# 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:

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

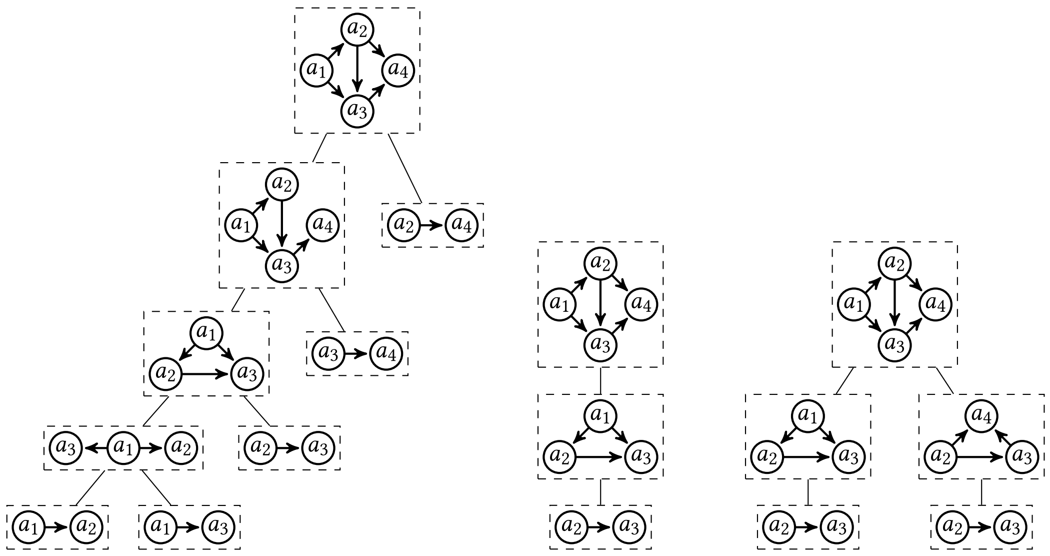
We study evaluating a general class of subgraph queries where $V_Q$ and $E_Q$ can have labels. For labelled queries, the edge table corresponding to the query edge $a_i \rightarrow a_j$ contains only the edges in $G$ that are consistent with the labels on $a_i$, $a_j$, and $a_i \rightarrow a_j$. Subgraph 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* $\sigma$ of $V_Q$ and matches $Q$ one query vertex at a time according to $\sigma$. 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 suboptimality. 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.

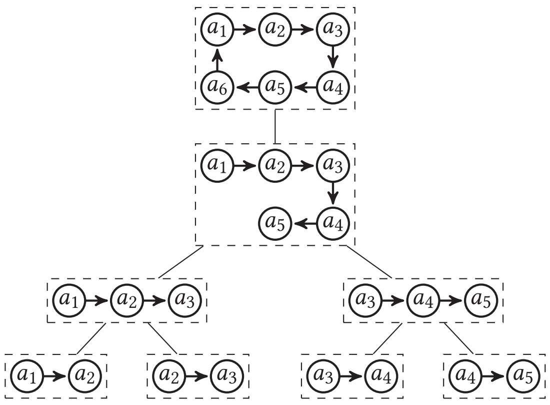


Fig. 2. Example plan not in EmptyHeaded's GHD-based plan space. Top box is the actual query.

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 $\sigma^*$ 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 $\sigma_j$ that is more efficient than $\sigma^*$. Our adaptive executor re-evaluates the cost of each $\sigma_j$ 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 $E_\delta$ 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 $\delta$, $o$, or $n$, indicating, upon an update to $G$, whether the edge should match an edge

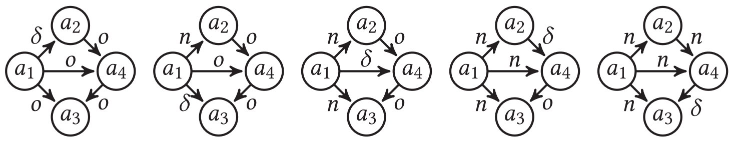


Fig. 3. Example delta subgraph queries for the diamond-X continuous subgraph query.

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 $E_\delta$, $E_o$, 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 $\sigma$ 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 $\sigma$ 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 $\sigma$: $a_1 \ldots a_k$. 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 $E_\delta$ be a set of updates, $E_o$ be the `old` edges in $G$ before $E_\delta$, and $E_n$ the `new` edges, i.e., $E_n = E_o \cup E_\delta$. We assume added and deleted edges in $E_\delta$ are identified by +/− labels, respectively. Emerged and deleted outputs are identified similarly. We will use $a_i \xrightarrow{\delta/o/n} a_j$ notation to refer to the query edges that should match edges in $E_\delta$, $E_o$, 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 $Q$:

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

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

$$\cdots$$

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

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 $\delta$ 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 $\sigma$ of $Q$ is effectively a different WCO plan for $Q$. Figure 1(b) shows an example $\sigma$, 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 $\sigma$. 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 $\ell$ many $k$-matches, all of these $k$ matches are identical on the first $k$-1 query vertices of $\sigma$ (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 $v_2$ $\ell$ consecutive times. Instead, $o_4$ can compute this intersection for $(v_1, v_2, s_1)$ once and reuse it for the following $\ell$−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 $\ell$ 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 $\ell$ 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

| | $\sigma_1$ | $\sigma_2$ | $\sigma_3$ | $\sigma_4$ | $\sigma_5$ | $\sigma_6$ | $\sigma_7$ | $\sigma_8$ |
|---|---|---|---|---|---|---|---|---|
| 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 $\sigma$: (1) directions of the adjacency lists $\sigma$ intersects, (2) the amount of intermediate partial matches $\sigma$ generates, and (3) how much $\sigma$ utilizes the intersection cache. We discuss each effect next.

