# RapidFlow: An Efficient Approach to Continuous Subgraph Matching

Shixuan Sun
National University of Singapore
Singapore
sunsx@comp.nus.edu.sg

Xibo Sun
Hong Kong University of Science and Technology
Hong Kong, China
xsunax@cse.ust.hk

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Qiong Luo
Hong Kong University of Science and Technology
Hong Kong, China
luo@cse.ust.hk

## ABSTRACT

Continuous subgraph matching (CSM) is an important building block in many real-time graph processing applications. Given a subgraph query $Q$ and a data graph stream, a CSM algorithm reports the occurrences of $Q$ in the stream. Specifically, when a new edge $e$ arrives in the stream, existing CSM algorithms start from the inserted $e$ in the current data graph $G$ to search $Q$. However, this rigid matching order of always starting from $e$ can lead to a massive number of partial results that will turn out futile. Also, if $Q$ contains automorphisms, there will be a lot of redundant computation in the matching process. To address these two problems, we propose RapidFlow, an effective approach to CSM. First, we design a query reduction technique, which reduces CSM to batch subgraph matching (BSM) where we enumerate all results in a region of $G$ that will be affected by the update. The well-established BSM techniques can determine effective matching orders, not necessarily starting from the newly inserted edge. Second, to eliminate redundant computation caused by automorphisms in $Q$, we propose dual matching, which leverages the duality of $Q$ and $G$ in the matching process. Extensive experiment results show that RapidFlow outperforms state-of-the-art algorithms, including TurboFlux and SymBi, by up to two orders of magnitude on various workloads.

**PVLDB Reference Format:**
Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. PVLDB, 15(11): 2415 - 2427, 2022.
doi:10.14778/3551793.3551803

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/shixuansun/RapidFlow.

## 1 INTRODUCTION

*Continuous subgraph matching* (CSM) reports the occurrences of a query graph in a graph stream. Specifically, given a query graph $Q$, a data graph $G$ and a sequence $\Delta\mathcal{G}$ of updates on $G$, CSM finds the *incremental matches* of $Q$ in $G$ for each update $\Delta G \in \Delta\mathcal{G}$. For example, $\{(u_0, v_0), (u_1, v_1), (u_2, v_4), (u_3, v_{10}), (u_4, v_{11})\}$ is a match given $Q$ and $G$ in Figure 1. As matches $\{(u_0, v_2), (u_1, v_3), (u_2, v_8), (u_3, v_{12}), (u_4, v_{13})\}$ and $\{(u_0, v_2), (u_1, v_3), (u_2, v_8), (u_3, v_{13}), (u_4, v_{12})\}$ appear in $G'$ when inserting $e(v_2, v_3)$ to $G$ in Figure 1c, they are incremental results for the update.

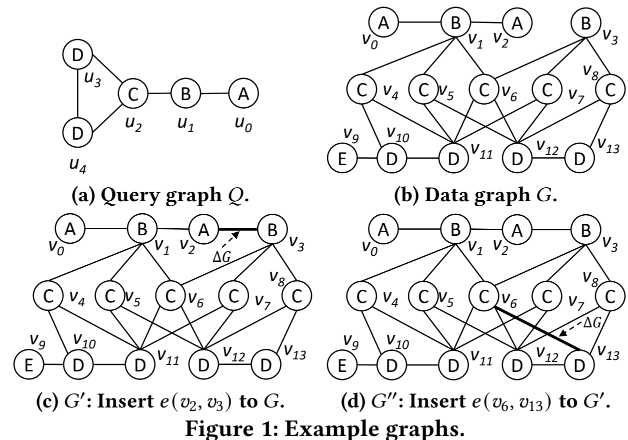



**Figure 1: Example graphs.**

CSM is an important operation in many real-time graph analysis applications, for example, monitoring cycles in transaction graphs to detect merchant frauds in e-commerce [27], matching rumor patterns in message transmission graphs to identify the spread of rumors [37], and spotting system anomalies by analyzing communication logs among computers [20]. Thus, CSM has recently received significant research interests [8, 10, 16, 18, 23]. To facilitate online monitoring of subgraph patterns, we investigate how to further improve the performance of CSM.

Researchers have recently proposed a variety of incremental methods such as SJ-Tree [8], Graphflow [16], TurboFlux [18] and SymBi [23]. All these studies start a search procedure from the updated edge because a match is an incremental result for $\Delta G$ *iff* the match contains the updated edge in $\Delta G$. The search procedure recursively extends partial results, which are mappings from query vertices to data vertices, by binding a *query vertex* (i.e., vertices in $Q$) to a *data vertex* (i.e., vertices in $G$) at each step along a *matching order* (i.e., a sequence of query vertices). In order to reduce the

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551803

search space size, existing research has developed powerful filtering rules to minimize the number of candidate data vertices for each query vertex and optimized matching orders to reduce the number of partial results.

Although these techniques significantly accelerate the CSM performance, we observe that they have a lot of redundant or unnecessary computation (More details are presented in Section 2.4). First, the matching order always starts from the updated edge. This choice may lead to massive *invalid partial results* (i.e., partial results that cannot be extended to final results). Second, the redundant computation is even more severe if $Q$ contains more than one automorphism. Specifically, given edges $e$ and $e'$ that can be mapped to each other in an automorphism $M_Q$, the search procedure for $e$ finds the same set of subgraphs as that for $e'$, and the computation for either edge is sufficient for the other.

In this paper, we propose **RapidFlow** to improve matching order and reduce redundant computation in CSM. Given the updated edge $e(v_a, v_b)$ and a query edge $e(u_a, u_b)$, we propose a novel *query reduction* technique that reduces the problem of finding the set $\Delta\mathcal{M}_{e(u_a,u_b)}$ of incremental matches mapping $e(u_a, u_b)$ to $e(v_a, v_b)$ to that of enumerating all matches of $Q_R$ where $Q_R = Q - \{u_a, u_b\}$ ($Q_R$ is the graph with $u_a, u_b$ as well as adjacent edges removed from $Q$). In particular, we extract a region of $G$ affected by the updated edge such that there is a one-to-one mapping relation between matches in $\Delta\mathcal{M}_{e(u_a,u_b)}$ and those in $\mathcal{M}_{Q_R}$, where $\mathcal{M}_{Q_R}$ is the set of matches of $Q_R$ in the affected region. Thus, we can find $\Delta\mathcal{M}_{e(u_a,u_b)}$ by searching $\mathcal{M}_{Q_R}$. This query reduction approach essentially transforms CSM into a *batch subgraph matching* (BSM) problem, i.e., finding all subgraphs of a data graph identical to a query graph. BSM has been widely studied in the past decade [1–5, 13, 14, 17, 22, 29, 30, 33–35, 38, 39]. We utilize effective filter rules of BSM in extracting the affected region, and take advantage of efficient matching orders by applying BSM to the affected region.

The efficiency of the query reduction approach highly depends on the efficiency of extracting the affected region. We propose an efficient two-level indexing mechanism to speed up the extraction. The first-level index is a query-dependent *global index* $I$ through which we can find all matches of $Q$ in $G$. The maintenance of $I$ is lightweight. The second-level index is an update-dependent *local index* $A$ through which we can find incremental matches for the update by enumerating all matches of $Q_R$ in the affected region. Upon each update, $A$ is constructed by extracting relevant regions from $I$, rather than scanning $G$, which may contain many invalid candidates. The construction of the local index is also efficient.

To eliminate the duplicate computation incurred by automorphisms of $Q$, we design the *dual matching* technique. Specifically, based on automorphisms, we group query edges into a set $\mathcal{X}$ of disjoint sets $X$, called *auto-set*, such that given $X$, the search procedure for each query edge in $X$ finds the same set of subgraphs in $G$. Given an update, we first find incremental matches $\Delta\mathcal{M}_e$ for an arbitrary edge $e \in X$. Then, we obtain incremental matches for the other edges in $X$ by reversing the roles of query graphs and data graphs and permutating query vertices in $\Delta\mathcal{M}_e$ instead of searching in the data graph. This way, we eliminate the redundant computation problem and reduce the number of independent search procedures from $|E(Q)|$ to $|\mathcal{X}|$.

Our experiment results on a variety of datasets show that RapidFlow achieves speedups of up to two orders of magnitude over state-of-the-art CSM methods including SymBi [23] and TurboFlux [18]. Furthermore, RapidFlow dramatically reduces the number of queries that cannot be resolved within a time limit (60 minutes).

In summary, we make the following contributions in this paper:

- We study the CSM problem and propose RapidFlow, an efficient approach to CSM.
- We design a query reduction technique that optimizes matching orders and enables CSM to utilize efficient BSM methods to process graph streams.
- We propose a dual matching technique to eliminate redundant computation incurred by automorphisms of $Q$.
- We conduct detailed experiments to evaluate the effectiveness of RapidFlow.

## 2 BACKGROUND

In this section, we present the background related to this paper.

### 2.1 Preliminaries

We focus on the undirected and labeled graph $g = (V, E)$ in this paper. $V$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. Given a vertex $u \in V$, $N(u)$ is the set of $u$'s neighbors (i.e., vertices adjacent to $u$ in $g$) and $d(u)$ is the degree of $u$ (i.e., $d(u) = |N(u)|$). $L$ is the function mapping a vertex to a label $l$ in a label set $\Sigma$. In our implementation, RapidFlow supports both vertex and edge labels. $Q$ and $G$ denote the query graph and data graph, respectively. We call vertices and edges of $Q$ query vertices and query edges, and those of $G$ data vertices and data edges. $\Delta\mathcal{G}$ is a sequence of graph update operations $(\Delta G_1, \Delta G_2, ...)$ on $G$ where $\Delta G = (\oplus, e)$. $\oplus = +$ is the insertion of an edge $e$, and $\oplus = -$ is the deletion of $e$. Table 1 lists the notations frequently used in this paper.

Definition 2.1 defines *subgraph isomorphism*. We call a subgraph isomorphism a *match* in short. *Batch subgraph matching* (BSM) enumerates the set of all matches of $Q$ in $G$. Given an update $\Delta G \in \Delta\mathcal{G}$, $G'$ is the graph resulted from applying $\Delta G$ to $G$. Then, the set $\Delta\mathcal{M}$ of *incremental matches* on $\Delta G$ is the difference between $\mathcal{M}$ and $\mathcal{M}'$ where $\mathcal{M}$ and $\mathcal{M}'$ represent the matches of $Q$ in $G$ and $G'$, respectively. We define the *continuous subgraph matching* (CSM) problem as follows. Note that both BSM and CSM are NP-hard [10].

**Problem Statement.** Given $Q$, $G$ and $\Delta\mathcal{G}$, continuous subgraph matching is to find the set $\Delta\mathcal{M}$ of incremental matches for each $\Delta G \in \Delta\mathcal{G}$.

