Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins

Amine Mhedhbi
University of Waterloo
amine.mhedhbi@uwaterloo.ca

Semih Salihoglu
University of Waterloo
semih.salihoglu@uwaterloo.ca

ABSTRACT

We study the problem of optimizing 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 design a cost-based optimizer 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. Our cost metric combines the cost of binary joins with a new cost metric called *intersection-cost*. The plan space of our optimizer contains plans that are not in the plan spaces based on tree decompositions from prior work. In addition to our optimizer, we describe an *adaptive technique* that changes the orderings of the worst-case optimal sub-plans during query execution. We demonstrate the effectiveness of the plans our optimizer picks and adaptive technique through extensive experiments. Our optimizer is integrated into the Graphflow DBMS.

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. Subgraph queries appear in many applications where graph patterns reveal valuable information. For example, Twitter searches for diamonds in their follower network for recommendations [17], clique-like structures in social networks indicate communities [29], and cyclic patterns in transaction networks indicate fraudulent activity [10, 26].

As observed in prior work [3, 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:

$$Q_{DX} = E_1 \bowtie E_2 \bowtie E_3 \bowtie E_4 \bowtie E_5$$

where $E_1(a_1, a_2)$, $E_2(a_1, a_3)$, $E_3(a_2, a_3)$, $E_4(a_2, a_4)$, and $E_5(a_3, a_4)$ 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 labeled queries, the edge table corresponding to the query edge $a_i \rightarrow a_j$ contains only the edges in G that are consistent with the labels on a_i, a_j, and $a_i \rightarrow a_j$. Subgraph queries are evaluated with two main approaches:

- *Query-edge(s)-at-a-time* approach executes a sequence of binary joins to evaluate Q. Each binary join effectively matches a larger subset of the query edges of Q in G until Q is matched.
- *Query-vertex-at-a-time* approach picks a *query vertex ordering* σ of V_Q and matches Q one query vertex at a time according to σ, using a multiway join operator that performs multiway intersections. This is the computation performed by the recent worst-case optimal join algorithms [30, 31, 40]. 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 both operations as *hybrid* plans. Figures 1 a , 1 b , and 1 c show an example of each plan for the diamond-X query.

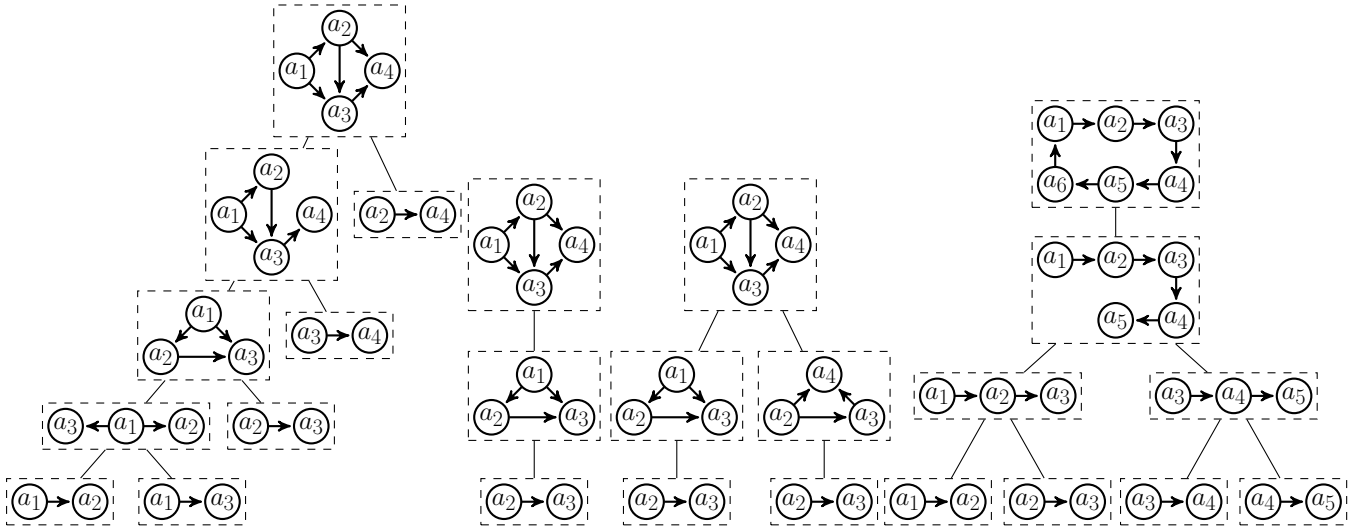
Recent theoretical results [8, 31] showed that BJ plans can be suboptimal on cyclic queries and have asymptotically worse runtimes than the worst-case (i.e., maximum) output sizes of these queries. This worst-case output size is now known as a query's *AGM bound*. These results also showed that WCO plans correct for this sub-optimality. However, this theory has two shortcomings. First, the theory gives no advice as to *how to pick a good query vertex ordering* for WCO plans. Second, the theory ignores 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 [3, 23].

We study how to generate efficient plans for subgraph queries using a mix of worst-case optimal-style multiway intersections and binary joins. We describe a cost-based optimizer we developed for the Graphflow DBMS [18] that generates BJ plans, WCO plans, as well as hybrid plans. Our cost metric for WCO plans capture the various runtime effects of query vertex orderings we have identified. Our 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 [25], as well as those that are developed on top of RDBMSs, such as SAP's graph database [36], are often cost-based. As such, our work gives insights into how to integrate the new worst-case optimal join algorithms into the cost-based optimizers of existing systems.

1.1 Existing Approaches

Perhaps the most common approach adopted by graph databases (e.g. Neo4j), RDBMSs, and RDF systems [28, 42], is to evaluate subgraph queries with BJ plans. As observed in prior work [30], BJ plans are inefficient in highly-cyclic queries, such as cliques. Several prior solutions, such as BiGJoin [6], our prior work on Graphflow [18], and the LogicBlox system have studied evaluating queries with only WCO plans, which, as we demonstrate in this paper, are not efficient for acyclic and sparsely cyclic queries. In addition, these solutions either use simple heuristics to select query vertex orderings or arbitrarily select them.

The EmptyHeaded system [3], which is the closest to our work, is the only system we are aware of that mixes worst-case optimal



(a) BJ plan. (b) WCO plan. (c) Hybrid plan. (d) Non-GHD Hybrid plan.

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

joins with binary joins. EmptyHeaded 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, 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.

