Characterizing Parallel Subgraph Matching Performance: A Systematic Study of Interactions, Scalability, and Enumeration

Tao Yu
Fudan University
yut23@m.fudan.edu.cn

Zhijie Zhang
Fudan University
zhangzj22@m.fudan.edu.cn

Weiguo Zheng*
Fudan University
zhengweiguo@fudan.edu.cn

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Qiang Zhou
Ant Group
zhichen.zq@antgroup.com

Chuntao Hong
Ant Group
chuntao.hct@antgroup.com

ABSTRACT

Subgraph matching is a fundamental yet NP-hard problem in graph algorithms. Modern multi-core shared-memory architectures present substantial opportunities to accelerate subgraph matching through parallelism. However, while several parallel subgraph matching algorithms have been proposed, it warrants a systematic empirical study to evaluate: (1) the interaction effect of different parallel strategies, (2) their scalability, (3) underlying performance factors, and (4) the potential for efficiently parallelizing existing sequential algorithms. In this paper, we present a comprehensive study of parallel subgraph matching by analyzing three key components: task splitting, task scheduling, and match enumeration. To investigate their interplay, we evaluate 100 feasible combinations of representative techniques for each component. We further assess scalability across varying thread counts and explore performance variations under diverse query and data graph characteristics.

PVLDB Reference Format:
Tao Yu, Zhijie Zhang, Weiguo Zheng, Jeffrey Xu Yu, Qiang Zhou, and Chuntao Hong. Characterizing Parallel Subgraph Matching Performance: A Systematic Study of Interactions, Scalability, and Enumeration. PVLDB, 19(7): 1659 - 1673, 2026.
doi:10.14778/3801059.3801076

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

1 INTRODUCTION

Subgraph matching, a fundamental graph query operation, has wide applications in areas such as social networks [22, 51], bioinformatics [8], and knowledge graphs [57]. It aims to identify all embeddings in a large data graph that are isomorphic to a query graph. As shown in Figure 1, subgraph matching gets all subgraphs in data graph G that are isomorphic to query graph Q [25, 61, 75, 91].

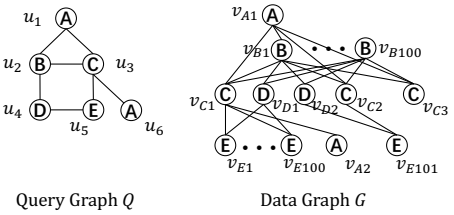


Figure 1: A running example for subgraph matching.

1.1 Motivation

Parallel subgraph matching has gained significant importance with the widespread adoption of multi-core processors [17, 18, 24, 45, 79]. Modern servers with terabyte-scale memory can process large graphs entirely in-memory. This capability is particularly critical for latency-sensitive tasks, such as real-time financial analysis [60]. The performance requirements of such applications are exemplified by benchmarks like the LDBC FinBench [58]. Moreover, algorithmic principles from shared-memory parallelism offer valuable insights for distributed systems, where intra-node efficiency is pivotal to overall performance.

Over the past decade, numerous parallel subgraph matching algorithms for multi-core shared-memory systems have been proposed, e.g., PRI [39], PRS [71], PVF3 [11, 12], CECI [6], MPMatch [34], and LIGHT [70]. These approaches introduce a variety of effective strategies, such as MPMatch's focus on workload estimation and LIGHT's emphasis on matching-tree partitioning. However, there is no comprehensive survey dedicated to parallel subgraph matching on shared-memory architectures, leaving key aspects of shared-memory parallelism underexplored.

As summarized in Table 1, we decompose parallel subgraph matching into three components: *task splitting, task scheduling, and match enumeration*. Task splitting contrasts data graph splitting (abbreviated as GSplit), which partitions the search tree over the data graph, with query graph splitting (abbreviated as QSplit), which decomposes the query into join units whose embeddings are later joined. Task scheduling decides when to split, balancing workload while minimizing synchronization and preserving locality. Match enumeration captures whether backtracking and pruning are local and independent (easier to parallelize) or global and dependency-heavy (requiring extra coordination). Their cross-product yields

*Weiguo Zheng is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801076

Table 1: Techniques to Evaluate.

Component		Techniques	Abbrev.
Task Splitting	Data graph splitting	Upper-one Split [12, 39]	pUO
		Equal-count Split [70, 71]	pEC
		Full Split [82]	pFL
		Workload-aware Split [6, 34]	pWA
	Query graph splitting	Stwig [76]	pSW
		TwinTwig [40]	pTW
		CliqueJoin [41]	pCJ
		SketchTree [94]	pST
Task Scheduling		Static-Assigning [6, 34, 82]	cST
		Busy2Idle-NoneStop [39, 70]	cBNS
		Busy2Idle-DepthStop [71]	cBDS
		Timeout-Assigning [92]	cTO
Match Enumeration		LFTJ [73, 78]	eLFTJ
		GraphQL [32]	eGQL
		CFL [7]	eCFL
		CECI [6]	eCECI
		DPiso [27]	eDPiso
		Circinus [33]	eCIR

100 feasible strategies. Figure 2 shows this design space and component interactions: red lines mark combinations adopted by existing methods, while gray lines highlight substantial unexplored ones.

Through in-depth analysis and experimental evaluation, we aim to address the following key research questions:

- *How to evaluate the effectiveness of a specific technique and its interaction with other strategies?*
  As shown in Table 1, various methods have been proposed to enhance parallel subgraph matching efficiency, each targeting different aspects of parallelization yet often sharing underlying techniques. However, prior studies typically evaluate complete systems, making it difficult to isolate individual techniques' contributions. Moreover, the effectiveness of combining techniques from different methods remains underexplored. Thus, a systematic investigation of individual techniques and their interactions is essential.
- *How is the scalability of these parallel algorithms, and which factors impact the scalability?*
  Existing approaches show improved efficiency as thread count increases. However, the performance of parallel subgraph matching is also influenced by query and data graph characteristics. How these factors affect scalability, and whether linear scalability can consistently be maintained across different scenarios, remains a critical consideration.
- *How does parallel subgraph matching leverage efficient techniques of sequential methods to enhance performance?*
  Sequential subgraph matching methods, such as DPiso [27] and Circinus [33], have achieved notable success in sequential settings by employing backtracking and pruning techniques. A natural idea is to extend them to parallel settings. However, it remains an open question whether parallel algorithms can effectively adapt these techniques, and how such adaptations would impact overall performance, necessitating systematic investigation and empirical validation.

1.2 Contributions

(1) Individual Parallel Techniques and Their Combination. Prior systems for parallel subgraph matching are typically evaluated end to end, making it difficult to pinpoint the effectiveness

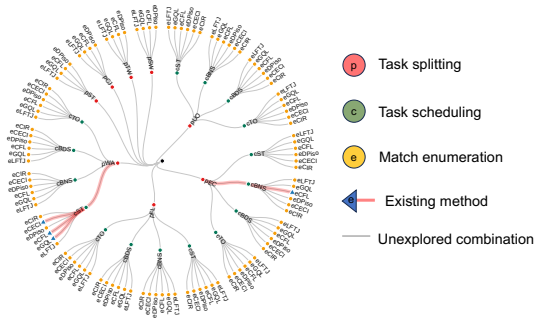


Figure 2: Combinations of different components.

of individual strategies or the impact of combining them. To address this, we factor parallel subgraph matching algorithm into three components—task splitting, task scheduling, and match enumeration. Treating these components as orthogonal dimensions maximizes the design space and reveals intrinsic performance free from implementation bias. Since each component admits multiple alternatives, their feasible pairings lead to a large combinatorial space (100 combinations). Crucially, this decomposition goes beyond theoretical completeness and directly addresses challenges in real-world scenarios like real-time financial analysis over skewed transaction networks. Within this framework: (a) *Task Splitting* determines granularity to effectively decompose high-degree hub nodes to prevent stragglers; (b) *Task Scheduling* dynamically balances highly irregular workloads to ensure low latency; and (c) *Match Enumeration* governs computational efficiency of verifying complex patterns within each task. Building on this structured approach, we systematically isolate individual techniques and evaluates their combinations within a unified implementation.

(2) Scalability and Key Factors. We systematically evaluate scalability, analyzing whether execution time decreases proportionally with thread count or if bottlenecks emerge. Beyond thread scalability, we explore how task splitting, task scheduling, and the parallelizability of algorithmic techniques influence scaling behavior. Additionally, we study query-specific features, including the size of the query graph, the number of labels, and the structural properties of the data graph. These features can significantly affect the behavior of splitting methods, the efficiency of scheduling strategies, and ultimately the overall performance of the algorithm.

(3) Parallelizing Advanced Sequential Solvers. Many pruning techniques have been developed within backtracking enumeration [95] to eliminate redundant exploration across search subtrees, significantly improving sequential efficiency. Despite the success in sequential settings, their parallelizability remains unclear, i.e., the extent to which the advanced pruning techniques used in sequential subgraph-matching methods can run in parallel with minimal adaptation overhead and negligible performance loss. We adapt state-of-the-art subgraph matching algorithms to parallel settings, and analyze both the complexity of adaptation and the efficiency improvements achieved. We further evaluate scalability across diverse scenarios, providing insights into how sequential algorithms and parallel strategies jointly affect performance.

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

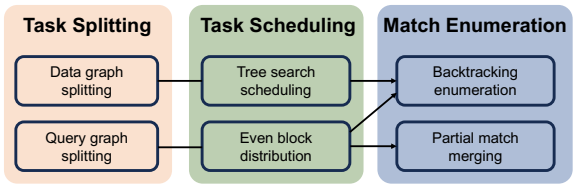


Figure 3: Key components of parallel subgraph matching.

- To the best of our knowledge, we are the first to present a comprehensive experimental study of parallel subgraph matching in a multi-core shared-memory machine.
- We categorize the key factors influencing parallel subgraph matching into three dimensions and explore 100 feasible combinations, implement them in a unified framework, and comprehensively analyze each component's methods.
- We evaluate scalability by investigating how various factors and query-specific features affect parallel performance.
- Through extensive experiments on real-world datasets, we uncover critical factors affecting parallel performance, identify optimization opportunities, and provide insights to guide the design of scalable and efficient parallel subgraph matching approaches.