*Definition 2.1.* Given graphs $g$ and $g'$, a subgraph isomorphism of $g$ in $g'$ is a **bijective function** $M$ from $V(g)$ to $V(g'')$ where $g''$ is a subgraph of $g'$ such that

(1) $\forall u \in V(g), L(u) = L(M(u))$;
(2) $\forall e(u, u') \in E(g), e(M(u), M(u')) \in E(g'')$.

### 2.2 Related Work

In the following, we discuss related work on batch subgraph matching and continuous subgraph matching to put our work in context.

**Batch subgraph matching** has been widely studied since Ullmann [36] proposed a graph exploration-based backtracking approach in 1976. Existing graph-exploration based methods can be

**Table 1: Notations frequently used in this paper.**

| Notations | Descriptions |
|---|---|
| $g, Q, G$ | graph, query graph, data graph |
| $V(g), E(g)$ | vertex set of $g$, edge set of $g$ |
| $N(u), d(u), L(u)$ | neighbors, degree and label of $u$ |
| $e(u, u')$ | edge between $u$ and $u'$ |
| $I, A$ | global index and local index |
| $C_I(u)$ (or $C_A(u)$) | candidate set of $u$ in global (or local) index |
| $I^u_{u'}(v)$ | $v$'s neighbors in $C_I(u')$ given $v \in C_I(u)$ |
| $A^u_{u'}(v)$ | $v$'s neighbors in $C_A(u')$ given $v \in C_A(u)$ |
| $X, \mathcal{X}$ | auto-set and the set of auto-sets |
| $\Delta\mathcal{M}$ | incremental matches for an update |
| $\Delta\mathcal{M}_e$ | incremental matches mapping $e$ to the updated edge |
| $M$ | mappings from query vertices to data vertices |
| $\varphi$ | matching order |
| $N^{\varphi}_{+}(u)$ | neighbors of $u$ before $u$ in $\varphi$ |
| $\Delta G, \Delta\mathcal{G}$ | graph update, graph stream |
| $\oplus = +/-$ | the insertion/deletion of an edge |

categorized by whether to use indexes or auxiliary structures [33]. The direct-enumeration methods such as Ullmann [36], VF2 [9], QuickSI [30], and RI [5] directly search on $G$ to find all matches. The indexing-enumeration methods, including GADDI [38], SPath [39], and SGMatch [29] construct indices on sub-structures (e.g., paths) of $G$ and use the index to serve all queries. Latest algorithms, including TurboIso [14], CFLMatch [4], CECI [3], DP-iso [13] and VEQ [17], build an auxiliary data structure for $Q$ in a preprocessing step, and then enumerate all matches with the assistance of the data structure. In contrast to these exploration-based methods, the join-based approaches [1, 2, 22, 34, 35] model the problem as a join query and conduct multi-way joins to answer the query.

**Continuous subgraph matching** recently received significant research interests because many real-world graphs change over time. To the best of our knowledge, IncIsoMatch [10] is the first CSM algorithm. The method first extracts a subgraph $G'$ of $G$ within the distance $D$ from the updated edge where $D$ is the diameter of $Q$, then finds the matches $\mathcal{M}/\mathcal{M}'$ of $Q$ in $G'$ with/without the updated edge, and finally gets incremental matches by computing the difference between $\mathcal{M}$ and $\mathcal{M}'$. However, the method is inefficient since it enumerates many stale matches.

To solve the problem, latest algorithms adopt the incremental methodology. SJ-Tree [8] models a CSM query as a multi-way join and evaluates the query with a left-deep tree. SJ-Tree stores all partial results of the join as the index to serve the query. Consequently, the index can take a large amount of memory space because of the exponential number of partial results. Graphflow [16] starts from the updated edge and enumerates all results in $G$. However, many invalid candidates can involve in the computation. As such, TurboFlux [18] constructs a tree-structured index where each node contains the candidates of a query vertex. TurboFlux dynamically maintains the index to keep consistency with each snapshot of $G$, and starts from the updated edge in the index to enumerate incremental matches. SymBi [23] improves the pruning power by constructing a graph-structured index and designs an adaptive ordering method. Nevertheless, these incremental methods start the search from the updated edge to ensure that each reported match is incremental.

**Algorithm 1:** Existing CSM Framework

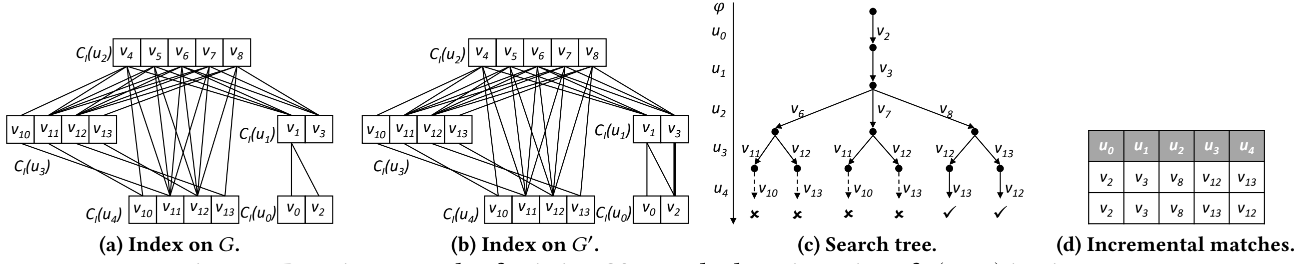
```
   Input: a query graph Q, a data graph G, an update stream Δ𝒢
   Output: incremental matches Δℳ for each ΔG ∈ Δ𝒢
1  I ← build an index based on Q and G;
2  foreach ΔG = (⊕, e) ∈ Δ𝒢 do
3      if ⊕ is + then
4          Add e to G and update I;
5          FindIncrementalMatch(Q, I, e);
6      else
7          FindIncrementalMatch(Q, I, e);
8          Remove e from G and update I;
9  Procedure FindIncrementalMatch(Q, I, e(v_a, v_b))
10     Δℳ ← {};
11     foreach e(u_a, u_b) ∈ E(Q) do
12         if L(u_a) = L(v_a) and L(u_b) = L(v_b) then
13             φ ← generate a matching order beginning with u_a, u_b;
14             M ← {(u_a, v_a), (u_b, v_b)};
15             Δℳ_{e(u_a,u_b)} ← Enumerate(φ, I, M, 3);
16             Δℳ ← Δℳ ∪ Δℳ_{e(u_a,u_b)};
17     Output Δℳ;
18 Procedure Enumerate(φ, I, M, i)
19     if i = |φ| + 1 then Output M, return;
20     else if i = 1 then u ← φ[i], C_M(u) ← C_I(u);
21     else u ← φ[i], C_M(u) ← ⋂_{u'∈N^φ_+(u)} I^{u'}_u(M(u'));
22     foreach v ∈ C_M(u) do
23         if v is not visited then
24             Add (u, v) to M;
25             Enumerate(φ, I, M, i + 1);
26             Remove (u, v) from M;
```

In addition to the generic CSM methods targeting at queries of arbitrary structures, there are also studies about CSM on specific query types such as paths [26, 32]. Casqd [24] finds cliques, stars and bi-cliques in graph streams. GraphS [27] detects cycles with length constraints. Moreover, researchers proposed approximate algorithms [7, 10, 11, 15, 31] because finding exact results can be time-consuming due to the hardness of the problem and subgraph isomorphism may be too restrictive for some applications. Additionally, there are solutions on optimizing the processing of multiple queries [21]. In this paper, we focus on the problem of finding exact results of a single query of arbitrary structures.

## 2.3 A Framework for Existing CSM Approaches

We review existing work on CSM and find that they follow the same algorithmic framework as illustrated in Algorithm 1. The differences are in rules for pruning candidates and methods of generating the matching order. Given $Q$ and $G$, Line 1 builds an index $I$, which maintains a candidate set $C_I(u)$ for $u \in V(Q)$ and records edges between $C_I(u)$ and $C_I(u')$ if $e(u, u') \in E(Q)$. The use of $I$ is to rule out data vertices unrelated to the query, and the search procedure enumerates results based on $I$ instead of $G$.

In particular, given insertion of $e$, Lines 4-5 first update $G$ and $I$ and then find incremental matches. FindIncrementalMatches executes a search procedure for each $e(u_a, u_b) \in E(Q)$ (Lines 11-16). If vertices pass the label filter at Line 12, then Line 13 generates a




(a) Index on $G$. (b) Index on $G'$. (c) Search tree. (d) Incremental matches.

**Figure 2: Running example of existing CSM methods on insertion of $e(v_2, v_3)$ in Figure 1.**

matching order $\varphi$, which begins with $u_a$ and $u_b$, and Line 14 initializes $M$ to records mappings from query vertices to data vertices. After that, ENUMERATE finds the set $\Delta\mathcal{M}_{e(u_a,u_b)}$ of incremental matches mapping $e(u_a, u_b)$ to $e(v_a, v_b)$. Line 15 sets the initial recursive depth to 3 because Line 14 has mapped $u_a$ and $u_b$ to $v_a$ and $v_b$ correspondingly.

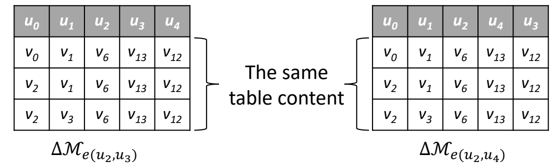
Overall, ENUMERATE uses the backtracking search that extends the partial result $M$ by mapping a query vertex to a candidate along $\varphi$ to find matches. The integer $i$ is the recursive depth. $\varphi[i]$ is the $i$th vertex in $\varphi$. For ease of presentation, we let $i$ to start from 1 instead of 0. Given $e(u, u') \in E(Q)$ and $v \in C_I(u)$, $I^u_{u'}(v) = N(v) \cap C_I(u')$ (i.e., the neighbors of $v$ who are in the candidate set of $u'$). For the first vertex $u$ in $\varphi$, Line 20 sets $C_M(u)$ to $C_I(u)$. Otherwise, Line 21 sets $C_M(u)$ to the set of common neighbors of candidates who are mapped to query vertices $u' \in N^\varphi_+(u)$ where $N^\varphi_+(u)$ is the set of $u$' neighbors before $u$ in $\varphi$. Lines 22-26 loop over $C_M(u)$ to extend $M$. If all query vertices are mapped in $M$, then Line 19 outputs $M$. During the enumeration, each partial result $M$ containing $i$ mappings is a match of $Q[\varphi[1 : i]]$ in $G$ where $Q[\varphi[1 : i]]$ is the vertex-induced subgraph of $Q$ on the first $i$ vertices in $\varphi$. Note that ENUMERATE is a common method for searching matches, which is used in both CSM and BSM [33]. RapidFlow uses this procedure in the enumeration of results as well.

