. Therefore we can use the partitioning levels of the primary index also to access the lists of the secondary index and save space. Figure 3a gives an example. The bottom offset lists are for a secondary vertex-partitioned index that has the same partitioning structure as the primary index but sorts on neighbors' IDs instead of neighbors' `city` properties. Recall that since edge-partitioned indexes need to contain predicates between adjacent edges, this storage can only be used for vertex-partitioned indexes.
- *With predicates or different partitioning structure*: In this case, the inner-most sublists of the indexes may contain different sets of edges, so the CSR offsets in the partitioning levels of the primary index cannot be reused and we store new partitioning levels as shown in Figure 3b.

We give the details of the memory page structures that store ID and offset lists in Section IV.

IV. Implementation Details

We implemented our indexing subsystem in GraphflowDB [4], [5] and describe our changes to the system to enable the use of A+ indexes for fast join processing.

A. Query Processor, Optimizer and Index Store

A+ indexes are used in evaluating subgraph pattern component of queries, which is where the queries' joins are described. We give an overview of the join operators that use A+ indexes and the optimizer of the system. Reference [4] describes the details of the Extend/Intersect operator and the DP join optimizer of the system in absence of A+ indexes.

Join Operators: Extend/Intersect (E/I) is the primary join operator of the system. Given a query $Q(V_Q, E_Q)$ and an input graph $G(V, E)$, let a *partial k-match* of Q be a set of vertices of V assigned to the projection of Q onto a set of k query vertices. We denote a sub-query with k query vertices as Q_k. E/I is configured to intersect $z{\geq}1$ adjacency lists that are sorted on neighbour IDs. The operator takes as input (k-1)-matches of Q, performs a z-way intersection, and extends them by a single query vertex to k-matches. For each (k-1)-match t, the operator intersects z adjacency lists of the matched vertices in t and extends t with each vertex in the result of this intersection to produce k-matches. If z is one, no intersection is performed, and the operator simply extends t to each vertex in the adjacency list. The system uses E/I to generate plans that contain WCOJ multi-way intersections.

To generate plans that use A+ indexes, we first extended E/I to take adjacency lists that can be partitioned by edges as well as vertices. We also added a variant of E/I that we call Multi-Extend, that performs intersections of adjacency lists that are sorted by properties other than neighbour IDs and extends partial matches to more than one query vertex.

Dynamic Programming (DP) Optimizer and Index Store: GraphflowDB has a DP-based join optimizer that enumerates queries one query vertex at a time [4]. We extended the system's optimizer to use A+ indexes as follows. For each $k{=}1, ..., m{=}|V_Q|$, in order, the optimizer finds the lowest-cost plan for each sub-query Q_k in two ways: (i) by considering extending every possible sub-query Q_{k-1}'s (lowest-cost) plan by an E/I operator; and (ii) if Q has an equality predicate involving $z{\geq}2$ query edges, by considering extending smaller sub-queries Q_{k-z} by a Multi-Extend operator. At each step, the optimizer considers the edge and vertex labels and other predicates together, since secondary A+ indexes may be indexing views that contain predicates other than edge label equality. When considering possible Q_{k-z} to Q_k extensions, the optimizer queries the Index Store to find both vertex- and edge-partitioned indexes, $I_1, ..., I_t$, that can be used. Index Store maintains the metadata of each A+ index in the system such as their type, partitioning structure, and sorting criterion, as well as additional predicates for secondary indexes. An index I_ℓ can potentially be used in the extension if the edges in the lists in a level j of I_ℓ satisfy two conditions: (i) extend partial matches of Q_{k-z} to Q_k, i.e., can be bound to a vertex or edge in Q_{k-z} and match a subset of the query edges in Q_z; and (ii) the predicates $p_{\ell,j}$ satisfied in these lists subsume the predicate p_Q (if any) that is part of this extension. We search for two types of predicate subsumption. First is conjunctive predicate subsumption. If both $p_{\ell,j}$ and p_Q are conjunctive predicates, we check if each component of $p_{\ell,j}$ matches a component of p_Q. Second is range subsumption. If p_Q and $p_{\ell,j}$ or one of their components are range predicates comparing a property against a constant, e.g., $e_{adj}.amt{>}15000$ and $e_{adj}.amt{>}10000$, respectively, we check if the range in $p_{\ell,j}$ is less selective than p_Q.

Then for each possible index combination retrieved, the optimizer enumerates a plan for Q_k with: (i) an E/I or Multi-Extend operator; and (ii) possibly a Filter operator if there are any predicates that are not fully satisfied during the extension (e.g., if $p_{\ell,j}$ and p_Q are conjunctive but $p_{\ell,j}$ does not satisfy all components of p_Q). If the Q_{k-z} to Q_k extension requires using multiple indices, so requires performing an intersection, then the optimizer also checks that the sorting criterion on the indices that are returned are the same. Otherwise, it discards this combination. The systems' cost metric is *intersection cost* (i-cost), which is the total estimated sizes of the adjacency lists that will be accessed by the E/I and Multi-Extend operators in a plan.

We note that our optimizer extension to use A+ indexes is similar to the classic *System R-style* approach to enumerate plans that use views composed of select-project-join queries directly in a DP-based join optimizer [14], [15]. This approach also performs a bottom up DP enumeration of join orders of a SQL query Q and for a sub-query Q' of Q, considers evaluating Q' by joining a smaller Q'' with a view V. The primary difference is that GraphflowDB's join optimizer enumerates plans for progressively larger queries that contain, in relational terms, one more column instead of one more table (see reference [4] for details). Other GDBMSs that use bottom up join optimizers can be extended in a similar way if they implement A+ indexes. For example, Neo4j also uses a mix of DP and greedy bottom up enumerator [1] called *iterative DP*, which is based on reference [16]. However, extending the optimizers of GDBMSs that use other techniques might require other approaches, e.g., RedisGraph, which converts Cypher queries into GraphBLAS linear algebra expression [17] and optimizes this expression.

We also note that we implemented a limited form of predicate subsumption checking. The literature on query optimization using views contains more general techniques for logical implication of predicates between queries and views [15], [18], [19], e.g., detecting that $A > B$ and $B > C$ imply $A > C$. These techniques can enhance our implementation and we have not integrated such techniques within the scope of our paper.

B. Details of Physical Storage

Primary and secondary vertex-partitioned A+ indexes are implemented using a CSR for groups of 64 vertices and allocates one data page for each group. Vertex IDs are assigned consecutively starting from 0, so given the ID of v, with a division and mod operation we can access the second partitioning level of the index storing CSR offsets of v. The CSR offsets in the final partitioning level point to either ID lists in the case of the primary A+ indexes or offset lists in the case of secondary A+ indexes. The neighbour vertex and edge ID lists are stored as 4 byte integer and 8 byte long arrays, respectively. In contrast, the offset lists in both cases are stored as byte arrays by default. Offsets are fixed-length and use the maximum number of bytes needed for any offset across the lists of the 64 vertices, i.e. it is the logarithm of the length of the longest of the 64 lists rounded to the next byte.

C. Index Maintenance

Each vertex-partitioned data page, storing ID lists or offset lists, is accompanied with an update buffer. Each edge addition $e{=}(u, v)$ is first applied to the update buffers for u's and v's pages in the primary indexes. Then we go over each secondary vertex-partitioned A+ index I_V in the INDEX STORE. If I_V indexes a view that contains a predicate p, we first apply p to see if e passes the predicate. If so, or if I_V does not contain a predicate, we update the necessary update buffers for the offset list pages of u and/or v. The update buffers are merged into the actual data pages when the buffer is full. Edge deletions are handled by adding a "tombstone" for the location of the deletion until a merge is triggered.

Maintenance of an edge-partitioned A+ index I_E is more involved. For an edge insertion $e{=}(u, v)$, we perform two separate operations. First, we check to see if e should be inserted into the adjacency list of any adjacent edge e_b by running the predicate p of I_E on e and e_b. For example, if I_E is defined as Destination-FW, we loop through all the backward adjacent edges of u using the system's primary index. This is equivalent to running two *delta-queries* as described in references [5], [20] for a continuous 2-hop query. Second, we create a new list for e and loop through another set of adjacency lists (in our example v's forward adjacency list in `D`) and insert edges into e's list.

D. Index Selection

Our work focuses on the design and implementation of a tunable indexing subsystem so that users can tailor a GDBMS to be highly efficient on a wide range of workloads. However, an important aspect of any DBMS is to help users pick indexes from a space of indexes that can benefit their workloads. Given a workload W, the space of A+ indexes that can benefit W can be enumerated by enumerating each 1-hop and 2-hop sub-query Q' of each query Q in W and identifying the equality predicates on categorical properties of these sub-queries, which are candidates for partitioning levels, and non-equality predicates on other properties, which are candidates for sorting criterion (any predicate is also a candidate predicate of a global view). Given a workload W and possibly a space budget B, one approach from prior literature to automatically select a subset of these candidate indices that are within the space budget B is to perform a "what if" index simulation to see the effects of this candidate indices on the estimated costs of plans. For example, this general approach is used in Microsoft SQL Server's AutoAdmin [21] tool. We do not focus on the problem of recommending a subset of these indexes to users. There are several prior work on index and materialized view recommendation [21], [22], [23], [24], [25], which are complementary to our work. We leave the rigorous study of this problem to future work.

V. EVALUATION

The goal of our experiments is two-fold. First, we demonstrate the tunability and space-efficiency of A+ indexes on three very different popular applications that GDBMSs support: (i) labelled subgraph queries; (ii) recommendations; and (iii) financial fraud detection. By either tuning the system's primary A+ index or adding secondary A+ indexes, we improve the performance of the system significantly, with minimal memory overheads. Second, we evaluate the performance and memory overhead tradeoffs of different A+ indexes on these workloads. Finally, as a baseline comparison, we benchmark our performance against Neo4j [1] and TigerGraph [3], two commercial GDBMSs that have fixed adjacency list structures. For completeness of our work, we also evaluate the maintenance performance of our indexes in the longer version of our paper [10].

A. Experimental Setup

We use a single machine with two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical ones. Table I shows the datasets used. We ran our experiments on all datasets and report numbers on a subset of datasets due to limited space. Our datasets include social, web, and Wikipedia knowledge graphs, which have a variety of graph topologies and sizes ranging from several million edges to over a hundred-million edges. A dataset G, denoted as $G_{i,j}$, has i and j randomly generated vertex and edge labels, respectively. We omit i and j when both

Name	#Vertices	#Edges	Avg. degree
Orkut (Ork)	3.0M	117.1M	39.03
LiveJournal (LJ)	4.8M	68.5M	14.27
Wiki-topcats (WT)	1.8M	28.5M	15.83
BerkStan (Brk)	685K	7.6M	11.09

TABLE I: Datasets used.

are set to 1. We use query workloads drawn from real-world applications: (i) edge- and vertex-labelled subgraph queries; (ii) Twitter MagicRecs recommendation engine [26]; and (iii) fraud detection in financial networks. For all index configurations (Configs), we report either the index reconfiguration (IR) or the index creation (IC) time of the newly added secondary indexes. All experiments use a single thread except the creation of edge-partitioned indexes, which uses 16 threads.

B. Primary A+ Index Reconfiguration

We first demonstrate the benefit and overhead tradeoff of tuning the primary A+ index in two different ways: (i) by only changing the sorting criterion; and (ii) by adding a new secondary partitioning. We used a popular subgraph query workload in graph processing that consists of labelled subgraph queries where both edges and vertices have labels. We followed the data and subgraph query generation methodology from several prior work [4], [27]. We took the 14 queries from reference [4] (omitted due to space reasons), which contain acyclic and cyclic queries with dense and sparse connectivity with up to 7 vertices and 21 edges. This query workload had only fixed edge labels in these queries, for which GraphflowDB's default indexes are optimized. We modify this workload by also fixing vertex labels in queries. We picked the number of labels for each dataset to ensure that queries would take time in the order of seconds to several minutes. Then we ran GraphflowDB on our workload on each of our datasets under three Configs:

1) `D`: system's default configuration, where edges are partitioned by edge labels and sorted by neighbour IDs.
2) $\mathtt{D}_s$: keeps `D`'s secondary partitioning but sorts edges first by neighbour vertex labels and then on neighbour IDs.
3) $\mathtt{D}_p$: keeps `D`'s sorting criterion and edge label partitioning but adds a new secondary partitioning on neighbour vertex labels.

Table II shows our results. We omit Q14, which had very few or no output tuples on our datasets. First observe that $\mathtt{D}_s$ outperforms `D` on all of the 52 settings and by up to 10.38x and without any memory overheads as $\mathtt{D}_s$ simply changes the sorting criterion of the indexes. Next observe that by adding an additional partitioning level on `D`, the joins get even faster consistently across all queries, e.g., SQ_{13} improves from 2.36x to 3.84x on $\mathrm{Ork}_{8,2}$, as the system can directly access edges with a particular edge label and neighbour label using $\mathtt{D}_p$. In contrast, under $\mathtt{D}_s$, the system performs binary searches inside lists to access the same set of edges. Even though $\mathtt{D}_p$ is a reconfiguration, so does not index new edges, it still has minor memory overhead ranging from 1.05x to 1.15x because of the cost of storing the new partitioning layer. This demonstrates the effectiveness of tuning A+ indexes to optimize the system to be much more efficient on a different workload without any data remodelling, and with no (or minimal) memory overhead.

C. Secondary Vertex-Partitioned A+ Indexes

We next study the tradeoffs offered by secondary vertex-partitioned A+ indexes. We use two sets of workloads drawn from real-world applications that benefit from using both the system's primary A+ index as well as a secondary vertex-partitioned A+ index. Our two applications highlight two separate benefits users get from vertex-partitioned A+ indexes: (i) decreasing the amount of predicate evaluation; and (ii) allowing the system to generate new WCOJ plans that are not possible with only the primary A+ index.

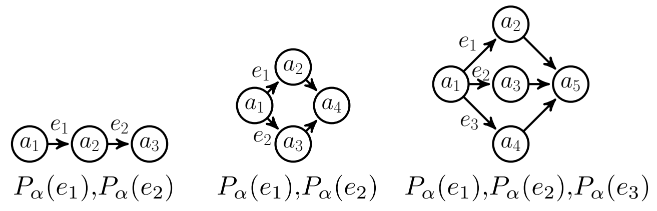
1) Decreasing Predicate Evaluations

In this experiment, we take a set of the queries drawn from the MagicRecs workload described in reference [26]. MagicRecs was a recommendation engine that was developed at Twitter that looks for the following patterns: for a given user a_1, it searches for users $a_2 ... a_k$ that a_1 has started following recently, and finds their common followers. These common followers are then recommended to a_1. We set k=2,3 and 4. Our queries, MR_1, MR_2, and MR_3 are shown in Figure 4. These queries have a time predicate on the edges starting from a_1 which can benefit from indexes that sort on time. MR_2, and MR_3 are also cyclic can benefit from the default sorting order of the primary A+ index on neighbour IDs. We evaluate our queries on all of our datasets on two Configs. The first Config consists of the system's primary A+ index denoted by `D` as before. The second Config, denoted by $\mathtt{D}$+$\mathtt{VP}_t$ adds on top of D a new secondary vertex-partitioned index $\mathtt{VP}_t$ in the forward direction that: (i) has the same partitioning structure as primary forward A+ index so shares the same partitioning levels as the primary A+ index; and (ii) sorts the inner-most lists on the `time` property of edges. In our queries we set the value of α in the time predicate to have a 5% selectivity. For MR_3, on datasets LJ and Ork, we fix a_1 to 10000 and 7000 vertices, respectively, to run the query in a reasonable time.

