Subgraph Enumeration: Beyond Tree Decomposition

Qiyan Li
The Chinese University of Hong Kong
Hong Kong, China
qyli@se.cuhk.edu.hk

Jeffrey Xu Yu
The Hong Kong University of Science and Technology (Guangzhou), China
jeffreyxuyu@hkust-gz.edu.cn

Zongyan He
The Chinese University of Hong Kong
Hong Kong, China
zyhe@se.cuhk.edu.hk

ABSTRACT

We address the subgraph enumeration problem: given an unlabeled pattern graph p and an unlabeled data graph G, find all subgraphs in G isomorphic to p. Unlike labeled matching, the absence of label constraints creates exponentially larger search spaces with limited pruning opportunities. To address this challenge, we follow tree decomposition (TD) approaches that break complex patterns into smaller subgraphs (bags), compute matches for each bag, and join them to obtain final results. However, existing TD approaches suffer from suboptimal decomposition selection, incomplete symmetry-breaking usage, and expensive intermediate result materialization. We present MDSE (Minimal Decomposition-based Subgraph Enumeration) with three key contributions. We introduce minimal fractional hypertree decompositions (MinFHDs) that ensure compact bags and an efficient algorithm to explore all optimal-width decompositions. We develop new symmetry-breaking integration using complete rule sets with systematic selection for maximum pruning effect. To reduce materialization costs, we design MixJoin by embedding final result assembly within bag processing and formulate an enhanced cost model for attribute orders, incorporating both intersection and materialization overhead. Evaluation across 101 pattern graphs and 8 real-world datasets shows MDSE substantially outperforms existing algorithms.

PVLDB Reference Format:
Qiyan Li, Jeffrey Xu Yu, and Zongyan He. Subgraph Enumeration: Beyond Tree Decomposition. PVLDB, 19(9): 2303 - 2316, 2026.
doi:10.14778/3819518.3819552

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/magic62442/subgraph-enumeration.

1 INTRODUCTION

Given an unlabeled pattern graph p and an unlabeled data graph G, subgraph enumeration finds all subgraphs in G that are isomorphic to p. As one of the fundamental graph analysis operations, subgraph enumeration is widely used in real-world applications, including motif discovery [68], fraud detection [50], graph comparison [54], and functional prediction [7]. Unlabeled subgraph enumeration naturally arises in quantum computing compilation, where a compiler must map virtual qubits to physically connected qubits on quantum hardware. Since qubits are homogeneous and differ only in connectivity, this mapping is solved as an unlabeled subgraph matching problem [58], where multiple mappings must be evaluated since different mappings lead to different costs.

Traditional approaches for subgraph enumeration rely on backtracking algorithms [64] or worst-case optimal join algorithms (WCOJ) [45, 47, 65]. Sun et al. [61] show that these two algorithms are essentially equivalent. When node labels are available, these algorithms can be enhanced through sophisticated filtering and pruning techniques [3, 5, 6, 13, 22, 23, 32, 33, 42, 53, 61, 78]. The surveys summarizing these methods can be found in [29, 60, 76].

However, these labeled matching techniques cannot be effectively applied to unlabeled graphs due to the lack of label constraints. For unlabeled subgraph enumeration, researchers have developed various structural techniques to improve efficiency. These include symmetry-breaking methods that exploit automorphisms to eliminate redundant enumerations [18], intermediate result reuse strategies that cache neighbor set intersections across backtracking branches [20, 21, 59], and independent set optimizations that avoid unnecessary intersections when matching independent nodes in the pattern graph p [55, 56, 59]. Additionally, parallel and distributed approaches have been explored to scale across multiple cores and machines [34–36, 49, 51, 59, 71]. However, the performance improvements from these structural techniques remain limited for complex patterns, as the fundamental exponential search space is not substantially reduced.

To further reduce the search space, decomposition-based approaches have emerged as an alternative strategy. These methods partition a complex pattern graph into multiple sub-queries or join units $\{p_i\}$ and assemble their results to obtain matches for the original pattern p. Early decomposition approaches used join units such as stars [35, 62], cliques [36], and path sketches [75] that are subgraphs induced by selected paths of p. However, these approaches face a fundamental dilemma: either join unit processing is complex, or assembly is complex. For cliques and stars, join unit processing is efficient since their sizes are bounded, but assembly becomes expensive due to complex cyclic joins. Conversely, path sketches enable simple assembly but may result in large subgraphs as join units, making unit processing complex.

Tree decomposition (TD) [16] provides a more principled approach that does not restrict join units to a specific type. Given a pattern p, tree decomposition constructs a tree-structured representation T where each tree node (termed a bag) corresponds to a subgraph of p. The algorithm computes subgraph matches for individual bags, materializes intermediate results as relations, and joins these relations to produce final matches. Tree decomposition offers key advantages: it bounds bag sizes by the tree width or fractional hypertree width and guarantees that the assembly is a simple acyclic join. Recent TD-based approaches have shown effectiveness for subgraph counting and labeled subgraph matching [2, 38, 39, 74].

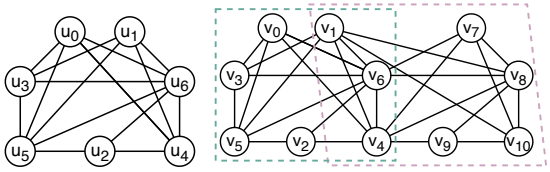
Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819552

However, unlabeled subgraph enumeration presents significantly greater challenges than both subgraph counting and labeled matching. The unlabeled setting generates exponentially more matches due to the absence of label constraints, while enumeration tasks require complete match materialization without the benefit of aggregation pushdown techniques available in counting. Consequently, substantial improvements to the tree decomposition framework are essential for effective unlabeled subgraph enumeration.

We identify three critical limitations in existing TD approaches. ① *suboptimal decomposition selection*: Existing approaches use brute-force enumeration or adopt a new branch-and-bound algorithm [24] to select the decomposition with the minimum fractional hypertree width. However, they only consider a single decomposition with the minimum width, while decompositions of identical width can exhibit dramatically different computational efficiency. ② *incomplete symmetry-breaking integration*: Integration of symmetry-breaking with tree decomposition has been explored only in subgraph counting [39], which only partially integrates symmetry-breaking with tree decomposition, focusing on rules compatible with aggregation pushdown. ③ *expensive materialization overhead*: After computing bag relations, TD approaches must join them to produce final results, which we term the global join. While subgraph counting approaches [39, 74] integrate the global join within bag processing via aggregation pushdown, subgraph matching approaches [2, 38] employ separate processing phases that incur high materialization costs to build complex data structures for all bags. Although existing work [38] proposed techniques to reduce memory usage for storing bag relations, it does not consider reducing the computational overhead for constructing such data structures. Furthermore, for bag attribute ordering, existing approaches rely on heuristics [2, 39, 74] or consider only intersection costs [38], ignoring that materialization overhead also depends on attribute ordering.

Main Contributions: To address these limitations, we present MDSE (Minimal Decomposition based Subgraph Enumeration) with four key contributions. First, we introduce minimal fractional hypertree decompositions (MinFHDs), whose bags cannot be replaced by smaller ones while preserving correctness, and develop an efficient enumeration algorithm that explores all high-quality minimum-width decompositions rather than settling for a single one, avoiding the scalability issues of brute-force methods. Second, we integrate symmetry-breaking with tree decomposition for subgraph enumeration. For a given set of symmetry-breaking rules, we show that subgraph matching can use all rules rather than only a subset. Since different rule sets vary in effectiveness, we select the one that maximizes filtering opportunities. Third, we propose MixJoin, a novel algorithm that embeds the global join within bag computation for subgraph matching. It reduces bag materialization cost ignored by prior work. We also develop a cost model that accounts for both intersection and materialization costs, enabling better attribute-ordering decisions. Fourth, we conduct extensive experiments on 101 pattern graphs and 8 large real-world datasets against state-of-the-art subgraph enumeration algorithms. MDSE consistently outperforms existing approaches, achieving up to 1-2 orders of magnitude speedup.

Organization: We give preliminaries and problem formulation in Section 2. We introduce the tree decomposition framework in Section 3. In Section 4, we propose minimal fractional hypertree decompositions and present our efficient enumeration algorithm. We develop the systematic symmetry-breaking integration framework in Section 5. We propose the MixJoin algorithm and our cost model in Section 6. Section 7 discusses related work. We conduct comprehensive experimental studies and report the results in Section 8, and conclude this paper in Section 9.



(a) pattern graph p (b) data graph G

Figure 1: An example of subgraph enumeration

2 PRELIMINARIES

Following the notation in [39], we model a simple undirected graph as $G = (V, E)$, where V and E are the sets of nodes and edges in G, respectively. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, and is an induced subgraph of G if E' contains all the edges in G such that both endpoints belong to V'. We call a graph a k-node graph (or k-graph) if it has k nodes. The number of nodes and the number of edges are denoted as $n = |V|$ and $m = |E|$.

Subgraph Isomorphism: Let $G = (V, E)$ be a data graph and $p = (V_p, E_p)$ be a pattern graph. A function $f : V_p \mapsto V$ is called a homomorphism of p if for each edge $(u, u') \in E_p$, we have $(f(u), f(u')) \in E$, and a homomorphism f of p is called a subgraph isomorphism of p if f is injective. That is, for any $u, u' \in V_p, u \neq u'$, $f(u) \neq f(u')$. A subgraph $G_f = (V_f, E_f)$ in a data graph G is an *iso-match* of a pattern graph p if f is a subgraph isomorphism.

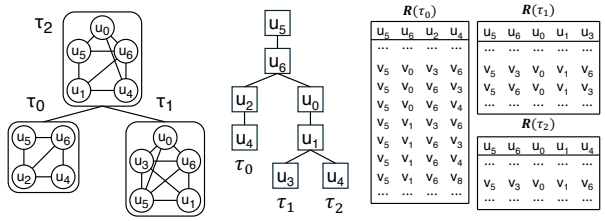
Automorphisms and Orbits: For a given graph $G = (V, E)$, an automorphism is a bijective function $\gamma : V \mapsto V$ such that $(v, v') \in E$ iff $(\gamma(v), \gamma(v')) \in E$. Here, an automorphism γ maps G to itself in a structure-preserving manner. The set of automorphisms, denoted as $\mathrm{Aut}(G)$, forms a group called the automorphism group of G. With $\mathrm{Aut}(G)$, two nodes $v, v' \in V$ are in an equivalence relation iff there exists an automorphism γ such that $\gamma(v) = v'$ (or $v \to v'$) for $\gamma \in \mathrm{Aut}(G)$. Such equivalence relations partition nodes into equivalence classes, where an equivalence class is called an automorphism orbit, and is denoted as ϑ.

Problem Statement: In this work, we study unlabeled subgraph enumeration. Given an unlabeled pattern graph p and an unlabeled data graph G, subgraph enumeration returns all distinct iso-matches of p in G. This differs from listing subgraph isomorphisms because there are multiple subgraph isomorphisms for the same iso-match due to automorphisms of p.