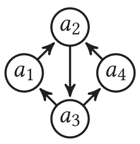
**Directions of Intersected Adjacency Lists:** Perhaps surprisingly, there are WCO plans that have very different runtimes *only because* they compute their intersections using different directions of the adjacency lists. The simplest example of this is the asymmetric triangle query $a_1{\rightarrow}a_2$, $a_2{\rightarrow}a_3$, $a_1{\rightarrow}a_3$. 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):

- $\sigma_1$:$a_1a_2a_3$: intersects both $u$ and $v$'s forward lists.
- $\sigma_2$:$a_2a_3a_1$: intersects both $u$ and $v$'s backward lists.
- $\sigma_3$:$a_1a_3a_2$: intersects $u$'s forward and $v$'s backward lists.

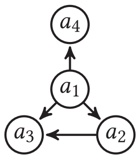
Table 5 shows a demonstrative experiment studying the performance of each plan on the BerkStan and LiveJournal graphs (the i-cost column in the table will be discussed in Section 3.1.3 momentarily). For example, $\sigma_1$ is 12.1× faster than $\sigma_2$ on the BerkStan graph. Which combination of adjacency 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 $u \to v$ 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 $\sigma$ 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 $\sigma$ 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 $\sigma$ 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 $\sigma$. 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.

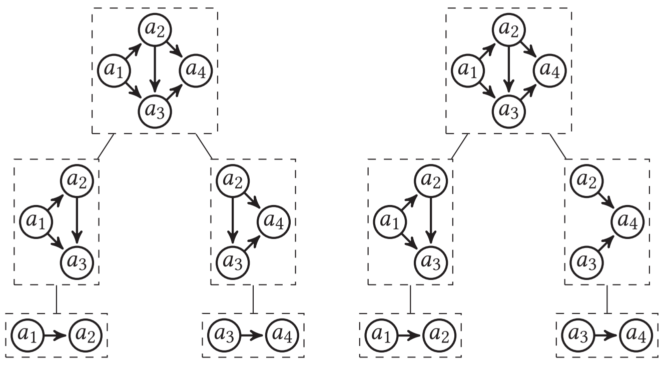
- Leaf nodes are labeled with a single query edge of $Q$.
- Root is labeled with $Q$.
- Each internal node $o_k$ is labeled with $Q_k = \{V_k, E_k\}$, with the *projection constraint* that $Q_k$ is a projection of $Q$ onto a subset of query vertices. $o_k$ has either one child or two children. If $o_k$ 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

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

$A$ is a set of adjacency list descriptors; $\mu$ 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 $\sigma_2$ ($a_2a_3a_4a_1$). Instead, if we ignore the cache and make our optimizer "cache-oblivious" by always estimating i-cost with Equation (2), it picks the slower $\sigma_4$ ($a_1a_2a_3a_4$) 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.

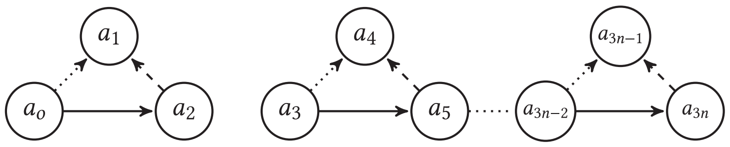


Fig. 6. Input graph for adaptive QVO example.

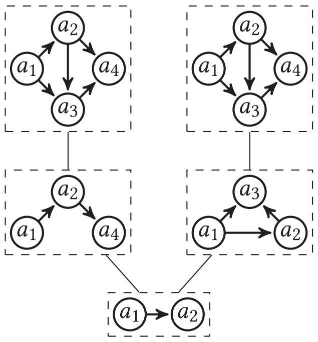


Fig. 7. Example adaptive WCO plan.

## 3.5 Adaptive WCO Plan Evaluation

Recall that the $|A|$ and $\mu$ 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 $\sigma^*$ and extends partial matches of a sub-query $Q_i$ to matches of $Q_k$. Our optimizer picks $\sigma^*$ 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 $\sigma^*$ 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 $\sigma$ for the dotted and dashed edges as before but $\sigma' = a_2a_3a_1a_4$ for the solid edges. For the solid edges, $\sigma'$ 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 $\sigma_j$ that can extend $Q_i$ to $Q_k$, $f$ re-evaluates the estimated i-cost of $\sigma_j$ by re-calculating the cost of plans using *updated cost estimates* (explained momentarily). $o_i$ gives $t$ to the next E/I operator of $\sigma_j^*$ that has the lowest re-calculated cost. The cost of $\sigma_j$ 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 $\mu_j$ 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 $\mu_j$ to $10 \times (50/100) \times 200/2000 = 0.5$.