1.2 Our Contributions

Table 1 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-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 which are not in EmptyHeaded's plan space. Figure 1 d 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.

	Q. Vertex Ordering	**Binary Joins**
BiGJoin	Arbitrarily	No
LogicBlox	Heuristics or Cost-based[1]	No
EH	Arbitrarily	Cost-based: depends on Q
Graphflow	Cost-based & Adaptive	Cost-based: depends on Q and G

Table 1: Comparisons against solutions using worst-case optimal joins. EH stands for EmptyHeaded.

Our second main 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 extend matches of sub-query Q_i into a larger sub-query Q_k. Suppose there are r possible query vertex orderings, σ_1, ..., σ_r, to perform these extensions. Our optimizer tries to pick the ordering σ^* with the lowest cumulative i-cost when extending all partial matches of Q_i in G. However, for any specific match t of Q_i, there may be another σ_j that is more efficient than σ^*. Our adaptive executor re-evaluates the cost of each σ_j for t based on the actual sizes of the adjacency lists of the vertices in t, and picks a new ordering for t.

We incorporate our optimizer into Graphflow [18] 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

[1]LogicBlox is not open-source. Two publications describe how the system picks query vertex orderings; a heuristics-based [32] and a cost-based [7] technique (using sampling).

Abbrv.	Explanation	Abbrv.	Explanation
BJ	Binary Join	GHD	Generalized Hypertree Decompositions
EH	EmptyHeaded	QVO	Query Vertex Ordering
E/I	Extend/Intersect	WCO	Worst-case Optimal

Table 2: Abbreviations used throughout the paper.

not in EmptyHeaded's plan space, are up to 68x more efficient than EmptyHeaded's plans. We show that adaptively picking query vertex orderings improves the runtime of some plans by up to 4.3x, in some queries improving the runtime of every plan and makes our optimizer more robust against picking bad orderings. For completeness, in Appendix C we include comparisons against Neo4j and another subgraph matching algorithm called CFL [9]. Both of these baselines were not as performant as our plans in our setting.

Table 2 summarizes the abbreviations used throughout the paper.

2. PRELIMINARIES

We assume a subgraph query $Q(V_Q, E_Q)$ is directed, connected, and has m query vertices $a_1, ..., a_m$ 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 $G(V, E)$. Unlabeled queries can be thought of as labeled queries on a version of G with a single edge and single vertex label. The outgoing and incoming neighbors of each $v \in V$ are indexed in forward and backward adjacency lists. We assume the adjacency lists are partitioned first by the edge labels and then by the labels of neighbor vertices. This allows, for example, detecting a vertex v's forward edges with a particular edge label in constant time. The neighbors in a partition are ordered by their IDs, which allow fast intersections.

Generic Join [30] is a WCO join algorithm that evaluates queries one attribute at a time. We describe the algorithm in graph terms; reference [30] gives an equivalent relational description. In graph terms, the algorithm evaluates queries one query vertex at a time with two main steps:

- **Query Vertex Ordering (QVO):** Generic Join first picks an order σ of query vertices to match. For simplicity we assume $\sigma = a_1...a_m$ and the projection of Q onto any prefix of k query vertices for $k = 1, ..., m$ is connected.
- **Iterative Partial Match Extensions:** Let $Q_k = \Pi_{a_1,...,a_k} Q$ be a sub-query that consists of Q's projection on the first k query vertices $a_1...a_k$. Generic Join iteratively computes $Q_1, ..., Q_m$. Let *partial k-match* (k-match for short) t be any set of vertices of V assigned to the first k query vertices in Q. For $i \leq k$, let $t[i]$ be the vertex matching a_i in t. To compute Q_k, Generic Join extends each (k–1)-match t' in the result of Q_{k-1} to a possibly empty set of k-matches by intersecting 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 $i \leq k-1$. Let the result of this intersection be the *extension set* S of t. The k-matches t produces is the Cartesian product of t with S.

3. OPTIMIZING WCO PLANS

This section demonstrates our WCO plans, the effects of different QVOs we have identified, and our i-cost metric for WCO plans.

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

Table 3: Experiment on intersection cache utility for diamond-X.

Throughout this section we present several experiments on unlabeled queries for demonstration purposes. The datasets we use in these experiments are described in Table 8 in Section 8.

3.1 WCO Plans and E/I Operator

Each query vertex ordering σ of Q is effectively a different WCO plan for Q. Figure 1 b shows an example σ, which we represent as a chain of m–1 nodes, where the $(k$–$1)$'th node from the bottom contains a sub-query Q_k which is the projection of Q onto the first k query vertices of σ. We use two 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 labeled $Q_k(V_k, E_k)$ that have a child labeled $Q_{k-1}(V_{k-1}, E_{k-1})$ are evaluated with an E/I operator. The E/I operator takes as input (k–1)-matches and extends each tuple t to one or more 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. 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 of t by intersecting the adjacency lists described by its descriptors, ensuring they match the specified edge and destination vertex labels, and then extends t to $t \times S$. When there is a single descriptor, S is the list described by the descriptor. Otherwise we use iterative 2-way in-tandem intersections.

Multiple (k–1)-matches that are processed consecutively in an E/I operator may require the same extension set if they perform the same intersections. Our E/I operator caches and reuses the last extension set S in such cases. We store the cached set in a flat array buffer. The intersection cache overall improves the performance of WCO plans. As a demonstrative example, Table 3 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 run time, one by 1.9x.

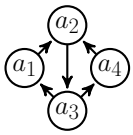
3.2 Effects of QVOs

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

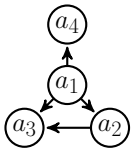
3.2.1 Directions of Intersected Adjacency Lists

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 3 QVOs, all of which have the same SCAN operator, which scans each $u \rightarrow v$ edge in G, followed by 3 different intersections (without utilizing the intersection cache):

- σ_1:$a_1a_2a_3$: intersects both u and v's forward lists.



(a) Diamond-X with symmetric triangle.



(b) Tailed triangle.

Figure 2: Queries used to demonstrate the effects of QVOs.

		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

Table 4: Runtime (secs), intermediate partial matches (part. m.), and i-cost of different QVOs for the asymmetric triangle query.

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

Table 4 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 3.3 momentarily). For example, σ_1 is 12.1x faster than σ_2 on the BerkStan graph. Which combination of adjacency list directions is more efficient depends on the structural properties of the input graph, e.g., forward and backward adjacency list distributions.