Example 2.1: Fig. 1 illustrates the subgraph enumeration problem. Given the pattern graph p and the data graph G, we identify two distinct iso-matches of p in G, highlighted by green and purple dashed lines respectively. For the green iso-match, there are two different subgraph isomorphisms: $f_1 = \{u_0 \mapsto v_0, u_1 \mapsto v_1, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5, u_6 \mapsto v_6\}$ and $f_2 = \{u_0 \mapsto v_1, u_1 \mapsto v_0, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5, u_6 \mapsto v_6\}$. These arise due

Algorithm 1: The tree decomposition framework

Input: pattern graph p, data graph G
Output: subgraph matches of p in G
1 $T \leftarrow$ select a tree decomposition with minimum width;
2 Select symmetry-breaking rules;
3 **foreach** *bag* $\tau_i \in T$ **do**
4 select a $\mathcal{JAO}$ of $V(\tau_i)$;
5 Compute and materialize $\mathbf{R}(\tau_i)$;
6 output $\bowtie_{\tau_i \in T} \mathbf{R}(\tau_i)$;



(a) T_1 (b) T_A (c) bag relations

Figure 2: FHD**, attribute tree and bag relations**

to pattern symmetry: p has an automorphism that swaps u_0 with u_1. Thus $\{u_0, u_1\}$ forms an automorphism orbit.

Tree decomposition is a powerful tool to decompose and process a pattern graph [16]. It is used for subgraph matching [2, 38], subgraph counting [39, 74] and theoretical approaches [15, 43]. The tree decomposition framework is summarized in Algorithm 1.

Tree decomposition (line 1). A hypergraph is defined as $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of hyperedges, where a hyperedge $e \in \mathcal{E}$ is a subset of $\mathcal{V}$. A pattern graph $p = (V_p, E_p)$ is a special hypergraph where every hyperedge only has two nodes. Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a tree decomposition is a tree $T = (V_T, E_T)$ where each tree node $\tau_i \in V_T$ maintains a bag $\mathcal{V}(\tau_i) \subseteq \mathcal{V}$ with which an induced subgraph of $\mathcal{H}$ can be constructed, denoted as $\mathcal{H}(\tau_i)$. The tree satisfies the following conditions: (1) Every node and edge in $\mathcal{H}$ is covered by at least one bag; (2) For any vertex $v \in \mathcal{V}$, bags containing v form a connected subtree (running intersection property). We use $V_S(\tau_i)$ to denote the nodes in bag τ_i that also appear in neighbors of τ_i.

A fractional hypertree decomposition (FHD) is (T, ρ) where ρ assigns each bag τ_i a fractional edge cover value $\rho(\tau_i)$ [19]. The width of (T, ρ) is $\max_{\tau_i \in T} \rho(\tau_i)$, and the fractional hypertree width $\mathsf{fhtw}(\mathcal{H})$ is the minimum width among all FHDs. For a pattern graph p, subgraph matching complexity is bounded by $O(|E|^{\mathsf{fhtw}(p)} + OUT)$ where OUT is the output size [2, 16]. Existing works select decompositions with minimum $\mathsf{fhtw}(p)$.

Example 2.2: Fig. 2(a) shows an FHD T_1 of the pattern graph p in Fig. 1(a). The decomposition T_1 consists of three bags: τ_0, τ_1, τ_2, each representing an induced subgraph of p. The decomposition satisfies all tree decomposition properties: every node and edge of the pattern graph is contained in at least one bag, and for each node, the bags containing that node form a connected subtree. For bag τ_0, the shared nodes u_4, u_5 and u_6 appear in multiple bags, yielding $V_S(\tau_0) = \{u_4, u_5, u_6\}$. The induced subgraph $p(\tau_0)$ over nodes $\{u_5, u_6, u_2, u_4\}$ contains 5 edges, as illustrated in Fig. 2(a). The fractional hypertree width of T_1 is 2.5, which is the minimum among all FHDs of p, and $\mathsf{fhtw}(p) = 2.5$.

Symmetry-breaking (line 2). Symmetry-breaking reduces redundant iso-matches when pattern graphs contain automorphisms [18, 49, 55, 56, 59, 67]. For an automorphism orbit $\vartheta = \{u_1, u_2, \ldots, u_k\}$, a symmetry-breaking rule $\theta = \{u_1 < u_2, \ldots, u_1 < u_k\}$ imposes ordering constraints. A subgraph isomorphism f is valid if $f(u_1) < f(u_i)$ for all $u_1 < u_i$ in θ. In Example 2.1, we can apply a symmetry-breaking rule $u_0 < u_1$, which filters the redundant mapping f_2.

Attribute order and attribute tree (line 4). A join attribute order ($\mathcal{JAO}$) defines a total order of pattern graph nodes. WCOJ processes subgraph matching by incrementally extending partial matches following $\mathcal{JAO}$. Emptyheaded, DISC and SCOPE select $\mathcal{JAO}$s by heuristics. ASDMatch proposes an attribute tree T_A where the $\mathcal{JAO}$ of each bag τ_i is represented as a root-to-leaf path, enabling shared computation among overlapping attributes. It selects an attribute tree by minimizing the cost for computing set intersections.

Join processing (lines 5-6). Tree decomposition approaches store data graph G in edge table $\mathbf{R}_g$ where each undirected edge (v, v') is stored as two tuples (v, v'), (v', v) and interpret pattern p as a self-join on $\mathbf{R}_g$. For each bag τ_i, it computes matches of the induced subgraph $p(\tau_i)$ and materializes them in relation $\mathbf{R}(\tau_i)$. After materializing all bag relations, it joins all $\mathbf{R}(\tau_i)$, which is an acyclic join. DISC [74] and SCOPE [39] use binary joins with aggregation pushdown, integrating this step into bag processing. Emptyheaded [2] uses Yannakakis' algorithm [72] with complexity $O(IN + OUT)$, where $IN = O(|E|^{\mathsf{fhtw}(p)})$ and OUT is the output size. ASDMatch [38] uses WCOJ for practical efficiency.

Example 2.3: Fig. 2(b) shows an attribute tree T_A constructed from the FHD T_1 in Fig. 2(a). Each root-to-leaf path in T_A represents a $\mathcal{JAO}$ for one of the bags. ASDMatch processes the three bag relations following their respective $\mathcal{JAO}$s while sharing computations for overlapping attribute sequences. For instance, the leftmost path u_5, u_6, u_2, u_4 serves as the $\mathcal{JAO}$ for bag τ_0. $\mathbf{R}(\tau_0)$ is computed by executing the join $\mathbf{R}_g(u_2, u_4) \bowtie \mathbf{R}_g(u_2, u_5) \bowtie \mathbf{R}_g(u_2, u_6) \bowtie \mathbf{R}_g(u_4, u_6) \bowtie \mathbf{R}_g(u_5, u_6)$ In the attribute tree T_A, nodes u_5, u_6 are shared among all three bags, while u_0, u_1 are shared between τ_0 and τ_1. By reusing intermediate results, the matches of $p(\{u_5, u_6\})$ are computed once and shared across all three bags, and the matches of $p(\{u_5, u_6, u_0, u_1\})$ are shared between τ_0 and τ_1. Using the data graph G from Fig. 1(b), some tuples of bag relations are shown in Fig. 2(c). For example, the tuple $[v_5, v_0, v_3, v_6]$ in $\mathbf{R}(\tau_0)$ represents an iso-match of the induced subgraph $p(\tau_0)$ in G. Finally, we compute the join query $\mathbf{R}(\tau_0) \bowtie \mathbf{R}(\tau_1) \bowtie \mathbf{R}(\tau_2)$ to obtain all subgraph matches of p in G.

Limitations of existing approaches. While tree decomposition is effective for labeled subgraph matching and subgraph counting, existing approaches have three key limitations for unlabeled subgraph enumeration. **Limitation 1: suboptimal decomposition selection.** Existing approaches compute only one decomposition with minimum fractional hypertree width [24]. However, decompositions with identical width can have vastly different computational costs. For instance, consider the pattern graph p in Fig. 1(a).

Algorithm 2: MDSE (p, G)

Input: pattern graph p, data graph G
Output: subgraph matches of p in G

1 $T_A^* \leftarrow \emptyset, \Theta^* \leftarrow \emptyset, minCost \leftarrow \infty$;
2 $\mathbb{T} \leftarrow$ MinFHDs of p with minimum width;
3 **foreach** $T \in \mathbb{T}$ **do**
4 $\quad \Theta \leftarrow$ SelectSymmetries(p, T);
5 $\quad T_A \leftarrow$ select an attribute tree for T, Θ using our cost model;
6 $\quad$ **if** cost$(T_A) < minCost$ **then**
7 $\quad\quad T_A^* \leftarrow T_A, \Theta^* \leftarrow \Theta, minCost \leftarrow$ cost(T_A);
8 MixJoin(p, T_A^*, Θ^*, G);

Table 1: Summary of tree-decomposition-based approaches

Algorithm	decomposition selection	symmetry breaking	global join	attribute order
Emptyheaded [2]	brute-force	✗	extra step	heuristic
DISC [74]	brute-force	✗	within bag	heuristic
SCOPE [39]	brute-force	some	within bag	heuristic
ASDMatch [38]	branch-&-bound	✗	extra step	intersection cost
MDSE	minimal elimination order	all	within bag	intersection and materialization cost

On top of the FHD T_1 in Example 2.2, there are two other tree decompositions T_2 in Fig. 3(a) and T_3 in Fig. 3(b) with the same width, and their processing time varies between 119 seconds and 1,079 seconds. Simply selecting one decomposition misses many potentially better decompositions. **Limitation 2: incomplete symmetry-breaking.** Most decomposition-based algorithms ignore symmetry-breaking entirely, leading to redundant enumeration. SCOPE [39] applies symmetry-breaking but only uses rules compatible with its aggregation pushdown technique. For the pattern graph p in Fig. 1(a), nodes u_0 and u_1 are symmetric. However, for T_2 in Fig. 3(a) and T_3 in Fig. 3(b), SCOPE cannot apply the symmetry-breaking rule $u_0 < u_1$. **Limitation 3: expensive materialization cost.** Existing matching algorithms process bag relations and global joins as separate phases (lines 5-6 in Algorithm 1). Each bag relation $\mathbf{R}(\tau_i)$ must be fully materialized with indexing structures before the global join. As shown in Example 2.3, this requires storing all tuples. The computation cost for materializing the tuples into indexing structures is overlooked in previous works.

3 AN OVERVIEW OF MDSE

To address the limitations identified in Section 2, we present MDSE (Minimal Decomposition-based Subgraph Enumeration). Algorithm 2 shows our framework, and Table 1 summarizes our improvements. Below we describe the technical challenges in overcoming each limitation and our solutions.

To overcome Limitation 1, we need to explore multiple tree decompositions. However, to our best knowledge, there is no efficient algorithm to enumerate multiple decompositions. Existing brute-force methods [2, 39, 74] enumerate all possible node subsets as potential bags and all combinations of bags to enumerate decompositions, which is computationally prohibitive. We introduce *minimal* fractional hypertree decompositions (MinFHDs), where no bag can be replaced by smaller bags while preserving correctness. We show that MinFHDs achieve faster processing time and can be characterized through elimination orders, which enables efficient enumeration. In Algorithm 2, line 2 enumerates all MinFHDs, and lines 3-7 evaluate each candidate to select the best one.

To overcome Limitation 2, we need to fully apply the rules in a selected symmetry-breaking rule set. Symmetry-breaking rules come from symmetric (i.e. structurally equivalent by automorphism) nodes in p. However, symmetric nodes of p may not remain symmetric in $p(\tau)$ for a bag τ, making some rules inapplicable in SCOPE [39]. In our enumeration setting, we develop a predicate pushdown mechanism that can use any complete symmetry-breaking rule set, regardless of whether its symmetric nodes remain symmetric in bags. In addition, different symmetry-breaking rule sets lead to different predicate distributions, and evaluating which rule set minimizes computation requires considering their filtering effects. Our scoring mechanism (line 4 in Algorithm 2) evaluates each rule set based on the computational cost reduction at each bag, selecting the optimal one.

