Multiple Continuous Subgraph Query Optimization Using Delta Subgraph Queries

by

Chathura Kankanamge

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis studies the problem of optimizing and evaluating multiple directed structural subgraph queries, i.e., those without highly selective predicates on the edges or vertices, continuously in a changing graph. Existing techniques focus on queries with highly selective predicates or are designed for evaluating a single query. As such, these techniques do not scale when evaluating multiple structural queries either because their computations become prohibitively inefficient or they use prohibitively large auxiliary data structures.

We build upon the *delta subgraph query (DSQ)* framework that was introduced in prior work. This framework decomposes queries into multiple *delta queries*, which are then evaluated one query vertex at a time, without requiring any auxiliary data structures. We study the problem of picking good query vertex orderings for a set of DSQs cumulatively to share computation across different DSQs and achieve efficient runtimes in practice. We describe a greedy cost-based optimizer that takes as input a set of DSQs and a *subgraph extension catalogue*, and generates a single low cost *combined plan* that cumulatively evaluates all of the DSQs by sharing computation across them. Our combined plans consist of a SCAN and multiple EXTEND/INTERSECT (E/I) operators. The E/I operator takes a set of partial matches and extends them by one query vertex. We adopt as our cost metric *intersection cost* (i-cost), which we show is a good estimate of the actual work performed during query evaluation. We further describe an optimization to the base optimizer that expands the DSQs algebraically to even more DSQs to allow more computation sharing between them. On small query sets we demonstrate that our cost-based greedy optimizer is able to find close to optimal combined plans in terms of run time. On larger query sets, we demonstrate that our optimizer and expanded DSQ optimization can yield significant performance improvements against several baselines.

Acknowledgements

I want to first thank my supervisor, Professor Semih Salihoglu for all the guidance he gave thoughout this project. Semih introduced me to query evaluation in graphs and much of the framework on which this thesis is based. In addition, he made many contributions in formulating the manner in which we express the work in thesis. I am grateful to him for all the help and advice he gave me, as well as the many nights he spent reading through the thesis.

I next want to thank my colleagues Amine Mhedhbi and Siddhartha Sahu for the many contributions they made to this project. They both contributed to creating the Graphflow system which formed the basis for the experiments described in the thesis. In addition, Amine contributed many key ideas used in this thesis such as the i-cost metric and using the subgraph extension catalog.

I also want to thank my lab mate Barber Memon for participating in many discussions regarding my research and giving valueable comments. I also want to thank my friends Abeer Khan, Nimesh Ghelani, Tarun Patel, and others who contributed in various ways to this thesis.

Last but not least I want to thank my parents, Vijitha Senevirathne and Upul Wasantha, and my sister Chathuri Kankanamge for the invaluable support and guidance they have given me thoughout my life.

Dedication

This is dedicated to my mother and father.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A subgraph query (SQ) is the problem of finding instances of a query subgraph in an input graph. SQs are one of the most fundamental queries supported by several graph processing systems, such as graph databases and RDF engines. Many applications require finding instances of subgraphs on a graph that is changing, i. e. detecting the emergence or deletion of subgraphs that were created or were deleted due to changes in the graph. We refer to SQs issued on such dynamic graphs as *continuous subgraph queries (CSQs)*. SQs (and CSQs) in general have two components: a *structural* component that describes the attributes of the subgraph to be matched and a *filter* component that limits results to subgraphs that satisfy certain predicates on the nodes and edges of the matched subgraphs. In this thesis we focus on the first component and study *how to efficiently evaluate and optimize multiple structural CSQs, i.e., without highly selective predicates*. As one of the two main components of subgraph queries, the insights gained by optimizing structural CSQs can be useful in studying subgraph matching in general. In addition, several applications directly use structural queries. For example, Twitter has reported searching for diamonds in their who-follows-whom graph to make recommendations [1], we are aware of a financial company that continuously searches for multiple cycle patterns with up to eight edges in a transaction network, and detecting clique-like structures in friendship networks which indicate emerging or dissolving communities [2].

The high-level problem we address in this work is as follows:

Given a set $\bar{Q}$ of directed subgraph queries, detect the emergence or deletion of instances dQ of each $Q \in \bar{Q}$ in a graph G that is changing due to a batch of updates which include additions and/or deletions of edges. When it is not clear from context, we refer to vertices and edges in queries as *query vertices and edges* and vertices and edges in the input graph

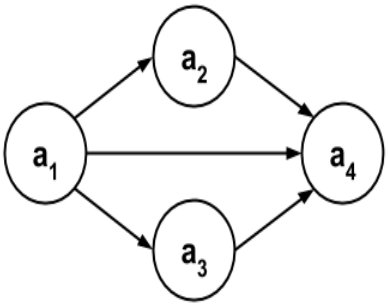
G as *data vertices and edges*.

1.1 Existing Work and Challenges

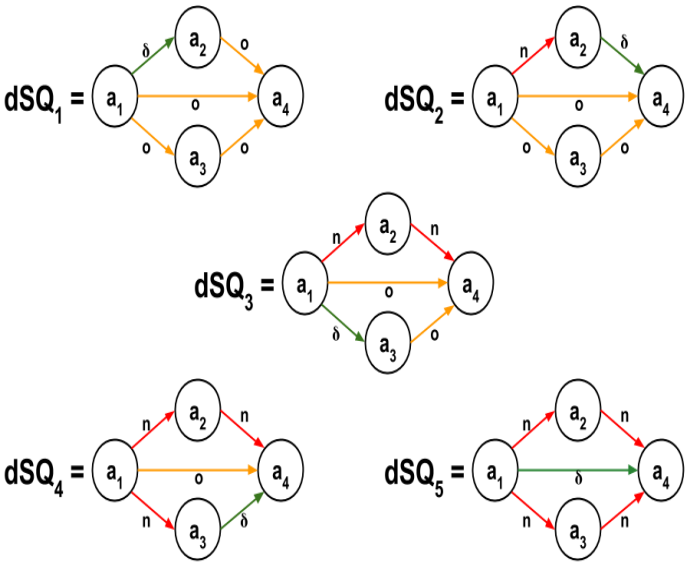
Prior work on continuous subgraph queries have two main shortcomings: (i) They are either designed for evaluating a single query instead of multiple queries [3, 4, 5], so do not benefit from optimization possibilities across multiple queries; and/or (ii) they are designed for queries that contain highly selective predicates on the edges and vertices of the queries [4,5,6,7,8]. As such, for multiple structural queries, the approaches of the second group either become prohibitively inefficient or require prohibitively large auxiliary data structures. For example, several of existing approaches [6,8] make the explicit assumption that the number of partial matches of paths with more than one edge will be significantly smaller than one-edge paths, which often will not hold in the absence of heavy predicates in the queries. Similarly, approaches that use auxiliary data structures [4, 5] often index some representation of partial matches of the queries, e.g., 2-edge paths or triangles, which may be prohibitively large, limiting the scalability of these approaches. We review existing approaches in detail in Chapters 6 and 7 and provide experimental comparisons against some of them in Chapter 6.

1.2 Summary of Approach

As we review in Chapter 2, any SQ Q can be equivalently characterized as a multiway join query on the replicas of the "edge table" of G. Similarly, any CSQ can be equivalently characterized as the incremental view maintenance problem of a multiway join query. Our approach is based on the *Delta Generic Join (Delta-GJ)* algorithm for continuously evaluating a single CSQ, first described in reference [3]. Delta-GJ adopts the multiway join query view of subgraph queries and is based on an algebraic incremental view maintenance technique called *delta join query decomposition* of queries [9]. This technique decomposes a single CSQ Q into multiple *delta subgraph queries* (DSQs). Figure 1.1a shows the `diamond-with-edge` query and Figure 1.1b the five DSQs of the `diamond-with-edge` query. Let E_o be the **o**ld edges of G, E_δ a set of new updates, and E_n the **n**ew edges of G after the update, i.e., $E_n = E_o \cup E_\delta$. Each edge of DSQs are labeled with δ, o, or n, indicating whether the edge should match an edge, respectively, in E_δ, E_o, or E_n, upon an update to G. As we review in Chapter 2, a CSQ can be computed by evaluating and taking the union of the results of each DSQ of Q. In the Delta-GJ algorithm, the evaluation of



(a) `Diamond-with-edge` CSQ



(b) DSQs of `diamond-with-edge` CSQ

Figure 1.1: `Diamond-with-edge` Continuous Subgraph Query (CSQ) and its Delta Subgraph Queries (DSQs)

each DSQ starts by partially matching an edge with label δ and then extending these partial matches one (query) vertex at a time. To extend, we intersect one or more adjacency lists of the vertices in these partial matches, either from E_o or E_n. Several prior work has demonstrated, both in the serial and distributed settings [3,10][1], the run time and memory benefits and theoretical properties of the overall Delta-GJ approach for evaluating a single query. In this work, we build upon this framework and describe techniques for optimizing multiple queries cumulatively.

The fundamental decision an optimizer has to make within this framework is to pick an order for matching vertices for each DSQ. We describe a general greedy cost-based optimizer that takes as input the DSQ decompositions of a set of queries $\bar{Q}$ and a *subgraph extension catalogue* (catalogue for short) and decides on a query vertex ordering for each DSQ in order to optimize some cost metric. The catalogue contains information about the estimated cost of extending each subgraph S to its possible one-vertex extensions S' and the associated *extension rate* of each extension. The output of our optimizer is a *combined*

[1]One interesting property that was proven was that this approach is worst-case optimal under insertion only workloads [3].

plan that evaluates all of the queries continuously as updates arrive to G. In Chapter 3 we discuss factors that the optimizer must take into account when optimizing multiple DSQs, such as sharing computation across DSQs and minimizing intermediate partial matches generated during evaluation.

The combined plans are evaluated using two physical operators, a SCAN and EXTEND/INTERSECT (E/I) operator. Using different catalogues, we use our optimizer to optimize three different natural cost metrics: (i) the number of operators in the combined plan (`num-ops`); (ii) the number of intermediate partial matches generated (`int-matches`); and (iii) a new cost metric we call intersection-cost (`i-cost`), which estimates the total sizes of the adjacency lists intersected during the evaluation of a plan. We use `i-cost` as our default metric of choice in our implementation. On several small queries and several real world data sets, we demonstrate that our greedy optimizer finds optimal (or close to optimal) plans in terms of `i-cost` and also run-time in practice.

We then describe one optimization we call *expanded DSQ optimization* to improve the performance of combined plans. We show that the original DSQs of a CSQ can be algebraically further expanded such that only edges with δ and o labels exist on DSQ edges. This sometimes allows more sharing opportunities that the optimizer can take advantage of to reduce the cost of the combined plan.

1.3 The Graphflow Database and Evaluations

In addition to studying how to optimize multiple CSQs in the DSQ framework, the second major contribution of this thesis is the implementation of a prototype active graph database called *Graphflow* [10]. Graphflow supports the property graph data model and a subset of Neo4j's declarative Cypher language. To support CSQs we extend Cypher with subgraph-condition-action triggers and refer to this extended language as the `Cypher++`. The initial version of the system was developed by the author and several colleagues. We give an overview of this initial version of the system in Chapter 5.

We implemented our optimizer and the expanded DSQ optimization on top of Graphflow. On several query sets and input graphs, we show that the combined plans generated with `i-cost` yield up to 2x performance improvement over a baseline which orders and evaluates DSQs separately without combining. We also study the performances of the combined plans generated by using catalogues based on each of the different cost metrics we consider. We demonstrate that combined plans optimized for `int-matches` and `i-cost` generally perform significantly better than the `num-ops` metric. In addition, there are cases

where plans generated with `i-cost` outperform the plans generated with `int-matches`. We also show that the expanded DSQ optimization further improves runtime across several query sets and datasets. Finally, on single CSQs, we present experiments comparing our approach to the most efficient continuous subgraph query evaluation algorithm we are aware of called Turboflux [5]. We demonstrate that on structural queries, our approach surpasses that of Turboflux without using any extra memory. In contrast, Turboflux can require extra memory for its auxiliary data structure that can be much larger than the input graph, limiting its scalability.

1.4 Contributions of the Thesis

We make the following contributions in this thesis.

- We describe a general greedy optimizer that takes multiple CSQs and generates a combined plan that can evaluate all of them together. Our optimizer effectively decomposes each query into multiple DSQs, and for each one picks a query vertex ordering. An important component of our optimizer is a subgraph extension catalogue, which contains cost and extension-rate information for extending a subgraph S to another subgraph S', where S' extend S by one query vertex.
- Given a catalogue, we formally define the optimization problem that our optimizer has to solve. We show that for arbitrary input catalogues, this problem is NP-hard.
- We analyze three important effects of query vertex orderings on run time: (i) computation sharing across queries; (ii) amount of intermediate results generated; and (iii) the amount of intersection work done during the evaluation of queries.
- We study three different natural cost metrics that can be optimized by our greedy optimizer: (i) number of operators (`num-ops`); (ii) number of intermediate results generated (`int-matches`); and (iii) sizes of adjacency lists intersected (`i-cost`). Each of these metrics tries to capture subsets of the three run-time effects of query vertex orderings. We show that `i-cost`, which captures all of the three effects together, generally performs the best.
- We introduce the expanded DSQ optimization that further decomposes the DSQs of each CSQ and allows for more computation sharing.
- We describe the overall architecture of the Graphflow prototype graph database on top of which we implemented our optimizer and the expanded DSQ optimization.

We demonstrate the effectiveness of our greedy optimizer, the `i-cost` metric, and the expanded DSQ optimization on several workloads, data sets, and query sets.

1.5 Outline of Thesis

The rest of this thesis is organized as follows:

- Chapter 2 provides background on the algebraic DSQ decomposition of CSQs, the vertex at a time evaluation of DSQs, and the logical and physical plans we generate for each DSQ. We also review alternative evaluation techniques for DSQs to discuss why we use vertex at a time evaluation.
- Chapter 3 describes the run time effects of query vertex ordering selection for DSQs.
- Chapter 4 describes our greedy cost-based optimizer that produces a single combined plan for a set of CSQs. We show how our optimizer can be used to optimize three different cost metrics and justifies why we pick `i-cost` as our default cost metric in Graphflow. We also describe our expanded DSQ optimization.
- Chapter 5 gives an overview of the Graphflow system and our implementation on our optimizer.
- Chapter 6 presents experimental evaluation of our optimizer and expanded DSQ optimization.
- Chapters 7 and 8 review related work and conclude, respectively.

Chapter 2

Background and Problem Statement

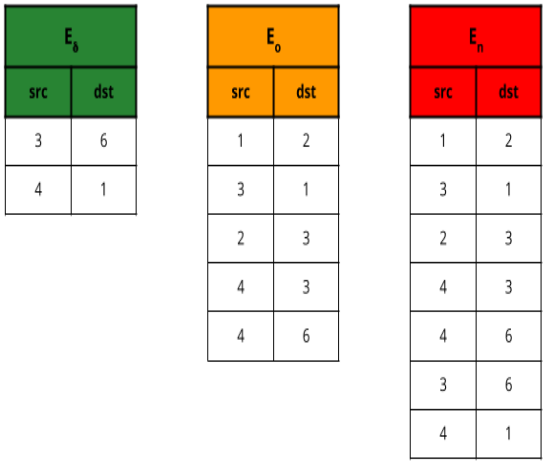
In this section we first draw the connection between the problem of matching CSQs and incremental view maintenance (IVM). Then, we describe the DSQ framework for evaluating CSQs in more detail than in Chapter 1. Finally, we review three alternative strategies available for evaluating DSQs; (i) the *query-vertex-at-a-time* evaluation adopted by Delta-GJ algorithm; (ii) *query-edge-at-a-time* evaluation; and (iii) bushy evaluation. We discuss why we pick the query-vertex-at-a-time approach in our setting.