2 PRELIMINARY

2.1 Problem Definition

In this paper, we focus on an undirected labeled graph $g = (V, E)$, where V is the set of vertices and E the set of edges. For a vertex $u \in V$, $N(u)$ denotes its neighbors, i.e., $N(u) = \{u' \in V \mid (u, u') \in E\}$. For $V' \subseteq V$, $g[V']$ refers to the vertex-induced subgraph of g on V'. A label function L maps each vertex $u \in V$ to a label from a label set Σ, and is shared between both graphs. We naturally extend this notation to vertex sets: for $S \subseteq V$, $N(S) = \bigcup_{v \in S} N(v)$.

Definition 2.1 (Subgraph Isomorphism). Given a query graph $Q = (V_Q, E_Q, \Sigma, L_Q)$ and a data graph $G = (V_G, E_G, \Sigma, L_G)$, Q is subgraph isomorphic to G iff there exists a mapping $f : V_Q \to V_G$ that: (1) $L_Q(u) = L_G(f(u))$ for all $u \in V_Q$. (2) $(f(u_1), f(u_2)) \in E_G$ for all $(u_1, u_2) \in E_Q$, and (3) For all $u_i \in V_Q$, $u_i \neq u_j$, $f(u_i) \neq f(u_j)$.

The mapping function f is also called an embedding of Q in G. $M : I \to V(G)$ denotes a partial matching, where $I \subseteq V(Q)$.

Definition 2.2 (Subgraph Matching). Given a query graph Q and a data graph G, subgraph matching identifies all the embeddings of Q within G.

Problem Statement. Given a query graph Q and a data graph G, **Parallel Subgraph Matching** on a multi-core shared memory machine aims to enumerate all the embeddings of Q within G by utilizing multiple computing cores.

2.2 Components of Parallel Subgraph Matching

In parallel shared-memory settings, the end-to-end efficiency of subgraph matching is constrained not only by algorithmic complexity but by hardware bottlenecks: synchronization contention and workload imbalance. We organize the design space into three dimensions because they collectively govern how the system navigates these constraints, as illustrated in Figure 3.

Table 2: Summary of task splitting

Method	Splitting strategies	Key features
pUO	pop one candidate	low task number
pEC	split candidates evenly	workload balancing
pFL	split all candidates	more tasks
pWA	estimate workload	balanced workload
pSW	star structure	minimal merge rounds
pTW	edge or twig	reduces intermediate results
pCJ	twig or triangle	reduces intermediate results
pST	path tree	reduces redundancy

Task splitting decomposes the overall problem into smaller subtasks to facilitate parallel execution. There are two strategies for task splitting: data graph splitting and query graph splitting. Data graph splitting splits the vertex of data graph, while the latter divides the query graph into several sub-query graphs. These two different splitting strategies also result in different task scheduling strategies and match enumeration methods.

Task scheduling governs workload distribution by dynamically determining when and how to split tasks, ensuring balanced computation across workers. For data graph splitting, task scheduling strategy considers when to split the search tree of backtracking enumeration, whereas for query graph splitting, the task scheduling goal evenly distributes the workloads of merging partial matches.

Match enumeration enumerate complete subgraph matches, with prior research reducing redundant computation. Notably, data graph splitting methods directly output final results, whereas query graph splitting methods generate partial matches for sub-queries and later merge them into final results.

3 TASK SPLITTING

We summarize the characteristics of these task splitting methods as shown in Table 2, and provide a detailed explanation as follows.

3.1 Data Graph Splitting Techniques

Data graph splitting (abbreviated as GSplit) partitions the vertices of the data graph (i.e., the candidate sets) into multiple groups. In subgraph matching, it decomposes the search tree by isolating subtrees as independent subtasks, each responsible for enumerating all possible embeddings of the remaining unmatched vertices. These subtrees originate from distinct matching states, distributed into a task pool for parallel execution across multiple threads.

Definition 3.1 (Matching State). Let M_{pre} be a partial matching prefix and u_{split} represent the next query vertex to be extended. A matching state, denoted by st, is defined as $st = (M_{pre}, LC(u_{split}))$, where $LC(u_{split})$ is local candidates of u_{split}, i.e., the data vertices could be mapped to u_{split} under the partial match M_{pre}.

For data graph splitting techniques, the search space is divided by splitting the local candidate set (i.e., a subset of data vertices) of the next vertex to be mapped. For example, let u denote the query vertex next to match, with its local candidate set $LC(u)$ partitioned

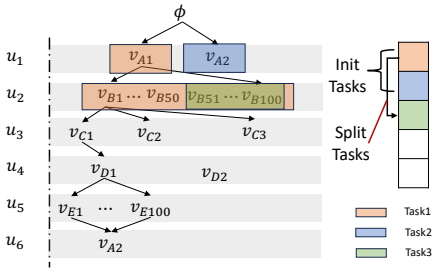
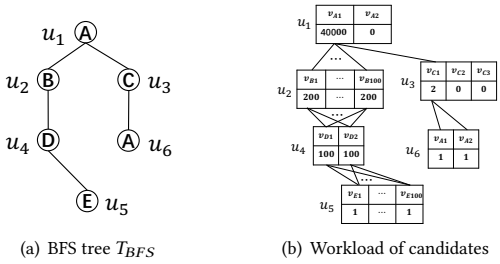


Figure 4: Example of data graph splitting.



(a) BFS tree T_{BFS} (b) Workload of candidates

Figure 5: Example of workload estimation.

into two disjoint subsets, LC_1 and LC_2, such that $LC = LC_1 \cup LC_2$ and $LC_1 \cap LC_2 = \emptyset$. This results in two smaller child search spaces.

Example 3.2. Given Q and G in Figure 1, the splitting process is shown in Figure 4. At depth one, $LC(u_1) = \{v_{A1}, v_{A2}\}$ is divided into $LC_1(u_1) = \{v_{A1}\}$ and $LC_2(u_1) = \{v_{A2}\}$, generating tasks $task_1$ and $task_2$. When $task_1$ adds mapping (u_1, v_{A1}) and reaches depth two, it splits $LC(u_2) = \{v_{B1}, \ldots, v_{B100}\}$ into $LC_3(u_2) = \{v_{B1}, \ldots, v_{B50}\}$ and $LC_4(u_2) = \{v_{B51}, \ldots, v_{B100}\}$. Task $task_3$ is uniquely identified by matching state $st = (\{u_1 : v_{A1}\}, LC_4)$. ■

Building on the general definition of splitting techniques, we now delve into the existing specific approaches.

Upper-one Split [12, 39]. A worker thread selects the closest unexplored candidate vertex to the root and splits it into a new independent task. Because higher-level splits encompass larger sub-search spaces, the resulting tasks typically involve heavier computational workloads and thus require longer execution times.

Equal-count Split [70, 71]. The Equal-count Split strategy partitions the unexplored candidates into two equal-sized subsets at the midpoint, creating a new task. It prioritizes splitting candidates nearest to the root of the search subtree. If no candidates remain at the current level, the algorithm recursively checks the next lower level for viable splits. This ensures that higher-level splits produce two tasks with well-balanced workloads.

Example 3.3. As shown in Figure 4, consider the search task rooted at v_{A1}. When performing a split, the algorithm divides the unexplored candidates nearest to the root $\{v_{B1}, \ldots, v_{B100}\}$ into two equal parts: $\{v_{B1}, \ldots, v_{B50}\}$ and $\{v_{B51}, \ldots, v_{B100}\}$. ■

Full Split [82]. Full Split decomposes the candidate set by creating independent tasks, each corresponding to a unique partial matching state derived from expanding individual candidates. Given a partial matching state M_{prev}, let $ULC(u_{\text{next}})$ denote the unexplored local candidate set for the next query vertex u_{next}. It extends M_{prev} by mapping u_{next} to each candidate $v \in ULC(u_{\text{next}})$, thereby generating $|ULC(u_{\text{next}})|$ distinct tasks for these extended partial matches.

Workload-aware Split [6, 34]. The workload-aware mechanism dynamically decomposes tasks via recursive splitting until their estimated workloads fall below a specified threshold. This process consists of two key phases: (1) Workload estimation: A bottom-up dynamic programming approach calculates the expected workload for each candidate. (2) Recursive decomposition: Beginning with the root vertex's candidate set in the query graph, the mechanism applies Full Split iterations until all resulting tasks meet the threshold. For $v \in LC(u)$, the workload estimation is formulated as: $W_u^v = \prod_{u' \in u.\text{Child}} \sum_{v' \in \mathcal{A}_u^{u'}(v)} W_{u'}^{v'}$, where $\mathcal{A}_u^{u'}(v) = \{v' \in LC(u') \mid (v, v') \in E_G\}$ denotes u's candidates that are adjacent to v ($v \in C(u)$).

When considering workload, the equal-count split strategy naturally adapts by dividing the candidate set into two sets while balancing their total workloads. The algorithm proceeds as follows: (1) Initialize two empty candidate sets with zero accumulated workload; (2) Iteratively allocate the remaining candidate with the largest workload to the set currently having the smaller workload.

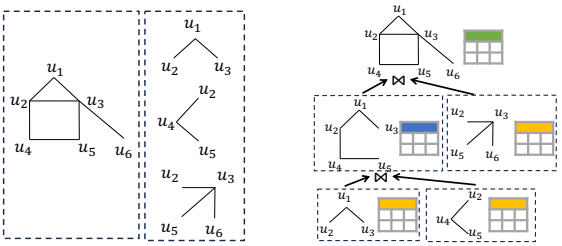
Example 3.4. Figure 5(a) illustrates the BFS tree constructed from the query graph in Figure 1. Figure 5(b) demonstrates the *bottom-up workload estimation*: leaf nodes like $\{u_5, v_{E1}\}$ have $W(\{u_5, v_{E1}\}) = 1$, and intermediate nodes aggregate workloads, e.g., $W(\{u_4, v_{D1}\}) = \sum_{i=1}^{100} W(v_{Ei}) = 100$. For the candidate $(\{u_1, v_{A1}\})$, the workload is computed by multiplying child node workloads: $W(\{u_1, v_{A1}\}) = \left(\sum_{i=1}^{100} W(\{u_2, v_{Bi}\})\right) \times \left(\sum_{i=1}^{3} W(\{u_3, v_{Ci}\})\right) = 40000$. ■

A more accurate workload estimate would require a fine-grained analysis of the query task, such as modeling dependencies among query vertices and their candidates, including connectivity constraints and candidate conflicts [4, 55, 69]. Most existing subgraph matching algorithms rely on heuristic workload estimators, and we follow this practice.