To overcome Limitation 3, we need to reduce the computation cost for building indexing structures for bag relations. Existing approaches [2, 38] store each bag relation $\mathbf{R}(\tau_i)$ in a sorted trie data structure, whose irregular branching causes memory fragmentation and poor cache locality, making trie construction expensive. However, it is challenging to reduce the materialization while preserving the correctness and intersection efficiency of the algorithm. We propose the MixJoin algorithm with three techniques to reduce materialization overhead. First, we use *prefix attributes* that are processed like standard WCOJ, while only materializing the remaining attributes. This reduces trie levels without increasing intersection costs. Second, we identify a *containment condition* to completely avoid the materialization of the last bag to process. Third, we completely avoid building one-level tries. Our cost model (line 5 in Algorithm 2) evaluates both intersection and materialization cost to select the optimal attribute tree, and line 8 executes MixJoin with the selected configuration.

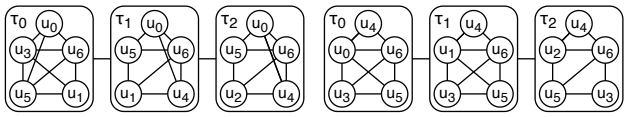
Remark. Although this paper focuses on unlabeled subgraph enumeration, our techniques are orthogonal to label-based filtering and can be integrated with labeled matching methods. (1) MinFHDs also apply in labeled settings because the decomposition structure is independent of labels, and compact bags still reduce intermediate results. (2) Labels may reduce but not eliminate pattern automorphisms, so our predicate pushdown can still exploit residual symmetries. (3) MixJoin and our cost model also improve join processing in labeled matching, since materialization overhead remains.

4 MINIMAL DECOMPOSITIONS

To explore multiple high-quality decompositions efficiently, we introduce minimal fractional hypertree decompositions (MinFHDs).

Definition 4.1: A minimal fractional hypertree decomposition (MinFHD) of a hypergraph $\mathcal{H}$ is (T, ρ) where T is the tree decomposition and ρ is the fractional edge cover function, and there does not exist another FHD (T', ρ'), such that there is a function $f : V_T \mapsto V_{T'}$, where $\forall \tau_i \in T, f(\tau_i) \subseteq \tau_i$, and $\exists \tau_j \in V_T, f(\tau_j) \subsetneq \tau_j$.

Example 4.1: The FHD T_1 in Fig. 2(a) is an MinFHD. Fig. 3(a) and Fig. 3(b) show two non-minimal FHDs with identical fractional



(a) T_2 (b) T_3

Figure 3: Different FHDs **of the pattern graph p in Fig. 1(a)**

hypertree width. In T_2, bag τ_0 contains an additional node u_0 compared to T_1, violating minimality. Similarly, T_3 is non-minimal since node u_3 can be removed from bag τ_2 in Fig. 3(b).

MinFHDs achieve computational advantages by ensuring each bag is fully utilized and cannot be replaced by smaller subsets. This compactness leads to smaller sub-queries and reduced intermediate results. Empirically, on the *yt* dataset, the three FHDs exhibit substantially different execution times: T_1 completes in 119s, T_2 requires 140s, and T_3 takes 1,079s, with minimal FHD T_1 achieving the best performance. We focus on finding an FHD that minimizes computational cost. This objective involves addressing two key challenges: finding FHDs with minimum width and ensuring minimality where bags cannot be further compacted. The following lemma establishes that an FHD satisfying both conditions exists—one that is both minimal and achieves minimum width.

Lemma 4.1: *Given a graph $p = (V_p, E_p)$, there exists a* MinFHD *(T, ρ) whose fractional hypertree width is* $\mathsf{fhtw}(p)$.

The proofs in this paper are provided in the technical report [1].

Finding FHDs with minimum width has been addressed in [24]. We focus on handling the minimality condition. Direct enumeration of all MinFHDs is computationally expensive since it requires exploring the exponential space of possible decompositions while ensuring minimality conditions. To address this challenge efficiently, we establish a connection between MinFHDs and *minimal triangulations* [25] in graph theory, and leverage the *elimination order* technique to systematically enumerate all MinFHDs.

Triangulation and clique tree. A *triangulation* p^+ of p is a graph obtained by adding edges to p, i.e., $p^+ = (V_p, E_p \cup F)$ where F is a set of fill-in edges such that p^+ is chordal. A graph is chordal if every induced cycle in the graph is a triangle. A triangulation p^+ is called *minimal* if no proper subset of its fill-in edges produces a triangulation of p.

A *clique tree* of a triangulated graph p^+_π is a tree where each node represents a maximal clique of p^+_π, and the tree satisfies the running intersection property: for any vertex $v \in V_p$, the set of clique tree nodes containing v forms a connected subtree. The chordal graph p^+_π has at most $|V_p|$ maximal cliques. A standard way to construct a tree decomposition of p is to triangulate p as p^+, and then build a clique tree from the maximal cliques of p^+. Because chordal graphs admit a clique tree whose nodes are the maximal cliques and satisfies the running intersection property, taking each maximal clique of p^+ as a bag and the edges of the clique tree yields a valid tree decomposition for p.

Example 4.2: Fig. 5(a) shows a minimal triangulation p^+_1 with fill-in edges $F = \{(u_0, u_1), (u_5, u_4)\}$ (dashed red). The resulting graph is chordal since no induced subgraph forms a cycle of length ≥ 4. Fig. 5(b) displays the clique tree where each tree node represents a maximal clique of p^+_1, corresponding to FHD T_1 in Fig. 2(a). In contrast, triangulation p^+_2 in Fig. 5(c) is non-minimal, corresponding to FHD T_2 in Fig. 3(a).

We show that the proposed MinFHDs can be found using minimal triangulations.

Lemma 4.2: *Given a pattern graph $p = (V_p, E_p)$ and an* FHD *(T, ρ) of p, let $p^+_T = (V_p, E_p \cup \{(v, v') \mid \exists \tau_i \in T, v, v' \in \tau_i, v \neq v'\})$. The* FHD *$(T, \rho)$ is an* MinFHD *if and only if p^+_T is a minimal triangulation of p.*

Having established the equivalence between MinFHDs and minimal triangulations, we now need a practical method to generate all minimal triangulations.

Elimination order. Given a pattern graph $p = (V_p, E_p)$, an *elimination order* π is a linear order of V_p. For simplicity, we use π_i to represent a node $v \in V_p$ as the i-th vertex in the linear order ($\pi(v) = i$). With a given linear order, a sequence of $|V_p| + 1$ graphs, $(p^0_\pi, p^1_\pi, \cdots, p^{|\pi|}_\pi)$, can be constructed for $p^i_\pi = (V^i_\pi, E^i_\pi)$. Here, $p^0_\pi = p$. $p^i_\pi = \mathsf{elim}(p^{i-1}_\pi, \pi_i)$ where $\mathsf{elim}(p, v)$ is an elimination operation that removes a vertex v from the input graph p, adds fill-in edges $D(p, v) = \{(v', v'') \mid v' \in N_p(v), v'' \in N_p(v), (v', v'') \notin E_p\}$ into p, and produces a new graph $p' = (V_p \setminus \{v\}, E_p \cup D(p, v))$. The last $p^{|\pi|}_\pi$ in the sequence is an empty graph.

Given an elimination order π, a triangulation p^+_π can be produced by the elimination process as $p^+_\pi = (V_p, E_p \cup \bigcup_{i=0}^{|\pi|-1} D(p^i_\pi, \pi_i))$ [25]. An elimination order π is minimal if there does not exist another elimination order π' such that $p^+_{\pi'} \subsetneq p^+_\pi$.

Example 4.3: Fig. 4 demonstrates an elimination order $u_2, u_3, u_0, u_1, u_4, u_5, u_6$ for the pattern graph p in Figure 1(a). First, we perform the elimination operation $\mathsf{elim}(p, u_2)$, which removes vertex u_2 from p and adds edges between all pairs of its neighbors that are not already connected. Since u_2's neighbors are $\{u_4, u_5, u_6\}$ and vertices u_4 and u_5 lack an edge, we add the fill-in edge $D(p_0, u_2) = \{(u_4, u_5)\}$, yielding $p_1 = (V_{p_0} \setminus \{u_2\}, E_{p_0} \cup \{(u_4, u_5)\})$. Next, when eliminating u_3, its neighbors in the remaining graph are $\{u_0, u_1, u_4, u_5, u_6\}$. Since u_0 and u_1 are not connected, we add the fill-in edge $D(p_2, u_3) = \{(u_0, u_1)\}$. The elimination process continues until we reach $p_7 = \emptyset$. This elimination order produces exactly the chordal graph p^+_1 shown in Figure 5(a), where $p^+_1 = (V_p, E_p \cup \{(u_4, u_5), (u_0, u_1)\})$. It is a minimal elimination order.

The following theorem guarantees that we can build all minimal triangulations as well as all MinFHDs using elimination orders.

Theorem 4.1: *Given a pattern graph $p = (V_p, E_p)$, p^+ is a minimal triangulation if and only if there exists a minimal elimination order of p such that $p^+ = p^+_\pi$ [48].*

Based on the established theoretical connections, we enumerate MinFHDs with minimum width $\mathsf{fhtw}(p)$ using elimination orders. We first compute $\mathsf{fhtw}(p)$ using [24], then enumerate all $|V_p|!$ permutations as elimination orders and use the clique trees of the corresponding triangulations as tree decompositions. We filter those that are not minimal or do not achieve minimum width. Due to automorphisms, there may exist decompositions T, T' whose bags induce isomorphic subgraphs, i.e., there is a bijection $f : V_T \mapsto V_{T'}$

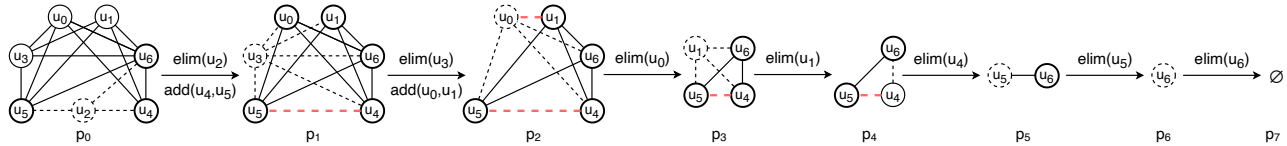
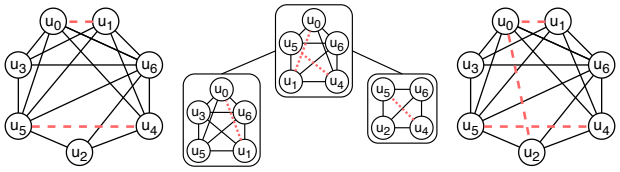


Figure 4: An elimination order $u_2, u_3, u_0, u_1, u_4, u_5, u_6$ for the pattern graph p in Fig. 1(a)



(a) p_1^+ (minimal) (b) The clique tree of p_1^+ (c) p_2^+ (not minimal)

Figure 5: Triangulations of p in Fig. 1(a) and a clique tree

where $\forall \tau_i \in V_T$, $p(\tau_i)$ is isomorphic to $p(f(\tau_i))$, although they may not contain exactly the same pattern graph nodes. Since the computations are identical for such decompositions, we only keep one representative. Although exponential in $|V_p|$, this is efficient since pattern graphs are typically small.