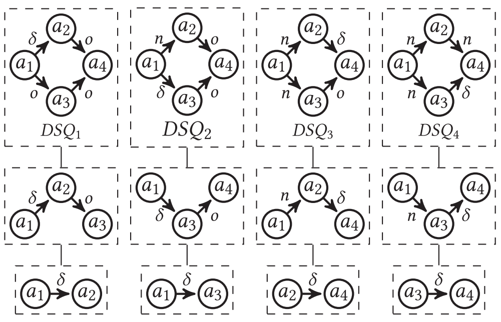
As we show in our evaluations, adaptive QVO selection improves the performance of many WCO plans but more importantly guards our optimizer from picking bad QVOs.

## 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 $E_{\delta_1}$, $E_{\delta_2}$\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.

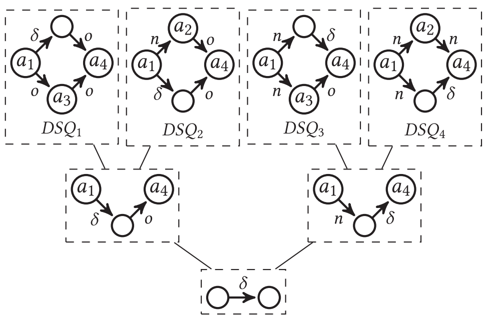

(b) Combined plan of the individual plans.

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 $E_\delta$, there are three types of adjacency lists in memory for each vertex $v$ in $G$:

- `delta` contains $v$'s neighbours in $E_\delta$.
- `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 $\delta$ 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 $\delta$, $n$, or $o$ to indicate whether they match $E_\delta$, $E_n$, 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 $\delta$ 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 $E_\delta$, 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 $E_\delta$).

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 $\delta$ 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 $\delta$, $n$, 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 $\bar{Q}_{DSQ}$.

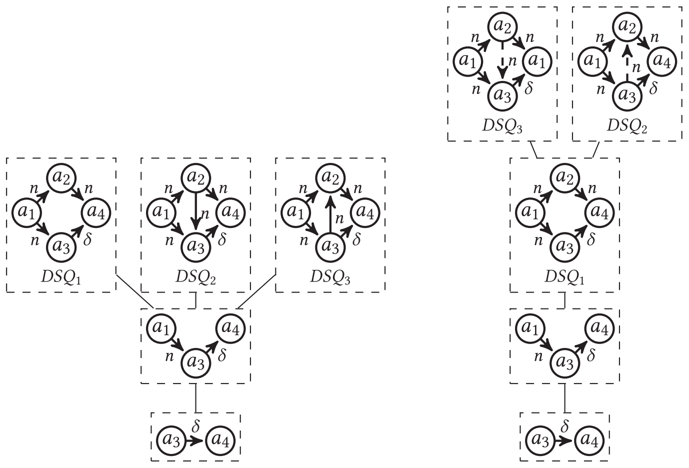
We do not establish the hardness of this formal problem in this article and leave this to future work. However, in our supplementary Appendix A.1, we show that the natural decision version of a generalized version of this problem, in which we assume that $\bar{Q}_{DSQ}$ can contain arbitrary DSQs and 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].

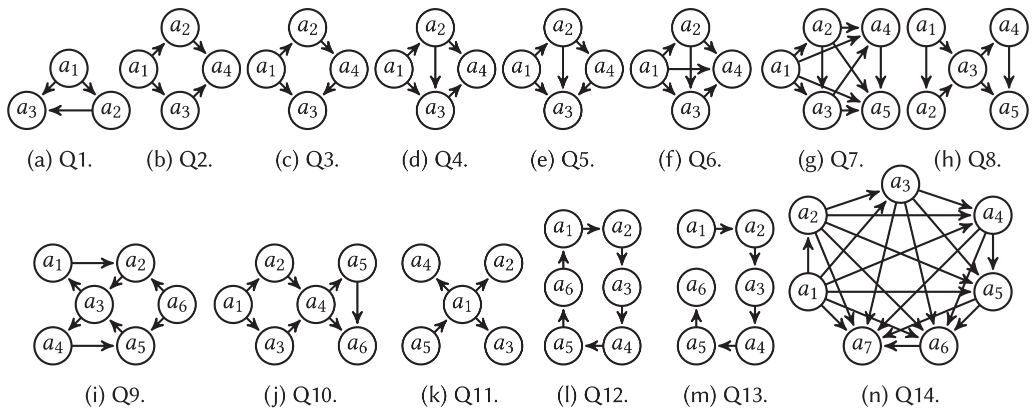


Fig. 10. Subgraph queries used for evaluations.