Time and Space Complexity. Let $C_{\max}$ be the maximum candidate set size, and d_Q, d_G the maximum degrees of Q and G. Each task maintains a partial mapping with states of size $|\mathcal{A}| = O(|V(Q)|d_Q^2 d_G)$, dominated by candidate and adjacency structures. Creating one task costs $O(|\mathcal{A}|)$ time and space. Upper-one Split and Equal-count Split generate one subtask per split, incurring $O(|\mathcal{A}|)$ overhead. Full Split spawns one subtask per candidate in $LC(u)$, yielding $O(|LC(u)|)$ tasks and $O(C_{\max}|\mathcal{A}|)$ total cost. Workload-Aware Split precomputes workloads in $O(C_{\max}\, d_G\, |V(Q)|)$ time and $O(C_{\max}|V(Q)|)$ space. During splitting, it greedily assigns candidates in $LC(u)$, incurring $O(|\mathcal{A}|)$ per subtask.

3.2 Query Graph Splitting Techniques

We investigate the performance of QSplit technology (originally proposed for distributed environments) in multi-core settings. It decomposes the query graph into subqueries and offers two key advantages: (1) Partial embeddings of each task unit can be retrieved independently. (2) The subsequent integration of these units with the partial query graph can be performed in parallel.



(a) Query graph decomposition (b) Merging subquery task units

Figure 6: Example of query graph splitting.

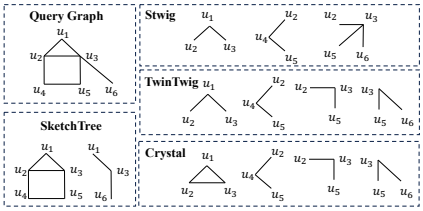


Figure 7: Subqueries of different QSplit techniques.

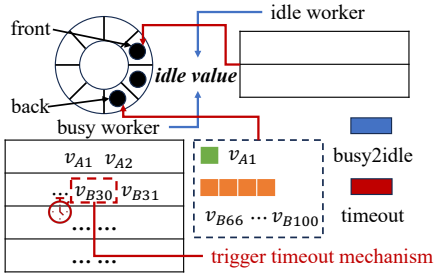


Figure 8: Example of task scheduling techniques.

Table 3: Summary of task scheduling

Method	Trigger Condition	Applicable Scenario
cST	init time	light $\mathcal{A}$
cBNS	work-stealing	default choice
cBDS	add depth limit	moderate workloads
cTO	timeout	embedding-intensive

Definition 3.5 (Subquery Task Unit). Given a query graph Q, a query graph splitting is represented as a set of subqueries $D = (q_0, q_1, \cdots, q_t)$, where each q_i is a *subquery task unit* (that is, a connected subgraph of Q) and the join of all subqueries forms Q.

Query graph splitting adopts a divide-and-conquer strategy comprising three main modules. First, it decomposes the query into independent sub-tasks $D = (p_0, \ldots, p_t)$ and generates a merge tree $\mathcal{J}$ to minimize processing overhead. Next, the system retrieves partial embeddings $R(p_i)$ for each task unit in parallel. These disjoint results are then integrated via a structured bottom-up join sequence defined by $\mathcal{J}$ to reconstruct the complete subgraph matches.

Example 3.6. As shown in Figure 6(a), query graph Q is decomposed into multiple subquery task units. Taking D_1 as an example, its merging process T is depicted in Figure 6(b). Embeddings of each subquery task unit are retrieved via one-hop neighbor access (yellow tables). The intermediate matches are in the blue table. The merging follows a t-round ($t = 2$) bottom-up binary join process, finally retrieving embeddings in the green table. ■

Stwig [76]. Stwig targets distributed memory settings by decomposing queries into a sequence of edge-disjoint stars. To reduce communication, it minimizes the star count and prioritizes high-selectivity edges. Furthermore, it optimizes the selection of a "head Stwig" to manage data locality, determining which partial results are maintained locally versus those aggregated across machines.
TwinTwig [40]. TwinTwig utilizes the MapReduce framework. Firstly, the query graph is divided into a sequence of $t+1$ subgraphs, each being an edge or two adjacent edges. These subgraphs overlap at certain vertices, but do not contain duplicate edges like the 4 units in Figure 7. In the first round, the matching results for $R(P_1) = R(p_0) \bowtie R(p_1)$ are obtained. In each subsequent round i, $R(p_i)$ is calculated and joined with $R(P_{i-1})$ to produce $R(P_i)$. A cost model with A* search is used to find the decomposition sequence minimizing communication overhead.
CliqueJoin [41]. For undirected and unlabeled graphs, the method optimizes the join unit by constructing Crystals from TwinTwigs: if the root has the smallest ID and an edge exists between leaves (e.g., (u_2, u_3) in Figure 7), this edge is included to reduce intermediate results. Furthermore, instead of left-deep joins, it employs a bushy join strategy optimized via dynamic programming. This algorithm iteratively computes the minimal execution cost and optimal splitting strategy for all connected subgraphs.
Sketch Tree [94]. Implemented on a vertex-centric Pregel+ system, this technique decomposes the query graph Q into path sketches (induced subgraphs corresponding to root-to-leaf paths). The sketch tree is defined as a collection of path sketches, where no edges exist between any two path sketches except for the shared edges. For instance, the query in Figure 7 is decomposed into two such sketches sharing a common prefix. Execution uses an exploration-based algorithm with left-deep joins for local matching, while applying result compression during global assembly.
Time and Space Complexity. For query graph splitting, Stwig runs in $O(n^3)$, TwinTwig/CliqueJoin in $O(md_{\max}2^m)$, and Sketch Tree in $O(m)$, where $n = |V(Q)|$, $m = |E(Q)|$, and $d_{\max}$ is the maximum degree. Their costs of space are $O(|V(Q)|)$. However, the matching stage can still generate exponentially many partial embeddings; hence the matching time and space are exponential.

4 TASK SCHEDULING

Task scheduling is critical for parallel efficiency. For partial match merging, we employ standard fork-join parallelism [18, 52] to distribute tasks evenly across threads. We summarize the characteristics of these algorithms in Table 3. For tree search scheduling, we categorize existing strategies into four groups:

Static-Assigning [6, 34, 82]. Static-Assigning predefines a splitting mechanism to split tasks upfront, ensuring simplicity but limiting flexibility [6, 34, 82]. It employs Full Split approach to continuously expand current partial matches that do not satisfy the condition. MPMatch [34] and CECI [6] adopt this strategy in combination with the Workload-aware Split method, iteratively extending partial matches until they fall below a preset workload threshold. BENU [82] utilizes this strategy while specifying a predefined depth.
Busy2Idle-NoneStop [39, 70]. PRI [39] and LIGHT [70] adopt the Busy2Idle-NoneStop strategy, which dynamically splits tasks at any depth to fulfill requests from idle workers. Upon receiving a request, a busy thread partitions the unexplored candidate set $ULC(u_{next})$ of the next vertex, offloading new subtasks to idle threads while continuing its own search. While this maximizes utilization, the unconditional splitting upon every idle request introduces significant synchronization overheads.

Example 4.1. As shown in the blue pipeline in Figure 8, a busy worker finds that there is an idle worker and splits half of its unexplored candidates $ULC(u_{current})$ to the idle worker. During the task-splitting period, both workers are idle. ■

Busy2Idle-DepthStop [71]. PRS [71] employs a busy2idle and depthstop strategy, utilizing a depth threshold α and a candidate size threshold β to regulate task decomposition. Specifically, parallel splitting halts at depth α unless the candidate set size exceeds β. However, optimal parameter tuning is challenging due to complex structural dependencies, and quantifying the scenario-specific cost-benefit trade-off of splitting remains difficult.
Timeout-Assigning [92]. T-DFS [92] proposes a timeout mechanism to automatically detect expensive tasks. If a task is processed by a worker for longer than a user-specified time threshold τ, it is decomposed into smaller tasks and added to a task queue. The red pipeline in Figure 8 illustrates the process of the timeout mechanism. Specifically, a busy worker's backtracking starts from the $top\ depth = 1$ with v_{A1} at time t_0. Before reaching the next depth, the worker retrieves the current time (using $now()$) to calculate the elapsed time of the task, defined as $now() - t_0$. If it exceeds the predefined time threshold τ, the task is partitioned into subtasks, and the newly generated subtasks are enqueued for further processing.

Example 4.2. As shown in Figure 8, the subtree of (v_{A1}, v_{B_1}), $\cdots$, (v_{A1}, v_{B29}) has been fully traversed within τ. However, while extending to v_{B30}, the elapsed time reaches τ. The busy worker then divides $ULC(u_2)$ into two subtasks: $LC_1(u_2) = \{v_{B31}, \cdots, v_{B65}\}$ and $LC_2(u_2) = \{v_{B66}, \cdots, v_{B100}\}$, which are added to the task queue. ■

Time and Space Complexity. Scheduling strategies have the same split cost, while the total cost depends on $\#tasks$. Static-Assigning creates tasks at initialization with time/space $O(\#tasks|\mathcal{A}|)$ (with static depth k, typically $\#tasks = |LC(u)|^k$). cBNS/cBDS and cTO trigger splits dynamically, so $\#tasks$ is not determined in advance and the task queue can be $O(\#tasks|\mathcal{A}|)$ in the worst case.

5 MATCH ENUMERATION

For QSplit paradigm, the corresponding match generation merges partial matches of different subquery graphs with join [41]. Next, we review recent sequential enumeration algorithms as shown in Table 4 and discuss how to effectively parallelize these methods.

