Beyond Maximum Common Subgraph: A Framework Maximizing Shared Computation for Multi-Query Subgraph Matching

ZHIJIE ZHANG, School of Data Science, Fudan University, China
SITAO YANG, School of Data Science, Fudan University, China
WEIGUO ZHENG*, School of Data Science, Fudan University, China
ZHOU QIANG, Ant Group, China
CUNTAO HONG, Ant Group, China

Subgraph matching, a fundamental operation in graph analytics, delivers all occurrences of a query pattern within a large data graph. Graph systems often require processing large batches of such queries simultaneously. The existing multi-query approaches typically aim to optimize this process by exploiting the maximum common subgraph (MCS) among query graphs. However, identifying MCS is computationally prohibitive and does not necessarily maximize computational reuse, leading to substantial redundancy and limited scalability. In this paper, we propose MASC, a novel framework for subgraph matching with Multi-query Acceleration via Shared Computation. We introduce *signature-based joint filtering* that enables shared computation in the filtering phase, significantly improving filtering efficiency across queries. Building on this, we propose the *visit-only-once paradigm* that leverages shared candidates rather than just shared structure size, ensuring each shared candidate in the data graph is processed only once and further maximizing computation sharing. To further enhance sharing, we develop the *shared block reuse* that exploits multiple overlapping substructures among queries. Compared to existing methods that leverage only a single overlap, shared computation are further substantially increased. We conducted extensive experiments on real-world datasets across various domains. The results demonstrate that MASC achieves significant improvements, achieving substantial speedup over state-of-the-art methods, with up to two orders of magnitude improvement.

CCS Concepts: • **Information systems** → *Information retrieval*; **Graph-based database models**.

Additional Key Words and Phrases: Subgraph Matching, Multi-query Processing, Shared Computation

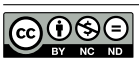
ACM Reference Format:
Zhijie Zhang, Sitao Yang, Weiguo Zheng, Zhou Qiang, and Cuntao Hong. 2026. Beyond Maximum Common Subgraph: A Framework Maximizing Shared Computation for Multi-Query Subgraph Matching. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 145 (June 2026), 27 pages. https://doi.org/10.1145/3802022

1 Introduction

Given their inherent flexibility and expressive power, graphs have become indispensable for representing the rich, interconnected structures, ranging from social networks [50] and knowledge graphs [51] to biological systems [3] and financial transaction systems [31]. A fundamental task in graph analytics is subgraph matching, which aims to find all isomorphic instances of a given

*Corresponding Author: zhengweiguo@fudan.edu.cn

Authors' Contact Information: Zhijie Zhang, School of Data Science, Fudan University, Shanghai, China, zhangzj22@m.fudan.edu.cn; Sitao Yang, School of Data Science, Fudan University, Shanghai, China, styang21@m.fudan.edu.cn; Weiguo Zheng, School of Data Science, Fudan University, Shanghai, China, zhengweiguo@fudan.edu.cn; Zhou Qiang, Ant Group, Hangzhou, China, zhichen.zq@antgroup.com; Cuntao Hong, Ant Group, Hangzhou, China, chuntao.hct@antgroup.com.



ACM 2836-6573/2026/6-ART145
https://doi.org/10.1145/3802022

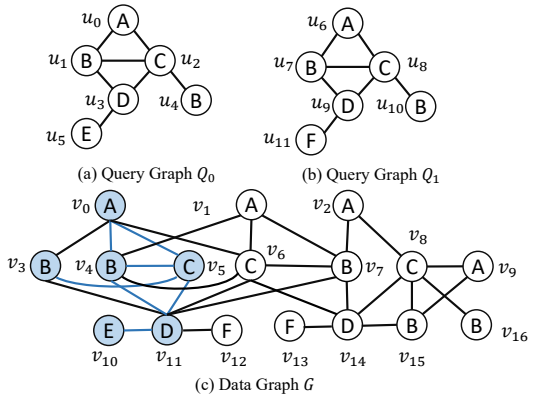


Fig. 1. Running example for subgraph matching.

query pattern within a large data graph. For example, given the query graph Q_0 and the data graph G in Figure 1, the subgraph highlighted in light blue is isomorphic to the query graph Q_0. Subgraph matching serves as the backbone for a wide range of applications, such as fraud detection in financial networks, community discovery in social platforms, and molecular structure search in bio-informatics. However, subgraph matching is computationally challenging because subgraph isomorphism is NP-complete [9], making efficient solutions especially important for large graphs.

In practical graph-based applications, a system usually needs to simultaneously monitor and evaluate a large set of query patterns. For example, financial institutions may track numerous fraud patterns in transaction networks, and bio-informatics tools frequently search for various molecular structures or interaction pathways. When subgraph matching queries overlap, naively evaluating each query independently could result in significant computational redundancy and resource overhead, which severely degrade system throughput. To address this, **multi-query optimization** becomes essential for efficiently processing multiple queries and improving the scalability of graph analytics.

1.1 Limitations of Existing Methods

Extensive research has been conducted to accelerate single-query subgraph matching [4–6, 12, 16, 18, 19, 22, 33, 34, 39, 40]. Most existing methods adopt a filtering-and-enumeration framework [38, 49], in which a filtering phase first computes candidate sets for query vertices to prune the search space, followed by an enumeration phase that performs a backtracking search to enumerate all matches. To address batch processing needs, recent years have seen increasing interest in multi-query subgraph matching, which aims to exploit overlap among queries to reduce redundant computation [27, 28, 32, 47]. A straightforward approach merges similar queries into a single composite SPARQL query using the `OPTIONAL` keyword and evaluates it with the underlying database engine [27]. However, this limits optimization opportunities at the algorithmic level. In contrast, MQO [32] identifies the maximum common subgraph (MCS) as the shared component. It first explores the MCS and then incrementally matches the remaining parts for each query. In dynamic graph scenarios, methods such as TRIC+ [47], IncMQO [41], and MQMatch [28] address multi-query processing as the data graph evolves. Nevertheless, finding the optimal shared structure remains challenging, and these methods may still incur significant overhead when handling complex or diverse query workloads. In particular, the existing algorithms still suffer from several limitations, leaving ample room for further multi-query optimization.

Limitation 1: Failing to leverage shared computation in the filtering phase. The filtering phase is conducted before the enumeration and constructs a candidate set $C(u)$ for each query vertex u based on specific filtering rules. Only data vertices in $C(u)$ are considered possible matches

for u during enumeration, which reduces the search space. Classic filters such as the neighbor label frequency (NLF) filter [12, 33] check whether the frequency of each label in the neighborhood of a query vertex is satisfied by the data vertex, which typically incurs a time complexity of $O(|E_Q|\times|E_G|)$, where $|E_Q|$ and $|E_G|$ denote the number of edges in query graph Q and data graph G, respectively. This polynomial cost can become a significant overhead on large-scale data graphs. Furthermore, recent filtering techniques [11, 22] introduce more sophisticated and dynamic filtering rules to improve pruning power, but often at the expense of even higher computational overhead per query. Therefore, efficiently sharing filtering computations across multiple queries would also be crucial for scalable multi-query processing. However, current multi-query optimization techniques mainly focus on sharing computation in the enumeration phase, while opportunities for shared computation during filtering are largely overlooked.

Limitation 2: Shared structure but not shared candidates. Most existing approaches focus on identifying common subgraphs among queries to enable computation sharing. However, discovering maximum common subgraphs is NP-hard and often too restrictive, missing other opportunities for shared computation. More importantly, shared computation occurs only when queries have shared candidate sets in the data graph. Even if two queries share a large common structure, in the extreme case where their candidate filters select disjoint sets of data vertices, there would be no actual overlap in computation. This mismatch reduces the effectiveness of structure-based sharing in practice, underscoring the need for more flexible and data-aware approaches.

Limitation 3: Limited use of multiple shared substructures. A major limitation of existing multi-query subgraph processing approaches is their insufficient exploitation of common subgraphs shared among multiple query graphs. In practice, groups of queries frequently exhibit several overlapping sections, a common occurrence when they are derived from recurring, domain-specific patterns. However, existing approaches typically identify and leverage only a single common substructure. Once this initial shared component is processed, the remaining segments of each query are executed as independent, isolated tasks. As a result, these methods fail to exploit further opportunities for computational reuse, leaving potential performance gains unrealized.

1.2 Our Approach and Contributions

To address the above limitations, in this paper, we propose a novel multi-query subgraph processing framework called MASC, which handles subgraph matching with Multi-query Acceleration via Shared Computation. MASC enables shared computation in both the filtering and enumeration phases, supports data-aware sharing beyond structural overlap, and exploits multiple shared substructures across queries.

Signature-based joint filter. To enable shared computation in the filtering phase, we introduce the *Signature-based Joint Filter*. The key idea is to identify identical filtering predicates across query vertices and evaluate them jointly. Each query vertex is associated with a signature that encodes its filtering conditions, which allows vertices with the same signature to share filtering results. For advanced propagation-based filtering methods like DPiso [11], the filtering conditions depend on the dynamically evolving candidate sets, making them difficult to explicitly express. To handle this challenge, we design a *hierarchy signature* that captures the context of these dynamic predicates, thereby enabling identical filtering checks to be detected and merged during propagation.

Visit-only-once paradigm for shared candidates. Structural overlap alone does not guarantee effective computation sharing, as reuse is possible only when queries also share candidate sets in the data graph. In practice, structurally similar queries may induce disjoint candidate sets due to query-specific label constraints or structural filters, preventing computation reuse. Motivated by this observation, we propose a *visit-only-once* paradigm that enables computation sharing at the

candidate level. Our approach explicitly tracks shared candidate sets across queries and ensures that each shared candidate in the data graph is visited only once.

Shared block reuse for multiple shared overlaps. Queries often share multiple overlapping substructures, yet existing methods typically exploit only one such overlap. We introduce *Shared Block Reuse*, which systematically exploits multiple shared substructures across queries. The key idea is to identify subgraphs (*blocks*) within each query whose matching results depend only on their connections to the rest of the query (*boundaries*). Once the matching of a block's boundary is determined, the matching of the block itself is uniquely determined (see Lemma 5.3) and can be reused across queries. Consequently, when different queries encounter the same block with an identical boundary mapping during backtracking, the block's matching results are reused directly, which enables computation reuse across multiple shared overlaps.

In summary, our main contributions are listed as follows.

- We propose a signature-based joint filtering technique that unifies identical filter conditions and enables shared computation across queries, substantially reducing redundant filtering work.
- We introduce a visit-only-once paradigm at the candidate level, which ensures that each *shared candidate* in the data graph is visited and processed only once across queries, achieving fine-grained computation sharing.
- We present shared block reuse, a mechanism that leverages the structural independence of overlapping query subgraphs (blocks) to enable more computation reuse across multiple queries, thereby eliminating redundant exploration.
- We conduct extensive experiments on real-world datasets, showing that our approach delivers significant performance improvements over state-of-the-art methods.

