An Extensive Experimental Study of Indexes in Continuous Subgraph Matching:[Experiments & Analysis]

XIANGYANG GOU, University of New South Wales, Australia
LEI ZOU, Peking University, China
JEFFREY XU YU, Hong Kong University of Science and Technology (Guangzhou), China
WENJIE ZHANG, University of New South Wales, Australia

Continuous subgraph matching (CSM), which finds incremental matches of a query graph for each update in a dynamic graph, has gained significant research attention. Most CSM algorithms follow a common indexing-enumeration paradigm: they first use indexes to identify candidate vertices and edges for the query graph, and then enumerate matches based on these candidates. Although there have been several comprehensive experimental analyses of CSM algorithms, they tend to evaluate CSM algorithms holistically, obscuring the distinct contributions of the indexing and enumeration methods to overall performance. In this paper, we decouple the indexing method and enumeration method of existing CSM algorithms, and focus on the comparison of indexing methods. Our experimental results offer guidance on index selection across different scenarios, serving as a reference for future research and industrial applications. They further reveal the relative importance of different index components, informing strategies to discard less essential parts when memory is limited. Additionally, we show that the commonly used candidate count metric may underestimate the filtering effectiveness of certain indexes, suggesting that future research should adopt more reliable evaluation metrics.

CCS Concepts: • **Information systems** → **Database query processing**; **Graph-based database models**.

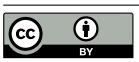
Additional Key Words and Phrases: Dynamic Graph, Subgraph Matching, Index, Experimental Analysis

ACM Reference Format:
Xiangyang Gou, Lei Zou, Jeffrey Xu Yu, and Wenjie Zhang. 2026. An Extensive Experimental Study of Indexes in Continuous Subgraph Matching:[Experiments & Analysis]. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 9 (February 2026), 26 pages. https://doi.org/10.1145/3786623

1 Introduction

Subgraph matching is a basic operator in graph analysis. It requires finding subgraphs in a data graph that are isomorphic to a given query graph. These subgraphs are called matches of the query graph. As subgraph matching is a well-known NP-hard problem, extensive studies have been conducted to improve its efficiency, including choosing optimal matching orders, designing auxiliary indexes, and enabling computation share [7, 8, 12, 15, 19, 34, 37, 44]. On the other hand, most graphs in real-world applications are dynamic. In order to keep the timeliness of the query result, a new variant of subgraph matching has emerged, named Continuous Subgraph Matching (CSM). CSM requires finding matches including the updated edge whenever there is an edge insertion or deletion in the dynamic graph. CSM is widely used in various real-life domains, such

Authors' Contact Information: Xiangyang Gou, University of New South Wales, Sydney, Australia, xiangyang.gou@unsw.au; Lei Zou, Peking University, Beijing, China, zoulei@pku.edu.cn; Jeffrey Xu Yu, Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, jeffreyxuyu@hkust-gz.edu.cn; Wenjie Zhang, University of New South Wales, Sydney, Australia, wenjie.zhang@unsw.edu.au.



ACM 2836-6573/2026/2-ART9
https://doi.org/10.1145/3786623

as recommendation systems [14], quantum circuit design [23], and cybersecurity [11]. It has also gained significant research attention [11, 13, 20].

Most existing CSM algorithms follow an indexing-enumeration paradigm. In the offline phase, they construct an auxiliary index based on the initial data graph and the query graph. During the online phase, they update the index with each data graph modification and use it to identify candidate vertices and edges for potential matches. Then they enumerate incremental matches with these candidates. The indexes filter out unrelated vertices and edges and shrink the search space of CSM. Significant efforts have been made in designing more efficient indexes [22, 32].

Existing experimental studies typically evaluate CSM algorithms holistically, making it difficult to isolate the contributions of indexing and enumeration to overall performance. For example, the most recent study [25] demonstrates the superior performance of the Symbi algorithm [32], which differs from others in both its indexing and enumeration strategies. However, such comparisons do not reveal the impact of individual components. For instance, it remains unclear whether Symbi's indexing method outperforms alternatives like the CaliG index [43] when combined with the same enumeration techniques. Although some ablation studies have been conducted [25, 36], they remain limited in scope. In terms of indexing, only Symbi's DCS index [32] and TurboFlux's DCG index [22] are included, while other indexing methods [28, 35, 43] are not compared.

Owing to the shared framework, CSM algorithms allow for interchangeable combinations of indexing, matching order, and enumeration techniques. Therefore, isolating and comparing each part provides valuable guidance for selecting optimal combinations under different application requirements. Moreover, recent works show a divergence in algorithm design strategies. Some researchers [43] enhance the filtering capability of indexes at the expense of increased update and storage overhead, while others [28, 35] prioritize lightweight indexes. A comparative analysis can thus inform future research directions and guide the development of more balanced algorithm designs.

In this paper, we focus on a thorough experimental analysis of the indexing methods. The indexes account for the majority of memory usage in CSM algorithms. Effective index selection not only improves time efficiency but also enables a better balance between computational speed and memory cost. With this experimental study, we hope to answer the following questions:

(1) What is the best choice of indexes in different situations, considering both the memory cost and time efficiency?
(2) How important are different components in indexes?
(3) Are the metrics used to evaluate indexes in previous works effective enough?

These questions are crucial for guiding index selection in future applications and resolving recent divergences in index design strategies. However, they cannot be addressed by previous experimental studies [25, 36], which evaluate CSM algorithms as a whole and include only a limited range of indexing methods in their ablation studies. Moreover, the two compared indexes follow a similar paradigm, while other diverse indexes [28, 35, 43] remain uncompared.

In our experimental study, we decouple the indexing and enumeration methods of all CSM algorithms that adopt the state-of-the-art vertex-at-a-time enumeration framework[1]. We then compare the performance of 6 indexing methods under the same enumeration strategy. Furthermore, we decompose the indexes to analyze the impact of individual components, identifying less critical parts that can be trimmed for efficiency. We also evaluate the effectiveness of commonly used performance metrics.

[1]Only a few early-stage methods do not follow this framework and have been shown to be significantly less efficient [36].

Table 1. Notation Table

Notation	Meaning
V/ V_Q	Vertex set of data graph G / query graph Q
E/E_Q	Edge set of data graph G/ query graph Q
$N(v)$	Neighbor set of vertex v
$L(v)$/$L(v,v')$	Label of vertex v / edge (v,v')
$N_{l_i,l_j}(v)$	Neighbor set of vertex v, where the neighbors have label l_i, and the edges between these neighbors and v have label l_j.
$Idx.C(u)$	Candidate set of a query graph vertex u generated by index Idx
$\langle u,v\rangle$	Mapping between query vertex u and data vertex v in matches or indexes.
$\Delta M_{u_x,u_y}$	Incremental matches with the updated edge (v_x,v_y) mapped to query edge (u_x,u_y)

In summary, we make the following contributions in our experimental study, and get corresponding conclusions:

(1) We conduct a thorough and fair comparison of all existing indexing methods under a unified enumeration algorithm. Our results show that a lightweight index materializing NLF (Neighborhood Label Frequency) comparisons performs best in most scenarios. However, for query graphs with large diameters (≥ 4) and low label diversity, the DCS index can be attempted, as it may better capture multi-hop neighborhood information. When memory is limited, the DCG index serves as a practical alternative to DCS. **Moreover, we do not recommend further exploration of indexes with higher complexity and maintenance cost, as our results show that their improvements are rarely sufficient to justify their additional cost.** These findings directly address our first research question.
(2) We further decompose the indexes and dig into the effect of different components. **We find that the NLF filter is a powerful tool and suggest researchers to integrate it when designing indexes.** Besides, we find that the edge indexes storing candidate edges are less important than the vertex indexes storing candidate vertices, but they may occupy a large portion of memory usage. **We suggest removing edge indexes when there is a memory shortage.** These findings answer our second question.
(3) We experimentally examine the effectiveness of index evaluation metrics. We find that candidate count is not a good metric as it may underestimate the filtering ability of some indexes. **We recommend researchers to compare the partial match count or end-to-end processing time rather than the candidate count in future index comparison**. These findings address our third question.

2 Preliminaries

In this Section, we formally define our problem and list frequently-used notations in Table 1.

Definition 2.1. ***Graph:*** *An undirected, edge and vertex labeled graph is defined as $G = (V, E, \Sigma, L)$, where V is a vertex set, $E \subset V \times V$ is an edge set, Σ is a set of labels, and $L : E \cup V \rightarrow \Sigma$ is a labeling function.*

Specifically, we use v to denote an arbitrary vertex in V, and use $e = (v, v')$ to denote an edge between vertex v and v' in E. Moreover, we use $L(v)$ to denote the label of vertex v and use

$L(v, v')$ to denote the label of edge (v, v'). Besides, we use $N(v)$ to denote the neighbor set of v, namely $\{v'|(v, v') \in E\}$. We use $N_{l_i,l_j}(v)$ to denote the label-constrained neighbor set of v, namely $\{v'|(v, v') \in E \text{ and } L(v') = l_i \text{ and } L(v, v') = l_j\}$.

Note that we focus on undirected graphs. Most previous work on CSM uses this semantic [28, 32, 35]. Transferring these algorithms to directed graphs needs considerable modification, and is beyond the scope of this paper.

Definition 2.2. ***Subgraph matching:*** *Given a* ***data graph*** *G and a* ***query graph*** *Q, a subgraph g of G is isomorphic to Q if and only if there is a bijective mapping F from V_Q to V_g, so that*

(1) For each vertex $u \in V_Q$, $L(F(u)) = L_Q(u)$.

(2) For each edge $(u, u') \in E_Q$, $(F(u), F(u')) \in E_g$. Moreover, $L(F(u), F(u')) = L_Q(u, u')$.

In this case, F is an isomorphism of Q. We call g a ***match*** *of Q, and call v a match of u if $F(u) = v$.*

L and L_Q denotes labeling function of G and Q, respectively. As they are defined on different domains, there is no conflict among these functions. We unify them with L in the following sections.