Table III shows our results. First observe that despite indexing all of the edges again, our secondary index has only 1.08x memory overhead because: (i) the secondary index can share the partitioning levels of the primary index; and (ii) the secondary index stores offset lists which has a low-memory footprint. In return, we see up to 10.6x performance benefits. We note that GraphflowDB uses exactly the same plans under both Configs that start reading a_1, extends to its neighbours and finally performs a multiway intersection (except for MR_1, which is followed by a simple extension). The only difference is that under $\mathtt{D}$+$\mathtt{VP}_t$ the first set of extensions require fewer predicate evaluation because of accessing a_1's adjacency list in $\mathtt{VP}_t$, which is sorted on time. Overall this memory performance tradeoff demonstrates that with minimal overheads of an additional index, users obtain significant performance benefits on applications like MagicRecs that require performing complex joins for which the system's primary indexes are not tuned.

		SQ_1	SQ_2	SQ_3	SQ_4	SQ_5	SQ_6	SQ_7	SQ_8	SQ_9	SQ_{10}	SQ_{11}	SQ_{12}	SQ_{13}	Mm	IR
$Ork_{8,2}$	`D`	1.68	5.47	3.66	1.30	1.58	1.45	1.73	2.49	0.95	17.74	7536.9	54.86	131.5	2778	-
	$\texttt{D}_s$	0.91	3.12	2.04	1.19	1.05	1.22	1.33	1.51	0.77	4.89	725.9	41.92	55.62	2778	38.90
		(1.85x)	**(1.75x)**	**(1.79x)**	**(1.09x)**	**(1.50x)**	**(1.19x)**	**(1.30x)**	**(1.65x)**	**(1.23x)**	**(3.63x)**	**(10.38x)**	**(1.31x)**	**(2.36x)**	**(1.0x)**	-
	$\texttt{D}_p$	0.68	2.61	1.35	0.97	0.77	0.60	1.30	1.46	0.60	3.89	704.9	28.32	34.22	3106	27.71
		(2.48x)	**(2.10x)**	**(2.71x)**	**(1.34x)**	**(2.05x)**	**(2.44x)**	**(1.33x)**	**(1.71x)**	**(1.25x)**	**(4.56x)**	**(10.69x)**	**(1.94x)**	**(3.84x)**	**(1.12x)**	-
$LJ_{2,4}$	`D`	1.47	7.87	6.46	1.69	1.59	1.60	1.91	3.35	4.07	41.54	807.8	397.1	468.8	1016	-
	$\texttt{D}_s$	1.45	6.22	5.42	1.49	1.51	1.52	1.40	2.39	2.82	28.07	241.2	268.6	259.2	1016	20.83
		(1.01x)	**(1.27x)**	**(1.19x)**	**(1.13x)**	**(1.05x)**	**(1.05x)**	**(1.36x)**	**(1.40x)**	**(1.44x)**	**(1.48x)**	**(3.35x)**	**(1.48x)**	**(1.81x)**	**(1.0x)**	-
	$\texttt{D}_p$	1.04	5.18	4.64	1.09	0.98	1.08	1.07	1.85	2.26	25.86	235.63	235.85	161.82	1164	19.92
		(1.41x)	**(1.52x)**	**(1.39x)**	**(1.55x)**	**(1.62x)**	**(1.48x)**	**(1.79x)**	**(1.81x)**	**(1.80x)**	**(1.61x)**	**(3.43x)**	**(1.68x)**	**(2.90x)**	**(1.15x)**	-
$WT_{4,2}$	`D`	0.61	4.59	5.48	0.84	1.17	0.90	0.73	11.25	2.85	1116.2	340.0	487.8	767.5	713	-
	$\texttt{D}_s$	0.37	2.43	3.50	0.69	0.71	0.65	0.61	3.93	1.36	697.9	77.11	319.0	386.8	713	8.70
		(1.65x)	**(1.89x)**	**(1.56x)**	**(1.22x)**	**(1.65x)**	**(1.38x)**	**(1.20x)**	**(2.87x)**	**(2.09x)**	**(1.60x)**	**(4.41x)**	**(1.53x)**	**(1.98x)**	**(1.0x)**	-
	$\texttt{D}_p$	0.32	2.09	3.05	0.55	0.59	0.54	0.61	2.86	1.09	639.7	76.32	259.1	235.7	795	6.25
		(1.91x)	**(2.20x)**	**(1.80x)**	**(1.53x)**	**(1.99x)**	**(1.66x)**	**(1.21x)**	**(3.94x)**	**(2.62x)**	**(1.74x)**	**(4.45x)**	**(1.88x)**	**(3.26x)**	**(1.12x)**	-

TABLE II: Runtime (secs) and memory usage in MBs (Mm) evaluating subgraph queries using three different index configurations: `D`, $\texttt{D}_s$, and $\texttt{D}_p$ introduced in Section V-B. We report index reconfiguration (IR) time (secs).


$P_\alpha(e_1)$,$P_\alpha(e_2)$
$P_\alpha(e_1)$,$P_\alpha(e_2)$
$P_\alpha(e_1)$,$P_\alpha(e_2)$,$P_\alpha(e_3)$

(a) MR_1. (b) MR_2. (c) MR_3.

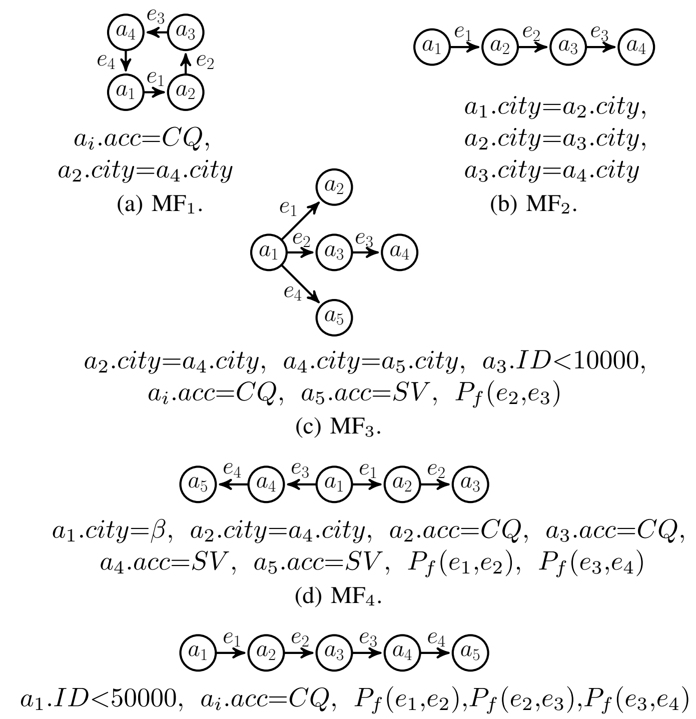
Fig. 4: MagicRec (MR) queries. $P_\alpha(e_i)$ = e_i.time < α

		MR_1	MR_2	MR_3	Mm	IC
Ork	`D`	29.37	255.4	22.65	2755	-
	$\texttt{D+VP}_t$	14.36(**2.0x**)	166.3(**1.5x**)	3.33(**6.8x**)	2982(**1.1x**)	42.10
LJ	`D`	18.19	38.17	842.8	1689	-
	$\texttt{D+VP}_t$	8.83(**2.1x**)	27.26(**1.4x**)	79.72(**10.6x**)	1820(**1.1x**)	21.79
WT	`D`	6.87	9.67	136.5	700	-
	$\texttt{D+VP}_t$	2.69(**2.6x**)	5.36(**1.8x**)	22.74(**6.0x**)	755(**1.1x**)	9.14

TABLE III: Runtime (secs) and memory usage in MBs (Mm) evaluating MagicRec queries using Configs: `D` and $\texttt{D+VP}_t$ introduced in Section V-C1. We report index creation (IC) time (secs) for secondary indexes.

2) WCOJ Plans

We next evaluate the benefit and overhead tradeoff of secondary vertex-partitioned indexes when the secondary index allows the system to generate new WCOJ plans that are not in the plan space with primary indexes only. We take a set of queries drawn from cyclic fraudulent money flows reported in prior literature [28], as well as acyclic patterns that contain the money flow paths from our running examples. Figure 5 shows our queries MF_1, ..., MF_5. We focus on MF_1 to MF_4 here and use MF_5 in the next section. These four queries have equality conditions on the `city` property of the vertices, so can benefit from multiway joins computed by intersecting lists that are presorted on `city`. We evaluate these queries on two Configs. The first Config consists of the system's primary A+ index denoted by `D` as before. The second Config, denoted by $\texttt{D+VP}_c$ adds on top of D a new secondary vertex-partitioned index $\texttt{VP}_c$ in both forward and backward directions that: (i) has the same partitioning structure as primary A+ indexes; and (ii) sorts the inner-most lists on neighbour's `city` property.


$a_i.acc$=CQ,
$a_2.city$=$a_4.city$
$a_1.city$=$a_2.city$,
$a_2.city$=$a_3.city$,
$a_3.city$=$a_4.city$
$a_2.city$=$a_4.city$, $a_4.city$=$a_5.city$, $a_3.ID$<10000,
$a_i.acc$=CQ, $a_5.acc$=SV, $P_f(e_2,e_3)$
$a_1.city$=β, $a_2.city$=$a_4.city$, $a_2.acc$=CQ, $a_3.acc$=CQ,
$a_4.acc$=SV, $a_5.acc$=SV, $P_f(e_1,e_2)$, $P_f(e_3,e_4)$
$a_1.ID$<50000, $a_i.acc$=CQ, $P_f(e_1,e_2)$,$P_f(e_2,e_3)$,$P_f(e_3,e_4)$

(a) MF_1. (b) MF_2. (c) MF_3. (d) MF_4. (e) MF_5.

Fig. 5: Fraud detection queries. $P_f(e_i,e_j)$ defined as $e_i.date<e_j.date$, $e_i.amt>e_j.amt$, $e_i.amt<e_j.amt+\alpha$.

For each dataset, we randomly added each vertex an account type property from [CQ, SV], a city from 4417 cities, and to each edge an amount in the range of [1, 1000] and a date within a 5 year range.

Table IV shows our results (ignore the MF_5 column and the $\texttt{D+VP}_c\texttt{+EP}_c$ rows for now). Similar to our previous experiment, despite indexing all of the edges (this time twice), our secondary index has only 1.17x memory overhead (the increase from 1.08x is due to double indexing), whereas we see uniform and up to 24.7x improvements in run time. We note that in all of these queries, the benefits are solely coming from using new plans that use WCOJ processing. For example in query MF_1, the $\texttt{D+VP}_c$ configuration allows the system to generate a plan

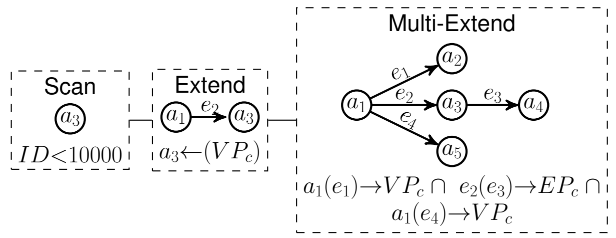

Scan
a_3
$ID<10000$
Extend
a_1 e_2 a_3
$a_3 \leftarrow (VP_c)$
Multi-Extend
a_1 e_1 a_2 e_2 a_3 e_3 a_4 e_4 a_5
$a_1(e_1) \rightarrow VP_c \cap\ e_2(e_3) \rightarrow EP_c \cap$
$a_1(e_4) \rightarrow VP_c$

Fig. 6: WCOJ Plan for MF_3 from Figure 5c using two $\mathtt{VP}_c$ indexes and one $\mathtt{EP}_c$ index from Sections V-C2 and V-D.

that: (1) reads a_1; (2) uses MULTI-EXTEND to intersect a_1's forward and backward lists in $\mathtt{VP}_c$, which matches a_2 and a_4; and (3) uses E/I that intersects a_2's forward and a_4's backward lists in the primary A+ index to match the a_3's. Such plans are not possible in absence of the $\mathtt{VP}_c$ index. Instead for MF_1, under the default configuration D, the system extends a_1 to a_2, then to a_3 separately, runs a FILTER operator to match the cities, and then uses E/I to match the a_3's.

D. Secondary Edge-Partitioned A+ Indexes

Finally, we evaluate the tradeoffs of our secondary edge-partitioned A+ indexes on our financial fraud application. We add a third Config to our experiment denoted by $\mathtt{D}$+$\mathtt{VP}_c$+$\mathtt{EP}_c$. The configuration adds the edge-partitioned index from Example 7 in Section III-B2. We change the second-level partitioning to be on v.$_{adj}$.acc instead of edge labels and add the predicate e.$_b$.amt < e.$_{nbr}$.amt + α. We pick the "intermediate cut" value α in our examples to have 5% selectivity.

Table IV shows our results. First we observe that the addition of $\mathtt{EP}_c$ only allows new plans to be generated for MF_3, MF_4 and MF_5, so we report numbers only for these queries. The improvements in run time range from 6.14x to up to 72.2x for a 2.22x memory overhead. Naturally the memory and performance tradeoff will change with the selectivity of α. What is more important to note is that the speedups are primarily due to the system producing significantly more efficient plans in the presence of the $\mathtt{EP}_c$ index. For example, the system now generates a highly complex plan for MF_3, shown in Figure 6, that uses a mix of vertex and edge-partitioned indexes and performs a 3-way intersection.

E. Neo4j and TigerGraph Comparisons

We next compare GraphflowDB to Neo4j and TigerGraph. These experiments are provided for completeness only. These are full-fledged commercial systems that support transactions. However Neo4j is perhaps the most popular existing GDBMS and TigerGraph, to the best of our knowledge, is the most performant one in terms of read performance. Our goal is to and show that the benefits of A+ indexes reported are on top of a system that is already competitive with existing GDBMSs.

We report numbers for four of our labelled subgraph queries SQ_1, SQ_2, SQ_3, and SQ_{13} on $LJ_{12,2}$ and $WT_{4,2}$ on Neo4j and TigerGraph, using their default Configs and using the $\mathtt{D}$ and $\mathtt{D}_p$ configurations from Section V-B for GraphflowDB. Table V shows our results. We found GraphflowDB to be faster on all queries on the $\mathtt{D}$ configuration except for SQ_{13} on $WT_{4,2}$. In addition, similar to our experiments from Table II, the $\mathtt{D}_p$ configuration makes GraphflowDB even more performant. TigerGraph was the fastest system on SQ_{13}, which is a long 5-edge path. We cannot inspect the source code but we suspect for paths TigerGraph extends each distinct intermediate node only once and they only report pairs of reachable nodes. However, note that using the reconfigured index D_p, GraphflowDB outperforms TigerGraph on $LJ_{12,2}$ and closes the gap on $WT_{4,2}$.

We note that system-to-system comparisons should not be interpreted as one system being superior to another. What is more important is that neither of these systems has a mechanism for tuning through index reconfiguration or construction to close their performance gaps on join-heavy queries.

F. Index Maintenance Performance

We next benchmark the maintenance speed of each type of A+ index on a micro-benchmark. We report our numbers for two datasets $LJ_{2,4}$ and $Brk_{2,2}$. We load 50% of the dataset from the MagicRec application and insert the remaining 50% of the edges one at a time and evaluate the speed of 5 Configs, each requiring progressively more maintenance work: (i) $\mathtt{D}_s$ has no partitioning and sorts by the the adjacent vertices IDs; (ii) $\mathtt{D}_p$ partitions each adjacency list on adjacent edges `label`; (iii) $\mathtt{D}_{ps}$ sorts each partition in $\mathtt{D}_p$ by the adjacent vertices IDs; (iv) $\mathtt{D}_{ps}$+$\mathtt{VP}_t$ creates a secondary adjacency list index on the time property for $\mathtt{D}_{ps}$; and finally (v) $\mathtt{D}_{ps}$+$\mathtt{EP}_t$: an edge bound adjacency list index with the same partitioning and sorting as $\mathtt{VP}_t$ for the query $v_s{-}[e_b]{\leftarrow}v_d{-}[e_{adj}]{\rightarrow}v_{adj}$ with predicate e_b.time ¡ e_{adj}.time + α that has a 1% selectivity.