2.1 CSQs as IVM

A subgraph query is a directed graph $Q(V_Q, E_Q)$ and evaluating Q on an input graph $G(V, E)$ is the task of finding all instances of Q in G. In this work, we primarily focus on queries without labels or filters on vertices and edges. We adopt the relational view of subgraph queries as done by many previous work [11, 12, 13], in which any subgraph query can be seen as a multiway self join on replicas of an edge table of the input graph. We label the vertices of Q as $a_1, a_2, ..., a_m$. Finding all instances of Q in G is equivalent to the multiway join of tables $E(a_{i_1}, a_{i_2})$ for each edge (a_{i_1}, a_{i_2}) in Q, where each $E(a_{i_1}, a_{i_2})$ table contains each edge (u, v) in G. For example, the directed asymmetric triangle query, shown in Figure 2.1b, in relational syntax is equivalent to:

$$Q(a_1, a_2, a_3) = E(a_1, a_2) \bowtie E(a_2, a_3) \bowtie E(a_1, a_3) \tag{2.1}$$

Continuously evaluating Q is the task of finding all instances of Q that emerge or are deleted in an evolving graph. Specifically, we assume G receives a series of updates

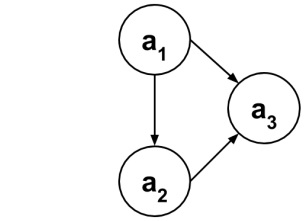


E_δ	
src	dst
3	6
4	1

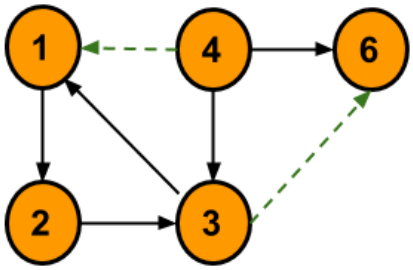
E_o	
src	dst
1	2
3	1
2	3
4	3
4	6

E_n	
src	dst
1	2
3	1
2	3
4	3
4	6
3	6
4	1

(a) E_δ, E_o and E_n tables for input graph



(b) Query graph



(c) Input graph with delta edge

Figure 2.1: DSQ evaluation example

$E_{\delta_1}, E_{\delta_2}, \ldots$ where each E_{δ_i} is a batch of additions and/or deletions of edges. A very important assumption in this thesis is that we assume the batch sizes are small, say several edges, compared to existing edges in the input graph. Given the equivalence of subgraph queries and multiway joins, CSQs are equivalent to IVM of join queries. Let E_δ, E_o, and E_n be defined as before. Then, for example the new triangles that emerge or are deleted can be expressed in relational terms exactly as:

$$\begin{aligned} dQ(a_1, a_2, a_3) =& (E_n(a_1, a_2) \bowtie E_n(a_2, a_3) \bowtie E_n(a_1, a_3)) \; - \\ & (E_o(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_o(a_1, a_3)) \end{aligned} \tag{2.2}$$

We assume added and deleted edges in E_δ are identified by $+/-$ labels, respectively. Emerged and deleted output tuples in dQ are identified similarly.

2.2 Delta Query Framework for Evaluating IVM

Let $n = |E_Q|$. Blakeley et al. has shown that the union of the results from the following n *delta queries* is equivalent to the results of Equation 2.2 [9]. Throughout this thesis we

refer to delta queries as delta subgraph queries (DSQs). When referring to specific DSQs, we use the notation dSQ_i

$$\begin{aligned} dSQ_1 &= E_\delta(a_{11}, a_{12}) \bowtie E_o(a_{21}, a_{22}) \bowtie E_o(a_{31}, a_{32}), \ldots, \bowtie E_o(a_{n1}, a_{n2}) \\ dSQ_2 &= E_n(a_{11}, a_{12}) \bowtie E_\delta(a_{21}, a_{22}) \bowtie E_o(a_{31}, a_{32}), \ldots, \bowtie E_o(a_{n1}, a_{n2}) \\ &\ldots \\ dSQ_n &= E_n(a_{11}, a_{12}) \bowtie E_n(a_{21}, a_{22}) \bowtie E_n(a_{31}, a_{32}), \ldots, \bowtie E_\delta(a_{n1}, a_{n2}) \end{aligned} \tag{2.3}$$

For example, the DSQs of the asymmetric triangle subgraph query are:

$$\begin{aligned} dSQ_1(a_1, a_2, a_3) &= E_\delta(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_o(a_1, a_3) \\ dSQ_2(a_1, a_2, a_3) &= E_n(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_o(a_1, a_3) \\ dSQ_3(a_1, a_2, a_3) &= E_n(a_1, a_2) \bowtie E_n(a_2, a_3) \bowtie E_\delta(a_1, a_3) \end{aligned} \tag{2.4}$$

Note that each dSQ_i has one E_δ, which we assume contains a very small number of tuples. This allows each dSQ_i to be efficiently evaluated.

Figure 2.1 provides an example input graph, two edges (3, 6) and (4, 1) that are inserted into this graph, and the E_δ, E_o and E_n that result upon this insertion. The reader can verify that dSQ_2 detects the emergence of the triangle between vertices 3, 4 and 6, while dSQ_3 detects the emerged triangle between vertices 4, 1 and 3. dSQ_1 does not find any matches.

We note that there can be multiple different DSQ decompositions of a query. For example an alternative decomposition of the triangle query could be: $dQ_1 = E_o(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_\delta(a_3, a_1)$, $dQ_2 = E_o(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_n(a_3, a_1)$, and $dQ_3 = E_\delta(a_1, a_2) \bowtie E_n(a_2, a_3) \bowtie E_n(a_3, a_1)$, which gives a different set of queries. In practice, for a given query we saw little performance difference between different decompositions, mainly because there is very little difference between E_n and E_o relations as we assume that E_δ is very small. So, we choose the decomposition that puts the delta relations in lexicographic order of the attributes of the relations.

2.2.1 Delta Subgraph Query Notation

We will use labeled graphs to visually represent DSQs. We label each edge in a DSQ with δ, o, or n indicating whether the edge should match an edge in E_δ, E_o or E_n. We will refer to a query edge labeled with δ, o, and n as *delta*, *old*, and *new query edge*, respectively. For example, Figure 2.2 shows the second DSQ of the triangle query.

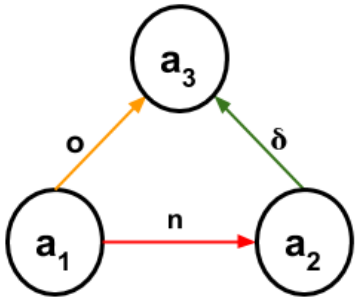


Figure 2.2: Delta Query Notation.

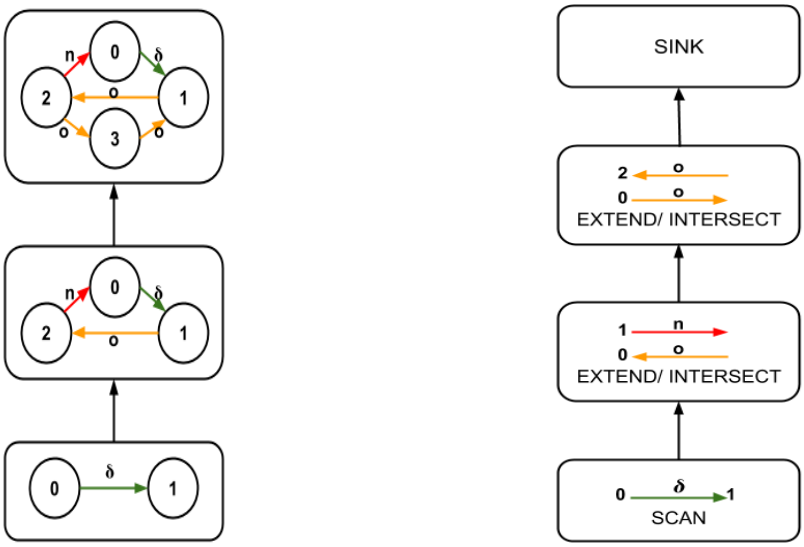
2.3 DSQ Evaluation

We divide the strategies for evaluating DSQs broadly into three categories. (i) *query-vertex-at-a-time* strategy adopted by the Delta-GJ algorithm which iteratively finds matches for subsets of query vertices; (ii) *query-edge-at-a-time* strategy which starts from a single edge and in each iteration extends partial matches by one more edge; and (iii) *bushy evaluation* which matches and then joins subsets of the edges or vertices recursively. We first describe in detail the query-vertex-at-a-time approach we use to evaluate DSQs, then go on to discuss the shortcomings of the other two strategies in our setting.

2.3.1 Query-vertex-at-a-time Strategy: Delta-GJ Algorithm

We gave a brief overview of Delta-GJ algorithm in Section 1.2. We give a more detailed overview here. Let $dQ_k(V_Q, E_{dQ_k})$ be the kth DSQ of Q. First, the algorithm picks a *query vertex ordering (QVO)* for dQ_k, starting with the two query vertices forming the delta query edge. For simplicity, let (a_1, a_2) correspond to the delta query edge and let the QVO be $a_1 a_2 \dots a_m$. We start evaluation of dQ_k with the edges in E_δ, which match (a_1, a_2) in dQ_k, giving us an initial small set of *partial matches*. We extend these partial matches one query vertex at a time until we match the full dQ_k. Specifically, we construct $m-1$ sets of partial matches, where the j'th set contains the partial matches of dQ_k projected onto the first j query vertices and all of the query edges between them. We extend a partial match $t = (v_1, v_2, ..., v_j)$ matching query vertices $(a_1, ..., a_j)$ to partial matches matching $(a_1, ..., a_{j+1})$ as follows.

1. For each v_i which matches a_i where (a_i, a_{j+1}) or (a_{j+1}, a_i) is a query edge e_q in dQ_k, we take v_i's forward or backward adjacency list, respectively. We take v_i's adjacency list



(a) Logical Plan: QVO $a_2a_4a_1a_3$.

(b) Physical Plan.

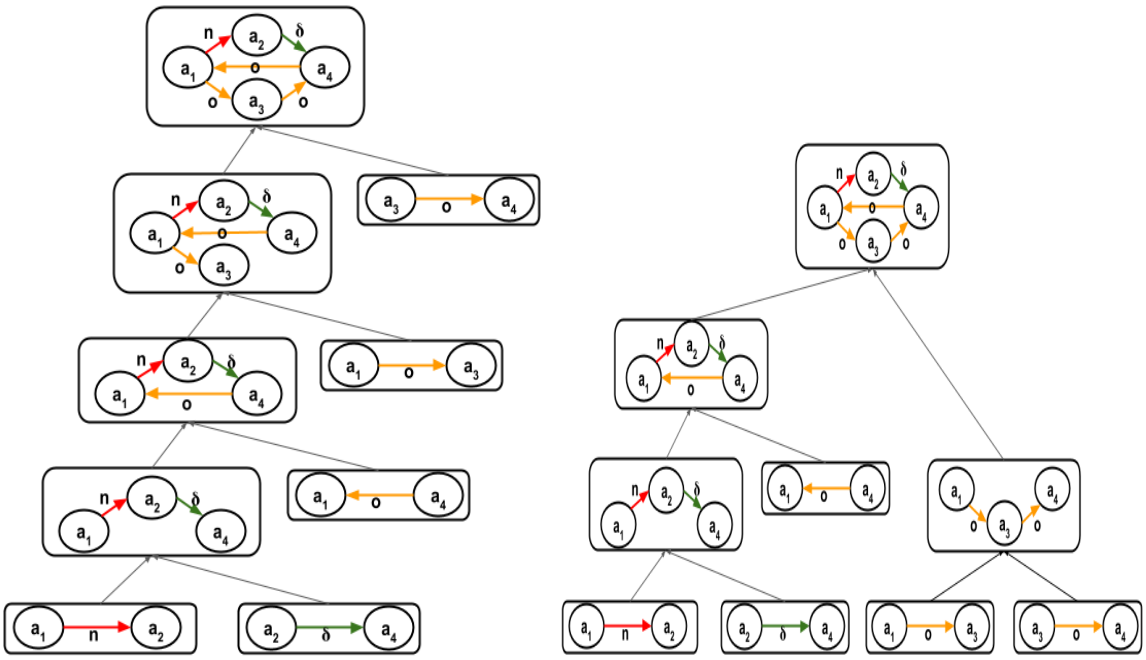
Figure 2.3: Query-vertex-at-a-time evaluation of a single DSQ.

from E_o or E_n if e_q is an old or new query edge, respectively. Let $L = [Adj_1, \ldots, Adj_p]$ be the adjacency lists chosen. We intersect these adjacency lists to get a set of vertices S matching a_{i+1}. In the special case where $|L| = 1$, S equals Adj_1.

2. t is extended to $t \times S$, i.e., the Cartesian product of t with S.

Recall that we assume E_δ has a low cardinality. Using E_δ as the first set of partial matches reduces the number of partial matches significantly. Figure 2.3a shows an example of a logical plan for query-vertex-at-a-time evaluation of the second DSQ of the CSQ in Figure 1.1b when the QVO is $a_2a_4a_1a_3$.

Reference [3] has called the above algorithm Delta-GJ because it combines two techniques; (i) the delta query decomposition of join queries for IVM; and (ii) the worst-case optimal GJ algorithm introduced by Ngo et al. in reference [14], which evaluates join queries one attribute-at-a-time. Reference [3] has also shown that for a given CSQ under insertion-only updates, Delta-GJ is worst-case optimal. However neither reference [3] nor [14] study and give guidance as to how to pick a good QVO for DSQs. The ordering of DSQs is studied extensively in this thesis. Finally, in reference [10] the author of this thesis and his colleagues have shown that for several simple CSQs the Delta-GJ algorithm



(a) Edge-at-a-time plan (b) Bushy plan

Figure 2.4: Alternative Plans For DSQ Execution

is more efficient than evaluating the DSQs by binary joins, which is equivalent to the query-edge-at-a-time strategy we review next.

2.3.2 Alternative DSQ Evaluation Strategies

We discuss two natural alternative approaches and argue why they will be less efficient than the query-vertex-at-a-time approach we adopt in this thesis.

- *Query-edge-at-a-time evaluation:* In the case of a DSQ, the query-edge-at-a-time approach starts with the delta edge to take advantage of its low cardinality, orders the rest of the edges, and matches the query one query edge at a time according to this ordering. Figure 2.4a shows a logical plan for a query-edge-at-a-time strategy. On the second DSQ of the `diamond-with-edge` query shown in Figure 1.1b, the evaluation first finds matches for the open triangles between the delta query edge (a_2, a_4) and new query edge (a_1, a_2) and then closes these triangles. Evaluation then extends these triangles by one edge to a_3 and then closes the bottom triangle by matching

(a_3, a_4). Instead, our query-vertex-at-a-time plan would perform a single intersection operation starting from the delta query edge (a_2, a_4), avoiding the explicit construction of any open triangles. In general, query-vertex-at-a-time plans are more efficient than query-edge-at-a-time plans because they perform intersections in a single step, instead of breaking them into multiple steps.

- *Bushy evaluation:* Query-vertex-at-a-time (and query-edge-at-a-time) plans effectively match a DSQ starting from the single delta query edge and grow a single connected subgraph. Instead, in a bushy plan one breaks the DSQ into multiple subgraphs and later join them. Individual subgraphs can be evaluated using either of the previously discussed methods. Consider dSQ_2 in Figure 1.1b. For this DSQ, one can match the upper (a_2, a_4, a_1) triangle and separately match the lower (a_1, a_3, a_4) open triangle and then later join these two intermediate results. However, because the DSQ contains only one delta query edge, one branch of a bushy plan would contain only old or new query edges, which we expect to have large cardinalities. For instance, the bushy plan in our example computes all of the (a_2, a_4, a_1) triangles in the entire graph, which we expect to be very large. Re-evaluating such branches of the bushy plan after each update to the input graph would be prohibitively inefficient. One can alternatively materialize such branches to avoid re-evaluation. However, this would result in storing possibly a very large set of intermediate partial matches, consuming a lot of memory. Instead, query-vertex-at-a-time and query-edge-at-a-time plans do not maintain any partial matches and, therefore, require a very low amount of memory. Ammar et al. in introducing Delta-GJ [3] show that such bushy plans that materialize some branches can still be used in the DSQ framework to trade off memory for run-time efficiency. In this thesis, we do not study approaches that materialize large amounts of memory.

2.4 Logical Plans And Physical Plans

The QVO for a DSQ is effectively a logical plan. The translation of QVOs into physical plans is straightforward. The first delta edge corresponding to the first two attributes in the combined QVO is mapped to a SCAN operator. The partial match extension into each possible attribute gets mapped to an EXTEND/INTERSECT (E/I) operator. Let *prefix j-tuple* be a tuple that contains j attributes. The SCAN and INTERSECT operators perform the following computations.

- SCAN: Scans the updates in E_δ and appends these as prefix 2-tuples to its child

operators.

- E/I: Takes as input prefix j-tuples and extends each tuple t to one or more (j+1)-tuples. The operator is configured with one or more *adjacency list descriptors (ALDs)*, describing which adjacency lists of the vertices in t to intersect. Let $t[i]$ be the value of the i^{th} vertex of t. Each ALD is an $(i, dir, vrsn)$ triple, where i is the index of a vertex in t, dir is forward or backward, and $vrsn$, for *version*, is `delta`, `old`, or `new`. The operator takes the adjacency lists described by these ALDs, intersects them to compute t's extension set S, and computes $t \times S$. Intersections are performed in tandem. When there are more than two adjacency lists, we do pairwise binary intersections to intersect all of them. The resulting prefix $j+1$-tuples are appended to the next operator.