3.2.2 *Number of Intermediate Partial Matches*

Different WCO plans generate different partial matches leading to different amount of intersection work. Consider the *tailed triangle* query in Figure 2 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.

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 5 shows the run times 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 unlabeled 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 labeled 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.

3.2.3 *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 $a_1...a_{k-2}$. Intersections that access the (k–1)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 $a_1...a_{k-2}$ 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 2 a . One type of WCO plans for this query extend $u\rightarrow v$ edges to (u, v, w) symmetric triangles by intersecting u's backward and v's forward adjacency lists. Then each triangle is extended to complete the query, intersecting again the forward and backward adjacency lists of one of the edges of the triangle. There are two sub-groups of QVOs that fall under this type of plans: (i) $a_2a_3a_1a_4$ and $a_2a_3a_4a_1$, which are equivalent plans due to symmetries in the query, so will perform exactly the same operations; and (ii) $a_1a_2a_3a_4$, $a_3a_1a_2a_4$, $a_3a_4a_2a_1$, and $a_4a_2a_3a_1$, which are also equivalent plans. Importantly, all of these plans cumulatively perform exactly the same intersections but those in group (i) and (ii) have different orders in which these intersections are performed, which lead to different intersection cache utilizations.

		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

Table 5: Runtime (secs), 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
$a_2a_3a_1a_4$	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

Table 6: Runtime (secs), intermediate partial matches (part. m.), and i-cost of some QVOs for the symmetric diamond-X query.

Table 6 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 4.4x faster on Epinions and 3x faster on Amazon. 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),...,(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* [33]. 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.

3.3 Cost Metric for WCO Plans

We introduce a new cost metric called *intersection cost* (i-cost), which we define as the size of adjacency lists that will be accessed and intersected by different WCO plans. Consider a WCO plan σ that evaluates sub-queries Q_2,...,Q_m, respectively, where Q=Q_m. Let t be a (k–1)-match of Q_{k-1} and suppose t is extended to instances of Q_k by intersecting a set of adjacency lists, described

with adjacency list descriptors A_{k-1}. Formally, i-cost of σ is:

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

We discuss how we estimate i-costs of plans in Section 5. 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 4, 5, and 6. The actual i-costs were computed in a profiled re-run of each experiment. 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} [12] and C_{mm} [20] 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 4 and 6 have the same actual C_{out} and C_{mm} costs.

4. FULL PLAN SPACE & DP 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.

4.1 Hybrid Plans and HashJoin Operator

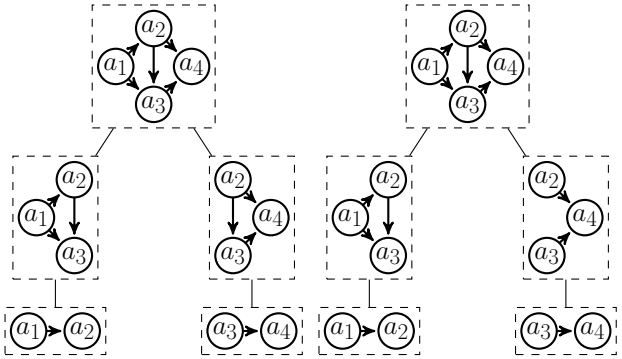
In Section 3, we represented a WCO plan σ as a chain, where each internal node o_k had a single child labeled with Q_k, which was the projection of Q onto the first k query vertices in σ. A plan in our full plan space is a rooted tree as follows. Below, Q_k refers to a projection of Q onto an arbitrary set of k query vertices.

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

As before, leaves map to SCAN operator, 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 Appendix A 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 1 d 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.



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

Figure 3: Two plans: $P1$ shares a query edge and $P2$ does not.

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. Such BJ plans are in the plan spaces of existing optimizers, e.g., PostgreSQL, 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, avoiding the computation of open triangles.
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 (P_1 is repeated from Figure 1 c). 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, P_1 takes 14.2 seconds and P_2 56.4 seconds.

4.2 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 [20], but we pick them empirically. In particular we run experiments in which we profile plans with E/I and HASH-JOIN operators and we log the (i-cost, time) pairs for the E/I operators, and the (n_1, n_2, time) triples for the HASH-JOIN operators. The (i-cost, time) pairs allows us to convert time unit in the triples to i-cost units. We then pick w_1 and w_2 that best fit these converted (n_1, n_2, i-cost) triples.

Algorithm 1 DP Optimization Algorithm

Input: $Q(V_Q, E_Q)$
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, ..., $|V_Q|$ **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 = ∞
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);

4.3 Dynamic Programming Optimizer

Algorithm 1 shows the pseudocode of our optimizer. We next describe our optimizer, whose pseudocode is in the longer version of our paper [1]. Our optimizer takes as input a query $Q(V_Q, E_Q)$. We start by enumerating and computing the cost of all WCO plans. We discuss this step momentarily. We then initialize the cost of computing 2-vertex sub-queries of Q, so each query edge of Q, to the selectivity of the label on the edge. 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 optimal plan in terms of estimated cost. However, we verified that our optimizer returns the same plan as a full enumeration optimizer in all of our experiments. So at least for our experiments here, 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 $a_1 \to a_2 \to a_3$ query. Instead of using a HASH-JOIN to materialize the $a_2 \to a_3$ edges and then probe a scan of $a_1 \to a_2$ edges, it is more efficient to use an E/I to extend $a_1 \to a_2$ edges to a_3 using a_2's forward adjacency list.

4.4 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, we keep only a subset (5 by default) k-vertex sub-queries of Q with the lowest cost plans. So we store a subset of sub-queries in our sub-query map and enumerate only the Q_k that can be generated from the sub-queries we stored previously in the map.

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

Table 7: A subgraph catalogue. A is a set of adjacency list descriptors; μ is selectivity.

5. COST & 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 sub-query Q_k has labels on the edges and the vertices. We use a data structure called the *subgraph catalogue* to make the estimations. Table 7 shows an example catalogue.

Each entry contains a key (Q_{k-1}, A, $a_k^{l_k}$), where A is a set of (labeled) query edges and $a_k^{l_k}$ is a query vertex with label l_k. Let Q_k be the subgraph that extends Q_{k-1} with a query vertex labeled 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 a set of adjacency lists described by A:

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 7, 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 A with different i-costs. The fourth and fifth entries of Table 7, which extend a single edge to an asymmetric triangle, demonstrate this possibility.

5.1 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 (1000 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.