## 2.4 Problems in Existing Work

Despite that existing CSM methods significantly accelerate some queries, the common framework has inherent flaws. In the following, we use two running examples to illustrate these issues.

*1. The matching order is required to begin with query edges mapped to the updated edge, which may lead to many invalid partial results.* Given $Q$ and $G$ in Figure 1, the index $I$ is illustrated in Figure 2a. In the example, $C_I(u)$ is generated based on the vertex label. When inserting $e(v_2, v_3)$ in Figure 1c, we first update $I$ in Figure 2b to keep it consistent with $G'$. As $u_0$ and $u_1$ have the same label as $v_2$ and $v_3$, we start a search procedure for $e(u_0, u_1)$ with $\varphi$ beginning with $u_0$ and $u_1$. Suppose that $\varphi = (u_0, u_1, u_2, u_3, u_4)$. Figure 2c visualizes the enumeration procedure where a node denotes a partial result and an edge represents a mapping from a query vertex to a data vertex. The enumeration explores the search tree in a depth-first search order. Ticks and crosses denote matches and invalid results, respectively. Finally, we find two incremental matches for the update, and the other four invalid search paths fail.

A simple idea of reducing the search space size is to optimize the matching order as $\varphi' = (u_4, u_3, u_2, u_1, u_0)$ because triangles with labels $(C, D, D)$ are fewer than paths with labels $(A, B, C, D)$ in $G'$. However, this method cannot outperform existing CSM approaches since many matches of $Q$ do not contain the updated edge $e(v_2, v_3)$ and the enumeration with $\varphi'$ leads to many stale matches (e.g., $\{(u_0, v_0), (u_1, v_1), (u_2, v_4), (u_3, v_{10}), (u_4, v_{11})\}$). Thus, existing methods force $\varphi$ to begin with $u_0$ and $u_1$, which are mapped to the newly inserted edge $e(v_2, v_3)$. In a word, starting the search from the updated edge can ensure that each reported match is an incremental result, but downside is that it can lead to many invalid partial results.



| $u_0$ | $u_1$ | $u_2$ | $u_3$ | $u_4$ |
|---|---|---|---|---|
| $v_0$ | $v_1$ | $v_6$ | $v_{13}$ | $v_{12}$ |
| $v_2$ | $v_1$ | $v_6$ | $v_{13}$ | $v_{12}$ |
| $v_2$ | $v_3$ | $v_6$ | $v_{13}$ | $v_{12}$ |

$\Delta\mathcal{M}_{e(u_2,u_3)}$

The same table content

| $u_0$ | $u_1$ | $u_2$ | $u_4$ | $u_3$ |
|---|---|---|---|---|
| $v_0$ | $v_1$ | $v_6$ | $v_{13}$ | $v_{12}$ |
| $v_2$ | $v_1$ | $v_6$ | $v_{13}$ | $v_{12}$ |
| $v_2$ | $v_3$ | $v_6$ | $v_{13}$ | $v_{12}$ |

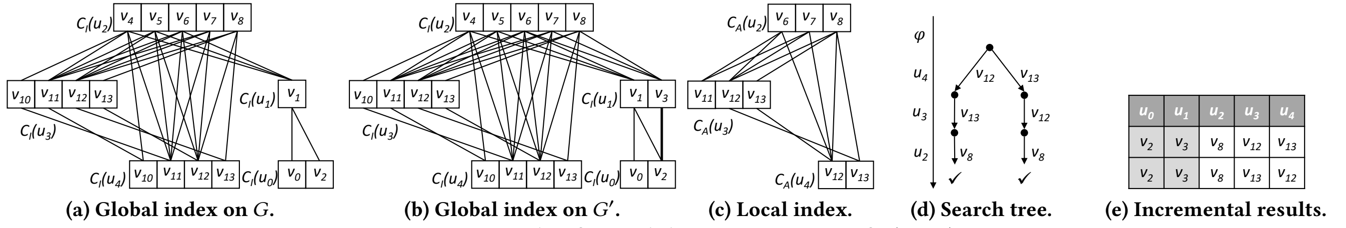
$\Delta\mathcal{M}_{e(u_2,u_4)}$

**Figure 3: Incremental matches generated by existing CSM methods given insertion of $e(v_6, v_{13})$ to $G'$ in Figure 1d.**

*2. Existing approaches may perform redundant computation if $Q$ has more than one automorphism.* In Figure 1a, $M_Q = \{(u_0, u_0), (u_1, u_1), (u_2, u_2), (u_3, u_4), (u_4, u_3)\}$ is an automorphism of $Q$. $e(u_2, u_3)$ can be mapped to $e(u_2, u_4)$ in $M_Q$. When inserting $e(v_6, v_{13})$ to $G'$ in Figure 1d, we find incremental matches in Figure 3. $\Delta\mathcal{M}_{e(u_2,u_3)}$, which is the set of incremental matches mapping $e(u_2, u_3)$ to $e(v_6, v_{13})$, is reported by the search procedure for $e(u_2, u_3)$, and $\Delta\mathcal{M}_{e(u_2,u_4)}$ is found by the search procedure for $e(u_2, u_4)$. However, the contents of the two tables are the same, which indicates that the two search procedures find the same set of subgraphs in $G''$. This duplication of results indicates the redundancy in the search process. Table 2 lists the number of queries containing more than one automorphism in our benchmark consisting of four datasets, amazon(*az*), livejournal(*lj*), netflow(*nf*), and lsbench(*ls*). The detailed statistics of the datasets is listed in Table 3 in Section 6.1. In Table 2, we can see that this redundancy issue frequently appears in the workload.

**Table 2: The number of queries with more than one automorphism in our benchmark. A query set on a dataset contains 100 queries each of which has 6 vertices. Based on graph density, we categorized queries into tree, sparse and dense.**

| Tree | | | | Sparse | | | | Dense | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***az*** | ***lj*** | ***nf*** | ***ls*** | ***az*** | ***lj*** | ***nf*** | ***ls*** | ***az*** | ***lj*** | ***nf*** | ***ls*** |
| 34 | 8 | 59 | 46 | 12 | 2 | 87 | 52 | 44 | 2 | 30 | 77 |




**(a) Global index on $G$.** **(b) Global index on $G'$.** **(c) Local index.** **(d) Search tree.** **(e) Incremental results.**

**Figure 4: Running example of RapidFlow on insertion of $e(v_2, v_3)$ in Figure 1.**

## 3 AN OVERVIEW OF RAPIDFLOW

In order to address the issues in Section 2.4, we propose an end-to-end CSM approach, called RapidFlow. Algorithm 2 gives an overview. Overall, RapidFlow has two stages. In the offline stage, we group query edges into a set $\mathcal{X}$ of disjoint sets based on automorphisms of $Q$ and build a global index $I$ where we can find all matches of $Q$ in $G$ (Lines 1-2). Given an update, if the operation is insertion, then we add the edge to $G$ and update the global index to keep it consistent with $G$ (Lines 5-6). This update on $I$ is light-weight given $\Delta G$ is small. After that, we find incremental matches based on $I$ (Line 7). In contrast, for the deletion operation, we reverse the order of these operations (Lines 9-11). As the global index $I$ is consistent with $G$, we directly invoke FindIncrementalMatches to find incremental matches (i.e., matches containing the edge to be deleted) based on $I$. After that, we remove the edge from $G$ and update $I$ to keep its consistency. Thus, FindIncrementalMatches is *symmetric*, i.e., tackling the insertion and deletion with the same logic. Therefore, we focus on the insertion of an edge in the following of this paper for brevity.

Given the updated data edge $e(v_a, v_b)$, we find incremental matches for each set $X \in \mathcal{X}$ seperately (Lines 14-23). Given $e(u_a, u_b) \in X$, we propose the query reduction technique that converts the problem of finding the set $\Delta\mathcal{M}_{e(u_a,u_b)}$ of incremental matches for $e(u_a, u_b)$ to that of enumerating all matches of $Q_R$ where $Q_R = Q - \{u_a, u_b\}$ in the region affected by the update. Specifically, we generate a local index $A$ for $Q_R$ from the global index given the update. $A$ maintains a candidate set for each vertex $u \in V(Q_R)$ and edges between candidates. Moreover, $A$ guarantees that there is a one-to-one mapping from matches of $Q_R$ in $A$ to those in $\Delta\mathcal{M}_{e(u_a,u_b)}$. Therefore, we can enumerate all matches of $Q_R$ and then generate $\Delta\mathcal{M}_{e(u_a,u_b)}$ (Lines 20-21). Line 20 sets the initial recursive depth to 1 since the input mapping set is empty. This way, we can evaluate the query with any matching order and use the well-studied BSM techniques to process the stream.

After that, we use the dual matching technique to find incremental matches for remaining query edges in $X$ to obtain $\Delta\mathcal{M}_X$, which is the set of incremental matches mapping $e(v_a, v_b)$ to query edges in $X$. The dual matching technique finds incremental matches by permutating query vertices in matches in $\Delta\mathcal{M}_{e(u_a,u_b)}$, rather than executing the recursive search in $G$. This way, we eliminate the redundant computation incurred by automorphisms of $Q$. Example 3.1 presents a running example of RapidFlow.