We report our numbers for two datasets $LJ_{2,4}$ and $Brk_{2,2}$ using a single thread. We were able to maintain the following update rates per second (reported respectively for $LJ_{2,4}$ and $Brk_{2,2}$): 1.203M and 2.108M for $\mathtt{D}_s$, 1.024M and 1.892M for $\mathtt{D}_p$, 1.081M and 1.832M for $\mathtt{D}_{ps}$, 706K and 1.691M for $\mathtt{D}_{ps}$+$\mathtt{VP}_t$, and 41K and 110K for $\mathtt{D}_{ps}$+$\mathtt{EP}_t$. Our update rate gets slower with additional complexity but we are able to maintain insert rates of between 50-100k edges/s for our edge-partitioned index and between 706K-2.1M for our vertex-partitioned indexes. Note that our implementation is not write optimized and these speeds, though we believe is sufficient for modern applications, can be further improved.

VI. RELATED WORK

View-based Query Processing: Answering queries using views has been well studied in the context of relational, XML, or RDF data management. We refer the reader to several surveys and references on the topic [14], [29], [30]. This extensive literature studies numerous topics, such as rewriting queries using a set of views [31], selecting a set of views for a workload e.g., web databases [32], or the computational complexities of deciding whether a query can be answered with a given set of views [33]. In this work, we observed that the lists that are stored in the adjacency list indexes can be seen as views and systems provide fast access to these lists/views through CSR-like data structures. In contrast to prior work, we explored how to extend the views that can be accessed through adjacency list indexes in a space-efficient manner. Specifically, we identified a restricted but still much larger set of views than existing indexes, that can be stored by either merely tuning

		MF_1	MF_2	MF_3	MF_4	MF_5	Mem(MB)	$\|E_{indexed}\|$	IC
Ork	`D`	73.35	5.53	32.85	71.46	890.8	2730	117.1M	-
	`D+VP`$_c$	8.99 **(8.16x)**	2.75 **(2.01x)**	1.33 **(24.7x)**	19.03 **(3.76x)**	—	3183 **(1.17x)**	117.1M	85.83
	`D+VP`$_c$`+EP`$_c$	—	—	0.56 **(58.7x)**	0.99 **(72.2x)**	60.59 **(14.7x)**	6000 **(2.20x)**	513.2M	288.4
LJ	`D`	47.09	4.24	84.78	7.60	52.04	1649	68.5M	-
	`D+VP`$_c$	11.45 **(4.11x)**	2.86 **(1.48x)**	5.12 **(16.6x)**	3.66 **(2.08x)**	—	1910 **(1.16x)**	68.5M	46.43
	`D+VP`$_c$`+EP`$_c$	—	—	2.16 **(39.3x)**	0.39 **(19.5x)**	5.79 **(8.99x)**	3585 **(2.17x)**	276.2M	279.8
WT	`D`	20.27	1.47	9.02	0.86	9.02	685	28.5M	-
	`D+VP`$_c$	2.29 **(8.85x)**	1.12 **(1.31x)**	1.55 **(5.82x)**	0.53 **(1.62x)**	—	796 **(1.16x)**	28.5M	21.26
	`D+VP`$_c$`+EP`$_c$	—	—	0.50 **(18.0x)**	0.14 **(6.14x)**	0.79 **(11.4x)**	1521 **(2.22x)**	125.4M	843.5

TABLE IV: Runtime (secs) of GraphflowDB plans and memory usage (Mem) in MB evaluating fraud detection queries using different Configs: `D`, `D+VP`$_c$, and `D+VP`$_c$`+EP`$_c$ introduced in Section V-C2. The run time speedups and memory usage increase shown in parenthesis are in comparison to `D`. We report index creation time (IC) in secs for secondary indexes.

	$LJ_{12,2}$				$WT_{4,2}$			
	SQ_1	SQ_2	SQ_3	SQ_{13}	SQ_1	SQ_2	SQ_3	SQ_{13}
`D`	**0.4**	1.4	1.1	31.3	0.6	4.6	5.5	767.5
`D`$_p$	**0.4**	**0.7**	**0.6**	**6.0**	**0.3**	**2.1**	**3.1**	235.7
`TG`	2.5	11.8	15.2	30.5	1.6	7.1	10.2	**29.5**
`N4`	29.3	35.3	36.8	TL	1.65k	876	82.9	TL

TABLE V: Runtime (secs) of GraphflowDB on Configs D and D_p introduced in Section V-B, runtime of TigerGraph (TG), and runtime of Neo4j (N4). TL indicates >30 mins.

the partitioning schemes of a multi-level CSR data structure or lightweight offset lists.

Kaskade [34] (KSK) is a graph query optimization framework that uses *materialized graph views* to speed up query evaluation. Specifically, KSK takes as input a query workload Q and an input graph G. Then, KSK enumerates possible *views* for Q, which are other graphs G' that contain a subset of the vertices in G and other edges that can represent multi-hop connections in G. For example, if G is a data provenance graph with job and file vertices, and there are "consumes" and "produces" relationships between jobs and files, an example graph view G' could store the job vertices and their 2-hop dependencies through files. KSK materializes its selected views in Neo4j, and then translates queries over G to appropriate graphs (views) that are stored in Neo4j, which is used to answer queries. Therefore, the framework is limited by Neo4j's adjacency lists.

There are several differences between the views provided by KSK and A+ indexes. First, KSK's views are based on "constraints" that are mined from G's schema based only on vertex/edge labels and not properties. For example, KSK can mine "job vertices connect to jobs in 2-hops but not to file vertices" constraints but not "accounts connect to accounts in 2-hops with later dates and lower amounts", which can be a predicate in an A+ index. Second, because KSK stores its views in Neo4j, KSK views are only vertex ID and edge label partitioned, unlike our views which are stored in a CSR data structure that support tunable partitioning, including by edge IDs, as well as sorting. Similarly, because KSK uses Neo4j's query processor, its plans do not use WCOJs.

Adjacency List Indexes in Graph Analytics Systems: There are numerous graph analytics systems [35], [36], [37] designed to do batch analytics, such as decomposing a graph into connected components. These systems use native graph storage formats, such as adjacency lists or sparse matrices. Work in this space generally focuses on optimizing the physical layout of the edges in memory. For systems storing the edges in adjacency list structures, a common technique is to store them in CSR format [8]. To implement A+ indexes we used a variant of CSR that can have multiple partitioning levels. Reference [37] studies CSR-like partitioning techniques for large lists and reference [38] proposes segmenting a graph stored in a CSR-like format for better cache locality. This line of work is complementary to ours.

Indexes in RDF Systems: RDF systems support the RDF data model, in which data is represented as a set of (subject, predicate, object) triples. Prior work has introduced different architectures, such as storing and then indexing one large triple table [39], [40] or adopting a native-graph storage [41]. These systems have different designs to further index these tables or their adjacency lists. For example, RDF-3X [39] indexes an RDF dataset in multiple B+ tree indexes. As another example, the gStore system encodes several vertices in fixed length bit strings that captures information about the neighborhoods of vertices. Similar to the GDBMSs we reviewed, these work also define fixed indexes for RDF triples. A+ indexes instead gives users a tunable mechanism to tailor a GDBMS to the requirements of their workloads.

Indexes for XML Data: There is prior work focusing on indexes for XML and the precursor tree or rooted graph data models. Many of this work provides complete indexes, such as DataGuides [42] or IndexFabric [43], or approximate indexes [44], [45] that index the paths from the roots of a graph to individual nodes in the data. These indexes are effectively summaries of the graph that are used during query evaluation to prune the search of path expressions in the data. These indexes are not directly suitable for contemporary GDBMS which store non-rooted property graphs, where the paths that users search in queries can start from arbitrary nodes.

Other complex subgraph indexes: Many prior algorithmic work on evaluating subgraph queries [46], [47], [48] have also proposed auxiliary indexes that index subgraphs more complex than edges, such as paths, stars, or cliques. This line of work effectively demonstrates that indexing such subgraphs can speed up subgraph query evaluation. Unlike our work, these subgraphs can be more complex but their storage is not optimized for space efficiency.

VII. Conclusions

Ted Codd, the inventor of the relational model, criticized the GDBMSs of the time as being restrictive because they only performed a set of "predefined joins" [49], which causes physical data dependence and contrasts with relational systems that can join arbitrary tables. This is indeed still true to a good extent for contemporary GDBMSs, which are designed to join vertices with only their neighbourhoods, which are predefined to the system as edges. However, this is specifically the major appeal of GDBMSs, which are highly optimized to perform these joins efficiently by using adjacency list indexes. Our work was motivated by the shortcoming that existing GDBMSs have fixed adjacency list indexes that limit the workloads that can benefit from their fast join capabilities. As a solution, we described the design and implementation of a new indexing subsystem with restricted materialized view support that can be stored using a space-efficient technique. We demonstrated the flexibility of A+ indexes, and evaluated the performance and memory tradeoffs they offer on a variety of applications drawn from popular real-world applications.

References

[1] "Neo4j," https://neo4j.com.
[2] "JanusGraph," https://janusgraph.org.
[3] "TigerGraph," https://www.tigergraph.com.
[4] A. Mhedhbi and S. Salihoglu, "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins," *PVLDB*, 2019.
[5] C. Kankanamge *et al.*, "Graphflow: An Active Graph Database," *SIGMOD*, 2017.
[6] "Neo4j Property Graph Model," https://neo4j.com/developer/graph-database, 2019.
[7] S. Sahu *et al.*, "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey," *VLDBJ*, 2019.
[8] A. Bonifati *et al.*, *Querying Graphs*. Morgan & Claypool, 2018.
[9] H. Q. Ngo *et al.*, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Rec.*, 2014.
[10] A. Mhedhbi *et al.*, "A+ indexes: Tunable and space-efficient adjacency lists in graph database management systems," *CoRR*, 2021.
[11] "openCypher," https://www.opencypher.org.
[12] C. R. Aberger *et al.*, "EmptyHeaded: A Relational Engine for Graph Processing," *TODS*, 2017.
[13] J. Zhou, P. Larson, and J. Goldstein, "Partially materialized views," Tech. Rep., 2005.
[14] A. Y. Halevy, "Answering queries using views: A survey," *VLDBJ*, 2001.
[15] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *ICDE*, 1995.
[16] D. Kossmann and K. Stocker, "Iterative dynamic programming: A new class of query optimization algorithms," *TODS*, vol. 25, no. 1, 2000.
[17] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the graphblas api for c," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.
[18] J. Goldstein and P.-r. Larson, "Optimizing queries using materialized views: A practical, scalable solution," in *SIGMOD*, 2001.
[19] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc., 1989.
[20] K. Ammar *et al.*, "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows," *PVLDB*, 2018.
[21] S. Chaudhuri and V. Narasayya, "Autoadmin what-if index analysis utility," *SIGMOD Rec.*, 1998.
[22] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *VLDB*, 1997.
[23] B. Ding *et al.*, "Ai meets ai: Leveraging query executions to improve index recommendations," *SIGMOD*, 2019.
[24] S. Agrawal *et al.*, "Automated selection of materialized views and indexes in sql databases," *PVLDB*, 2000.
[25] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *TKDE*, 2005.
[26] P. Gupta *et al.*, "Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs," *PVLDB*, 2014.
[27] F. Bi *et al.*, "Efficient Subgraph Matching by Postponing Cartesian Products," *SIGMOD*, 2016.
[28] X. Qiu *et al.*, "Real-Time Constrained Cycle Detection in Large Dynamic Graphs," *PVLDB*, 2018.
[29] S. Abiteboul, "On views and xml," *PODS*, 1999.
[30] R. Castillo *et al.*, "Selecting materialized views for rdf data," 2010.
[31] F. Goasdoué *et al.*, "Materialized View-Based Processing of RDF Queries," *Bases de Données Avancées*, 2010.
[32] F. Goasdoué *et al.*, "View selection in semantic web databases," *PVLDB*, 2011.
[33] W. Fan *et al.*, "Answering pattern queries using views," *TKDE*, 2016.
[34] J. M. F. da Trindade *et al.*, "Kaskade: Graph views for efficient graph analytics," *ICDE*, 2020.
[35] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *IJHPCA*, 2011.
[36] G. Malewicz *et al.*, "Pregel: A System for Large-Scale Graph Processing," *SIGMOD*, 2010.
[37] J. Shun *et al.*, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," *ACM SIGPLAN Notices*, 2013.
[38] Y. Zhang *et al.*, "Making Caches Work for Graph Analytics," *IEEE Big Data*, 2017.
[39] T. Neumann *et al.*, "RDF-3X: A RISC-style Engine for RDF," *PVLDB*, 2008.
[40] C. Weiss *et al.*, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *PVLDB*, 2008.
[41] L. Zou *et al.*, "gStore: A Graph-Based SPARQL Query Engine," *VLDBJ*, 2014.
[42] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," *PVLDB*, 1997.
[43] B. Cooper *et al.*, "A fast index for semistructured data," *PVLDB*, 2001.
[44] T. Milo *et al.*, "Index structures for path expressions," *ICDT*, 1999.
[45] R. Kaushik *et al.*, "Exploiting local similarity for indexing paths in graph-structured data," *ICDE*, 2002.
[46] J. M. Sumrall *et al.*, "Investigations on path indexing for graph databases," *Euro-Par*, 2016.
[47] J. Cheng *et al.*, "Fast Graph Pattern Matching," *ICDE*, 2008.
[48] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *PVLDB*, 2016.
[49] "Edgar f. ("ted") codd turing award lecture," https://amturing.acm.org/award_winners/codd_1000892.cfm.

Columnar Storage and List-based Processing for Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta,amine.mhedhbi,semih.salihoglu}@uwaterloo.ca

ABSTRACT

We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on their access patterns. We then present the design of columnar storage, compression, and query processing techniques based on these desiderata. In addition to showing direct integration of existing techniques from columnar RDBMSs, we also propose novel ones that are optimized for GDBMSs. These include a novel list-based query processor, which avoids expensive data copies of traditional block-based processors under many-to-many joins, a new data structure we call single-indexed edge property pages and an accompanying edge ID scheme, and a new application of Jacobson's bit vector index for compressing NULL values and empty lists. We integrated our techniques into the GraphflowDB in-memory GDBMS. Through extensive experiments, we demonstrate the scalability and query performance benefits of our techniques.

PVLDB Reference Format:
Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu. Columnar Storage and List-based Processing for Graph Database Management Systems. PVLDB, 14(11): 2491-2504, 2021.
doi:10.14778/3476249.3476297

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/graphflow/graphflow-columnar-techniques.

1 INTRODUCTION

Contemporary GDBMSs are data management software such as Neo4j [47], Neptune [5], TigerGraph [59], and GraphflowDB [32, 41] that adopt the property graph data model [48]. In this model, application data is represented as a set of vertices and edges, which represent the entities and their relationships, and key-value properties on the vertices and edges. GDBMSs support a wide range of analytical applications, such as fraud detection and recommendations in financial, e-commerce, or social networks [56] that search for patterns in a graph-structured database, which require reading large amounts of data. In the context of RDBMSs, column-oriented systems [29, 53, 57, 66] employ a set of read-optimized storage, indexing, and query processing techniques to support traditional analytical applications, such as business intelligence and reporting, that also process large amounts of data. As such, these techniques are relevant for improving the performance and scalability of GDBMSs.