5.2 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, 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, ...a_z from Q_{x-1} and Q_x, and the adjacency list descriptors from A_x that include 1, ..., 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.

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 will not utilize the intersection cache, we estimate i-cost of o_k as:

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

  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 3 to demonstrate that the intersection cache broadly improves all plans for the diamond-X query. Our optimizer, which is "cache-conscious" picks σ_2 ($a_2a_3a_4a_1$). Instead, if we ignore the cache and make our optimizer "cache-oblivious" by always estimating i-cost with Equation 2, it picks the slower σ_4 ($a_1a_2a_3a_4$) plan. Similarly, our cache-conscious optimizer picks $a_2a_3a_1a_4$ in our experiment from Table 6. 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-JOIN operator:** Consider a HASH-JOIN operator joining Q_b and Q_p. The estimated cost of this operator is simply w_1n_1 + w_2n_2 (recall Section 4.2), where n_1 and n_2 are now the estimated cardinalities of Q_b and Q_p, respectively.

5.3 Limitations

Similar to Markov tables [4] and MD- and Pattern-tree summaries [24], 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 9. Here, we discuss several limitations that are inherent in such techniques.

First, 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 Appendix B for cardinality estimates.

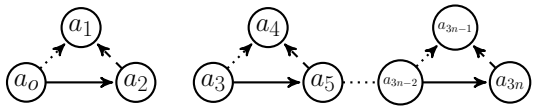


Figure 4: Input graph for adaptive QVO example.

6. ADAPTIVE WCO PLAN EVALUATION

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

EXAMPLE 6.1. *Consider the input graph G shown in Figure 4. G contains 3n edges. Consider the diamond-X query and the WCO plan P with σ=$a_2a_3a_4a_1$. Readers can verify that this plan will have an i-cost of 3n: 2n from extending solid edges, n from extending dotted edges, and 0 from extending dashed edges. Now consider the following* adaptive *plan that picks σ for the dotted and dashed edges as before but σ'=$a_2a_3a_1a_4$ for the solid edges. For the solid edges, σ' incurs an i-cost of 0, reducing the i-cost to n.*

6.1 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 $o_i, o_{i+1}..., o_k$, 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 5 shows the adaptive version of the fixed plan for the diamond-X query from Figure 1 b . We note that in addition to WCO plans, we adapt hybrid plans if they have a chain of two or more E/I operators.

6.2 Adaptive Operators

Unlike the operators in fixed plans, our adaptive operators can feed their outputs to multiple operators. An adaptive operator o_i is

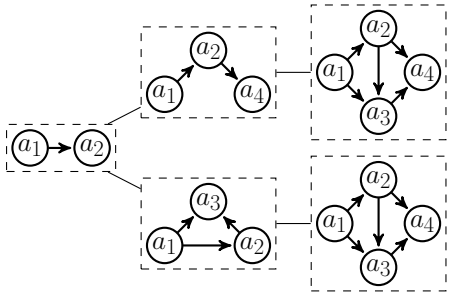


Figure 5: Example adaptive WCO plan (drawn horizontally).

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

EXAMPLE 6.2. *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 σ_1 : $a_2a_3a_4a_1$ or σ_2 : $a_2a_3a_1a_4$. Suppose the catalogue estimates the sizes of $|a_2{\rightarrow}|$ and $|a_3{\rightarrow}|$ as 100 and 2000, respectively. So we estimate the i-cost of extending an a_2a_3 edge to $a_2a_3a_4$ as 2100. Suppose the selectivity μ_j of the number of triangles this intersection will generate is 10. Suppose* SCAN *reads an edge $u{\rightarrow}v$ where u's forward adjacency list size is 50 and v's backward adjacency list size is 200. Then we update our i-cost estimate directly to 250 and μ_j to 10 $\times$ (50/100) $\times$ 200/2000=0.5.*

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

7. SYSTEM IMPLEMENTATION

We build our new techniques on top of Graphflow DBMS [18]. Graphflow is a single machine, multi-threaded, main memory graph DBMS implemented in Java. The system supports a subset of the Cypher language [34]. 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, *types* in Cypher jargon, and further by the labels of the destination vertices. 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. Our query plans follow a Volcano-style plan execution [16]. 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 implemented a work-stealing-based technique to parallelize the evaluation of our plans. 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.

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

Table 8: Datasets used.

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. If HASH-JOIN's hash and probe children compute completely symmetric sub-queries, we compute that sub-query once, use it to construct the hash table, and then re-use it to probe.

8. EVALUATION

Our experiments aim to answer four questions: (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? As part of our EH comparisons, we also tested the scalability of our single-threaded and parallel implementation on our largest graphs LiveJournal and Twitter. Finally, for completeness of our study, Appendix C compares our plans against CFL and Neo4j.

8.1 Setup

8.1.1 Hardware

We use a single machine that has two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical cores. Except our scalability experiments in Section 8.5, we use only one physical core. We set the maximum size of the JVM heap to 500 GB and keep JVM's default minimum size. We ran each experiment twice, one to warm-up the system and recorded measurements for the second run.

8.1.2 Datasets

The datasets we use are in Table 8.[2] 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 the amount of cliques in it. The datasets also come from a variety of application domains: social networks, the web, and product co-purchasing. Each dataset's catalogue was generated with z=1000 and h=3 except for Twitter, where we set h=2.

8.1.3 Queries

For the experiments in this section, we used the 14 queries shown in Figure 6, which contain both acyclic and cyclic queries with dense and sparse connectivity with up to 7 query vertices and 21 query edges. In our experiments, we consider both labeled and unlabeled queries. Our datasets and queries are not labeled by default and we label them randomly. We use the notation QJ_i to refer to evaluating the subgraph query QJ 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 $Q3_2$ on Amazon indicates randomly adding

[2]We obtained the graphs from reference [22] except for the Twitter graph, which we obtained from reference [19].