Table 4: Summary of match enumeration

Method	Pruning Techniques	Applicable Scenario
eGQL	rule-driven pruning	light workload
eLFTJ / eCFL / eCECI	neighborhood	medium embedding
eDPiso	failing set	large search space
eCIR	freeze candidates	embedding-intensive

5.1 Sequential Backtracking Enumeration

Sequential methods consist of candidate generation, matching ordering, and backtracking [95]. We review representative algorithms including GQL, CFL, CECI, DPiso, LFTJ, and Circinus. For notation, let q_t denote a BFS spanning tree of the query graph, where $u.p$ and $u.c$ represent the parent and child of u in q_t.
GraphQL [32]. The filter process checks the vertex count with each label and performs a pseudo-match within k-hop subgraph of Q and G. The matching order is determined by $|C(u)|$, from smallest to largest. $LC(u)$ is generated by checking edges connecting the $v \in GC(u)$ with the upper neighbors' mapped vertices.
CFL [7]. It first generates an initial candidate set by label and degree constraints and then generates $GC(u)$ by $\bigcup_{v \in GC(u.p)} N(v)$ and refines by check the existence of $e(v, v')$ where $v \in GC(u)$ and $v' \in C(u')(u' \in N^{\varphi}_{-}(u))$. It chooses a root of $\arg\min_u \frac{|v \in V(G)|L(u)=L(v)|}{d(u)}$ and build a BFS tree, and iteratively selects the path with the fewest expected embeddings to add the vertices of the path to the order. It generates LC by: $LC_M(u) \leftarrow \bigcap_{u' \in N^{\varphi}_{-}(u)} \mathcal{A}^u_{u'}(M[u'])$ (a set of edges that can be obtained in O(1) time via auxiliary data structures).
CECI [6]. Its filtering method is similar to CFL, but performs a bottom-up filtering to check the edge existence between $GC(u)$ and $GC(u')$ where $u' \in N^{\varphi}_{+}(u)$. It selects the root of $\arg\min_u \frac{|C(u)|}{d(u)}$ and builds a BFS tree, determining the matching order based on the BFS tree. Like CFL, CECI gets the local candidate set by intersecting the edges corresponding to the mapped vertices of $N^{\varphi}_{-}(u)$. If there are no preceding neighbors, then $LC(u) = GC(u)$.
DPiso [27]. DPiso performs LDF and refines three times alternately between bottom-up and top-down. Each filtering checks edge existence between $v \in GC(N^{\varphi}_{-}(u))$ (or $GC(N^{\varphi}_{+}(u))$) and $GC(u)$. It proposes a dynamic matching order using dynamic programming to choose the branch that tends to have minimum embeddings. DPiso gets local candidates by intersecting edges corresponding to the mapped v of all $N^{\varphi}_{-}(u)$. If no upper neighbors exist, $LC(u) = GC(u)$. DPiso uses failing set pruning: when backtracking to u finds no embedding and the failure is unrelated to u's candidate extension, $LC(u)$ is skipped and backtracking continues to a higher level.
LFTJ [73, 78]. LFTJ is a backtracking strategy that uses set intersection to compute local candidate sets. We utilize the filtering and ordering strategies of CFL to complete this approach. LFTJ employs the same local candidate generation technique as DPiso and recursively maps data graph vertices to query graph vertices.
Circinus [33]. Circinus constructs a vertex cover V_{VC} and identifies a freezing set V_F (where $V_F \cap V_{VC} = \emptyset$). During backtracking, vertices are processed based on their membership: (1) For a non-frozen vertex $u \in V_{VC}$, the algorithm iteratively maps u to $v \in LC(u)$ and refines candidate sets of its backward neighbors $N^{\varphi}_{-}(u)$; (2) For a frozen vertex $u \in V_F$, it defers instantiation, treating the $LC(u)$ as a

Table 5: Datasets.

Dataset	Abbreviation	$\|V\|$	$\|E\|$	d_{avg}
Maayan-Figeys	mf	2,239	6,432	5.7
YeastS	ys	2,361	7,182	6.1
Citeseer	ct	3,279	4,552	2.8
Human	hm	4,674	86,282	36.9
HPRD	hp	9,303	34,998	7.5
WordNet	wn	146,005	656,999	9.0
DBLP	db	317,080	1,049,866	6.6
Twitch	tw	168,114	6,797,557	80.9
Youtube	yt	1,134,890	2,987,624	5.3
Orkut	ok	3,072,441	117,185,083	76.3

monolithic unit. At the leaf level, final embeddings are generated by efficiently enumerating the maintained frozen sets $frozen(u)$.

Time and Space Complexity. Since exact-match enumeration is exponential, we compare the per-step cost of constructing $LC(u)$. LFTJ forms $LC(u)$ via multi-way neighbor intersections. GraphQL checks, for each $v \in GC(u)$, adjacency to already mapped vertices in $BN(u)$, costing $O(d_Q d_G \log d_G)$ per query vertex u. CECI/CFL costs $O(d_Q C_{\max})$, while DPiso adds failing-set updates of $O(|V(Q)|)$. Circinus performs frozen-candidate generation and refinement in $O(d_Q C_{\max})$ and adds space $O(|V(Q)| d_Q C_{\max})$.

5.2 Parallelizing Sequential Methods

Sequential methods rely on backtracking, making them naturally suited for data graph splitting, with each task represented as $st = \{M, u_{next}\}$ (initially $M = \emptyset$, $u_{next} = \varphi[0]$). Standard backtracking supports parallelism via independent subtree exploration. However, advanced optimizations (e.g., failure sharing) rely on cross subtree information, introducing state dependencies that complicate parallelization. Consequently, adapting DPiso and Circinus thus presents dual challenges: (1) splitting must sever cross-subtree dependencies for correctness, and (2) each task requires replicated auxiliary structures, increasing memory consumption.

DPiso computes *matchable* vertices and $LC(u)$ dynamically; forking a task on $LC(u_2)$ means backtracking to u_1 invalidates u_2's context, requiring expensive recomputation. Its failing set pruning relies on bottom-up result aggregation, but splitting fragments this view: failing sets become *unusable* above the splitting depth, remaining valid only *below* it where the thread retains full control.

Circinus freezes certain vertices, deferring their instantiation and refining $LC(u)$ once children are matched. This requires an additional frozen LC data structure, whose size is d times that of the LC structure, i.e. comparable to $\mathcal{A}$. To retain the benefits of the freezing mechanism, we treat frozen candidate sets as atomic units during splitting, prohibiting splits on them.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Datasets. We use 10 widely adopted real-world graphs from different domains as datasets, as shown in Table 5. Following previous work [6, 28, 36, 72, 95], the label sizes are set to 15, 30, 45, and 60 through a random allocation strategy.

Query graphs. Following the method [3, 27, 72, 95], we extract induced subgraphs as query graphs by using a random walk strategy [30]. The sizes of the query graphs are set to 10, 20, 30, 40, and 50, with each query set consisting of 100 query graphs.

Evaluated techniques. Table 1 presents alternative techniques for each component. We perform a comprehensive analysis of the interactions between these components, resulting in 100 feasible algorithm combinations after excluding incompatible pairings.

Metrics. We adopt *EPS (Embeddings Per Second)* [95] as our primary metric to quantify throughput, defined as the total number of output embeddings generated per second. To ensure timely evaluation, we enforce a 30-second timeout for each query. In addition, we calculate the *idle time ratio* as the thread idle time divided by the total time to measure thread utilization.

Experimental environment. All experiments were conducted on a dual-socket server equipped with two Intel Xeon E5-2696 v4 CPUs (44 physical cores total) and 128GB of memory. We implement all techniques on a lightweight shared-memory runtime to avoid framework-induced bias in the component comparison. We use OpenMP only for the match enumeration join process in the query split method. Evaluations spanned thread counts from 1 to 64.

6.2 Impact of Task Splitting Strategies

6.2.1 Data Graph Splitting Strategies. We evaluate the parallel speedup of each splitting strategy (relative to single-threaded execution) when combined with its optimal task scheduling and backtracking configurations across different thread counts. Figure 9 presents the experimental results.

EPS speedup with increasing thread count. Our experiments reveal divergent speedup patterns governed by dataset scale and task granularity. For small graphs (ct and ys), performance peaks at single-threaded execution since enumeration completes within 0.1ms and thread management overheads dominate. For medium-scale graphs (hp, mf, and wn), speedup follows an inverted-U pattern. pEC emerges as optimal due to low-overhead bisection and effective coarse-grained balancing, while pWA's costly workload estimation yields no advantage. Fine-grained strategies (pFL and pUO) degrade significantly with ≥8 threads, as excessive decomposition forces frequent backtracking restarts and task requests that negate parallelism gains. For large graphs (db, hm, tw, yt and ok), all data graph splitting strategies show sustained improvement, yet scaling diverges. db and hm achieve only sublinear speedups (3x–6x at 64 threads) due to task granularity limitations, while dense-output datasets tw and yt demonstrate superior scalability (e.g., tw achieves a 66x linear speedup). On ok, speedup is nearly linear up to 32 threads but plateaus at 64 threads due to cross-socket communication overheads in multi-socket architectures.

Analysis of idle time ratio. Figure 10 illustrates thread idle time ratios. On ct, all methods exhibit excessive idle ratios (>95%) because sub-millisecond task completion times (<0.1ms) are outweighed by thread creation and scheduling overheads. For hp, performance degrades beyond 8 threads, with idle time surging to 70% at 64 threads as scheduling costs dominate. In contrast, the large *tw* graph maintains minimal idle time across all thread counts; its heavy per-task workloads significantly reduce the frequency of task requests, enabling near-linear scalability.

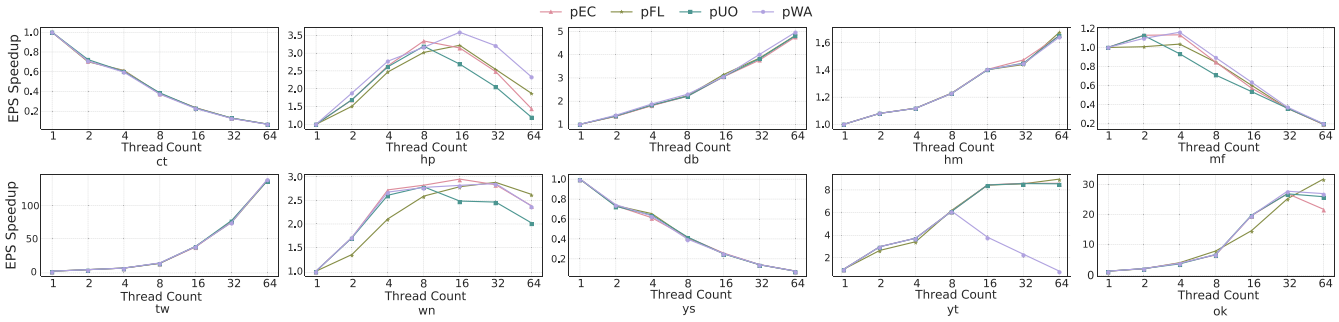


Figure 9: EPS speedup of different data graph splitting strategies vs. the number of threads.