*Example 3.1.* Figure 4a illustrates the global index $I$ given $Q$ and $G$ in Figure 1. Given the update in Figure 1c, RapidFlow first updates $I$ to keep it consistent with $G'$ in Figure 4b. Next, RapidFlow extracts the local index $A$ where each match of $Q_R$ corresponds to an incremental match mapping $e(u_0, u_1)$ to $e(v_2, v_3)$. After that, we find all matches $\mathcal{M}_{Q_R}$ of $Q_R$ in $A$ with the matching order $(u_4, u_3, u_2)$ in Figure 4d. Finally, we obtain incremental matches based on $\mathcal{M}_{Q_R}$ given the initial mapping $\{(u_0, v_2), (u_1, v_3)\}$ in Figure 4e. Suppose that $e(v_2, v_3)$ is to be deleted from $G'$ in Figure 1c and the data graph will evolve from $G'$ in Figure 1c to $G$ in Figure 1b. RapidFlow will directly extract the local index in Figure 4c from the global

**Algorithm 2:** An Overview of RapidFlow

**Input:** a query graph $Q$, a data graph $G$, an update stream $\Delta\mathcal{G}$
**Output:** incremental matches $\Delta\mathcal{M}$ for each $\Delta G \in \Delta\mathcal{G}$

/* The offline stage. */
1 $I \leftarrow$ BuildGlobalIndex($Q, G$);
2 $\mathcal{X} \leftarrow$ GenerateAutoSet($Q$);
/* The online stage. */
3 **foreach** $\Delta G = (\oplus, e) \in \Delta\mathcal{G}$ **do**
4  **if** $\oplus$ *is* + **then**
5   $G \leftarrow G \oplus \Delta G$;
6   UpdateGlobalIndex($Q, G, I, \oplus, e$);
7   FindIncrementalMatch($Q, I, e, \mathcal{X}$);
8  **else**
9   FindIncrementalMatch($Q, I, e, \mathcal{X}$);
10   $G \leftarrow G \oplus \Delta G$;
11   UpdateGlobalIndex($Q, G, I, \oplus, e$);

12 **Procedure** FindIncrementalMatch($Q, I, e(v_a, v_b), \mathcal{X}$)
13  $\Delta\mathcal{M} \leftarrow \{\}$;
14  **foreach** $X \in \mathcal{X}$ **do**
15   $e(u_a, u_b) \leftarrow$ an arbitrary edge in $X$;
16   $Q_R \leftarrow Q - \{u_a, u_b\}$;
17   $A \leftarrow$ BuildLocalIndex($Q_R, I, e(u_a, u_b), e(v_a, v_b)$);
18   **if** *there are empty candidate sets in* $A$ **then** Continue;
19   $\varphi \leftarrow$ generate a matching order of $Q_R$;
20   $\mathcal{M}_{Q_R} \leftarrow$ Enumerate($\varphi, A, \{\}, 1$);
21   $\Delta\mathcal{M}_{e(u_a,u_b)} \leftarrow \{\{(u_a, v_a), (u_b, v_b)\} \cup M | M \in \mathcal{M}_{Q_R}\}$;
22   $\Delta\mathcal{M}_X \leftarrow$ DualMatch($\Delta\mathcal{M}_{e(u_a,u_b)}, X$);
23   $\Delta\mathcal{M} \leftarrow \Delta\mathcal{M} \cup \Delta\mathcal{M}_X$;
24  Output $\Delta\mathcal{M}$;

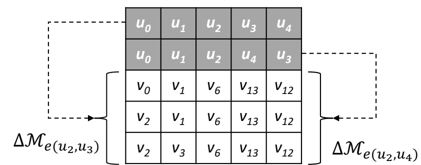



**Figure 5: Incremental matches generated by dual matching given insertion of $e(v_6, v_{13})$ to $G'$ in Figure 1d.**

index in Figure 4b and enumerate incremental matches with the same procedure as processing the insertion of $e(v_2, v_3)$ in Figure 4d. After that, RapidFlow will delete $e(v_2, v_3)$ from $G'$ and update the global index in Figure 4b to that in Figure 4a.

Figure 5 illustrates incremental matches for the update in Figure 1d. After finding the set $\Delta\mathcal{M}_{e(u_2,u_3)}$ of incremental matches mapping $e(u_2, u_3)$ to $e(v_6, v_{13})$, we generate $\Delta\mathcal{M}_{e(u_2,u_4)}$ by permutating the sequence of query vertices instead of issuing a search procedure for $e(u_2, u_4)$, which avoids the redundant computation.

# 4 QUERY REDUCTION

We introduce the query reduction technique in this section.

## 4.1 Reduce CSM to BSM

According to Definition 2.1, each incremental match contains the updated edge.

FACT 1. *Given insertion of $e(v_a, v_b)$, the set $\Delta\mathcal{M}$ of incremental matches is the set of matches $M$ of $Q$ in $G$ such that $e(M^{-1}(v_a), M^{-1}(v_b))$ belongs to $E(Q)$ where $M^{-1}$ is the inverse function of $M$.*

Based on Fact 1, a straightforward incremental CSM method is to 1) start a search procedure for each $e(u_a, u_b)$ to find the set $\Delta\mathcal{M}_{e(u_a,u_b)}$ of matches mapping $e(u_a, u_b)$ to $e(v_a, v_b)$; and 2) obtain $\Delta\mathcal{M}$ by computing $\bigcup_{e\in E(Q)} \Delta\mathcal{M}_e$. As $\Delta\mathcal{M}_{e(u_a,u_b)}$ generally accounts for a small portion of matches of $Q$ in $G$, the search for $e(u_a, u_b)$ starts the enumeration by mapping $e(u_a, u_b)$ to $e(v_a, v_b)$ to ensure that each match reported in the ENUMERATE procedure (see Algorithm 1) maps $e(u_a, u_b)$ to $e(v_a, v_b)$. Thus, this method must put $u_a, u_b$ at the beginning of the matching order. However, this requirement may lead to many invalid partial results, as illustrated in Section 2.4. To solve this problem, we propose the *query reduction* technique that enables the enumeration of $\Delta\mathcal{M}_e$ with any matching order.

Intuitively, matches in $\Delta\mathcal{M}_{e(u_a,u_b)}$ appear in the region of $G$ around the updated edge $e(v_a, v_b)$ because they must contain $e(v_a, v_b)$. Therefore, if we can extract an affected region from $G$ given the update such that each match in the region is a match in $\Delta\mathcal{M}_{e(u_a,u_b)}$, then we can obtain $\Delta\mathcal{M}_{e(u_a,u_b)}$ by enumerating all matches in this region.

Specifically, each match $M \in \Delta\mathcal{M}_{e(u_a,u_b)}$ maps $u_a, u_b$ to $v_a, v_b$, respectively, and therefore we only need to determine candidate sets for remaining query vertices. According to Definition 2.1, query vertices $u$ adjacent to $u_a$ (resp. $u_b$) must be mapped to the neighbors of $v_a$ (resp. $v_b$) in $M$. Thus, the candidate set $C(u)$ is a subset of $N(v_a)$ (resp. $N(v_b)$). Similarly, the neighbors $u'$ of $u$ must be mapped to the neighbors of candidates $v$ in $C(u)$, and therefore $C(u')$ is a subset of $\bigcup_{v\in C(u)} N(v)$.

As a result, we can first obtain candidate sets for query vertices adjacent to $u_a, u_b$, then iteratively generate candidate sets for the other query vertices based on the candidate sets of their neighbors, and finally map query vertices excluding $u_a, u_b$ to candidates to find $\Delta\mathcal{M}_{e(u_a,u_b)}$. In short, we find $\Delta\mathcal{M}_{e(u_a,u_b)}$ by enumerating matches of $Q_R = Q - \{u_a, u_b\}$ from candidate sets generated based on the update.

**Algorithm 3:** Global Index

```
1  Procedure BuildGlobalIndex(Q, G)
2      foreach u ∈ V(Q) do
3          C_I(u) ← {v ∈ V(G) | L(u) = L(v) ∧ NLF(u, v) is true};
4      foreach e(u, u') ∈ E(Q) do
5          foreach v ∈ C_I(u) do
6              I^u_{u'}(v) ← N(v) ∩ C_I(u');
7      return I;
   /* Maintain the index given an update.      */
8  Procedure UpdateGlobalIndex(Q, G, I, ⊕, e(v_a, v_b))
9      foreach {(u, u'), (v, v')} ∈ E(Q) × {(v_a, v_b), (v_b, v_a)} do
10         if v ∈ C_I(u) and v' ∈ C_I(u') then
11             Add v' to I^u_{u'}(v) and add v to I^{u'}_u(v');
12     ΔC_I ← {};
13     foreach (u, v) ∈ V(Q) × {v_a, v_b} do
14         if L(u) = L(v) and v ∉ C_I(u) and NLF(u, v) is true then
15             Add v to C_I(u) and add (u, v) to ΔC_I;
16     foreach u' ∈ N(u) where (u, v) ∈ ΔC_I do
17         I^u_{u'}(v) ← N(v) ∩ C_I(u');
18         Add v to I^{u'}_u(v') given v' ∈ I^u_{u'}(v);
```

## 4.2 Two-Level Indexing Mechanism

The efficiency of the query reduction technique highly depends on the efficiency of extracting the affected region. To improve the performance, we design a two-level indexing mechanism to obtain the affected region.

**Global Index.** The goal of the first-level index, called the *global index*, is to rule out data vertices irrelevant to the query. Specifically, the global index $I$ is query-dependent; it maintains a candidate set $C_I(u)$ for each query vertex $u$ and records edges between candidates. The candidate set $C_I(u)$ is *global complete* (Definition 4.1) in terms of all matches of $Q$ in $G$, where $G$ is the data graph after the insertion.

*Definition 4.1.* Given $Q$ and $G$, the global complete candidate set $C_I(u)$ for $u \in V(Q)$ is a set of data vertices $v$ such that if a mapping $(u, v)$ appears in a match of $Q$ in $G$, then $v$ must belong to $C_I(u)$. If $C_I(u)$ is global complete for each $u \in V(Q)$, then $I$ is global complete.

Given $Q$ and $G$, we build $I$ in the offline and dynamically update it to keep its completeness online. Algorithm 3 depicts the construction and update of the global index. Given $Q$ and $G$, BUILDGLOBALINDEX generates a candidate set for each query vertex based on the *neighbor label frequency* (NLF) filter (Lines 2-3), which is a widely used filtering rule [33]. Particularly, given $u \in V(Q)$ and $v \in V(G)$, NLF requires that given $l \in L(N(u))$, $|N(u,l)| \leqslant |N(v,l)|$ where $L(N(u)) = \{L(u')|u' \in N(u)\}$ (i.e., the set of labels of $u$'s neighbors) and $N(u, l) = \{u' \in N(u)|L(u') = l\}$ (i.e., the set of $u$'s neighbors with label $l$). Next, Lines 4-6 record edges between candidates in $C_I(u)$ and $C_I(u')$ given $e(u, u') \in E(Q)$. $I^u_{u'}(v)$ is the set of $v$'s neighbors in $C_I(u')$.

UPDATEGLOBALINDEX keeps the completeness of $I$ given the updated edge $e(v_a, v_b)$. Lines 9-18 presents the index update for the insertion. Given $e(u, u') \in E(Q)$, if $v$ and $v'$ belong to $C_I(u)$ and $C_I(u')$, respectively, then Lines 9-11 add $e(v, v')$ to $I$. Lines 13-15 compute the modification on candidate sets. As the insertion of

**Algorithm 4:** Local Index

```
1  Procedure BuildLocalIndex(Q_R, I, e(u_a, u_b), e(v_a, v_b))
2      if v_a ∉ C_I(u_a) or v_b ∉ C_I(u_b) then return;
3      M ← {(u_a, v_a), (u_b, v_b)};
4      Φ ← V(Q_R) ∩ (N_Q(u_a) ∪ N_Q(u_b));
5      foreach u ∈ Φ do
6          C_A(u) ← ⋂_{u'∈N_Q(u)∩{u_a,u_b}} I_u^{u'}(M(u')) − {v_a, v_b};
7      δ ← sort vertices u ∈ Φ in the ascending order of |C_A(u)|;
8      foreach u ∈ Φ along the order of δ do
9          foreach u' ∈ N_+^δ(u) do
10             C_A(u) ← C_A(u) ⋂(⋃_{v∈C_A(u')} I_u^{u'}(v));
11     Φ̄ ← V(Q_R) − Φ;
12     while Φ̄ ≠ ∅ do
13         u ← arg max_{u'∈Φ̄} |N(u) − Φ̄|;
14         C_A(u) ← C_I(u) − {v_a, v_b};
15         foreach u' ∈ N(u) − Φ̄ do
16             Do the same operation as Line 10;
17         Remove u from Φ̄;
18     foreach e(u, u') ∈ E(Q_R) do
19         foreach v ∈ C_A(u) do
20             A_{u'}^u(v) ← I_{u'}^u(v) ∩ C_A(u');
21     return A;
```

$e(v_a, v_b)$ updates $N(v_a)$ and $N(v_b)$ in $G$, we only check whether $v_a$ and $v_b$ can be inserted into certain candidate sets based on NLF. If so, we add it to $C_I(u)$ and record the update in $\Delta C_I$. Next, we update edges between candidates in $I$ correspondingly (Lines 16-18).