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 |

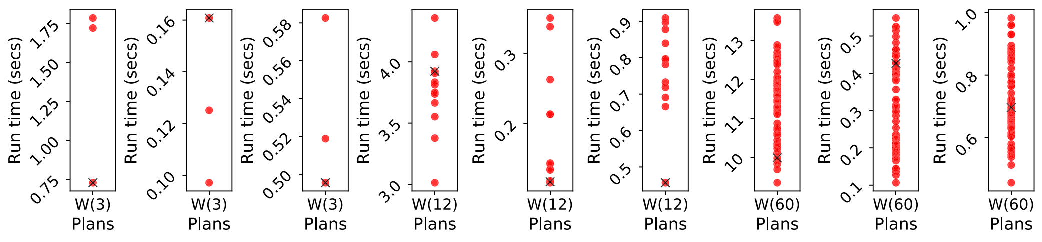
*6.2.1 Plan Suitability For Different Queries and Optimizer Evaluation.* To evaluate how good are the plans our optimizer generates, we compare the runtime of plans we pick against a query's plan spectrum, i.e., the set of all plans enumerated by GraphflowDB for the query. This also allows us to study 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 $Q8$.

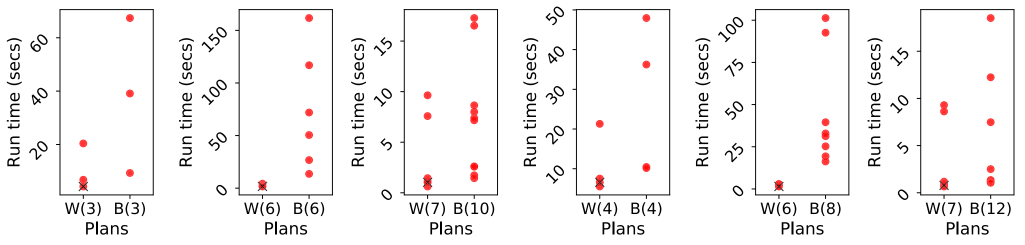
Our most interesting query is $Q12$, which is a 6-cycle query. $Q12$ can be evaluated efficiently with both WCO and hybrid plans (and reasonably well with some BJ plans). The hybrid plans first perform binary joins to compute 4-paths, and then extend 4-paths into 6-cycles with an intersection. Figure 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).

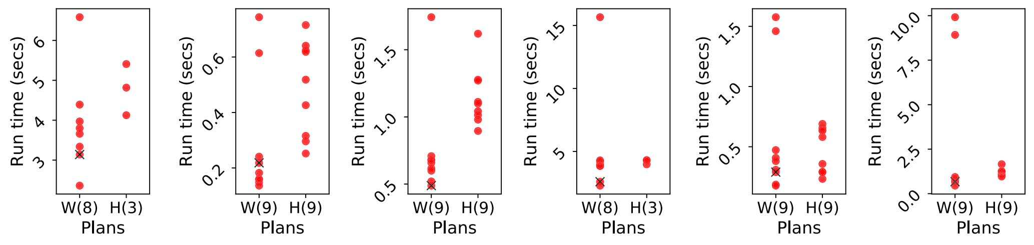
Although we do not study query optimization time in this article, our optimizer generated a plan within 331 ms in all of our experiments except for $Q7_5$ on Google, which took 1.4 s.



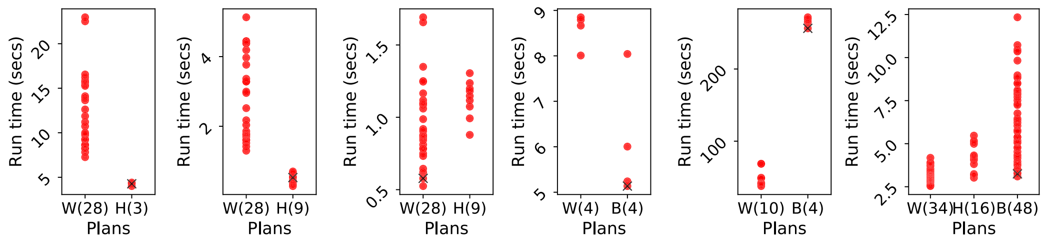
(a) Q1,Am. (b) Q1$_3$,Ep. (c) Q1$_5$,Go. (d) Q6,Am. (e) Q6$_3$,Ep. (f) Q6$_5$,Go. (g) Q7,Am. (h) Q7$_3$,Ep. (i) Q7$_5$,Go.



(j) Q2, Am. (k) Q2$_3$, Ep. (l) Q2$_5$, Go. (m) Q3, Am. (n) Q3$_3$, Ep. (o) Q3$_5$, Go.



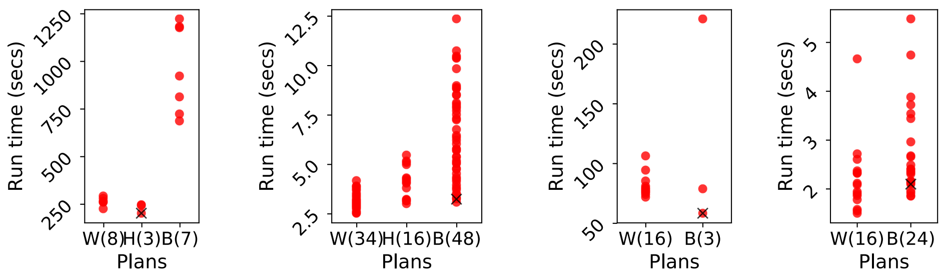
(p) Q4,Am. (q) Q4$_3$, Ep. (r) Q4$_5$, Go. (s) Q5, Am. (t) Q5$_3$, Ep. (u) Q5$_5$, Go.