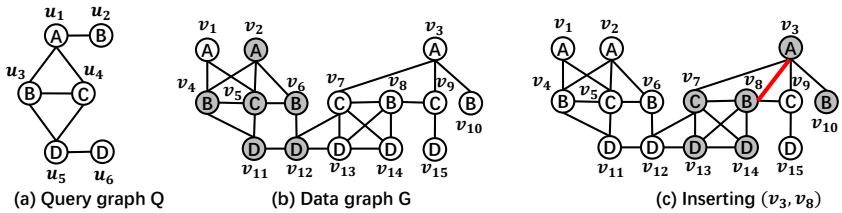


Fig. 1. Example of continuous subgraph matching

Example 1. *Consider the query graph Q in Figure 1(a) and the data graph G in Figure 1(b). The subgraph marked with shadow in G is a match of Q, with u_1 mapped to v_2, u_2 mapped to v_6, u_3 mapped to v_4, u_4 mapped to v_5, u_5 mapped to v_{11} and u_6 mapped to v_{12}.*

Note we use the semantics of isomorphism, in accordance with the original papers of existing CSM algorithms [28, 35]. But these algorithms can also be extended to homomorphism. Subgraph homomorphism is similar to isomorphism, except that it does not require the mapping to be bijective.

Definition 2.3. ***Update stream and dynamic graph:*** *A dynamic graph G is composed of an initial graph G_0 and an update stream ΔG. ΔG is a sequence of operations $\{\Delta_0, \Delta_1, \Delta_2 .....\}$, where each operation $\Delta_i = (+/-, e)$ is to insert or delete an edge e in G. Initially, we have $G = G_0$, and then G is continuously updated by the operations in the update stream ΔG.*

Definition 2.4. ***Continuous Subgraph Matching (CSM):*** *given a dynamic graph G and a query graph Q, a continuous subgraph matching (CSM) query is to find the added/deleted matches of Q in G whenever G is updated by an operation $\Delta_i = (+/-, e)$.*

Example 2. *Figure 1(c) shows the scenario of inserting an edge (v_3, v_8) to G. The subgraph marked with shadow is a new match which is added due to the insertion of (v_3, v_8), with u_1 mapped to v_3, u_2 mapped to v_{10}, u_3 mapped to v_8, u_4 mapped to v_7, u_5 mapped to v_{13} and u_6 mapped to v_{14}.*

In this paper, we focus on CSM algorithms with vertex-at-a-time enumeration. These algorithms find matches of a query graph Q by gradually mapping each vertex in Q to data vertices in G following a matching order ϕ. All the state-of-the-art CSM algorithms belong to this kind.

Definition 2.5. ***Matching order and left/right neighbors:*** *A matching order ϕ is a reordering of all the query vertices in Q. We use $\phi(u)$ to denote the position of u in matching order ϕ. We call u' a left neighbor of u if $(u, u') \in E$ and $\phi(u') < \phi(u)$, namely u' appears before u in ϕ. In contrast, we call u' a right neighbor of u if $(u, u') \in E$ and $\phi(u') > \phi(u)$.*

Definition 2.6. ***Index/materialized view of CSM:*** *An index (also known as a materialized view) of a CSM algorithm is a data structure that records candidate set $C(u)$ for each query vertex u in Q. $C(u)$ is a superset of the data vertices that are matches of u.*

Note that the same data vertex may appear multiple times in the candidate sets of different query vertices. To distinguish them, we use $\langle u, v \rangle$ to denote v as a candidate of u. Some indexes store edges between candidates to speed up index updates and match enumeration. We further name the recorded candidate sets as the **vertex index**, and name the recorded edges as the **edge index**. The vertex index is essential, while the edge index is optional.

3 Related Work

Subgraph Matching: Subgraph matching is a fundamental task in graph analysis and has been extensively studied for decades. Ullmann first introduced a backtracking-based exploration algorithm [38], which recursively maps query vertices to data vertices according to a predefined matching order. Subsequent studies [8, 12, 34, 37] proposed various optimizations to reduce matching cost. More recent works [7, 8, 15, 21] adopt an indexing–enumeration paradigm, where an auxiliary index is built to identify candidate vertices and edges before enumerating matches. In parallel, join-based approaches [4, 6, 30] model graphs as relational databases and answer subgraph queries through multi-way join operations.

Continuous Subgraph Matching (CSM): The earliest work to address the CSM problem is IncIsoMatch [13]. It extracts a subgraph G' from G induced by vertices within d hops of the updated edge. d is the diameter of the query graph Q. All matches in G' are then computed using static subgraph isomorphism methods, from which incremental matches containing the updated edge are derived. However, this algorithm has been proved to be very inefficient [36] as G' may be very large and a large number of stale matches will be produced. To overcome this limitation, subsequent algorithms adopt an incremental paradigm. SJ-tree [11] performs edge-at-a-time enumeration, joining matches of individual edges while maintaining an index of partial results. However,its index has a memory cost of $O(|E|^{|E_Q|})$, which is much higher than the $O(|E| \times |E_Q|)$ cost of later works. GraphFlow [20] uses vertex-at-a-time enumeration, which reduces the number of partial matches generated in the matching process and improves the time efficiency. Subsequent algorithms include TurboFlux [22], Symbi [32], and RapidFlow [35], which follow a similar framework but use different indexing methods and enumeration techniques. The most recent works of CSM have shown divergence in the use of indexes. CaliG [43] adds injective-match checks into indexes and designs a stronger yet more complex index. On the other hand, NewSP [28] removes query-dependent indexes and only uses Neighbor-Label-Frequency (NLF) rules to compute candidates on-the-fly. It focuses on enumeration techniques that enable computation sharing and search space shrinking. There are also several works about multi-query optimization or hardware acceleration in CSM [29, 33, 40, 41], and algorithms for time-constrained CSM [27, 31, 42]. We focus on single query optimization and CPU-based implementation in this paper.

Experimental study of CSM: Sun et.al. [36] first carried out an in-depth experimental study of CSM algorithms. Limited by its publication time, it does not include the latest algorithms proposed after 2021, including RapidFlow [35], CaliG [43] and NewSP [28]. A following experimental study [25] points out several inefficient implementation in the code of Sun et.al., and proposes suggestions on the physical implementation of indexes and enumeration methods. It also proposes several

improvements for Symbi and TurboFlux, including combining them with the NLF filter. We adopt their suggestions and improvements in our experimental study. These two works evaluate CSM algorithms as a whole in most experiments.

4 A General Model

In this section, we first introduce a general CSM algorithm model with vertex-at-a-time enumeration. It lays the foundation of the state-of-the-art CSM algorithms.

In the **offline phase**, we build an index according to the initial graph G_0 and query graph Q, which stores candidate vertex set for each query vertex $u \in V_Q$. Besides, for each query edge $e \in E_Q$, we generate a matching order ϕ that begins with the endpoints of e, and store it in the system.

In the **online phase**, updates in the stream are processed sequentially. For an edge insertion, we update the dynamic graph and index, then search for new matches involving the added edge. For an edge deletion, we first search for affected matches before applying the update, to avoid losing the edge's topology information.

Algorithm 1 describes the process to compute incremental matches ΔM given an updated edge (v_x, v_y). It can be divided into the following steps:

Mapping of the updated edge: First, we map the updated edge (v_x, v_y) to query edges (u_x, u_y) with the same label. For each (u_x, u_y), we check via the index whether v_x and v_y are valid candidates (line 4). If so, we map u_x and u_y to v_x and v_y, respectively. The matched vertices are recorded with a partial match $\mathcal{M}$.

Enumerate matches with pre-defined matching order: After mapping the updated edge to a query edge (u_x, u_y), we retrieve the matching order ϕ starting from u_x and u_y, and extend $\mathcal{M}$ along ϕ to enumerate $\Delta M_{u_x,u_y}$ (lines 6–7). This process follows a depth-first search (DFS). For each query vertex u visited in ϕ, we compute its candidate vertices v, which must satisfy three conditions:

(1) For each left neighbor u' of u, the candidate v must be connected to $\mathcal{M}[u']$ (match of u' in $\mathcal{M}$) with the same edge label, i.e., $L(v, \mathcal{M}[u']) = L(u, u')$. To ensure this, we join the label-constrained neighbor sets of all $\mathcal{M}[u']$ (line 11).
(2) v should be a candidate of u according to the index. We need to join the result set of the above step with $Idx.C(u)$ to meet this condition (line 11).
(3) v is not in $\mathcal{M}$ yet (line 13). This condition is intended to meet the bijection mapping constraint of subgraph isomorphism. For homomorphism, we can remove this check.

For each candidate v that meets the above conditions, we add mapping $\langle u, v \rangle$ to $\mathcal{M}$. Then we check if all vertices in Q have been matched, and add $\mathcal{M}$ to the incremental match set if so (line 15-16). Otherwise, we generate a new DFS search branch and continue to match the remaining query vertices (line 18).

5 Algorithms Under Study

5.1 GraphFlow [20]

GraphFlow is a system that supports both static subgraph matching and continuous subgraph matching. In this paper we only introduce its CSM part:

Index: GraphFlow dose not build additional index.

Matching order: For each query edge (u, u'), GraphFlow generates matching order ϕ with the following steps: (1) Add u and u' to ϕ. (2) Iteratively find the vertex that is not in ϕ and has the maximum number of neighbors in ϕ, and add it to the end of ϕ. If there is a tie, select the vertex with the highest degree.

Enumeration techniques: GraphFlow has no additional enumeration techniques.

Algorithm 1: Enumeration of CSM

Input: Dynamic graph G, query graph Q, index Idx, updated edge (v_x, v_y).
Output: Incremental match ΔM.