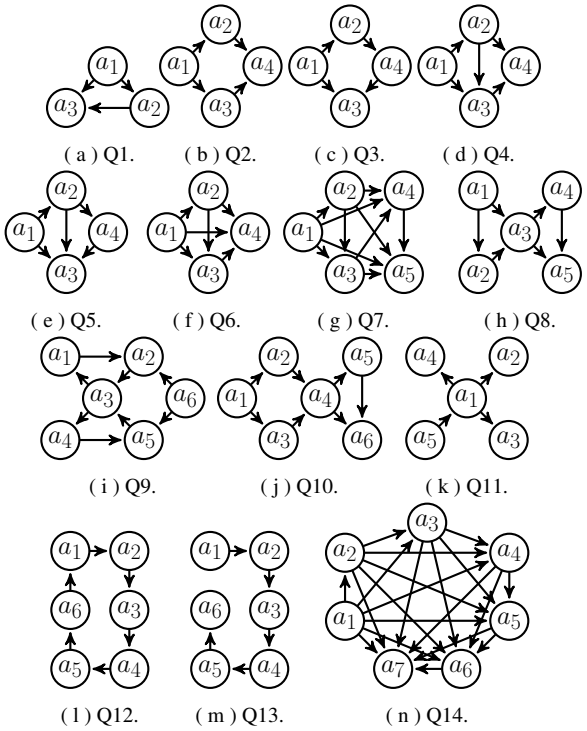


Figure 6: Subgraph queries used for evaluations.

one of two possible labels to each data edge in Amazon and query edge on $Q3$. If a query was unlabeled we simply write it as QJ.

8.2 Plan Suitability For Different Queries and Optimizer Evaluation

In order to evaluate how good are the plans our optimizer generates, we compare the plans we pick against all other possible plans in a query's plan spectrum. 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. All of our spectrums are shown in Figure 7. Each circle in Figure 7 is the runtime of a plan and $\times$ is the plan our optimizer picks.

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 clique-like densely cyclic queries, such as $Q5$, and small sparsely-cyclic 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 1 d 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 Google graph our optimizer picks an efficient BJ plan although the optimal plan is WCO.

Our optimizer's plans were broadly optimal or very close to optimal across our experiments. Specifically, our optimizer's plan was optimal in 15 of our 31 spectrums, was within 1.4x of the optimal in 21 spectrum and within 2x in 28 spectrums. In 2 of the 3 cases we were more than 2x of the optimal, the absolute runtime difference was in sub-seconds. There was only one experiment in which our plan was not close to the optimal plan, which is shown in Figure 7 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 data sets ($Q8$ and $Q13$ are other examples).

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

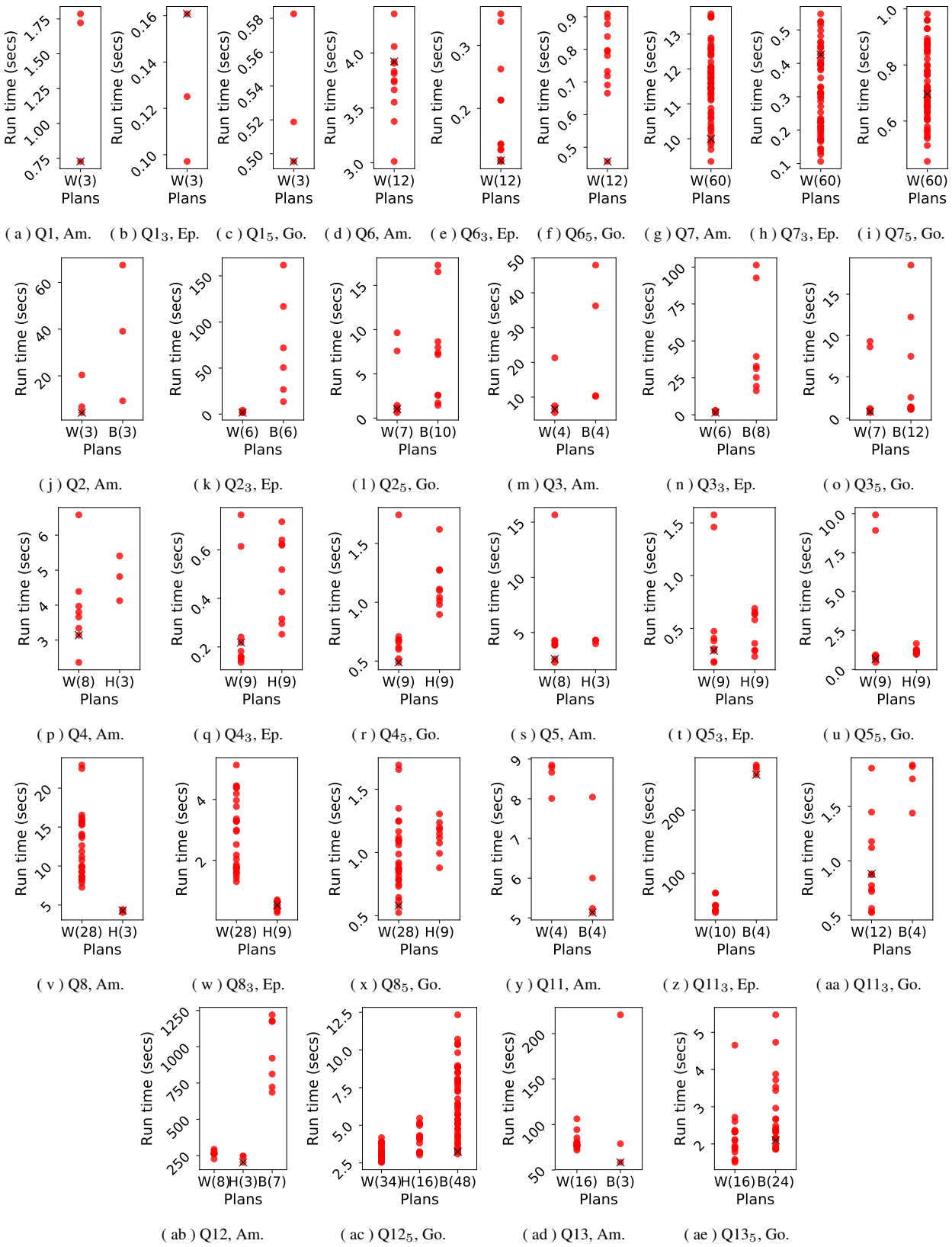
8.3 Adaptive WCO Plan Evaluation

In order 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 8 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 1.2x but other plans improve by up to 1.6x. $Q10$'s spectrum for 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 2.1x. On $Q5$ most plans' runtimes remain similar but one WCO plan improves by 4.3x. 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 plan's performance gets 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.), and the benefits are low.

8.4 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 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 though that



(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.