2 Preliminary

In this work, we consider undirected graphs where each vertex carries a label. Our methods, however, can be adapted to directed or edge-labeled graphs. We represent a query graph as $Q = (V_Q, E_Q, L_Q, \Sigma)$ and a data graph as $G = (V_G, E_G, L_G, \Sigma)$. Here, V_Q and V_G are the sets of vertices, E_Q and E_G are sets of edges, Σ denotes the set of vertex labels, and $L_Q : V_Q \to \Sigma$ ($L_G : V_G \to \Sigma$) assigns labels to the vertices in Q (G). For convenience, we may abbreviate $Q = (V_Q, E_Q)$ and $G = (V_G, E_G)$ when label functions are clear from context. Table 1 lists the frequently used notations in this paper.

Definition 2.1 (Subgraph Isomorphism). Let $Q = (V_Q, E_Q, L_Q, \Sigma)$ be a query graph and $G = (V_G, E_G, L_G, \Sigma)$ a data graph. The query graph Q is subgraph isomorphic to G if there exists an injective function $f : V_Q \to V_G$ such that:

(1) $L_Q(u) = L_G(f(u))$ for each $u \in V_Q$;
(2) For any edge $e(u_0, u_1) \in E_Q$, edge $e(f(u_0), f(u_1)) \in E_G$.

Such a mapping f is called an *embedding* of Q into G. An embedding can be described as a set of pairs $\{(u, f(u))\}$. If f is defined on a subset $I \subsetneq V_Q$, we call it a *partial embedding*, denoted as f_I. Extending a partial embedding is to a mapping for some $u \notin I$ to $v \in V_G$, forming $f_{I\cup\{u\}} = f_I \cup \{(u, v)\}$.

Definition 2.2 (Subgraph Matching). Given a query Q and a data graph G, the subgraph matching problem seeks to enumerate every possible embedding of Q in G.

Filtering–Ordering–Enumeration Framework. As surveyed in [38, 49], most subgraph matching algorithms adopt a filtering–ordering–enumeration framework. In the filtering stage, a candidate set $C(u)$ is computed for each query vertex u by discarding data vertices that cannot be matched to u under specific filtering rules. Filtering serves as an early pruning step and restricts the search space considered in subsequent stages. Based on the candidate sets, a matching order is determined to specify the order in which query vertices are processed during enumeration. Given the candidate

Table 1. Notations

Symbol	Description
$\boldsymbol{Q} = \{Q_1, \ldots, Q_m\}$	Set of query graphs
G	Data graph
V_Q, E_Q	Vertex set and edge set of query Q
V_G, E_G	Vertex set and edge set of data graph G
$u \in V_Q, v \in V_G$	Query vertex in Q, Data vertex in G
$(v_i, v_j) \in E_G$	Edge between v_i and v_j in G
$C(u)$	Candidate data vertices of u in G
$N_Q(u)$	Neighbors of u in query Q
$N_G(v)$	Neighbors of v in data graph G
$\sigma_u^k(v)$	hierarchical signature function at level k
B, ∂B	A block B and the boundary of B

sets and the matching order, an enumeration procedure recursively constructs embeddings by incrementally extending partial matches (one query vertex at a time) and checking consistency with already matched vertices.

In practical scenarios, it is common to process a batch of queries on the same data graph. When queries share similar structures, independent evaluation could lead to duplicate computations.

Definition 2.3 (Multi-Query Optimization of Subgraph Matching). Let $\boldsymbol{Q} = \{Q_0, Q_1, \ldots, Q_m\}$ be a set of query graphs and G be the data graph. The multi-query subgraph matching problem requires finding all embeddings for each $Q_i \in \boldsymbol{Q}$ in G.

Example 2.4. Consider the set of query graphs $\boldsymbol{Q} = \{Q_0, Q_1\}$ and the data graph G shown in Figure 1. For Q_0, one embedding in G is $\{(u_0, v_0), (u_1, v_4), (u_2, v_5), (u_3, v_{10}), (u_4, v_3), (u_5, v_8)\}$. Similarly, for Q_1, an embedding is $\{(u_6, v_0), (u_7, v_4), (u_8, v_5), (u_9, v_{10}), (u_{10}, v_3), (u_{11}, v_6)\}$. This example demonstrates that Q_0 and Q_1 share structural patterns and vertex labels, leading to overlapping embeddings in G. Exploiting such overlaps is crucial for multi-query subgraph matching. ■

In this paper, we focus on a read-only workload with no transactional writes or updates; consequently, all queries are executed over the same snapshot of the underlying data.

3 Filter with Shared Signature

Filtering is a crucial stage in subgraph matching, as it significantly reduces the search space by eliminating impossible candidates before verification. During the filtering phase, for each query vertex u, a candidate set $C(u)$ is constructed by eliminating data vertices that do not satisfy the required label and structural constraints. For example, under LDF filtering [40], a data vertex must share the same label as u and have a degree no smaller than that of u.

However, filtering can also account for a large portion of the overall processing time, particularly when dealing with massive graphs or a large number of queries. Since most existing filtering techniques are designed for single-query scenarios, current approaches for multi-query subgraph matching typically apply the filtering phase independently to each query. Such independent processing may lead to redundant filtering computation. In this section, we introduce the notion of *signatures* to capture and share common computations during filtering, thereby improving efficiency in the multi-query setting.

3.1 Signature

Filtering algorithms generally utilize graph invariants, such as labels, degrees, and neighborhood structures, to determine whether a data vertex v can be a candidate for a query vertex u. To

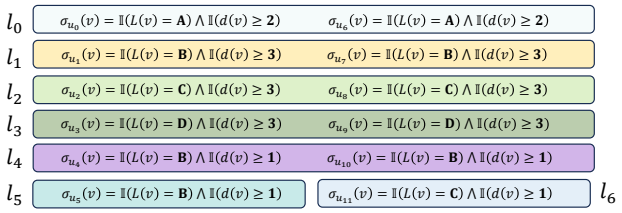


Fig. 2. Illustration of equivalent signature for LDF filter.

systematically capture these filtering conditions, we introduce the concept of *signature* for each query vertex, which encodes the specific criteria that a data vertex must satisfy to remain a candidate. By formalizing these conditions as signature functions, we identify and group query vertices from different queries that require identical filtering, enabling the sharing and reuse of filtering computation across multiple queries. The signature function is defined as follows:

Definition 3.1 (Signature). Given the query graph Q and data graph G, a *signature function* $\sigma_u : V_G \rightarrow \{0, 1\}$ is an indicator of whether a data vertex $v \in V_G$ is a feasible candidate for a query vertex $u \in V_Q$ based on a given filtering rule. If $\sigma_u(v) = 0$, then v can be safely excluded from the candidate set of u.

A natural idea arising for multi-query optimization is that, when multiple queries share identical signature functions for their respective vertices, there exists opportunities for shared computation during filtering phase.

Definition 3.2 (Shared Signature). Given two query vertices u_1 and u_2 (possibly from different queries), we say their signature functions have a *shared signature* if, for every possible data graph G and for every $v \in V_G$, it holds that $\sigma_{u_1}(v) = \sigma_{u_2}(v)$. We denote this as $\sigma_{u_1} \equiv \sigma_{u_2}$.

Query vertices with *equivalent signatures* are grouped into the same cell, and for query vertex u_i, we denote its cell as $\pi(u_i)$. We further define a mapping $\mathcal{L}$ that assigns the same identifier to all query vertices with equivalent signatures; i.e., $\mathcal{L}(u_i)$ gives the signature cell id for query vertex u_i.

Example 3.3. In the popular label and degree filter (LDF [40]), the signature is defined as $\sigma_u(v) = \mathbb{I}(L(v) = L(u)) \wedge \mathbb{I}(d(v) \geq d(u))$, where $\mathbb{I}(\cdot)$ is the indicator function. If multiple query vertices from different queries share the same label and degree, their signature functions are *shared signature*.

For the queries Q_0 and Q_1 in Figure 1, the signatures under the LDF filter are shown in Figure 2. Query vertices with equivalent signatures are grouped into the same cell and assigned the same cell id. For example, u_0 and u_6 both have label A and degree 2, and are thus clustered into the same cell: $\pi(u_0) = \pi(u_6) = \{u_0, u_6\}$, and $\mathcal{L}(u_0) = \mathcal{L}(u_6) = l_0$. In such cases, instead of evaluating the filtering predicate separately for each query vertex, we can only perform the computation once and reuse the result. In this example, instead of checking 12 times for the 12 query vertices, we only need to perform the check 7 times, once per cell of equivalent signature. ■

3.2 Hierarchical Signature

As discussed in a recent survey [49], filtering techniques in subgraph matching can be broadly categorized into one-round filtering and propagation filtering. One-round filtering (e.g., LDF) utilizes only local features of vertices, while propagation filtering iteratively refines candidate sets by propagating pruning decisions among neighboring vertices. Specifically, when a candidate is eliminated, this change is immediately propagated to update the candidate sets of its neighbors, leading to more effective pruning.

Despite its effectiveness, propagation filtering presents unique challenges in the multi-query setting. Unlike one-round filters, whose conditions are static and easy to share across queries, the

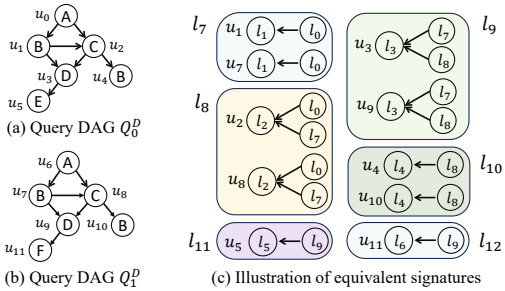


Fig. 3. Illustration of hierarchical signature for DPiso filter.

filtering conditions in propagation filtering are inherently dynamic: whether a data vertex remains a candidate for a query vertex depends not only on local invariants, but also on the evolving candidate sets of its neighbors, which may vary across different queries.

To illustrate, consider the filter of DPiso [11], a representative propagation filtering algorithm. DPiso initializes candidate sets $C(u)$ for each query vertex u using LDF. It then iteratively refines these sets by propagating pruning decisions along a BFS traversal order of the query. DPiso alternates backward and forward refinement phases for a fixed number of rounds. In each refinement round, $C(u)$ is updated based on the candidate sets of its neighbors using the following rule:

Filter Rule 3.1. *Given $v \in C(u)$, if there exists $u' \in N(u)$ such that $C(u') \cap N(v) = \emptyset$, then v can be safely removed from $C(u)$.*

To apply multi-query optimization on the filter of DPiso, there are two main challenges. First, due to the propagation of pruning decisions, it is difficult to define an explicit signature function, making it challenging to capture shared computation across queries. Second, the filtering rule of DPiso is quite strict such that it is rare for two query vertices to have identical signature functions and identical candidate sets, i.e., $C(u_1) = C(u_2)$ seldom occurs.