*Example 4.2.* Figure 4a demonstrates $I$ given $Q$ and $G$ in Figure 1. $I_{u_0}^{u_1}(v_1) = \{v_0, v_2\}$. Although $L(v_3) = L(u_1)$ in Figure 1b, $v_3 \notin C_I(u_1)$ because $|N(v_3, A)| = 0$, which is less than $|N(u_1, A)| = 1$. Given insertion of $e(v_2, v_3)$ in Figure 1c, $NLF(u_1, v_3)$ is true. Therefore, we add $v_3$ to $C_I(u_1)$ and update edges between candidates in Figure 4b.

**Local Index.** The second-level index, called the *local index*, is built on top of the global index for each update. In particular, the local index $A$ is update-dependent, which keeps a candidate set $C_A(u)$ for $u \in V(Q_R)$ and maintains edges between candidate sets $C_A(u)$ and $C_A(u')$ if $e(u, u') \in E(Q_R)$. $C_A(u)$ is local complete (Definition 4.3). Therefore, the local index is the affected region where we can find incremental matches.

*Definition 4.3.* Given $Q$, $G$, the updated edge $e(v_a, v_b)$ and the query edge $e(u_a, u_b)$ that maps to $e(v_a, v_b)$, the local complete candidate set $C_A(u)$ for $u \in V(Q_R)$ is a set of data vertices $v$ such that if a mapping $(u, v)$ belongs to a match in the set $\Delta\mathcal{M}_{e(u_a,u_b)}$ of incremental matches mapping $e(u_a, u_b)$ to $e(v_a, v_b)$, then $v$ must belong to $C_A(u)$. If $C_A(u)$ is local complete for each $u \in V(Q_R)$, then $A$ is local complete.

The local index is generated for each update and immediately destroyed after the search procedure. Algorithm 4 presents the generation method of the local index. Given a query vertex $u$, $N_Q(u)$ and $N(u)$ denote the neighbors of $u$ in $Q$ and $Q_R$, respectively. $e(v_a, v_b)$ is the updated data edge and $e(u_a, u_b)$ is the target query edge. $M$ records initial mappings (Line 3). Lines 4-6 compute $C_A(u)$ based on $v_a$ and $v_b$ where $u \in \Phi$ (i.e., query vertices adjacent to $u_a, u_b$). After that, Lines 7-10 prune candidate sets $C_A(u)$ for $u \in \Phi$ based on the filtering rule: we can remove $v$ from $C_A(u)$ without breaking its completeness if there exists $u' \in N_+^{\delta}(u)$ such that $v$ has no neighbor in $C_A(u')$ where $N_+^{\delta}(u)$ is the set of vertices positioned before $u$ in a sequence $\delta$ of $\Phi$. In particular, $\delta$ prioritizes query vertices with fewer candidates to utilize small candidate sets to prune large ones. Given $u' \in N_+^{\delta}(u)$, we first compute the union of neighbors of candidates in $C_A(u')$ based on $I$ (i.e., $\bigcup_{v \in C_A(u')} I_u^{u'}(v)$ at Line 10), and then intersect the union with $C_A(u)$ to eliminate invalid candidates. Lines 11-17 generate candidate sets for $u \in \overline{\Phi}$ (i.e., query vertices not adjacent to $u_a, u_b$). At each step, we select $u \in \overline{\Phi}$ who has the largest number of neighbors that have candidate sets generated. Based on Definitions 4.1 and 4.3, $C_I(u)$ must be local complete. As such, Lines 14-16 initialize $C_A(u)$ as $C_I(u)$ and prune it with the same method as Line 10. Finally, we record edges between candidates in $C_A(u)$ and $C_A(u')$ if $e(u, u') \in E(Q_R)$.

*Example 4.4.* Given the updated edge $e(v_2, v_3)$ and the query edge $e(u_0, u_1)$ mapped to $e(v_2, v_3)$, the mapping $M$ is initialized to $\{(u_0, v_2), (u_1, v_3)\}$. Figure 4c presents the local index given $M$. As $u_2$ is adjacent to $u_1$, $C_A(u_2) = I_{u_2}^{u_1}(M(u_1)) = \{v_6, v_7, v_8\}$. Next, we generate $C_A(u_3)$ by pruning $C_I(u_3)$ based on $C_A(u_2)$. $v_{10}$ is invalid since $v_{10}$ has no neighbor in $C_A(u_2)$. Thus, $C_A(u_3) = \{v_{11}, v_{12}, v_{13}\}$. Next, we generate $C_A(u_4)$ by pruning $C_I(u_4)$ based on $C_A(u_2)$ and $C_A(u_3)$. $v_{10}$ has no neighbor in $C_A(u_2)$, and $v_{11}$ has no neighbor in $C_A(u_3)$. Therefore, $C_A(u_4) = \{v_{12}, v_{13}\}$. Finally, we add edges between candidate sets if $e(u, u') \in E(Q_R)$ where $Q_R$ is the triangle in $Q$.

## 4.3 Analysis

In the following, we analyze the time and space cost, and discuss the connections with existing work.

**Time and Space.** We first analyze the cost of the global index. Given $u \in V(Q)$ and $v \in V(G)$, we perform NLF check on $N(v)$. Thus, the time complexity of generating candidate sets (Lines 2-3 in Algorithm 3) is $O(\sum_{u \in V(Q)} \sum_{v \in V(G)} d(v)) = O(|V(Q)| \times |E(G)|)$. Given two sets $S_1, S_2$ where $|S_1| \leqslant |S_2|$, the cost of set intersection on them is $O(|S_1|)$ [1]. Then, the time complexity of recording edges between candidates (Lines 4-6 in Algorithm 3) is $O(\sum_{e(u,u') \in E(Q)} \sum_{v \in C_I(u)} d(v)) = O(|E(Q)| \times |E(G)|)$. Thus, the time complexity of building the global index is $O(|E(Q)| \times |E(G)|)$. The space complexity is $O(|V(Q)| \times |V(G)| + |E(Q)| \times |E(G)|)$.

The neighbor set is sorted in the index. The cost of adding an edge $e(v, v')$ to $I_{u'}^u(v)$ is $O(\log |I_{u'}^u(v)|) = O(d(v))$. For simplicity, we use the average degree $d$ of $G$ in the analysis. Thus, the cost of Lines 9-11 is $O(|E(Q)| \times \log d)$. The cost of updating candidate sets and neighbor sets is $O(|V(Q)| \times d + |V(Q)| \times d \times \log d)$. Therefore, the time cost of updating the global index given an update is $O(|E(Q)| \times \log d + |V(Q)| \times d \times \log d)$.

Next, we analyze the cost of the local index. Given $e(u, u') \in E(Q)$, the cost of pruning $C_A(u)$ with $C_A(u')$ is $O(\sum_{v \in C_I(u')} |I_u^{u'}(v)|) = O(|I_u^{u'}|)$ where $|I_u^{u'}|$ is the number of edges between $C_I(u)$ and $C_I(u')$. Lines 8-17 in Algorithm 4 utilize each edge in $Q_R$ to generate candidate sets. Thus, the time complexity is $O(\sum_{e(u,u') \in E(Q_R)} |I_u^u|)$. The time complexity of recording edges is the same as the pruning. Therefore, the time and space complexity of constructing the local

index is $O(\sum_{e(u,u')\in E(Q_R)} |I_u^u|)$. In practice, the construction of the local index is very efficient as only a small portion of vertices in $I$ is involved in the computation for each update.

**Discussion.** Given $e(u_a, u_b)$, $Q_R = Q - \{u_a, u_b\}$ can be a disconnected graph. The Enumerate procedure in Algorithm 1 can handle disconnected graphs by setting $C_M(u)$ to $C_A(u)$ at Line 21 if $N_+^{\varphi}(u) = \emptyset$.

Latest algorithms [18, 23] build an index to serve the enumeration as presented in Section 2.3. Such an index has the same structure as the global index. These algorithms keep candidate sets global complete given the stream, and utilize advanced filtering rules to prune invalid candidates. Formally, the rule is: given $u \in V(Q)$ and $v \in C_I(u)$, $v$ has at least one neighbor in $C_I(u')$ for each $u' \in N(u)$. This rule is widely used in both CSM and BSM approaches [33]. The cost of maintaining the index for each update is $O(|E(Q)| \times |E(G)|)$ in SymBi [23], the latest CSM algorithm. Although these rules can be applied to our global index, we use a simpler filtering rule (i.e., NLF) because 1) the overhead of complex filtering rules may offset the benefit on short-running queries; and 2) the simple filtering rule is sufficient for pruning candidate sets in the global index.

In summary, our two-level indexing mechanism consisting of the global index and the local index has a lower maintenance cost than existing CSM approaches because the global index $I$ uses a simple filtering rule and the affected part for each update in $I$ is small. In case an affected region is large, our index maintenance cost can be higher than existing methods because of the local index. However, the benefit of our approach offsets the overhead because there are many incremental results in a large affected region and therefore the enumeration time dominates the cost of processing the update. We evaluate the query reduction technique in Section 6.3.1.

# 5 DUAL MATCHING

We introduce the dual matching technique in this section.

## 5.1 Reverse Roles of Query and Data Graphs

Given a match from $Q$ to $G$, the subgraph $G'$ consisting of the matched edges of $G$ is isomorphic to $Q$. Let $Q'$ be a graph isomorphic to $Q$. $G'$ is isomorphic to $Q'$ because the subgraph isomorphism relation is transitive. We start $|E(Q)|$ search procedures each of which finds the set $\Delta\mathcal{M}_e$ of incremental matches mapping $e \in E(Q)$ to the updated data edge. Intuitively, if $e'$ can be mapped to $e$ in an automorphism of $Q$, then each subgraph of matches in $\Delta\mathcal{M}_e$ appear in a match in $\Delta\mathcal{M}_{e'}$. In other words, there is a one-to-one mapping relationship between matches in $\Delta\mathcal{M}_e$ and $\Delta\mathcal{M}_{e'}$, which is described formally in the following proposition.