In this paper, we revisit columnar storage and query processing techniques in the context of GDBMSs. Specifically, we focus on an in-memory GDBMS setting and discuss the applicability of columnar storage and compression techniques for storing different components of graphs [1, 2, 57, 68], and block-based query processing [3, 11]. Despite their similarities, workloads in GDBMSs and columnar RDBMSs also have fundamentally different access patterns. For example, workloads in GDBMSs contain large many-to-many joins, which are not frequent in column-oriented RDBMSs. This calls for redesigning columnar techniques in the context of GDBMSs. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for designing the physical data layout and query processor of a GDBMS.

Columnar Storage: Section 4 explores the application of columnar data structures for storing different data components in GDBMSs. While existing columnar structures can directly be used for storing vertex properties and many-to-many (n-n) edges, we observe that using straightforward edge columns, to store properties of n-n edges does not guarantee sequential access when reading edge properties in either forward or backward directions. An alternative, which we call *double-indexed property CSRs*, can achieve sequential access in both directions but requires duplicating edge properties. We then describe an alternative design point, *single-directional property pages*, that avoids duplication and achieves good locality when reading properties of edges in one direction and still guarantees random access in the other. This requires using a new edge ID scheme that is conducive to extensive compression when storing them in adjacency lists without any decompression overheads. Lastly, as a new application of vertex columns, we show that single cardinality edges and edge properties, i.e. those with one-to-one (1-1), one-to-many (1-n) or many-to-one (n-1) cardinalities, are stored more efficiently with vertex columns instead of the structures we describe for n-n edges.

Columnar Compression: In Section 5, we review existing columnar compression techniques, such as dictionary encoding, that satisfy our desiderata and can be directly applied to GDBMSs. We

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476297

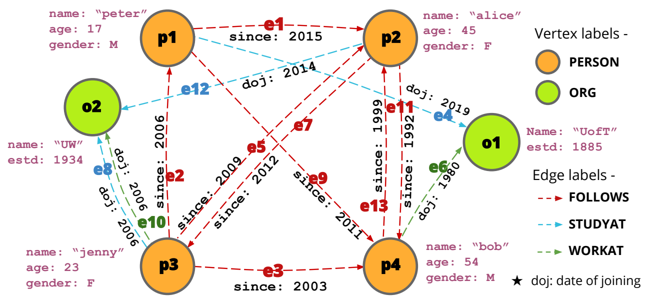


Figure 1: Running example graph.

next show that existing techniques for compressing NULL values in columns from references [1, 2] by Abadi et al. lead to very slow accesses to arbitrary non-NULL values. We then review Jacobson's bit vector index [30, 31] to support constant time rank queries, which has found several prior applications e.g., in a range filter structure in databases [64], in information retrieval [21, 44] and computational geometry [14, 45]. We show how to enhance one of Abadi's schemes with an adaptation of Jacobson's index to provide constant-time access to arbitrary non-NULL values, with a small increase in storage overhead compared to prior techniques.

List-based Processing: In Section 6, we observe that traditional block-based processors or columnar RDBMSs [3, 67] process fixed-length blocks of data in tight loops, which achieves good CPU and cache utility but results in expensive data copies under n-n joins. To address this, we propose a new block-based processor we call *list-based processor (LBP)*, which modifies traditional block-based processors in two ways to tailor them for GDBMSs: (i) Instead of representing the intermediate tuples processed by operators as a single group of equal-sized blocks, we represent them as multiple factorized groups of blocks. We call these *list groups*. LBP avoids expensive data copies by flattening blocks of some groups into single values when performing n-n joins. (ii) Instead of fixed-length blocks, LBP uses variable length blocks that take the lengths of adjacency lists that are represented in the intermediate tuples. Because adjacency lists are already stored in memory consecutively, this allows us to avoid materializing adjacency lists during join processing, improving query performance.

We integrated our techniques into GraphflowDB [32]. We present extensive experiments that demonstrate the scalability and performance benefits (and tradeoffs) of our techniques both on microbenchmarks and on the LDBC and JOB benchmarks against a row-based Volcano-style implementation of the system, an open-source version of a commercial GDBMSs, and two column-oriented RDBMSs. Our code, queries, and data are available here [25].

2 BACKGROUND

In the property graph model, vertices and edges have labels and arbitrary key value properties. Figure 1 shows a property graph that will serve as our running example, which contains vertices with PERSON and ORGANIZATION (ORG) labels, and edges with FOLLOWS, STUDYAT and WORKAT labels.

There are three storage components of GDBMSs: (i) topology, i.e., adjacencies of vertices; (ii) vertex properties; and (iii) edge properties. In every native GDBMS we are aware of, the topology

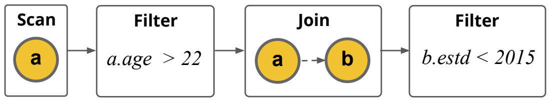


Figure 2: Query plan for the query in Example 1.

is stored in data structures that organize data in *adjacency lists* [12], such as in compressed sparse row (CSR) format. Typically, given the ID of a vertex v, the system can in constant-time access v's adjacency list, which contains a list of (edge ID, neighbour ID) pairs. Typically, the adjacency list of v is further clustered by edge label which enables efficient traversal of the neighbourhood of v, given a particular label. Vertex and edge properties can be stored in a number of ways. For example, some systems use a key-value store, such as DGraph [15] and JanusGraph [6], and some use a variant of *interpreted attribute layout* [9], where records consist of variable-sized key-value properties. Records can be located consecutively in disk or memory or have pointers to each other, as in Neo4j.

Queries in GDBMSs consist of a subgraph pattern Q that describes the joins in the query (similar to SQL's FROM) and optionally predicates on these patterns with final group-by-and-aggregation operations. We assume a GDBMS with a query processor that uses variants of the following relational operators, which is the case in many GDBMSs, e.g., Neo4j [47], Memgraph [40], or GraphflowDB:
Scan: Scans a set of vertices from the graph.
Join (e.g. Expand in Neo4j and Memgraph, Extend in GraphflowDB): Performs an index nested loop join using the adjacency list index to match an edge of Q. Takes as input a partial match t that has matched k of the query edges in Q. For each t, Join extends t by matching an unmatched query edge $qv_s \rightarrow qv_d$, where qv_s or qv_d has already been matched. For example if qv_s has already been matched to data vertex v_i, then the operator produces one $(k+1)$-match for each edge-neighbour pair in v_i's forward adjacency list[1].
Filter: Applies a predicate ρ to a partial match t, reading any necessary vertex and edge properties from storage.
Group By And Aggregate: Performs a standard group by and aggregation computation on a partial match t.

EXAMPLE 1. *Below is an example query written in the Cypher language [19]:*

```
MATCH (a:PERSON)−[e:WORKAT]→(b:ORG)
WHERE a.age > 22 AND b.estd < 2015 RETURN *
```

The query returns all persons a and their workplaces b, where a is older than 22 and b was established before 2015. Figure 2 shows a typical plan for this query.

3 GUIDELINES AND DESIDERATA

We next outline a set of guidelines and desiderata for organizing the physical data layout and query processor of GDBMSs. We assume edges are doubly-indexed in forward and backward adjacency lists, as in every GDBMS we are aware of. We will not optimize this duplication as this is needed for fast joins from both ends of edges.

GUIDELINE 1. *Edge and vertex properties are read in the same order as the edges appear in adjacency lists after joins.*

[1]GraphflowDB can perform an intersection of multiple adjacency lists if the pattern is cyclic (see reference [41]).

Observe that JOIN accesses the edges and neighbours of a vertex v_i in the order these edges appear in v_i's adjacency list $L_{v_i} = \{(e_{i1}, v_{i1})..., (e_{i\ell}, v_{i\ell})\}$. If the next operator also needs to access the properties of these edges or vertices, e.g., Filter in Figure 2, these accesses will be in the same order. Our first desiradata is to store the properties of e_{i1} to $e_{i\ell}$ sequentially in the same order. Ideally, a system should also store the properties v_{ij} sequentially in the same order but in general this would require prohibitive data replication because while each e_{ij} appears in two adjacency lists, each v_{ij} appears in as many lists as the degree of v_{ij}.

Desideratum 1. *Store and access the properties of edges sequentially in the order edges appear in adjacency lists.*

Guideline 2. *Access to vertex properties will not be to sequential locations and many adjacency lists are very small.*

Guideline 1 implies that we should expect random accesses in memory when an operators access vertex properties. In addition, real-world graph data with n-n relationships have power-law degree distributions [37]. So, there are often many short adjacency lists in the dataset. For example, the FLICKR, WIKI graphs that we use, have single edge labels with average degrees of only 14 and 41, and the Twitter dataset used in many prior work on GDBMSs [33] has a degree of 35. Therefore when processing queries with two or more joins, reading different adjacency lists will require iteratively reading a short list followed by a random access. This implies that techniques that require decompressing blocks of data, say a few KBs, to only read a single vertex property or a single short adjacency list can be prohibitively expensive.

Desideratum 2. *If compression is used, decompressing arbitrary data elements in a compressed block should happen in constant time.*

Guideline 3. *Graph data often has partial structure.*

Although the property graph model is semi-structured, data in GDBMSs often have some structure. One reason for this is because the data in GDBMSs sometimes comes from structured data from RDBMSs as observed in a recent user survey [56]. In fact, several vendors and academics are actively working on defining a schema language for property graphs [13, 27]. Common structure are:

(i) *Edge label determines source and destination vertex labels.* For example, in the popular LDBC social network benchmark (SNB), KNOWS edges exist only between vertices of label PERSON.

(ii) *Label determines vertex and edge properties.* Similar to attributes of a relational table, properties on an edge or vertex and their datatypes can often be determined by the label. For example, this is the case for every vertex and edge label in LDBC.

(iii) *Edges with single cardinality.* Edges might have cardinality constraints: 1-n (single cardinality in the backward edges), n-1 (single cardinality in the forward edges), 1-1, and n-n. An example of 1-n cardinality from LDBC SNB is that each organization has one isLocatedIn edge.

We refer to edges that satisfy properties (i) and (ii) as *structured edges* and properties that satisfy property (ii) as *structured vertex/edge property*. Other edges and properties will be called *unstructured*. The existence of such structure in some graph data motivates our third desideratum:

Desideratum 3. *Exploit structure in the data for space-efficient storage and faster access to data.*

Table 1: Columnar data structures and data components they are used for. V-Column stands for vertex column.

Data	Columnar data structure
Vertex Properties	V-Column
Edge Properties	V-Column: of src when n-1, of dst when 1-n, of either src or dst when 1-1
	Single-indexed prop. pages when n-n
Fwd Adj. lists	V-Column when 1-1 and n-1, CSR o.w.
Bwd Adj. lists	V-Column when 1-1 and 1-n, CSR o.w.

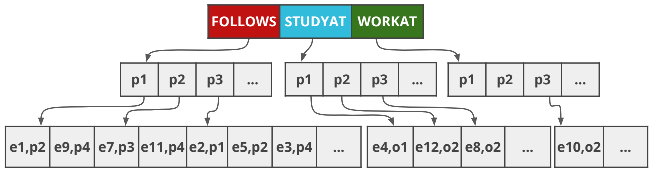


Figure 3: Example forward adjacency lists implemented as a 2-level CSR structure for the example graph.

4 COLUMNAR STORAGE

We next explore using columnar structures for storing data in GDBMSs to meet the desiderata from Section 3. For reference, Table 1 presents the summary of the columnar structures we use and the data they store. We start with directly applicable structures and then describe our new single-indexed property pages structure and its accompanying edge ID scheme to store edge properties.

4.1 Directly Applicable Structures

4.1.1 CSR for n-n Edges. CSR is an existing columnar structure that is widely used by existing GDBMSs to store edges. A CSR, shown in Figure 3, effectively stores a set of (vertex ID, edge ID, neighbour ID) triples sorted by vertex ID, where the vertex IDs are compressed similar to run-length encoding. In this work, we store the edges of each edge label with n-n cardinality in a separate CSR. As we discuss next, we can store the edges with other cardinalities more efficiently than a CSR by using vertex columns.

4.1.2 Vertex Columns for Vertex Properties, Single Cardinality Edges and Edge Properties. With an appropriate vertex ID scheme, columns can be directly used for storing structured vertex properties in a compact manner. Let $p_{i,1}, p_{i,2}, ...p_{i,n}$ be the structured vertex properties of vertices with label lv_i. We have a *vertex column* for each $p_{i,j}$, that stores $p_{i,j}$ properties of vertices in consecutive locations. Then we can adopt a (vertex label, label-level positional offset) ID scheme and ensure that offsets with the same label are consecutive. As we discuss in Section 5.2, this ID scheme also can be compressed by factoring out vertex labels.

Similarly, we can store single cardinality edges, i.e., those with 1-1, 1-n, or n-1 constraints, and their properties directly as a *property* of source or destination vertex of the edges in a vertex column and directly access them using a vertex positional offset. As we momentarily discuss, this is more efficient both in terms of storage and access time than the structures we cover for storing properties of n-n edges (Desideratum 3). Figure 4 shows single cardinality

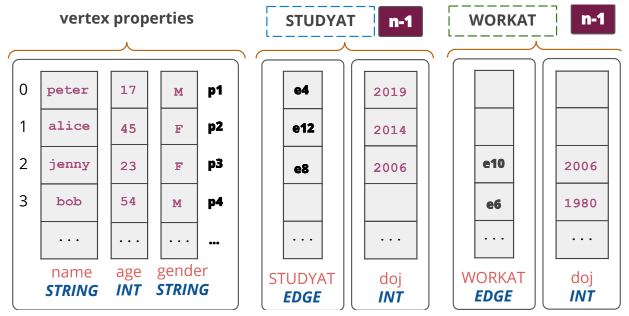


Figure 4: Example vertex columns storing vertex properties and single-cardinality edges and their properties.

STUDYAT and WORKAT edges from our example and their properties stored as vertex column of PERSON vertices.

4.2 Single-indexed Edge Property Pages for Properties of n-n Edges

Recall Desideratum 1 that access to edge properties should be in the same order of the edges in adjacency lists. We first review two columnar structures, *edge columns* and *double-indexed property CSRs*, the former of which has low storage cost but does not satisfy Desideratum 1 and the latter has high storage cost but satisfies Desideratum 1. We then describe a new design, which we call *single-indexed property pages*, which has low storage cost as edge columns and with a new edge ID scheme can partially satisfy Desideratum 1, so dominates edge columns in this design space.

Edge Columns: We can use a separate *edge column* for each property $q_{i,j}$ of edge label le_i. Then with an appropriate edge ID scheme, such as (edge label, label-level positional offset), one can perform a random access to read the $q_{i,j}$ property of an edge e. This design has low storage cost and stores each property once but does not store the properties according to any order. In practice, the order would be determined by the sequence of edge insertions and deletions.

Double-Indexed Property CSRs. An alternative is to mimic the storage of adjacency lists in the CSRs in separate CSRs that store edge properties. For each vertex v we can store $q_{i,j}$ twice in *forward* and *backward property lists*. This design provides sequential read of properties in both directions, thereby satisfying Desideratum 1, but also requires double the storage of edge columns. This can often be prohibitive especially for in-memory systems, as many graphs have orders of magnitude more edges than vertices.