5 APPLYING SYMMETRY-BREAKING

Symmetry-breaking eliminates redundant matches when pattern graphs contain automorphisms. For any given pattern graph, multiple symmetry-breaking rule sets exist and can be efficiently enumerated using the recursive approach from [39]. However, SCOPE can only use a subset of rules from the selected set that are compatible with its aggregation pushdown technique. In contrast, our enumeration-based approach can utilize a complete rule set. We treat symmetry-breaking rules as join predicates and partition them into local predicates that can be pushed down to individual bags and global predicates applied during final result assembly. Since different rule sets exhibit varying effectiveness due to different predicate distributions, we employ a scoring mechanism to select the optimal rule set.

5.1 Predicate Pushdown on Bags

In unlabeled subgraph enumeration, there are two types of predicates, i.e., injectivity predicates and symmetry-breaking predicates. Injectivity predicates ensure that mappings of pattern graph nodes are distinct, as required by subgraph isomorphism. As established in Section 2, symmetry-breaking rules impose ordering constraints on automorphism orbits to eliminate redundant matches. For an automorphism orbit $\vartheta = \{u_1, u_2, \ldots, u_k\}$, a symmetry-breaking rule $\theta = \{u_1 < u_2, \ldots, u_1 < u_k\}$ ensures that mappings f satisfy $f(u_1) < f(u_i)$ for all $u_1 < u_i$ in θ. We define the complete predicate set $\mathcal{P}$ as the union of injectivity constraints and symmetry-breaking constraints:

$$\mathcal{P} = \{u_i \neq u_j \mid i \neq j, u_i, u_j \in V_p\} \cup \bigcup_{\theta \in \Theta} \{(u_i < u_j) \mid (u_i < u_j) \in \theta\} \tag{1}$$

where Θ is the set of symmetry-breaking rules. We denote the injectivity predicates as $\mathcal{P}^{inj} = \{u_i \neq u_j \mid i \neq j, u_i, u_j \in V_p\}$, which remain fixed regardless of Θ. Our focus is on efficiently handling the symmetry-breaking predicates $\mathcal{P}^{sym} = \bigcup_{\theta \in \Theta} \{(u_i < u_j) \mid (u_i < u_j) \in \theta\}$. We use $\phi_{i,j}$ to denote an individual predicate in $\mathcal{P}$ whose variables are u_i and u_j. In the following, we discuss how to use a symmetry-breaking rule set Θ.

With this unified predicate framework, subgraph enumeration can be expressed as a join over edge relations:

$$Q(p, \mathcal{P}) = \sigma_{\mathcal{P}} \left(\bowtie_{(u_i, u_j) \in E_p} \mathbf{R}(u_i, u_j) \right) \tag{2}$$

where $\sigma_{\mathcal{P}}$ is the selection operator that filters matches based on predicates in $\mathcal{P}$.

In the tree decomposition framework, this query can be rewritten as:

$$Q(p, \mathcal{P}) = \sigma_{\mathcal{P}} \left(\bowtie_{\tau_i \in T} \mathbf{R}(\tau_i) \right) \tag{3}$$

where $\mathbf{R}(\tau_i) = \bowtie_{(u_k, u_l) \in E_{p(\tau_i)}} \mathbf{R}(u_k, u_l)$ represents matches of subgraph τ_i. Applying predicates $\mathcal{P}$ as global filters after computing $\bowtie_{\tau_i \in T} \mathbf{R}(\tau_i)$ is inefficient, requiring materialization before filtering and producing exponentially large intermediate results. Instead, we employ predicate pushdown [37, 69] to apply predicates early, significantly reducing intermediate relation sizes and computation cost. We can partition the predicate set $\mathcal{P}$ into local predicates $\mathcal{P}^{local}(\tau_i)$ for each bag and global predicates $\mathcal{P}^{global}$ for T. Local predicates are pushed down to individual bags. Global predicates are those where no bag contains both variables u_i and u_j.

$$\mathcal{P}^{local}(\tau_i) = \{\phi_{j,k} \in \mathcal{P} \mid u_j, u_k \in \tau_i\} \tag{4}$$

$$\mathcal{P}^{global} = \mathcal{P} \setminus \bigcup_{\tau_i \in T} \mathcal{P}^{local}(\tau_i) \tag{5}$$

With the predicate pushdown, a bag τ_i computes a relation $\mathbf{R}_{\mathcal{P}}(\tau_i)$ that is smaller than $\mathbf{R}(\tau_i)$.

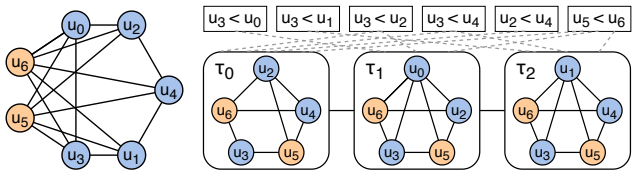
$$\mathbf{R}_{\mathcal{P}}(\tau_i) = \sigma_{\mathcal{P}^{local}(\tau_i)} \left(\bowtie_{(u_k, u_l) \in E_{p(\tau_i)}} \mathbf{R}(u_k, u_l) \right) \tag{6}$$

This allows us to rewrite the overall query as:

$$Q(p, \mathcal{P}) = \sigma_{\mathcal{P}^{global}} \left(\bowtie_{\tau_i \in T} \mathbf{R}_{\mathcal{P}}(\tau_i) \right) \tag{7}$$

Note that if a predicate $\phi_{i,j}$ can be pushed down to multiple bags, it is beneficial to apply the predicate to all such bags.

Example 5.1: For the pattern graph p in Fig. 6(a), where symmetric nodes are represented by the same color (blue and orange), we construct the FHD T shown in Fig. 6(b) with three bags: $\tau_0 = \{u_2, u_6, u_4, u_3, u_5\}$, $\tau_1 = \{u_0, u_6, u_2, u_3, u_5\}$, and $\tau_2 = \{u_1, u_6, u_4, u_3, u_5\}$. Consider the symmetry-breaking rule set $\Theta = \{u_3 < u_0, u_3 < u_1, u_3 < u_2, u_3 < u_4, u_2 < u_4, u_5 < u_6\}$. The figure illustrates how predicates in Θ can be pushed down to individual bags, with gray dashed lines connecting each predicate to the bags where it can be locally applied. For instance, the predicate $u_3 < u_0$ can be pushed down to τ_1 because both variables u_3 and u_0 are present in bag τ_1.



(a) pattern graph p (b) FHD and predicate pushdown

symmetry-breaking rule sets	score	Time(s)
$u_1 < u_0, u_1 < u_2, u_1 < u_3, u_1 < u_4, u_0 < u_2, u_5 < u_6$	$[1, 6(2 \times 3)]$	18.9
$u_1 < u_0, u_1 < u_2, u_1 < u_3, u_1 < u_4, u_3 < u_4, u_5 < u_6$	$[2, 4(1 \times 4)]$	12.3
$u_4 < u_0, u_4 < u_1, u_4 < u_2, u_4 < u_3, u_1 < u_2, u_5 < u_6$	$[3, 3(1 \times 3)]$	10.9
$u_4 < u_0, u_4 < u_1, u_4 < u_2, u_4 < u_3, u_0 < u_3, u_5 < u_6$	$[3, 6(2 \times 3)]$	10.0
$u_3 < u_0, u_3 < u_1, u_3 < u_2, u_3 < u_4, u_0 < u_1, u_5 < u_6$	$[3, 9(3 \times 3)]$	9.6
$u_3 < u_0, u_3 < u_1, u_3 < u_2, u_3 < u_4, u_2 < u_4, u_5 < u_6$	$[4, 9(3 \times 3)]$	7.3

(c) symmetry-breaking rule sets, scores and execution times

Figure 6: An example for pushing down and selecting *SymR*s

Remark. Although each match of p is enumerated exactly once, our predicate pushdown approach does not guarantee that each match of $p(\tau_i)$ is enumerated exactly once. A bag relation $\mathbf{R}(\tau_i)$ may lose some matches of $p(\tau_i)$ when symmetry-breaking predicates from $\mathcal{P}^{sym}$ are pushed down. For instance, in Fig. 6(b), u_3 and u_0 are not symmetric in τ_1, but we still apply $u_3 < u_0$. However, since these matches cannot be extended to matches of $Q(p, \mathcal{P})$, it is safe to prune them early during bag computation. The counting-based approach SCOPE in [39] can only use *SymR*s that are compatible with its aggregation pushdown technique. A symmetry-breaking rule set Θ can be used if for each bag τ and each automorphism orbit ϑ in Θ, either $\tau \cap \vartheta = \emptyset$ or $\vartheta \subseteq \tau$ and ϑ is an automorphism orbit of $p(\tau)$. For example, for the pattern graph p in Fig. 1(a) and FHD T_1 in Fig. 2(a), a predicate $u_0 < u_1$ can be used since u_0 and u_1 are symmetric in $p(\tau_1)$ and $p(\tau_2)$, and are not present in τ_0. In Fig. 6, none of the predicates $u_i < u_j$ in the table can be used by SCOPE due to these constraints. We do not require aggregation pushdown and can utilize any symmetry-breaking rule set.

5.2 Selecting Symmetry-breaking Rules

Algorithm 3 presents our scoring mechanism for selecting the optimal symmetry-breaking rule set Θ^* from all candidate rule sets. It selects Θ^* that maximizes the number of local symmetry-breaking predicates for the most computationally expensive bags, thereby minimizing the computational overhead during query processing.

The algorithm operates in several key phases. First, we establish a canonical ordering of bags (line 1) by sorting them according to: (1) fractional hypertree width in descending order, as bags with larger width typically dominate computational cost; (2) for bags with identical width, the number of edges in the induced subgraph in ascending order, since fewer edges indicate more intermediate results where early filtering is more beneficial. Next, we group bags by graph isomorphisms to account for structural equivalence (line 3). For each candidate symmetry-breaking rule set Θ_i, we extract all binary comparison predicates $\mathcal{P}_i$ by flattening the partial orders within each symmetry-breaking rule $\theta \in \Theta_i$ (line 6), then compute the local predicate count for each bag τ_j, measuring how many predicates can be pushed down locally (lines 7-8). Within each group, local predicate counts are compared using a multiplicative scoring function to ensure predicates are distributed evenly across

Algorithm 3: SelectSymmetries (p, T)

Input: pattern graph p, tree decomposition T with bags $\{\tau_1, \ldots, \tau_k\}$
Output: selected symmetry-breaking rule set Θ

1 $\Theta_1, \ldots, \Theta_n \leftarrow$ candidate symmetry-breaking rule sets of p;
2 $\tau_{sorted} \leftarrow$ SortBags($\{\tau_1, \ldots, \tau_k\}$);
3 $groups \leftarrow$ GroupByGraphIsomorphism(τ_{sorted});
4 $bestScore \leftarrow (0, 0, \ldots, 0)$, $\Theta^* \leftarrow \emptyset$;
5 **foreach** $\Theta_i \in \{\Theta_1, \ldots, \Theta_n\}$ **do**
6 $\quad \mathcal{P}_i \leftarrow \bigcup_{\theta \in \Theta_i} \{(u_j < u_k) : (u_j < u_k) \in \theta\}$;
7 $\quad$ **foreach** $\tau_j \in \tau_{sorted}$ **do**
8 $\quad\quad localCount_j \leftarrow |\{(u_k < u_l) \in \mathcal{P}_i : u_k, u_l \in \tau_j\}|$;
9 $\quad score_i \leftarrow \emptyset$;
10 $\quad$ **foreach** $group \in groups$ **do**
11 $\quad\quad groupCounts \leftarrow \{localCount_j : \tau_j \in group\}$;
12 $\quad\quad groupProduct \leftarrow \prod_{c \in groupCounts} c$;
13 $\quad\quad$ append $groupProduct$ to $score_i$;
14 $\quad$ **if** $score_i > bestScore$ **then**
15 $\quad\quad bestScore \leftarrow score_i$;
16 $\quad\quad \Theta^* \leftarrow \Theta_i$;
17 **return** Θ^*;

structurally equivalent bags, providing balanced computation reduction (lines 11-13). We compare score vectors in lexicographical order and prefer Θ_i that achieves higher scores for computationally challenging groups (lines 14-16).