There is also a SINK operator, which collects and unions the outputs of possibly multiple E/I operators, which we do not discuss for simplicity. Figure 2.3b gives an example of the physical plan for the logical plan shown in Figure 2.3a corresponding to QVO $a_2a_4a_1a_3$ for the second DSQ of the `diamond-with-edge` query. As in Figure 2.3a, Figure 2.3b shows the ALDs as labeled edges, where forward and backward adjacency lists point right and left, respectively.

In both the logical and physical plans for DSQs, we use the indices i, e.g. 0,1,2,3 of the tuples that are produced by the operators, instead of query vertex labels $a_1 \dots a_m$. This is done to help readers see operators from multiple logical plans that produce isomorphic partial matches, whose computation we share in combined plans our greedy optimizer produces (See Section 3.1).

Chapter 3

Runtime Effects of QVOs

In the previous Chapter we discussed the DSQ framework and indicated that it will form the basis for our solution to the general problem set out in Chapter 1. Using that background, we are now able to state the problem we are studying in a more specific manner.

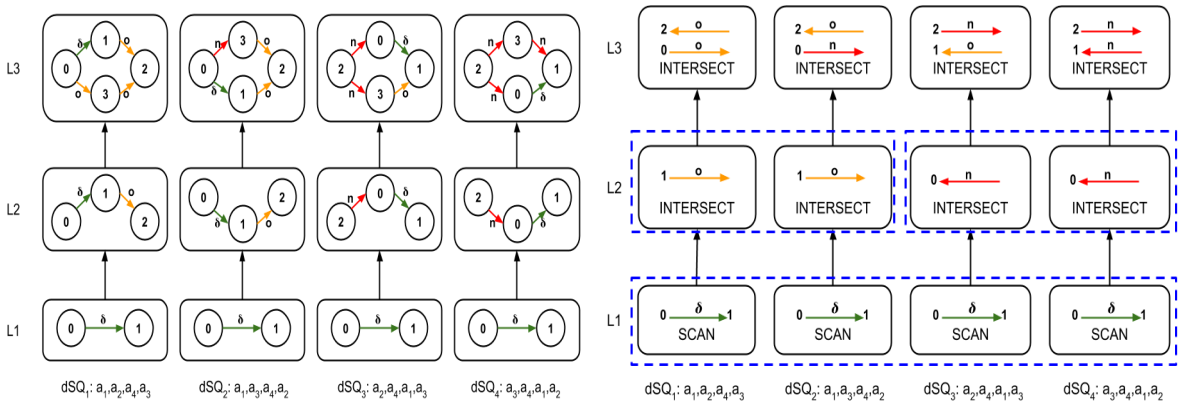
Given a set of queries $\bar{Q}$ which expand into DSQs $\bar{D}$, find QVOs for each $D \in \bar{D}$ such that for a set of updates $E_{\delta_1}, ..., E_{\delta_k}$ on graph G, the cumulative time to evaluate $\bar{Q}$ is minimized.

This version of the problem statement is still underdefined as runtime is not a formal cost metric. In this Chapter, we first review how QVOs affect runtime. In the next Chapter, we will define our final formal optimization problem.

QVOs affect runtime in our framework in two broad ways: (i) by determining how much computation can be shared between DSQ evaluations; and (ii) by determining the amount of intersection work done in the E/I operators, which further depend on the number of intermediate prefixes the E/I operators generate and the directions of the adjacency lists being intersected. We review each of these effects next. When discussing the intersection work effects, we also define the `i-cost` metric which we use in our optimizer in Chapter 4.

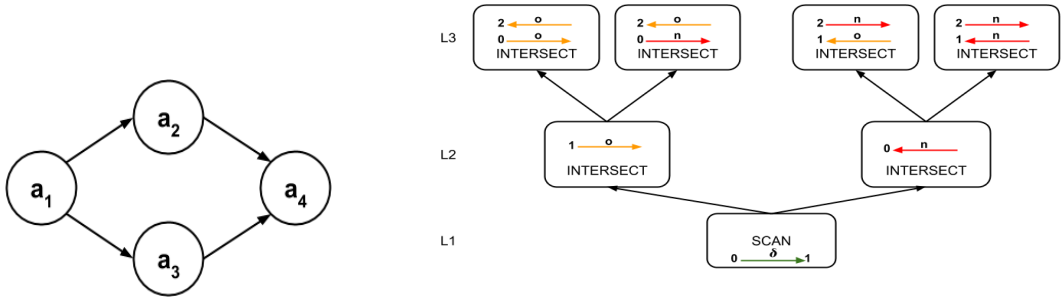
3.1 Shared Computation Between DSQs

When evaluating multiple DSQs at the same time, there might be opportunities to share computation between the physical plans of the DSQs. Due to these opportunities, instead of evaluating each DSQ separately, we evaluate all of them together using a *combined plan*.



(a) Individual Logical Plans.

(b) Individual Physical Plans.



(c) Diamond CSQ

(d) Combined Physical Plan.

Figure 3.1: Differences in sharing opportunities of different QVOs for the DSQs of a diamond query

.

In Figure 3.1 we give an example of a set of DSQs with opportunities for computation sharing. Figure 3.1a shows the logical plans of the DSQs of the *diamond* query, shown in Figure 3.1c for QVOs, $a_1a_2a_4a_3$, $a_1a_3a_4a_2$, $a_2a_4a_1a_3$ and $a_3a_4a_1a_2$. Figure 3.1b shows the corresponding physical plans with each set of operators which can share computation surrounded by dashed lines. First, all the SCAN operators perform the same operation, i.e., read the set of edges in E_δ. Second, the second stage E/I operators of dSQ_1 and dSQ_2 as well as dSQ_3 and dSQ_4, can also share computation. This is because they find partial matches for isomorphic subgraphs of their respective DSQs, as shown in the logical plans in Figure 3.1a. We construct the combined plan shown in Figure 3.1d for these physical plans, where each similar operator is evaluated only once.

If we were to pick a different QVO for one of the DSQs, the number of operators that can be shared may change. For instance, picking QVO $a_3a_4a_2a_1$ for dSQ_4 means no operators can be shared between dSQ_3 and dSQ_4, which adds an additional operator to the combined plan. Therefore, QVO $a_3a_4a_2a_1$ is a suboptimal ordering compared to the original set of QVOs. Sharing computation is also possible between DSQs from different CSQs, as long as they have isomorphic subgraphs which the QVOs can exploit.

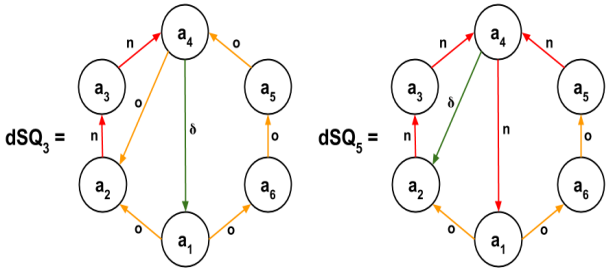
3.2 Intersection Work and the `I-cost` Metric

In Section 2.4 we showed that a QVO of a DSQ translates into a physical plan consisting of one SCAN operator and a series of E/I operators. Since SCAN consumes relatively little time, much of the evaluation time of a DSQ is spent primarily in the E/I operators, which take as input a set of prefixes and perform the intersections indicated by their ALDs. Therefore, the amount of time for evaluating a plan is commensurate with the cumulative intersection work done in its E/I operators. Recall from Chapter 2 that intersections in our implementation are performed in tandem. Therefore, we quantify the intersection work done by an operator or plan as the sum of the lengths of all the adjacency lists it intersects. We call this quantity the *intersection-cost* (`i-cost`) of an operator or plan.[1]

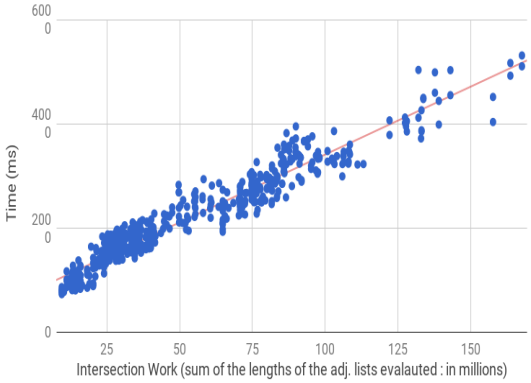
3.2.1 `I-cost` vs Runtime

To demonstrate that `i-cost` is a good indicator of run-time when evaluating DSQs, we considered evaluating each of the 576 possible QVO combinations of the two DSQs shown in Figure 3.2a in an experiment. In the experiment, we formed 576 different combined plans for the two DSQs corresponding to the 576 different QVO combinations. Then, for each combined plan we performed the following evaluation. We took a product co-purchasing network called Amazon0302 from reference [15] with 1.2 million edges, loaded a random 90% of the dataset to the database and submitted the remaining 10% as insert-only updates in batches of 5 edges and measured the cumulative run-time of the query across all of the updates. We measured the actual `i-cost` in a separate profiled run where we summed the sizes of the actual adjacency lists in the E/I operators. Figure 3.2b shows on the y-axis the run time and on the x-axis the actual `i-cost` for each QVO combination. Each QVO combination is shown as a single point in the figure. Note that there is a nearly-linear

[1] In an ongoing work, the author and his colleagues use the `i-cost` metric to order one time subgraph queries.



(a) 2 DSQs with varying QVOs.



(b) Intersection work vs. time for different combined plans.

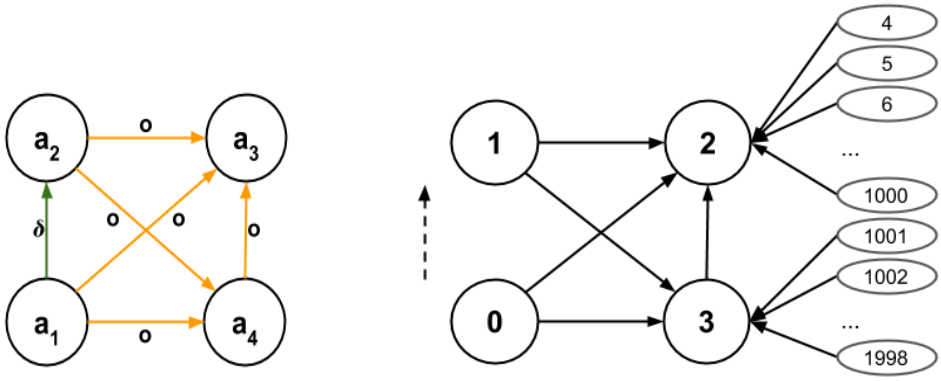
Figure 3.2: Dependence of runtime on intersection work performed during query evaluation.

relationship between run-time and `i-cost`. We highlight this in the figure by drawing the best fit line which minimizes the squared error. Figure 3.2b also shows the importance of picking good QVOs. As shown in the figure, the difference between good and bad orderings can be very large, roughly 8x in this simple experiment. Due to this strong dependency between `i-cost` and run-time in combined plans, in Chapter 4 we will adopt `i-cost` as the default unit of cost in our cost-based greedy optimizer.

3.2.2 Factors Affecting `I-cost`

There are two main factors that determine `i-cost` of a DSQ plan: (i) the direction and number of ALDs in each E/I operator; and (ii) The number of intermediate prefix tuples generated. We give brief outlines of each below.

- *Number and directions of ALDs:* Not surprisingly, given the same number of input prefixes, and ALDs of the same type, an operator with more ALDs will have higher `i-cost`. However, surprisingly, given the same input prefixes, operators with different ALD directions may have different `i-costs`. This behaviour is observed due to the structure of the graph, particularly the distribution of the sizes of the forward and backward adjacency lists of vertices. We demonstrate this with a toy DSQ and query graph in Figure 3.3. The input graph in the figure has two nodes each with 1000 incoming edges, and a single edge insertion (the edge $(0, 1)$). We consider two QVOs for the DSQ in Figure 3.3a, $QVO_1 : a_1a_2a_3a_4$ and $QVO_2 : a_1a_2a_4a_3$. In each case,



(a) 4-clique DSQ (b) Input Graph

Figure 3.3: Example for demonstrating the effect of different ALDs on `i-cost`.

we start by matching the δ edge (a_1, a_2) to $(0, 1)$ and intersect the forward adjacency lists of 0 and 1, which give 2 and 3. In QVO_1, 2 and 3 match a_3 and is extended by doing two intersections: (i) 0's forward adjacency list, 1's forward adjacency list, and 2's backward adjacency list; and (ii) 0's forward adjacency list, 1's forward adjacency list, and 3's backward adjacency list. In QVO_2, 2 and 3 match a_4 and is extended by two intersections, in which we use 0 and 1's forwards lists as in QVO_1 but instead use 2 and 3's forward lists. Note that 2 and 3's backward adjacency lists are of size 1000, whereas their forward adjacency lists are of size only 2. Therefore, QVO_2 will do significantly less work than QVO_1.

- *Number of intermediate prefixes:* The number of intermediate prefixes input to operators directly affects the intersection work they have to do. The more intermediate prefixes generated, the more intersections will be performed.

Chapter 4

Cost Based Greedy CSQ Optimization

In this Chapter we describe our greedy optimizer that generates a combined plan to evaluate a set of CSQs. The outline of this Chapter is as follows:

- Section 4.1 describes the catalogue data structure.
- Section 4.2 defines our optimization problem formally, which depends on the contents of the catalogue. We show that in its most generality, this optimization problem is NP-Complete.
- Section 4.3 describes our general greedy optimizer.
- Section 4.4 describes how to make our optimizer optimize for three different cost metrics: (i) number of operators in the combined plan; (ii) number of intermediate prefixes generated; and (iii) `i-cost`.
- Section 4.5 describes our expanded DSQ optimization.

Our optimizer takes two inputs: (a) a set of DSQs $\bar{Q}_{DSQ}$ of a set of CSQs $\bar{Q}$; and (b) a *subgraph extension catalogue*. The output is a combined plan CP with a single SCAN and (possibly) multiple E/I operators.

S	ALDs	S'	Cost (c)	Extension Rate (α)
$0 \rightarrow 1$	$1F$	$\bullet \rightarrow \bullet \rightarrow \bullet$	0.15	0.2
$0 \rightarrow 1$	$0B$	$\bullet \rightarrow \bullet \rightarrow \bullet$	0.2	0.2
$0 \rightarrow 1$	$1B$	$\bullet \rightarrow \bullet \leftarrow \bullet$	1.8	1.8
$0 \rightarrow 1$	$0F$	$\bullet \leftarrow \bullet \rightarrow \bullet$	3.0	3.0
$2 \leftarrow 0 \rightarrow 1$	$2B$	$\bullet \rightarrow \bullet \leftarrow \bullet \rightarrow \bullet$	1	1
$2 \rightarrow 1 \leftarrow 0$	$0F$	$\bullet \rightarrow \bullet \leftarrow \bullet \rightarrow \bullet$	1	1
$0 \rightarrow 1 \rightarrow 2$	$0F$, $2B$	Diamond	4	2
$0 \rightarrow 1 \leftarrow 2$	$0B$, $2B$	Diamond	0.4	0.1
$0 \leftarrow 1 \rightarrow 2$	$0F$, $2F$	Diamond	0.6	0.06

Table 4.1: Example Subgraph Catalogue.

4.1 Subgraph Extension Catalogue

Table 4.1 shows an example catalogue. Each entry of the catalogue contains two pieces of information about extending a subgraph S to S' using a particular set of ALDs A:

- Estimated cost c of extending a single S to S' using A according to some cost metric.
- Estimated *extension rate* α of this extension, i.e., how many S's will be generated from S. The extension rate is a cardinality estimate.

In Table 4.1, the vertices of each subgraph S have integer IDs, e.g., 0, 1 or 2, to differentiate between different ALD combinations that can extend S but we do not put IDs on S'. The IDs on S are put arbitrarily but consistently, i.e., we use the same IDs for multiple instances of S. We also note that the same S can be extended to S' using different ALDs with different costs depending on the cost metric. The first two entries in Table 4.1 demonstrate this possibility. We review a more realistic example in Section 4.4. One can consider more complex catalogues with δ, o, and n labels on S and S'. For the expanded DSQ optimization we describe in 4.5, we explore a catalogue with δ and o labels on the edges of S. When optimizing compact DSQs, we do not differentiate between δ, o, and n edge labels because for our purposes, each entry in the catalogue is an estimate for the work and extension rate of extending "an average" S to S'. Therefore we do not store information about how S was generated (for example which edge of S is the delta edge). Finally, the number of entries in the catalogue can be prohibitively large. To tackle this problem, we construct partial catalogues that contain only the subgraphs that are relevant for a particular query workload $\bar{Q}$. We discuss partial catalogue construction in Section 5.3.2.

4.2 Specific Problem Definition

We are now ready to define the specific optimization problem we will be solving using the parameters defined in the earlier section.