$\Delta M \leftarrow \emptyset$
$E_{match} \leftarrow \{(u_x, u_y) | L(u_x) = L(v_x), L(u_y) = L(v_y), L(u_x, u_y) = L(v_x, v_y)\}$
for *each* $(u_x, u_y) \in E_{match}$ **do**
  if $v_x \in Idx.C(u_x)$ *and* $v_y \in Idx.C(u_y)$ **then**
    $\mathcal{M} \leftarrow \{\langle u_x, v_x\rangle, \langle u_y, v_y\rangle\}$
    $\phi \leftarrow$ matching order starts with u_x, u_y
    $Enumerate(\mathcal{M}, \Delta M, \phi, G, Q, Idx, 2)$
Function $Enumerate(\mathcal{M}, \Delta M, \phi, G, Q, Idx, x)$**:**
  $u \leftarrow \phi[x]$
  $LN \leftarrow \{u' | u' \; is \; left \; neighbor \; of \; u\}$
  $S \leftarrow \bigcap_{u' \in LN} N_{L(u),L(u,u')}(\mathcal{M}[u']) \cap Idx.C(u)$
  for $v \in S$ **do**
    if v *has not been used in* $\mathcal{M}$ **then**
      $\mathcal{M} \leftarrow \mathcal{M} \cup \langle u, v\rangle$
      if $x + 1 == \phi.size$ **then**
        Add $\mathcal{M}$ to ΔM
      else
        $Enumerate(\mathcal{M}, \Delta M, \phi, G, Q, Idx, x + 1)$
      Delete $\langle u, v\rangle$ form $\mathcal{M}$

5.2 TurboFlux [22]

Index: TurboFlux constructs an index called DCG. It first extracts a spanning tree T from the query graph Q, then filters each query vertex u's candidates by ensuring that every candidate has neighbors corresponding to u's parent and children in T.

Specifically, DCG defines the cardinality of a query edge e as the number of edges in the initial graph G_0 that share the same label as e. Let e_{min} be the query edge with the smallest cardinality. DCG selects the endpoint of e_{min} whose label appears less frequently as the root of T, denoted as u_r. In case of a tie, DCG prioritizes the endpoint with the higher degree. Then DCG repeatedly finds edge e with the smallest cardinality among the edges that are not in T and can keep T cycle-free and connected, and adds e into T. This process stops when all the query vertices are included in T.

The **vertex index** of DCG can be divided into two parts:

Implicit candidates C_{imp}: $v \in C_{imp}(u)$ if: (1) v has the same label as u. (2) Assume u' is the parent of u in T. There is at least one $v' \in C_{imp}(u')$ where $L(v', v) = L(u', u)$.

Explicit candidates C_{exp}: $v \in C_{exp}(u)$ if: (1) $v \in C_{imp}(u)$. (2) For each u' that is a child of u in T, there is at least on $v' \in C_{exp}(u')$ where $L(v, v') = L(u, u')$.

Only explicit candidates will be regarded as **valid candidates**, and included in $Idx.C(u)$.

The **edge index** of DCG can also be divided into implicit candidate edges and explicit candidate edges. Suppose $(u, u') \in T$ and u is the parent. There is an candidate edge from $\langle u, v\rangle$ to $\langle u', v'\rangle$ if $L(v, v') = L(u, u')$ and v, v' are candidates of u, u', respectively. It is an implicit candidate edge if $v' \in C_{imp}(u')$, and it is an explicit candidate edge if $v' \in C_{exp}(u')$. The edge index helps to directly fetch connected candidates in the index update and match enumeration.

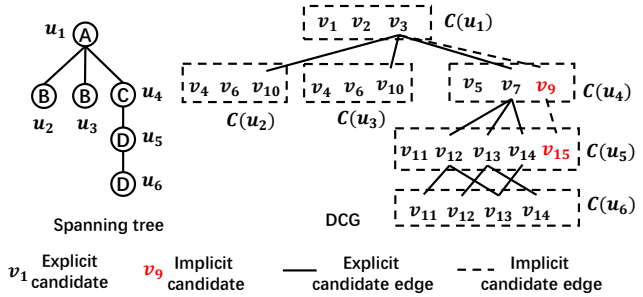


Fig. 2. Example of DCG structure

Example 5.1. Figure 2 shows an example of the spanning tree T extracted from the query graph in Figure 1a and the corresponding DCG built on the data graph in Figure 1b. To keep the figure clear, we only present the candidate edges in the spanning tree with root $\langle u_1, v_3 \rangle$. Note that v_{15} is an implicit candidate of u_5, as it has no neighbor in $C_{exp}(u_6)$. v_9 is also an implicit candidate, since it is only connected to implicit candidate v_{15} and no explicit candidate of u_5. The candidate edges connected to them are also implicit.

Matching Order: TurboFlux generates a global order ϕ_0 that starts from the tree root u_r and minimizes the estimated count of partial matches when matching T in G_0. For each query edge (u_x, u_y), a matching order ϕ that starts from it is computed by first traversing back to u_r and then enumerating other vertices according to ϕ_0.

Enumeration techniques: Different from Algorithm 1, TurboFlux uses edge-at-a-time enumeration when traversing back to u_r, and uses vertex-at-a-time starting from u_r.

5.3 Symbi [32]

Index: The index of Symbi is called DCS. It is very similar to the index of TurboFlux, except that it is based on a directed acyclic graph (DAG) rather than a spanning tree. DCS first transforms the query graph Q into a DAG D by performing a BFS from a root u_r. Each edge is directed from the early-visited endpoint to the late-visited one. We define the height of D as the maximum distance between u_r and other vertices, and u_r is selected to maximize the height of D.

The **vertex index** of DCS can also be divided into implicit candidates and explicit candidates:

Implicit candidates: $v \in C_{imp}(u)$ if: (1) v has the same label as u. (2) For each precursor u' of u in D. There is at least one $v' \in C_{imp}(u')$ where $L(v', v) = L(u', u)$.

Explicit candidates: $v \in C_{exp}(u)$ if: (1) $v \in C_{imp}(u)$. (2) For each successor u' of u in D, There is at least one $v' \in C_{exp}(u')$ where $L(v, v') = L(u, u')$.

Similar to DCG, only explicit candidates are valid candidates.

DCS builds a coarser **edge index** compared to DCG: an edge (v, v') is selected as a candidate of (u, u') as long as it has the same label as (u, u'), namely $L(v) = L(u)$, $L(v') = L(u')$ and $L(v, v') = L(u, u')$. This index can be used to find labeled-constrained neighbors $N_{L(u'),L(u,u')}(v)$. DCS also maintains some additional indexes to support a fast check of the neighborhood match status. $N(u, v, u')$ monitors the number of v's neighbors v' in the implicit/explicit candidate set of a precursor/successor u' of u, such that $L(v', v) = L(u', u)$. $P(u, v)$ and $C(u, v)$ monitors the number of precursor/successor u' where $N(u, v, u') > 0$. With these indexes, we can quickly check if v is a candidate of u.

Example 5.2. Figure 3 shows an example of the DAG generated from the query graph in Figure 1a and the corresponding DCS built on the data graph in Figure 1b (we omit the structures that

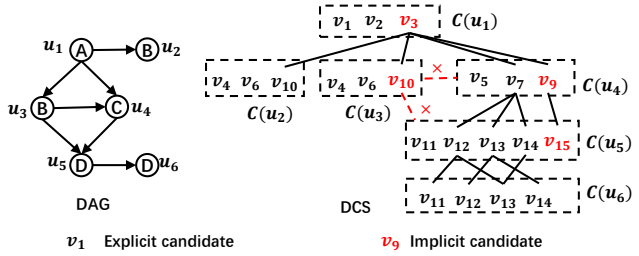


Fig. 3. Example of DCS structure

aids DCS update). To keep the figure clear, we only present the candidate edges connected with $\langle u_1, v_3 \rangle$. Compare to DCG in Figure 2, we further filter out $\langle u_3, v_{10} \rangle$ and $\langle u_1, v_3 \rangle$. As v_{10} does not have neighbors in the candidate set of u_4 and u_5, it is an implicit candidate of u_3. Moreover, as v_3 does not have neighbors among the explicit candidates of u_3, it is also an implicit candidate.

Matching order: For each query edge (u_x, u_y), the mathing order ϕ is dynamically generated online. In line 9 of Algorithm 1, the next vertex to enumerate is selected to be the unmatched query vertex with at least one matched neighbor and the smallest estimated candidate set S.

Enumeration techniques: Symbi employs two enumeration optimizations: isolated vertex and Neighborhood Equivalence Class (NEC). The isolated-vertex strategy postpones the enumeration of query vertices without right neighbors, as their matches do not affect later steps—thus reducing branching and shrinking the search tree. The NEC technique groups query vertices with identical neighborhoods, allowing candidate set reuse. It also enables the computation sharing of $\Delta M_{u,u'}$ and $\Delta M_{u,u''}$ where u' and u'' are in the same NEC.

5.4 RapidFlow [35]

Index: The index of RapidFlow is based on the Neighbor Label Frequery (NLF) filter, which is first proposed in static subgraph matching algorithms [16]. We denote this index as NLFI for short. v is a candidate of u in NLFI only if for each edge label and vertex label pair (l_i, l_j), $|N_{l_i,l_j}(v)|$ is no smaller than $|N_{l_i,l_j}(u)|$. Otherwise, the neighbors of v cannot match all neighbors of u. For edge index, RapidFlow stores edges between candidates $\langle u, v \rangle$ and $\langle u', v' \rangle$ if $L(v, v') = L(u, u')$. The edge index is used to support fast retrieval of connected candidates in match enumeration.

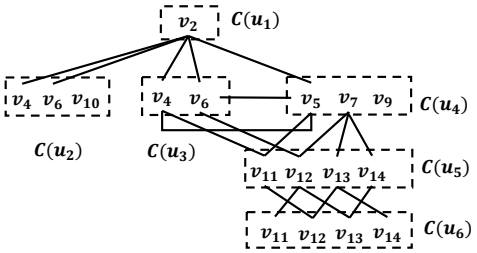


Fig. 4. Example of NLFI structure

Example 5.3. Figure 4 shows an example of the NLFI built on the query graph in Figure 1a and the data graph in Figure 1b. The filtering ability of NLFI is compatible with DCS in Figure 3. NLFI can filter out $\langle u_1, v_1 \rangle$, which is an explicit candidate in Figure 3. Because v_1 only has one neighbor

with label B while u_1 has two. However, it also stores candidate $\langle u_4, v_9 \rangle$, which is filtered out by DCS in Figure 3.