To address these challenges, we propose hierarchical signature that enables fine-grained shared computation across queries.

Definition 3.4 (Hierarchical Signature). Given the query graph Q and data graph G, let $\sigma_u^0 : V_G \to \{0, 1\}$ be a base signature function for each vertex $u \in V_Q$. For any integer $k \geq 1$, the signature function $\sigma_u^k : V_G \to \{0, 1\}$ at the k-th level is recursively defined as

$$\sigma_u^k(v) = \mathbb{I}\left(\sigma_u^{k-1}(v) = 1 \text{ and } \tau_u^{k-1}(v) = 1\right),$$

where $\tau_u^{k-1}(v)$ is a neighborhood constraint function that evaluates to 1 if and only if the $(k-1)$-th level signatures of v's neighbors satisfy a specific condition:

$$\tau_u^{k-1}(v) = g\left(\left\{\sigma_{u_i}^n(v_i) \mid u_i \in N_Q(u), v_i \in N_G(v), 0 \leq n \leq k-1\right\}\right)$$

for some application-specific function $g(\cdot)$, where $N(v)$ denotes the set of neighbors of v.

At each level k, the neighborhood constraint $\tau_u^{k-1}(v)$ is evaluated by a concrete filtering rule instantiated by $g(\cdot)$. Dependencies may occur among signatures at the same level; however, they are acyclic and adhere to a fixed evaluation order. Therefore, no circular dependency arises. In particular, if $\tau_u^{k-1}(v_0)$ depends on $\tau_u^{k-1}(v_1)$, the reverse dependency does not occur. The hierarchical refinement terminates when the filtering results converge, i.e., when $\tau_u^k(v) = \tau_u^{k-1}(v)$ for all data vertices v. In practice, we use a small fixed number of refinement levels to control the filtering cost.

Similarly, at the k-th level, $\mathcal{L}^k$ denotes the mapping from each query vertex to its signature cell identifier, and $\pi^k(u_i)$ denotes the signature cell containing u_i at the k-th level.

Algorithm 1: Filter_Signature_DPiso(Q, G)

Input: set of query graphs Q, data graph G
Output: Candidate sets $\{C(u_i)\}$

```
1  {q_D} ← BuildDAG(Q) ;
2  for k ← 0 to 3 do
3      foreach Q ∈ Q do
4          foreach u ∈ V_Q (reverse if k is even) do
5              if k = 0 then
6                  σ_u^0(v) = 𝕀(L(v) = L(u)) ∧ 𝕀(d(v) ≥ d(u))
7              else
8                  σ_u^k(v) = 𝕀(σ_u^{k-1}(v) = 1) ∧ 𝕀(∀u' ∈ N*(u), ∃v' ∈ N(v), σ_{u'}^{k-1}(v') = 1)
9      ClusterCells ← ClusterSignatures(Q) ;
10     UpdateCandidateSets(ClusterCells, G) ;
```

Specifically, for DPiso, the base signature is defined as $\sigma_u^0(v) = \sigma_u(v)$. The neighborhood constraint function is given by

$$\tau_u^k(v) = \mathbb{I}\left(\forall u' \in N^*(u),\ \exists v' \in N(v),\ \sigma_{u'}^k(v') = 1\right),$$

where $N^*(u)$ denotes either the forward neighbors $N^+(u)$ or the backward neighbors $N^-(u)$ of u, depending on whether forward or backward refinement is applied. The number of refinement levels is set to 3 for DPiso, as suggested in the original work [11].

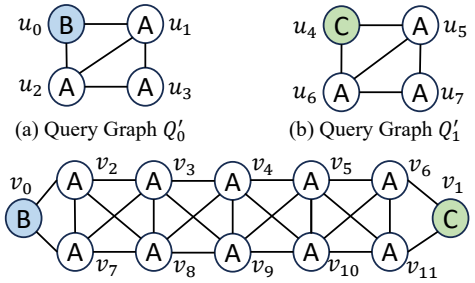
3.3 Filter with Signature

With hierarchical signatures, we organize the filtering process such that, for each refinement round k, we compute the k-level signature function σ_u^k for every query vertex u from different queries. When a group of vertices share the same signature at round k, we can merge their computations and perform the filtering only once for this shared signature. The results are then shared by all these vertices in subsequent rounds, which avoids redundant computation during filtering stage.

Algorithm 1 presents the DPiso filter enhanced with equivalent signature-based filtering. First, it constructs a DAG for each query (Line 1). The algorithm initializes base signatures using LDF (Line 5), then iteratively refines these signatures over three additional rounds by incorporating neighborhood information (Line 8). Vertex order is reversed in even rounds, following the original DPiso strategy (Line 4). After each round, query vertices with equivalent signatures are clustered (Line 9), and the filter is evaluated once per cell to update candidate sets (Line 10). This process efficiently eliminates redundant filtering across multiple queries.

Example 3.5. For the queries Q_0 and Q_1 in Figure 1, their DAGs are depicted in Figures 3(a) and 3(b), respectively. The base signatures derived from LDF, along with the corresponding cells and cell identifiers, are shown in Figure 2. The signatures and cell groupings after round 1 are illustrated in Figure 3(c).

The clustering process follows the vertex order. We represent each vertex's signature along with its neighbors and their respective cell identifiers. For example, u_0 and u_6 have no backward neighbors, so $\mathcal{L}^1(u_0) = \mathcal{L}^1(u_6) = \mathcal{L}^0(u_0) = \mathcal{L}^0(u_6) = l_0$. For u_1 and u_7, whose backward neighbors are u_0 and u_6, respectively, since $\mathcal{L}^0(u_1) = \mathcal{L}^0(u_7)$ and $\mathcal{L}^1(u_0) = \mathcal{L}^1(u_6)$, we assign $\mathcal{L}^1(u_1) = \mathcal{L}^1(u_7) = l_7$. Consequently, for u_2 and u_8, the backward neighbors of u_2 are u_0 and u_1, while the backward neighbors of u_8 are u_6 and u_7. Since we have $\mathcal{L}^0(u_2) = \mathcal{L}^0(u_8)$, as well as $\mathcal{L}^1(u_0) = \mathcal{L}^1(u_6) = l_0$ and $\mathcal{L}^1(u_1) = \mathcal{L}^1(u_7) = l_7$, it follows that $\mathcal{L}^1(u_2) = \mathcal{L}^1(u_8) = l_8$. By applying this process iteratively, we



(a) Query Graph Q'_0 (b) Query Graph Q'_1

(c) Data Graph G'

Fig. 4. Example of no shared candidate.

obtain all equivalent signatures for round 1 as shown in Figure 3(c). As a result, the need for filter checking is significantly reduced, since redundant checks are eliminated. ■

With hierarchical signatures, we address the challenge of defining explicit signature functions by decomposing the filtering process into incremental layers, making the computation more manageable. At each layer, shared intermediate results are exposed, enabling the reuse of overlapping computations that would otherwise be recomputed for different query vertices. Although the final signature functions and candidate sets may still differ between queries, this hierarchical structure ensures that redundant checks are efficiently merged and reused throughout the filtering process, enhancing the filtering efficiency of multi-query processing.

Complexity Analysis. In each refinement round, generating hierarchical signatures for all query vertices requires aggregating their neighbors, costing $O(|E_Q|)$ time, where $|E_Q| = \sum_{Q \in \mathcal{Q}} |E_Q|$ is the total number of edges across all queries. Identifying the identical signatures takes $O(|V_Q| \log |V_Q|)$ time, where $|V_Q| = \sum_{Q \in \mathcal{Q}} |V_Q|$ is the total number of query vertices.

For the filtering step, let $|S|$ denote the number of distinct signatures, and $\overline{|C|}$ the average number of candidate data vertices per signature. The filtering cost is $O(|S| \cdot \overline{|C|} \cdot d_G)$, where d_G is the average degree in the data graph. Overall, the total time complexity per refinement round is: $O(|E_Q| + |V_Q| \log |V_Q| + |S| \cdot \overline{|C|} \cdot d_G)$.

In the worst case, when all signatures are distinct, the filtering cost is equivalent to evaluating each query independently, with an additional overhead for signature construction. In practice, however, many query vertices share the same signature, enabling shared filtering. Moreover, since $\overline{|C|}$ is typically much larger than V_Q, the overhead of signature construction is negligible compared to filtering. Overall, signature-based filtering brings substantial efficiency gains in multi-query scenarios.

4 Visit-Only-Once Backtracking

4.1 Visit-Only-Once for Shared Candidates

While most existing approaches focus on identifying common substructures among queries for computation sharing, shared structure alone is not sufficient. In practice, substantial computational savings are possible only when the candidate sets of these shared substructures also overlap on the data graph. Motivated by this observation, we propose a visit-only-once strategy that enables computation sharing at the candidate level, ensuring that each shared candidate is processed only once across multiple queries.

Definition 4.1 (Shared Candidates). Given a set of query graphs $\mathcal{Q} = \{Q_0, Q_1, \ldots, Q_m\}$ and a data graph G, let $C(u)$ denote the candidate set of a query vertex u, i.e., $v \in C(u)$ means data vertex v is a candidate for u. For a set of query vertices $\mathcal{U} = \{u_0, u_1, \ldots, u_m\}$ with $u_i \in Q_{j_i}$ for some $Q_{j_i} \in \mathcal{Q}$, a

data vertex v is called a *shared candidate* for $\mathcal{U}$ if $v \in \bigcap_{i=1}^{m} C(u_i)$, i.e., v is a candidate for each u_i in their respective queries.

Example 4.2. Consider another example in Figure 4 with two queries, Q'_0 and Q'_1, and a data graph G'. Both Q'_0 and Q'_1 contain a shared triangle substructure: the subgraph induced by (u_1, u_2, u_3) in Q'_0 and the subgraph induced by (u_5, u_6, u_7) in Q'_1 are isomorphic. However, the vertices in the shared substructure are required to be adjacent to vertices with label B in Q'_0 and label C in Q'_1, resulting in different filter requirements for each query. As a result, after applying the filters, there are no shared candidates for the triangle substructure in G'. Thus, despite the structural overlap, no computation would be reused between the two queries. ■

Advantages of shared candidates. When a data vertex serves as a candidate for multiple queries (i.e., a shared candidate), redundant checks can be eliminated by evaluating its validity for all relevant queries in a single visit. To this end, the *visit-only-once paradigm* is proposed, which guarantees that each shared candidate is checked only once at each backtracking step.