Given a set of DSQs $\bar{D}$ derived by decomposing a set of CSQs $\bar{Q}$, and a subgraph extension catalogue C, find QVOs for $D \in \bar{D}$ such that the resulting combined plan, where we merge the common operators across the individual plans for DSQs, has the least possible cost according to C, where the cost of C is the sum of the costs of each E/I *operator op_i in C and the cost of op_i is given as:*

$$cost(op_i) : (\pi_{k=1}^{k=i-1}\alpha_k)c_i \tag{4.1}$$

That is, the cost of each E/O operator is the estimated number of partial subgraphs it will process multiplied by the estimated cost of processing each partial subgraph. We next analyze the size of the solution space to this optimization problem and its computational complexity.

4.2.1 Solution Space

Given a CSQ Q_i, with n_i query vertices and m_i query edges, any DSQ of the CSQ has $(n_i - 2)!$ possible QVOs. This is because the first two query vertices are always mapped to the delta query edge in the DSQ, and any permutation of the other $n_i - 2$ query vertices is a possible QVO. Given k CSQs, the size of the possible plan space in our setting is $((n_1{-}2)!)^{m_1} \times ((n_2{-}2)!)^{m_2} \times \ldots \times ((n_k{-}2)!)^{m_k}$, since each CSQ Q_i decomposes into m_i many DSQs. This quantity is very large even for a small number of small size queries, so adopting an exhaustive search for picking a good plan cannot work. This leads us to the greedy optimization algorithm we describe in Section 4.3, to find good plans.

4.2.2 Computational Complexity

The computational complexity of this problem naturally depends on the contents of C. For example, if the cost and expansion rate of every entry in C is zero, then any combined plan will be optimal, so the problem is trivially solvable. Similarly, the problem also depends on $\bar{Q}$. However, for an arbitrary catalogue C and a $\bar{Q}$, the problem is NP-Complete. Specifically, for the catalogue that has a value of 1 for each cost and expansion rate, the optimal combined plan is the plan that contains the smallest number of operators. For an

arbitrary $\bar{Q}$, we next show that the maximum common induced subgraph problem [16], which is NP-Complete, reduces to finding the plan with the minimum number of operators.

The reduction is as follows. Take two arbitrary graphs G_1 and G_2. Suppose the nodes in G_1 are labeled with $a_1, a_2, ..., a_{m_1}$ and each node in G_2 is labeled with $b_1, b_2, ..., b_{m_2}$. Label each edge of G_1 and G_2 with o, and extend both G_1 and G_2 with a new edge (x, y) with label δ and connect 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 DSQs, and we refer to them as dSQ_1 and dSQ_2. Recall that we get a combined plan by picking a QVO for each DSQ and then overlapping the common operators. So the combined plan will start with the (x, y) edge, contain a chain of t operators that are common to both of the QVOs for dSQ_1 and dSQ_2, and at some point split into two branches, where one branch will contain $m_1 - t$ and the other branch will contain $m_2 - t$ operators (and these branches will never converge again). So the total number of E/I operators will be $m_1 + m_2 - t$. Note also that the last operator in the chain will be a subgraph that is common to both G_1 and G_2 with exactly t vertices (ignoring the x and y vertices that we artificially added). Therefore the optimal combined plan, when all the entries in the catalogue are 1, effectively computes the maximum common induced subgraph to G_1 and G_2, completing the proof.

4.3 Greedy Optimization Algorithm

Algorithm 1 provides an overview of our greedy optimization algorithm. We start with an empty combined plan CP. In each iteration, the algorithm goes through each QVO of each DSQ in $\bar{Q}_{DSQ}$ and finds the pair $<qvo^*, dsq^*>$ with the minimum additional cost to CP. This fixes the QVO of dsq^* to be qvo^* and we merge the plan P^* induced by $<qvo^*, dsq^*>$ to CP and remove dsq^* from $\bar{Q}_{DSQ}$. We repeat this until $\bar{Q}_{DSQ}$ is empty. Note that if there are multiple $<qvo^*, dsq^*>$ pairs with identical minimum cost, we randomly pick one of the $<qvo^*, dsq^*>$ pairs with minimum cost.

The additional cost of a $<qvo, dsq>$ pair is computed as follows. We first compute the logical plan P induced by $<qvo, dsq>$. Recall that P is a linear plan that starts with Scan followed by a series of E/I operators. Then we overlap the maximum prefix of P with CP. This is done by simply checking for each operator op_i in P starting with the first E/I operator, whether an existing operator in CP has the same ALDs. The suffix operators of P that do not overlap CP must be added as new operators to CP, so the additional cost of $<qvo, dsq>$ is exactly the sum of the costs of these operators. Let op_i have $op_1 ... op_{i-1}$ as a prefix (we assume Scan is op_0 and we ignore it). Each op_k extends a subgraph S_{op_k} to S'_{op_k} and has an estimated cost c_k and extension rate α_k according to the catalogue C.

Algorithm 1 Greedy Optimization Algorithm

Input: $\bar{Q}_{DSQ}$, catalogue C

combinedPlan $= \emptyset$
$minDSQ = \emptyset$
minCost $= \infty$
while $\bar{Q}_{DSQ} \neq \emptyset$ **do**
  for $dsq^* \in \bar{Q}_{DSQ}$ **do**
    for $qvo^* \in dsq*$ **do**
      $cost = FindAdditionalCost(< qvo^*, dsq^* >, C)$
      if $cost < minCost$ **then**
        $minDSQ =< qvo^*, dsq^* >$
      end if
    end for
  end for
  $P* = CreatePlan(minDSQ)$
  $Merge(combinedPlan, P*)$
  $\bar{Q}_{DSQ}.remove(minDSQ)$
end while
return *combinedPlan*

Algorithm 2 FindAdditionalCost

Input: $< qvo, dsq >$, catalogue C, *combinedPlan*

previousOperator $= \emptyset$
$P = getLogicalPlan(< qvo, dsq >)$
for *operator* in P **do**
  if *operator* not in *combinedPlan* **then**
    return $calculateCostIncludingDescendants(operator) -$ calculateCost(previousOperator)
  end if
  $previousOperator = operator$
end for
return 0

The cost of op_i is calculated by using Equation 4.1. Recall that the cost of an operator op_i is the estimated number of S partial matches op_i will get times the estimated work c_i that will be do on a single S. This computation requires looking up the subgraphs S_k and the set of ALDs of $op_1, ..., op_{i-1}$ in C. We do this in a brute force fashion by checking for all

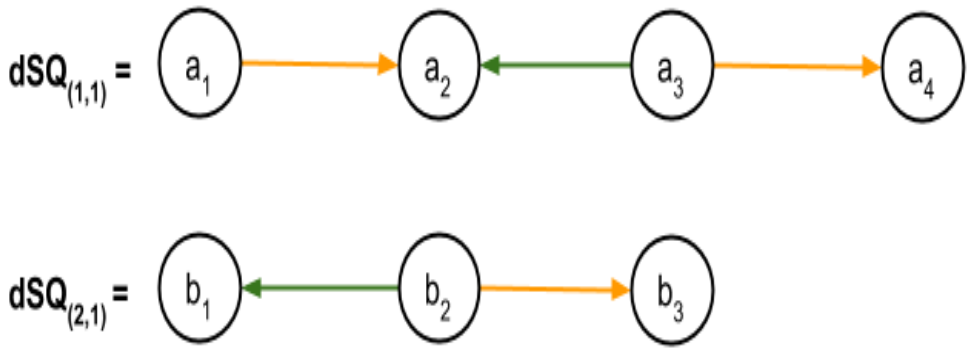


Figure 4.1: Two DSQs for evaluation

possible integer labelings of S_k and checking if it makes S_k isomorphic to any S in C.

Our optimizer is very efficient in its computational time. Specifically, the number of catalogue lookups it makes is $O((\Sigma_{Q_{DSQ} \in \bar{Q}_{DSQ}} \#\text{VQOs of } Q_{DSQ})^2)$, so quadratic in the sum of the different VQOs of all the DSQs in $\bar{Q}$. This quantity, for all of the workloads we consider in this paper is in the order of several thousands. In all of our query sets, our optimization has taken less than 2 seconds. The more expensive part of our optimization phase is the catalogue construction which we describe in Section 5.3.

Example 4.3.1 *Suppose $\bar{Q}_{DSQ}$ contains the 4 DSQs of the diamond query given in Figure 3.1c and the catalogue from Table 4.1. Our optimizer will consider all of the 8 possible $<VQO, DSQ>$ pairs of the 4 DSQs and would pick $<a_1a_2a_4a_3, dSQ_1>$ or $<a_1a_3a_4a_2, dSQ_2>$, both of which have a cost of $0.15 + 0.2 * 4 = 0.95$. Suppose it picks $<a_1a_2a_4a_3, dSQ_1>$, so CP contains the plan induced by this pair, shown as the first individual plan in Figure 3.1a. Then it picks $<a_1a_3a_4a_2, dSQ_2>$, which now has a cost of 0.8 as its first* E/I *operator would be shared with the first* E/I *operator of the current CP. Then, it would pick one of $<a_2a_4a_1a_3, dSQ_3>$ or $<a_3a_4a_1a_2, dSQ_4>$, both having $0.2+0.2*4 = 1$ cost and then the other, again sharing the first* E/I *operators of the plans induced by these picks. The final CP returned by the optimizer would be the plan in Figure 3.1d.*

Readers might suspect that one can use a dynamic programming based optimization algorithm for this problem. Dynamic programming is an often used paradigm to find good join plans in relational systems during optimization. However this approach will not work because an optimal combined plan to a subset of $\bar{Q}_{DSQ}$ is not necessarily part of an optimal combined plan to $\bar{Q}_{DSQ}$. We give an example of this below.

Example 4.3.2 *Consider the problem of creating a combined plan using the two DSQs in Figure 4.1. Suppose we first try to minimize the cost for* $dSQ_{(1,1)}$*. We consider the possible QVOs* $a_3a_2a_4a_1$ *and* $a_3a_2a_1a_4$*. Using the catalogue in Table 4.1, we see that these QVOs have costs of* $1.8 + 1.8 * 1 = 3.6$ *and* $3 + 3 * 1 = 6$ *respectively, which suggests that the optimal QVO for* $dSQ_{(1,1)}$ *is* $a_3a_2a_1a_4$*. Since the only available QVO for* $dSQ_{(2,1)}$ *is* $b_2b_1b_3$*, with a cost of 3, we see that the combined plan for these orderings have a combined cost of 3.6. Now consider the alternative QVO pair* $a_3a_2a_1a_4$ *and* $b_2b_1b_3$ *for* $dSQ_{(1,1)}$ *and* $dSQ_{(2,1)}$*. As the subgraphs matched by* $a_3a_2a_1$ *and* $b_2b_1b_3$ *are isomorphic, we can share computation between the two DSQs. Therefore, the combined cost given these QVOs is* $3 + 3 * 1 = 6$*, which is the optimal cost for these DSQs.*

4.4 Three Natural Cost Metrics

We next describe three different natural cost metrics that combined plans can be optimized for and how we can use different catalogues in our optimizer to optimize for each metric. We evaluate the effectiveness of each metric in 6.

4.4.1 Number of Operators (`num-ops`)

One of the most natural cost metrics to optimize for is to make the output combined plans as small as possible. Indeed, reference [7], which is the only existing work we are aware of that optimizes multiple continuous subgraph queries, constructs a data structures similar to our combined plan, with the goal of making these combined plans as small as possible. Our optimizer can optimize for this metric by simply setting each cost c and extension rate α of each entry of the catalogue to 1. This will make the cost of each additional operator op_i to be 1 (recall Equation 4.1).

4.4.2 Number of Intermediate Partial Prefixes (`int-matches`)

Another natural cost metric is the amount of intermediate partial matches the combined plan will generate, with the intuition that the smaller the number of partial matches, the less computation that will be performed in the combined plan. Our optimizer can optimize for this metric by setting, for each entry in the catalogue, the α to be an estimate of how many partial matches an S to S' extension will generate and the cost c to 1. This will make the cost of evaluating each operator be the number of tuples it receives.

4.4.3 I-Cost (`i-cost`)

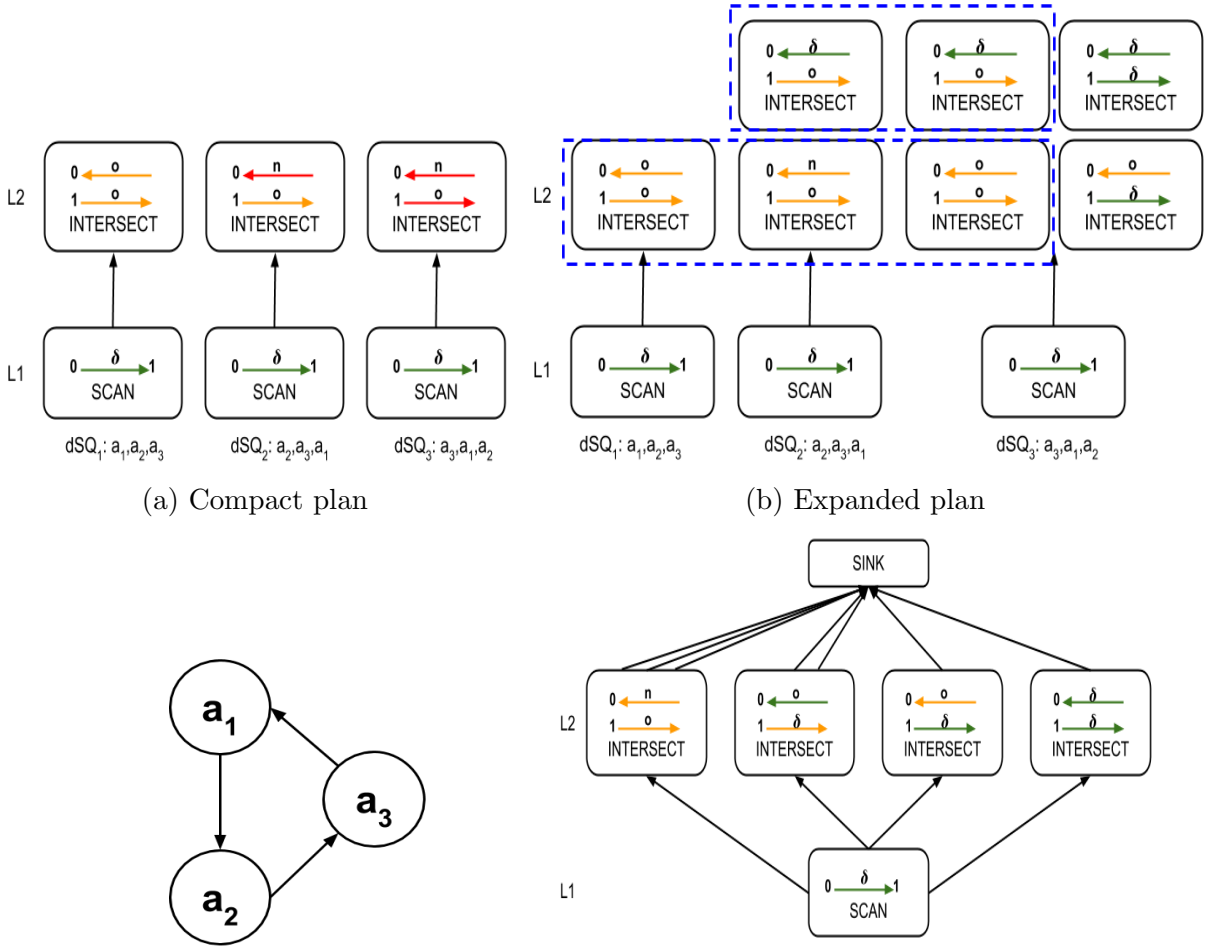
Finally, `i-cost` can be optimized for by using a catalogue in which, for each entry (S, ALD set A, S') the α values are the same as in the `int-matches` metric, and the cost c is the estimated amount of intersection work that will be performed on a partial match S when performing the intersections implied by A. Note that perhaps surprisingly, a single S can be extended to the same S' using different ALD sets, each with different work. For example, a single edge $S : 0 \rightarrow 1$ to an acyclic triangle $S' : a \rightarrow b \leftarrow c, a \rightarrow c$ with three different sets of ALDs: $0F, 1F$, $0F, 1B$, $0B, 1B$ and on many real world graphs the $0F, 1F$ is significantly cheaper than the other alternatives. `i-cost` can differentiate between these different extensions while `num-ops` and `int-matches` metrics cannot.