Proposition 5.1. *Given an automorphism $M_Q$ of $Q$, $e$ denotes $e(u_a, u_b) \in E(Q)$ and $e'$ denotes $e(M_Q(u_a), M_Q(u_b)) \in E(Q)$. Then, $\Delta\mathcal{M}_{e'}$ is equal to $\{M \circ M_Q | M \in \Delta\mathcal{M}_e\}$ where $\circ$ is the function composition operation.*

Proof. Let $\mathcal{M}$ be $\{M \circ M_Q | M \in \Delta\mathcal{M}_e\}$. We first show $\mathcal{M} \subseteq \Delta\mathcal{M}_{e'}$. Suppose $M' = M \circ M_Q$ where $M \in \Delta\mathcal{M}_e$. As $M_Q$ and $M$ are bijective functions, $M'$ must also be a bijective function and $M'(u) = M(M_Q(u))$ given $u \in V(Q)$. Given $e(u, u') \in E(Q)$, we have $e(M_Q(u), M_Q(u')) \in E(Q)$ since $M_Q$ is a match of $Q$ in $Q$. As $M$ is a match of $Q$ in $G$, we have $e(M(M_Q(u)), M(M_Q(u'))) \in E(G)$. Therefore, $M'$ is a bijective function that satisfies given $e(u, u') \in E(Q)$, $e(M'(u), M(u)) \in E(G)$. Because $M$ maps $e(u_a, u_b)$ (i.e., $e$) to the updated edge, we have $M'$ maps $e(M_Q(u_a), M_Q(u_b))$ (i.e., $e'$) to it. Thus, $M'$ belongs to $\Delta\mathcal{M}_{e'}$ and $\mathcal{M} \subseteq \Delta\mathcal{M}_{e'}$. Similarly, we have $\Delta\mathcal{M}_{e'} \subseteq \mathcal{M}$. Therefore, the proposition holds. □

**Algorithm 5:** Dual Matching

```
1  Procedure GenerateAutoSet(Q)
2      M_Q ← find matches of Q in Q;
3      X ← ∅;
4      foreach e(u, u') ∈ E(Q) do
5          if e(u, u') is not selected then
6              X ← ∅;
7              foreach M_Q ∈ M_Q do
8                  if e(M_Q(u), M_Q(u')) is not selected then
9                      X ← X ∪ {(e(M_Q(u), M_Q(u')), M_Q)};
10                     Mark e(M_Q(u), M_Q(u')) as selected;
11             X ← X ∪ {X};
12     return X;
13 Procedure DualMatch(ΔM_e, X)
14     foreach (e', M_Q) ∈ X do
15         if e' ≠ e then ΔM_e' ← {M ∘ M_Q | M ∈ ΔM_e};
16     ΔM_X ← ⋃_{e∈X} ΔM_e;
17     return ΔM_X;
```

According to the proposition, we have the following two observations on the impact of automorphisms in CSM. First, automorphisms that do not map each query vertex to itself lead to redundant computation. Second, the set of subgraphs of $G$ corresponding to matches in $\Delta\mathcal{M}_e$ is the same as that in $\Delta\mathcal{M}_{e'}$ if $e$ can be mapped to $e'$ in an automorphism.

Based on these observations, we propose the *dual matching* technique to eliminate redundant computation incurred by automorphisms of $Q$. In principle, the procedure of enumerating matches permutates data vertices to find results. In contrast, given $\Delta\mathcal{M}_e$, the dual matching technique swaps the roles of query and data graphs and enumerates matches by permutating query vertices to find $\Delta\mathcal{M}_{e'}$ based on Proposition 5.1.

## 5.2 Incremental Matching based on Auto-Sets

Based on automorphisms of $Q$, the dual matching technique groups query edges into a set $\mathcal{X}$ of *auto-sets* $X$ (Definition 5.2). GenerateAutoSet in Algorithm 5 presents the techniques. Given $e(u, u') \in E(Q)$ that does not belong to any auto-sets, Lines 6-11 iterate each match $M_Q \in \mathcal{M}_Q$ to find query edges $e(M_Q(u), M_Q(u'))$ and group these edges into a new auto-set. $X$ records the edge and the corresponding match (Line 9). As a query graph has at least one automorphism (i.e., the match mapping each query vertex to itself), the function ensures that each query edge belongs to exactly one auto-set. Auto-sets are generated in the offline processing stage, since $Q$ is fixed during the online processing.

*Definition 5.2.* Let $\mathcal{M}_Q$ denote the automorphisms of $Q$. An auto-set $X$ is a set of query edges that satisfies the following condition:

given any two edges $e(u_a, u_b), e(u'_a, u'_b) \in X$, there exists $M_Q \in \mathcal{M}_Q$ such that $u_a = M_Q(u'_a)$ and $u_b = M_Q(u'_b)$.

Given $\Delta\mathcal{M}_e$ and the auto-set $X$ that $e$ belongs to, DualMatch generates incremental matches for remaining edges $e'$ in $X$. Particularly, Line 15 loops over $M \in \Delta\mathcal{M}_e$ and generates $\Delta\mathcal{M}_{e'}$ based on Proposition 5.1 where $M_Q$ is the automorphism mapping $e$ to $e'$. Finally, we union the results and return the set $\Delta\mathcal{M}_X$ of incremental matches that map edges in $X$ to the updated edge.

**Optimization.** To further improve the performance, we optimize the procedure of generating matches for auto-sets $X$. In practice, $\Delta\mathcal{M}_e$ is stored as a table where the header is a sequence of query vertices $(..., u_i, ...)$ and each tuple is a sequence of data vertices. Figure 3 presents an example. Given $e' \in X$ and the corresponding automorphism $M_Q$, we can generate $\Delta\mathcal{M}_{e'}$ by simply adding another header $(..., M_Q(u_i), ...)$ instead of iterating each match in $\Delta\mathcal{M}_e$. Therefore, the set $\Delta\mathcal{M}_X$ is stored as a table with $|X|$ headers each of which is a sequence of query vertices based on automorphisms of $Q$.

*Example 5.3.* Given $Q$ in Figure 1a, $\mathcal{M}_Q = \{M_1 = \{(u_0, u_0), (u_1, u_1), (u_2, u_2), (u_3, u_3), (u_4, u_4)\}, M_2 = \{(u_0, u_0), (u_1, u_1), (u_2, u_2), (u_3, u_4), (u_4, u_3)\}\}$. The set $\mathcal{X}$ of auto-sets is $\{X_1 = \{(e(u_0, u_1), M_1)\}, X_2 = \{(e(u_1, u_2), M_1)\}, X_3 = \{(e(u_3, u_4), M_1)\}, X_4 = \{(e(u_2, u_3), M_1), (e(u_2, u_4), M_2)\}\}$. Given insertion of $e(v_6, v_{13})$, suppose that we obtain $\Delta\mathcal{M}_{e(u_2,u_3)}$ in Figure 3. As $e(u_2, u_3) \in X_4$, the dual matching technique generates $\Delta\mathcal{M}_{X_4}$ by adding a header based on $M_2$. The results are shown in Figure 5. Thus, we do not need to execute a search procedure for $e(u_2, u_4)$.

## 5.3 Analysis

In the following, we analyze the time and space cost of the dual matching.

**Time and Space.** After finding $\mathcal{M}_Q$, the time complexity of GenerateAutoSet is $O(|E(Q)| \times |\mathcal{M}_Q|)$. As the query graph is small, the procedure is fast. Given $X$ and $\Delta\mathcal{M}_e$, the time complexity of DualMatch is $O((|X| - 1) \times |\Delta\mathcal{M}_e|)$. The optimization avoids iterating each match in $\Delta\mathcal{M}_e$ and reduces the cost to $O((|X| - 1) \times |V(Q)|)$.

$\mathcal{X}$ generated by Algorithm 5 maintains an automorphism $M_Q$ for each query edge and has no duplicate query edge. Therefore, the space cost of storing $\mathcal{X}$ is $O(|E(Q)| \times |V(Q)|)$. If we need to store incremental matches for the update, then the space cost of the dual matching is $O(|V(Q)| \times \sum_{X \in \mathcal{X}} |\Delta\mathcal{M}_X|)$ where $\Delta\mathcal{M}_X$ is the set of incremental matches mapping edges in $X$ to the updated edge. Otherwise, we can find an incremental match and emit it immediately. Therefore, the space cost of maintaining the output is negligible.

**Discussion.** Given $Q$, $G$ and $\Delta\mathcal{G}$, for simplicity, assume that finding incremental matches mapping $e \in E(Q)$ to the updated edges in the entire stream takes $T$ time. The cost of processing the stream can be estimated as $T \times |E(Q)|$. The dual matching technique reduces the execution time to $T \times |\mathcal{X}|$. Thus, the speedup on the entire stream is $\frac{|E(Q)|}{|\mathcal{X}|}$. For example, the speedup on the entire stream given $Q$ in Figure 1 is 1.25 under the assumption because $|E(Q)| = 5$ and $|\mathcal{X}| = 4$. In contrast, the speedup for the update in Example 5.3 is 2 because the cost of finding incremental matches for $X_{1-3}$ can be neglected for this update. We evaluate the dual matching technique in Section 6.3.2.

The symmetry-breaking technique [12] eliminates duplicate results in the subgraph enumeration problem [25, 28], which is to find all subgraphs in the data graph identical to the query graph. Symmetry-breaking assigns partial orders to query vertices and requires data vertices mapped to query vertices to satisfy these orders. However, this technique cannot be directly applied to CSM because it may miss valid results. Moreover, it may issue more sub-queries than our dual matching technique on update streams.

# 6 EXPERIMENTAL RESULTS

We conduct experiments to evaluate the performance of RapidFlow.

## 6.1 Experiment Setup

In our experiments, we compare RapidFlow (named RF) with TurboFlux (named TF) [18] and SymBi (named SYM) [23], which are state-of-the-art CSM methods. RapidFlow utilizes the ordering method of RI [5] to generate the matching order for enumerating matches of $Q_R$ in the local index, because previous studies on BSM [33] show that the ordering method of RI is simple and effective. For a fair comparison, we implement all competing algorithms in C++ and optimize them with our best efforts. The source code is compiled with g++ 8.3.1. We conduct experiments on a Linux server with two Intel Xeon Gold 5218 CPUs and 512GB DRAM.