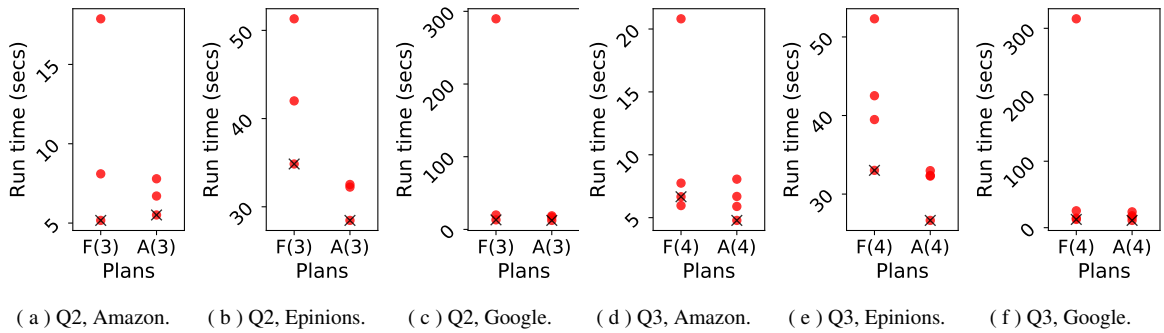
(j) Q2, Am. (k) Q2$_3$, Ep. (l) Q2$_5$, Go. (m) Q3, Am. (n) Q3$_3$, Ep. (o) Q3$_5$, Go.

(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$_3$, Go.

(ab) Q12, Am. (ac) Q12$_5$, Go. (ad) Q13, Am. (ae) Q13$_5$, Go.

Figure 7: Plan spectrum charts for suitability and goodness of DP optimizer experiments.



(a) Q2, Amazon. (b) Q2, Epinions. (c) Q2, Google. (d) Q3, Amazon. (e) Q3, Epinions. (f) Q3, Google.

(g) Q4, Amazon. (h) Q4, Epinions. (i) Q4, Google. (j) Q5, Amazon. (k) Q5, Epinions. (l) Q5, Google.

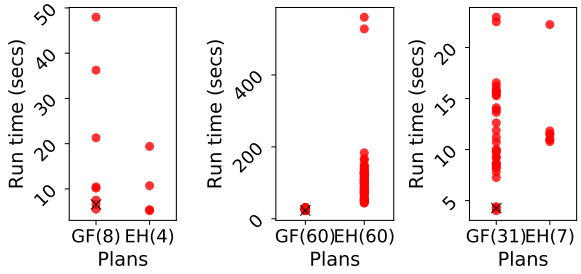
(m) Q6, Amazon. (n) Q6, Epinions. (o) Q6, Google. (p) Q10, Amazon. (q) Q10, Google.

Figure 8: Adaptive plan spectrums.

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" variables (`EH-b`) and "good" variables (`EH-g`). For good orderings we use the ordering that Graphflow 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.

8.4.1 *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 9 shows a sample of the spectrums for Q3 and Q7 on Amazon and for Q8 on Epinions along with Graphflow'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 minutes time limit. For Q7, both Graphflow and EH generate only WCO plans. For Q8, EH generates two GHDs (two triangles joined on a_3) whose different QVOs give 4 different plans



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

Figure 9: Plan spectrum charts for EmpthHeaded (EH).

for a total of 8. One of the plans in the spectrum is omitted as it had memory issues. We note that out of these queries, Q8 and Q9 were the only queries for which EH generated two different decompositions (ignoring the QVOs of sub-queries). For Q but neither decomposition under any QVO ran within our time limit on our datasets.

8.4.2 *Graphflow vs EH Comparisons*

We ran our queries on Graphflow 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

		Q1	Q3	$Q3_2$	Q5	$Q5_2$	Q7	$Q7_2$	Q8	$Q8_2$	Q9	$Q9_2$	Q12	$Q12_2$	Q13	$Q13_2$
Amazon	EH-b	1.0	19.0	3.4	47.1	9.2	91.4	11.6	22.2	1.8	Mm	Mm	Mm	Mm	Mm	Mm
	EH-g	0.6	5.4	1.3	3.3	1.5	21.2	1.7	10.6	1.4	Mm	Mm	Mm	Mm	Mm	Mm
	GF	0.6	5.5	2.1	1.9	0.8	9.53	0.9	5.1	2.0	24.7	2.4	209.2	14.8	48.0	11.25
Google	EH-b	1.9	444.5	42.6	401.1	77.6	1.04K	23.4	66.6	16.0	TL	TL	Mm	Mm	Mm	Mm
	EH-g	1.4	12.0	2.1	11.3	2.3	107.3	4.8	35.8	3	TL	TL	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
Epinions	EH-b	0.5	42.7	6.5	64.5	11.4	560.7	2.9	1.01K	22.0	TL	TL	Mm	Mm	TL	TL
	EH-g	0.2	26.6	1.7	3.5	0.9	45.7	0.8	117.2	7.0	TL	TL	Mm	Mm	TL	TL
	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	TL	TL

Table 9: Runtime in secs of Graphflow plans (GF) and EmptyHeaded with good orderings (EH-g) and bad orderings (EH-b). TL indicates the query did not finish in 30 mins. Mm indicates the system ran out of memory.

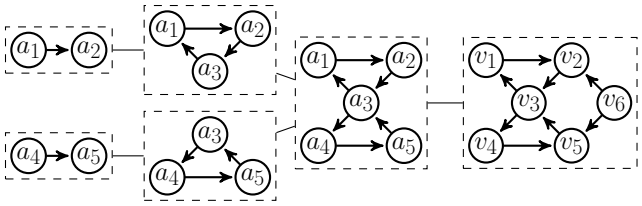
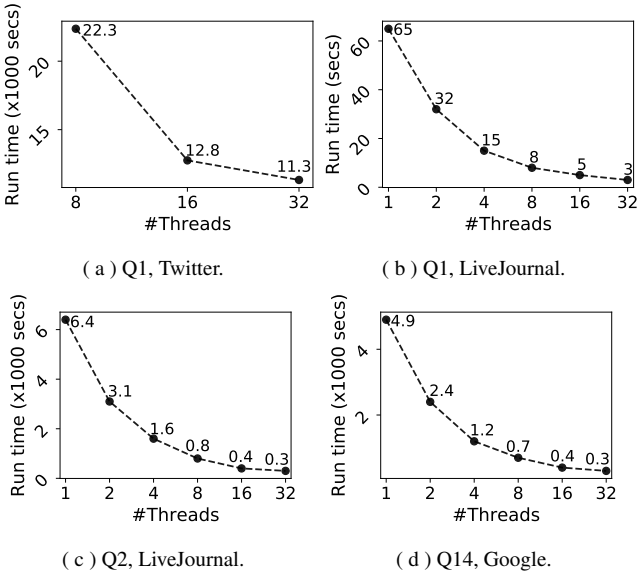


Figure 10: Plan (drawn horizontally) with seamless mixing of intersections and binary joins on Q9.