4.5 Expanded vs. Compact Evaluation of DSQs

Figure 4.2a shows the physical plan for the DSQs of the symmetric triangle shown in Figure 4.2c. Note that the E/I operators cannot share computation despite the fact that their ALDs have the same prefix indices and directions, because the ALDs differ in their graph versions. However, since in our setting the difference between E_o and E_n is small, almost all intersections performed by these operators are identical. In this section we introduce an optimization that removes the E_n version of the graph, which sometimes allows more sharing of computation.

4.5.1 Expanded DSQs

In Section 2.2 we showed the decomposition of a CSQ into multiple DSQs. Recall that each E_n term in the DSQs is equivalent to $(E_\delta + E_o)$. Using the distributive property of join operators, we can further expand the DSQs into a larger set of *expanded DSQs* (EDSQs) algebraically. In the case of the triangle CSQ, Equation 4.2 shows the seven expanded DSQs derived from the three compact DSQs of Equation 2.4.



(a) Compact plan

(b) Expanded plan

(c) Symmetric triangle

(d) Expanded combined plan

Figure 4.2: Compact and Expanded physical plans for a symmetric triangle

dSQ_1	$\text{dESQ}_1(a_1, a_2, a_3) = E_\delta(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_o(a_1, a_3)$	
dSQ_2	$\text{dESQ}_2(a_1, a_2, a_3) = E_o(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_o(a_1, a_3)$	
	$\text{dESQ}_3(a_1, a_2, a_3) = E_\delta(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_o(a_1, a_3)$	
dSQ_3	$\text{dESQ}_4(a_1, a_2, a_3) = E_o(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_\delta(a_1, a_3)$	(4.2)
	$\text{dESQ}_5(a_1, a_2, a_3) = E_\delta(a_1, a_2) \bowtie E_o(a_2, a_3) \bowtie E_\delta(a_1, a_3)$	
	$\text{dESQ}_6(a_1, a_2, a_3) = E_o(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_\delta(a_1, a_3)$	
	$\text{dESQ}_7(a_1, a_2, a_3) = E_\delta(a_1, a_2) \bowtie E_\delta(a_2, a_3) \bowtie E_\delta(a_1, a_3)$	

Figure 4.2b shows the corresponding *expanded physical plans* for the EDSQs of the symmetric triangle query. Each compact E/I operator in Figure 4.2a with one or more ALDs with version `new` splits into multiple E/I operators with versions `old` or `delta`. In contrast to Figure 4.2a where no ops can be combined, the circled operators in Figure 4.2b can be combined. The expanded combined plan of the triangle query is shown in Figure 4.2d. Although it contains four E/I operators in L2 (compared to 3 operators in the compact combined plan), only one operator has both of its ALDs with versions `old`. The other three E/I operators have at least one `delta` ALD the size of which we assume is very small. Therefore this optimization efficiently replaces three expensive ops with one expensive and three cheap operators. We show in some workloads this can lead to performance improvements.

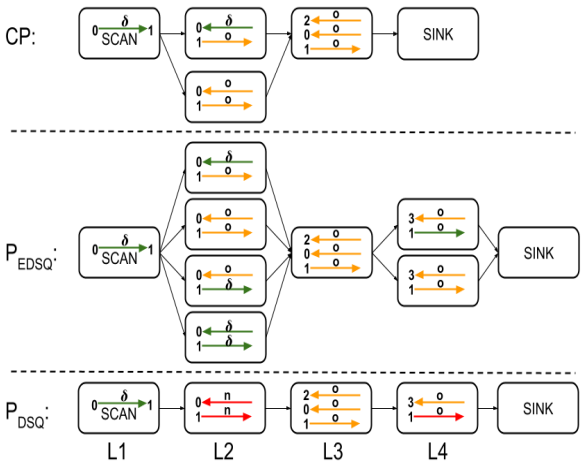
4.5.2 Expanded DSQ Optimization

In our implementation we do not expand each DSQ into EDSQs purely algebraically because the number of EDSQs can get very large for even small size queries. For example, a single DSQ for a 5-clique may decompose into as many as 512 EDSQs. Instead we modify line 2 of Algorithm 2 where we generate and calculate the cost of each plan P_{DSQ} of a compact DSQ dsq^* ordered by a qvo^*. We refer to the operators of P_{DSQ} as *compact operators*. Our procedure has 3 steps: (i) We first expand each operator of P_{DSQ} and construct a new plan P_{EDSQ}. We refer to the operators of P_{EDSQ} as *expanded operators*. (ii) We try to merge each expanded operator of P_{EDSQ} with CP starting from Scan in increasing levels; and (iii) if any compact operator of P_{DSQ} that has been expanded can be made compact again, we revert them back to compact operators. We call the final plan P_{Hybrid}. The final cost of P_{Hybrid} is calculated, as before, as the sum of the costs of each operator of P_{Hybrid} that has not been merged to CP. We next give an example.

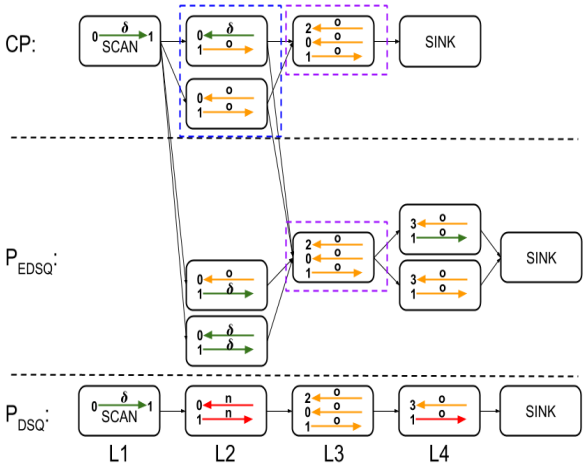
Example 4.5.1 *Consider the CP and P_{DSQ} shown in Figure 4.3a. Note that no sharing is possible between CP and P_{DSQ}. Our expanded DSQ optimization is implemented as follows.*

(i): Constructing P_{EDSQ}: We first expand each operator of P_{DSQ} that has at least one ALD with version `new` *and construct P_{EDSQ}. For instance, in Figure 4.3a, the L2 operator in P_{DSQ} with ALD versions (*`new`*,* `new`*) expands into four operators in P_{EDSQ} with ALD versions (*`old`*,* `old`*), (*`delta`*,* `old`*), (*`old`*,* `delta`*) and (*`delta`*,* `delta`*).*

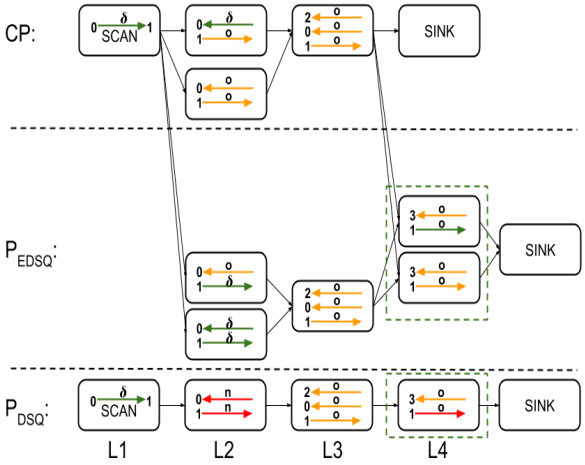
(ii): Merging P_{EDSQ} and CP operators: We next try to share computation between the operators of CP and P_{EDSQ} in two ways. First, as in compact DSQs, starting from the operators in L1 up to the last level, we merge operators in CP and P_{EDSQ} that have the



(a) CP, EDSQ and DSQ for qvo*.



(b) After exact operator merging.



(c) After partial operator merging.

Figure 4.3: Optimizing using EDSQs.

same set of ALDs and the exact same ancestors. We call this operation exact operator merging. Figure 4.3b shows two operators of P_{EDSQ} merged into CP this way. Second, the algorithm tries to partially merge operators in CP and P_{EDSQ}. The algorithm looks for pairs of operators in CP and P_{EDSQ} with identical ALDs where all the ancestors of the operator from CP match a subset of the ancestors of the operator from P_{EDSQ}. For instance, the L3 operators shown circled in purple in Figure 4.3b have identical ALDs and share the ancestors shown circled in blue. Note that the circled L3 operator from P_{EDSQ}

(L3-op) duplicates all the work performed by the corresponding circled operator from CP as it receives the same set of prefixes (plus some more from two other operators) and has the same ALDs. We therefore partially merge L3-op with CP by connecting the output of the circled CP operator to all L4 operators in P_{EDSQ}. Note that the cost of evaluating L3-op is now reduced as it has fewer ancestor operators and therefore will receive fewer prefixes. Figure 4.3c shows the P_{EDSQ} and CP after performing this partial operator merging.

(iii): Reverting expanded operators to compact: Finally, we revert expanded operators in P_{EDSQ} that have not been merged into CP back into compact operators. For instance, the two L4 operators circled in green in the P_{EDSQ} in Figure 4.3c would be reverted back to the single compact L4 operator circled in green in P_{DSQ}. This operation always reduces `i-cost` *because the expanded versions of compact operators always take the same amount of prefixes but loop over, cumulatively, longer adjacency lists. For example, the two expanded L4 operators in Figure 4.3c will loop over three* `old` *and one* `delta` *adjacency lists, whereas the compact version will loop over one* `old` *and one* `new` *(which in the worst case is as long as an* `old` *and a* `delta` *adjacency list combined). Therefore, in this instance, the expanded operators loop over one extra* `old` *adjacency list.*

Chapter 5

Implementation

We implemented our optimizer on top of a prototype in-memory graph database called Graphflow. The implementation of the initial version of Graphflow is a separate contribution of this thesis. The system is currently being improved by other researchers. We limit our description to the first version of the system and the version we use for our work in this thesis. In Section 5.1 we give an overview of Graphflow. Section 5.2 describes several important implementaion details about our E/I operator. Finally, Section 5.3 describes how our catalogues are constructed.

5.1 Graphflow System

Figure 5.1 shows the high-level architecture of Graphflow. Graphflow is a single node in-memory system implemented in Java. The system has four main components: (1) an in-memory property graph store; (2) the Cypher++ query language, which extends Neo4j's declarative Cypher language, described in reference [17] with subgraph-condition-action triggers; (3) a *One-time Query Processor* (OQP); and (4) a *Continuous Query Processor* (CQP). We next describe each component.

5.1.1 In-memory Property Graph Store

Graphflow's data model is a *property graph*, i.e., a labeled graph. Graphs are directed and vertices and edges can have arbitrary key-value properties on them. The graph is stored in two components.

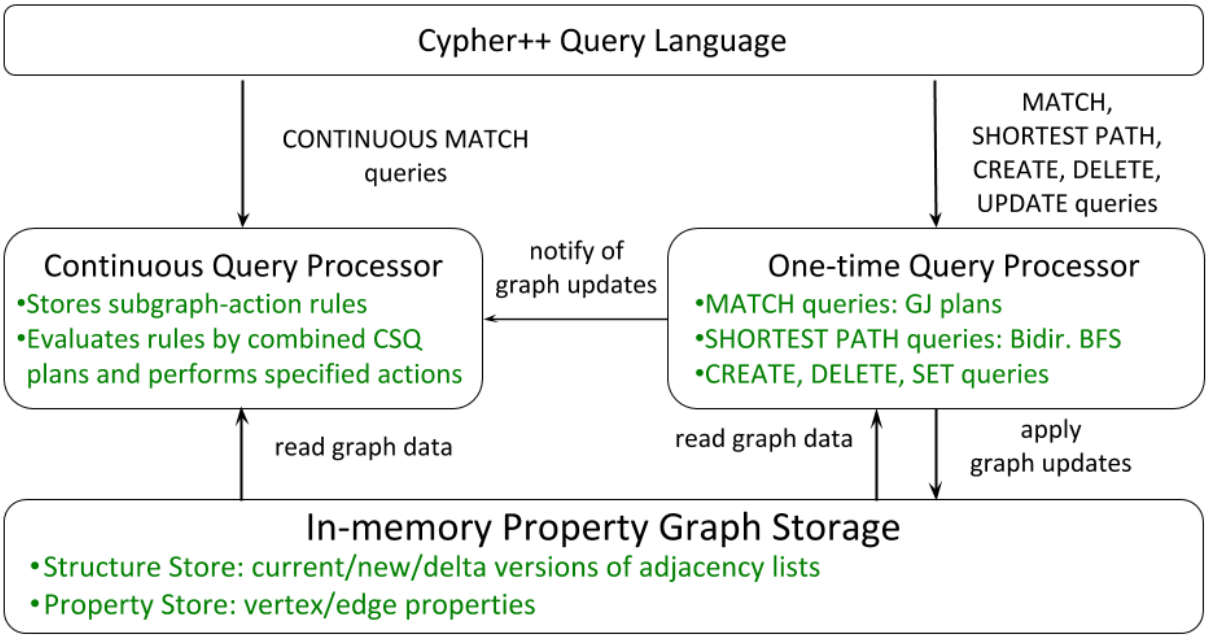


Figure 5.1: Graphflow architecture.

Structure Store: Stores, for each vertex u, u's incoming and outgoing adjacency lists stored as arrays. The adjacency lists are stored in sorted order of the outgoing neighbor IDs, which allows efficient intersections of two adjacency lists.

Property Store: Stores the key-value properties on vertices and edges. The keys can be arbitrary strings, and values can be integers, doubles, booleans, or strings.

5.1.2 One-time Query Processor

The OQP supports two types of queries; (i) one-time SQs; and (ii) single-pair shortest path queries (SPQs).

One-time SQs are expressed using original Cypher's MATCH clause. The following is the one-time query that finds all of the triangles in the graph that are formed by transfer type edges:

MATCH a_1-[type='transfer']->a_2-[type='transfer']->a_3, a_3-[type='transfer']->a_1

The type='transfer' brackets specify a filter on the edges. One time SQs are evaluated using the WCO GJ algorithm which was briefly discussed in Section 2.3.1. GJ evaluates SQs one query-vertex-at-a-time.

We express SPQs using the SHORTEST PATH clause we introduced in Cypher++. A query which matches the shortest path between nodes `1` and `20` takes the following form:

```
SHORTEST PATH (1, 20)
```

The OQP evaluates SPQs using a bi-directional breadth-first search technique where evaluation moves forward from the source node and backwards from the destination node in a breadth first manner until one or more common nodes in the middle are reached. We then backtrack from those middle nodes to the source and destination to get the shortest path.

5.1.3 CSQ Registration and Evaluation

Continuous subgraph queries are expressed through subgraph-condition-action triggers using our CONTINUOUS MATCH clause extension to Cypher. The following is an example of a query that detects the directed triangles in a transactions graph and calls a reportCircularTransfer UDF for each emerged triangle.

```
CONTINUOUSLY MATCH a1-[:type='transfer']->a2-[:type='transfer']->a3,
                   a3-[:type='transfer']->a1
ON EMERGENCE ACTION UDF reportCircularTransfer IN udf.jar
```

The other values for the ON clause are DELETION or ALL, which detect only the deletions or both the emergence and the deletions of the subgraph. The user specifies the action to be invoked for any matches in the form of a *user defined function* (UDF). CQP stores each registered CSQ in memory. Upon registering a new CSQ, our greedy optimizer optimizes a new CP for all of the registered CSQs from scratch.

Updates, such as insertions or additions of edges, are one-time queries that arrive at OQP. Upon an update, OQP first temporarily modifies the in-memory graph store, which implicitly stores three different versions of the graph upon receiving an update:

- `old`: Version of the graph prior to the update.
- `delta`: Only the updates.
- `new`: New version of the graph after applying the updates.

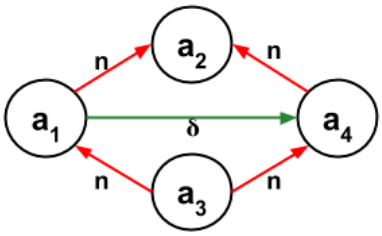


Figure 5.2: Asymmetric `diamond-with-edge` DSQ

OQP then notifies the CPQ of the updates, which runs the CP and for each registered CSQ, if the subgraph specified in the trigger emerged or was deleted, runs the UDF of the CSQ.

We perform the evaluation of CP in a *depth first* (DF) manner. We start evaluation with a SCAN operator. Suppose SCAN has children $o_1 \dots o_n$. We next evaluate the first child (o_1) of SCAN, and then the first child of o_1 and so forth recursively. When recursion unfolds, we then evaluate o_2, until all operators are executed in DF manner.

5.2 Optimizations in E/I Operator

In this section, we describe two simple optimizations we implemented in the E/I operator.

- *Batching of prefixes:* Each E/I operator pre-allocates a batch of integer arrays, each of which is designed to hold one output prefix. When the operator produces a batch of prefixes, it pushes them to the next operator. When the DFS execution returns back to the operator, it produces more prefixes, reusing the same arrays. Batching improves query evaluation runtime for two reasons; (i) it reduces the number of method calls between operators; and (ii) by reusing the same arrays, it makes fewer memory allocations, which reduces Java garbage collection time.