**Table 3: The detailed information of datasets. $|\Sigma_V|$ is the number of distinct vertex labels. $|\Sigma_E|$ is the number of distinct edge labels. $d_{avg}$ is the average degree. $d_{max}$ is the maximum degree. $c_{max}$ is the maximum core value.**

| Datasets | $\|V\|$ | $\|E\|$ | $\|\Sigma_V\|$ | $\|\Sigma_E\|$ | $d_{avg}$ | $d_{max}$ | $c_{max}$ |
|---|---|---|---|---|---|---|---|
| Amazon (*az*) | 0.4M | 2.4M | 6 | 1 | 12.2 | 0.2M | 10 |
| LiveJournal (*lj*) | 4.9M | 42.9M | 30 | 1 | 18.1 | 4.3M | 350 |
| Netflow (*nf*) | 3.1M | 2.9M | 1 | 7 | 2.0 | 0.2M | 8 |
| LSBench (*ls*) | 5.2M | 20.3M | 1 | 44 | 8.2 | 2.3M | 27 |

**Data Graphs.** We use Netflow and LSBench in our experiments to keep consistent with previous work [18, 23]. Netflow is a real-world dynamic graph representing passive traffic traces [6]. LSBench is a synthetic dynamic social network generated by Linked Stream Benchmark [19]. Netflow and LSBench have 7 and 44 distinct edge labels, respectively, and all vertices in both graphs have the same label. The edge label distribution is highly skewed. For example, 70.9% edges in Netflow have the same label.

Following existing research on streaming graphs [16, 21], we further generate dynamic graphs from static graphs including Amazon and LiveJournal by randomly sampling edges as the update stream. As the original datasets are unlabeled, we assign labels from a label set to vertices randomly. Table 3 presents the statistics of datasets. We can see that the datasets in our experiments cover a wide range of settings, e.g., the graph size, the graph density and the label distribution.

The insertion (resp. deletion) rate is the ratio of the number of edge insertions (resp. deletions) to the number of edges in the dataset. We set the rate to 10%, the same as previous work [16, 21]. Because competing algorithms are generally symmetric (i.e.,

processing insertion and deletion with the same algorithm), we report the results on the insertion stream for brevity.

**Query Graphs.** Following previous work [18, 23], we obtain query graphs by randomly extracting connected subgraphs from the data graph and categorize queries into tree queries and cyclic queries. Moreover, following previous studies on batch subgraph matching [4, 13], we classify cyclic query graphs into sparse queries ($d_{avg} \leqslant 3$) and dense queries ($d_{avg} > 3$) to study the performance of competing algorithms on queries with different densities. Note that if the threshold increases further, there will be few incremental matches of dense queries since real-world graphs are sparse. We increase $|V(Q)|$ from 4 to 12 at a step of 2. We do not further increase the query size because most of the data graph will involve in the computation for each update if the query graph is big and consequently the CSM problem is close to the BSM problem. For each query type and size, we generate a query set containing 100 queries. We report experiment results on the query sets with $|V(Q)| = 6$ by default.

**Metrics.** The offline preprocessing stage of all competing algorithms in our paper is efficient. Therefore, we focus on evaluating the online processing. We measure the *query time*, which is the elapsed time of processing the stream online, of each algorithm. The query time excludes the time of modifying the data graph because the overhead is the same for all algorithms. To complete our experiments in a reasonable time, we set the time limit for each query to one hour. If a query cannot be completed within the time limit, we terminate the query and mark it as *unsolved*. If the algorithm finds fewer than $10^9$ results on a unsolved query, the query is marked as a *hard unsolved* query.

Given a set $Y$ of competing algorithms and a query set $\mathcal{Q}$, $\mathcal{Q}'$ is the set of queries that all algorithms in $Y$ can complete within the time limit, and $\overline{\mathcal{Q}'}$ is the supplementary set of $\mathcal{Q}'$ (i.e., the set of queries that at least one algorithm in $Y$ cannot complete within the time limit). The *average query time* $t_y$ of an algorithm $y \in Y$ on $\mathcal{Q}$ is equal to $\frac{1}{|\mathcal{Q}'|}\sum_{Q \in \mathcal{Q}'} t_y(Q)$ where $t_y(Q)$ is the query time of $y$ on $Q$. We also count the number of unsolved queries for each algorithm and measure the number of edges processed on the query in $\overline{\mathcal{Q}'}$ to study their performance on unsolved queries.

Moreover, we evaluate the *response time*, which is the time finding one incremental match, for each *relevant update*. Specifically, a relevant update given $Q$ is an update $\Delta G$ in the stream such that at least one edge $e(u, u') \in E(Q)$ has the same label as the updated edge $e(v, v')$ in $\Delta G$ (i.e., $L(u) = L(v)$ and $L(u') = L(v')$). We omit the irrelevant updates in the stream because the cost of pruning them with the label filter is negligible. The query time (resp. index update time) per update is also computed in terms of the relevant updates in the stream. Additionally, we evaluate the candidate set size for each query vertex to study the pruning power.

## 6.2 Overall Comparison

**Average Query Time.** Figure 6 presents the average query time one each query set. As shown in the figure, TF runs faster than SYM on tree queries, but slower on cyclic queries. In contrast, RF significantly outperforms both TF and SYM on each query set and the speedups are up to two orders of magnitude, e.g., on tree and sparse queries on *nf*. As the query time on different queries varies greatly and algorithms can have performance variance on different queries, the average value can hide the performance of competing algorithms on each individual query. Therefore, we measure the speedup of RF on each query.

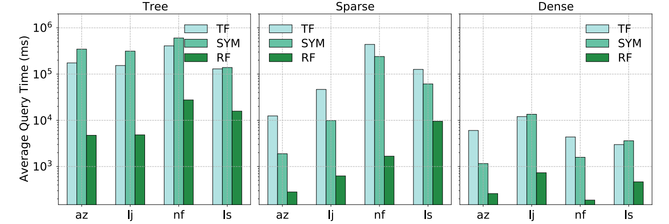


**Figure 6: Comparison of competing methods on average query time.**

**Individual Speedup.** We measure the speedup of RF over TF and SYM on each query in Figure 7. The individual speedup on a query $Q$ is computed by $\frac{t_a(Q)}{t_{RF}(Q)}$ where $t_{RF}(Q)$ is the query time of RF and $t_a(Q)$ is the query time of TF or SYM. We can see that there is no value below 1, which shows that RF outperforms SYM and TF on all cases that competing algorithms can complete within the time limit. The speedups are up to four orders of magnitude on some cases. These results demonstrate the high performance of RapidFlow.

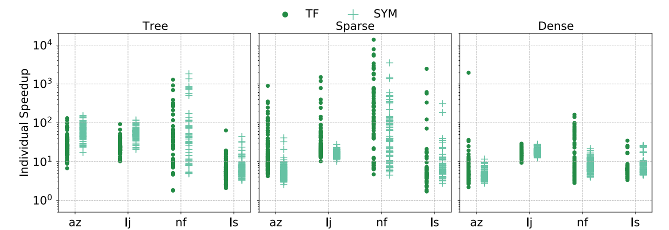


**Figure 7: Individual speedup of RF over TF and SYM in terms of the query time. Each dot denotes the speedup on a query.**

**Unsolved Queries.** We count the number of unsolved queries, including hard unsolved queries, for each algorithm in Table 4. As shown in the table, TF has more unsolved queries than SYM and RF. RF significantly reduces the number of unsolved queries, for example, RF has no unsolved queries on *lj*. Moreover, both TF and SYM have hard unsolved queries, which cause performance issues. In contrast, RF has no hard unsolved queries. In other words, the unsolved queries in RF are because of the large number of results. For example, RF has 16 unsolved tree queries on *nf* where RF reports as many as $10^{12}$ results in one hour.

To compare the performance of competing algorithms on unsolved queries, we measure the number of relevant updates processed on unsolved queries in Figure 8. There are in total 116 queries that at least one algorithm cannot complete within the time limit on the four data graphs. The number of relevant updates processed by either TF or SYM is fewer than or equal to that of RF on all these queries except one where TF deals with 177209 relevant updates while RF handles 163375. Both TF and SYM encounter performance

**Table 4: The number of unsolved queries and hard unsolved queries. The unsolved queries include hard unsolved ones. *az* is omitted because there is no unsolved query on *az*.**

| Query Structure | Method | #Unsolved Queries | | | #Hard Unsolved | | |
|---|---|---|---|---|---|---|---|
| | | *lj* | *nf* | *ls* | *lj* | *nf* | *ls* |
| Tree | TF | 3 | 38 | 11 | 0 | 3 | 0 |
| | SYM | 4 | 41 | 12 | 0 | 1 | 0 |
| | RF | 0 | 16 | 9 | 0 | 0 | 0 |
| Sparse | TF | 0 | 20 | 35 | 0 | 15 | 32 |
| | SYM | 0 | 11 | 4 | 0 | 6 | 1 |
| | RF | 0 | 0 | 3 | 0 | 0 | 0 |
| Dense | TF | 0 | 1 | 2 | 0 | 1 | 2 |
| | SYM | 0 | 0 | 0 | 0 | 0 | 0 |
| | RF | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | TF | 3 | 59 | 48 | 0 | 19 | 34 |
| | SYM | 4 | 52 | 16 | 0 | 7 | 1 |
| | RF | 0 | 16 | 12 | 0 | 0 | 0 |

issues on some queries and consequently process much fewer updates than RF. The results demonstrate the performance advantage of RF on unsolved queries.

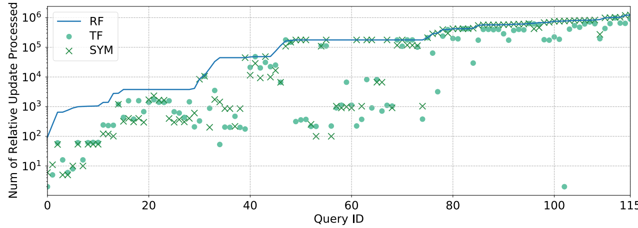



**Figure 8: Comparison of competing methods on the number of relevant updates processed on unsolved queries. The query ID is labeled in the ascending order of relevant updates processed by RF.**

## 6.3 Evaluation of Individual Techniques

In this subsection, we evaluate the effectiveness of the query reduction and dual matching, respectively.

*6.3.1 Effectiveness of Query Reduction.* We first evaluate the effectiveness of the query reduction technique by measuring the number of partial results generated in the enumeration.

**Partial Results.** Figure 9 illustrates the average number of partial results (excluding final results) generated in the enumeration for each query set. The value of each query is normalized to the min value among competing algorithms. The value of RF is close to one and therefore RF generally generates the minimum number of partial results for each query. In contrast, TF and SYM lead to much more partial results than RF. The experiment results show that existing CSM approaches generate many invalid partial results, and utilizing the well-studied BSM techniques can significantly reduce the search space size. Thus, the query reduction technique, which reduces CSM to BSM, is an efficient approach for CSM.