(v) Q8, Am. (w) Q8$_3$, Ep. (x) Q8$_5$, Go. (y) Q11, Am. (z) Q11$_3$, Ep. (aa) Q11$_5$, Go.

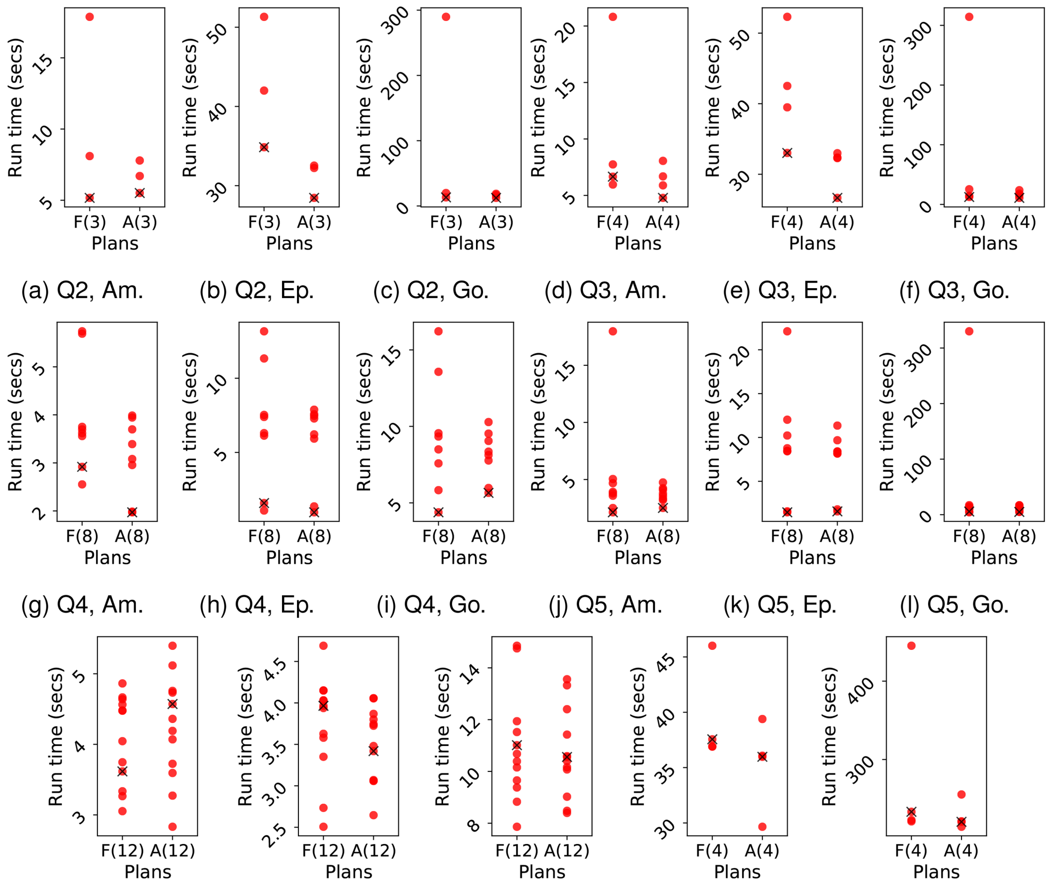
Fig. 11. Runtime (seconds) of the set of all plans enumerated by GraphflowDB for queries Q1–Q8 and Q11. "x" specifies the plan picked by GraphflowDB.

*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.

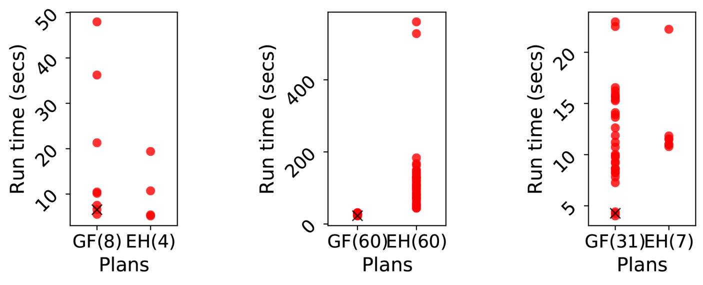
hybrid plans are similar to $Q3$ and $Q4$'s. Each hybrid plan of $Q10$ computes the diamonds on the left and triangles on the right and joins on $a_4$. Here, we can adaptively compute the diamonds (but not the triangles). Each fixed hybrid plan improves by adapting and some improve by up to 2.1×. On $Q5$ 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 ($\mathsf{EH}_b$) and "good" orderings ($\mathsf{EH}_g$). 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 ($\text{EH}_g$) and Bad Orderings ($\text{EH}_b$)