Visit-Only-Once Paradigm. Given a batch of queries $\mathcal{Q}$ over a data graph G, let M denote a partial mapping that currently covers a subset of queries $\mathcal{Q}_V \subseteq \mathcal{Q}$ with corresponding data vertices $\{v_0, v_1, \ldots, v_k\}$. During backtracking, when extending M, the set of candidate data vertices V^* that are candidates for multiple queries in $\mathcal{Q}_V$ are identified. The *visit-only-once paradigm* ensures that each $v \in V^*$ is visited and checked only once for all relevant queries at the current backtracking node. Specifically, the following cases are considered for each $v \in V^*$:

- If v satisfies the constraints of all queries in $\mathcal{Q}_V$, M is extended with v, and the search proceeds.
- If v satisfies only a subset of queries $\mathcal{Q}_S$ in $\mathcal{Q}_V$, M is split accordingly, and the search continues for the satisfied queries in $\mathcal{Q}_S$, and $\mathcal{Q}_V$ is updated as the satisfied queries.
- If v satisfies none of the queries, backtracking is performed.

This paradigm eliminates redundant computation by ensuring that each shared candidate is evaluated *only once* at each backtracking step. In cases where no shared candidate exists, the procedure falls back to selecting candidates for individual queries, following standard single-query backtracking.

To enable efficient multi-query processing, we introduce the Shared Candidate Set (SCS), which merges candidate data vertices that are valid for multiple queries into unified structures. In the SCS, each candidate vertex v is annotated with a tag, $\mathrm{Tag}(v)$, representing the set of queries for which v is a valid candidate. This organization enables direct retrieval of candidates for any query or query subset, eliminating redundant checks. Specifically, for a query vertex u, a data vertex v, and a query subset $\mathcal{Q}_V$, we define $N_{SCS}(u, v, \mathcal{Q}_V)$ as the set of candidate neighbors of v (for u) whose tag matches $\mathcal{Q}_V$. These interfaces provide efficient access to shared candidates and their neighbors, supporting scalable multi-query processing.

Example 4.3. Consider $\mathcal{Q} = \{Q_0, Q_1\}$ and the data graph G in Figure 1, the candidates after filtering and the shared candidate set are shown in Figure 5. For vertices with shared candidates (e.g., u_0 and u_6), the SCS consolidates their candidates into unified entries. For instance, the entry for u_0 and u_6 includes v_1 with $\mathrm{Tag}(v_1) = \{Q_0, Q_1\}$, indicating v_1 is a candidate for both queries, while v_9 with $\mathrm{Tag}(v_9) = \{Q_1\}$ is only a candidate for Q_1. When performing multi-query matching, the algorithm first selects candidates like v_1 that satisfy multiple queries simultaneously. Suppose v_1 is chosen to match u_0 and u_6, and the current valid queries are $\{Q_0, Q_1\}$. The SCS can then be used to directly retrieve the candidate neighbors of v_1 with $\mathrm{Tag} = \{Q_0, Q_1\}$, which are v_4 and v_7. ■

The pseudo-code in Algorithm 2 implements the *visit-only-once paradigm*. At each backtracking step, the algorithm first checks if all queries are matched (LINE 1); if so, it outputs the current mapping and terminates this branch (LINE 2). Otherwise, the algorithm selects the next subset of

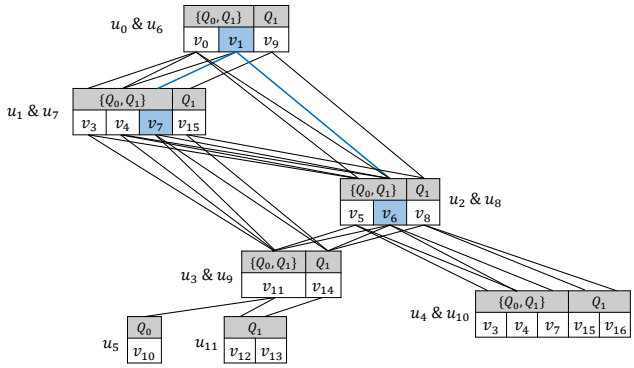


Fig. 5. Illustration of shared candidate set.

Algorithm 2: VISIT-ONLY-ONCE BACKTRACK(Q, M)

Input: Current valid queries Q, Current partial mapping M
Output: All valid mappings of Q extended from M

```
1 if all queries are matched in M then
2     output M; return
3 Select next subset of queries Q_V to match, prioritizing largest shared candidate sets from SCS;
4 foreach candidate v in SCS for Q_V do
5     Q_S ← subset of queries in Q_V for which v satisfies all constraints in M;
6     if Q_S ≠ ∅ then
7         Extend M with v for queries in Q_S; VISIT-ONLY-ONCE BACKTRACK(Q_S, M);
8     else
9         continue;
```

queries $Q_V \subset Q$(LINE 3) and extracts the shared candidate set directly from SCS for Q_V. For each v, the algorithm determines the subset of queries for which v satisfies all current matching constraints (LINE 5); if $Q_S \neq \emptyset$, the mapping is extended for these queries and the search continues recursively (LINE 7- LINE 7); otherwise, v is skipped (LINE 9). This approach guarantees that each candidate is checked only once per backtracking node, eliminating redundant computation as described in the visit-only-once paradigm.

Example 4.4. Consider the backtracking search on the candidate sets in Figure 5, where the current partial matching consists of the blue-highlighted vertices v_1, v_4, and v_7, and the set of valid queries is $Q_V = \{Q_0, Q_1\}$. According to the visit-only-once paradigm, we first consider candidates with Tag = $\{Q_0, Q_1\}$, that is, those that are valid for both queries. For v_4 and v_7, we retrieve their candidate neighbors for $\{Q_0, Q_1\}$ and compute their intersection, resulting in v_{11}. Since v_{11} satisfies the constraints of all queries in Q_V, the mapping M can be extended with v_{11}, and the search proceeds with both Q_0 and Q_1. In the other branch, we consider candidates with Tag$(\cdot) = \{Q_1\}$ only. This yields v_{14} as a candidate neighbor, which now corresponds only to Q_1. In this case, the set of valid queries is reduced to $\{Q_1\}$, and the search continues as in the single-query scenario. This ensures that the shared candidate is visited only once, improving efficiency in multi-query subgraph matching. ■

4.2 Order Optimization with Shared Candidates

As previously discussed, computation sharing in multi-query evaluation depends not just on structural overlap among queries, but more critically on the overlap of their candidate data vertices. Maximizing structural overlap alone is not always optimal (see Example 4.2), as it does not guarantee the largest set of shared candidates. In fact, a smaller structural overlap may sometimes yield more shared candidates and thus lead to greater computational reuse.

To address this, we propose a novel ordering strategy that explicitly considers the size of shared candidate sets at each matching step. Our method is guided by two main principles: (1) maximizing the overlap of candidate sets across all queries to fully exploit the visit-only-once paradigm, and (2) aggressively reducing the search space early by prioritizing query vertices with fewer candidates, i.e., those with high selection rates.

In this multi-query setting, we generate a matching order for each query individually. At each step of order construction, the selection of the next query vertex is guided by candidate overlap statistics aggregated across the entire batch of queries. By combining local query structure with global candidate sharing information, our greedy strategy aims to enhance computation reuse and improve the efficiency of multi-query evaluation.

Initial Vertex Selection. For each query, we select the initial matching vertex by computing $|C(u)|/\deg(u)$ for each query vertex u, where $|C(u)|$ is the candidate set size in the data graph and $\deg(u)$ is its degree in the query graph. The vertex with the smallest value is chosen as the root, favoring vertices that are both selective and highly connected, which helps prune the search space early. After selecting the root, we group all query vertices with the same label to facilitate subsequent candidate sharing.

Subsequent Vertex Selection. At each step, we consider unchosen query vertices adjacent to those already selected. For each candidate group of query vertices with the same label (denoted as V_L), we compute the shared candidate ratio $ratio(V_L)$, defined as the size of the intersection of their candidate sets divided by the size of their union, and the coverage $coverage(V_L)$, defined as $|V_L|$. Let r_{max} be the maximum shared candidate ratio among all groups. For a threshold δ, we select all groups with $ratio(V_L) \geq r_{max} - \delta$, and then choose the group with the largest $coverage(V_L)$. This relaxed selection balances candidate sharing efficiency and coverage.

5 Shared Block Reuse

The previous section presented the Visit-only-once strategy, which adopts the shared candidates for reducing overlap computation at each backtracking step. In this process, the set of valid queries $\boldsymbol{Q}$ keeps shrinking as queries are removed once their shared substructure is processed. However, there may be more than one overlapping substructure among the queries. Focusing on only one at each step may overlook additional overlaps, thus limiting opportunities for further computational sharing.

Example 5.1. Consider $Q_0^{''}$ and $Q_1^{''}$ in Figure 6, which share two pairs of corresponding substructures: $\{u_0, u_1, u_2\}$ in $Q_0^{''}$ with $\{u_7, u_8, u_9\}$ in $Q_1^{''}$, and $\{u_4, u_5, u_6\}$ in $Q_0^{''}$ with $\{u_{11}, u_{12}, u_{13}\}$ in $Q_1^{''}$. Under the visit-only-once strategy, selecting any one shared substructure (either the first or the second) leads to single-query evaluation for $Q_0^{''}$ and $Q_1^{''}$ once their query-specific parts are reached. As a result, other overlapping substructures are not leveraged, limiting further opportunities for computational sharing. ■

Definition 5.2 (Block and Boundary). Given a query graph Q, a *block* B is a subgraph of Q. The *boundary* of B, denoted as ∂B, is the set of vertices in B that are adjacent to at least one vertex in $Q \setminus B$.

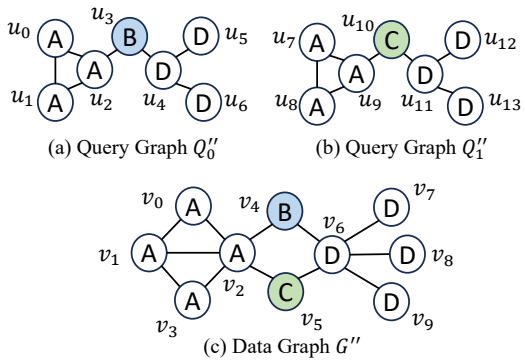


Fig. 6. Example of multiple overlap substructure.

Intuitively, if no vertex in B shares a label with any vertex outside B, then once the mapping of the boundary ∂B is fixed, the matching of B is fully determined and unaffected by the rest of the query.