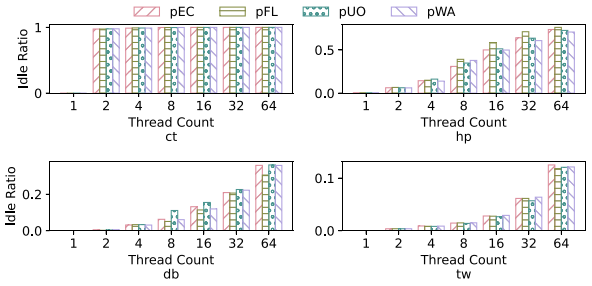


Figure 10: Idle time ratio of data graph splitting strategies.

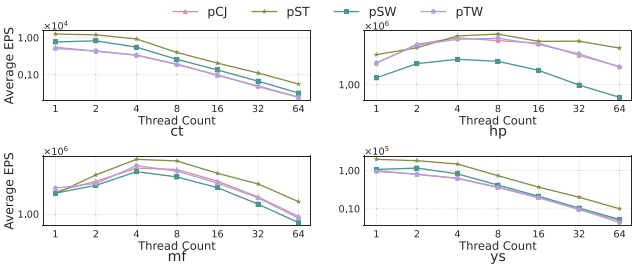


Figure 11: Effect of query graph splitting strategies.

6.2.2 Query Graph Splitting Strategies. We evaluate query graph splitting strategies under their optimal backtracking enumeration across varying thread counts (Figure 11). SketchTree (pST) consistently achieves the best performance by leveraging sketch paths to construct embeddings, effectively reducing merge iterations and intermediate results. TwinTwig (pTW) and CliqueJoin (pCJ) show comparable efficiency, while Stwig (pSW) outperforms them on ct and ys, as its larger join units enable completion in fewer iterations and the limited embeddings on these datasets mitigate pSW's intermediate result explosion [40]. But on larger graphs (hp and mf), the increased intermediate results favor pTW and pCJ over pSW.

Efficiency on ct and ys declines with increasing thread counts, as their small embedding counts make thread initialization and result merging outweigh join parallelization benefits. For hp and mf, EPS shows good speedup with 1–4 threads but declines significantly beyond 8 threads. Quantitative analysis reveals that even on hp and mf, only queries with few embeddings (on the order of 10^4) complete within the time limit. Excessive thread counts introduce overheads that degrade performance.

6.3 Impact of Task Scheduling Techniques

We systematically investigate task scheduling techniques. Figure 12 presents the EPS across different thread counts for each task scheduling technique when paired with its optimal task splitting and backtracking techniques. cBNS (Busy2Idle-NoneStop) performs best on hp, mf, and wn, with cBDS (Busy2Idle-DepthStop) closely approaching it. Notably, cTO (Timeout-Assigning) exhibits lower speedup ratios on hp, mf, and wn, whereas cST (Static-Assigning) consistently underperforms cBNS and cBDS on tw and wn.

EPS speedup with increasing thread count. Scaling patterns diverge across scheduling strategy and dataset characteristics. For small datasets (ct and ys), performance degrades monotonically as parallelization overhead outweighs sub-millisecond execution times. Medium-scale datasets (hp, mf, and wn) exhibit an inverted-U pattern, with efficiency declining beyond optimal thread counts due to oversaturation. Among scheduling strategies, cBNS consistently achieves the highest speedup, outperforming cBDS—indicating that early stopping is unnecessary, as deeper splitting incurs negligible overhead. cST often lags due to workload imbalance in static partitioning, while cTO scales poorly on lightweight workloads (e.g., hp), where its fixed timeout fails to generate enough tasks. On large datasets (db, wn, yt, tw, and ok), abundant solutions and heavy per-task workloads mitigate scheduling idleness, enabling monotonic EPS growth with thread count.

Analysis of idle time ratio. Figure 13(a) shows idle time ratios on ct, hp, db, and tw. On the small dataset ct, all methods exhibit $> 95\%$ idle time as synchronization overheads outweigh the sub-millisecond workloads. On hp, cTO and cST suffer high idle ratios, with cTO exceeding 50% at 8 threads due to its rigid task generation. Conversely, db generally maintains low idleness, cTO degrades at high concurrency. On tw, cST exhibits notable underutilization compared to cBNS and cBDS, which sustain minimal idle time.

6.4 Impact of Backtracking Enumeration

Figure 14 reports the average EPS of various backtracking strategies when paired with its optimal task splitting and task scheduling.

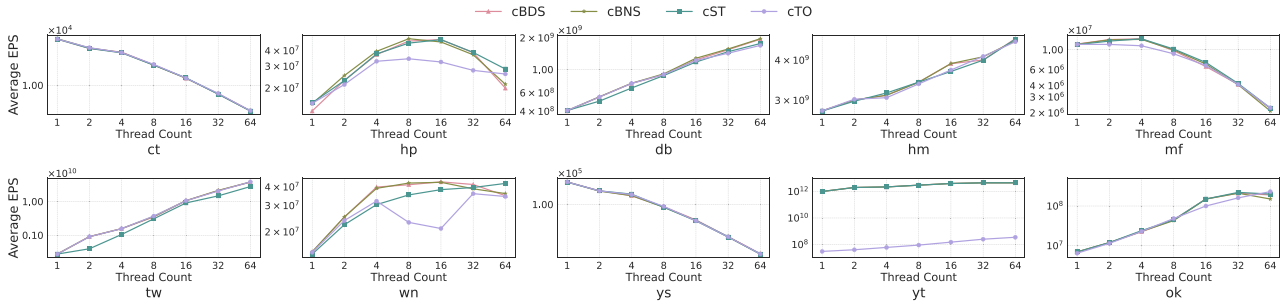
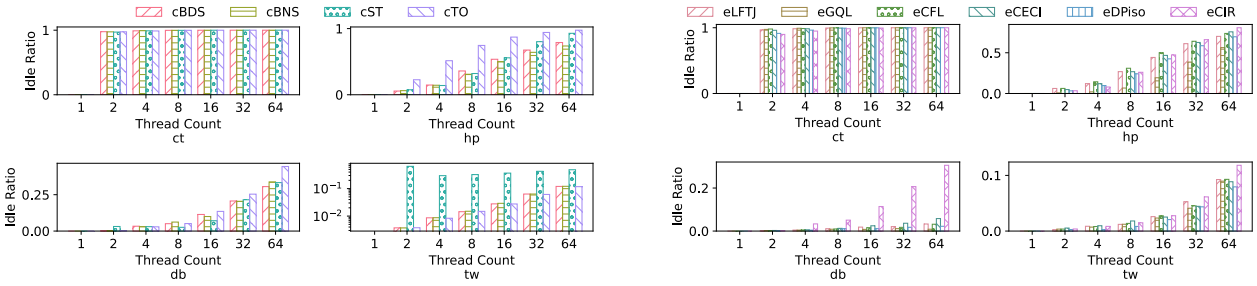


Figure 12: EPS of different task scheduling techniques vs. the number of threads.



(a) Idle time ratio of task scheduling techniques.

(b) Idle time ratio of backtracking techniques.

Figure 13: Idle time ratio of task scheduling and backtracking techniques.

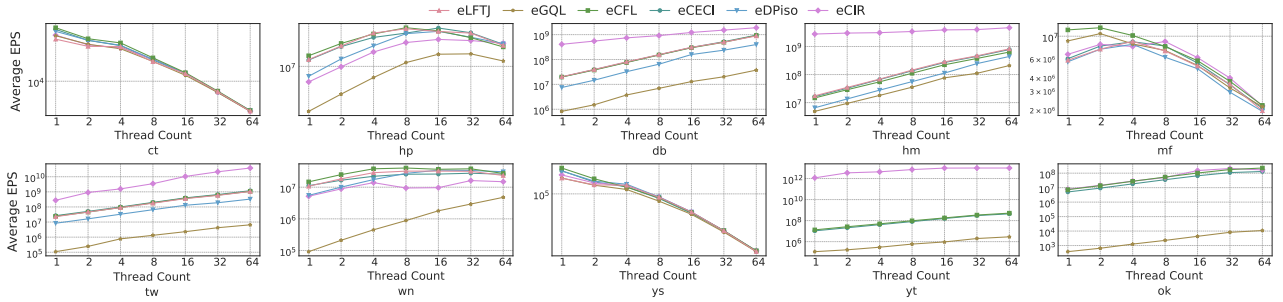


Figure 14: EPS of different backtracking enumeration techniques vs. the number of threads

EPS speedup with increasing thread count. As shown in Figure 14, on small datasets (ct and ys), EPS consistently decreases with thread count, with eCIR performing particularly poorly due to the overhead of its "freezing" technique. On medium datasets (hp), performance inflection points vary; notably, algorithms with higher sequential efficiency (such as eCFL) tend to peak at lower thread counts. On large datasets (db, hm, yt, and tw), eCIR demonstrates a substantial baseline advantage, achieving EPS orders of magnitude higher (10^8). However, its scalability is inconsistent: while it suffers from poor speedup on db and hm (3x–5x) compared to others (40x), it achieves superior speedup (66x) on dense dataset tw.

Analysis of idle time ratio. Figure 13(b) illustrates idle time trends. On small datasets (*ct*), all methods exhibit excessive idle ratios (≈ 1) at 64 threads. On medium datasets (*hp*), scaling stalls at 16 threads as idle ratios climb to 50%. For large datasets, behaviors diverge. On *db*, eCIR suffers from a notably higher idle ratio than competitors, restricting its speedup to 6x (vs. ≈30x for others) despite superior single-threaded performance. Conversely, on *tw*, idle ratios remain universally low (< 10%), enabling strong scalability.

6.5 Performance of All Feasible Combinations

We evaluate EPS scalability for all 100 feasible combinations on hp (Figures 15(a)–(c)), with color-coded trend lines categorizing methods by splitting, scheduling, and backtracking enumeration, respectively. Different algorithm combinations show marked performance variations across graph characteristics and thread counts.