Example 5.2: Continuing from Example 5.1, Fig. 6(c) demonstrates our algorithm's effectiveness. We sort the bags by computational complexity as τ_0, τ_1, τ_2, where $p(\tau_1)$ and $p(\tau_2)$ induce isomorphic subgraphs and are grouped together during scoring. In Fig. 6(c), we evaluate six candidate symmetry-breaking rule sets with score vectors in the format [localCount(τ_0), groupProduct(τ_1, τ_2)], using the wiki-vote data graph. For instance, the notation $[1, 6(2 \times 3)]$ indicates that τ_0 can push down 1 predicate locally while τ_1 and τ_2 can push down 2 and 3 predicates respectively, yielding a group score of $2 \times 3 = 6$. The experimental results demonstrate a strong correlation between higher scores and superior performance. The optimal rule set with score $[4, 9(3\times3)]$ achieves the fastest execution time (7.3s), while the worst-scoring rule set $[1, 6(2 \times 3)]$ takes 18.9s, yielding a 2.6× speedup. This validates our scoring mechanism: higher scores indicate more effective predicate pushdown, leading to reduced intermediate result sizes and faster query execution.

6 QUERY PROCESSING AND PLANNING

Previous methods [2, 38] process the global join $\bowtie_{\tau_i \in T} \mathbf{R}_{\mathcal{P}}(\tau_i)$ separately after computing bag relations $\mathbf{R}_{\mathcal{P}}(\tau_i)$, requiring expensive materialization using complex data structures. Following ADMatch [38], each bag relation $\mathbf{R}_{\mathcal{P}}(\tau_i)$ is stored in a sorted trie to support efficient global joins. A sorted trie is a tree where each level corresponds to an attribute in τ_i, with each node storing multiple values representing possible extensions from partial tuples encoded by the root-to-node path. The irregular branching structure causes memory fragmentation and poor cache locality, making trie construction expensive. To reduce the materialization overhead, we propose the MixJoin algorithm and a comprehensive cost model.

Algorithm 4: MixJoin(p, T_A, Θ, G)

Input: pattern graph p, the attribute tree T_A for a TD T with bags $\{\tau_1, \ldots, \tau_k\}$, symmetry-breaking rules Θ, data graph G
Output: Unique matches of p in G

1 $\mathcal{P} \leftarrow$ predicates by Θ and T; Pushdown $\mathcal{P}$ on bags;
2 let path_S be the rightmost path of shared attributes of T_A;
3 **foreach** $i \in [1,k]$ **do** $\text{prefix}_i \leftarrow \text{path}_i \cap \text{path}_S$;
4 **foreach** $i \in [1,k], \text{prefix}_i = \emptyset$ **do**
5 | Compute and Materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$
6 let $\pi_1, \cdots, \pi_l$ be the order of path_S;
7 TJoin(π_1, $V(G)$, $\emptyset$, *1*);
8 **Procedure** TJoin(π_c, I_c, f, c)
9 | **while** $I_c \neq \emptyset$ **do**
10 | | $v \leftarrow \text{Inext}(I_c)$; $I_c \leftarrow I_c \setminus \{v\}$;
11 | | $f \leftarrow f \cup \{\pi_c \mapsto v\}$;
12 | | **foreach** $i \in [1,k], \text{prefix}_i = [\pi_1, \cdots, \pi_c]$ **do**
13 | | | **if** $i \neq k$ **then**
14 | | | | Compute and Materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$;
15 | | | **else if** $\bigcup_{\tau_i \in T} V_S(\tau_i) \subseteq \tau_k$ **then**
16 | | | | **foreach** $f' \in \mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$ **do**
17 | | | | | **foreach** $f'' \in f' \bowtie_{\tau_j \in T, j<k} \mathbf{R}_{\mathcal{P}}(\widehat{\tau_j})$ **do**
18 | | | | | | output $f \cup f''$;
19 | | | **else**
20 | | | | Compute and Materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$;
21 | | | | **foreach** $f' \in \bowtie_{\tau_j \in T} \mathbf{R}_{\mathcal{P}}(\widehat{\tau_j})$ **do**
22 | | | | | output $f \cup f'$;
23 | | **if** $c < l$ **then**
24 | | | $I_{c+1} \leftarrow \text{Intersect}(\pi_{c+1}, f)$;
25 | | | TJoin(π_{c+1}, I_{c+1}, f, $c+1$);
26 | | $f \leftarrow f \setminus \{\pi_c \mapsto v\}$;

6.1 Mixing Global and Local Joins

Algorithm 4 presents MixJoin to enumerate a pattern graph $p = (V_p, E_p)$ over a data graph $G = (V, E)$. It reduces materialization overhead while preserving intersection operations unchanged. We propose three techniques. First, we use some attributes at the front of the *JAO* of each bag as prefix attributes (prefix) and the remainder as materialized attributes. Prefix attributes are processed similarly as WCOJ for a single query, while materialized attributes follow the decomposition-based approach. Previous works [38, 39, 74] studied prefix usage but focused on reducing memory usage and their approaches increase intersection cost. We focus on reducing materialization computation while keeping intersection operations identical to cases without prefix. Second, we identify a containment condition allowing processing the global join with the sub-query of the last bag, avoiding materializing its sorted trie. Third, we avoid materialization for bags with only one materialized attribute.

We use the notation $\widehat{\tau_i} = \tau_i \setminus \text{prefix}_i$ to denote materialized attributes of bag τ_i that require trie storage after processing prefix attributes. Correspondingly, $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$ represents iso-matches for the materialized portion of bag τ_i.

Prefix Selection. Suppose the processing order of bags in the attribute tree is $\tau_1, \cdots, \tau_k$. As discussed in [38], for a bag τ_i, the materialized attributes are processed for every match of $\bigcup_{j \leq i} \text{prefix}_j$. If $\bigcup_{j<i} \text{prefix}_j \subseteq \text{prefix}_i$, then the materialized attributes are processed for every match of prefix_i, ensuring that the intersections to compute τ_i remain identical to the case without prefix. To find such prefixes, our algorithm identifies the shared attributes in the attribute tree and uses its rightmost branch, denoted as path_S (line 2), to set the prefixes of each bag. Specifically, we compute $\text{prefix}_i = \text{path}_i \cap \text{path}_S$ for each bag τ_i (line 3). This construction guarantees that $\bigcup_{j<i} \text{prefix}_j \subseteq \text{prefix}_i$, preserving intersection behavior while maximizing the prefix usage. For bags where $\text{prefix}_i = \emptyset$ (lines 4-5), we immediately compute and materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$ since no prefix optimization applies.

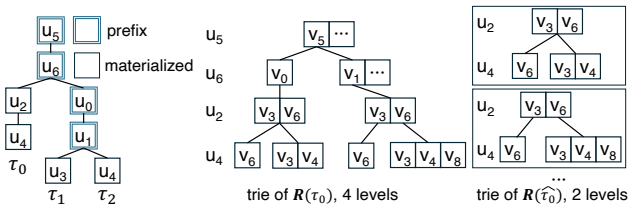


Figure 7: An example for prefix selection and MixJoin

The core computation is handled by the recursive TJoin procedure, which traverses the shared attribute path path_S in order $\pi_1, \cdots, \pi_l$ (line 6). For each attribute π_c and each candidate value v, the procedure extends the current partial mapping f and processes bags whose prefix matches the current path $[\pi_1, \cdots, \pi_c]$ (lines 12-22). When reaching deeper levels ($c < l$), the procedure recursively continues with the next attribute π_{c+1} and its corresponding intersection results (lines 23-25). In lines 12-14, for a bag τ_i that is not the last one, we materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$, thus reducing $|\text{prefix}_i|$ levels of trie materialization overhead.

Example 6.1: Fig. 7 illustrates prefix selection using attribute tree T_A from Fig. 2(b). The rightmost shared path $\text{path}_S = \{u_5, u_6, u_0, u_1\}$ determines prefix assignments: $\text{prefix}_0 = \{u_5, u_6\}$, $\text{prefix}_1 = \{u_5, u_6, u_0, u_1\}$, $\text{prefix}_2 = \{u_5, u_6, u_0, u_1\}$. For each prefix match $\{u_5, u_6\}$ in bag τ_0, we compute partial relation $\mathbf{R}(\widehat{\tau_0})$ and materialize it as a two-level sorted trie. Fig. 7 compares trie structures for $\mathbf{R}(\tau_0)$ and $\mathbf{R}(\widehat{\tau_0})$ using the data graph in Fig. 1(b) and the results in Fig. 2(c). The original trie requires 4 levels, while the optimized trie needs only 2 levels. This approach reduces materialization cost by limiting trie construction to non-prefix attributes.

Avoiding materialization for τ_k. The global join $\bowtie_{\tau_i \in T} \mathbf{R}_{\mathcal{P}}(\tau_i)$ requires joining on $\bigcup_{\tau_i \in T} V_S(\tau_i)$, i.e., all attributes that appear in at least two bags, and traversing the remaining attributes. When the final bag τ_k contains all join attributes ($\bigcup_{\tau_i \in T} V_S(\tau_i) \subseteq \tau_k$), we can eliminate the materialization of $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$ entirely. When processing bag τ_k with a complete prefix match, we check the containment condition (line 15). If satisfied, we enumerate each tuple f' in $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$ on-the-fly (line 16) and immediately join it with previously computed bag relations, outputting final results directly (lines 17-18). This streaming approach avoids materializing $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$ entirely, eliminating the expensive trie construction step. When the containment condition fails, we materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_k})$ (line 20) and perform the global join separately. With prefix, part of the global join $\bowtie_{\tau_i \in T} \mathbf{R}_{\mathcal{P}}(\tau_i)$ is processed during bag computations, requiring joins only on attributes not in prefix. For unlabeled subgraph enumeration with typically small pattern graphs, our empirical

analysis shows that 98 out of 101 queries satisfy the containment condition, making this optimization highly effective in practice.

Optimization for one-level trie. When $|\widehat{\tau_i}| = 1$, the materialized relation contains only one attribute, and we can reuse the intersection result I_c of that attribute without constructing an explicit trie. Since we store the data graph G using compressed sparse rows (CSR) format with sorted consecutive arrays for neighbor lists, the intersection operation naturally produces a sorted array I_c. When $|\widehat{\tau_i}| = 1$, the relation $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$ represents exactly those data graph nodes that can match the single remaining attribute given the current match of prefix_i, corresponding directly to I_c. This optimization eliminates 1-level trie construction.