A natural question is: *Can we avoid duplicate storage of double-indexed property CSRs but still achieve sequential reads?* We next show a structure that with an appropriate edge ID scheme obtains sequential reads in one direction, so partially satisfying Desideratum 1. This structure therefore dominates edge columns in design.

Single-indexed property pages: A first natural design uses only one property CSR, say forward. We call this structure *single-indexed property CSR*. Then, properties can be read sequentially in the forward direction. However, reading a property in the other direction quickly, specifically with constant time access, requires a new edge ID scheme. To see this suppose a system has read the backward adjacency lists of a vertex v with label le_i, $\{(e_1, nbr_1), ..., (e_k, nbr_k)\}$, and needs to read the $q_{i,j}$ property of these edges. Then given say

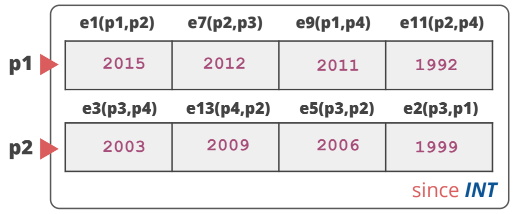


Figure 5: Single-indexed property pages for since property of FOLLOWS edges in the example graph. $k = 2$.

e_1, we need to be able to read e_1's $q_{i,j}$ property from the forward property list P_{nbr_1} of nbr_1. With a standard edge ID scheme, for example one that assigns consecutive IDs to all edges with label le_i, the system would need to first find the offset o of e_1 in forward adjacency list of nbr_1, L_{nbr_1}, which may require scanning the entire L_{nbr_1}, which is not constant time.

Instead, we can adopt a new edge ID scheme that stores the following: (edge label, source vertex ID, list-level positional offset)[2]. With this scheme a system can: (i) identify each edge, e.g., perform equality checks between two edges; and (ii) read the offset o directly from edge IDs, so reading edge properties in the opposite direction (backward in our example) can now be constant time. In addition, this scheme can be more space-efficient than schemes that assign consecutive IDs to all edges as its first two components can often be compressed (see Section 5.2). However, single-indexed property CSR and this edge ID scheme has two limitations. First access to properties in the 'opposite direction' requires two random accesses, e.g., first access obtains the P_{nbr_1} list using nbr_1's ID and the second access reads a $q_{i,j}$ property from P_{nbr_1}. Second, although we do not focus on updates in this paper, using edge IDs that contain positional offsets has an important consequence for GDBMSs. Observe that positional offsets that are used by GDBMSs are explicitly stored in data structures. Therefore, when deletions happen, we need to leave gaps in adjacency lists and recycle them when insertions happen. This may leave many gaps in adjacency lists because to recycle a list-level offset, the system needs to wait for another insertion into the same adjacency list, which may be infrequent.

Our *single-indexed property pages* addresses these two issues (Figure 5). We store k property lists (by default 128) in a property page. In a property page, properties of the same list does not have to be consecutively. However, because we use a small value of k, these properties are stored in close-by memory locations. We modify the edge ID scheme above to use page-level positional offsets. This has two advantages. First, given a positional offset, the system can directly read an edge property (so we avoid the access to read P_{nbr_1}). Second, the system can recycle a page-level offset whenever any one of the k lists get a new insertion. For reference, Figure 5 shows the single-indexed property pages in the forward direction for since property of edges with label FOLLOWS when k=2.

5 COLUMNAR COMPRESSION

Compression and query processing on compressed data are widely used in columnar RDBMSs. We start by reviewing techniques that apply directly to GDBMSs and are not novel. We then discuss the

[2] If we use the backward property CSR, the second component would instead be the destination vertex ID.

cases when we can compress the new vertex and edge ID schemes from Section 4. Finally, we review existing NULL compression schemes from columnar RDBMSs [1, 2] and enhance one of them with Jacobson's bit vector index to make it suitable for GDBMSs.

5.1 Directly Applicable Techniques

Recall our Desideratum 2 that because access to vertex properties cannot be localized and because many adjacency lists are very short, the compression schemes that are suitable for in-memory GDBMSs need to either avoid decompression completely or support decompressing arbitrary elements in a block in constant time. This is only possible if the elements are encoded in *fixed-length codes* instead of variable-length codes. We review dictionary encoding and leading 0 suppression, which we integrated in our implementation and refer readers to references [2, 20, 36] for details of other fixed-length schemes, such as frame of reference.

Dictionary encoding: This is perhaps the most common encoding scheme to be used in RDBMSs [2, 63, 68].This scheme maps a domain of values into compact codes using a variety of schemes [2, 23, 68], some producing variable-length codes, such as Huffmann encoding, and others fixed-length codes [2]. We use dictionary encoding to map a categorical edge or vertex property p, e.g., gender property of PERSON vertices in LDBC SNB dataset, that takes on z different values to $\lceil log_2(z)/8 \rceil$ bytes (we pad $log_2(z)$ bits with 0s to have a fixed number of bytes).

Leading 0 Suppression: This scheme omits storing leading zero bits in each value of a given block of data [9]. We adopt a fixed-length variant of this for storing components of edge and vertex IDs, e.g., if the maximum size of a property page is t, we use $\lceil log_2(t)/8 \rceil$ many bytes for the page-level positional offset of edge IDs.

5.2 Factoring Out Edge/Vertex ID Components

Our vertex and edge ID schemes from Sections 4 decompose the IDs into many small components, which can be factored out when the data depicts some structure (Desideratum 3). This allows compression without the need to decompress while scanning. Recall that the ID of an edge e is a triple (edge label, source/destination vertex ID, page-level positional offset) and the ID of a vertex v is a pair (vertex label, label-level positional offset). Recall also that GDBMSs store (edge ID, neighbour ID) pairs in adjacency lists. First, the vertex IDs inside the edge ID can be omitted because this is the neighbour vertex ID, which is already stored in the pairs. Second edge labels can be omitted because we cluster our adjacency lists by edge label. The only components that need to be stored are: (i) positional offset of the edge ID; and (ii) vertex label and positional offset of neighbour vertex ID. When the data depicts some structure, we can further factor out some of these components as follows:

- *Edges do not have properties:* Often, edges of a particular label do not have any properties and only represent the relationships between vertices. For example, 10 out of 15 edge labels in LDBC SNB do not have any properties. In this case, edges need not be identifiable, as the system will not access their properties. We can therefore distinguish two edges by their neighbour ID and edges with the same IDs are simply replicas of each other. Hence, we can completely omit storing the positional offsets of edge IDs.
- *Edge label determines neighbour vertex label.* Often, edge labels in the graph are between a single source and destination vertex label, e.g., Knows edges in social networks are between Person nodes. In this case, we can omit storing the vertex label of the neighbour ID.
- *Single cardinality edges:* Recall from Section 4.1.2 that the properties for single cardinality edges can be stored in vertex columns. So we can directly read these properties by using the source or destination vertex ID. So, the page-level positional offsets of these edges can be omitted.

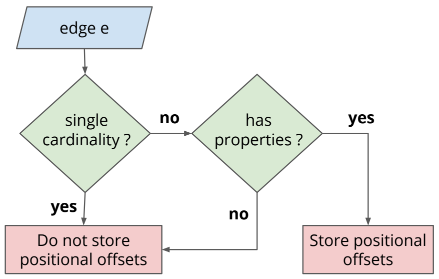


Figure 6: Decision tree for storing page-level positional offsets of edges in adjacency lists.

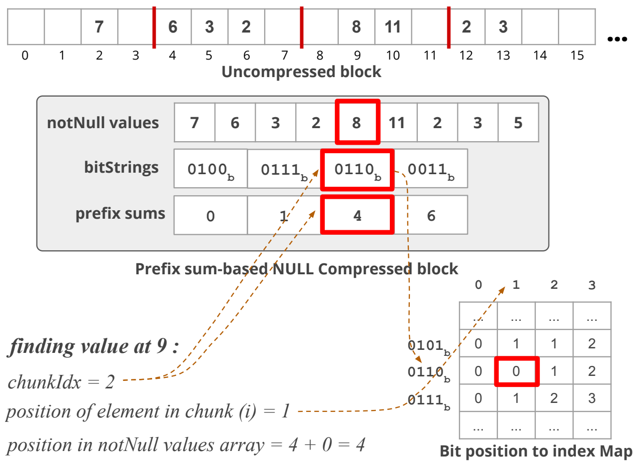
Figures 6 shows our decision tree to decide when to omit storing the page-level positional offsets in edge IDs.

5.3 NULL and Empty List Compression

Edge and vertex properties can often be quite sparse in real-world graph data. Similarly, many vertices can have empty adjacency lists in CSRs. Both can be seen as different columnar structures containing NULL values. Abadi in reference [1] describes a design space of optimized techniques for compressing NULLs in columns. All of these techniques list non-NULL elements consecutively in a 'non-NULL values column' and use a secondary structure to indicate the positions of these non-NULL values. The first technique in Abadi's paper, lists positions of each non-NULL value consecutively, which is suitable for very sparse columns, e.g., with > 90% NULLs. Second, for dense columns, lists non-NULL values as a sequence of pairs, each indicating a range of positions with non-NULL values. Third, for columns with intermediate sparsity, uses a bit vector to indicate if each location is NULL or not. The last technique is quite compact and requires only 1 extra bit per each element in a column.

However, none of these techniques are directly applicable to GDBMSs as they do not allow constant-time access to non-NULL values (Desideratum 2). To support constant-time access to a non-NULL value at position p, the secondary structure needs to support two operations in constant time: (i) check if p is NULL or not; and (ii) if it is non-NULL, compute the *rank* of p, i.e., the number of non-NULL values before p.

Abadi's third design, that uses a bit vector, already supports checking if the value at p is NULL. To support rank queries, we enhance this design with a simplified version of Jacobson's bit vector index [30, 31]. Figure 7 shows this design. In addition to the array of non-NULL values and the bit-string, we store prefix sums for each c (16 by default) elements in a block of a column, i.e., we divide the block into chunks of size c. The prefix sum holds the number of non-NULL elements before the current chunk. We also maintain a pre-populated static 2D *bit-string-position-count*



Uncompressed block
notNull values
bitStrings
prefix sums
Prefix sum-based NULL Compressed block
finding value at 9 :
chunkIdx = 2
position of element in chunk (i) = 1
position in notNull values array = 4 + 0 = 4
Bit position to index Map

Figure 7: NULL compression using a simplified Jacobson's bit vector rank index with chunk size 4.

map M with $2^c * c$ cells. $M[b,i]$ is the number of 1s before the i'th bit of a c-length bit string b. Let p be the offset which is non-NULL and b the c-length bit string chunk in the bit vector that p belongs to, and ps the array storing prefix sums in a block. Then rank(p) = $ps[p/c] + M[b, p \mod c]$.

The choice of c affects how big the pre-populated map is. A second parameter in this scheme is the number of bits m used for each prefix sum value, which determines how large a block we are compressing and the overhead this scheme has for each element. For an arbitrary m, c, we require: (i) $2^c * c * \lceil \log(c)/8 \rceil$ byte size map, because the map has $2^c * c$ cells and needs to store a log(c)-bit count value in each cell; (ii) we can compress a block of size 2^m; and (iii) we store one prefix sum for each c elements, so incur a cost of m/c extra bits per element. By default we choose $m = 16, c = 16$. We require $2^c * c * 1$ = 1MB-size map, can compress 2^m = 64K blocks, and incur $m/c = 1$ extra bit overhead for each element, so increase the overhead of reference [1]'s scheme from 1 to only 2 bits per element (but provide constant time access to non-NULL values).

6 LIST-BASED PROCESSING

We next motivate our list-based processor by discussing limitations of traditional Volcano-style tuple-at-a-time processors and block-based processors of columnar RDBMSs when processing n-n joins.

Example 2. *Consider the following query. P, F, S, and O abbreviate* PERSON, FOLLOWS, STUDYAT, *and* ORGANISATION.

```
MATCH (a:P)−[:F]→(b:P)−[:F]→(c:P)−[:S]→(d:O)
WHERE a.age > 50 and d.name = "UW"  RETURN *
```

Consider a simple plan for this query shown in Figure 8, which is akin to a left-deep plan in an RDBMS, on a graph where FOLLOWS are n-n edges and STUDYAT edges have single cardinality. Volcano-style tuple-at-a-time processing [22], which some GDBMSs adopt [41, 47], is efficient in terms of how much data is copied to the intermediate tuple. Suppose the scan matches a to a_1 and a_1 extends to k_1 many b's, $b_1 \ldots b_{k_1}$, and each b_i extends to k_2 many c's to $b_{ik_2} ..., b_{(i+1)k_2}$ (let us ignore the d extension for now). Although this generates $k_1 \times k_2$ tuples, the value a_1 would be copied only once to the tuple.

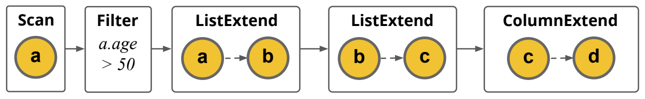


Figure 8: Query plan for the query in Example 2.

This is an important advantage for processing n-n joins. However, Volcano-style processors are known to achieve low CPU and cache utility as processing is intermixed with many iterator calls.

Column-oriented RDBMSs instead adopt block-based processors [10, 29], which process an entire block at a time in operators. Block sizes are fixed length, e.g. 1024 tuples [11, 17]. While processing blocks of tuples, operators read consecutive memory locations, achieving good cache locality, and perform computations inside loops over arrays which is efficient on modern CPUs. However, traditional block-based processors have two shortcomings for GDBMSs. (1) For n-n joins, block-based processing requires more data copying into intermediate data structures than tuple-at-a-time processing. Suppose for simplicity a block size of k_2 and k_1<k_2. In our example, the scan would output an array $a : [a_1]$, the first join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_{k_1}]$ blocks, and the second join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_1]$, $c : [c_1, ..., c_{k_2}]$, where for example the value a_1 gets copied k_2 times into intermediate arrays. (2) Traditional block-based processors do not exploit the list-based data organization of GDBMSs. Specifically, the adjacency lists that are used by join operators are already stored consecutively in memory, which can be exploited to avoid materializing these lists into blocks.

We developed a new block-based processor called *list-based processor* (LBP), which we next describe. LBP uses *factorized representation of intermediate tuples* [8, 51, 52] to address the data copying problem and uses block sizes set to the lengths of adjacency lists in the database, to exploit list-based data storage in GDBMSs.

6.1 Intermediate Tuple Set Representation

Traditional block-based processors represent intermediate data as a set of *flat* tuples in a single group of blocks/arrays. In our example we had three variables a, b, and c corresponding to three arrays. The values at position i of all arrays form a single tuple. Therefore to represent the tuples that are produced by n-n joins, repetitions of values are necessary. To address these repetitions we adopt a *factorized tuple set representation* scheme [52]. Instead of flat tuples, factorized representation systems represent tuples as unions of Cartesian products. For example, the k_2 flat tuples $[(a_1, b_1, c_1) \cup (a_1, b_1, c_2) \cup ... \cup (a_1, b_1, c_{k_2})]$ from above can be represented more succinctly in a factorized form as: $[(a_1) \times (b_1) \times (c_1 \cup ... \cup c_{k_2})]$.

To adopt factorization in block-based processing, we instead use multiple groups of blocks, which we call *list groups*, to represent intermediate data. Each list group has a curIdx field and can be in one of two states:

- Flat: If curIdx ≥ 0, the list group is flattened and represents a single tuple that consists of the curIdx'th values in the blocks.
- Unflat list of tuples: If curIdx $=-1$, the list groups represent as many tuples as the size of the blocks it contains.