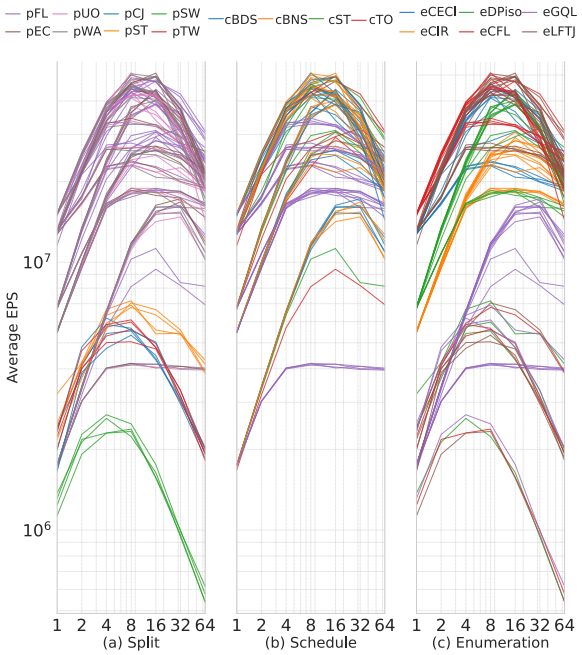


Figure 15: EPS of all feasible technique combinations on hp.

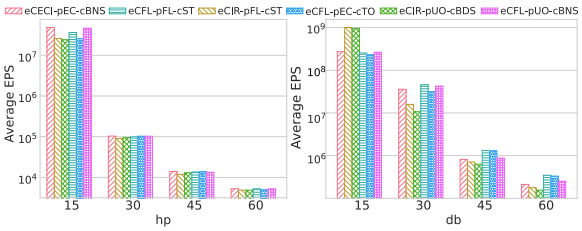


Figure 16: Evaluation of varying $|\Sigma|$.

Figure 15(a) shows that GSplit strategies consistently outperform QSplit counterparts, suggesting GSplit's paradigm is more efficient for shared-memory architectures. In Figure 15(b) (excluding 16 combinations unrelated to scheduling), cTO consistently underperforms while cBDS and cBNS exhibit comparable effectiveness. Figure 15(c) reveals that backtracking enumeration exerts the most substantial impact on parallel efficiency: techniques with significant 1-thread performance gaps (e.g., eCFL vs. eGQL) maintain their ranking regardless of splitting or scheduling choices.

6.6 Effect of Query and Data Graphs

We investigate the effect of query graph and data graphs on two datasets (hp and db) by measuring EPS under 16 threads to evaluate performance across varying query graph sizes ($V(Q)$), label sizes ($|\Sigma|$) and density of query graph. For each configuration, we report six technique combinations ranked 1st, 11th, ..., 51st in performance.

Varying the label size. In Figure 16, the top-performing combinations exhibit an initial EPS increase followed by a decline as label size grows, while lower-ranked combinations demonstrate continuous EPS degradation. This aligns with the observation that larger

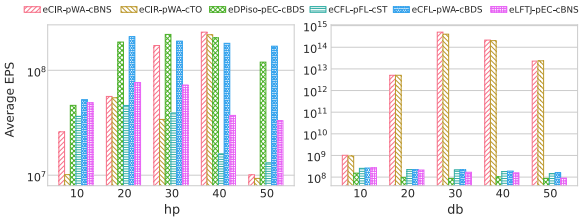


Figure 17: Evaluation of varying $|V(Q)|$.

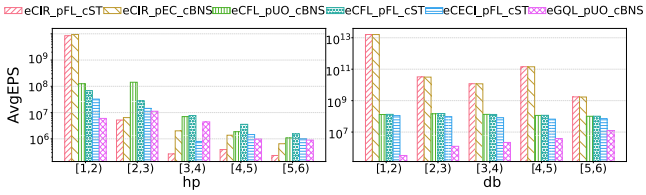


Figure 18: Effect of different query graph densities.

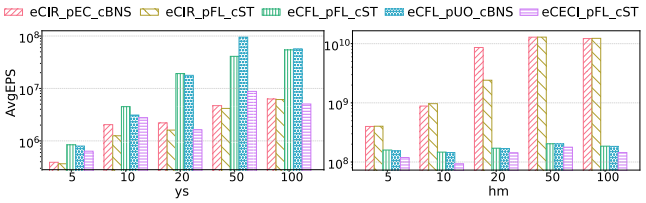


Figure 19: Performance on datasets of different scales.

label sizes reduce embedding counts, decreasing EPS. Notably, the performance gap between the two best-performing combinations and the others widened progressively with increasing label size.

Varying the query size. Figure 17 reveals that with query graph size expansion, the optimal combinations with the eCIR backtracking strategy similarly experience an initial EPS growth followed by decline, whereas other methods show overall decreasing trends. The EPS initially increases and decreases with query size growth. We observe dynamic performance ranking shifts between configurations with fixed $V(Q)$ (varying $|\Sigma|$) versus fixed $|\Sigma|$ (varying $V(Q)$), as different settings yield different optimal combinations.

Varying the density of query graphs. We generate query graphs with varying densities using random walks [95], spanning degree ranges of [1,2), [2,3), ..., [5,6), with 100 graphs per range (Figure 18). On hp, EPS of all methods decreases as query density grows. eCIR leads at degree [1,2), but eCFL surpasses it once the degree reaches 2 or higher. As queries densify, additional constraints make embeddings sparser, and eCIR shows the largest EPS decline. On db, increasing density also reduces EPS, but eCIR maintains a significant advantage, indicating substantial pruning potential.

Varying the size of data graph. To assess scalability, we generated synthetic graphs from HPRD and Human using EvoGraph [54] with scaling factors 5 to 100, each paired with 100 queries. We tested combinations uniformly sampled from the performance ranking 1st, 11th, ..., 41st at 16 threads (Figure 19). While eCFL dominates on the smaller hp, its efficiency drops on the larger hm graph. In

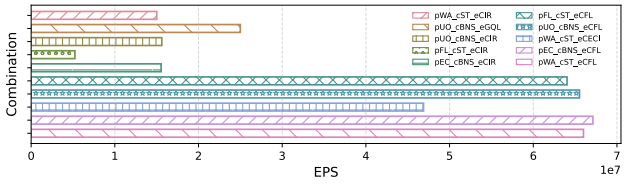


Figure 20: Performance on real queries.

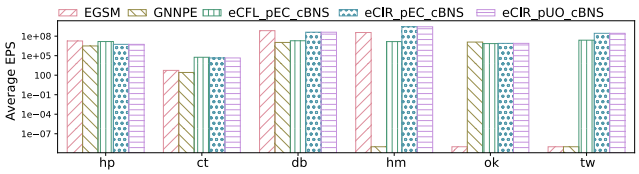


Figure 21: Comparison with GPU and GNN methods.

contrast, eCIR-based combinations (notably `pEC_cBNS_eCIR` and `pFL_cST_eCIR`) exhibit superior scalability, as eCIR's equivalence class becomes increasingly effective with growing graph size: larger graphs exacerbate search redundancy, allowing eCIR to aggressively compress the search space and outperform eCFL.

6.7 Real Query Workload

We further evaluate the system using real-world SPARQL workloads from LC QuAD [77], as shown in Figure 20. For **match enumeration**, eCFL consistently performs best. Since these queries yield sparse embeddings with limited search depths, CIR's equivalence-class computation incurs unnecessary overhead compared to eCFL's lightweight intersection. Regarding **task splitting**, no significant difference exists among methods. However, for **task scheduling**, static scheduling (cST) demonstrates a clear advantage over dynamic work-stealing (cBNS), particularly under pWA and eCFL settings. The total workload is relatively light, so frequent stealing in cBNS introduces overhead that outweighs its load-balancing gains, making upfront execution (cST) more efficient.

6.8 Comparing with GPU and GNN Approaches

We compare the state-of-the-art (SOTA) GPU method EGSM [75] and the SOTA learning-based method GNNPE [88] with three stable CPU PSM combinations (Figure 21). EGSM achieves optimal or near-optimal EPS on db and hm, but degrades significantly on ok and tw due to larger query sizes and severe timeouts. GNNPE performs best on ok but experiences significant degradation on hm and tw; moreover, its offline learning time is excluded from the reported total. Overall, both methods achieve strong EPS in specific cases but lack stability on large graphs, where CPU-based combinations perform better.

6.9 Existing PSM Algorithms

The overall trend of EPS variation with the number of threads for the existing PSM algorithms is shown in Figure 22. On ct, all methods show decreasing EPS as thread overhead far outweighs parallelization benefits. On hp, all methods experience an initial EPS

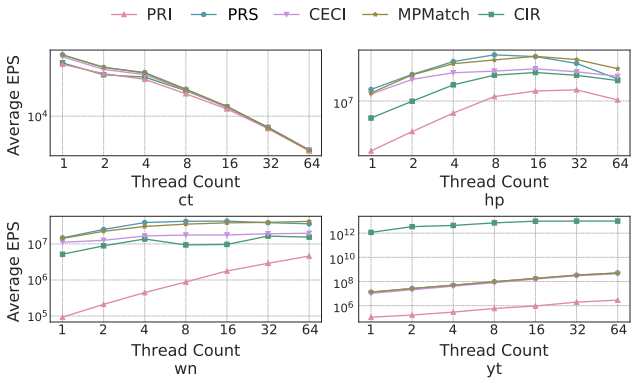


Figure 22: Comparison of existing PSM algorithms.

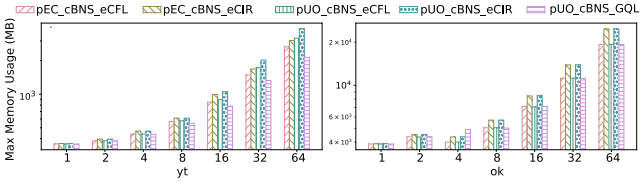


Figure 23: The peak memory cost across datasets.

increase followed by a decline. On wn and yt, which have abundant solutions, EPS growth gradually slows. CIR performs best on graphs with dense results, such as yt, where it exceeds other methods by four orders of magnitude. The PRI method, due to its less advanced local candidate generation technique, performs suboptimally on these graphs. On wn, which has a moderate number of solutions, MPMatch show relatively good efficiency.