Lemma 5.3. *Let B be a block in a query graph Q with boundary ∂B. If no vertex in B shares a label with any vertex in $Q \setminus B$, then the matching of B is uniquely determined by the mapping of ∂B, independent of the mapping of vertices outside B.*

Proof. Since the boundary ∂B contains all vertices in B adjacent to $Q \setminus B$, and label sets are disjoint, no vertex in B can be mapped to the same data vertex as any vertex in $Q \setminus B$. Once the mapping of ∂B is fixed, the remaining vertices in B can only be matched based on internal structure and their connections to ∂B. Thus, the matching of B is independent of the mapping outside B. □

Example 5.4. Consider the queries Q_0'' and Q_1'' in Figure 6. The block B_0 induced by u_0, u_1, u_2 in Q_0'' has boundary $\partial B_0 = \{u_2\}$, as u_2 is the only vertex in the block adjacent to vertices outside the block. Similarly, the block B_1 induced by u_7, u_8, u_9 in Q_1'' has boundary $\partial B_1 = \{u_9\}$. ■

Definition 5.5 (Shared Block). Given two query graphs Q_0 and Q_1, let $B_0 \subseteq Q_0$ and $B_1 \subseteq Q_1$ be blocks. B_0 and B_1 form a *shared block* if there exists a bijection $f : V(B_0) \to V(B_1)$ such that:

(1) B_0 is isomorphic to B_1; that is, for any $(u, v) \in E(B_0)$, $(f(u), f(v)) \in E(B_1)$, and for any $u \in V(B_0)$, $L(u) = L(f(u))$;
(2) $f(\partial B_0) = \partial B_1$. Here, $f(\partial B_0) = \{f(u) \mid u \in \partial B_0\}$.

Lemma 5.6. *Let $\mathbf{Q}$ be a set of query graphs, and let $\mathcal{B} = \{B_Q \subseteq Q : Q \in \mathbf{Q}\}$ be a set of blocks that form a shared block. If the mapping of boundary is fixed, the set of valid matchings for each B_Q is identical across all $Q \in \mathbf{Q}$.*

Proof. By the definition of shared block, all B_Q are isomorphic to each other, and their boundaries correspond under bijections f. Although each B_Q is a subgraph of a different query graph Q, potentially with distinct structures outside B_Q, Lemma 5.3 guarantees that the set of valid matchings for B_Q is fully determined by its boundary mapping. Therefore, any differences in the remaining parts of the query graphs have no impact on the matchings within B_Q. Consequently, for any fixed boundary mapping, all blocks in $\mathcal{B}$ admit the same set of valid matchings, regardless of the surrounding query context. This property enables the reusable computation of matchings for shared blocks across multiple queries. □

Building upon the visit-only-once paradigm, which focuses on leveraging a single overlap at a time, the notion of shared blocks provides the opportunity to systematically identify and exploit

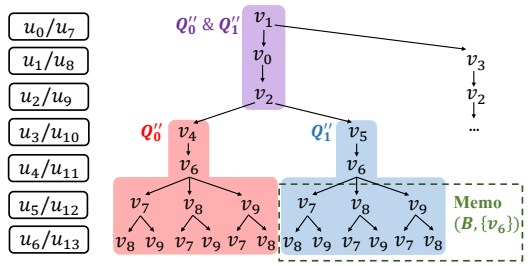


Fig. 7. Example of shared block reuse.

multiple overlaps among queries. Even as the set of currently valid queries Q becomes smaller, for example, when only a single query remains valid and backtracking needs to explore query-specific regions, shared blocks remain effective. In particular, as long as the matchings at the boundaries of the shared block are identical, the computation within the block can be reused across occurrences, regardless of query-specific explorations outside the block.

Example 5.7. As shown in Figure 6, blocks B_0 and B_1 are shared blocks, where there exists mapping $f : f(u_0) = u_7$, $f(u_1) = u_8$, and $f(u_2) = u_9$ satisfying Definition 5.5. Even if the matching order first processes u_3 or u_{10}, which are vertices specific to individual queries due to their distinct labels, the computation within the shared block remains reusable because it depends solely on the boundary matching and is independent of the other vertices. ■

To facilitate the reuse of matching results among shared blocks, we introduce the following **Block Memorization Table**. This data structure records the internal matching results of a block under different boundary mappings. When a shared block is encountered again with an identical boundary mapping, the previously computed results can be directly reused, effectively avoiding redundant computation and improving efficiency.

Definition 5.8 (Block Memorization Table). A *Block Memorization Table* is a data structure that maps each shared block B and a boundary mapping m to the set of partial matches of the interior vertices of B under m: $\texttt{Memo} : (B, m) \mapsto \mathcal{R}(B, m)$, where B is a shared block, m is a mapping of the boundary vertices ∂B, and $\mathcal{R}(B, m)$ is the set of partial matches of $V(B) \setminus \partial B$ given m.

Algorithm 3 extends the visit-only-once backtrack procedure by introducing block-level Memorization with shared block reuse. At certain steps (LINES 4-6), the algorithm identifies the boundary mapping of a block B and checks whether the results for this subproblem have already been stored in the Memo table (LINE 7). If so, it directly reuses the memorized partial solutions across all relevant queries (LINES 8-12), thus avoiding redundant computation through shared block reuse. Otherwise, it enumerates matches and stores the partial mapping of B for future use (LINE 20). This technique of shared block reuse enables efficient pruning of multiple overlaps in multi-query scenarios, and further enhances the visit-only-once backtracking paradigm.

Example 5.9. Consider queries $Q = \{Q_0'', Q_1''\}$ and the data graph G'' in Figure 6. Figure 7 illustrates a part of the corresponding backtracking tree. Following the *visit-only-once paradigm*, the algorithm first explores the shared candidate vertices v_1, v_0, and v_2 (highlighted in purple), where both queries are simultaneously valid. Upon reaching vertices with distinct labels, the search branches: v_4 is matched for Q_0 and v_5 for Q_1, resulting in two separate subtrees. At this step, the valid query set for each branch reduces to a single query, and the process continues as single-query matching (with Q_0 shown in red and Q_1 in blue).

However, a key observation is that the two subtrees rooted at v_6 in both branches are identical. This occurs because Q_0 and Q_1 have shared more than one overlaps. In Figure 6, the regions B_2 =

Algorithm 3: MEMO BACKTRACK(Q, M, Memo)

Input: Current valid queries Q, current partial mapping M, Block Memorization Table Memo
Output: All valid mappings of Q extended from M

```
if all queries are matched in M then
    output M; return
Select next subset of queries Q_V to match, prioritizing largest shared candidate sets from SCS;
if Block Memorization is applied at this step then
    B ← the current block to check;
    m ← the boundary mapping for ∂B induced by M;
if (B, m) in Memo then
    // retrieve memorized results
    foreach partial mapping m_B in Memo(B, m) do
        Q_V ← Q∩ queries of m_B;
        M' ← M extended by m_B;
        MEMO BACKTRACK(Q_V, M', Memo);
    return
// Otherwise, enumerate matches and memorize them
R ← ∅;
foreach candidate v in SCS for Q_V do
    Q_S ← subset of queries in Q_V for which v satisfies all constraints in M;
    if Q_S ≠ ∅ then
        M' ← M extended by assigning v to Q_S;
        Add the partial mapping of B into R;
        MEMO BACKTRACK(Q_S, M', Memo);
Memo(B, m) ← R;
```

$\{u_4, u_5, u_6\}$ and $B_3 = \{u_{11}, u_{12}, u_{13}\}$ form *shared blocks*. When $Q_0^{''}$ first encounters the subtree rooted at v_6, the algorithm computes and stores the result in the memorization table as Memo(B_2, $\{v_6\}$). Subsequently, when the backtracking search on $Q_1^{''}$ encounters the block B_3, which is the shared block of B_2 and the match of boundary ∂B_3 is $\{v_6\}$, it finds the precomputed entry in Memo and directly reuses the result, avoiding redundant computation. ■

The cost of shared block reuse consists of both time and space overheads associated with managing boundary mappings. For each access to a reusable block B, we must check whether the current boundary mapping matches a stored key. This requires storing all possible boundary mappings, which grow exponentially with the size of the boundary: specifically, the number of keys could be large as $O(V_G^{|\partial B|})$ where V_G is the number of vertices in the data graph. While the values (i.e., the sets of valid matchings) can be stored on disk to save memory, the keys themselves must remain in memory to support efficient lookup and collision checking.

In principle, there is a tradeoff between supporting larger boundaries for greater reuse potential and keeping overhead manageable. We address this by only selecting blocks with $|\partial B| = 1$, that is, blocks separated by a single cut-point. This approach ensures efficient in-memory storage and fast lookup, while still capturing a significant portion of reusable computation in practice.

Table 2. Real-world data graphs with vertex labels.

Dataset	$\|V\|$	$\|E\|$	d_G	$\|\Sigma\|$	Type
Citeseer	3,279	4,552	2.8	6	Citation
YeastS	2,361	7,182	6.1	66	Protein
HPRD	9,303	34,998	7.5	307	Protein
Human	4,674	86,282	36.9	44	Protein
Figeys	2,239	6,432	5.7	100	Protein
DBLP	317,080	1,049,866	6.6	100	Collab.
WordNet	146,005	656,999	9.0	100	Lexical
Stanford	281,903	1,992,636	14.1	100	Web
Youtube	1,134,890	2,987,624	5.3	100	Social

To identify shared blocks among different query graphs, we employ a two-step process. First, we decompose each query graph into a collection of *blocks* by partitioning the graph according to boundary size constraints. This results in a set of candidate blocks for each query. Second, we compare blocks from different queries to identify isomorphic pairs. For each pair of isomorphic blocks, we further verify whether their boundary nodes correspond under the isomorphism. Only blocks that are isomorphic and have matching boundaries are considered as *shared blocks* across queries. This process enables efficient detection of reusable substructures.

6 Experimental Evaluation

6.1 Experimental Setting

Comparison Methods. We compare our approach with three representative subgraph matching methods: MQO [32], TRIC+ [47], MQMatch [28], IncMQO [41], GuP [2], and Pilos [37] . Among them, MQO is specifically designed for multi-query subgraph matching in the static setting, which is the focus of our paper, where both the query set and the data graph remain unchanged throughout processing. In contrast, TRIC+, IncMQO, and MQMatch are developed for dynamic data graphs, allowing continuous updates to the data graph while maintaining a fixed set of queries. To ensure a fair comparison, we adapt these methods to the static multi-query scenario by evaluating only their batch query processing capabilities, with no updates applied. Moreover, we include GuP and Pilos, two state-of-the-art single-query subgraph matching methods, as representative baselines to provide a broader performance comparison.

Data Graphs. We conduct experiments on both real and synthetic graph datasets to comprehensively evaluate our methods. Our real-world collection includes 9 graphs, covering a broad spectrum of graphs, such as collaboration, social, biological, and lexical graphs. The details and statistics for these datasets are listed in Table 2. For datasets with native vertex labels, we use the original label sets without modification. For datasets without labels, vertex labels are assigned uniformly at random using 100 distinct labels by default, following established experimental protocols [2, 38, 49]. For synthetic datasets, we use EvoGraph [29] to synthesize large graphs by scaling smaller inputs while maintaining their essential structural properties. The labeling of synthetic vertices follows the same method as that of real graphs, ensuring consistent comparisons throughout all experiments.