| | | Q1 | Q3 | $\text{Q3}_2$ | Q5 | $\text{Q5}_2$ | Q7 | $\text{Q7}_2$ | Q8 | $\text{Q8}_2$ | Q9 | $\text{Q9}_2$ | Q12 | $\text{Q12}_2$ | Q13 | $\text{Q13}_2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Am | $\text{EH}_b$ | 1.0 | 19.0 | 3.4 | 47.1 | 9.2 | 91.4 | 11.6 | 22.2 | 1.8 | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* |
| | $\text{EH}_g$ | 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 | $\text{EH}_b$ | 1.9 | 444.5 | 42.6 | 401.1 | 77.6 | 1.04K | 23.4 | 66.6 | 16.0 | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* |
| | $\text{EH}_g$ | 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 | $\text{EH}_b$ | 0.5 | 42.7 | 6.5 | 64.5 | 11.4 | 560.7 | 2.9 | 1.01K | 22.0 | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* | *Mm* |
| | $\text{EH}_g$ | 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.

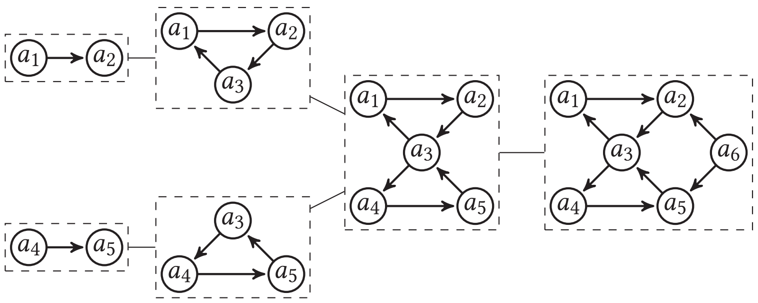
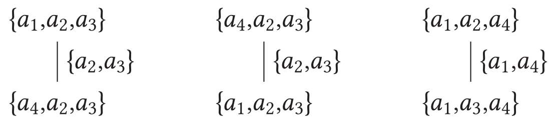


Fig. 15. Plan (drawn horizontally) with seamless mixing of intersections and binary joins on Q9.

faster than $\text{EH}_b$, 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 $\text{EH}_b$ on $Q3$ Google, it is 1.2× slower than $\text{EH}_g$. Importantly $\text{EH}_g$ is always faster than $\text{EH}_b$, 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) $\text{TD}_{2_1}$ for Q2. (b) $\text{TD}_{2_2}$ for Q2. (c) $\text{TD}_{2_3}$ 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 $\text{TD}_{1_2}$. A preorder traversal on $\text{TD}_{1_2}$ 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 $\text{TD}_{1_2}$ 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:

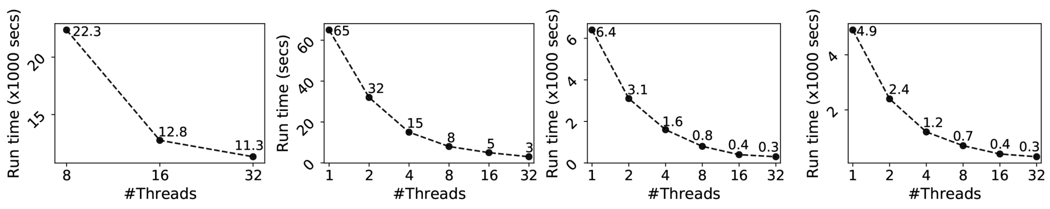
1. On some queries, the heuristics that CTJ uses to pick a TD cannot distinguish between efficient TDs from inefficient ones. For example, consider the diamond query Q2 from Figure 10(b). CTJ's heuristics will not be able to tie break between $\text{TD}_{2_1}$, $\text{TD}_{2_2}$, and $\text{TD}_{2_3}$ 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, $\text{TD}_{2_1}$ and $\text{TD}_{2_2}$ 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 $\text{TD}_{2_1}$ and 1.38B for $\text{TD}_{2_2}$. 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 $\mathsf{E}$ 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

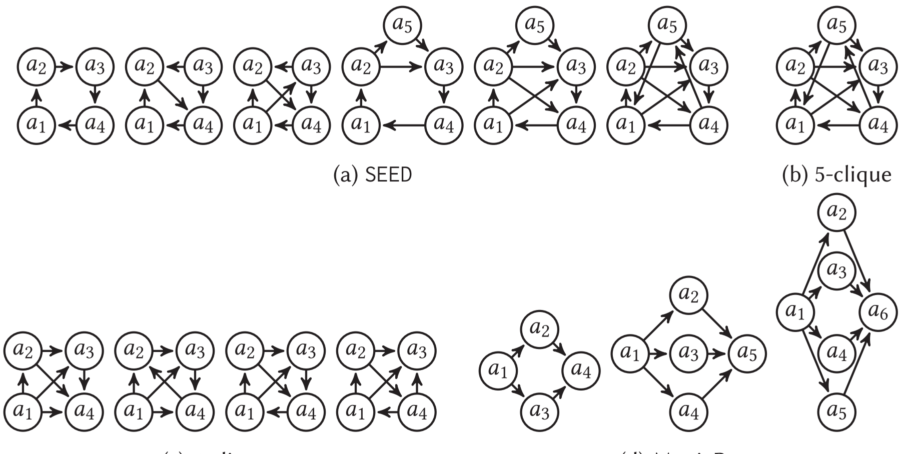


Fig. 18. Queries used for continuous subgraph query evaluations.