- *Caching:* In some operators, we cache a single prefix and the extensions it produces after intersections and reuse these if the subsequent input prefixes have the same set of intersections. We give an example below:

  Example 5.2.1 *Consider the DSQ in Figure 5.2, with QVO $a_1a_4a_2a_3$ and a single new edge (0, 5) as an example. We start by getting one 2-prefix, $\{(0, 5)\}$, matching*

a_1a_4. Next, we match $a_1a_4a_2$ by intersecting the **new** *forward adjacency lists of 0 and 5 to get a set of extensions, say $\{2, 3\}$, producing two 3-prefixes $\{(0, 5, 2), (0, 5, 3)\}$. Finally, we extend each 3-prefix to a_4. For prefix (0, 5, 2), we intersect the* **new** *backward adjacency lists of 0 and 5 and get, say, $\{10, 20\}$ as extensions. Note that we will do exactly the same intersection for (0, 5, 3). To avoid this intersection, we cache the result of the last intersection made and see if the next intersection is identical. Note that prefixes with identical intersections will appear consecutively in batches as they are produced consecutively in the previous operator.*

5.3 Catalogue Construction

In Section 4.1 we described the subgraph extension catalogue we input to our optimizer. Constructing and storing a full catalogue even if we limit the sizes of the subgraphs in the catalogue can be prohibitively expensive. For example, there are roughly 2.1 million different connected 7-vertex directed graphs [18], so a full catalogue containing all S's with 7 vertices would contain at least over 2 million entries. Therefore we use several optimizations to limit the amount of work we do to construct the catalogue.

5.3.1 Using a Combined Plan for Catalogue Construction

In the case of the `i-cost` catalogue, we calculate the <i-cost, α> tuple for a <S, A, S'> key (recall A is a set of ALDs), using information from the original edges in G (E_o in our notation). Recall from Section 4.1 that each S is given a canonical ordering. Note that an S with k vertices is effectively a DSQ without graph versions and that its canonicalized $0, 1, \ldots, k$ ordering is a QVO. For example the 2←0→1 open triangle would match 0→1 first and extend to 0's backward ALD. Therefore we construct a query-vertex-at-a-time plan P to evaluate S where all ALDs of P intersect adjacency lists from E_o. In order to calculate <i-cost, α> for extending from S to S' with ALDs A', we extend plan P with an E/I operator op' with ALDs A'. The input to op' is the output of P, which are instances of subgraph S. Then the <i-cost, α> of <S, A, S'> is the actual i-cost and α produced by evaluating op' on E_o, divided by the number of S-prefixes op' receives. When there are other subgraphs, say S'' extending S, we construct other operators, say op'' for each of them, effectively constructing a CP for computing the <i-cost,α> of multiple entries that extend S. By recursively considering S of increasing sizes, we create one very large CP for constructing a catalogue. We refer to this combined plan as CP_{cat}.

5.3.2 Partial Catalogue Construction and Sampling

For optimizing a given set of CSQs with max vertex size n, the catalogue only needs a subset of all the possible entries of size upto n. Specifically, we only need the subgraphs induced by each QVO of each DSQ. For example, for our SEED query set, which contains 6 queries and 39 DSQs over subgraphs with up to 5 vertices, we only need 46 entries, instead of 575 which is the number of different directed graphs with up to 5 vertices. Therefore, our system constructs partial catalogues and determines each entry needed by analyzing the DSQs in the query set and constructs a CP_{cat} for only those entries.

Finally, instead of evaluating the CP_{cat} by scanning all of the edges in E_o, we sample k edges (5000 by default) uniformly at random from E_o, and compute the i-cost numbers based on these k edges.

Chapter 6

Evaluation

In this chapter we evaluate multiple aspects of the performance of our optimizer. First, we validate our decision to optimize and evaluate CSQs together by comparing the performance of our optimizer against baselines which optimize queries individually. Next, we show that the i-cost based optimizer outperforms the other natural cost metrics described in 4.4. We then evaluate the effectiveness of the expanded DSQ optimization we described in 4.5.1. Finally, we compare the runtime of our system against Turboflux [5], which is the most efficient system in literature we are aware of or evaluating a single CSQ. We were not able to obtain the code for the EMVM system from reference [7], which is the only other work that studies evaluating multiple CSQs. We do not expect this system to be competitive with our optimizer as it does query-edge-at-a-time evaluation and is designed for CSQs with highly selective predicates. We review EMVM in Chapter 7.

6.1 Experimental Setup

CSQ Registration: In each of our experiments, we use a query set that consists of one or more CSQs (discussed momentarily). We register each CSQ to Graphflow, which then optimizes the CSQs either using our greedy optimizer or a baseline optimizer we describe in Section 6.2. We modified our code to not run the UDF of the CSQs we register. Instead, we increment a counter in order to speed up our experiments. We also do not materialize the final outputs of CSQs, which is a common cost of any correct plan.

Optimizers: The list of optimizers we experimented with are as follows:

- `Baseline`$_{no-share}$: This is our baseline optimizer that picks a QVO for each DSQ independently using a catalogue with i-cost information and runs each DSQ independently.

- `Greedy`$_{i-cost}$: Generates a CP using our greedy optimizer and a catalogue with i-cost information.

- `Greedy`$_{num-ops}$: Generates a CP using our greedy optimizer and a catalogue with all c and α set to 1, which as we discussed in Section 4.1, tries to generate a CP with the smallest number of operators.

- `Greedy`$_{num-int}$: Generates a CP using our greedy optimizer and a catalogue that modifies the i-cost catalogue by setting the c values to 1. As we discussed in Section 4.1, our greedy optimizer in this case tries to generate a CP that will generate the smallest number of intermediate partial matches.

- `Greedy`$_{expanded}$: Generates a CP greedily using expanded DSQs and a catalogue with i-cost information.

Input Graphs: Table 6.1 shows the input graphs we used in our experiments. We picked these graphs for three reasons:

- *Scale:* They contain both small graphs with ∼3M edges and large graphs with ∼69M edges.

- *Adjacency list size skew:* They contain graphs with both high and low skew in the distribution of the sizes of the forward and backward adjacency lists.

- *Global clustering coefficient:* Our set of graphs contains graphs with both high and low average global clustering coefficients. Clustering coefficient is a measure of how "clustered" a graph is. Formally, clustering coefficient is the fraction of closed triangles to the sum of closed and open triangles. Intuitively, graphs with higher clustering coefficients contain more triangles, cycles, and cliques.

We comment on these structural properties when discussing different experiments.

Updates and Execution: In each experiment, we take an input graph G, randomly pick 90% of G and preload it into Graphflow. We then submit the remaining 10% edges as a stream of insert-only updates in random order in batches of size 5. We do not experiment with workloads consisting of deletions. In other experiments we do not report here, we

Dataset	Num. Edges	Num. Vertices	Domain	Characteristics
Amazon	403,394	3,387,388	Product co-purchasing network	Dense and uniformly distributed
Google	875,713	5,105,039	Web graph	Dense and skewed
LiveJournal	4,847,571	68,993,773	Social network	Moderately dense and uniformly distributed
Patents	3,774,768	16,518,948	Citation Network	Sparse

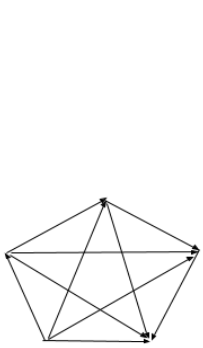
Table 6.1: Datasets used in evaluations.

used several other batch sizes from single edges to 100 edges. Batch sizes within this range did not have visible effects on our conclusions. To get reliable numbers, we repeat each experiment 10 times, each time starting with a new JVM process. We report the average runtimes of these 10 runs for each optimizer we experiment with.

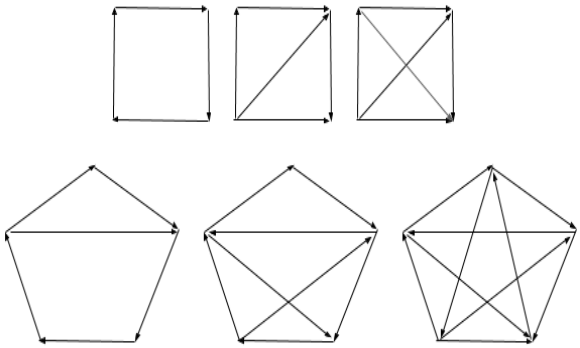
Query Sets: We picked query sets consisting of directed queries with 4 or 5 vertices, each with upto six queries. Each query set has some structural property that allows us to demonstrate how our optimizer behaves in different situations. We comment on these properties when discussing different experiments.

- All 4-cliques (`4Cs`): Consists of the 4 unique directed 4-cliques.
- All 4-cliques and one 5-clique (`4Cs-1-5C`): Consists of the 4 unique directed 4-cliques and the 5-clique from Figure 6.1a.
- 5 Random Sparse 5 vertex queries `5RS5`: Consists of 5 randomly generated CSQs with 5 vertices, each with at least one cycle. We took all 5-vertex connected queries that contain at most 5 edges and have at least one cycle, and picked 5 of these randomly. These queries are shown in Figure 6.1c.
- `SEED`: Consists of six CSQs with 4 and 5 vertices from a workload used for evaluating one-time SQs in reference [19]. These queries are shown in Figure 6.1b.

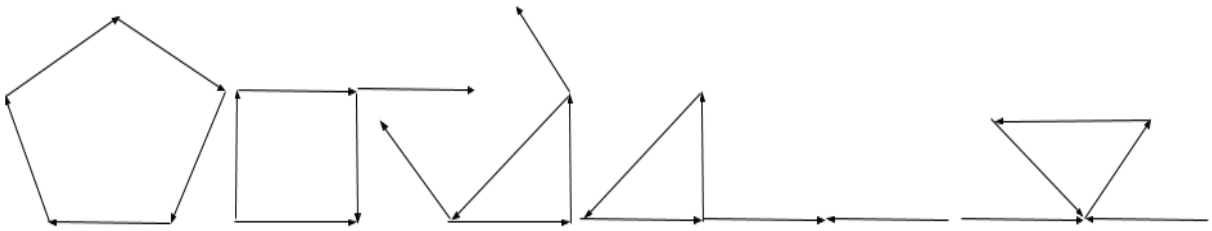
System Setup: We ran all experiments except those involving the LiveJournal dataset on a Linux machine with 256GB RAM and an Intel E5-2670 processor with 8 cores. For each run we set the initial and maximum memory allocation of the JVM to 200GB. For experiments with LiveJournal we used a machine with 512GB RAM and initialized the JVM memory size to 400GB. We did not perform any memory initialization for Turboflux as it is implemented in C++.



(a) `4Cs-1-5C`



(b) `SEED`



(c) `5RS5`

Figure 6.1: Query Sets

	Patents	Amazon	Live Journal	Google
4Cs	10.6 6.3 (**1.7x**)	8.4 6.7 (**1.3x**)	30.6 19.0 (**1.5x**)	18.4 15.5 (**1.2x**)
4Cs-1-5C	14.8 7.1 (**2.1x**)	17.8 13.1 (**1.4x**)	45.7 22.3 (**2.0x**)	66.5 55.7 (**1.2x**)
5RS5	110.8 103.2 (**1.07x**)	97.8 92.6 (**1.06x**)	839.3 844.0 (**1.0x**)	915.3 910.5 (**1.0x**)
SEED	18.3 9.7 (**1.9x**)	31.7 26.1 (**1.2x**)	271.7 189.3 (**1.4x**)	91.6 79.9 (**1.1x**)

Table 6.2: Runtime comparison(Seconds) of $\texttt{Baseline}_{no-share}$ and $\texttt{Greedy}_{i-cost}$

6.2 $\texttt{Baseline}_{no-share}$ Optimizer Comparisons

In this experiment we compared the performance of $\texttt{Baseline}_{no-share}$ with $\texttt{Greedy}_{i-cost}$ on all of our query sets and input graphs. Our goal is to compare how much runtime benefit we get by sharing computations and generating a CP. For reference Table 6.2 shows our results. In the table, the top and bottom rows are the runtimes for $\texttt{Baseline}_{no-share}$ and $\texttt{Greedy}_{i-cost}$, respectively. The bold numbers in parantheses are the runtime improvement factors $\texttt{Greedy}_{i-cost}$ has over $\texttt{Baseline}_{no-share}$.

Our first observation is that $\texttt{Greedy}_{i-cost}$ outperforms $\texttt{Baseline}_{no-share}$ across all query and data sets. Our second observation is that the degree to which $\texttt{Greedy}_{i-cost}$ improves over $\texttt{Baseline}_{no-share}$ varies significantly, specifically, from almost no improvement of 1.01x to significant improvement of 2.1x. Our next key observation explains this variety in the runtime benefits our greedy optimizer gets by sharing.

OBSERVATION: *The degree to which* $\texttt{Greedy}_{i-cost}$ *improves runtime over* $\texttt{Baseline}_{no-share}$ *depends on the ratio of work done in the last level of the CP to the total work of the CP.*

We note that in CPs no computation sharing is possible in the last level. Recall that we share two operators at level i between two DSQs, if they have exactly the same E/I operators in every level starting from 1 to i. If there were two DSQs which shared operators in the last level of a CP, this would imply that the CSQs in which the DSQs come from are already isomorphic, i.e., we have two duplicate registered CSQs. To demonstrate this, Table 6.3 shows, for each of our query sets, the number of operators in each level that the DSQs of $\texttt{Baseline}_{no-share}$ and the CP generated by $\texttt{Greedy}_{i-cost}$ have. As seen in the table, the CP of $\texttt{Greedy}_{i-cost}$ and DSQs of $\texttt{Baseline}_{no-share}$ always have the same number of operators in the last level. Since computation sharing in CPs can only happen in earlier levels, the amount of work done in these levels upper bounds how much $\texttt{Greedy}_{i-cost}$ optimizer can improve over $\texttt{Baseline}_{no-share}$.

We have picked our query sets and input graphs to be able to control how much of the total work is done at the last level. Specifically, in the `4Cs` query set, the number of last-level operators in CP is high compared to the total number of operators (24 out of 33 operators are in the last level). In contrast, in the `4C-1-5C` query set, the number of last-level operators is low compared to the total number of operators (only 10 out of 43 operators are in the last level). Therefore we expect $\texttt{Greedy}_{i-cost}$ to outperform $\texttt{Baseline}_{no-share}$ with a larger factor in `4C-1-5C` than in `4Cs`. Similarly, because all of the queries in `4Cs` and `4C-1-5C` 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 will

	Baseline$_{no-share}$	Greedy$_{i-cost}$
4Cs	24, 24, 24	1, 8, 24
4C5C	34,34,34,10	1,8,24,10
5RS5	25,25,25,25	1,9,21,25
SEED	39,39,39,24	1,9,22,24

Table 6.3: Number of operators in each level of the plans of Baseline$_{no-share}$ and Greedy$_{i-cost}$.

	Patents	Amazon
4Cs	**1.7x**	**1.26x**
	0 (0%), 330m (53%), 293m (47%)	0, 71m (11%), 555m (89%)
4C5C	**2.1x**	**1.37x**
	0, 330m, 293m, 191m	0, 71m, 555m, 870m

Table 6.4: Dependence of runtime gains on the work done in the last level

do less work. Therefore, we expect Greedy$_{i-cost}$ to outperform Baseline$_{no-share}$ with a larger factor on graphs with lower clustering coefficients. Table 6.4 shows a subsets of our experiments to demonstrate these effects clearly. The table shows the performance of `4Cs` and `4C-1-5C` query sets on the `Patents` and `Amazon` graphs, which respectively have 0.08 and 0.42 clustering coefficients. In each cell of the table, the first line shows the relative performance improvement Greedy$_{i-cost}$ has over Baseline$_{no-share}$. The second line shows the amount of actual i-cost in each level in the CP of the Greedy$_{i-cost}$. Consistent with our expectations, Greedy$_{i-cost}$ has the largest runtime improvements on the <`4C-1-5C`, `Patents`> cell and the lowest in the <`4C`, `Amazon`> cell.