We call the union of list groups *intermediate chunk*, which represents a set of intermediate tuples as the Cartesian product of each tuple that each list group represents.

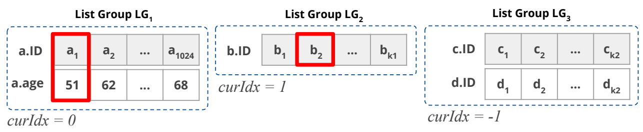


Figure 9: Intermediate chunk for the query in Example 2. The first two list groups are flattened to single tuples, while the last one represents k_2 many tuples.

Example 3. *Figure 9 shows an intermediate chunk, that consists of three list groups. The first two groups are flattened and the last is unflat. In its current state, the intermediate chunk represents k_2 intermediate tuples as:* $(a_1, 51) \times (b_2) \times ((c_1, d_1) \cup ... \cup (c_2, d_2))$.

In addition, instead of using fixed-length blocks as in existing block-based processors, the blocks in each group can take different lengths, which are aligned to the lengths of adjacency lists in the database. As we shortly explain, this allows us to avoid materializing adjacency lists into the blocks.

6.2 Operators

We next give a description of the main relational operators we implemented to process intermediate chunks in LBP.

`Scan`: Scans are the ame as before and read a fixed size (1024 by default) nodeIDs into a block in a list group.

`ListExtend` and `ColumnExtend`: In contrast to a single `Join` operator that implements index nested loop join algorithm using the adjacency list indices, such as `Expand` of Neo4j, we have two joins.

`ListExtend` is used to perform joins from a node, say, a to nodes b over 1-n or n-n edges e. The input list group LG_a that holds the block of a values can be flat or unflat. If LG_a is not flat, `ListExtend` first flattens it, i.e., sets the `curIdx` field of the list group to 0. It then loops through each a value, say, a_ℓ, and extends it to the set of b and e values using a_ℓ's adjacency list Adj_{a_ℓ}. The blocks holding b and e values are put to a new list group, LG_b. This allows factoring out a list of b and e values for a single a value. The lengths of all blocks in LG_b, including those storing b and e as well as blocks that may be added later, will be equal to the length of Adj_{a_ℓ}. This contrasts with fixed block sizes in existing block-based processors. In addition, we exploit that Adj_{a_ℓ} already stores b and e values as lists, and do not copy these to the intermediate chunk. Instead, the b and e blocks simply points to Adj_{a_ℓ}.

`ColumnExtend` is used to perform 1-1 or n-1 joins. We call the operator `ColumnExtend` because recall from Section 4.1.2 that we store such edges in vanilla vertex *columns*. Suppose now that each a can extend to at most one b node. `ColumnExtend` expects a block of unflat a values. That is, it expects LG_a to be unflat and adds two new blocks into LG_a, for storing b and e, that are of the same length as a's block (so unlike `ListExtend` does not create a new list group). Inside a for loop, `ColumnExtend` copies the matching e and b of each a from the vertex column to these two blocks. Note that because each a value has a single b and e value, these values do not need to be factored out.

`Filter`: LBP requires a more complex filter operator than those in existing block-based processors. In particular, in traditional block based processors, binary expressions, such as a comparison expression, can always assume that their inputs are two blocks of values. Instead, now binary expressions need to operate on three possible value combinations: two flat, two lists or one list and one flat, because any of the two blocks can now be in a flattened list group.

`Group By And Aggregate`: We omit a detailed description here and refer the reader to our code base [25]. Briefly, similar to `Filter`, `Group By And Aggregate` needs to consider whether the values it should group by or aggregate are flat or not, and performs a group by and aggregation on possibly multiple factorized tuples. Factorization allows LBP to sometimes perform fast group by and aggregations, similar to prior techniques that compute aggregations on compressed data [2, 60]. For example, count(*) simply multiplies the sizes of each list group to compute the number of tuples represented by each intermediate chunk it receives.

Example 4. *Continuing our example, the three list groups in Figure 9 are an example intermediate chunk output by the* `ColumnExtend` *operator in the plan from Figure 8. In this, the initial* `Scan` *and* `Filter` *have filled the 1024-size a and a.age blocks in LG_1. The first* `ListExtend` *has: (i) flattened LG_1 to tuple* $(a_1, 51)$*; and (ii) filled a block of k_1 b values in a new list group LG_2. The second* `ListExtend` *has (i) flattened LG_2 and iterated over it once, so its* `curIdx` *field is 1, and LG_2 now represents the tuple* (b_2)*; and (ii) has filled a block of k_2 c values in a new list group LG_3. Finally, the last* `ColumnExtend` *fills a block of k_2 d values, also in LG_3, by extending each c_j value to one d_j value through the single cardinality* `STUDY_AT` *edges.*

7 UPDATES AND QUERY OPTIMIZATION

Although we do not focus on handling updates and query optimization within the scope of this paper, these components require further considerations in a complete integration of our techniques. As in columnar RDBMSs, the columnar storage techniques we covered are read-optimized and necessarily add several complexities to updates [60]. First recall from Section 4.1.1 that CSR data structure for storing adjacency list indexes are effectively sorted structures that are compressed by run-length encoding. So handling deletions or insertions requires resorting the CSRs and recalculating the CSR offsets. Insertion of edge properties in single-directional property pages are append only and do not require any sorting. Insertions to vertex columns are also simple as these too are unsorted structure. However, deletions of nodes or edges, require leaving gaps in vertex columns and single-directional property pages. This requires keeping track of these gaps and reusing them for new insertions. Note that this is also how node deletions are handled in Neo4j [46]. Finally, the null compression scheme we adopted requires three updates upon insertion and deletions: (i) changing the bit values in the bitstrings; (ii) re-calculating prefix sum values for prefixes after the location of update; and (iii) shifting the non-NULL elements array. These added complexities are an inherent trade off when integrating read-optimized techniques and can be mitigated by several existing techniques, like bulk updates or keeping a write-optimized second storage that keeps track of recent writes, which are not immediately merged. Positional delta trees [28] or C-Store's write-store are examples [57] of the latter technique.

Two of our techniques also require additional considerations when modifying the optimizer. First, our use of factorized list groups changes the size of tuples that are passed between operators, as the intermediate tuples are now compressed. When assigning costs to plans, the compressed sizes, instead of the flattened sizes of these tuples should be considered. In addition, scans of properties that are stored in, say forward single-directional property pages, behave differently when the properties are scanned in the forward direction (e.g., after a join that has used the forward adjacency lists) vs the backward direction. The former leads to sequential reads while the latter to random reads. The optimizer should assign costs based on this criterion as well. We leave a detailed study of how to handle updates and optimize queries under our techniques to future work.

8 EVALUATIONS

We integrated our columnar techniques into GraphflowDB, an in-memory GDBMS written in Java. We refer to this version of GraphflowDB as `GF-CL` (**C**olumnar **L**ist-based). We based our work on the publicly available version here [24], which we will refer to as `GF-RV` (**R**ow-oriented **V**olcano). `GF-RV` uses 8 byte vertex and edge IDs and adopts the interpreted attribute layout to store edge and vertex properties. `GF-RV` also partitions adjacency lists by edge labels and stores the (edge ID, neighbour ID) pairs inside a CSR. We present both microbenchmark experiments comparing `GF-RV` and `GF-CL` and baseline experiments against Neo4j, MonetDB, and Vertica using LDBC and JOB benchmarks. Due to space constraints our experiment demonstrating benefits of vertex columns for single cardinality edges appears in the longer version of our paper [26].

8.1 Experimental Setup

Hardware Setup: For all our experiments, we use a single machine that has two Intel E5-2670 @ 2.6GHz CPUs and 512 GB of RAM. We only use one logical core. We set the maximum size of the JVM heap to 500 GB and keep JVM's default minimum size.

Datasets: Our LBP is designed to yield benefits under join queries over 1-n and n-n relationships. Our storage compression techniques exploit some structure in the dataset and NULLs. These techniques are not designed for datasets that do not depict structure, e.g., a highly heterogenous knowledge graph, such as DBPedia. We choose the following datasets and queries that satisfy these requirements:

LDBC: We generated the LDBC social network data [18] using scale factors 10 and 100, which we refer to as LDBC10 and LDBC100, respectively. In LDBC, all of the edges and edge and vertex properties are structured but several properties and edges are very sparse. LDBC10 contains 30M vertices and 176.6M edges while LDBC100 contains 1.77B edges and 300M vertices. Both datasets contain 8 vertex labels, 15 edge labels and 34 (29 vertex, 5 edge) properties.

JOB: We used the IMDb movie database and the JOB benchmark [35]. Although the workload has originally been created to study optimizing join order selection, the dataset contains several n-n, 1-n, and 1-1 relationships between entities, like actors, movies, and companies, and structured properties, some of which contain NULLs. This makes it suitable to demonstrate the benefits from our storage and compression techniques. JOB also contains join queries over n-n relationships, making it suitable to demonstrate benefits of LBP. We created a property graph version of this database and workload as follows. IMDb contains three groups of tables: (i) *entity tables* representing entities, such as actors (e.g., `name` table), movies, and companies; (ii) *relationship tables* representing n-n relationships between the entities (e.g., the `movie_companies` table represents relationships between movies and companies); and (iii) *type tables*, which denormalize the entity or relationship tables to indicate the types of entities or relationships. We converted each row of an entity table to a vertex. Let u and v be vertices representing, respectively, rows r_u and r_v from tables T_u an T_v. We added two sets of edges between u and v: (i) a *foreign key edge* from u to v if the primary key of row r_u is a foreign key in row r_v; (ii) a *relationship edge* between u to v if a row r_ℓ in a relationship table T_ℓ connects row r_u and r_v. The final dataset can be found in our codebase [25].

`FLICKR` *and* `WIKI`*:* To enhance our microbenchmarks further, we use two additional datasets from the popular Konect graph sets [33] covering two application domains: a Flickr social network (`FLICKR`) [42] and a Wikipedia hyperlink graph between articles of the German Wikipedia (`WIKI`) [34]. Flickr graph has 2.3M nodes and 33.1M edges while Wikipedia graph has 2.1M nodes and 86.3M edges. Both datasets have timestamps as edge properties.

In each experiment, we ran our queries 5 times consecutively and report the average of the last 3 runs. We did not observe large variances in these experiments. Across all of the LDBC and JOB benchmark queries we report, the median difference between the minimum and maximum of the 3 runs we report was 1.02% and the largest was 25%, which was a query in which the maximum run was 24ms while the minimum was 19ms.

8.2 Memory Reduction

We first demonstrate the memory reduction we get from the columnar storage and compression techniques we studied using LDBC100 and IMDb. We start with `GF-RV` and integrate one additional storage optimization step-by-step ending with `GF-CL`:

(i) `+COLS`: Stores vertex properties in vertex columns, edge properties in single-directional property pages, and single cardinality edges in vertex columns (instead of CSR).

(ii) `+NEW-IDS`: Introduces our new vertex and edge ID schemes and factors out possible ID components (recall Section 5.2).

(iii) `+0-SUPR`: Implements leading 0 suppression in the components of vertex and edge IDs in adjacency lists.

(iv) `+NULL`: Implements NULL compression of empty lists and vertex properties based on Jacobson's index.

Table 2a shows how much memory each component of the system as well as the entire system take after each optimization. On LDBC, we see 2.96x and 2.74x reduction for storing forward and backward adjacency lists, respectively. We reduce memory significantly by using the new ID scheme that factors out components, such as edge and vertex labels, and using small size integers for positional offsets. We also see a 1.58x reduction by storing vertex properties in columns, which, unlike interpreted attribute layout, saves on storing the keys of the properties explicitly. The modest memory gains in `+COLS` for storing adjacency lists is due to the fact that 8 out of 15 edge labels in LDBC SNB are single cardinality and storing them in vertex columns is cheaper than in CSRs, as we do not need CSR offsets. We see a reduction of 3.82x when storing edge properties in single-directional property pages. This is primarily because `GF-RV` stores a pointer for each edge, even if the edges with a particular label have no properties. `GF-CL` stores no columns for these edges, so incurs no overheads and avoids storing the keys of the properties explicitly. We see modest benefits in NULL compression since empty adjacency lists are infrequent in LDBC100

Table 2: Memory reductions after applying one more optimization on top of the configuration on the left.

(a) LDBC100

	GF-RV	+COLS	+NEW-IDS	+0-SUPR	+NULL	GF-CL
Vertex	31.40	19.87	19.87	19.87	19.28	-
Props.		**+1.58x**	-	-	**+1.03x**	**1.62x**
Edge	7.92	2.07	2.07	2.07	2.05	-
Props.		**+3.82x**	-	-	**+1.01x**	**3.87x**
F. Adj.	31.93	28.95	20.67	11.41	10.78	-
Lists		**+1.10x**	**+1.40x**	**+1.81x**	**+1.06x**	**2.96x**
B. Adj.	31.29	31.07	24.93	13.10	11.41	-
Lists		**+1.01x**	**+1.25x**	**+1.90x**	**+1.15x**	**2.74x**
Total (GB)	102.56	81.97	67.55	46.45	43.54	-
		+1.25x	**+1.21x**	**+1.45x**	**+1.07x**	**2.36x**

(b) IMDb

	GF-RV	+COLS	+NEW-IDS	+0-SUPR	+NULL	GF-CL
Vertex	2.54	1.98	1.98	1.98	1.96	-
Props.		**+1.28x**	-	-	**+1.01x**	**1.29x**
Edge	2.81	1.63	1.63	1.63	0.90	-
Props.		**+1.72x**	-	-	**+1.83x**	**3.14x**
F. Adj.	1.13	1.02	0.65	0.41	0.36	-
Lists		**+1.10x**	**+1.57x**	**+1.57x**	**+1.15x**	**2.96x**
B. Adj.	1.10	1.10	0.76	0.50	0.49	-
Lists		**+1.00x**	**+1.45x**	**+1.51x**	**+1.01x**	**2.20x**
Total (GB)	7.57	5.74	5.02	4.53	3.72	-
		+1.32x	**+1.14x**	**+1.11x**	**+1.22x**	**2.03x**

and 26 of 29 vertex properties and all of the edge properties contain no NULL values. Overall, we obtained a reduction of 2.36x on LDBC100, reducing the memory cost from 102.5GB to 43.5GB.

The reductions on IMDb are shown in Table 2b and are broadly similar to LDBC. For example, we see 2.96x and 2.2x reduction factors in forward and backward lists, which is comparable to that of LDBC. However, there are two main differences. First, we save more by compressing the edge properties using NULL compression, because 7 of 12 edge properties in IMDb have more than 50% null values. Second, instead of a 3.82x reduction by storing edge properties using single directional property columns and single cardinality edges in vertex columns (+COLS column of Edge Props row), the factor is now 1.72x. This is because all of the edge properties in LDBC are 4-byte integers. Instead, IMDb has primarily string edge properties (8 out of 12 of the edge properties), so these take more space compared to integers. Hence, the storage savings per byte of actual data is higher in case of LDBC. Overall, the total reduction factor is 2.03x reducing the memory overheads from 7.57G to 3.72G.

8.3 Single-Directional Property Pages

We next demonstrate the query performance benefits of storing edge properties in single-directional property pages. We configure GraphflowDB in two ways: (i) EDGE COLS: Stores edge properties in an edge column in a randomized way as edges are given random edge IDs; (ii) PROP PAGES: Edge properties are stored in forward-directional property pages with k=128 . In the longer version of our paper [26], we test sensitivity of k that demonstrates read performance from property pages in our datasets are similar until k=512 and slows down for larger value of k .

Table 3: Runtime (in secs) of k-hop (H) queries when using property pages (PAGE$_P$) vs edge columns (COL$_E$).