**Candidate Set Size.** We evaluate the pruning power of the global index by comparing the candidate set size for each query vertex with that in indexes of TF and SYM. As the index is frequently updated given the stream, we use the index for the data graph after

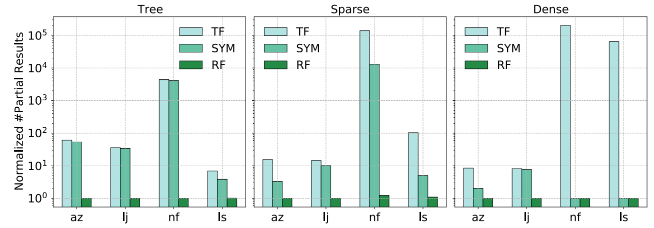



**Figure 9: Comparison of competing methods on #partial results generated in the enumeration.**

applying the entire stream in the comparison. Figure 10 presents the average candidate set size for each query vertex. We can see that RF achieves competitive performance although it uses a simple filtering rule. In some cases, RF even has fewer candidates than SYM, which adopts a complex filter rule. Looking into these cases, we find the reason is the filter rule of SYM requires a candidate $v \in C_I(u)$ must have at least one neighbor $v'$ in $C_I(u')$ for each $u' \in N(u)$, but does not consider the number of distinct neighbors with specific labels. As a neighbor $v' \in N(v)$ can appear in candidate sets of different neighbors $u' \in N(u)$, $v'$ can be valid for SYM, but is pruned by the neighbor label frequency filter (NLF) in RF. We do not report the number of candidates in the local index because it is generated for each update. We evaluate the cost of index update in the following.

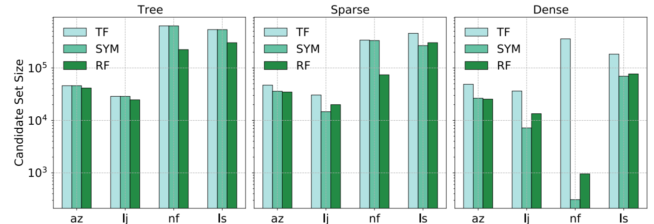


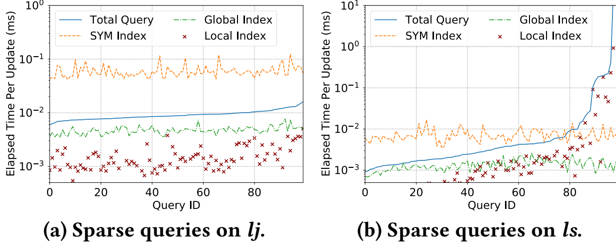

**Figure 10: Comparison of the average candidate set size for each query vertex.**

**Query Time Breakdown.** Given an update, RF first updates the global index and then builds the local index to obtain the affected region. Figure 11 shows the query time breakdown for each update on individual queries. As the local index time fluctuates, we represent it as dots instead of a line. The figure also illustrates the indexing time of SYM, the latest CSM algorithm, for reference. We present the results on sparse queries on *lj* and *ls* as representatives. Based on the experiment results, we have the following observations.

*Global Index.* The global index update time of RF is much shorter than that of SYM because the filter rule of RF is simpler. The global index update time is more steady than the local index generation time on a variety of queries since each update only affects candidates adjacent to the updated edge. Overall, the global index update is very efficient, less than 0.01 ms for each update, as shown in the figure.

*Local Index.* The local index generation time is very short in most cases because the global index significantly reduces the number of data vertices involved in the computation, and the region affected by the update is small. However, some queries in Figure 11b have much longer local index generation time than other queries because the update has a big affected region. For example, we find a local candidate set had more than 500,000 candidates for a query vertex.

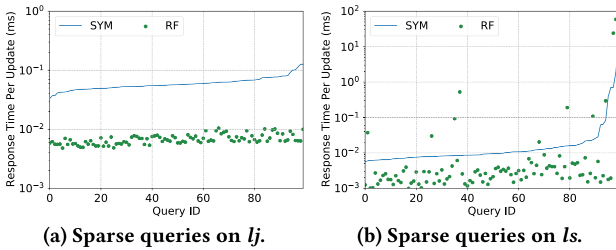
Nevertheless, RF built the local index efficiently (less than 50 ms) on those queries, and the enumeration time can dominate the cost because there are many incremental matches (e.g., some queries have more than $10^7$ results per update) in the big affected region. As shown in Figure 7, RF significantly outperforms SYM in terms of the query time. Therefore, the benefit of the query reduction offsets the overhead.




**(a) Sparse queries on *lj*.** **(b) Sparse queries on *ls*.**

**Figure 11: Query time breakdown of each individual query. The query ID is labeled in the ascending order of query time. Query time (*Total Query*) for an update includes the global index update time (*Global Index*), the local index construction time (*Local Index*) and the enumeration time. *SYM Index* is the index update time of SYM.**

**Response Time.** Figure 12 presents the response time for each update on individual queries. RF runs much faster than SYM on *lj* and the response time is less than 0.01 ms. RF cannot dominate SYM on each case in *ls*. Specifically, RF is slower than SYM on 10 of 100 queries in terms of the response time. Those 10 queries generally have a big affected region that contains many incremental results and the local index generation dominates the response time. In such cases, it is easy for SYM to find one result from the large result set, whereas the overhead of extracting the affected region in RF offsets the benefit of finding one result. Despite of the overhead of the local index, the response time of RF is generally less than 1 ms on most queries on *ls*.




**(a) Sparse queries on *lj*.** **(b) Sparse queries on *ls*.**

**Figure 12: Response time of each individual query. The query ID is labeled in the ascending order of response time of SYM.**

**Summary.** We have the following findings through the experiments in this subsection. 1) The query reduction technique can dramatically accelerate the query because the effective matching order reduces the search space size. 2) Compared with existing CSM approaches, our global index with a simple filter rule achieves a considerable pruning power at a much lower overhead. 3) The two-level indexing mechanism is effective for extracting the affected region. 4) RF can perform worse than existing CSM approaches if the workload has a large number of incremental matches for the update, whereas we only want to get a small portion (e.g., one result) of them.

*6.3.2 Effectiveness of Dual Matching.* We evaluate the effectiveness of the dual matching technique by comparing the performance of RF with and without the optimization. Table 5 lists the speedup achieved by enabling the technique. The value excludes the overhead of the global index update because it is fixed for RF with/without the optimization. The optimization generally accelerates the query, and the speedup is up to 10X on some queries. This result shows the effectiveness of the technique. Because the auto-sets are obtained in the offline stage, the technique does not incur any overhead for the online processing. Thus, we recommend to always enable dual matching for CSM.

**Table 5: Effectiveness of the dual matching technique. *Avg* is the average speedup achieved by enabling dual matching. *Max* is the maximum speedup on the query set.**

| Dataset | Tree | | Sparse | | Dense | |
|---|---|---|---|---|---|---|
| | Avg | Max | Avg | Max | Avg | Max |
| *az* | 1.27X | 1.69X | 1.28X | 1.68X | 1.82X | 5.00X |
| *lj* | 1.23X | 1.40X | 1.39X | 1.62X | 1.01X | 1.03X |
| *nf* | 1.74X | 4.14X | 2.15X | 10.16X | 1.40X | 4.14X |
| *ls* | 1.12X | 1.52X | 1.46X | 4.59X | 1.62X | 4.48X |

## 7 CONCLUSION

In this paper, we study the problem of continuous subgraph matching (CSM) and propose an efficient CSM approach, RapidFlow. We design the query reduction technique that reduces the problem of finding incremental matches for an update to that of enumerating all matches of a subgraph of $Q$ in the region of $G$ affected by the update, i.e., a batch subgraph matching (BSM) problem. In order to reduce the redundant computation caused by automorphisms of $Q$, we propose the dual matching technique, which reverses the roles of query graphs and data graphs and enumerates incremental matches by permutating query vertices. Extensive experiment results show that RapidFlow significantly outperforms state-of-the-art CSM approaches including TurboFlux and SymBi. The results also suggest that with the query reduction, existing BSM techniques are efficient in the CSM setting. An interesting research direction is to investigate how to implement and integrate the query-dependent index into existing systems. A promising approach is to regard the query-dependent index as a collection of materialized views. We can maintain these views incrementally given the update and find incremental matches based on the views instead of the base data.

## ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative and by Research Grants Council of Hong Kong (Grant 16209821). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

## REFERENCES


[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 431–446.

[2] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

[3] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1447–1462.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1199–1214.

[5] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, S-7 (2013), S13.

[6] CAIDA. 2013. *The CAIDA UCSD Anonymized Internet Traces 2013*. https://www.caida.org/data/passive/passive_2013_dataset.xml

[7] Lei Chen and Changliang Wang. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Trans. Knowl. Data Eng.* 22, 8 (2010), 1093–1109.

[8] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proceedings of the 18th EDBT International Conference on Extending Database Technology*. 157–168.

[9] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (2004), 1367–1372.

[10] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 925–936.

[11] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *Proc. VLDB Endow.* 3, 1 (2010), 264–275.

[12] Joshua A Grochow and Manolis Kellis. 2007. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Annual International Conference on Research in Computational Molecular Biology*. Springer, 92–106.

[13] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1429–1446.

[14] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 337–348.

[15] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science*. 453–462.

[16] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1695–1698.

[17] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 925–937.

[18] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 411–426.

[19] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. 2012. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*. 300–312.

[20] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. 2016. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1035–1044.

[21] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. *ACM Trans. Database Syst.* 46, 2 (2021), 6:1–6:45.

[22] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.

[23] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proc. VLDB Endow.* 14, 8 (2021), 1298–1310.

[24] Jayanta Mondal and Amol Deshpande. 2016. Casqd: continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 226–237.

[25] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.

[26] L Qin, Y Peng, Y Zhang, X Lin, W Zhang, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained st simple path enumeration. In *International Conference on Very Large Data Bases*. VLDB Endowment.

[27] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.

[28] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1344–1356.

[29] Carlos Rivero and Hasan Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge & Information Systems* 51, 1 (2017).

[30] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375.

[31] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. *Proc. VLDB Endow.* 8, 4 (2014), 413–424.

[32] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained st Path Enumeration. In *Proceedings of the 2021 International Conference on Management of Data*. 1758–1770.

[33] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[34] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14, 2 (2020), 176–188.

[35] Ha Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Proceedings of the 20th DASFAA International Conference on Database Systems for Advanced Applications*, Vol. 9049. 299–315.

[36] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[37] Shihan Wang and Takao Terano. 2015. Detecting rumor patterns in streaming social media. In *2015 IEEE International Conference on Big Data*. 2709–2715.

[38] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th EDBT International Conference on Extending Database Technology*. 192–203.

[39] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1 (2010), 340–351.