6.10 Peak Memory Cost

Figure 23 and Table 6 contrast the peak memory of different strategies. For dynamic combinations (Figure 23), peak memory grows near-monotonically with thread count, as private search states (e.g., partial embeddings, stacks, candidate buffers, and task metadata) need to be replicated for each worker. Static-assigning (Table 6), however, incurs higher but thread-count-independent peak memory, because upfront task materialization dominates the footprint (e.g., ~102 GB on orkut). Overall, these results highlight a critical trade-off: while static-assigning method simplifies scheduling but substantially amplifies memory consumption, requiring careful tuning within the machine's memory budget.

7 RECOMMENDATION AND DISCUSSION

Recommendation. Based on the experimental results, we recommend the following high-performance strategies, as summarized in Figure 24: (1) Among the task splitting strategies, pEC demonstrates superior efficiency and serves as the default choice. For dense datasets with substantial candidate sets, pFL can be effective. pWA is more suitable for scenarios with many small, fragmented tasks, as it helps reduce the generation of redundant tasks. (2) cBNS

Table 6: Peak memory under 1-threaded Static-Assigning.

Graph	CECI (MB)	CFL (MB)	CIR (MB)
youtube	350.09	358.06	361.95
clueweb	2970.14	3382.44	3403.27
orkut	3458.83	3891.76	3908.54
socweibo	20270.73	20183.40	21954.04

is recommended as the default scheduling strategy. When processing dense datasets containing large candidate sets, cTO achieves comparable acceleration performance and is an alternative. (3) eCFL performs well for queries with a moderate number of embeddings, while eCIR is the preferred choice for queries with very dense embeddings. On the tw graph, eCIR achieves two orders of magnitude higher EPS than other enumeration methods. Notably, several previously unexplored combinations outperform the original methods. For example, pWA-cBDS achieves an 11% speedup over the original LIGHT configuration (pEC-cBNS), indicating that the combined strategies across components exhibit greater potential than existing designs, depending on the specific nature of the query.

Discussion. Analyzing various technical combinations, we find that multiple dimensions of strategies collectively determine parallelization efficiency. (1) Experiments show that thread idle time is low (about 10%). Therefore, explicitly optimizing for workload balance during task splitting yields limited gains, as dynamic splitting inherently achieves effective workload distribution. (2) In shared-memory single-machine environments, data graph splitting outperforms query graph splitting due to its more efficient match-generation module, making it the recommended default choice. (3) The backtracking enumeration method plays a critical role in overall performance, as its computational efficiency becomes the key bottleneck in parallelized implementations.

8 RELATED WORK

8.1 Existing Surveys on Subgraph Matching

Prior work studies subgraph matching from complementary angles: iGraph [29] evaluates the I/O efficiency of various indexing techniques; Lee et al. [43], Sun et al. [72], and Zhang et al. [95] systematically compare in-memory filtering, ordering, and backtracking-reduction strategies; Wang et al. [81] categorize incremental maintenance for dynamic matching. Distributed surveys [9, 42] address graph partitioning and communication optimization. However, all these efforts target serial or distributed execution; the interplay between task splitting, scheduling, and enumeration under shared-memory parallelism remains largely unexplored.

8.2 Subgraph Matching & Variants

Sequential CPU-based subgraph matching. Sequential subgraph matching generally follows "filtering–ordering–enumerating" framework [43, 72, 95]. With filtering strategies maturing [6, 7, 27, 28, 37, 67], attention has shifted toward optimizing enumeration to mitigate redundancy. Recent advancements focus on pruning fruitless branches [3, 15, 27], merging equivalent search states [37, 49, 50], and leveraging structural reuse [10, 96].

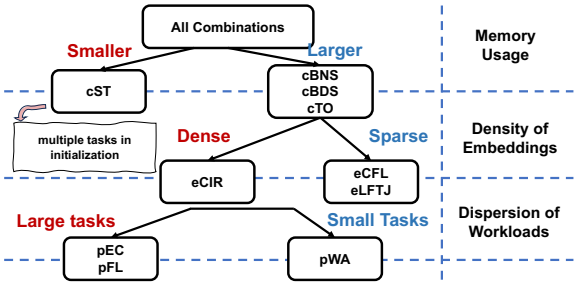


Figure 24: Recommendation.

Continuous Subgraph Matching. Continuous subgraph matching is a pivotal variant of subgraph matching under dynamic graph processing. Early index-free methods [23, 35] expand matches directly from changed edges. To improve scalability, subsequent research has evolved toward partial match materialization [16] and candidate maintenance [38, 53, 93]. Recent works further mitigate combinatorial explosion by optimizing search orders [74] and backtracking search execution [47, 86], culminating in a unified delta query compilation framework [44].

Learning-based Subgraph Matching. Learning-based solutions generally diverge into two streams. *Exact* methods adopt a "learning-assisted" paradigm, utilizing reinforcement learning for query ordering optimization [80] or learned embeddings to accelerate filtering and cost estimation [20, 85, 88]. Conversely, *approximate* methods [1, 5, 19, 46, 48, 63, 64, 90] reformulate the task as representation learning, utilizing probabilistic similarity scores for candidate retrieval [89] rather than strict structural verification.

System-Aware Subgraph Matching. *Disk-based approaches* prune the search space via signatures [66], structural features [14, 83, 84, 97], hierarchical indices [31], and spectral properties [98] to reduce I/O cost. *Distributed systems* partition graphs across clusters [2, 21, 56, 59, 68, 76, 82, 87, 94], adopting a *join-based* [40, 41] or *exploration-based* [62, 65] paradigm. Advanced designs like TriAD [26] and G-Miner [13] introduce asynchronous communication and dynamic scheduling to mitigate network bottlenecks.

9 CONCLUSION

This paper presents a comprehensive empirical investigation into parallel subgraph matching methods, systematically analyzing their performance across multiple critical dimensions. We identify three key factors that significantly influence parallel subgraph matching efficiency and examine their interaction effect by evaluating all feasible combinations of techniques. We assess scalability by measuring speedup across varying thread counts and analyze performance variability under different query and data graph settings.

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] Emily Alsentzer, Samuel G. Finlayson, Michelle M. Li, and Marinka Zitnik. 2020. Subgraph Neural Networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/5bca8566db79f3788be9efd96c9ed70d-Abstract.html

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704. https://doi.org/10.14778/3184470.3184473

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

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

[5] Yunsheng Bai, Hao Ding, Ken Gu, Yizhou Sun, and Wei Wang. 2020. Learning-Based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 3219–3226. https://doi.org/10.1609/AAAI.V34I04.5720

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

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

[8] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, S-7 (2013), S13. https://doi.org/10.1186/1471-2105-14-S7-S13

[9] Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2021. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.* 54, 2, Article 36 (Feb. 2021), 35 pages. https://doi.org/10.1145/3439724

[10] Lisheng Cao, Xiangyang Gou, Lei Zou, and Wenjie Zhang. 2025. MAVIS: Materialized View for Subgraph Matching. *Proc. ACM Manag. Data* 3, 6 (2025), 1–26. https://doi.org/10.1145/3769806

[11] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, and Mario Vento. 2020. Parallel Subgraph Isomorphism on Multi-core Architectures: A Comparison of Four Strategies Based on Tree Search. In *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshops, S+SSPR 2020, Padua, Italy, January 21-22, 2021, Proceedings (Lecture Notes in Computer Science)*, Andrea Torsello, Luca Rossi, Marcello Pelillo, Battista Biggio, and Antonio Robles-Kelly (Eds.), Vol. 12644. Springer, 248–258. https://doi.org/10.1007/978-3-030-73973-7_24

[12] Vincenzo Carletti, Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante. 2019. A Parallel Algorithm for Subgraph Isomorphism. In *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings (Lecture Notes in Computer Science)*, Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia (Eds.), Vol. 11510. Springer, 141–151. https://doi.org/10.1007/978-3-030-20081-7_14

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

[14] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. 2007. FG-Index: Towards Verification-Free Query Processing on Graph Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 857–872.

[15] 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 (2023), 2186–2198. https://doi.org/10.14778/3598581.3598591

[16] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens (Eds.). OpenProceedings.org, 157–168. https://doi.org/10.5441/002/EDBT.2015.15

[17] Melvin E. Conway. 1963. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference* (Las Vegas, Nevada) *(AFIPS '63 (Fall))*. Association for Computing Machinery, New York, NY, USA, 139–146. https://doi.org/10.1145/1463822.1463838

[18] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[19] Khoa D. Doan, Saurav Manchanda, Suchismit Mahapatra, and Chandan K. Reddy. 2021. Interpretable Graph Similarity Computation via Differentiable Optimal Alignment of Node Embeddings. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai (Eds.). ACM, 665–674. https://doi.org/10.1145/3404835.3462960

[20] Chi Thang Duong, Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. 2021. Efficient Streaming Subgraph Isomorphism with Graph Neural Networks. *Proc. VLDB Endow.* 14, 5 (2021), 730–742. https://doi.org/10.14778/3446095.3446097

[21] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. 2025. Divide, Conquer, and Match: A Distributed and Asynchronous Approach for Subgraph Isomorphism. In *2025 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025 - Workshops, Milano, Italy, June 3-7, 2025*. IEEE, 758–761. https://doi.org/10.1109/IPDPSW66978.2025.00120

[22] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, Alin Deutsch (Ed.). ACM, 8–21. https://doi.org/10.1145/2274576.2274578

[23] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Trans. Database Syst.* 38, 3 (2013), 18. https://doi.org/10.1145/2489791

[24] Pawel Gepner and Michal Filip Kowalik. 2006. Multi-Core Processors: New Way to Achieve High System Performance. In *Fifth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2006), 13-17 September 2006, Bialystok, Poland*. IEEE Computer Society, 9–13. https://doi.org/10.1109/PARELEC.2006.54

[25] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2023. Exploiting Reuse for GPU Subgraph Enumeration (Extended Abstract). In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3765–3766. https://doi.org/10.1109/ICDE55515.2023.00309

[26] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 289–300. https://doi.org/10.1145/2588555.2610511

[27] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1429–1446. https://doi.org/10.1145/3299869.3319880

[28] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 337–348. https://doi.org/10.1145/2463676.2465300

[29] Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. 2010. iGraph: A Framework for Comparisons of Disk-Based Graph Indexing Techniques. *Proc. VLDB Endow.* 3, 1 (2010), 449–459. https://doi.org/10.14778/1920841.1920901

[30] W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109. http://www.jstor.org/stable/2334940

[31] H. He and Ambuj K. Singh. 2006. Closure-Tree: An Index Structure for Graph Queries. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49.