		LDBC100		WIKI		FLICKR	
		1H	**2H**	**1H**	**2H**	**1H**	**2H**
P$_F$	COL$_E$	0.55	65.22	2.97	42.92	1.88	888.30
	PAGE$_P$	0.16	34.22	0.96	16.48	0.42	189.39
		3.4x	**1.9x**	**3.1x**	**2.6x**	**4.5x**	**4.7x**
P$_B$	COL$_E$	1.23	131.01	6.33	99.28	2.40	1009.84
	PAGE$_P$	1.29	134.43	6.10	91.75	2.25	1183.14
		0.9x	**1.0x**	**1.0x**	**1.1x**	**1.1x**	**0.9x**

We use LDBC100, WIKI, and FLICKR datasets. As our workload, we use 1- and 2-hop queries, i.e., queries that enumerate all edges and 2-paths, with predicates on the edges. For LDBC, the paths enumerate Knows edges (WIKI and FLICKR contain only one edge label). 1-hop query compares the edge's timestamp for WIKI and FLICKR and the creationDate property for LDBC to be greater than a constant. 2-hop query compares the property of each query edge to be greater than the previous edge's property. Since WIKI contains prohibitively many 2-hops we put a predicate on the source and destination nodes to make queries finish within reasonable time. For each query and configurations, we consider two plans: (i) the *forward plan* that matches vertices from left to right in forward direction; (ii) the *backward plan* that matches in reverse order.

Forward plans perform sequential reads of properties under PROP-PAGES, achieving good CPU cache locality. Therefore, they are expected to be more performant than backward plans under PROP-PAGES as well as both the plans plans under EDGE COLS, which all lead to random reads. We also expect backward plans to behave similarly under both configurations. Table 3 shows our results. Observe that forward plans under PROP-PAGES is between 1.9x to 4.7x faster than the forward plans under EDGE COLS and are also faster than the backward plans under PROP-PAGES. In contrast, the performance of both backward plans are comparable. This is because neither edge columns nor forward-directional property pages provide any locality for accessing properties in order of backward adjacency lists. This confirms our claim in Section 4.2 that PROP-PAGES is a better design than using vanilla edge columns.

8.4 Null Compression

We demonstrate the memory/performance trade-off of our NULL compression scheme on sparse vertex property columns. We create multiple versions of the LDBC100, with the creationDate property of Comment vertices containing different percentage of NULL values. LDBC100 contains 220M Comment vertices, so our column has 220M entries. We use the following 1-hop query: MATCH (a:Person)–[e:Likes]→(b:Comment) RETURN b.creationDate. This query is evaluated with a simple plan that scans a, extends to b, and then a sink operator that reads b.creationDate. We compare the query performance and the memory cost of storing the creationDate column, when it is stored in three different ways: (i) J-NULL compresses the column using Jacobson's bit index with default configuration (m=16, c=16); (ii) Vanilla-NULL is the vanilla bit string-based scheme from reference [1]; and (ii) Uncompressed stores the column in an uncompressed format. In the longer version of our paper [26], we demonstrate a sensitivity analysis for

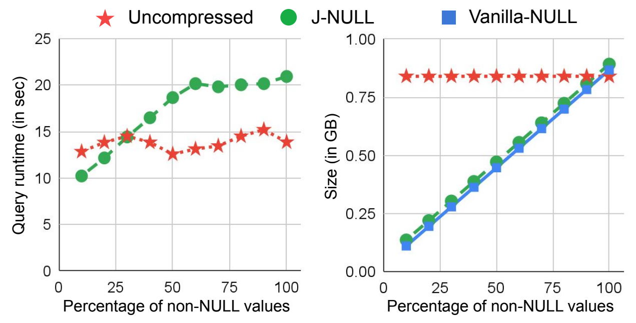


Figure 10: Query performance and memory consumption when storing a vertex property column as uncompressed, compressed with Jacobson's scheme, and the vanilla bit string scheme from Abadi, under different density levels.

J-NULL running under different m and c values. This experiment shows that read performance is insensitive to these parameters. The memory overhead increases as m increases, albeit marginally. So a reasonable choice is picking $m = c = 16$, which incurs 1 bit extra overhead per element for storing prefix sums.

Figure 10 shows the memory usage and query performance under three different configurations. Recall that with default configuration J-NULL requires slightly more memory than Vanilla-NULL, 2 bits per element instead of 1 bit. As expected the performance of J-NULL is slightly slower than Uncompressed, between 1.19x and 1.51x, but much faster than Vanilla-NULL, which was >20x slower than J-NULL and is therefore omitted in Figure 10. Interestingly, when the column is sparse enough (with >70% NULL values), J-NULL can even outperform Uncompressed. This is because when the column is very sparse, accesses are often to NULL elements, which takes one access for reading the bit value of the element. When the bit value is 0, iterators return a global NULL value which is likely to be in the CPU cache. Instead, Uncompressed always returns the value at element's cell, which has a higher chance of a CPU cache miss.

8.5 List-based Processor

We next present experiments demonstrating the performance benefits of LBP against a traditional Volcano-style tuple-at-a-time processor, which are adopted in existing systems, like Neo4j [47] or MemGraph [40]. LBP has three advantages over traditional tuple-at-a-time processor: (1) all primitive computations over data happen inside loops as in block-based operators; (2) the join operator can avoid copies of edge ID-neighbour ID pairs into intermediate tuples, exploiting the list-based storage; and (3) we can perform group-by and aggregation operations directly on compressed data. We present two separate sets of experiments that demonstrate the benefits from these three factors. To ensure that our experiments only test differences due to query processing techniques, we integrated our columnar storage and compression techniques into GF-RV (recall that this is GraphflowDB with row-based storage and Volcano-style processor). We call this version GF-CV, for **C**olumnar **V**olcano.

We use LDBC100, Wikipedia, and Flickr datasets. In our first experiment, we take 1-, 2-, and 3-hop queries (as in Section 8.3, we use the Knows edges in LDBC100), where the last edge in the path has a predicate to be greater than a constant (e.g., e.date > c). For both GF-CV and GF-CL, we consider the standard plan that

Table 4: Runtime (ms) of GF-RV and GF-CL (LBP) plans.

			1-hop	2-hop	3-hop
LDBC100	FILTER	GF-CV	24.6	1470.5	40252.4
		GF-CL	7.7 **3.2x**	116.2 **12.7x**	2647.3 **15.2x**
	COUNT(*)	GF-CV	13.4	241.9	6947.3
		GF-CL	4.2 **3.2x**	18.9 **12.8x**	357.9 **19.4x**
FLICKR	FILTER	GF-CV	32.6	1300.0	14864.0
		GF-CL	12.2 **2.7x**	95.3 **13.7x**	1194.7 **12.4x**
	COUNT(*)	GF-CV	35.3	519.2	4162.5
		GF-CL	16.9 **2.1x**	23.4 **21.4x**	51.7 **80.6x**
WIKI	FILTER	GF-CV	35.8	4500.2	236930.2
		GF-CL	11.9 **2.9x**	1192.5 **3.8x**	20329.3 **11.7x**
	COUNT(*)	GF-CV	32.7	1745.2	109000.2
		GF-CL	19.0 **1.7x**	27.6 **63.2x**	120.4 **905.1x**

scans the left most node, extends right to match the entire path, and a final Filter on the date property of the last extended edge. A major part of the work in these plans happen at the final join and filter operation, therefore these plans allow us to measure the performance benefits of performing computations inside loops and avoiding data copying in joins. Our results are shown in the FILTER rows of Table 4. We see that GF-CL outperforms GF-CV by large margins, between 2.7x and 15.2x.

In our second experiment, we demonstrate the benefits of performing fast aggregations over compressed intermediate results. We modify the previous queries by removing the predicate and instead add a return value of count(*). We use the same plans as before except we change the last Filter operator with a GroupBy operator. Our results for aggregation are shown in the COUNT(*) rows of Table 4. Observe that the improvements are much more significant now, up to close to three orders of magnitude on Wiki (by 905.1x). The primary advantage of GF-CL is now that the counting happens on compressed intermediate results.

8.6 Baseline System Comparisons

In our final experiment, we compare the query performance of GF-CL against GF-RV, Neo4j, which is another row-oriented and Volcano style GDBMSs, and two columnar RDBMSs, MonetDB and Vertica, which are not tailored for n-n joins. Our primary goal is to verify that GF-CL is faster than GF-RV also on an independent end-to-end benchmark. We also aim to verify that GF-RV, on which we base our work, is already competitive with or outperforms other baseline systems on workloads containing n-n joins. We used the SNB on LDBC10 and JOB, both of which contain n-n join queries.

We used the community version v4.2 of Neo4j GDBMS [47], the community version 10.0 of Vertica [61] and MonetDB 5 server 11.37.11 [43]. We note that our experiments should not be interpreted as one system being more efficient than another. It is difficult to meaningfully compare completely separate systems, e.g., all baseline systems have many tunable parameters, and some have more

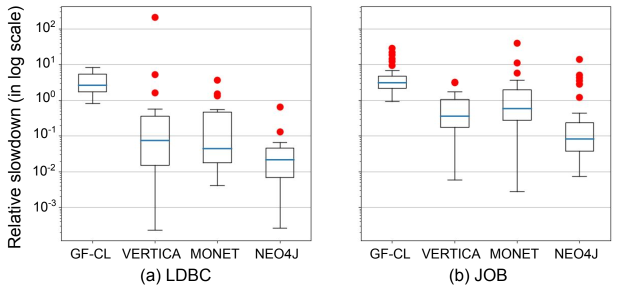


Figure 11: Relative speedup/slowdown of the different systems in comparison to GF-RV on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

efficient enterprise versions. For all baseline systems, we map their storage to an in-memory filesystem, set number of CPUs to 1 and disable spilling intermediate files to disk. We maintain 2 copies of edge tables for Vertica and MonetDB, sorted by the source and destination vertexIDs, respectively. For GF-RV and GF-CL, we use the best left-deep plan we could manually pick, which was obvious in most cases. For example, LDBC path queries start from a particular vertex ID, so the best join orders start from that vertex and iteratively extend in the same direction. For Vertica, MonetDB, and Neo4j, we use the better of the systems' default plans and the left-deep that is equivalent to the one we use in GF-RV and GF-CL.

8.6.1 LDBC. We use the LDBC10 dataset. GraphflowDB is a prototype system that implements parts of the Cypher language relevant to our research, so lack several features that LDBC queries exercise. The system currently has support for select-project-join queries and a limited form of aggregations, where joins are expressed as fixed-length subgraph patterns in the MATCH clause. We modified the Interactive Complex Reads (IC) and Interactive Short Reads (IS) queries from LDBC [18] in order to be able to run them. Specifically GraphflowDB does not support variable length queries that search for joins between a minimum and maximum length, which we set to the maximum length to make them fixed-length instead, and shortest path queries, which we removed from the benchmark. We also removed predicates that check the existence or non-existence of edges between nodes and the ORDER BY clauses. Our exact queries can be found in the longer version of our paper [26].

Figure 11a shows the relative speedup/slowdown of the different systems in comparison to GF-RV. We report individual runtime numbers of all the queries in the longer version of our paper [26]. As expected, GF-CL is broadly more performant than GF-RV on LDBC with a median query improvement factor of 2.6x. With the exception of one query, which slows down a bit, the performance of *every query* improves between 1.3x to 8.3x. The improvements come from several optimizations but primarily from LBP and our columnar storage. In GF-RV, scanning properties requires checking equality on property keys, which are avoided in columnar storage, so we observed large improvements on queries that produce large intermediate results and perform filters, such as IC05. IC05 has 4 n-n joins starting from a node and extending in the forward direction and a predicate on the edges of the third join. GF-CL has several advantages that become visible here. First, GF-CL's LBP, unlike GF-RV, does not copy any edge and neighbour IDs to intermediate tuples. More importantly, LBP performs filters inside loops and GF-CL's single-indexed property pages provides faster access to the edge properties that are used in the filter than GF-RV's row-oriented format. On this query, GF-RV takes 8.9s while GF-CL takes 1.6s.

As we expected, we also found other baseline systems to not be as performant as GF-CL or GF-RV. In particular, Vertica, MonetDB, and Neo4j have median slowdown factors of 13.1x, 22.8x, and 46.1x compared to GF-RV. Although Neo4j performed slightly worse than other baselines, we also observed that there were some queries in which it outperformed Vertica and MonetDB (but not GF-RV or GF-CL) by a large margin. These were queries that started from a single node, had several n-n joins, but did not generate large intermediate results, like IS02 or IC06. On such queries, GDBMSs, both GraphflowDB and Neo4j, have the advantage of using join operators that use the adjacency list indices to extend a set of partial matches. This can be highly efficient if the partial matches that are extended are small in number. For example, the first join of IC06 extends a single Person node, say p_i, to its two-degree friends. In SQL, this is implemented as joining a Person table with a Knows table with a predicate on the Person table to select p_i. In Vertica or MonetDB, this join is performed using merge or hash joins, which requires scanning both Person and Knows tables. Instead, Neo4j and GraphflowDB only scan the Person table to find p_i and then extend p_i to its neighbours, without scanning all Knows edges. For this, GF-RV, GF-CL, and Neo4j take 333ms, 113ms, and 515ms, while Vertica and MonetDB take 4.7s and 2.7s, respectively. We also found that all baseline systems, including Neo4j, degrade in performance on queries with many n-n joins that generate large intermediate results. For example, on IC05 that we reviewed before, Vertica take 1 minute, MonetDB 3.25 minutes, while Neo4j took over 10 minutes.

8.6.2 JOB. JOB queries come in four variants and we used their first variant. We converted the JOB queries to their Cypher equivalent following our conversion of the dataset. Many of the JOB queries returned aggregations on strings, such as min(name), where name is a string column. Since Graphflow supports aggregations only on numeric types, we removed these aggregations. Our final queries can be found in the longer version of our paper [26].

Figure 11b shows the relative performance of different systems in comparison to GF-RV. The individual runtime numbers of each query can be found in the longer version of our paper [26]. Similar to our LDBC results, we see GF-CL to improve the performance, now by 3.1x. Again similar to LDBC, with the exception of one query, we see consistent speed ups across all queries between 1.5x and 28.8x. Different from LDBC, we also see queries on which the improvement factors are much larger, i.e, >20x. In LDBC, the largest improvement factor was 8.3x. This is expected as most of the queries in JOB perform star joins while LDBC queries contained path queries that start from a node with a selective filter. On path queries, our plans start from a single node and extend in one direction, in which case only the last extension can truly be factorized, so be in unflat form. This is because each ListExtend that we use first flattens the previously extended node. Whereas on star queries, multiple extensions from the center node can remain unflattened. Therefore GF-CL's plans can benefit more from LBP as they can compress their intermediate tuples more. We also see that similar to LDBC, GF-RV is more performant than the columnar RDBMSs.

However, these systems are now more competitive. We noticed that one reason for this is that on star queries, these systems's default plans are often bushy plans (27 out of 33 for MonetDB and 26 out of 33 for Vertica), which produce fewer intermediate tuples than GF-RV, which does not benefit from factorization and uses left-deep plans. So these systems now benefit from bushy plans which they did not in LDBC. In contrast, on LDBC, these systems would also primarily use left-deep plans (only 2 out of 18 for MonetDB and 4 out of 18 for Vertica were bushy) because on these path queries, it is better to start from a single highly filtered node table and join iteratively in a left-deep plan to match the entire path. Finally, similar to LDBC, Neo4j is again least competitive of these baselines.