Matching order: RapidFlow removes the restriction that the matching order for $\Delta M_{u_x,u_y}$ must start from (u_x, u_y). For each query edge (u_x, u_y), it builds a reduced query graph Q' by removing u_x, u_y, and their incident edges, and precomputes a matching order ϕ for Q' with arbitrary order-generation methods (the authors choose RI algorithm [10] in their experiments, so as we). During the online phase, it constructs a local index Idx_l for Q', composed of candidates related to the update edge (v_x, v_y). Then it computes matches of Q' using ϕ and Idx_l, and finally concatenates them with $\langle u_x, v_x \rangle$ and $\langle u_y, v_y \rangle$ to obtain full matches.

Enumeration techniques: RapidFlow introduces dual matching, an enumeration technique leveraging query graph automorphisms. It groups query edges into auto-sets, where edges in the same auto-set are equivalent under automorphisms. After computing $\Delta M_{u_x,u_y}$, matches $\Delta M_{u'_x,u'_y}$ for other edges in the same auto-set can be derived by applying the automorphism mapping F.

5.5 CaliG [43]

Index: The index of CaliG algorithm, which is also named CaliG, is similar to DCS of Symbi, except for two differences: First, it is based on the undirected query graph Q. Second, for each valid candidate v of u, it requires an injective mapping between their neighborhoods

Specifically, vertex candidates in CaliG are also divided into implicit and explicit candidates [2]. v is a candidate of u if v has the same label as u. But v is an explicit candidate only if there is an injective mapping F from of $N(u)$ to $N(v)$ which satisfies:

(1) For any $u' \in N(u)$, it is mapped to an $F(u') \in N(v)$ that is an explicit candidate of u'.
(2) For any $u', u'' \in N(v)$, $F(u')! = F(u'')$.

To verify injective mappings, CaliG constructs a bipartite graph B for each candidate pair $\langle u, v \rangle$, where the vertex sets U and V of B are the neighbor sets of u and v. An edge connects $u' \in U$ and $v' \in V$ if v' is a valid candidate of u'. CaliG then checks for an injective matching in B using the Hopcroft–Karp algorithm [17].

CaliG's edge index records directed edges between candidate pairs. There is an edge from $\langle u, v \rangle$ to $\langle u', v' \rangle$ if $L(v, v') = L(u, u')$ and v is an explicit candidate of u, or $\langle u, v \rangle$ changes from explicit to implicit after $\langle u', v' \rangle$ changes to implicit. This index supports efficient updates. When a candidate becomes implicit (or explicit) due to graph updates, only its out-neighbors (or in-neighbors) need to be updated.

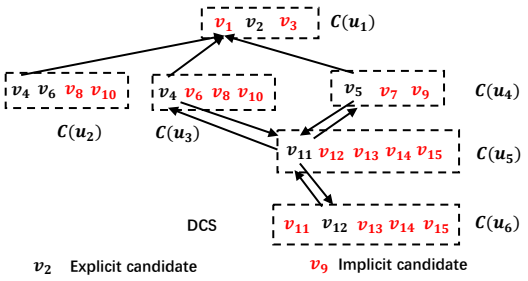


Fig. 5. Example of CaliG structure

[2]The original paper uses "on" state and "off" state to distinguish candidates. We change the names to keep consistent with other algorithms.

Example 5.4. Figure 4 shows an example of CaliG built on the query graph in Figure 1a and the data graph in Figure 1b (we omit the bipartite graphs). To keep the figure clear, we only present a part of the edge candidates that are connected with $\langle u_1, v_1 \rangle$. Compared to DCS in Figure 3, CaliG can additionally filter out $\langle u_1, v_1 \rangle$. v_1 is linked to the same neighbor v_4 in the candidate set of both u_2 and u_3, and thus cannot pass the injective check. CaliG can also filter out $\langle u_4, v_7 \rangle$, as CaliG requires v_7 to have explicit neighbors in u_1. On the other hand, DCS does not have this requirement as u_1 is a precursor of u_4 in the DAG, and thus it cannot filter out $\langle u_4, v_7 \rangle$.

The injective check gives CaliG a stronger filtering ability than DCS, but also a higher memory and update cost. We need to store $O(|V| \times |V_Q|)$ bipartite graphs and compute injective matches in them. Moreover, DSC uses DAG to build the index, while CaliG is based on the undirected query graph. This difference gives CaliG a stronger filtering ability, as pointed out in the above example. However, due to this difference, CaliG cannot directly tell whether the endpoints of an inserted edge become valid candidates or not. It follows a "propagate first, roll back later" strategy. which assumes that one of the endpoints is a valid candidate and tries to propagate the changes first, then rolls back the changes if they finally turn out to be illegal. Such a strategy further increases the update cost.

Matching order: The matching order of CaliG is called kernel-shell order. The query vertices are divided into two kinds. Shell vertices are an independent set of Q, and kernel vertices are the remaining vertices. After matching kernel vertices, the shell vertices are isolated and can be matched without backtracking. Therefore, for each query edge (u_x, u_y), CaliG finds a maximum independent set without u_x and u_y as shell vertices, and classifies the remaining vertices as kernel vertices. Then it generates a matching order that starts with u_x, u_y, followed by kernel vertices, and ends with shell vertices. The kernel vertices are ordered according to their degrees.

Enumeration techniques CaliG checks if the candidate set of a shell vertex is empty as soon as all its neighbors are matched, in order to prune failed search branches in advance.

5.6 NewSP [28]

Index: NewSP builds no indexes. Instead, it stores the frequency of each label in the neighborhood of a data vertex v, and uses NLF rule to filter candidates on-the-fly when computing candidate sets in the matching process.

Matching order: NewSP can use any order generation methods to generate the matching order ϕ that starts with a query edge. We use the same order as GraphFlow in our experiments.

Enumeration techniques: The key technical contribution of NewSP is the expansion delay, which is an extension of the isolated vertex method of Sybmi. NewSP splits the match of a query vertex u, namely line 8-19 of Algorithm 1, into two operations, candidates computation (line 10-11, denoted as $CPT(u)$) and match expansion (line 12-19, denoted as $EXP(u)$). It delays $EXP(u)$ to the position before $CPT(u')$, where u' is the first right neighbor of u. The vertices between u and u' are not connected with u, and thus their match process is not influenced by the match of u. By delaying $EXP(u)$, we can delay the split of partial match $\mathcal{M}$.

Besides the expansion delay, NewSP has two additional techniques: multi-expansion and candidate set reuse, which all target to share candidate set computation among different search branches. Details are omitted here due to space limitation.

5.7 Implementation and Analysis

Implementation: We implement the vertex index using bitmaps similar to [22, 25, 32]. For each data vertex v, we store a bitmap of $|V_Q|$ bits, where the i_{th} bit indicates whether v is a candidate for u_i in $|V_Q|$. This design enables constant-time candidate verification using only $O(|V| \times |V_Q|)$

bits. For the edge index, we build a two-layer map: the first layer maps each query edge (u, u') to a second-layer map, which then maps each data vertex v to its neighbors v', indicating that (v, v') is a candidate of (u, u'). Moreover, [25] suggests integrating the NLF filter into DCG and DCS. We adopt this improvement by considering v as an implicit candidate of u only if it satisfies both the original index rules and the NLF condition.

Complexity Analysis: All the indexes introduced above have a space complexity and time complexity of $O(|V_Q \times |V| + |E_Q \times |E|)$. Specifically, $O(|V_Q \times |V|)$ of space cost is consumed by the vertex index, and $O(|E_Q \times |E|)$ is consumed by the edge index. Though the complexity is the same, DCG and DCS are more complex than NLFI as they need to propagate changes to multi-hop neighbors. CaliG further increases the cost by maintaining a bipartite graph for each $\langle u, v\rangle$. Moreover, after integrating the NLF rule with DCG and DCS, the filter ability also obeys this order.

Metrics to evaluate the indexes. Many prior works use candidate count to evaluate the effect of the indexes [36, 43]. They assume that the gap in candidate count reflects the gap in acceleration power. However, according to our experimental results (Section 6.5), this assumption does not always hold. We can also use the example in Figure 1 to explain this phenomenon. The CaliG index shown built for the query graph in Figure 1a and data graph in Figure 1b, which is shown in Figure 5, only has 7 explicit candidates. On the other hand, NLFI shown in Figure 4 has 17 candidates. The candidate counts suggest that there is a huge gap in filtering ability. However, consider all the 24 cases where we delete an arbitrary edge from the data graph in Figure 1b, and use GraphFlow to enumerate the decremented matches. There will be 21 partial matches generated if we use index CaliG and 28 partial matches for NLFI. The difference is much smaller.

The reason is that many candidates are never, or seldom visited in the match process. We will only check $\langle u, v\rangle$ when all left neighbors of u are matched by neighbors of v. Otherwise, $\langle u, v\rangle$ will not be visited. Whether it is in the index or not does not matter. Consider NLFI in Figure 4. $\langle u_2, v_{10}\rangle$ and $\langle u_4, v_9\rangle$ are stored in the index, but they are isolated and never visited. $\langle u_4, v_7\rangle$ is visited only when we delete (v_7, v_{13}) and (v_7, v_{14}), and compute $\Delta M_{u_4,u_5}$. In this case, NLFI will generate one more partial match than CaliG, since it cannot filter out $\langle u_4, v_7\rangle$. But this partial match will also be quickly pruned, as the next vertex to match is u_3, and v_7 has no neighbor in the candidate set of u_3. It is similar for many other candidates like $\langle u_5, v_{13}\rangle$ and $\langle u_6, v_{14}\rangle$. As a result, candidate count underestimates the filtering ability of NLFI.

In conclusion, candidate count cannot correctly reveal the acceleration effect of indexes. We directly compare the end-to-end processing time in experiments. We also compare the number of partial matches generated in the matching process, as well as the update speed and memory usage of different indexes.

6 Experimental Evaluation

In this section, we experimentally evaluate the 6 algorithms that we introduce in our paper, especially focusing on the 6 indexing methods proposed in these algorithms. Specifically, the algorithms under comparison are GraphFlow [20], TurboFlux [22], Symbi [32], RapidFlow [35], CaliG [43] and NewSP [28]. The indexing methods we compare are: no index (denoted as NI, used in Graphflow), NLF filter (denoted as NLF, used in NewSP), materialized NLF index (denoted as NLFI, used in RapidFlow), DCG (used in TurboFlux), DCS (used in Symbi) and CaliG (used in CaliG).