Example 6.2: Continuing from Example 6.1, we demonstrate the one-level trie optimization for bag τ_1 and the containment condition optimization for bag τ_2. Bag τ_1 has only one materialized attribute. In Algorithm 4, when we have matched $\{u_5, u_6, u_0, u_1\}$ and process τ_1 (line 14), we compute I_c for candidate matches of u_3, which is reused as the one-level trie for τ_1. The global join $\mathbf{R}_{\mathcal{P}}(\tau_0) \bowtie \mathbf{R}_{\mathcal{P}}(\tau_1) \bowtie \mathbf{R}_{\mathcal{P}}(\tau_2)$ requires joining on $V_S(\tau_0) \cup V_S(\tau_1) \cup V_S(\tau_2) = \{u_0, u_1, u_4, u_5, u_6\}$. Since these attributes are contained within final bag τ_2, the containment condition $\bigcup_{\tau_i \in T} V_S(\tau_i) \subseteq \tau_2$ is satisfied, enabling streaming optimization. When processing bag τ_2 with prefix $\text{prefix}_2 = \{u_5, u_6, u_0, u_1\}$, partial mapping f covers these attributes while the remaining attribute $\widehat{\tau_2} = \{u_4\}$ needs to be processed. Without materializing $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_2})$, we enumerate matches f' for $\{u_4\}$ on-the-fly and immediately join with precomputed relations $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_0})$ and $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_1})$ to get matches of the pattern graph p extending the match of prefix_2. For each match of prefix_2, we generate new $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_0})$ and $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_1})$ and process the global join within τ_2. In this way, we enumerate all matches incrementally.

6.2 The Cost Model

The MixJoin algorithm enables various attribute trees with different prefix selections and materialization strategies. To select the optimal attribute tree among all candidates, we develop a comprehensive cost model that considers both intersection operations and materialization overhead. We first introduce the intersection cost for an attribute in a *JAO*, which differs from [38, 46] since we also consider symmetry-breaking effects. Given a partial match p_{k-1} induced by $\pi[1..k-1]$ for some $k > 1$, to match π_k by extending $\pi[1..k-1]$ to $\pi[1..k]$, for each $(\pi_l, \pi_k) \in E_p$, where $l < k$, we intersect the neighbors of $f(\pi_l)$. The intersection operation can be represented as $\cap_{(\pi_l,\pi_k)\in E_p} N_G(f(\pi_l))$ where $N_G(f(\pi_l))$ are the neighbors of the data graph G. The neighbor list size is estimated as the average degree of G if there are no symmetry-breaking constraints. When symmetry-breaking constraints are applied, the neighbor list size is reduced due to the ordering restrictions imposed by predicates in $\mathcal{P}$. Specifically, if there exists an ordering constraint between π_l and π_k, the symmetry-breaking predicates will filter out approximately half of the candidate neighbors. Therefore, we estimate the neighbor list size for each $l < k, (\pi_l, \pi_k) \in E_p$ as:

$$\mu_{\pi_k}^{\pi_l} = \begin{cases} |E(G)|/|V(G)| & \text{if } (\pi_l < \pi_k) \in \mathcal{P} \text{ or } (\pi_k < \pi_l) \in \mathcal{P}, \\ 2 \times |E(G)|/|V(G)| & \text{otherwise.} \end{cases} \tag{8}$$

Let $\mathbb{R}_{k-1}$ be a relation storing all partial iso-matches for $\pi[1..k-1]$ that satisfy the predicates $\mathcal{P}$ involving only variables $\{\pi_1, \ldots, \pi_{k-1}\}$, and $\mu(\mathbb{R}_{k-1})$ be the estimated cardinality of $\mathbb{R}_{k-1}$. Then the intersections for π_k are computed $\mu(\mathbb{R}_{k-1})$ times. The cost for computing each intersection is estimated as the total size of the sets. By multiplying the number of intersections, the cost is as follows.

$$\text{i-cost}(\pi_k) = \mu(\mathbb{R}_{k-1}) \sum_{l<k,(\pi_l,\pi_k)\in E_p} \mu_{\pi_k}^{\pi_l} \tag{9}$$

The intersection cost for an attribute tree $T_A = (V_A, E_A)$ is defined as the sum of individual costs:

$$\text{i-cost}(T_A) = \sum_{u \in V_A} \text{i-cost}(u) \tag{10}$$

The materialization cost. The materialization cost captures the overhead of building sorted tries to store intermediate relations. Different *JAOs* lead to different prefix selections prefix_i, resulting in varying materialization costs. A bag τ_i that is not the final bag (or the final bag when the containment condition fails) needs to materialize $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$. For each match of prefix_i, the algorithm materializes one instance of $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$. By summing over all matches of prefix_i, the total number of tuples across all materialized relations $\mathbf{R}_{\mathcal{P}}(\widehat{\tau_i})$ equals $\mu(\mathbf{R}_{\mathcal{P}}(\tau_i))$. The materialization cost depends on the number of trie levels $|\widehat{\tau_i}|$. We have shown that when $|\widehat{\tau_i}| = 1$, the materialization can be saved by reusing intersection results. For multi-level tries ($|\widehat{\tau_i}| > 1$), explicit materialization becomes necessary.

Therefore, the materialization cost is defined as:

$$\text{m-cost}(\tau_i) = \begin{cases} 0 & \text{if } (\tau_i = \tau_k \wedge \bigcup_{\tau_j \in T} V_S(\tau_j) \subseteq \tau_k) \vee |\widehat{\tau_i}| = 1, \\ \mu(\mathbf{R}_{\mathcal{P}}(\tau_i)) \times |\widehat{\tau_i}| & \text{otherwise.} \end{cases} \tag{11}$$

The total cost of processing an attribute tree combines both intersection and materialization costs:

$$\text{cost}(T_A) = \text{i-cost}(T_A) + \alpha \sum_{\tau_i \in T} \text{m-cost}(\tau_i) \tag{12}$$

We set the coefficient $\alpha = 20$ based on empirical evaluation and use the subgraph cardinality estimation algorithm in [57].

Example 6.3: We demonstrate cost calculation using the attribute tree T_A from Figure 7. Consider computing intersection cost for attribute u_3 during enumeration. For each partial match f of $p(\{u_5, u_6, u_0, u_1\})$, we compute candidate data graph nodes that can extend the match to include u_3 by intersecting neighbor sets $N_G(f(u_5))$, $N_G(f(u_6))$, $N_G(f(u_0))$, and $N_G(f(u_1))$ to obtain intersection result I_c in Algorithm 4. The number of such intersection operations equals $\mu(\mathbb{R}(u_5, u_6, u_0, u_1))$, which represents the estimated cardinality of $p(\{u_5, u_6, u_0, u_1\})$ after applying symmetry-breaking constraint $u_0 < u_1$. Each neighbor list size $|N_G(f(u_i))|$ for $i \in \{0, 1, 5, 6\}$ is estimated as $\frac{2|E_G|}{|V_G|}$ based on average degree in the data graph. Therefore, intersection cost for u_3 is computed as $\text{i-cost}(u_3) = \mu(\mathbb{R}(u_5, u_6, u_0, u_1)) \times \frac{8|E_G|}{|V_G|}$. Intersection costs of other attributes follow similar computations according to Equation (9).

For materialization cost analysis, bag τ_1 materializes only the single attribute u_3, yielding $\text{m-cost}(\tau_1) = 0$. Bag τ_2 serves as the final bag containing all shared attributes, enabling streaming optimization, so $\text{m-cost}(\tau_2) = 0$. However, bag τ_0 requires explicit

materialization since $\widehat{\tau_0} = \{u_2, u_4\}$ with $|\widehat{\tau_0}| = 2 > 1$, resulting in m-cost$(\tau_0) = 2 \times \mu(\mathbf{R}_{\mathcal{P}}(\tau_0))$.

Query Planning. With the cost model established, we select the optimal attribute tree from all valid candidates. The search space is large as we must consider final bag choice, *JAO* for each bag, and sharing opportunities among *JAOs*. We generate attribute trees from a restricted candidate set. First, we select a final bag from candidates that contain all shared attributes $\bigcup_{\tau_i \in T} V_S(\tau_i)$; if no such bag exists, we choose bags with largest $|V_S(\tau_i)|$. For each final bag choice, we construct the shared upper portion of the attribute tree (attribute nodes with ≥ 2 children) by combining permutations of each $V_S(\tau_i)$ and merging common prefixes. Remaining attributes within each bag are independent of other bags, with ordering following [7]. This process generates high-quality execution plans efficiently.

7 RELATED WORK

Subgraph Matching and Graph Mining Systems. Extensive research exists for subgraph enumeration on unlabeled graphs, spanning single-machine approaches [18, 34, 59] and distributed systems [35, 36, 49, 51, 71]. LIGHT [59] represents the state-of-the-art single-machine algorithm, employing lazy materialization to defer pattern node matching and reduce set intersection overhead. Among distributed approaches, some works decompose patterns into sub-structures before assembling their matches [35, 36, 49, 51]. HuGE [71] expresses queries as multi-way joins over clique and star units, supporting flexible pushing/pulling modes to balance memory and computational efficiency. Graph mining systems [4, 8–12, 20, 26, 27, 40, 55, 56, 63] extend to applications based on subgraph enumeration, such as subgraph counting and frequent subgraph mining. GraphSet [55] exemplifies this approach, leveraging set properties to eliminate control flow overhead. There is also work studying subgraph matching for labeled graphs. Most follow the classic filtering-ordering-enumerating framework [60, 76]. Research efforts have been devoted to pre-computing and filtering the search space [5, 6, 13, 22, 23, 32, 33, 61, 78], selecting good matching orders [6, 22, 23, 32, 33, 46, 53, 61], pruning invalid search branches [3, 22, 42], reusing previous matching results [28, 32, 33, 41, 42], and developing new enumeration algorithms [30, 38, 41, 77]. Recent surveys and experimental studies can be found in [29, 60, 76].

Tree decomposition and Query Processing. Tree decomposition has been widely used for constraint satisfaction problems [17], cyclic conjunctive queries [16] and their extensions [66]. The state-of-the-art approach to computing optimal FHD is BB4FHD [24]. For applying tree decompositions to query processing, CacheTrieJoin [31] uses tree decomposition to accelerate WCOJ on input queries. EmptyHeaded [2] follows a classic two-phase approach that computes bags first and then joins materialized views of bags. ADJ [73] studies using the two-phase approach in distributed systems. DISC [74] and SCOPE [39] study subgraph counting by converting the count of p to a linear combination of the counts of other subpatterns and processing each count by tree decomposition. ASDMatch [38] optimizes attribute ordering and proposes an adaptive approach to improve efficiency while bounding memory usage.

Table 2: Data graphs

Category	Dataset	Name	$\|V\|$	$\|E\|$	d
Web	Wiki-vote [70]	*wv*	7,115	100,762	28.3
Collaboration	Mico [14]	*mc*	96,638	1,080,156	22.4
Social	Youtube [70]	*yt*	1,134,890	2,987,624	5.3
Web	eu2005 [52]	*eu*	862,664	16,138,468	37.4
Citation	US Patents [70]	*up*	3,774,768	16,518,947	8.8
Social	Livejournal [70]	*lj*	4,846,609	42,851,237	17.7
Social	Orkut [70]	*ok*	3,072,441	117,185,083	76.2
Social	Friendster [70]	*fs*	65,608,366	1,806,067,135	55.1

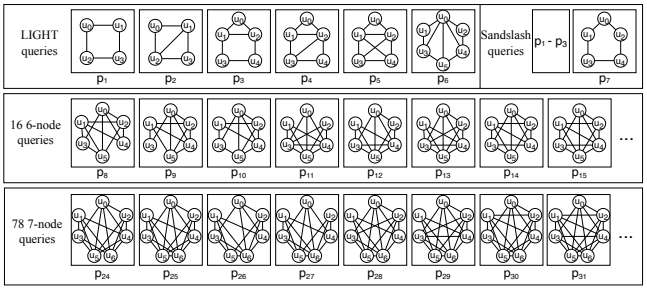


Figure 8: 101 pattern graphs

8 EXPERIMENTS

We conduct comprehensive experiments to evaluate the effectiveness of our MDSE algorithm.