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:

- $B_{ns}$ (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 $B_{ns}$, $B_s$, $G_r$, 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$_3$** | |
| **Am** | $B_{ns}$ | 6.29 | 0.602B (**65%**) | 1.33 | 0.094B (**48%**) | 0.72 | 0.033B (**32%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 8.61 | 0.858B (**29%**) | 3.28 | 0.178B (**12%**) | 2.12 | 0.076B (**7%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 13.4 | 1.226B (**35%**) | 1.87 | 0.129B (**6%**) | 0.90 | 0.043B (**1%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 9.73 | 1.015B (**0%**) | 4.50 | 0.217B (**0%**) | 2.84 | 0.094B (**0%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 28.8 | 2.675B (**59%**) | 3.60 | 0.201B (**23%**) | 1.45 | 0.060B (**10%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 15.5 | 1.343B (**6%**) | 7.02 | 0.261B (**4%**) | 4.01 | 0.101B (**0.3%**) |
| | $B_s$ | 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**) |
| | $Gr_p$ | 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** | $B_{ns}$ | 36.6 | 5.238B (**18%**) | 5.41 | 0.659B (**20%**) | 2.03 | 0.180B (**17%**) |
| | $B_s$ | 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** | $B_{ns}$ | 87.6 | 11.16B (**16%**) | 11.7 | 1.406B (**14%**) | 6.88 | 0.435B (**13%**) |
| | $B_s$ | 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 $B_{ns}$ 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 $B_{ns}$.

- $B_s$ (for **s**haring): Puts the plans of $B_{ns}$ into a combined plan.
- Gr: The combined plan generated by our greedy optimizer.
- $Gr_p$ (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 $B_{ns}$, $B_s$, Gr, and $Gr_p$. We show the i-cost numbers to explain an important pattern we discuss momentarily. The numbers in the parantheses next to $B_s$, Gr, and $Gr_p$ report the relative performance improvements of Gr and $Gr_p$ over $B_{ns}$. We explain the percentage value next to the i-cost value of $B_{ns}$ 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.