6.3 Effectiveness of the Greedy Optimizer

We experimentally demonstrate that our greedy optimizer is effective at constructing good CPs in the following manner. First we calculated the i-cost of each possible CP of a queryset consisting of two 4-clique queries for the `Amazon` dataset. We show the spectrum of i-costs for the 4096 such CPs in Figure 6.2. Note that for our queryset it is possible to construct CPs ranging in i-cost from 427 to 539. Next, we picked a plan using the Greedy$_{i-cost}$ optimizer for the same query set on `Amazon`. Greedy$_{i-cost}$ yielded a plan with a cost of 432, which is near optimal for this query set. We make the observation that while Greedy$_{i-cost}$ is capable of picking very good plans, the specific plan it picks may not be

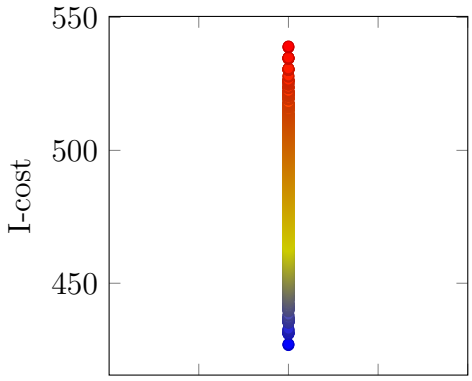


Figure 6.2: I-cost of 4096 possible plans for 2 4-clique queries on `Amazon`

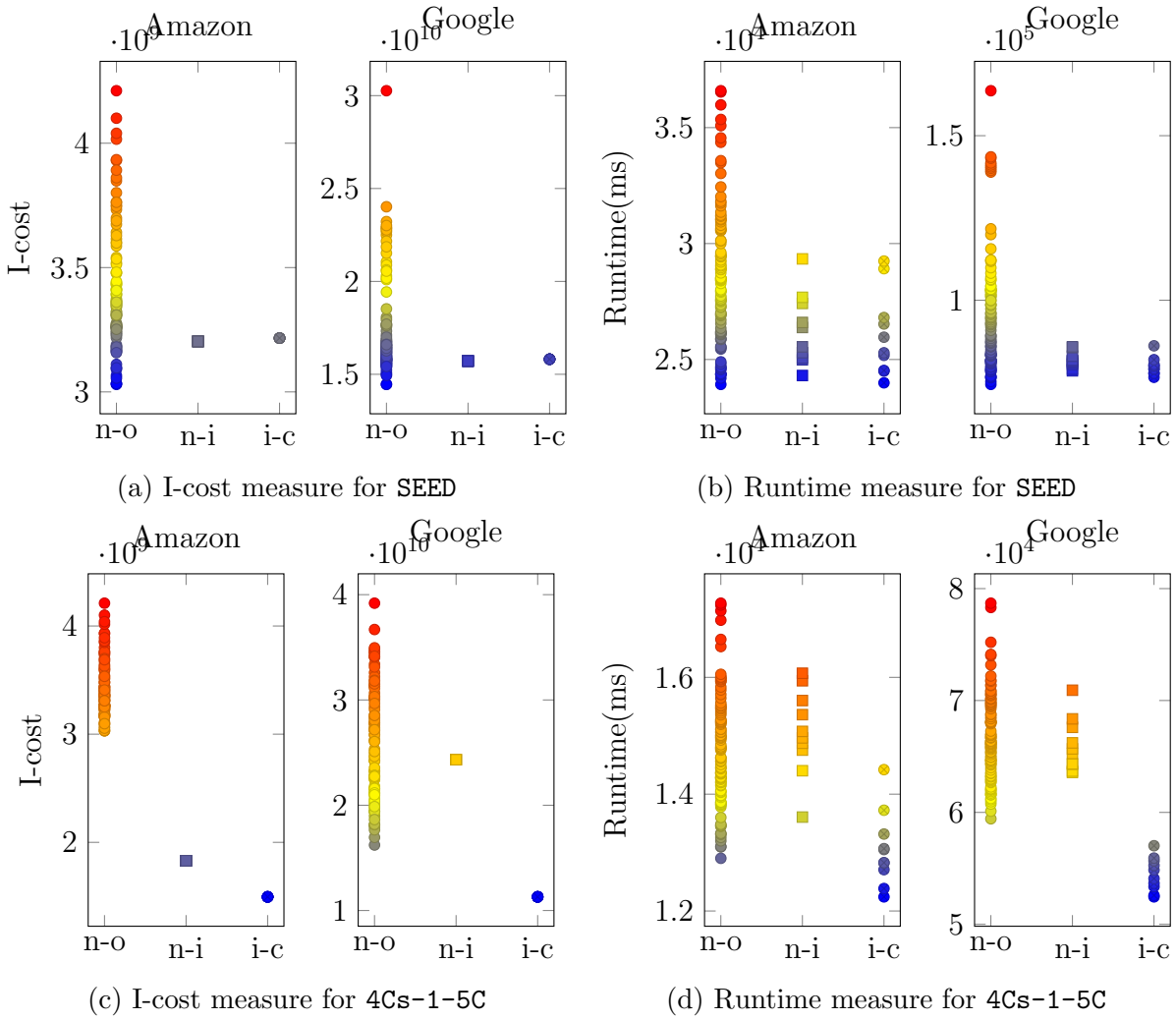
the optimal because $\texttt{Greedy}_{i-cost}$ greedily picks the best ordering one DSQ at a time.

We do not perform similar experiments where we exhaustively search all possible plans for the optimal plan for large querysets because of the large space of possible plans that has to be searched.

6.4 Effectiveness of Different Cost Metrics

In these experiments we compared our greedy optimizer's performance when optimizing for three different metrics we discussed in Section 4.4: (i) `num-ops` ($\texttt{Greedy}_{num-ops}$), (ii) `num-int` ($\texttt{Greedy}_{num-int}$) and (iii) the default metric of `i-cost` ($\texttt{Greedy}_{i-cost}$).

We start by noting that our greedy optimizer sometimes has to break ties when picking a <QVO, DSQ> pair to integrate into CP next. This is because there may be multiple <QVO, DSQ> pairs that add the same marginal estimated cost to the CP. We break those ties randomly. Therefore, our optimizer can generate different CPs when optimizing the same query set if we run it multiple times. In our first experiment, we assess the sensitivity of the plans our greedy optimizer generates to such tie-breaks under different cost metrics. Specifically, we ran two sets of experiments. First we took the `SEED` query set and `Amazon` data set and ran our optimizer with 100 different random seeds for each cost metric. Then we ran the updates for `Amazon` on each CP generated by our greedy optimizer once (a total of 300 executions in the end). In our second experiment, we repeated the first experiment with the `4Cs-1-5C` query set and `Google` graphs. Figures 6.3a and 6.3c show our results. Each column in the figures is one cost metric and each point on each



(a) I-cost measure for `SEED`

(b) Runtime measure for `SEED`

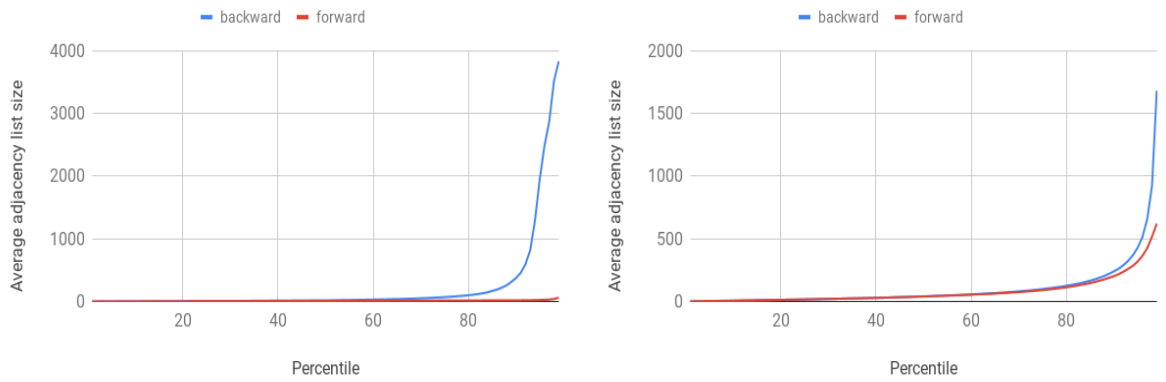
(c) I-cost measure for `4Cs-1-5C`

(d) Runtime measure for `4Cs-1-5C`

Figure 6.3: 100 run comparison of cost metrics

column is one experiment. The y-axes in the figures is the actual i-costs we measured for each plan. Figures 6.3b and 6.3d show the same experiments with runtime.

Our observation here is that $\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{num-int}$ have no and $\texttt{Greedy}_{num-ops}$ has a high variability in the CPs they generate. Surprisingly, despite being a very naive metric, $\texttt{Greedy}_{num-ops}$ can also find good CPs. For example, in the <`Amazon`, `SEED`> experiment it can even find plans with slightly better i-cost than $\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{num-int}$. However, it often finds plans that are significantly worse than the CPs generated by



(a) Google (b) Live Journal

Figure 6.4: Forward and backward adjacency list histograms for Google and Live Journal datasets.

$\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{num-int}$. This is seen in the <`Google`, `4Cs-1-5C`> experiment, where at least in our 100 runs, even the best plan $\texttt{Greedy}_{num-ops}$ finds has 1.45x worse i-cost than $\texttt{Greedy}_{i-cost}$ (and 1.04x worse in runtime). Essentially, $\texttt{Greedy}_{num-ops}$, if executed across many attempts, can pick good QVOs for DSQs that also share well. This suggests a possible optimizer can run $\texttt{Greedy}_{num-ops}$ a large number of times and then use the `i-cost` metric to estimate the `i-cost` of the CPs and pick the best CP. We have not experimented with this optimizer in this thesis. However, we emphasize that this optimizer still needs to estimate costs using the `i-cost` metric when picking the best plan. For example, in <`Google`, `4Cs-1-5C`> experiment, the plan with the highest runtime has lower number of operators (40) than the plan with the lowest runtime (41). Therefore, picking plans merely based on `num-ops` will be ineffective.

In our second experiment, we compared the performance of the CPs generated by $\texttt{Greedy}_{num-int}$ and $\texttt{Greedy}_{i-cost}$ across all of our query and data sets. Table 6.5 shows our results. In the table, each cell is an experiment for one query and data set pair. The first and second lines in each cell is the runtimes of $\texttt{Greedy}_{num-int}$ and $\texttt{Greedy}_{i-cost}$, respectively. The bold numbers in parantheses is how much runtime improvement $\texttt{Greedy}_{i-cost}$ has over $\texttt{Greedy}_{num-int}$.

Our first observation here is that $\texttt{Greedy}_{i-cost}$ often finds better plans than $\texttt{Greedy}_{num-int}$ (14 out of our 16 experiments). Our second observation is that the performances of both cost metrics is very close on every dataset except `Google`. This demonstrates that $\texttt{Greedy}_{num-int}$ is also a good metric. The reason $\texttt{Greedy}_{i-cost}$ performs better than

	Patents	Amazon	Live Journal	Google
4Cs	6.6 6.2 (**1.06x**)	7.0 6.8 (**1.04x**)	21.1 19.8 (**1.06x**)	18.1 15.4 (**1.17x**)
4Cs-1-5C	7.1 7.1 (**1.0x**)	15.0 13.1 (**1.15x**)	24.4 22.3 (**1.1x**)	66.0 54.5 (**1.2x**)
5RS5	95.7 103.2 (**0.92x**)	92.3 92.6 (**1.0x**)	777.0 844.1 (**0.92x**)	1001.8 910.5 (**1.1x**)
SEED	10.5 9.7 (**1.07x**)	26.3 26.1 (**1.0x**)	189.7 189.3 (**1.0x**)	81.8 79.9 (**1.0x**)

Table 6.5: Runtime comparison(Seconds) of $\texttt{Greedy}_{num-int}$ and $\texttt{Greedy}_{i-cost}$

	Patents	Amazon	Live Journal	Google
4Cs	6.2 6.3(**1.01x**)	6.8 6.4(**1.06x**)	19.8 17.5(**1.13x**)	15.4 15.3(**1.01x**)
5RS5	103.2 97.3(**1.06x**)	92.6 72.3 (**1.28x**)	844.1 602.2(**1.4x**)	910.5 813.9 (**1.1x**)
SEED	9.7 8.0 (**1.2x**)	26.1 23.5 (**1.1x**)	189.3 147.7 (**1.3x**)	79.9 71.9 (**1.1x**)

Table 6.6: Runtime comparison(Seconds) of $\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{expanded}$

$\texttt{Greedy}_{num-int}$ on the `Google` graph (up to 1.21x) is that out of our four data sets, `Google` has the most skew in terms of the distribution of its forward and backward adjacency lists sizes. To demonstrate this, Figure 6.4 shows the distributions of the forward and backward adjacency lists of `Google` and `Live Journal` graphs. As seen in the figures, there is a very big difference between the distributions of the forward and backward adjacency list sizes of `Google` and significantly less in `Live Journal`. Therefore, picking the directions of the adjacency lists correctly in the E/I operators is more important in `Google` than on `Live Journal`. Recall that `num-int` metric only estimates the number of partial matches that will be processed, as opposed to the intersection work done in an operator, which is partly determined by the directions of the adjacency lists that will be intersected. Instead, `i-cost` considers the directions of the adjacency lists that will be intersected.

6.5 Performance of the Expanded DSQ Optimization

We experimentally compared the performance of the CPs generated by $\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{expanded}$ across all of our query and data sets. Table 6.6 shows our results. In the table, each cell is an experiment for one query and data set pair. The first and second lines in each cell is the runtimes of $\texttt{Greedy}_{i-cost}$ and $\texttt{Greedy}_{expanded}$, respectively. The bold numbers in parentheses is how much runtime improvement $\texttt{Greedy}_{expanded}$ has over $\texttt{Greedy}_{i-cost}$.

We first make the observation that $\texttt{Greedy}_{expanded}$ improves runtime compared to $\texttt{Greedy}_{i-cost}$ for all dataset, queryset pairs. While the improvements are modest in some instances (1.01x on < `Patents`, `4Cs`>), in other cases, the improvements are significant (1.4x for < `Live Journal`, `5RS5`>). The gains changes across querysets and input graphs as; (i) runtime savings depend on the number of shared operators that resulted from the DSQ expansion, which in turn depends on the specific CSQs being optimized; and (ii) the intersection work done in the extra operators shared in $\texttt{Greedy}_{expanded}$, which depends on the types of ALDs in the operators and the input graphs. However there is no specific pattern that we could identify in the improvements given by $\texttt{Greedy}_{expanded}$.

6.6 Turboflux

Reference [5] describes an algorithm called TurboFlux that evaluates a query using a *data centric graph* (DCG), which is a compressed representation of partial matches of the query in $G(E_G, V_G)$. Given a query $Q(V_Q, E_Q)$, TurboFlux first converts Q into a rooted query tree $Q'(V_{Q'}, E_{Q'})$ by removing some query edges that form cycles. Conceptually, the DSG is a multigraph $G_{DSG}(V_{DSG}, E_{DSG})$ where $V_{DSG} = V_G$ and E_{DSG} contains $|V_Q| - 1$ parallel edges for each $e \in E_G$. Each parallel edge has a *state* of *numm* (N), *implicit* (IM) (IM), or *explicit* (EX) and a *label* which is one of the query vertices in Q'. In practice only IM and EX edges are stored in DCG. An EX edge $e_{DSG}(v_i, v_j)$ with label $u' \in V_{Q'}$ indicates that there is a path $p : v_r \rightsquigarrow v_i \rightarrow v_j$, where the last edge is (v_i, v_j), and p matches, in order, all of the query vertices in the path from u_r to u_j in Q' (so u_r matches v_r and u' matches v_j). In addition every subtree of u' in Q' matches a subtree starting from v_j. An implicit edge $e_{DSG}(v_i, v_j)$ matches path p but not the subtrees from u'. Upon updates to G, TurboFlux transitions the states of the edges, say from IM to EX or to N. Transitions of edges to and from EX trigger a subgraph matching algorithm on the DSG to detect the emerged or deleted instances of Q. The reference uses a modified version of the TurboHOM++ [20] algorithm for subgraph matching, which performs the matching using

traversals on a DSG-like compressed representation, instead of tuple based processing as we do in our work.

In reference [5], TurboFlux has been shown to be very efficient for queries with selective predicates both in terms of runtime performance as well as the size of the DCG it stores. We obtained a binary code of TurboFlux from the authors of reference [5]. The binary had several bugs, which made TurboFlux return different number of outputs for several queries. We verified that our numbers are correct by verifying the outputs with Neo4j. At the time of the writing of this thesis, this bug was not fixed. TurboFlux can only run one query instead of multiple queries. For our comparison, we picked the diamond query and the `Google` data set because TurboFlux was producing a very close number of outputs to the correct output (only a 846 deficit for 50259016 matches). In this experiment TurboFlux took 34.6 seconds and the CP that our $\texttt{Greedy}_{i-cost}$ optimizer produced took 3.8s seconds.

We emphasize that we did these comparisons only for completeness of this study. The systems and implementations are very different and the binary we obtained was buggy. Therefore, we do not interpret our runtime benefits to indicate that TurboFlux's method of working on compressed data structures is inefficient. We find techniques that perform computations on compressed data structures promising and a much more detailed study is needed to understand the advantages and disadvantages of such approaches.