8.1 Experimental Setup

Algorithms: We compare MDSE against four state-of-the-art baselines for subgraph enumeration. LIGHT [59] represents the state-of-the-art single-machine algorithm, employing a lazy materialization technique that defers pattern node matching until necessary to reduce computational overhead from set intersections. GraphSet [55] is a graph mining system. It incorporates sophisticated optimization techniques including a cost model for attribute ordering and symmetry-breaking selection inherited from GraphPi [56] and the elimination of redundant control flows via set properties. It does not use query decomposition. GraphZero [44] is a graph mining system that uses symmetry-breaking and generalizes orientation optimization to arbitrary patterns. Sandslash [10] is a graph mining system that automatically selects exploration order and data representations given only a pattern specification.

Datasets: We evaluate our approach on 8 large-scale real-world graphs commonly used in previous subgraph enumeration studies. Table 2 presents detailed statistics, including node counts, edge counts, and average degrees. Our dataset collection spans diverse graph types. The majority of datasets (*wv*, *yt*, *up*, *lj*, *ok*, *fs*) are sourced from [70], while *mc* and *eu* are obtained from [14] and [52] respectively. Following standard preprocessing practices, we treat all edges as undirected and remove isolated nodes and self-loops from the original dataset graphs.

Pattern graphs: Our evaluation encompasses 101 pattern graphs ranging from 4 to 7 nodes, as illustrated in Fig. 8. Due to their specialized implementations, LIGHT and Sandslash support only limited pattern sets. LIGHT handles p_1–p_6, while Sandslash supports p_1–p_3 and p_7. We omit clique patterns from LIGHT since tree

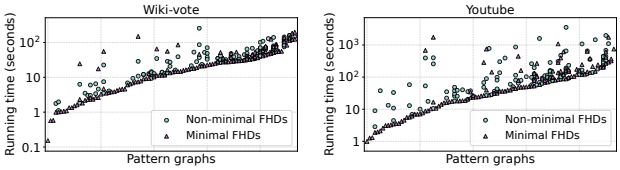


Figure 9: Comparing minimal and non-minimal FHDs

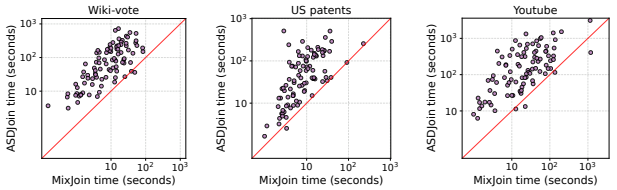


Figure 10: Effect of MixJoin

Table 3: Performance of MinFHD enumeration (time in ms)

# nodes	6	6	6	7	7	7	7	7
# edges	11	12	13	15	16	17	18	19
pattern count	9	5	2	40	21	10	5	2
Brute-force [39]	24.1	15.7	0.9	1184	661	90.6	15.4	1.0
MDSE	2.2	2.1	1.5	11.1	7.1	5.3	5.2	2.2

decompositions provide limited advantages for these structurally simple cases. LIGHT, Sandslash and our MDSE implement ad-hoc optimizations for these specific patterns. GraphSet and GraphZero support arbitrary pattern graphs, though existing evaluation [55] covered only 6 patterns for subgraph enumeration. To ensure a comprehensive comparison, we conduct extensive evaluation against GraphSet and GraphZero using 94 patterns without ad-hoc optimizations. These patterns include 16 six-node patterns with 11-13 edges and 78 seven-node patterns with 15-19 edges. We exclude patterns with very sparse edges, which generate prohibitively large results, and cliques or cliques missing a single edge, as these algorithms handle these cases with similar efficiency.

Environment: All experiments are conducted on a computing platform running CentOS 9.4, equipped with an Intel Xeon E7-8891 v3 processor featuring 80 cores and 256 GB of memory. All algorithms are implemented in C++ and compiled using g++ 11.4.1 with aggressive optimization (-O3) to achieve optimal performance. For parallel execution, all algorithms use 64 threads.

8.2 MDSE Testing

MinFHD **enumeration**. We compare our elimination order-based MinFHD enumeration against the brute-force method with pruning optimizations from [39]. Our method achieves 194.8× average speedup across all 94 patterns. Critically, while the brute-force approach reaches 9,182.8 ms maximum time, which exceeds the total execution time of many actual enumeration tasks, our approach achieves a maximum time of only 35.5 ms, maintaining negligible decomposition overhead. Table 3 shows detailed MinFHD enumeration results grouped by pattern sizes. Our approach achieves 14.7× speedup for 6-node patterns (16 instances) with average times of 2.1 ms vs. 18.6 ms, and 231.8× speedup for 7-node patterns (78 instances) with average times of 8.7 ms vs. 798.2 ms. For the two cases where brute-force is faster (6-node 13-edge and 7-node 19-edge patterns), these patterns contain large non-decomposable clique components. The brute-force method includes optimizations that detect such cliques and directly solve the remaining smaller subproblem, bypassing much of the enumeration. In contrast, our approach still performs elimination order enumeration and triangulation computation for these patterns. However, these cases are acceptable, as the performance difference is small, and they are simple to compute.

Table 4: Average Speedup of optimal *SymR*s vs. alternatives

Approaches	**Wiki-vote**		**Youtube**		**Patents**	
	$k=6$	$k=7$	$k=6$	$k=7$	$k=6$	$k=7$
no *SymR*s	6.38×	8.72×	4.84×	5.83×	4.02×	6.75×
partial *SymR*s	1.30×	1.96×	1.89×	3.43×	1.55×	2.66×
slowest full *SymR*s	1.49×	1.69×	3.00×	1.81×	1.27×	1.50×

Efficiency of MinFHDs. Fig. 9 compares minimal and non-minimal FHDs for 94 patterns on two real-world datasets. For each pattern, we evaluate all minimum-width decompositions and classify them as minimal or non-minimal; 37 patterns have only one decomposition and 57 have multiple decompositions. Patterns are ordered by their fastest execution time. Minimal decompositions (triangles) are generally faster than non-minimal ones (circles). This trend is particularly evident in the lower regions of both plots, where the fastest decompositions are mostly minimal. Quantitatively, MinFHDs achieve the fastest execution time in 86.2% of cases on *wv* and 84.0% on *yt*. When non-minimal decompositions are faster, the gap is modest: the fastest MinFHD is on average 1.3× faster than the fastest non-minimal FHD on *wv* and 2.6× faster on *yt*. Using the worst decomposition instead of the best MinFHD causes larger slowdowns, 2.1× on *wv* and 7.3× on *yt*, showing the importance of careful decomposition selection. Some triangular points still correspond to MinFHDs with exceptionally long execution time. This occurs because, for two minimum-width FHDs T_1 and T_1', where T_1 is minimal and T_1' is obtained by enlarging bags of T_1, T_1' is usually slower than T_1 but can be faster than another MinFHD T_2. The enlarged version T_2' of T_2 may be slower than T_2 yet absent from the figure because its width is larger. Thus, T_2 can appear as the slowest point despite being minimal. These cases are rare, and our cost model (Algorithm 2) avoids bad MinFHDs by selecting faster ones.

Effect of symmetry-breaking pushdown and selection. Table 4 compares our optimal symmetry selection against three alternatives: no symmetry-breaking, partial symmetry-breaking (as in [39]), and worst selection of full symmetry-breaking. Optimal selection achieves substantial speedups: 4.02×–8.72× vs. no symmetry-breaking, 1.30×–3.43× vs. partial symmetry, and 1.27×–3.00× vs. worst selection. This is because our predicate pushdown technique allows each bag to utilize more *SymR*s than prior work, and our scoring mechanism selects superior *SymR*s by maximizing local symmetry-breaking predicates for the most computationally expensive bags.

Effect of MixJoin. Fig. 10 compares our MixJoin algorithm (Sec. 6) against ASDJoin [38] across three datasets. The key difference is

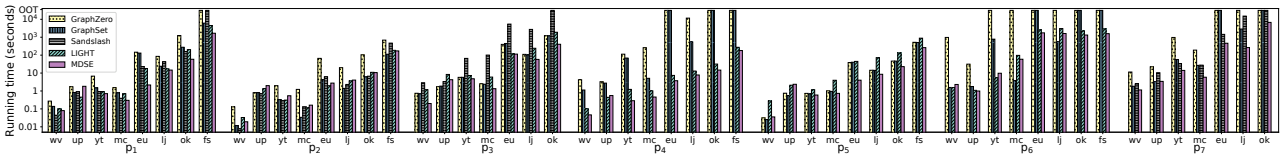


Figure 11: Evaluation on specific pattern graphs

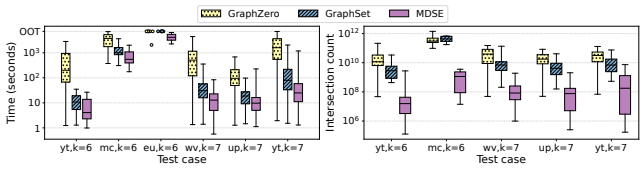


Figure 12: Evaluation on general pattern graphs

that MixJoin processes the global join within the final bag computation while ASDJoin requires an extra step. Each scatter plot shows execution times where the x-axis represents MixJoin time and the y-axis represents ASDJoin time. Points above the diagonal $y = x$ indicate cases where MixJoin is better. The results demonstrate overwhelming advantages for MixJoin, with the vast majority of points positioned above the diagonal across all datasets. MixJoin achieves remarkable average speedups of 11.6× on *wv*, 9.2× on *up*, and 12.8× on *yt*. These results validate that MixJoin's integration of global joins with bag computation significantly reduces materialization overhead while preserving set intersection efficiency.

Additional experiments on per-pattern symmetry score impact, the effect of materialization-aware cost modeling, and a detailed ablation study are provided in the technical report [1].

8.3 Performance Evaluation

Evaluation on Specific Patterns. Fig. 11 compares all five algorithms on specific patterns p_1–p_7 across multiple datasets. The y-axis shows execution time with a 6-hour timeout. LIGHT supports p_1–p_6 and Sandslash supports p_1–p_3 and p_7, while GraphZero, GraphSet, and MDSE support all patterns. Among the baselines, LIGHT is the most competitive. MDSE achieves competitive performance with LIGHT on p_2 and demonstrates clear advantages on other patterns, most notably 5.6× average speedup for p_5 and 3.4× for p_3. Against Sandslash, MDSE achieves 11.6× average speedup. On complex patterns p_1, p_3 and p_7, Sandslash times out on three cases while MDSE completes all these instances. GraphZero and GraphSet are relatively slow since they do not implement ad-hoc optimizations for these specific patterns, and MDSE outperforms them by more than two orders of magnitude. The speedups are less significant on *up*, which is sparser and has a more uniform degree distribution. As a result, large patterns (e.g., the full pattern) have fewer matches in *up*. Smaller patterns (e.g., pattern subgraphs within bags) still have many matches, leading to a high materialization cost as we need to store them in complex data structures. When the data graph is very sparse and subgraph matches are rare, matching the full pattern directly (as the baselines do) is already efficient, and the advantage of first matching smaller bags and joining results becomes less pronounced.