6.1 Experiment Implementation

Although many CSM algorithms provide open-source implementations, they differ in graph storage, join methods, and index implementations. To ensure consistency and decouple indexing from enumeration, we reimplemented all algorithms based on the original papers and available code. The graph is stored as sorted linked lists. Each vertex's neighbors are ordered first by edge label and then

by vertex ID within each label group. Label-based sorting allows direct access to label-constrained neighbors, while ID-based sorting accelerates neighborhood joins during enumeration (line 11 of Algorithm 1). Since the join method requires sorted sets, pre-sorted neighbor lists eliminate the need for additional sorting. These steps are the basic building blocks for all CSM methods, and thus do not bias toward specific ones. We also compare with other storage methods in Section 6.8. The join method used in the candidate generation step of the matching process is the leapfrog join [39]. The indexes are implemented with the methods we introduce in Section 5.7: bitmaps for vertex indexes and two-layer maps for edge indexes. We also integrated the improvements proposed in [25], namely combining the NLF filter with DCG and DCS, and adding NEC technique into Symbi. All codes are implemented with C++, compiled with GCC 11.4 and O3 option. Experiments were conducted on a server with 256 GB of memory and two Intel Xeon 2.9 GHz 16-core CPUs. We make all of our codes and data open-sourced [3]

6.2 Datasets and Query Workload

Datasets: We use 3 datasets which are frequently used in prior works [25, 28, 32, 35, 36] and WDBench[5], a benchmark for static knowledge graphs based on Wikidata. We list the metrics of these algorithms in Table 2, where $|\Sigma_V|$ and $|\Sigma_E|$ denote the number of vertex labels and edge labels, respectively. Details about these datasets are as follows:

LiveJournal: a static graph for social networks with no labels. We generate labels for vertices in them following a Zipf distribution, and randomly extract 5% of the edges as update streams.

Netflow: a dynamic graph for network traffic with original time orders and edge labels. We use the last 5% edges as update streams.

LSBench: a synthetic social network produced by the Linked Stream Benchmark data generator [24]. It is a dynamic graph with original time orders and edge labels. We find that most queries in it are very costly due to its size and high skewness (details will be analyzed in Section 6.3). Therefore, we decrease the update stream to the last $50k$ edges.

WDBench: a static knowledge graph derived from Wikidata. The original dataset contains 1.26 billion edges. However, storing the full graph and auxiliary data (e.g., neighbor label frequencies) requires over $150GB$ of memory, which is about 70% of the server capacity, leaving insufficient memory to build indexes (which costs $O(|V_Q| \times |V| + |E_Q| \times |E|)$ and may even exceed the graph itself). Therefore, we filter the original dataset by removing edges with labels not in the query set (details of the query set will be introduced in the following paragraphs). After filtering, there are 268 million edges left. Even with this filtered dataset, complex indexes like CaliG still frequently run out of memory, as will be shown in Section 6.3. **Thus, we emphasize that heavy index structures are impractical for massive graphs due to memory constraints.** Due to the massive edge number, we randomly extract 0.5% edges as update streams.

In experiments, we use all edges except those in the update stream to build the initial graph. CSM algorithms are initialized on this graph. Then we insert edges in the update streams into the initial graph and use CSM algorithms to compute incremental matches for each edge insertion. Since edge insertion and deletion processes in CSM algorithms are symmetric, we focus on insertion streams in the main part of experiments as in previous works [25, 35], but we will also carry out experiments with deletion streams and mixed update streams in Section 6.7.

Queries: We generate queries using random walks for LiveJournal, Netflow and LSBench. Queries are divided into three categories: tree queries, sparse queries (average degree ≤ 3), and dense queries (average degree > 3). For each category, we generate 50 queries. By default, each query contains 6 vertices. We also evaluate other sizes in Section 6.9. All queries are guaranteed to produce incremental matches, as not-qualified queries are filtered out during generation. The last dataset, WDBench, is paired with a SPARQL query set. We extract the basic graph pattern (BGP)

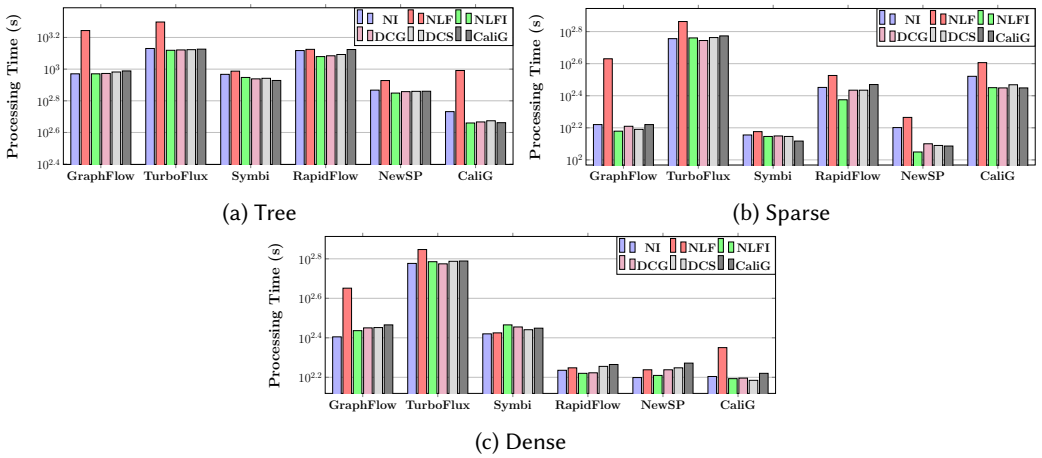
Table 2. Datasets

Dataset	|V|	|E|	$\|\Sigma_V\|$	$\|\Sigma_E\|$
LiveJournal[1, 26]	4.9M	42.9M	30	1
Netflow [36]	3.1M	2.9M	1	7
WDBench[5]	104M	268M	1	41
LSBench [2, 36]	5.2M	20.3M	1	44

queries and convert them into query graphs. Other SPARQL query types, such as path navigation or those containing OPTIONAL clauses, cannot be represented as subgraph matching queries and are therefore excluded. We also remove queries with two or fewer edges, as their incremental matching can be answered with simple one-hop neighbor access, making algorithmic or index differences negligible. After filtering, we obtain 86 queries, including 79 tree-shaped and 7 sparse graph queries. The query sizes range from 4 to 10, with 79 queries containing no more than 6 vertices.

We set a maximum processing time of 1 hour per query. Queries not completed within this time are marked as unsolved, and their processing time is set to 1 hour when calculating average processing time. Moreover, we set a memory usage upperbound of 150GB (including both the memory usage of the graph and the index). If the memory usage of an algorithm exceeds this upper bound, the algorithm will be terminated and the associated query will be marked as unsolved. Processing time of such unsolved queries is also set to 1 hour.

6.3 Overall Comparison



(a) Tree (b) Sparse

(c) Dense

Fig. 6. Processing time of algorithms in LiveJournal

Comparison of Algortithms: We first compare different algorithm variants and implementations. We use the LiveJournal dataset in this experiment. Figure 6 shows the average processing time of all 36 combinations of enumeration algorithms and indexing methods. We can see that the enumeration techniques make a huge difference. NewSP and Symbi generally have the best performance, as their computation share techniques and enumeration-delay techniques are very effective. The impact of indexing methods on LiveJournal dataset is less significant than enumeration methods. However, as will be shown in Section 6.3, indexes have higher acceleration power in

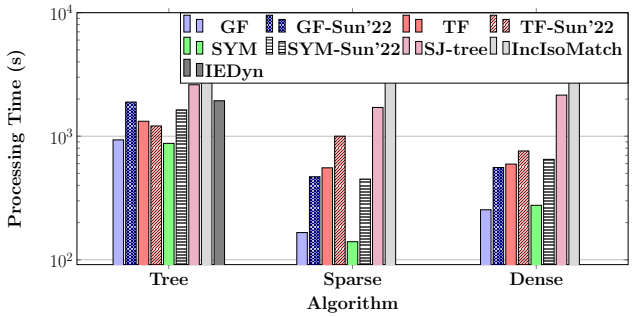


Fig. 7. Comparison of different implementations

more skewed datasets like Netflow and LSBench. The processing time of all combinations in other datasets is shown in the technical report [3] due to space limitations. In the following section, we will fix the enumeration method to NewSP, and focus on the influence of indexes.

Figure 7 further compares our implementation with the widely used library by [36] (denoted Sun'22). Both implementations include three shared algorithms—GraphFlow (GF), Turboflux (TF), and Symbi (SYM). We use their default indexing methods in our implementation (no index for GF, DCG for TF, and DCS for SYM). Our implementation differs from Sun'22 in three aspects: First, graph storage. Sun'22 sorts each vertex's neighbors by ID but mixes different labels, while we first sort by labels and then by IDs within each label group, enabling faster label-constrained neighbor retrieval. Second, index storage: Sun'22 stores candidate sets in vectors, whereas we use bitmaps for $O(1)$ lookup. Third, neighbor set join: Sun'22 uses edge verification, scanning the smallest neighbor set and checking the existence of each item in other sets, with a cost of $O(N_{\min} \log(N_{\max}))$, where N_{min} and N_{max} denote the size of the smallest and the largest set, respectively. We adopt a leapfrog join with $O(N_{\min}(1 + \log(N_{\max}/N_{\min})))$ cost, which is more efficient for large sets. These improvements yield higher efficiency in our implementation.