Chapter 7

Related Work

We review work in five related areas.

7.1 Multiple Continuous Subgraph Query Evaluation

Closest to our work is reference [7], which studies maintaining multiple subgraph query views under single-edge insertion workloads. Abbreviating the title of this reference, we refer to the technique here as EMVM. Given a set of queries $\bar{Q}$, EMVM partitions the queries in $\bar{Q}$ into separate query sets $\bar{Q}_{l_1}, ..., \bar{Q}_{l_k}$, 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 DSQs. For each $\bar{Q}_{l_i}$, EMVM constructs a larger "merged view" M_{l_i}, which is a (query) graph to which each EAV in $\bar{Q}_{l_i}$ is isomorphic to. Merged views are similar to our combined plans and are used to infer common subgraphs to multiple queries during query evaluation upon insertions. For example, upon an insertion of an edge e, the evaluation starts from a *central edge* in M, which is similar to the delta edge in our DSQs. Overall, EMVM can be viewed as constructing a specific combined plan in our framework. However, 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 the cyclic structures in queries[1]. Instead our greedy optimizer minimizes the i-cost of combined plans, considering cyclic structures of queries during

[1]To be precise, the metric the authors optimize when constructing M is slightly more complicated. For

optimization. Considering cyclic structures, e.g., closing triangles as early as possible, is fundamental for efficiently evaluating structural queries.

2. Queries are evaluated one query edge at a time. This might work well when each query edge matches a small number of data edges due to highly selective predicates, but will be inefficient for structural queries. Instead, we evaluate queries one vertex at a time utilizing intersections to filter partial matches.

7.2 Single Continuous Subgraph Query Evaluation

We review Turboflux in Chapter 6. There are several other existing work on evaluating a single continuous subgraph query both in the serial and distributed settings. Many of these techniques are primarily designed for queries with highly selective predicates and ignore structural properties of the queries, making them inadequate for structural queries. We review work on matching queries exactly here. Reference [21] studies approximate continuous subgraph matching on Pregel-like systems, which we do not review.

BigJoin [3]: Reference [3] studies two main aspects of the DSQ framework for a single query in the distributed setting. First, the authors study theoretically the round and communication complexity of evaluating DSQs in a distributed bulk synchronous parallel model of computation. Second, they study how to efficiently implement DSQs in practice using low cluster memory in the Timely Dataflow system [22]. This work does not study either the performance effects of different QVOs of the DSQs or optimizing multiple queries, which is our focus here.

IncIsoMat [23]: Reference [23] 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. Let d be the diameter of Q. The affected area contains the vertices that are within a radius d of u and v. IncIsoMat finds the matching instances of Q in this affected region, but how the matching is done is unspecified, so any subgraph matching algorithm can be used with IncIsoMat. Our DSQ framework automatically localizes its search to the same and sometimes smaller area around e, so IncIsoMat's explicit localization step would be unnecessary in our approach. Similar to our approach, IncIsoMat has no memory footprint but is designed to evaluate only a single query instead of multiple queries.

a given M, the score of M is the sum over each edge $e \in M$ of the score of e, where $score(e)$ is the square of the number of EAVs that share e.

SJ-Tree [4]: Reference [4] 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.The decomposition of Q is done by a greedy algorithm which iteratively decomposes Q into a Q' and l, where l is the 1-edge or 2-edge subgraph in Q with the lowest *relative selectivity*. Relative selectivity is a cost measure that incorporates the expected number of matches to l based on selectivity statistics and the average degree of the vertices in the input graph G, which captures how much more work will be done for searching for 2-edge paths compared to 1-edge paths. Upon updates to the graph, SJ-Tree maintains partial matches to each intermediate node of P using a hash join algorithm. The original technique is described for a streaming graph where the matches to Q are searched within a window of time interval but by keeping an unbounded time window, this technique can be adopted to our setting.

SJ-Tree has two main shortcomings for the queries we study in this thesis. First, SJ-Tree is designed for queries with highly selective predicates. For example, the reference assumes that the number of matches for 2-edge paths are expected to be significantly fewer than 1-edge paths, which would not hold for structural queries. As a result, for the queries we study here, the technique will materialize prohibitively large intermediate results, one for each intermediate subgraph of Q in the left-deep join plans. Second, the hash-join based evaluation of left-deep plans is equivalent to a query-edge-at-a-time matching strategy and is agnostic to cyclic structures in Q unlike our query-vertex-at-a-time evaluation. A prior study [5] has shown that even on queries with predicates, when the window interval is unbounded, SJ-Tree can maintain very large intermediate results and is slower than the TurboFlux technique. This is why we chose TurboFlux to compare our approach against in this thesis.

7.2.1 Multiple One-time Subgraph Query Evaluation

PCM [8]: Reference [8] studies the problem of evaluating multiple one-time subgraph queries $\bar{Q}$ together. Similar to our combined plans, the authors describe a data structure called *pattern containment map* (PCM), which is constructed to represent common subgraphs between groups of queries in $\bar{Q}$. The matching of the common subgraphs are shared across queries during evaluation of queries. The PCM construction subroutine is fundamentally designed for heterogenous graphs and queries with many predicates and becomes degenerate in the absence of predicates. In particular, during PCM construction, the reference uses a metric called the *grouping factor* of two queries to measure how much commonality they have in terms of their edges. The grouping factor is computed as a function of the common predicates between the queries and not their structural commonalities. In the absence of predicates, this measure would be similar for any two queries and

the subroutine would randomly group queries. Similar to several references we discussed above, the PCM construction subroutine also maximizes the number of common shared edges across queries ignoring the cyclic structures in queries.

SPARQL-MQO [6]: Reference [6] studies the problem of optimizing a set of SPARQL queries $\bar{Q}$, each asking for a labeled, directed subgraph in an RDF dataset. We refer to the approach in the reference as SPARQL-MQO. SPARQL-MQO is based on two high-level steps. In the first step, $\bar{Q}$ is broken into multiple groups using k-means clustering, using the Jaccard similarity of the common predicates that appear between queries. In the second step, each set S of queries in each cluster are independently translated into two components: (i) a single query Q^*_S, which is a subgraph shared by all queries in S; and (ii) one optional subgraph Q_{rem} to match exactly $Q \setminus Q^*_S$ for each $Q \in S$. The way the common subgraph Q^*_S is constructed is based on a cost model that tries to put (i) the most selective edge that is common to all queries in S; and (ii) as many extra edges as possible. SPARQL-MQO has the same shortcomings for structural queries as PCM. Specifically, its clustering technique (and selectivity information) becomes degenerate and the construction of Q^*_S ignores the cyclic structures in queries.

7.3 Single One-time Subgraph Query Evaluation, Complex Join Algorithms, and IVM

There is a rich literature on evaluating a single one-time subgraph queries on a static graph, which is equivalent to complex join queries in relational terms. A detailed review of subgraph matching and complex join algorithms is beyond the scope of this thesis. Pointers to several of these algorithms can be found in references [14,24,25]. The evaluation technique we use for DSQs fall under the query-vertex-at-a-time technique, which in relational terms is equivalent to attribute at a time processing of joins (in contrast to traditional table(s) at a time join processing). Query-vertex-at-a-time join processing was introduced by the new worst-case optimal join algorithms [14].

Similarly, the problem we study in this thesis is equivalent to multi-query IVM of complex join queries in relational terms. There is a vast body of work on incrementally maintaining views that contain selection, projection, joins, group-by-aggregates, among others. We refer the readers to reference [26] for a survey of these techniques. Briefly, our DSQ framework falls under the algebraic technique of representing updates to tables as delta relations and maintaining views through a set of relational algebraic queries. Prior work on algebraic techniques range from addressing limitations of delta query-based techniques,

e.g., when evaluating a top-k query [27], to techniques using higher-delta queries [28], e.g., delta queries of delta queries of a query.

7.4 Multi Query Optimizations For XML and Relational Databases

Several papers study optimizing multiple one-time XPath queries for XML based on sharing common prefixes of queries [29, 30]. These studies focus on path queries, where sharing maximum prefixes is easier than general queries we study in this paper, which can be cyclic or arbitrarily acyclic, e.g., tree-like. Reference [31] studies evaluating multiple XPath-like queries, with same structure running on different XML document streams. In this work, the queries do not share computation as they run on different documents. Instead, we study multi-query optimization on queries that are running on the same data set and share computations that are common to multiple queries.

There is a vast body of work of multi query optimizations for relational databases on various topics such as extending existing cost-based optimizers to handle multiple queries, sharing intermediate results between operators, or scheduling of operators in combined plans. We do not review this work here. Some pointers to these works can be found here [32].

Chapter 8

Conclusions and Future Work

We studied the problem of evaluating multiple directed structural subgraph queries on a changing graph. We built upon the DSQ framework, which decomposes a CSQ into multiple DSQs and evaluates DSQs in a query-vertex-at-a-time manner using an E/I operator that intersects one or more adjacency lists. We outlined three factors that affect runtime during DSQ evaluation; (i) amount of computation sharing between DSQs; (ii) number and direction of ALDs used in the E/I operators; and (iii) number of intermediate prefixes. We studied how to pick good QVOs for the DSQs of a set of CSQs together to create a CP which shares computation across DSQs and tries to minimizes the amount of intersection work done cumulatively. We proposed a cost-based optimizer that takes as input a set of DSQs and a subgraph extension catalogue, orders each DSQ greedily and outputs a CP. Based on our observations, we proposed three cost metrics that can be optimized in our greedy optimizer when constructing CPs; (i) the number of operators `num-ops`; (ii) the total number of intermediate results input to operators `int-matches`; and (iii) the sizes of the adjacency lists intersected (`i-cost`). We experimentally and analytically show that `i-cost` is a good approximation for the actual work done during the evaluation of our combined plans and adopt it as our default cost metric. We then described an optimization which algebraically decomposes DSQs further into EDSQs in order to allow more sharing between operators.

We experimentally justified optimizing DSQs together by comparing our greedy optimizer with two baselines; i) ordering DSQs canonically without sharing; ii) canonically ordering DSQs and sharing when possible. We showed that the greedy optimizer outperforms the baselines across a range of query workloads and input graphs. We further showed that the degree of improvement of the optimizer over the baselines depends on the amount of work done in the higher-level operators of the combined plan, which in turn depends

on two factors; (i) the query set; (ii) the input graph and set of updates. We showed that our CPs improve the most on query sets with smaller numbers of operators in higher levels and input graphs which produce few matches for the given queries. On a 4-clique CSQ we showed that our greedy optimizer using the i-cost metric finds the best possible plan by exhaustively considering all possible plans. Finally we showed that our framework outperforms the most efficient system we found reported in literature called Turboflux [5].

We outline three directions for future work.

- *Adaptive evaluation of updates:* In a continuous query setting, there is no advantage to using a single plan for a query (or set of queries) forever. Ideally, a system should have a set of possible plans and at different points in time pick different plans, say to adapt to changes in the workload. In our setting, it is possible that we can generate different CPs for different types of updates. For example, it is possible that a different CP than the one our optimizer picks could be faster for updates (u, v) where both u and v have low degrees or where both u and v have high degrees. Similarly, in our experiments we always chose as updates edges from the original graph, assuming that the update edges would look "similar" to the edges in the input graphs. If this assumption holds, for example, if the graph starts getting random edges, then the CP we generate might not be very efficient. Ideally, a system should be able to adapt to such changes in the workload.

- Most of our evaluations use query sets with less than ten queries, each with less than ten vertices. Future research can study the problem of optimizing and evaluating large numbers or large sizes of continuous queries together using our greedy optimizer. We have not studied the scalability of our optimizer in this thesis.

- We mentioned in Chapter 1 that our work here does not address queries with predicates on nodes or edges. An interesting avenue of future work is to study how predicates on nodes and edges would affect greedy optimization of DSQs, as the amount of sharing possible would be reduced due to the predicates. A potential solution might be to use auxiliary data structures to materialize certain computations such that sharing is maximized, as suggested in reference [33]. Queries with predicates are perhaps more important than the structural queries we studied here because we expect many applications to have some predicates on queries.

References

[1] 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*, vol. 7, no. 13, 2014.

[2] M. E. J. Newman, "Detecting community structure in networks," *The European Physical Journal B*, vol. 38, no. 2, 2004.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows," *PVLDB*, vol. 11, no. 6, 2018.

[4] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs," in *EDBT*, 2015.

[5] 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, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *SIGMOD*, 2018.

[6] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable Multi-query Optimization for SPARQL," in *ICDE*, 2012.

[7] A. Pugliese, M. Bröcheler, V. S. Subrahmanian, and M. Ovelgönne, "Efficient multiview maintenance under insertion in huge social networks," *ACM Transactions on the Web*, vol. 8, no. 2, 2014.

[8] X. Ren and J. Wang, "Multi-query optimization for subgraph isomorphism search," *PVLDB*, vol. 10, no. 3, 2016.

[9] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views," *SIGMOD Record*, vol. 15, no. 2, 1986.

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

[11] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Transactions of Database Systems*, vol. 42, no. 4, 2017.

[12] T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF Data," *The VLDB Journal*, vol. 19, no. 1, 2010.

[13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," *PVLDB*, vol. 6, no. 4, 2013.

[14] H. Ngo, C. Ré, and A. Rudra, "Skew Strikes Back: New Developments in the Theory of Join Algorithms," *SIGMOD Record*, vol. 42, no. 4, 2014.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection: Amazon product co-purchasing network, march 02 2003." https://snap.stanford.edu/data/amazon0302.html.

[16] "Maximum common induced subgraph." https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph.

[17] "Cypher graph query language." https://www.opencypher.org/.

[18] "The on-line encyclopedia of integer sequences." https://oeis.org/A086345.

[19] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proc. VLDB Endow.*, vol. 10, pp. 217–228, Nov. 2016.

[20] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming Subgraph Isomorphism for RDF Query Processing," *PVLDB*, vol. 8, no. 11, 2015.

[21] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous Pattern Detection Over Billion-edge Graph Using Distributed Framework.," in *ICDE*, 2014.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *SOSP*, 2013.

[23] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental Graph Pattern Matching," in *SIGMOD*, 2011.

[24] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition.," *IJPRAI*, vol. 18, no. 3, 2004.

[25] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases," *PVLDB*, vol. 6, no. 2, 2012.

[26] Rada Chirkova and Jun Yang, "Materialized Views," *Foundations and Trends in Databases*, vol. 4, no. 4, 2012.

[27] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient Maintenance of Materialized Top-k Views," in *ICDE*, 2003.

[28] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views," *PVLDB*, vol. 5, no. 10, 2012.

[29] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. Index-Based XML Multi-Query Processing," in *ICDE*, 2003.

[30] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To, "YFilter: Efficient and Scalable Filtering of XML Documents," in *ICDE*, 2002.

[31] Hong, Mingsheng and Demers, Alan J. and Gehrke, Johannes E. and Koch, Christoph and Riedewald, Mirek and White, Walker M., "Massively Multi-query Join Processing in Publish/Subscribe Systems," in *SIGMOD*, 2007.

[32] P. Roy and S. Sudarshan, *Multi-Query Optimization*. Springer US, 2009.

[33] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, (New York, NY, USA), pp. 205–216, ACM, 1996.

[34] A. Mhedhbi, C. Kankanamge, S. Salihoglu, "Evaluating Fixed-length Subgraph Queries With a Mix of Tradition and Modernity," tech. rep., University of Waterloo, 2018. http://dummy-url.uwaterloo.ca/.

[35] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. VLDB Endow.*, vol. 11, pp. 420–431, Dec. 2017.

[36] "Dgraph graph database." https://dgraph.io/.

[37] "Neo4j graph database." https://neo4j.com/.

[38] "Janusgraph distributed graph database." http://janusgraph.org/.

[39] G. Sadowksi and P. Rathle, "Fraud Detection: Discovering Connections with Graph Databases." https://neo4j.com/use-cases/fraud-detection/.

[40] "Apache jena." https://jena.apache.org/.

[41] "Sparksee high performance graph database." http://www.sparsity-technologies.com/.

[42] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*. O'Reilly Media, Inc., 1st ed., 2016.

[43] "Apache giraph." https://giraph.apache.org/.

[44] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," vol. 11, pp. 1876–1888, 08 2018.

[45] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," in *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pp. 133–144, VLDB Endowment, 2013.

[46] S. Choudhury, *Subgraph Search for Dynamic Graphs*. PhD thesis, Washington State University, 2014. https://research.libraries.wsu.edu/xmlui/handle/2376/5114.

[47] C. Joslyn, S. Choudhury, D. Haglin, B. Howe, B. Nickless, and B. Olsen, "Massive Scale Cyber Traffic Analysis: A Driver for Graph Database Research," in *GRADES*, 2013.

[48] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), pp. 411–426, ACM, 2018.

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