(a) Q1, Twitter. (b) Q1, LiveJournal.

(c) Q2, LiveJournal. (d) Q14, Google.

Figure 11: Scalability experiments.

$Q13$). We repeated the experiments once with no labels and once with two labels. Table 9 shows our results. Except for $Q1$ on Google and $Q8_2$ on Amazon where the difference is only 500ms and 200ms, respectively. Graphflow is always faster than EH-b, where the runtime is as high as 68x in one instance. The most performance difference is on $Q5$ and Google, for which both our system and EH uses 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 Graphflow is 32x faster than EH-b on $Q3$ Google, it is 1.2x slower than EH-g. Importantly EH-g is always faster than EH-b, showing that our QVOs improve runtimes consistently in a completely independent system that implements WCO-style processing.

We next discuss $Q9$, which demonstrates again the benefit we get by seamlessly mixing intersections with binary joins. Figure 10 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. 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.

8.5 Scalability Experiments

We next demonstrate the scalability of Graphflow on larger datasets and linear scalability across physical cores. We evaluated $Q1$ on LiveJournal and Twitter, $Q2$ 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 11 shows our results. Our plans scale linearly until 16 cores with a slight slow down when moving 32 cores which uses all system resources. For example, going from 1 core to 16 cores, our runtime is reduced by 13x for $Q1$ on LiveJournal, 16x for $Q2$ on LiveJournal and 12.3x for $Q14$ on Google.

9. RELATED WORK

We review related work in WCO join algorithms, subgraph query evaluation algorithms, and cardinality estimation techniques related to our catalogue. For join and subgraph query evaluation, we focus on serial algorithms and single node systems. Several distributed solutions have been developed in the context of graph data processing [38, 23], RDF engines [2, 42], or multiway joins of relational tables [5, 6, 35]. We do not review this literature here in detail. There is also a rich body of work on adaptive query processing in relational systems for which we refer readers to reference [15].

WCO Join Algorithms Prior to GJ, there were two other WCO join algorithms called NPRR [31] and Leapfrog TrieJoin (LFTJ) [40]. Similar to GJ, these algorithms also perform attribute-at-a-time join processing using intersections. The only reference that studies QVOs in these algorithms is reference [11], which studies picking the QVO for LFTJ algorithm in the context of multiway relational joins. The reference picks the QVO based on the distinct values in the attribute of the relations. In subgraph query context, this heuristic ignores the structure of the query, e.g., whether the query is cyclic or not, and effectively orders the query vertices based on the selectivity of the labels on them. For example, this heuristic becomes degenerate if the query vertices do not have labels.

Subgraph Query Evaluation Algorithms: Many of the earlier subgraph isomorphism algorithms are based on Ullmann's branch and bound or backtracking method [39]. 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 [13, 14, 37]. Turbo$_{ISO}$, for example, proposes

to merge similar query vertices (same label and neighbours) to minimize the number of partial matches and perform the Cartesian product to expand the matches at the end. CFL [9] decomposes the query into a dense subgraph and a forest, and process the dense subgraph first to reduce the number of partial matches. CFL also uses an index called *compact path index (CPI)* 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 Appendix C. A systematic comparison of our approach against these approaches is beyond the scope of this paper. Our approach is specifically designed to be directly implementable on any DBMS that adopts a cost-based optimizer and decomposable operator-based query plans. In contrast, these algorithms do not seem easy to decompose into a set of database operators. Studying how these algorithms can be turned into database plans is an interesting area of research.

Another group of algorithms index different structures in input graphs, such as frequent paths, trees, or triangles, to speed up query evaluation [43, 41]. Such approaches can be complementary to our approach. For example, reference [6] in the distributed setting demonstrated how to speed up GJ-based WCO plans by indexing triangles in the graph.

Cardinality Estimation using Small-size Graph Patterns: Our catalogue is closely related to Markov tables [4], and MD- and Pattern-tree summaries from reference [24]. Similar to our catalogue, both of these techniques store information about small-size subgraphs to make cardinality estimates for larger subgraphs. Markov tables were introduced to estimate cardinalities of paths in XML trees and store exact cardinalities of small size paths to estimate longer paths. MD- and Pattern-tree techniques store exact cardinalities of small-size acyclic patterns, and are used to estimate the cardinalities of larger subgraphs (acyclic and cyclic) in general graphs. These techniques are limited to cardinality estimation and store only acyclic patterns. In contrast, our catalogue stores information about acyclic and cyclic patterns and is used for both cardinality and i-cost estimation. In addition to selectivity (μ) estimates that are used for cardinality estimation, we store information about the sizes of the adjacency lists (the $|A|$ values), which allows our optimizer to differentiate between WCO plans that generate the same number of intermediate results, so have same cardinality estimates, but incur different i-costs. Storing cyclic patterns in the catalogue allow us to make accurate estimates for cyclic queries.

10. CONCLUSIONS

We described a cost-based dynamic programming optimizer that enumerates a plan space that contains WCO plans, BJ plans, and a large class of hybrid plans. Our i-cost metric captures the several runtime effects of QVOs we identified through extensive experiments. Our 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 approach has 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 queries, currently our optimizer enumerates a limited part of our plan space. Studying faster plan enumeration methods, similar to those discussed in [27], is an important future work direction. Finally, existing literature on subgraph matching has several optimizations, such as factorization [33] or postponing the Cartesian product optimization [9], for evaluating identifying and evaluating independent components of a query separately. We believe these are efficient optimizations that can be integrated into our optimizer.

11. REFERENCES

[1] A. Mhedhbi and S. Salihoglu. Evaluating Subgraph Queries by Combining Binary and Worst-case Optimal Joins. *CoRR*, abs/1903.02076, 2019.

[2] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. Spartex: A vertex-centric framework for RDF data analytics. *PVLDB*, 8(12), 2015.

[3] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. *TODS*, 42(4), 2017.

[4] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, 2001.

[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *TKDE*, 2011.

[6] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *PVLDB*, 11(6), 2018.

[7] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *SIGMOD*, 2015.

[8] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SICOMP*, 42(4), 2013.

[9] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.