We also compare 3 earlier algorithms in Figure 7: IEDyn [18], SJ-tree [11] and IncIsoMatch [13]. We use the SJ-tree and IEDyn implementation in [36], and implement IncIsoMatch by ourselves, as we found no open-sourced implementations. IEDyn applies only to acyclic (tree) queries and is far less efficient than later algorithms such as GraphFlow and Symbi. SJ-tree incurs $O(|E|^{|E_Q|})$ memory usage—orders of magnitude higher than modern methods—causing it to exceed memory limits in over 70% of the queries. Even when it runs, it remains inefficient. IncIsoMatch, in contrast, performs static subgraph matching within affected regions for every update and often generates billions of irrelevant matches that do not contain the updated edge. It exceeds the time limit in all queries. Given their inefficiency and incompatibility with modern indexing methods, we exclude these algorithms from further analysis.

Comparison of Indexes: We fix the enumeration method to NewSP and evaluates its performance with various indexing methods in all 4 datasets. with average processing time shown in Figure 8 and the percentage of unsolved queries shown in Table 3. We also report the percentage of queries unsolved due to memory limit exceeding behind the total unsolved rate in WDBench. Note that WDBench has only one unified graph type, as it contains no dense queries.

To avoid unsolved queries shadowing the effect of indexes in queries with shorter processing time, we also present the average speed improvement of different indexing methods compared to NI method in Figure 9, which is computed as

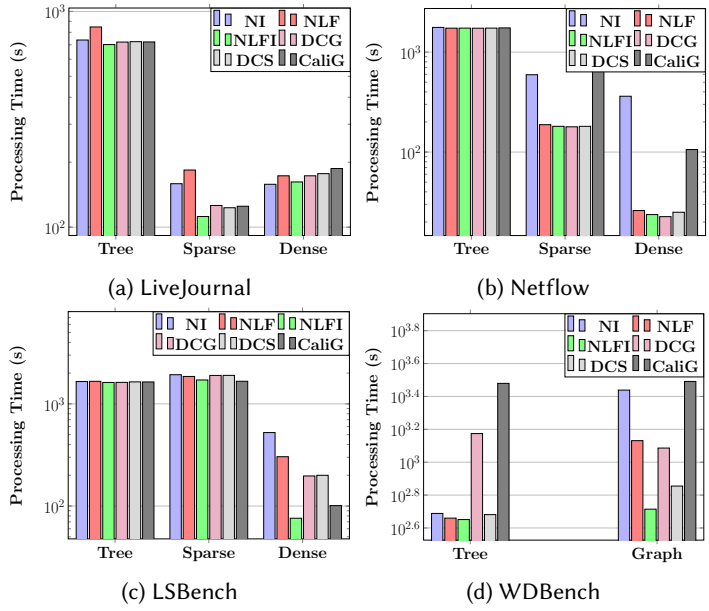


Fig. 8. Processing time of the NewSP algorithm combined with different indexes

Table 3. Percentage of unsolved queries(%)

DataSet	Query Type	NI	NLF	NLFI	DCG	DCS	CaliG
LiveJournal	Tree	6	10	6	6	6	6
	Sparse	0	0	0	0	0	0
	Dense	0	0	0	0	0	0
Netflow	Tree	54	42	42	42	42	42
	Sparse	8	4	4	4	4	16
	Dense	0	0	0	0	0	0
LSBench	Tree	44	44	40	40	40	42
	Sparse	52	50	44	52	52	42
	Dense	14	8	2	2	2	2
WDBench	Tree	10	10	10	34	10 (3.7)	84 (26.6)
	Graph	57	14.3	0	14.3	0	85.7(57)

$$exp(\frac{1}{|Q|}\Sigma_{q_i \in Q} ln \frac{t_i'}{t_i}) - 1 \tag{1}$$

where t_i and t_i' denote the processing time of NewSP with and without the given index for a query q_i, and $|Q|$ denotes the total number of queries. We adopt the geometric mean instead of the arithmetic mean to reduce the impact of outliers

We also present the index maintenance cost in Netflow dataset in Figure 10 (excluding NI and NLF, as they build no index), In the memory cost evaluation, we split the indexes into vertex indexes

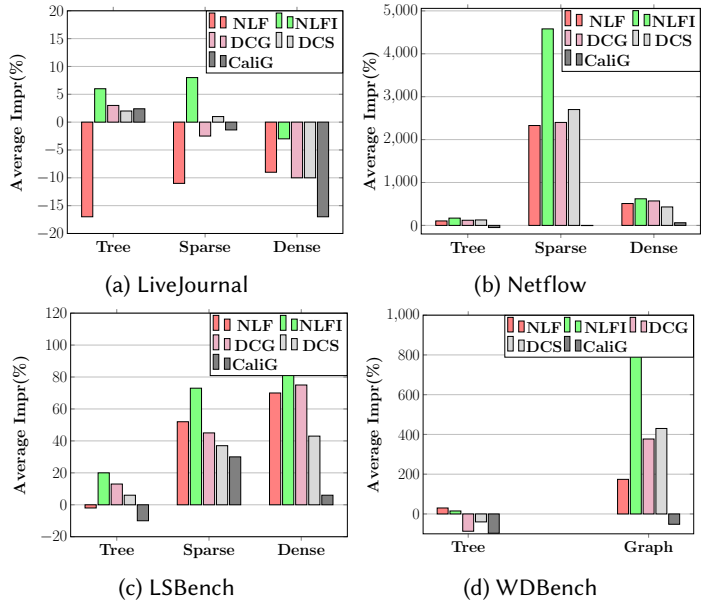


Fig. 9. Average improvement compared to NI

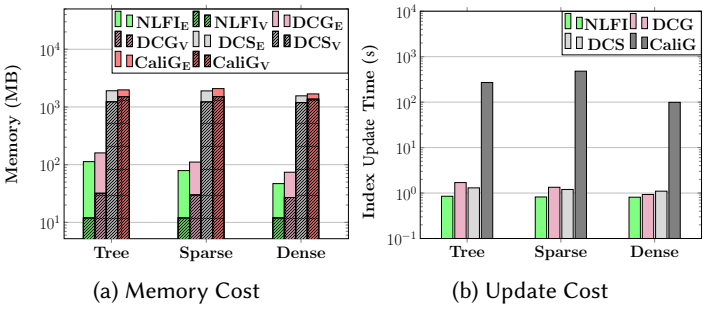


Fig. 10. Index maintenance cost in Netflow

and edge indexes (marked with subscription V and E, respectively), and present the memory usage of different parts separately. The maps and bipartite graphs to maintain neighborhood information in DCS and CaliG are considered as a part of the vertex indexes. In the update cost evaluation, we compute the time spent on index update when processing the update stream. This time is included as part of the total processing time.

From the perspective of indexing method: Taking into account both the maintenance cost and acceleration effect, we recommend NLFI for most circumstances. NLFI is the most lightweight indexing method with the lowest update cost and memory cost (as shown in Figure 10). Moreover, it has a comparable, and even better, acceleration effect than more complex indexes (as shown in Figure 6, 8 and 9). The NLF method, namely NLF filter without materialization, is not as good as NLFI in most queries due to the NLF computing cost. On the other hand, complex indexes like DCG and DCS can capture multi-hop information, but this advantage is not significant enough. A more detailed case study about NLFI and more complex indexes will be given in Section 6.4.

Moreover, Figure 10a shows that the edge index occupies most of the memory usage in NLFI and DCG. The vertex indexes are large in DCS and CaliG, as they need to build additional maps and bipartite graphs for each candidate. The update cost of CaliG is higher than other indexes, as it needs to compute injective matches in a bipartite graph for each candidate. In large datasets such as WDBench (Figures 8d and 9d), the cost of index maintenance becomes much more significant. CaliG runs out of memory in over 30% of the queries and often exceeds the time limit even when it can be executed. It has high overhead in maintaining bipartite graphs and performing injective matching checks, resulting in very expensive initialization and updates. DCG exhibits similarly high update costs. Although its performance is close to DCS on small graphs, verifying candidate validity through neighbor-set checks becomes a major bottleneck in larger graphs. In contrast, DCS maintains additional maps for neighborhood checking—more memory-intensive but more efficient for updates on large graphs.

From the perspective of different datasets: The processing time of tree and sparse queries in Netflow and LSBench is much higher than LiveJournal (Figure 8). These 2 graphs are highly skewed. 70.9% / 13.7% of edges in Netflow / LSBench share the most frequent label. Moreover, 79% / 42% of the edges in Netflow / LSBench are connected to the top 1% high-degree vertices, much higher than 29% edges in LiveJournal. Due to the skewness, it is highly possible to include a high-degree vertex v into matching, and the massive neighbors of v will lead to an explosion in the number of partial matches, This also increases the importance of indexes. When there is a rare-label edge or a complex local topology (like a triangle) in the query graph, indexes can capture this information and use it to filter the candidates, decreasing the search space. Therefore, the improvement of the indexing method in Netflow can be as large as 4580%. Moreover, in large datasets like WDBench, the index maintenance cost may become the system bottleneck, and we recommend to only use the lightweight NLFI method.

From the perspective of query types: The effect of indexes in cyclic graph queries is generally more significant than the tree queries, as their more complex topology enhances the filtering capability of indexes. However, for dense queries where the complex topology typically leads to very few matches, index updates can become a bottleneck, especially for complex indexes (Figure 9a).

6.4 Case Study of Different Indexes

Compared with NLFI, both DCS and DCG incorporate multi-hop neighborhood information. However, as shown in Section 6.3, they exhibit no performance advantage. To investigate this, we conduct an in-depth case study.

We use the Netflow dataset, enumeration method NewSP, and design a set of chain queries. Each query consists of 8 vertices. The first 7 vertices (u_1–u_7) form a core connected by a single and most frequent edge label A. The last vertex (u_8) connects to u_7 via an edge with a different label. we verify the label of (u_7, u_8) to generate 6 queries. (Figure 11). In this setup, only edge (u_7, u_8) exhibits low label frequency and high selectivity. In DCS and DCG, candidate sets are recursively refined through multi-hop neighborhood filtering, allowing the selectivity of (u_7, u_8) to propagate upward through the query structure. In contrast, NLFI relies solely on one-hop neighbor-label information, leading to weaker filtering capability.

Then, we gradually decrease the diameter of the core, namely the subgraph induced by $u_1 \sim u_7$, by 2 methods, either randomly moving vertices in the chain to other tree branches (generating tree queries), or adding cyclic edges (generating graph queries), as shown in Figure 11.