Evaluation on General Patterns. Fig. 12 presents a comprehensive comparison of GraphZero, GraphSet, and MDSE across different datasets and pattern graphs. The left part shows the running time. The timeout is 2 hours. The right part shows the number of set intersections to enumerate all subgraph matches, a metric to evaluate the cost of the set intersection computation in previous works [29, 30, 38, 60, 61]. For a fair comparison, we collect this count for instances where all three algorithms can finish within the time limit. The results demonstrate consistently strong MDSE advantages. On *eu*, MDSE finishes six 6-node patterns while GraphSet cannot run any 6-node pattern in the time limit, and GraphZero can only handle one pattern. In other cases, MDSE outperforms GraphSet by 2.0×–7.7× and outperforms GraphZero by 9.3×–156×. MDSE also significantly reduces the number of set intersections by 1,605×–4,139× compared to GraphSet and 185×–19,424× compared to GraphZero. In our decomposition-based approach, we enumerate $p(\tau_i)$ rather than p. When joining $\mathbf{R}(\tau_1) \bowtie \cdots \bowtie \mathbf{R}(\tau_k)$, we only need to join on attributes that are contained in multiple bags. Therefore, MDSE can use fewer set intersections to compute the results. This is the key reason why MDSE outperforms the baselines. The difference in running time is less significant than that of the intersection count. Moreover, in some cases MDSE can be slower than GraphSet. This is because tree decomposition-based approaches inherently incur *materialization cost*. After computing subgraph matches for each bag, intermediate results must be stored as sorted tries before performing the global join (Section 6). In contrast, GraphSet and GraphZero directly enumerate matches without constructing any intermediate data structures, and thus have no materialization cost. When the materialization cost is large and unavoidable, this overhead can outweigh the benefits of decomposition, allowing GraphSet to outperform MDSE.

In our technical report [1], we also confirm MDSE's strong parallelizability through thread scaling experiments.

9 CONCLUSION

We propose a novel decomposition-based approach MDSE for unlabeled subgraph enumeration. We introduce minimal fractional hypertree decompositions (MinFHDs) and develop an efficient elimination order algorithm to enumerate all MinFHDs with minimum width. We integrate symmetry-breaking with tree decomposition. Additionally, we design the MixJoin algorithm to reduce materialization overhead, and develop a comprehensive cost model that considers both intersection and materialization costs for optimal attribute ordering. Extensive experiments on eight data graphs demonstrate that MDSE consistently outperforms four state-of-the-art algorithms. On 7 selected patterns, MDSE achieves over two orders of magnitude speedup against GraphSet and GraphZero, 11.6× against Sandslash, and up to 5.6× against LIGHT. On 94 general 6/7-node patterns, MDSE outperforms GraphSet by 2.0×–7.7× and GraphZero by 9.3×–156×.

REFERENCES

[1] [n.d.]. Technical report of this paper. Retrieved May 25, 2026 from https://github.com/magic62442/subgraph-enumeration

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. ACM, 431–446.

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[4] Anna Arpaci-Dusseau, Zixiang Zhou, and Xuhao Chen. 2024. Accurate and Fast Approximate Graph Pattern Mining at Scale. *Proc. VLDB Endow.* 18, 2 (2024), 93–107. Retrieved May 25, 2026 from https://www.vldb.org/pvldb/vol18/p93-chen.pdf

[5] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*. ACM, 1447–1462.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. ACM, 1199–1214.

[7] 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.

[8] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[9] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 857–877.

[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion (Eds.). ACM, 378–391.

[11] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[12] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 581–594.

[13] 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.

[14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* 7, 7 (2014), 517–528. https://doi.org/10.14778/2732286.2732289

[15] Jörg Flum and Martin Grohe. 2006. *Parameterized Complexity Theory*. Springer.

[16] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proc. of PODS'16*. 57–74.

[17] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. 2020. Fast and Parallel Decomposition of Constraint Satisfaction Problems. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, Christian Bessiere (Ed.). ijcai.org, 1155–1162. https://doi.org/10.24963/IJCAI.2020/161

[18] Joshua A. Grochow and Manolis Kellis. 2007. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology, 11th Annual International Conference, RECOMB 2007, Oakland, CA, USA, April 21-25, 2007, Proceedings (Lecture Notes in Computer Science)*, Vol. 4453. Springer, 92–106.

[19] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1, Article 4 (aug 2014), 20 pages.

[20] Chuangyi Gui, Xiaofei Liao, Long Zheng, and Hai Jin. 2023. Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 71–85. Retrieved May 25, 2026 from https://www.usenix.org/conference/atc23/presentation/gui

[21] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*. ACM, 1067–1082.

[22] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*. 1429–1446.

[23] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*. 337–348.

[24] Zongyan He and Jeffrey Xu Yu. 2024. A Branch-&-Bound Algorithm for Fractional Hypertree Decomposition. *Proc. VLDB Endow.* 17, 13 (2024), 4655–4667. Retrieved May 25, 2026 from https://www.vldb.org/pvldb/vol17/p4655-he.pdf

[25] Pinar Heggernes. 2006. Minimal triangulations of graphs: A survey. *Discret. Math.* 306, 3 (2006), 297–317. https://doi.org/10.1016/J.DISC.2005.12.003

[26] Lin Hu, Yinnian Lin, Lei Zou, and M. Tamer Özsu. 2025. A graph pattern mining framework for large graphs on GPU. *VLDB J.* 34, 1 (2025), 6. https://doi.org/10.1007/S00778-024-00883-8

[27] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 13:1–13:16. https://doi.org/10.1145/3342195.3387548

[28] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* 16, 7 (2023), 1699–1711.

[29] Haolin Jiang, Santosh Pandey, and Hang Liu. 2025. A Comprehensive Survey of Subgraph Matching:[Experiments & Analysis]. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–30.

[30] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[31] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 282–293. https://doi.org/10.5441/002/EDBT.2017.26

[32] 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 *SIGMOD*. ACM, 925–937.

[33] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2023. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal* 32, 2 (2023), 343–368.

[34] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *SIGMOD*. ACM, 1231–1245.

[35] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2017. Scalable subgraph enumeration in MapReduce: a cost-oriented approach. *VLDB J.* 26, 3 (2017), 421–446.

[36] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228.

[37] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query Optimization by Predicate Move-Around. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 96–107.

[38] Qiyan Li, Jeffrey Yu, and Zongyan He. 2025. Subgraph Matching: A New Decomposition Based Approach. *Proc. VLDB Endow.* 18, 11 (2025), 4282–4294. Retrieved May 25, 2026 from https://www.vldb.org/pvldb/vol18/p4282-li.pdf

[39] Qiyan Li and Jeffrey Xu Yu. 2024. Fast Local Subgraph Counting. *Proc. VLDB Endow.* 17, 8 (2024), 1967–1980. Retrieved May 25, 2026 from https://www.vldb.org/pvldb/vol17/p1967-li.pdf

[40] Zhiheng Lin, Ke Meng, Chaoyang Shui, Kewei Zhang, Junmin Xiao, and Guangming Tan. 2024. Exploiting Fine-Grained Redundancy in Set-Centric Graph Pattern Mining. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi (Eds.). ACM, 175–187. https://doi.org/10.1145/3627535.3638507

[41] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BⓈX: Subgraph Matching with Batch Backtracking Search. *Proc. ACM Manag. Data* 3, 1, Article 15 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709665

[42] Yujie Lu, Zhijie Zhang, Weiguo Zheng, and Lei Zou. 2025. Accelerating Subgraph Matching through Fine-grained and Powerful Equivalences. *Proc. VLDB Endow.* 18, 11 (2025), 3896–3909. Retrieved May 25, 2026 from https://www.vldb.org/pvldb/vol18/p3896-zheng.pdf

[43] Dániel Marx and Michal Pilipczuk. 2014. Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France (LIPIcs)*, Vol. 25. 542–553.

[44] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. GraphZero: Breaking Symmetry for Efficient Graph Mining. *CoRR* abs/1911.12877 (2019). Retrieved May 25, 2026 from http://arxiv.org/abs/1911.12877

[45] Amine Mhedhbi, Amol Deshpande, and Semih Salihoglu. 2024. Modern Techniques For Querying Graph-structured Databases. *Found. Trends Databases* 14, 2 (2024), 72–185. https://doi.org/10.1561/1900000090

[46] 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.

[47] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms: [Extended Abstract]. In *Proc. of PODS'12*. 37–48.

[48] Tatsuo Ohtsuki, Lap Kit Cheung, and Toshio Fujisawa. 1976. Minimal triangulation of a graph and optimal pivoting order in a sparse matrix. *J. Math. Anal. Appl.* 54, 3 (1976), 622–633.

[49] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188.

[50] 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. https://doi.org/10.14778/3229863.3229874

[51] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and Robust Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 12, 11 (2019), 1344–1356.

[52] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. Retrieved May 25, 2026 from https://networkrepository.com

[53] 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.

[54] Nino Shervashidze, S. V. N. Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009 (JMLR Proceedings)*, David A. Van Dyk and Max Welling (Eds.), Vol. 5. JMLR.org, 488–495. Retrieved May 25, 2026 from http://proceedings.mlr.press/v5/shervashidze09a.html

[55] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 32:1–32:14. https://doi.org/10.1145/3581784.3613213

[56] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 100. https://doi.org/10.1109/SC41405.2020.00104

[57] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. *Proc. VLDB Endow.* 17, 7 (2024), 1697–1709. https://doi.org/10.14778/3654621.3654635

[58] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2019. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 120:1–120:29. https://doi.org/10.1145/3360546

[59] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *ICDE*. IEEE, 232–243.

[60] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *SIGMOD*. ACM, 1083–1098.

[61] 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.

[62] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (2012), 788–799.

[63] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 425–440. https://doi.org/10.1145/2815400.2815410

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

[65] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012).

[66] Qichen Wang and Ke Yi. 2023. Conjunctive Queries with Comparisons. *SIGMOD Rec.* 52, 1 (2023), 54–62. https://doi.org/10.1145/3604437.3604450

[67] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *ICDE*. IEEE, 136–147.

[68] Sebastian Wernicke and Florian Rasche. 2006. FANMOD: a tool for fast network motif detection. *Bioinform.* 22, 9 (2006), 1152–1153. https://doi.org/10.1093/BIOINFORMATICS/BTL038

[69] Dan E Willard. 1984. Efficient processing of relational calculus expressions using range query theory. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 164–175. https://doi.org/10.1145/602259.602281

[70] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, Mohammed Javeed Zaki, Arno Siebes, Jeffrey Xu Yu, Bart Goethals, Geoffrey I. Webb, and Xindong Wu (Eds.). IEEE Computer Society, 745–754. https://doi.org/10.1109/ICDM.2012.138

[71] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD*. ACM, 2049–2062.

[72] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[73] Hao Zhang, Miao Qiao, Jeffrey Xu Yu, and Hong Cheng. 2021. Fast Distributed Complex Join Processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2087–2092. https://doi.org/10.1109/ICDE51399.2021.00205

[74] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. 2020. Distributed Subgraph Counting: A General Approach. *Proc. VLDB Endow.* 13, 11 (2020), 2493–2507.

[75] Yuejia Zhang, Weiguo Zheng, Zhijie Zhang, Peng Peng, and Xuecang Zhang. 2022. Hybrid subgraph matching framework powered by sketch tree for distributed systems. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1031–1043.

[76] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1 (2024), 60:1–60:29. https://doi.org/10.1145/3639315

[77] Zhijie Zhang and Weiguo Zheng. 2025. BEE: Towards Redundancy Reduction via Block-Separator Decomposition for Subgraph Matching. *Proc. ACM Manag. Data* 3, 4, Article 241 (Sept. 2025), 27 pages. https://doi.org/10.1145/3749159

[78] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1-2 (2010), 340–351.