9 RELATED WORK

Column stores [29, 57, 66, 67] are designed primarily for OLAP queries that perform aggregations over large amounts of data. Work on them introduced a set of storage and query processing techniques which include use of positional offsets, schemes for compression, block-based query processing, late materialization and operations on compressed data, among others. A detailed survey of these techniques can be found in reference [60]. This paper aims to integrate some of these techniques into in-memory GDBMSs.

Existing GDBMSs and RDF systems usually store the graph topology in a columnar structure. This is done either by using a variant of adjacency list or CSR. Instead, systems often use row-oriented structures to store properties, such as an interpreted attribute layout [9]. For example, Neo4j [47] represents the graph topology in adjacency lists that are partitioned by edge labels and stored in linked-lists, where each edge record points to the next. Properties of each vertex/edge are stored in a linked-list, where each property record points to the next and encodes the key, data type, and value of the property. JanusGraph too [6] stores edges in adjacency lists partitioned by edge labels and properties as consecutive key-value pairs (a row-oriented format). These native GDBMSs adopt Volcano-style processors. In contrast, our design adopts columnar structures for vertex and edge properties and a block-based processor. In addition, we compress edge and vertex IDs and NULLs.

There are also several GDBMSs that are developed directly on top of an RDBMS or another database system [54], such as IBM Db2 Graph [58], Oracle Spatial and Graph [54] and SAP's graph database [55]. These systems can benefit from the columnar techniques in the underlying RDBMS, which are however not optimized for graph storage and queries. For example, SAP's graph engine uses SAP HANA's columnar-storage for edge tables but these tables do not have CSR-like structures for storing edges.

GQ-Fast [38] implements a limited SQL called relationship queries that support joins of tables similar to path queries, followed with aggregations. The system stores n-n relationship in tables with CSR-like indices and heavy-weight compression of lists and has a fully pipelined query processor that uses query compilation. Therefore, GQ-Fast studies how some techniques in GDBMSs, specifically joins using adjacency lists, can be done in RDBMS. In contrast, we focus on studying how some techniques from columnar RDBMSs can be integrated into GDBMSs. We intended to but could not compare against GQ-Fast because the system supports a very limited set of queries (e.g., none of the LDBC queries are supported).

Several RDF systems also use columnar structures to store RDF data. Reference [4] uses a set of columns, where each column store is a set of (subject, object) pairs for a unique predicate. However, this storage is not as optimized as the storage in GDBMSs, e.g., the edges between entities are not stored in native CSR or adjacency list format. Hexastore [62] improves on the idea of predicate partitioning by having a column for each RDF element (subject, predicate or object) and sorting it in 2 possible ways in B+ trees. This is similar but not as efficient as double indexing of adjacency lists in GDBMSs. RDF-3X [50] is an RDF system that stores a large triple table that is indexed in 6 B+ tree indexes over each column. Similarly, this storage is not as optimized as the native graph storages found in GDBMSs. Similar to our Guideline 3, reference [49] also observes that graphs have structure, and certain predicates in RDF databases co-exist together in a node. This is similar to the property co-occurrence structure we exploit, and is exploited in the RDF 3-X system for better cardinality estimation.

Several novel storage techniques for storing graphs are optimized for write-heavy workloads, such as streaming. These works propose data structures that try to achieve the sequential read capabilities of CSR while being write-optimized. Examples of this include LiveGraph [65], Aspen [16], and LLAMA [39]. We focus on a read-optimized system setting and use CSR to store the graph topology but these techniques are complementary to our work.

Our list groups represent intermediate results in a factorized form. Prior work on factorized representations in RDBMSs, specifically FDB [7, 8], represents intermediate data as tries, and have operators that transform tries into other tries. Unlike traditional processors, processing is not pipelined and all intermediate results are materialized. Instead, operators in LBP are variants of traditional block-based operators and perform computations in a pipelined fashion on batches of lists/arrays of data. This paper focuses on integration of columnar storage and query processing techniques into GDBMSs and does not studies how to integrate more advanced factorized processing techniques inside GDBMS.

10 CONCLUSIONS

Columnar RDBMSs are read-optimized analytical systems that have introduced several storage and query processing techniques to improve the scalability and performances of RDBMSs. We studied the integration of such techniques into GDBMSs, which are also read-optimized analytical systems. While some techniques can be directly applied to GDBMSs, adaptation of others can be significantly sub-optimal in terms of space and performance. In this paper, we first outlined a set of guidelines and desiderata for designing the storage layer and query processor of GDBMSs, based on the typical access patterns in GDBMSs which are significantly different than the typical workloads of columnar RDBMSs. We then presented our design of columnar storage, compression, and query processing techniques that are optimized for in-memory GDBMSs. Specifically, we introduced a novel list-based query processor, which avoids expensive data copies of traditional block-based processors and avoids materialization of adjacency lists in blocks, a new data structure we call single-indexed property pages and an accompanying edge ID scheme, and a new application of Jacobson's bit vector index for compressing NULL and empty lists.

REFERENCES

[1] Daniel J. Abadi. 2007. Column Stores for Wide and Sparse Data. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007*. 292–297. http://cidrdb.org/cidr2007/papers/cidr07p33.pdf

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2006*. 671–682. https://doi.org/10.1145/1142473.1142548

[3] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*. 967–980. https://doi.org/10.1145/1376616.1376712

[4] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data*. 411–422. http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf

[5] Amazon. 2020. Amazon Neptune. https://aws.amazon.com/neptune/. Last Accessed July 25, 2021.

[6] JanusGraph Authors. 2020. JanusGraph. https://janusgraph.org. Last Accessed July 25, 2021.

[7] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1990–2001. http://www.vldb.org/pvldb/vol6/p1990-zavodny.pdf

[8] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1232–1243. http://vldb.org/pvldb/vol5/p1232_nurzhanbakibayev_vldb2012.pdf

[9] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. 2006. Extending RDBMSs to Support Sparse Datasets using an Interpreted Attribute Storage Format. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*. 58. https://doi.org/10.1109/ICDE.2006.67

[10] Peter Boncz. 2002. *Monet: A Next-Generation Database Kernel for Query-Intensive Applications*. Ph.D. Dissertation. Universiteit van Amsterdam. https://ir.cwi.nl/pub/14832/14832A.pdf

[11] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005*. 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[12] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[13] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling - 38th International Conference, ER 2019 (Lecture Notes in Computer Science)*, Vol. 11788. 448–456. https://doi.org/10.1007/978-3-030-33223-5_37

[14] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. 2009. Succinct Orthogonal Range Search Structures on a Grid with Applications to Text Indexing. In *Algorithms and Data Structures, 11th International Symposium, WADS 2009 (Lecture Notes in Computer Science)*, Vol. 5664. 98–109. https://doi.org/10.1007/978-3-642-03367-4_9

[15] DGraph. 2020. DGraph Github Repository. https://github.com/dgraph-io/dgraph. Last Accessed July 25, 2021.

[16] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency Graph Streaming using Compressed Purely-functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*. 918–934. https://doi.org/10.1145/3314221.3314598

[17] DuckDB. 2020. DuckDB. https://duckdb.org/. Last Accessed July 25, 2021.

[18] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015*. 619–630. https://doi.org/10.1145/2723372.2742786

[19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, SIGMOD 2018*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[20] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, ICDE 1998*. 370–379. https://doi.org/10.1109/ICDE.1998.655800

[21] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. 1992. New Indices for Text: Pat Trees and Pat Arrays. In *Information Retrieval: Data Structures & Algorithms*. 66–82.

[22] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering, TKDE* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[23] G. Graefe and L.D. Shapiro. 1991. Data Compression and Database Performance. In *Proceedings of the 1991 Symposium on Applied Computing*. 22–27. https://doi.org/10.1109/SOAC.1991.143840

[24] Graphflow. 2020. GraphflowDB Source Code. https://github.com/queryproc/optimizing-subgraph-queries-combining-binary-and-worst-case-optimal-joins/. Last Accessed July 25, 2021.

[25] Graphflow. 2021. GraphflowDB Columnar Techniques. https://github.com/graphflow/graphflow-columnar-techniques. Last Accessed July 25, 2021.

[26] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. *Columnar Storage and List-based Processing for Graph Database Management Systems*. Technical Report. https://github.com/graphflow/graphflow-columnar-techniques/blob/master/paper.pdf

[27] Olaf Hartig and Jan Hidders. 2019. Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), GRADES-NDA 2019*. 6:1–6:11. https://doi.org/10.1145/3327964.3328495

[28] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. 2010. Positional Update Handling in Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*. 543–554. https://doi.org/10.1145/1807167.1807227

[29] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[30] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*. 549–554. https://doi.org/10.1109/SFCS.1989.63533

[31] Guy Jacobson. 1989. *Succinct Static Data Structures*. Ph.D. Dissertation. Carnegie Mellon University.

[32] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, SIGMOD 2017*. 1695–1698. https://doi.org/10.1145/3035918.3056445

[33] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *22nd International World Wide Web Conference, WWW 2013*. 1343–1350. https://doi.org/10.1145/2487788.2488173

[34] Jérôme Kunegis. 2021. Wikipedia Dynamic (de), (Konect). http://konect.cc/networks/link-dynamic-dewiki/. Last Accessed July 25, 2021.

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. http://www.vldb.org/pvldb/vol9/p204-leis.pdf

[36] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers per Second through Vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. https://doi.org/10.1002/spe.2203

[37] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD 2005*. 177–187. https://doi.org/10.1145/1081870.1081893

[38] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. *Proceedings of the VLDB Endowment* 10, 3 (2016), 265–276. http://www.vldb.org/pvldb/vol10/p265-lin.pdf

[39] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[40] Memgraph. 2020. Memgraph. https://memgraph.com/. Last Accessed July 25, 2021.

[41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704. http://www.vldb.org/pvldb/vol12/p1692-mhedhbi.pdf

[42] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the Flickr Social Network. In *Proceedings of the first Workshop on Online Social Networks, WOSN 2008*. 25–30. https://doi.org/10.1145/1397735.1397742

[43] MonetDB. 2020. MonetDB source code, (Jun2020-SP1). https://github.com/MonetDB/MonetDB/releases/tag/Jun2020_SP1_release. Last Accessed July 25, 2021.

[44] Gonzalo Navarro and Veli Mäkinen. 2007. Compressed Full-Text Indexes. *Comput. Surveys* 39, 1 (2007), 2. https://doi.org/10.1145/1216370.1216372

[45] Gonzalo Navarro, Yakov Nekrich, and Luís M. S. Russo. 2013. Space-efficient Data-Analysis Queries on Grids. *Theoretical Computer Science* 482 (2013), 60–72. https://doi.org/10.1016/j.tcs.2012.11.031

[46] Neo4j. 2020. Neo4j Blog on Deletions. https://neo4j.com/developer/kb/how-deletes-workin-neo4j/. Last Accessed July 25, 2021.

[47] Neo4j. 2020. Neo4j Community Edition. https://neo4j.com/download-center/#community. Last Accessed July 25, 2021.

[48] Neo4j. 2020. Neo4j Property Graph Model. https://neo4j.com/developer/graph-database. Last Accessed July 25, 2021.

[49] Thomas Neumann and Guido Moerkotte. 2011. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*. 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[50] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal* 19, 1 (2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y

[51] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Record* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[52] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[53] Oracle. 2020. Oracle In-Memory Column Store Architecture. https://tinyurl.com/vkvb6p6. Last Accessed July 25, 2021.

[54] Oracle. 2020. Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html. Last Accessed July 25, 2021.

[55] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS) (LNI)*, Vol. P-214. 403–420. https://dl.gi.de/20.500.12116/17334

[56] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDB Journal* 29, 2-3 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[57] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2019. C-store: A Column-Oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518. https://doi.org/10.1145/3226595.3226638

[58] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD 2020*. 345–359. https://doi.org/10.1145/3318464.3386138

[59] TigerGraph. 2020. TigerGraphDB. https://www.tigergraph.com. Last Accessed July 25, 2021.

[60] Syunsuke Uemura, Toshitsugu Yuba, Akio Kokubu, Ryoichi Ooomote, and Yasuo Sugawara. 1980. The Design and Implementaion of a Magnetic-Bubble Database Machine. In *Information Processing, Proceedings of the 8th IFIP Congress 1980*. 433–438.

[61] Vertica. 2020. Vertica 10.0.x Documentation. https://www.vertica.com/docs/10.0.x/HTML/Content/Home.html. Last Accessed July 25, 2021.

[62] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1008–1019. http://www.vldb.org/pvldb/vol1/1453965.pdf

[63] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Record* 29, 3 (2000), 55–67. https://doi.org/10.1145/362084.362137

[64] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020), 5:1–5:31. https://doi.org/10.1145/3375660

[65] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034. http://www.vldb.org/pvldb/vol13/p1020-zhu.pdf

[66] Marcin Zukowski and Peter A. Boncz. 2012. From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012*. 861–862. https://doi.org/10.1145/2213836.2213967

[67] Marcin Zukowski and Peter A. Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27. http://sites.computer.org/debull/A12mar/vectorwise.pdf

[68] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*. 59. https://doi.org/10.1109/ICDE.2006.150

Amine Mhedhbi
Phone: +1(438)-728-4635
Email: amine.mhedhbi@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

November 18, 2022

Courant Institute of Mathematical Sciences
New York University

Dear Members of the Search Committee:

I am writing in response to the call for a tenure-track faculty position in Computer Science at the Courant Institute of Mathematical Sciences, New York University. I am currently a Ph.D. student at the Cheriton School of Computer Science at the University of Waterloo. I expect to complete all requirements for my doctorate degree by May 2023.

The overarching goal of my research is to build data systems capable of efficient analytical processing of graph data at scale. My Ph.D. thesis focused on graph analytical workloads, which involve relational queries containing complex many-to-many joins that lead to an explosion in the size of intermediate relations. Traditional analytical DBMSs, were not optimized for complex many-to-many joins. My thesis argues that DBMSs that optimize for *graph analytical workloads* need to integrate a suite of modern techniques, most important of which are: (i) novel *worst-case-optimal join algorithms*; (ii) *factorized query processing*; and (iii) a flexible indexing sub-system with materialized views support. I do systems-oriented research and have integrated each of these techniques into a modern DBMS called GraphflowDB. Such an integration requires tackling algorithmic and system research questions that necessitate rethinking and introducing novel techniques to core DBMS components such as the query executor, the query optimizer, and the storage layer. In the future, I aim to research building data systems for machine learning on graphs while also continuing research in the topics of my Ph.D. I have completed two internships at Microsoft Research leading to two on-going collaborations, both of which expanded my research interests into transactional processing and cloud infrastructure.

Further details of my research and teaching experience can be found in my research statement, teaching statement, and curriculum vitae (submitted as part of my application). Details of my views on diversity can be found in my philosophy on diversity statement. My publications can be found on my personal webpage and Google Scholar profile. I have asked Prof. Semih Salihoglu (my Ph.D. advisor), Prof. M. Tamer Özsu at the University of Waterloo, and Dr. Phil Bernstein at Microsoft Research to provide reference letters.

Thank you for your time and consideration. Please do not hesitate to contact me if any further information is required in support of my application.

Sincerely,

Amine Mhedhbi

Application Forms

Affiliation Information

Do you currently, or have you had, a previous affiliation with NYU as a student, employee, or other appointment type?

No

Is any member of your family or household an employee, trustee/director, or officer of NYU or any NYU affiliate, or are you in a relationship otherwise covered under the policy listed below?

No

Computer Science Research Areas - Primary Interest (required, select one)

Computer Science Research Areas - Primary Interest (select one)

Databases

Computer Science Research Areas - Secondary Interest (required, select one)

Computer Science Research Areas - Secondary Interest (required, select one)

Systems