Query Graphs. We generate query graphs as connected induced subgraphs of the data graph using a random walk approach [15]. The query generation process is parameterized by three factors: the number of vertices in each query graph (k), the batch size (b), and the overlap rate among queries (β). Specifically, for each batch, we first extract a common core subgraph of size $k_{\text{core}} = k \times \beta$, which is shared by all b queries in the batch. The family size is set to 5, i.e., we generate 5 queries sharing the same core. Each individual query is then constructed by extending this shared core to k vertices via additional random walks. This strategy allows us to systematically control the

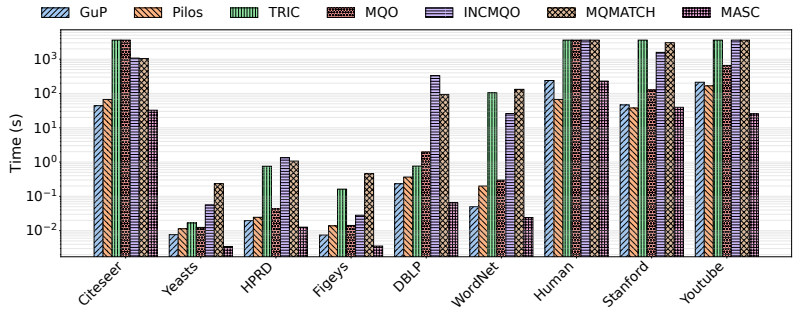


Fig. 8. Total elapsed time across different datasets.

overlap and diversity among query graphs, following the methodology in prior works [38, 49]. In addition, we generate small queries following common patterns observed in practice, including path- and star-shaped structures, to complement the randomly generated queries.

Metrics. To assess the efficiency and effectiveness of all methods, we employ the following metrics.

- *Total Elapsed time.* The total elapsed time measures the duration required to complete the entire query workload, i.e., subgraph matching for all queries in the batch. This metric reflects the end-to-end performance of each approach. Given the NP-hardness of subgraph matching, some cases may not finish within a reasonable time. We set a timeout of 3600s; if an experiment exceeds this limit, we record the elapsed time as 3600s. We use total elapsed time as the primary metric for evaluating algorithm performance.
- *Peak Memory.* Peak memory is the maximum memory consumed during query processing, reflecting the memory efficiency and scalability of each algorithm.
- *Completion Ratio.* Completion ratio measures the proportion of queries for which the algorithm successfully finds all valid matches within the time limit.
- *Filtering Time.* Filtering time measures the duration spent in the candidate filtering stage before enumeration.

Experiment Environment. All experiments were performed on a server with Ubuntu 22.04, featuring an Intel(R) E5-2596v4 CPU at 2.2 GHz and 128 GB RAM.

6.2 Overall Performance

We use the following experimental setup by default for the overall performance evaluation. Each query graph is generated with 15 vertices. In the main experiment, we set the overlap ratio to 0, meaning there is no predefined overlap between queries. Each batch contains 30 queries, and in total, 300 queries are evaluated in this experiment. For all runs, we set a time limit of 3600 seconds.

6.2.1 Total Elapsed Time. The total elapsed time results are presented in Figure 8. Overall, MASC significantly outperforms existing multi-query methods across most datasets, often achieving improvements of up to one to two orders of magnitude on both large and small graphs. Even without predefined overlap, MASC demonstrates a strong capability to effectively capture incidental overlap between randomly selected graphs. On small graphs, this advantage mainly comes from the filtering stage, which often dominates the total runtime while enumeration remains relatively simple. In such cases, our *sharing joint filtering* technique effectively reduces filtering overhead by consolidating identical checking conditions across queries. On large graphs such as Youtube, enumeration becomes more complex and often dominates the overall runtime. Here, the *visit-only-once on shared candidates* and *shared block reuse* techniques help improve efficiency by increasing computation sharing during enumeration. On Human and Stanford, MASC shows only marginal

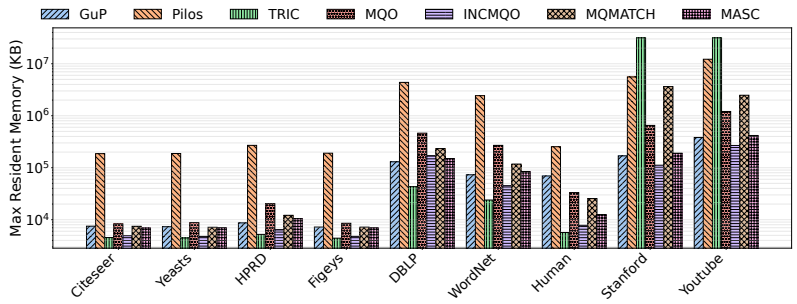


Fig. 9. Peak memory across different datasets.

Table 3. Completion rates (%) of different methods.

Dataset	TRIC	MQO	INCMQO	MQMATCH	MASC	GuP	Pilos
WordNet	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Citeseer	0.00	0.00	68.33	60.00	100.00	90.33	85.33
Human	0.00	0.00	0.00	0.00	90.00	91.66	83.00
Stanford	0.00	50.00	56.66	20.00	100.00	89.66	84.33
Youtube	0.00	70.00	0.00	0.00	100.00	84.33	77.66

improvement over the best state-of-the-art methods. This behavior is expected given the NP-hardness of subgraph matching: a single particularly hard query in a batch can dominate the total execution time and limit the benefit of shared computation.

6.2.2 Peak Memory. Memory usage is also a critical aspect of multi-query processing. Figure 9 presents the peak memory usage of different multi-query methods across various datasets. Among these methods, our approach MASC generally achieves lower peak memory consumption. Compared to single-query methods, MASC exhibits comparable peak memory consumption and does not introduce noticeable additional memory overhead. On small datasets such as Citeseer and Yeast, the memory usage of MASC is comparable to MQMatch and only slightly higher than TRIC. However, on large datasets like Youtube, where TRIC's memory consumption increases significantly, our approach achieves the lowest peak memory overall. This efficiency may result from our shared candidate structure, which avoids redundant storage by maintaining only a single copy of each shared candidate. Furthermore, unlike MQO and TRIC, our method does not explicitly materialize intermediate results in memory, but instead enables reuse through lightweight references. These design choices likely contribute to the reduced memory footprint observed in our experiments.

6.2.3 Completion Rate. We report completion rates under a 3600s time limit (120s per query for single-query methods), and Table 3 summarizes the completion rates (%) on the more challenging datasets. For brevity, datasets on which all methods achieve a 100% completion rate are omitted. As shown, MASC achieves a 100% completion rate on all datasets except Human. On the Human dataset, multi-query baseline methods frequently fail to complete: the presence of even a single hard query within a batch can cause the entire batch to exceed the time limit, leading to widespread timeouts. These results demonstrate the superior efficiency and reliability of MASC for multi-query subgraph matching across diverse scenarios.

6.2.4 Evaluation on Simple Query Patterns. In practice, many real-world graph queries are small and structurally simple, taking path- or star-shaped forms. We sample path- and star-shaped query graphs with 5 vertices from the data graph. Figure 10 reports the total elapsed time for path- and star-shaped queries. The results show that our approach MASC outperforms competing methods on most evaluated datasets for both query patterns. The observed performance gains can be attributed

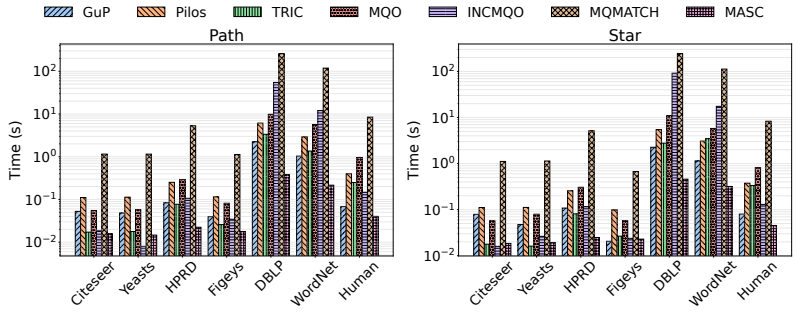


Fig. 10. Elapsed time on path- and star-shaped queries.

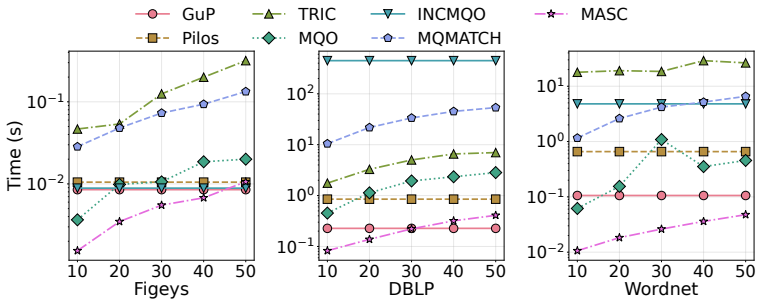


Fig. 11. Average time per query under different batch sizes.

to the fact that simpler query structures make it more likely for similar computations to arise across queries, which enables effective sharing through joint filtering and shared enumeration.

6.3 Evaluation of Parameter Sensitivity

6.3.1 Impact of Batch Size. We study the effect of batch size on multi-query optimization by varying the number of queries grouped and processed together. For each experiment, we use the same set of queries but adjust the batch size to observe its impact on performance. We divide the same set of 300 queries into batches of size 10, 20, 30, 40, and 50, ensuring that each batch contains a mix of similar and dissimilar queries. Figure 11 presents the average query time for different batch sizes for datasets Figeys, DBLP, and Wordnet. Notably, the average completion time remains largely stable, with a slight increase as the batch size grows, indicating the robustness of our approach. This indicates that our method can effectively separate similar and dissimilar queries, maintaining consistent performance regardless of batch size. Overall, our approach consistently outperforms the baselines across all batch sizes.