[32] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 405–418. https://doi.org/10.1145/1376616.1376660

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

[34] Xin Jin and Longbin Lai. 2019. MPMatch: A Multi-core Parallel Subgraph Matching Algorithm. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*. IEEE, 241–248. https://doi.org/10.1109/ICDEW.2019.000-6

[35] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1695–1698. https:

//doi.org/10.1145/3035918.3056445

[36] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 25–36. https://doi.org/10.5441/002/EDBT.2017.04

[37] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.

[38] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 411–426. https://doi.org/10.1145/3183713.3196917

[39] Raphael Kimmig, Henning Meyerhenke, and Darren Strash. 2017. Shared memory parallel subgraph enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 519–529.

[40] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985. https://doi.org/10.14778/2794367.2794368

[41] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228. https://doi.org/10.14778/3021924.3021937

[42] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112. https://doi.org/10.14778/3339490.3339494

[43] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proc. VLDB Endow.* 6, 2 (2012), 133–144. https://doi.org/10.14778/2535568.2448946

[44] Yukyoung Lee, Kyoungmin Kim, Wonseok Lee, and Wook-Shin Han. 2024. In-depth Analysis of Continuous Subgraph Matching in a Common Delta Query Compilation Framework. *Proc. ACM Manag. Data* 2, 3 (2024), 147. https://doi.org/10.1145/3654950

[45] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 227–242. https://doi.org/10.1145/1640089.1640106

[46] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 3835–3845. http://proceedings.mlr.press/v97/li19d.html

[47] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3324–3337. https://doi.org/10.1109/ICDE60146.2024.00257

[48] Xuanzhou Liu, Lin Zhang, Jiaqi Sun, Yujiu Yang, and Haiqin Yang. 2023. D2Match: Leveraging Deep Learning and Degeneracy for Subgraph Matching. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 22454–22472. https://proceedings.mlr.press/v202/liu23ba.html

[49] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. B$\circledS$*X*: Subgraph Matching with Batch Backtracking Search. *Proc. ACM Manag. Data* 3, 1 (2025), 15:1–15:27. https://doi.org/10.1145/3709665

[50] Yujie Lu, Zhijie Zhang, Weiguo Zheng, and Lei Zou. 2025. Accelerating Subgraph Matching through Fine-grained and Powerful Equivalences. *Proc. VLDB Endow.* 18, 11 (2025), 3896–3909. https://doi.org/10.14778/3749646.3749662

[51] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2018. A Comparative Study of Subgraph Matching Isomorphic Methods in Social Networks. *IEEE Access* 6 (2018), 66621–66631. https://doi.org/10.1109/ACCESS.2018.2875262

[52] Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. 1999. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th international conference on Supercomputing, ICS 1999, Rhodes, Greece, June 20-25, 1999*, Theodore S. Papatheodorou, Mateo Valero, Constantine D. Polychronopoulos, Yoichi Muraoka, and Jesús Labarta (Eds.). ACM, 294–301. https://doi.org/10.1145/305138.305206

[53] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proc. VLDB Endow.* 14, 8 (2021), 1298–1310. https://doi.org/10.14778/3457390.3457395

[54] 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, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 2051–2059. https://doi.org/10.1145/3219819.3220123

[55] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1099–1114. https://doi.org/10.1145/3318464.3389702

[56] Peng Peng, Shengyi Ji, Zhen Tian, Hongbo Jiang, Weiguo Zheng, and Xuecang Zhang. 2023. Locality Sensitive Hashing for Optimizing Subgraph Query Processing in Parallel Computing Systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, 1885–1896. https://doi.org/10.1145/3580305.3599419

[57] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45. https://doi.org/10.1145/1567274.1567278

[58] Shipeng Qi, Bing Tong, Jiatao Hu, Heng Lin, Yue Pang, Wei Yuan, Songlin Lyu, Zhihui Guo, Ke Huang, Xujin Ba, Qiang Yin, Youren Shen, Yan Zhou, Tao Lv, Jia Li, Lei Zou, Yongwei Wu, Gábor Szárnyas, Xiaowei Zhu, Wenguang Chen, and Chuntao Hong. 2025. The LDBC Financial Benchmark: Transaction Workload. *Proc. VLDB Endow.* 18, 9 (2025), 3007–3020. https://doi.org/10.14778/3746405.3746424

[59] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188. https://doi.org/10.14778/3149193.3149198

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

[61] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proc. VLDB Endow.* 8, 5 (2015), 617–628. https://doi.org/10.14778/2735479.2735493

[62] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and Robust Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 12, 11 (2019), 1344–1356. https://doi.org/10.14778/3342263.3342272

[63] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. 2018. Learning Graph Distances with Message Passing Neural Networks. In *24th International Conference on Pattern Recognition, ICPR 2018, Beijing, China, August 20-24, 2018*. IEEE Computer Society, 2239–2244. https://doi.org/10.1109/ICPR.2018.8545310

[64] Indradyumna Roy, Venkata Sai Baba Reddy Velugoti, Soumen Chakrabarti, and Abir De. 2022. Interpretable Neural Subgraph Matching for Graph Retrieval. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 8115–8123. https://doi.org/10.1609/AAAI.V36I7.20784

[65] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 625–636. https://doi.org/10.1145/2588555.2588557

[66] Dennis E. Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. 2002. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis (Eds.). ACM, 39–52. https://doi.org/10.1145/543613.543620

[67] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. Pilos: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 1180–1193. https://doi.org/10.1109/ICDE65448.2025.00093

[68] Yanyan Song, Yuzhou Qin, Wenqi Hao, Pengkai Liu, Jianxin Li, Farhana Murtaza Choudhury, Xin Wang, and Qingpeng Zhang. 2023. Optimizing subgraph matching over distributed knowledge graphs using partial evaluation. *World Wide Web (WWW)* 26, 2 (2023), 751–771. https://doi.org/10.1007/S11280-022-01075-6

[69] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW*

2018, Lyon, France, April 23-27, 2018, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 1043–1052. https://doi.org/10.1145/3178876.3186003

[70] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 232–243.

[71] Shixuan Sun and Qiong Luo. 2018. Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*. IEEE, 42–50. https://doi.org/10.1109/PADSW.2018.8644869

[72] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1083–1098. https://doi.org/10.1145/3318464.3380581

[73] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[74] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. *Proc. VLDB Endow.* 15, 11 (2022), 2415–2427. https://doi.org/10.14778/3551793.3551803

[75] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2 (2023), 181:1–181:26. https://doi.org/10.1145/3589326

[76] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (2012), 788–799. https://doi.org/10.14778/2311906.2311907

[77] Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. 2017. LC-QuAD: A Corpus for Complex Question Answering over Knowledge Graphs. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Claudia d'Amato, Miriam Fernández, Valentina Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin (Eds.), Vol. 10588. Springer, 210–218. https://doi.org/10.1007/978-3-319-68204-4_22

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

[79] Balaji Venu. 2011. Multi-core processors - An overview. *CoRR* abs/1110.3535 (2011). arXiv:1110.3535 http://arxiv.org/abs/1110.3535

[80] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement Learning Based Query Vertex Ordering Model for Subgraph Matching. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 245–258. https://doi.org/10.1109/ICDE53745.2022.00023

[81] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. 2023. A survey of continuous subgraph matching for dynamic graphs. *Knowl. Inf. Syst.* 65, 3 (2023), 945–989. https://doi.org/10.1007/S10115-022-01753-X

[82] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 136–147. https://doi.org/10.1109/ICDE.2019.00021

[83] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-Based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 335–346.

[84] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2005. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.* 30, 4 (2005), 960–993. https://doi.org/10.1145/1114244.1114248

[85] Bin Yang, Zhaonian Zou, and Jianxiong Ye. 2025. GNN-based Anchor Embedding for Efficient Exact Subgraph Matching. arXiv:2502.00031 [cs.SI] https://arxiv.org/abs/2502.00031

[86] 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 (2023), 15:1–15:26. https://doi.org/10.1145/3588695

[87] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2049–2062. https://doi.org/10.1145/3448016.3457237

[88] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding. *Proc. VLDB Endow.* 17, 7 (2024), 1628–1641. https://doi.org/10.14778/3654621.3654630

[89] Rex Ying, Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, and Jure Leskovec. 2020. Neural Subgraph Matching. *CoRR* abs/2007.03092 (2020). arXiv:2007.03092 https://arxiv.org/abs/2007.03092

[90] Xingtong Yu, Zemin Liu, Yuan Fang, and Xinming Zhang. 2023. Learning to Count Isomorphisms with Graph Neural Networks. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, 4845–4853. https://doi.org/10.1609/AAAI.V37I4.25610

[91] Lyuheng Yuan, Akhlaque Ahmad, Da Yan, Jiao Han, Saugat Adhikari, Xiaodong Yu, and Yang Zhou. 2024. G^2-AIMD: A Memory-Efficient Subgraph-Centric Framework for Efficient Subgraph Finding on GPUs. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3164–3177. https://doi.org/10.1109/ICDE60146.2024.00245

[92] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster Depth-First Subgraph Matching on GPUs. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3151–3163. https://doi.org/10.1109/ICDE60146.2024.00244

[93] Qianzhen Zhang, Deke Guo, Xiang Zhao, and Xi Wang. 2021. Continuous matching of evolving patterns over dynamic graph data. *World Wide Web* 24, 3 (2021), 721–745. https://doi.org/10.1007/S11280-020-00860-5

[94] Yuejia Zhang, Weiguo Zheng, Zhijie Zhang, Peng Peng, and Xuecang Zhang. 2022. Hybrid Subgraph Matching Framework Powered by Sketch Tree for Distributed Systems. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1031–1043. https://doi.org/10.1109/ICDE53745.2022.00082

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

[96] Zhijie Zhang and Weiguo Zheng. 2025. BEE: Towards Redundancy Reduction via Block-Separator Decomposition for Subgraph Matching. *Proc. ACM Manag. Data* 3, 4 (2025), 241:1–241:27. https://doi.org/10.1145/3749159

[97] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. 2007. Graph Indexing: Tree + Delta >= Graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. ACM, 938–949. http://www.vldb.org/conf/2007/papers/research/p938-zhao.pdf

[98] L. Zou, L. Chen, Jeffrey Xu Yu, and Y. Lu. 2008. A Novel Spectral Coding in a Large Graph Database. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 181–192.