[10] A. Bodaghi and B. Teimourpour. *Automobile Insurance Fraud Detection Using Social Network Analysis*. Springer International Publishing, 2018.

[11] S. Chu, M. Balazinska, and D. Suciu. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD*, 2015.

[12] S. Cluet and G. Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *ICDT*, 1995.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance Evaluation of the VF Graph Matching Algorithm. In *ICIAP*, 1999.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)graph Isomorphism Algorithm for Matching Large Graphs. *TPAMI*, 26(10), 2004.

[15] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1), 2007.

[16] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 6(1), 1994.

[17] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin. Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *PVLDB*, 7(13), 2014.

[18] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

[19] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[20] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3), 2015.

[21] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDBJ*, 27(5), 2018.

[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[23] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. Scalable Distributed Subgraph Enumeration. In *VLDB*, 2016.

[24] A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. Graph Summaries for Subgraph Frequency Estimation. In *The Semantic Web: Research and Applications*, 2008.

[25] Neo4j. `https://neo4j.com/`.

[26] Fraud Detection: Discovering Connections with Graph Databases. `https://neo4j.com/use-cases/fraud-detection`.

[27] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 677–692, New York, NY, USA, 2018. ACM.

[28] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal*, 19(1), 2010.

[29] M. E. J. Newman. Detecting Community Structure in Networks. *The European Physical Journal B*, 38(2), 2004.

[30] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 42(4), 2014.

[31] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms. In *PODS*, 2012.

[32] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. *CoRR*, abs/1503.04169, 2015.

[33] D. Olteanu and J. Závodný. Size Bounds for Factorised Representations of Query Results. *TODS*, 40(1), 2015.

[34] openCypher. `http://www.opencypher.org`.

[35] Paraschos Koutris and Semih Salihoglu and Dan Suciu. Algorithmic aspects of parallel data processing. *Foundations and Trends in Databases*, 8(4), 2018.

[36] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The Graph Story of the SAP HANA Database. In *BTW*, 2013.

[37] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *PVLDB*, 1(1), 2008.

[38] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel Subgraph Listing in a Large-scale Graph. In *SIGMOD*, 2014.

[39] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[40] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[41] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*, 2004.

[42] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *PVLDB*, 6(4), 2013.

[43] P. Zhao, J. X. Yu, and P. S. Yu. Graph Indexing: Tree + Delta ¡= Graph. In *VLDB*, 2007.

APPENDIX

A. SUBSUMED EH PLANS

We show that our plan space contains the EH's GHD-based plans that satisfy the projection constraint. For details on GHDs how EH picks GHDs we refer the reader to reference [3]. Briefly, a GHD D of Q is a decomposition of Q where each node i is labeled with a sub-query Q_i of Q. The interpretation of a GHD D as a join plan is as follows: each sub-query 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 4) by "expanding" each sub-query Q_i into a WCO (sub-) plan according to the σ that EH picks for Q_i and adding intermediate nodes in T that are the results of the joins that EH performs. Given Q, EH picks the GHD D^* for Q as follows. First, EH 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 [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 6, the plans EH picks satisfy the projection constraint. However, there are minimum-width GHDs that do not satisfy this constraint. For example, for Q10, EH 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. EH (arbitrarily) picks the first GHD. As we argued in Section 4.1, 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 [**?**] (neither GHD finished in a reasonable amount of time on our datasets from Table 8), the first GHD for Q10 takes around 150ms, while the second one does not finish within 30 minutes.

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 1000, 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=1000, and z=5000 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 work cardinality estimation [21] 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 10 and 11 as cumulative distributions as follows: for different q-error bounds τ, we show the number of queries that a particular catalogue estimated with q-error at most τ. As expected, larger h and larger z values lead to less q-error, while respectively yielding larger catalogue sizes and longer construction times,. The biggest q-error differences are obtained when moving from h=3 to h=4 and z=100 to z=500. There are a few exception τ values when the larger h or z values lead to very minor decreases in the number of queries within the τ bound but the trend holds broadly.

	z	time(s)	≤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	461	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 10: Q-error and construction time for different z values.

	h	|entries|	≤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

Table 11: Q-error and catalogue size (MB) for different h values.

C. CFL COMPARISON

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

Table 12: Average run-time (secs) of Graphflow (GF) and CFL.

CFL [9] is an efficient algorithm in literature that can evaluate labeled subgraph queries as in our setting. The main optimization of CFL is what is referred to as "postponing Cartesian products" in the query. Essentially, 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 [9]. We obtained the CFL code and 6 different query sets used in reference [9] from the authors. Each query set contains 100 randomly generated queries that are either sparse (average query vertex degree ≤ 3) or dense (average query vertex degree > 3). We used three sparse query sets Q10s, Q15, and Q20s containing queries with 10, 15, and 20 query vertices, respectively. Similarly, we used three dense query sets Q10d, Q15d, and Q20d. To be close to their setup, we use the human dataset from the original CFL paper. The dataset contains 86282 edges, 4674 vertices, 44 distinct labels. We report the average run-time per query for each query set when we limit the output to 10^5 and 10^8 matches as done in reference [9]. Table 12 compares the runtime of Graphflow and CFL on the 6 query sets. Except for one of our experiments, on Q10d with 10^5 output size limit, Graphflow's runtimes are faster (between 1.2x to 12.2x) 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 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.

D. NEO4J COMPARISON

Neo4j is perhaps the most popular graph DBMS that uses BJ plans to evaluate queries. We used Neo4j v.3.1.0. Our runtime results are significantly faster (up to 837x) as show in Table 13 and we expect similar results against systems using BJ plans. Aside from our advantage of using WCO plans with good QVOs, our implementation has several advantages against Neo4j: (1) Graphflow is a prototype system, which is inherently more efficient as it supports fewer features; and (2) instead of Neo4j's linked lists storing Java objects, our graph store is backed by Java primitive type arrays, which are faster in lookups; (3) we store our adjacency list in sorted vertex ID order. Similar to our note above, although baseline comparisons as our Neo4j comparisons are common in the database research community, we think there is little to learn from these experiments. Neo4j is not optimized for the complex subgraph queries we study in this paper.

		Q1	Q2	Q4
Am	GF	0.7	5.7	4.8
	Neo4j	332.1 (**474x**)	TL	745.2 (**155x**)
Ep	GF	0.6	42.2	1.5
	Neo4j	502.4 (**837x**)	TL	TL

Table 13: Run-time (secs) of Graphflow (GF) and Neo4j. TL indicates the query did not finish in 30 mins.