The experimental result is shown in Figure 12a and Figure 12b. **The original chain queries are classified into tree queries with core diameter** 6. From the results, we can see that DCS and DCG has an advantage when the core diameter is at least 4, but the advantage quickly disappears

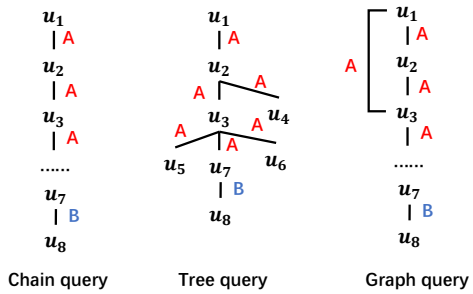
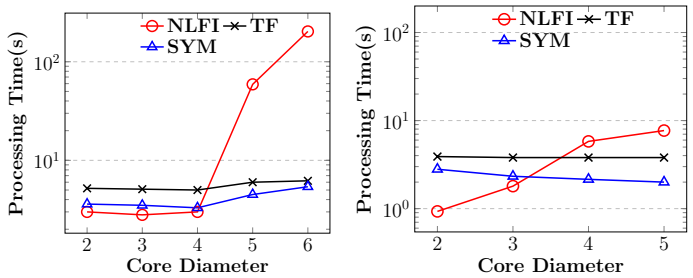


Fig. 11. Different types of designed queries

when the diameter shrinks. When the updated data edge is mapped to a query edge, like (u_2, u_3) in the tree query of Figure 11, we reach the endpoints of the high selectivity edge with only $1 \sim 2$ hops of enumeration if the diameter is small, and thus the difference among NLFI and complex indexes is not significant.



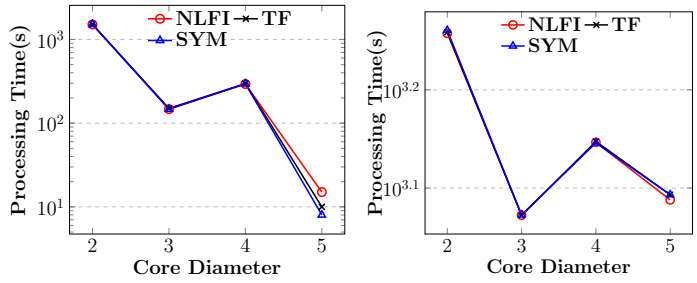
(a) Tree queries (b) Graph queries

Fig. 12. Case study with different designed queries

To test general cases, we generate arbitrary query graphs with 8 vertices, 2 labels, and a single-label core part made up of $u_1 \sim u_7$. We vary the core diameter and create 50 queries for each configuration, using both the Netflow and LiveJournal datasets. We test two query types: tree and cyclic graphs, but due to space limitations, only tree query results are shown in Figure 13, while graph query results are provided in the technical report [3]. The results show that DCG and DCS gain a slight advantage when the core diameter is 5. However, this advantage is smaller than in the designed query of Figure 12, as the high-selectivity vertex u_8 tends to be reached within fewer hops. In the less label-skewed LiveJournal dataset, the gap becomes almost negligible, since the label of u_8 becomes more frequent and less selective, diminishing the benefit of multi-hop information captured by DCG and DCS.

A more general study can be found in Section 6.9, where we further remove the constraint of a single label core and test queries with different diameters and label counts. The advantage of DCS and DCG disappears, since we can encounter different labels with much fewer hops.

In conclusion, DCS and DCG have an advantage against NLFI only when the candidate filtering relies heavily on multi-hop information. A representative case is that the query graph contains a single-label subgraph with a large diameter, as well as a high selectivity structure (like an edge with a very rare label) linked to this subgraph. Such a constraint is very difficult to reach in real-world

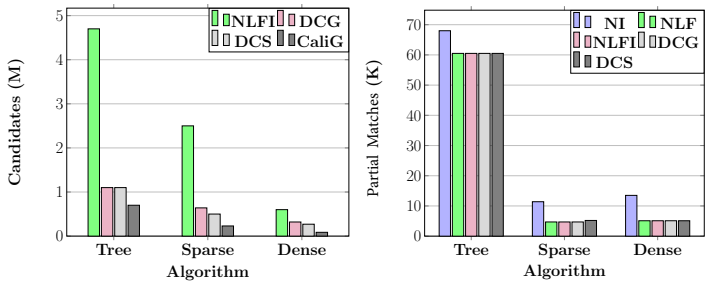


(a) Netflow (b) LiveJournal

Fig. 13. Case study on different dataset

applications. According to the former studies [9], large queries are rare in real-world applications. It is also in accordance with the query set in WDBench, where 90% of queries have a size no larger than 6 and diameter no larger than 3. It is even more difficult to get a low label diversity at the same time. **Therefore, we recommend using NLFI in most cases. DCS and DCG can be attempted only when the query graph contains a subgraph with a large diameter and a low label diversity**.

6.5 Verify the Effectiveness of Metrics



(a) Candidate count (b) Partial match count

Fig. 14. Evaluation result of different metrics in Netflow

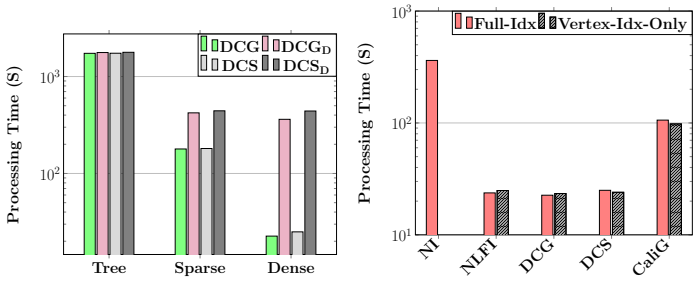
In this section, we test the effectiveness of different evaluation metrics. Figure 14a presents the candidate count of different indexes. Figure 14b shows the average partial match count in the incremental match enumeration of each edge insertion. The dataset used is Netflow and the enumeration method is NewSP. In Figure 14b we also include the NI method as a comparison. Note that we omit NLF as its candidate count and partial match count are the same as NLFI. We can see that all indexes have smaller partial match counts than the NI method, but the difference among these indexes is very small. This is consistent with the processing time in Figure 8b (CaliG has a higher processing time due to its high index update cost, as shown in Figure 10). However, the candidate counts of these indexes have a huge difference. Complex indexes have a much smaller candidate count than simple indexes like NLFI. The reason is as analyzed in Section 5.7. Many candidates in simple indexes will not be visited in enumeration. Therefore, **we suggest researchers not to use candidate count to estimate acceleration power of indexes, as it may be misleading**.

6.6 Ablation Study

In this section, we analyze the importance of different index components. Figure 15 shows the effects of two key components: the NLF filter in DCG/DCS, and edge indexes in NLFI, DCG, DCS, and CaliG. The experiment uses the Netflow dataset and the NewSP enumeration method.

In Figure 15a, DCG/DCS without NLF filter are denoted as DCG_D/ DCS_D. The NLF filter reduces processing time by up to 94%. **Thus, we always recommend using NLF filter in indexes.** When a query vertex u is connected to a set of neighbors S with the same label l, the NLF filter ensures that a candidate vertex v has at least $|S|$ neighbors with label l. These neighbors can be injectively mapped to query vertices in $|S|$, which is not guaranteed by DCG and DCS. Note that CaliG inherently enforces a stronger injective constraint, so applying the NLF filter to it is unnecessary. However, CaliG's bottleneck lies in update efficiency, as shown in Figure 10.

Figure 15b compares the processing time of full-index and vertex-index-only schemes to assess the impact of edge indexes. Due to space constraints, we only present results for dense queries, but trends remain similar for other query types. The NI method is included for comparison. The results show that most of the acceleration comes from vertex indexes alone. While edge indexes provide additional speedup, their effect is limited. Edge indexes can fetch candidates of a query vertex u among the neighbors of a data vertex v and avoid index checks. When using bitmaps for vertex indexes, an index check only takes $O(1)$ time, making its cost negligible. Moreover, edge indexes consume a significant part of memory usage in NLFI and DCG. **We recommend removing edge indexes if the memory is short.**



(a) Effect of NLF (b) Effect of edge index

Fig. 15. Ablation study of different components of indexes

6.7 Evaluation on Update Types

In this section, we evaluate the performance of the algorithms under different types of update streams using the Netflow dataset and the NewSP enumeration method. The results are shown in Figure 16. Due to space limitations, we only report the results for dense queries. For deletion streams, we first build the initial graph using all edges and then delete the last 5% of them. For mixed streams, we build the initial graph with the first 97.5% of edges, insert the last 2.5% of edges, and finally delete the same 2.5% of edges. From the results, we observe that the different update streams yield nearly identical processing times, confirming the symmetry of CSM algorithms. However, the variant using the CaliG index is an exception. It achieves higher efficiency in deletion and mixed streams because the update mechanism of CaliG is inherently asymmetric. During edge insertions, it adopts a costly "propagate first, roll back later" strategy, as it cannot immediately determine the validity of updated vertices (see Section 5.5). This drawback does not occur during deletions, where updates are simpler. Nonetheless, CaliG still shows no clear advantage over other indexing

methods. In consideration of its high memory cost and update cost in insertion streams, we still do not recommend it.