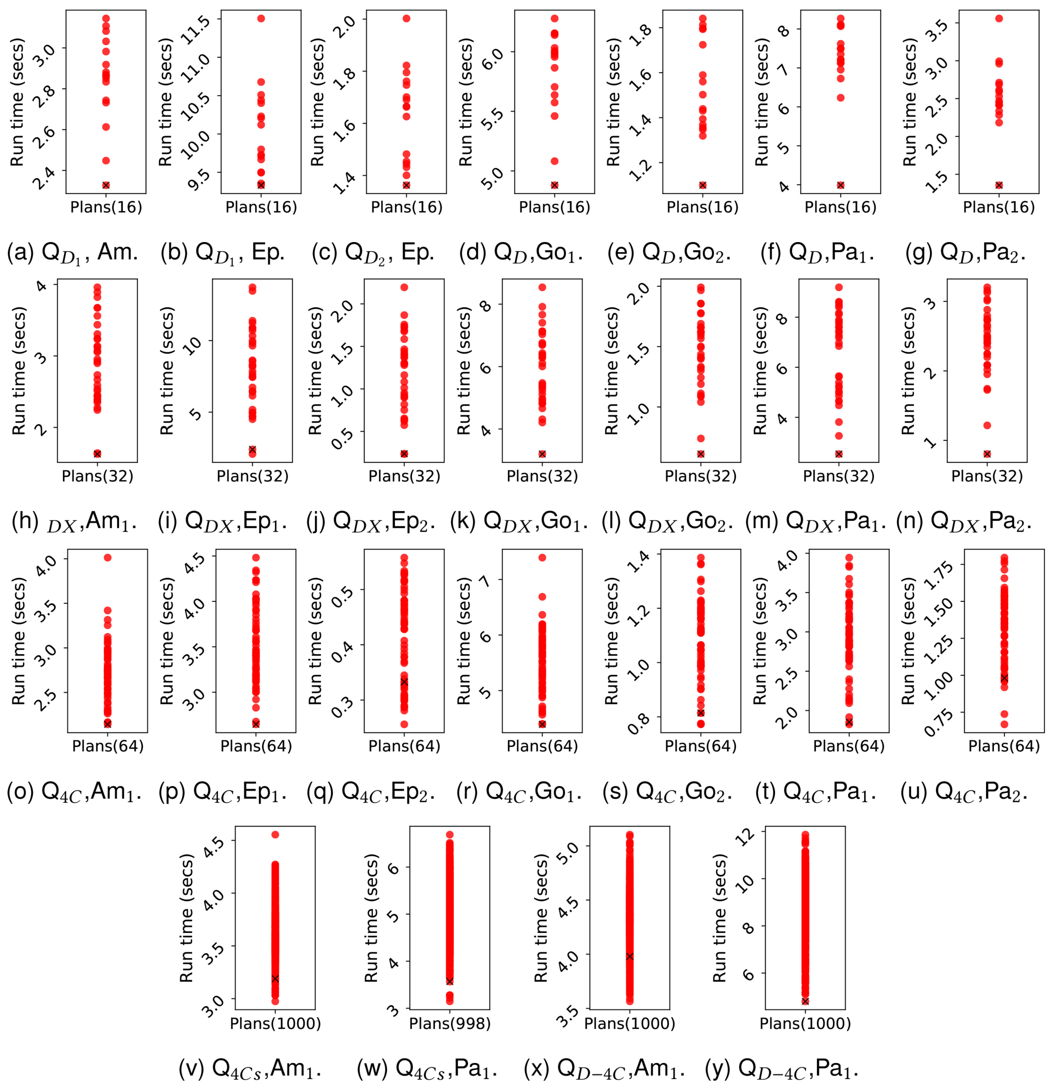


Fig. 19. Runtime (seconds) of a sample of the plans enumerated by the continuous query optimizer for Diamond ($Q_D$), Diamond-X ($Q_{DX}$), and 4-Clique ($Q_{4C}$) and two query sets, one of two 4-Cliques ($Q_{4Cs}$) and the other of a Diamond and a 4-Clique ($Q_{D-4C}$) on datasets Am, Ep, Go, and Pa 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 $i$th 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 ($\mu$) 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. EmptyHeaded: 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 $\delta$ 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 $\mu$ 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 $a_1, a_2, \dots, a_{m_1}$ and each node in $G_2$ is labeled with $b_1, b_2, \dots, b_{m_2}$. 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 $\delta$ and connect

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 $\tau$, we show the number of queries that a particular catalogue estimated with q-error at most $\tau$. 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 $\tau$ values when the larger h or z values lead to very minor decreases in the number of queries within the $\tau$ 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 $\leq 3$) 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 $\mathsf{B}_{ns}$, $\mathsf{B}_s$, Gr, and $\mathsf{G}_{opis}$ on 4Cs, 4Cs5C, SEED, and MagicRec Query Sets

| | | **Runtime** | **I-cost** | **Runtime** | **I-cost** | **Runtime** | **I-cost** |
|---|---|---|---|---|---|---|---|
| | | **4Cs$_1$** | | **4Cs$_2$** | | **4Cs$_3$** | |
| | $\mathrm{B}_{ns}$ | 5.51 | 1.488B (**84%**) | 0.56 | 0.163B (**65%**) | 0.33 | 0.052B (**50%**) |
| **Ep** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 4.78 (**1.15x**) | 1.244B (**1.20x**) | 0.49 (**1.14x**) | 0.126B (**1.30x**) | 0.18 (**1.83x**) | 0.030B (**1.73x**) |
| | $\mathrm{B}_{ns}$ | 14.61 | 3.032B (**58%**) | 2.70 | 0.515B (**38%**) | 1.31 | 0.188B (**39%**) |
| **Go** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 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$_3$** | |
| | $\mathrm{B}_{ns}$ | 16.71 | 5.607B (**64%**) | 0.97 | 0.278B (**22%**) | 0.42 | 0.074B (**7%**) |
| **Ep** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 15.80 (**1.06x**) | 5.235B (**1.07x**) | 0.66 (**1.47x**) | 0.209B (**1.33x**) | 0.27 (**1.56x**) | 0.041B (**1.80x**) |
| | $\mathrm{B}_{ns}$ | 33.96 | 5.422B (**36%**) | 3.54 | 0.607B (**6%**) | 1.63 | 0.211B (**1%**) |
| **Go** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 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$_3$** | |
| | $\mathrm{B}_{ns}$ | 205.9 | 47.76B (**86%**) | 8.45 | 2.071B (**61%**) | 2.43 | 0.430B (**43%**) |
| **Ep** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 192.8 (**1.07x**) | 44.75B (**1.07x**) | 7.45 (**1.13x**) | 1.691B (**1.22x**) | 1.81 (**1.34x**) | 0.311B (**1.38x**) |
| | $\mathrm{B}_{ns}$ | 97.9 | 13.04B (**74%**) | 7.25 | 0.715B (**34%**) | 3.31 | 0.208B (**28%**) |
| **Go** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{Gr}_p$ | 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$_3$ | |
| | $\mathrm{B}_{ns}$ | 1524 | 40.64B (**19%**) | 40.2 | 9.999B (**28%**) | 7.66 | 2.048B (**24%**) |
| **Ep** | $\mathrm{B}_s$ | 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**) |
| | $\mathrm{B}_{ns}$ | 475.3 | 43.77B (**20%**) | 18.2 | 7.345B (**26%**) | 5.48 | 1.600B (**23%**) |
| **Go** | $\mathrm{B}_s$ | 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 $\mathsf{B}_{ns}$ 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 $\mathsf{B}_{ns}$.