6.3.2 Impact of Overlap Ratio. To study the effect of query overlap in multi-query optimization, we vary the overlap ratio among queries in each batch. The overlap ratio measures the proportion of shared substructures, which directly affects the potential for computation sharing. We fix all other factors and evaluate performance across overlap sizes 0, 3, 6, 9, and 12 for queries of size 15, and the results are shown in Figure 12. Specifically, an overlap size of 0 corresponds to the case where queries in a batch are selected entirely at random, without any enforced shared substructures. In this setting, our method remains comparable to the best single-query state-of-the-art methods on Figeys, and further achieves clear advantages on DBLP and WordNet, indicating its ability to capture overlaps that are not predefined. As the overlap size increases, the average query time decreases only slightly, rather than significantly as one might expect. For baseline methods such as MQO that always select the maximum common subgraph, i.e., the largest overlap, the runtime does not

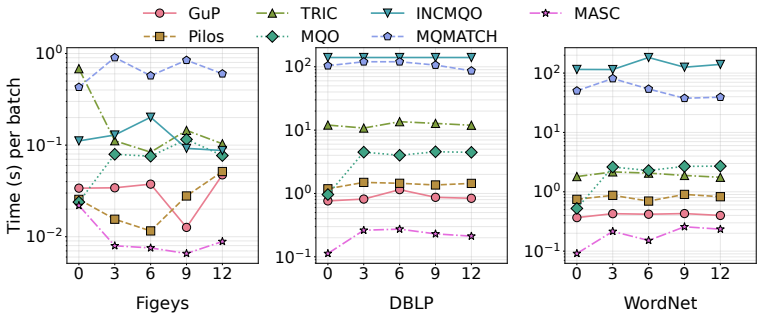


Fig. 12. The elapsed time under different overlap ratios.

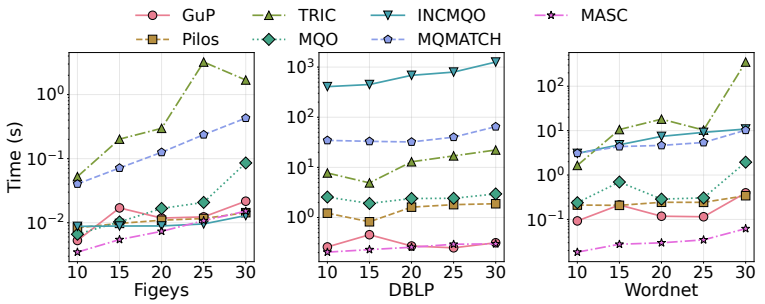


Fig. 13. The elapsed time under different query graph sizes.

decrease significantly either. This observation confirms a key insight of our paper that maximizing overlap does not necessarily improve performance, as shared substructures do not always lead to shared computation. Across all overlap settings, our method consistently outperforms the baselines and maintains the lowest average query time. This demonstrates that our approach is able to capture shared computation when varying query overlaps.

6.4 Evaluation of Scalability

6.4.1 Impact of Query Graph Size. We evaluate the impact of query graph size on algorithmic performance over a wide range of query sizes. Specifically, we consider query graphs with 10–30 vertices and larger query graphs with 32–256 vertices.

For queries with 10–30 vertices, we generate query graphs with 10, 15, 20, 25, and 30 vertices. Each algorithm is evaluated to examine how increasing query size impacts computational efficiency and matching quality. Figure 13 reports the total elapsed time as the query graph size varies on datasets Figeys, DBLP, and Wordnet. As expected, the enumeration time increases as the query size grows, reflecting the greater computational complexity involved. Nevertheless, our approach consistently maintains a clear advantage over competing methods in this setting.

For larger queries, we further extend the evaluation to query graphs with 32, 64, 128, and 256 vertices. Figure 14 reports the total elapsed time on Figeys, DBLP, and WordNet. As query size increases, the execution cost of all methods grows substantially. Overall, MASC remains competitive compared to other baseline approaches, while exhibiting different performance characteristics across datasets. On DBLP, MASC shows performance comparable to GuP for smaller queries and achieves clear advantages for larger queries with 128–256 vertices. In contrast, on WordNet, MASC performs favorably for queries 32–128 vertices, while its advantage diminishes at 256 vertices. These observations reflect the inherent complexity of multi-query processing on large graphs. For very large queries, identifying and exploiting overlaps becomes challenging, as processing large common substructures incurs substantial overhead even with polynomial-time relabeling. The deeper search

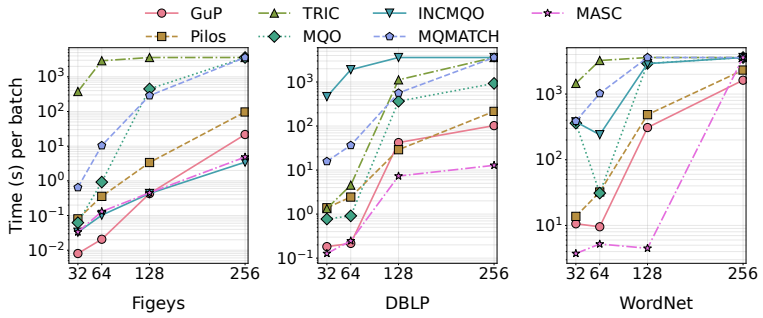


Fig. 14. The elapsed time under larger query graph sizes.

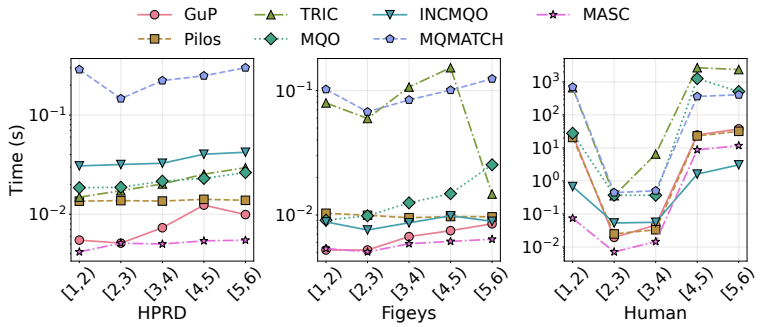


Fig. 15. The elapsed time under different query density.

required may further cause multi-query matching to partially degenerate toward single-query processing when effective overlap is limited. Nevertheless, when large overlap exists, MASC still achieves noticeable performance benefits, leading to the observed performance variability.

6.4.2 Impact of Query Graph Density. To evaluate the impact of query graph density on algorithm performance, we generate query graphs with different average degrees. We use the average degree as a measure of density and create query graphs via random walks with an accept-reject strategy. Specifically, we consider five average degree ranges: [1,2), [2,3), [3,4), [4,5), and [5,6), and generate 300 query graphs for each range. The corresponding results on HPRD, Figeys and Human are summarized in Figure 15. Figure 15 shows that on HPRD and Figeys, all methods exhibit relatively stable runtimes as query density increases, while on Human, runtimes first decrease and then increase with higher degree. This is because query density has two opposing effects: higher degrees would lead to more complex query structures and increased runtime, but also impose stricter constraints that reduce the candidate space after filtering. Across all settings, our method MASC consistently achieves the best and most stable performance, demonstrating strong robustness to varying query densities.

6.4.3 Impact of Data Graph Size. To examine scalability with respect to data graph size, we construct synthetic graphs by enlarging datasets Yeasts, HPRD, and Figeys using EvoGraph [29], as their relatively small base sizes ensure that the experiments on the scaled graphs remain computationally feasible. We apply the scaling factors of 5, 10, 20, 50, and 100. This experimental design enables a systematic evaluation of how the performance of our method and baseline algorithms changes as the data graph size increases. The results are shown in Figure 16. As expected, total elapsed time generally increases with larger data graphs, likely due to the expanded search space associated with larger graph sizes. The rate of increase is similar across all methods, and our approach consistently

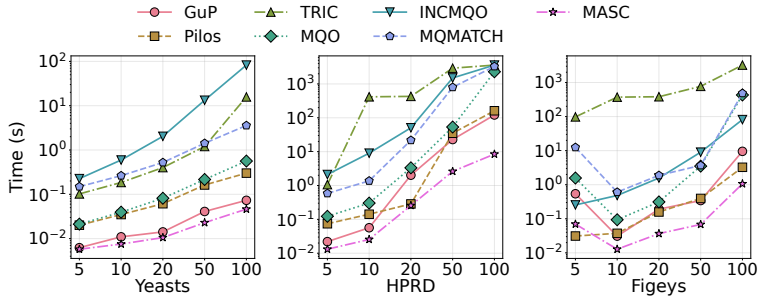


Fig. 16. The elapsed time under different data graph sizes.

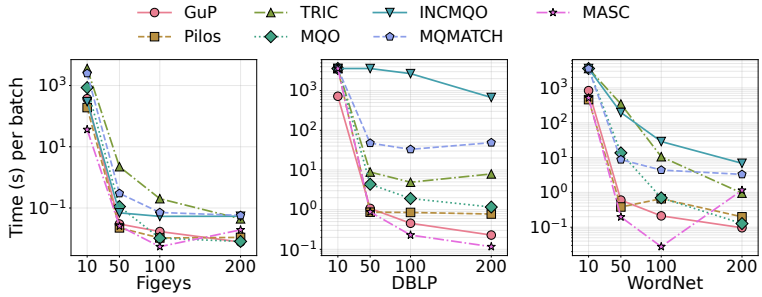


Fig. 17. The elapsed time under different label sizes.

maintains a comparable advantage over the baselines at each scale. These results suggest that our method scales well and preserves its efficiency as the size of data graph grows substantially.

6.4.4 Impact of Label Size. To assess the impact of label size, i.e., the diversity of labels in the dataset, on algorithm performance, we assign data graphs with varying numbers of distinct labels: 10, 50, 100, and 200. By systematically increasing the label size, we can investigate how greater label heterogeneity affects the efficiency of different algorithms. Figure 17 presents the performance results under these varying label size settings on the Figeys, DBLP, and WordNet datasets. As observed, query time decreases as the label size increases. This trend occurs because a larger label set reduces the number of data vertices sharing the same label with any given query node, leading to fewer candidates, a smaller search space, and thus lower overall computational cost. This reduction is initially rapid but gradually slows down, as enlarging the label set yields diminishing returns in shrinking the search space. Across the three figures, especially on WordNet, we observe a trend that the performance advantage of MASC first increases and then decreases as the label size grows. When the label size is small, many query vertices share the same labels, resulting in large candidate sets and substantial overlap among candidates. As the label size increases to 200, filtering becomes highly selective and the search space is already significantly reduced. Consequently, the overhead of maintaining and exploiting overlap information in MASC begins to dominate, reducing its relative advantage. Overall, the results show that our approach consistently achieves superior performance across different label diversities.

6.5 Ablation Study

We conduct an ablation study to evaluate the contribution of each major component in our framework. Specifically, we consider the following variants:

(i) **Base.** Queries are processed independently without any shared computation.

(ii) **Base+F.** The Signature-based Joint Filter is enabled to jointly evaluate identical filtering predicates across queries, while enumeration remains query-specific.