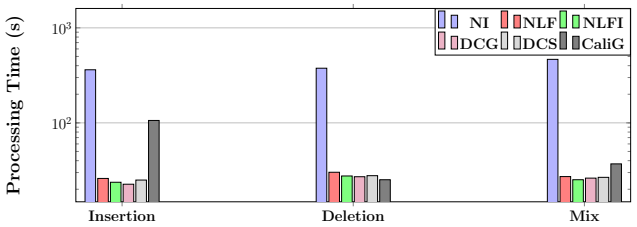


Fig. 16. Comparison of different update types

6.8 Evaluation on Storage Method

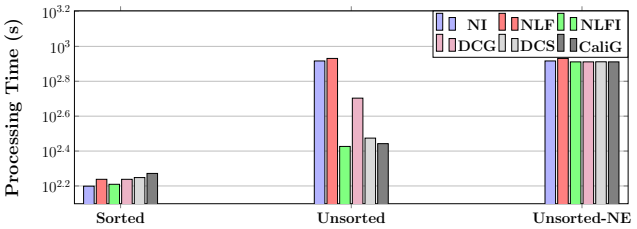


Fig. 17. Comparison of different storage methods

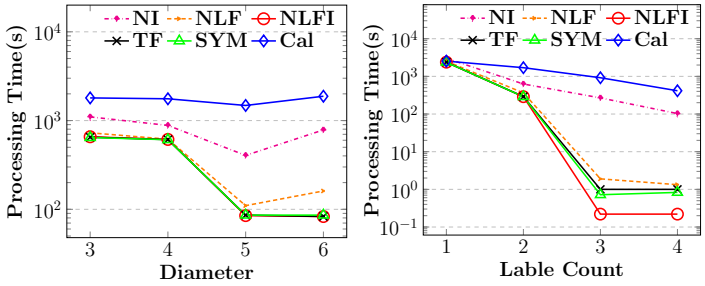
In this section, we evaluate different graph storage methods. We use the Netflow dataset and NewSP enumeration method. Due to space limitations, we focus on dense queries, and the results are shown in Figure 17. "Sorted" denotes the original implementation. "Unsorted" denotes the variant with unsorted neighbor lists in graph storage. "Unsorted-NE" denotes the unsorted variant without edge indexes. The results show that unsorted neighbor lists result in a much lower efficiency. The performance loss of NLFI, DCS, and CaliG is smaller due to their edge indexes, which maintain the efficient, sorted structure for edge candidates. DCG suffers a larger loss, as its edge index covers only a spanning tree instead of the entire query graph. After removing edge indexes (Unsorted-NE), the comparison among indexing methods becomes identical to that of the variants with sorted linked lists, indicating that the filtering ability remains unchanged. This suggests that when the graph storage is inefficient and cannot be modified, edge indexes can be used to store neighborhoods more efficiently.

6.9 Evaluation on Query Metrics

In this section, we evaluate the influence of query metrics, including label count, diameter and query size. We use the Netflow dataset and the NewSP enumeration method.

Label count and diameter: We fix the query size at 8 and vary the label count and diameter to conduct experiments. When evaluating label count, the diameter is fixed at 4. When evaluating diameter, the label count is fixed at 2. We generate 50 queries for each setting. Due to space limitations, we do not distinguish query types. The query set contains a mixture of tree and graph queries, each accounting for 50%. From the result, we can see that as the diameter increases, the processing time decreases. The degree distribution of real-world data graphs is skewed. Most

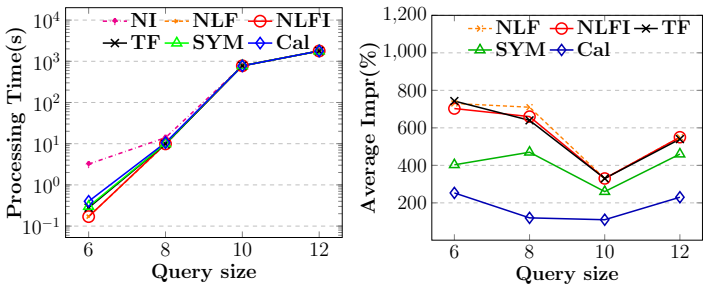
vertices have very low degree, even with degree 1, resulting in a constrained number of multi-hop chains (since a vertex in the middle of a chain needs a degree of at least 2). Moreover, as the diameter increases, the advantage of indexing methods becomes more significant, as they can better leverage topology information for filtering. A similar trend appears with label count: as it increases, query matches decrease, and the filtering effect of indexes becomes more pronounced. However, CaliG always has low efficiency due to its high update cost.



(a) Query diameter (b) Label count

Fig. 18. Performance varying with query metrics

Query size: We vary the query size from 6 to 12 and generate 50 queries for each setting. We focus on dense queries, as most size-12 sparse and tree queries are unsolved. Additionally, we reduce the update stream to 50k edges to mitigate the long processing time of larger queries. The results in Figure 19a show that processing time increases rapidly with query size, since the computational cost of CSM grows exponentially. Note that the gap among different methods in processing time decreases due to the increasing number of unsolved queries, as the same processing time (1 hour) is set to all methods for unsolved queries. But there is still a notable speed improvement for solved queries according to Figure 19b. However, we do not observe a notable change in the acceleration power of indexes in Figure 19b. As discussed in Section 6.4, DCG and DCS are advantageous mainly for queries with large diameters. Yet, query graph diameters grow slowly with query size, with an average value of only 4.8 for size-12 queries. Note that we do not test larger queries, as further increasing the query size results in higher unsolved query percentages and shadows the difference among algorithms. Prior studies [25, 36] also use sizes no larger than 12.



(a) Processing time (b) Average improvement

Fig. 19. Performance varying with query size

6.10 Summary

In summary, we have the following conclusions in our experiments:

(1) NLFI is the best choice of indexes in most cases, which is lightweight and has a comparable filtering ability as more complex indexes. DCS are recommended when the query graph contains a subgraph with a large diameter (≥ 4) and a low label diversity. DCG is an alternative to DCS in case of memory shortage.
(2) Indexes are particularly crucial for skewed datasets with long processing times. In dense queries, the update cost of indexes may become the bottleneck. We recommend using the lightweight NLFI as a conservative choice in this case.
(3) NLF filter is a powerful tool for candidate filtering, and we suggest researchers to integrate it when designing indexes.
(4) Vertex indexes are the base of indexes, and edge indexes are optional. We may remove edge indexes when there is a memory shortage, as their acceleration effect is limited.
(5) Candidate count is not a good metric to evaluate indexes' acceleration power. We suggest to directly compare end-to-end processing time or the partial match count.

Moreover, our experimental evaluation also reveals several open questions:

(1) Necessity for CSM benchmark: Our results show that CSM performance varies across query workloads and datasets. Although static benchmarks such as WDBench can be used as alternatives, they fail to capture the dynamics of real-world updates. We argue that a dedicated CSM benchmark is needed to reflect realistic application characteristics. Building such a benchmark requires real-world data and query logs from diverse domains (e.g., social networks, online finance) and joint efforts from both academia and industry.
(2) Building of workload-aware CSM algorithms: Our experiments show that different indexing and enumeration methods perform best under different data and query workloads. Since CSM algorithms are applied across diverse scenarios, it is desirable to detect workload characteristics and dynamically switch strategies for optimal performance. How to efficiently identify workload properties and adapt to workload shifts remains an open problem.

7 Conclusion and Future Work

In this paper, we conduct an extensive experimental study on indexes in continuous subgraph matching. We decouple indexes from enumeration methods and compare existing indexes under a unified enumeration algorithm. Additionally, we analyze index components to assess their importance and evaluate metric effectiveness. For future work, we will explore enumeration algorithms, examining the impact of ordering strategies and techniques. We also plan to design a benchmark that reflects features of real-world CSM applications.

Acknowledgments

This work was partially supported by New Generation Artificial Intelligence-National Science and Technology Major Project under grant 2025ZD0123304, NSFC under grant 62532001, ARC under grant DP230101445 and FT210100303, and Australian Research Council Centre of Excellence for Mathematical Modelling of Cellular Systems under grant CE230100001. Wenjie Zhang is the corresponding author of this work.

References

[1] 2009. *LiveJournal dataset on SNAP website.* https://snap.stanford.edu/data/socLiveJournal1.html

[2] 2017. *Lsbench codes.* https://code.google.com/archive/p/lsbench/

[3] 2025. *Source code, datasets and queries for this paper.* https://github.com/XiangyangGouUNSW/CSM-Experiments

[4] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[5] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. Wdbench: A wikidata graph query benchmark. In *International Semantic Web Conference*. Springer, 714–731.

[6] 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.

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

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

[9] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.

[10] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14 (2013), 1–13.

[11] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849* (2015).

[12] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[13] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

[14] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[15] 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 international conference on management of data*. 1429–1446.

[16] 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.

[17] John E Hopcroft and Richard M Karp. 1971. A n5/2 algorithm for maximum matchings in bipartite. In *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*. IEEE, 122–125.

[18] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[19] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: accelerating subgraph matching with historical data. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1699–1711.

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

[21] 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 international conference on management of data*. 925–937.

[22] 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 international conference on management of data*. 411–426.

[23] Raghav Kulkarni and Supartha Podder. 2015. Quantum query complexity of subgraph isomorphism and homomorphism. *arXiv preprint arXiv:1509.06361* (2015).

[24] 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*. Springer, 300–312.

[25] Yukyoung Lee, Kyoungmin Kim, Wonseok Lee, and Wook-Shin Han. 2024. In-depth Analysis of Continuous Subgraph Matching in a Common Delta Query Compilation Framework. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[26] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[27] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1082–1093.
[28] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3324–3337.
[29] Ziyi Ma, Jianye Yang, Xu Zhou, Guoqing Xiao, Jianhua Wang, Liang Yang, Kenli Li, and Xuemin Lin. 2024. Efficient Multi-Query Oriented Continuous Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3230–3243.
[30] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.
[31] Seunghwan Min, Jihoon Jang, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. 2024. Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3257–3269.
[32] 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. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1298–1310.
[33] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3204–3216.
[34] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.
[35] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. Rapidflow: An efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.
[36] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1403–1416.
[37] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *arXiv preprint arXiv:1205.6691* (2012).
[38] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.
[39] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.
[40] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. 2022. Continuous multi-query optimization for subgraph matching over dynamic graphs. *Semantic Web* 13, 4 (2022), 601–622.
[41] Yihua Wei and Peng Jiang. 2024. GCSM: GPU-Accelerated Continuous Subgraph Matching for Large Graphs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1046–1057.
[42] Jianye Yang, Sheng Fang, Zhaoquan Gu, Ziyi Ma, Xuemin Lin, and Zhihong Tian. 2024. Tc-match: Fast time-constrained continuous subgraph matching. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2791–2804.
[43] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[44] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. 2011. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment* 4, 8 (2011), 482–493.

Received July 2025; revised October 2025; accepted November 2025