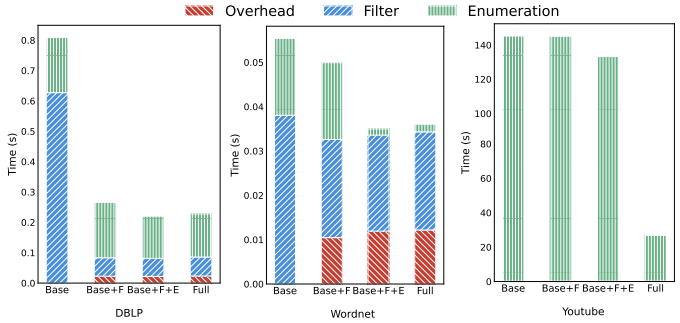


Fig. 18. Ablation Study.

(iii) **Base+F+E.** In addition to joint filtering, the visit-only-once paradigm is enabled to share enumeration over common candidate sets across queries.

(iv) **Full.** All components are enabled, including shared block reuse for exploiting multiple overlapping substructures across queries.

The study is conducted on DBLP, WordNet, and YouTube using queries with 15 vertices. Figure 18 presents stacked bar charts that decompose the total execution time into overall overhead, filtering time, and enumeration time. On DBLP and WordNet, where the overall query processing cost is lower, both the joint filtering and the visit-only-once paradigm provide clear benefits. In particular, on DBLP, comparing **Base** and **Base+F** shows that although capturing equivalent signature introduces additional overhead, the signature-based joint filter significantly reduces the total filtering time. In contrast, on YouTube, where query processing is dominated by long enumeration time, the preprocessing overhead and filtering cost become negligible, and the shared block reuse yields substantial performance improvements.

7 Related work

7.1 Subgraph Matching

Subgraph matching is a fundamental problem with broad applications, and has been extensively studied under various settings, including edge-labeled, vertex-labeled, and unlabeled graphs [2, 4–6, 11, 16, 17, 19, 22–26, 46]. To address the scalability challenges posed by large-scale graphs, researchers developed a range of parallel and distributed algorithms [1, 8, 20, 30, 36, 42, 43, 45, 48].

Most subgraph matching algorithms under the single-threaded setting adopt a *filtering-ordering-enumeration* framework [38, 49]. The filtering phase aims to efficiently prune the search space by establishing candidate sets for each query vertex. Techniques such as neighbor label frequency (NLF) [52], dynamic programming on DAGs (DPiso [11]), and BFS-based filtering (CFL [5]) are widely used, while recent work further improves candidate refinement by propagating local and structural constraints [22, 39, 44]. The ordering phase focuses on minimizing intermediate results and maximizing pruning. Representative strategies include selecting the least frequent label (VF2++ [2]), using candidate set sizes or selectivity-based heuristics (QuickSI [33] and GraphQL [16]), and decomposing queries via core-forest-leaf models based on cost estimation (CFL [5]). For enumeration, numerous optimizations have been proposed to further enhance performance. These include pruning based on repeated failures (KSS [11], GuP [2], and BICE [7]), leveraging repeated successes (VEQ [22]), and postponed extension strategies (Circinus [18], CaLiG [44], and CFL [5]). Collectively, these developments have significantly advanced the efficiency of subgraph matching techniques for many real-world applications.

7.2 Multi-Subgraph Query Optimization

With the growing demand for efficient processing of large-scale graph queries, multi-query optimization in subgraph matching has become a crucial research area. The main objective is to leverage shared computation and common substructures among multiple queries to reduce redundant work and improve overall efficiency.

In relational databases, shared execution has evolved from caching common sub-expressions [35] and synchronizing table scans [10, 14] to fine-grained Tagged Execution [21]. Targeting disjunctive queries (e.g., `A UNION B`), Tagged Execution [21] annotates tuples with membership tags to identify which logical branches they satisfy, thereby eliminating redundant predicate evaluations. However, in contrast to predicate-level reuse in SQL, multi-query optimization on graphs prioritizes structural reuse across multiple subgraph queries.

Representative approaches include MQO [32], which identifies query pairs with large common subgraphs via tri-let (tri-vertex label sequence) grouping, then computes maximum common subgraphs and constructs a Pattern Containment Map to cache and share intermediate results among overlapping queries. vMSQ [13] tackles vertex-constrained multi-subgraph matching by introducing a parallel scheduling algorithm that balances computational load and maximizes computation sharing.

In parallel, continuous multi-query subgraph matching focuses on monitoring multiple queries over dynamic graphs, where the underlying data evolves over time. In this setting, the set of queries is fixed and known in advance. This allows for offline identification of shared substructures, which avoids repeatedly solving the NP-hard maximum common subgraph problem online. In addition, specialized index structures can be constructed to support efficient incremental maintenance as the graph updates. For example, TRIC+ [47] proposes a trie-based clustering method that decomposes queries into shared directed paths and indexes them with tries, minimizing redundant computations. IncMQO [41] merges multiple queries into an annotated query graph and introduces a compact auxiliary data structure to store partial solutions, supporting efficient update processing. MQ-Match [28] further enhances computation sharing by devising a compact CCG index for effective pruning and a matching-tree-based incremental algorithm, ensuring that common substructures among queries are matched only once. Although developed for dynamic graphs, continuous multi-query techniques, especially the computation sharing, also offer valuable insights for advancing traditional multi-query optimization.

8 Conclusion

To enable efficient multi-query subgraph matching, we propose MASC, a novel framework that maximizes shared computation across overlapping queries. Through *signature-based joint filtering*, our approach significantly reduces redundant checks during the filtering stage. Based on shared candidates instead of just shared structure, the *visit-only-once paradigm* ensures that each shared candidate in the data graph is processed only once for all relevant queries during backtracking search. Furthermore, *shared block reuse* exploits multiple overlapping substructures to further amplify shared computation beyond existing methods. Our extensive experiments on real-world datasets show that MASC consistently outperforms state-of-the-art solutions.

Acknowledgments

This work was supported by National Natural Science Foundation of China (No. 62572126), Key Projects of the National Natural Science Foundation of China (No. U23A20496), and Ant Group through CCF-Ant Research Fund.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (oct 2017), 44 pages. doi:10.1145/3129246

[2] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-Based Pruning. *Proc. ACM Manag. Data* 1, 2, Article 167 (jun 2023), 26 pages. doi:10.1145/3589312

[3] Albert-László Barabási and Zoltán N Oltvai. 2004. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics* 5, 2 (Feb. 2004), 101–113.

[4] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1199–1214. doi:10.1145/2882903.2915236

[6] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2018. Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2018), 804–818. doi:10.1109/TPAMI.2017.2696940

[7] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. *Proc. VLDB Endow.* 16, 9 (may 2023), 2186–2198. doi:10.14778/3598581.3598591

[8] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. PARSEC: Parallel subgraph enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 168–178.

[9] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[10] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 526–537. doi:10.14778/2168651.2168654

[11] 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 2019 - Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[12] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 337–348. doi:10.1145/2463676.2465300

[13] Kongzhang Hao and Longbin Lai. 2020. Towards the Scheduling of Vertex-constrained Multi Subgraph Matching Query. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2857–2859. doi:10.1145/3318464.3384409

[14] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 383–394. doi:10.1145/1066157.1066201

[15] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (04 1970), 97–109. arXiv:https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf doi:10.1093/biomet/57.1.97

[16] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 405–418. doi:10.1145/1376616.1376660

[17] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1123–1134.

[18] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1, Article 12 (may 2023), 26 pages. doi:10.1145/3588692

[19] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81. doi:10.1016/j.dam.2018.02.018 Computational Advances in Combinatorial Optimization.

[20] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1695–1698. doi:10.1145/3035918.3056445

[21] Albert Kim and Samuel Madden. 2024. Optimizing Disjunctive Queries with Tagged Execution. *Proc. ACM Manag. Data* 2, 3, Article 158 (May 2024), 25 pages. doi:10.1145/3654961

[22] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 925–937. doi:10.1145/3448016.3457265

[23] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1231–1245. doi:10.1145/2882903.2915209

[24] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *arXiv preprint arXiv:1506.01973* (2015).

[25] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[26] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.

[27] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable Multi-query Optimization for SPARQL. In *2012 IEEE 28th International Conference on Data Engineering*. 666–677. doi:10.1109/ICDE.2012.37

[28] Ziyi Ma, Jianye Yang, Xu Zhou, Guoqing Xiao, Jianhua Wang, Liang Yang, Kenli Li, and Xuemin Lin. 2024. Efficient Multi-Query Oriented Continuous Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3230–3243. doi:10.1109/ICDE60146.2024.00250

[29] Himchan Park and Min-Soo Kim. 2018. EvoGraph: An Effective and Efficient Graph Upscaling Method for Preserving Graph Properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2051–2059. doi:10.1145/3219819.3220123

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

[31] 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 (aug 2018), 1876–1888. doi:10.14778/3229863.3229874

[32] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 121–132. doi:10.14778/3021924.3021929

[33] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (dec 2017), 420–431.

[34] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2018. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (oct 2018), 420–431. doi:10.1145/3164135.3164139

[35] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. doi:10.1145/42201.42203

[36] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. 625–636.

[37] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. Pilos: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 1180–1193. doi:10.1109/ICDE65448.2025.00093

[38] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098. doi:10.1145/3318464.3380581

[39] 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 (oct 2020), 176–188. doi:10.14778/3425879.3425888

[40] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (jan 1976), 31–42. doi:10.1145/321921.321925

[41] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. 2022. Continuous multi-query optimization for subgraph matching over dynamic graphs. *Semantic Web* 13, 4 (2022), 601–622.

[42] Zhaokang Wang, Weiwei Hu, Guowang Chen, Chunfeng Yuan, Rong Gu, and Yihua Huang. 2021. Towards Efficient Distributed Subgraph Enumeration Via Backtracking-Based Framework. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 2953–2969. doi:10.1109/TPDS.2021.3076246

[43] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big graph analytics platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

[44] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (may 2023), 26 pages. doi:10.1145/3588695

[45] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2049–2062. doi:10.1145/3448016.3457237

[46] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528.

[47] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2019. Efficient continuous multi-query processing over graph streams. *arXiv preprint arXiv:1902.05134* (2019).

[48] 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)*. 1031–1043. doi:10.1109/ICDE53745.2022.00082

[49] 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, Article 60 (mar 2024), 29 pages. doi:10.1145/3639315

[50] Yingli Zhou, Yixiang Fang, Wensheng Luo, and Yunming Ye. 2023. Influential community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2047–2060.

[51] Yingli Zhou, Yaodong Su, Youran Sun, Shu Wang, Taotao Wang, Runyuan He, Yongwei Zhang, Sicong Liang, Xilin Liu, Yuchi Ma, and Yixiang Fang. 2026. In-Depth Analysis of Graph-Based RAG in a Unified Framework. *Proc. VLDB Endow.* 18, 13 (Jan. 2026), 5623–5637. doi:10.14778/3773731.3773738

[52] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. 2012. TreeSpan: efficiently computing similarity all-matching. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 529–540. doi:10.1145/2213836.2213896

Received October 2025; revised January 2026; accepted February 2026