RapidMatch: A Holistic Approach to Subgraph Query Processing

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

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

Yulin Che
Hong Kong University of Science and Technology
yche@cse.ust.hk

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

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

ABSTRACT

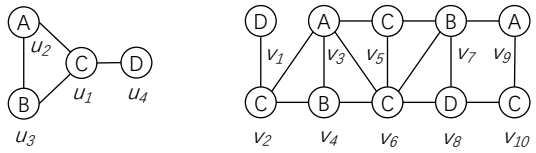
A subgraph query searches for all embeddings in a data graph that are identical to a query graph. Two kinds of algorithms, either graph exploration based or join based, have been developed for processing subgraph queries. Due to algorithmic and implementational differences, join-based systems can handle query graphs of a few vertices efficiently whereas exploration-based approaches typically process up to several tens of vertices in the query graph. In this paper, we first compare these two kinds of methods and prove that the complexity of result enumeration in state-of-the-art exploration-based methods matches that of the worst-case optimal join. Furthermore, we propose RapidMatch, a holistic subgraph query processing framework integrating the two approaches. Specifically, RapidMatch not only runs relational operators such as selections and joins, but also utilizes graph structural information, as in graph exploration, for filtering and join plan generation. Consequently, it outperforms the state of the art in both approaches on a wide range of query workloads.

PVLDB Reference Format:
Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. RapidMatch: A Holistic Approach to Subgraph Query Processing. PVLDB, 14(2): 176-188, 2021.
doi:10.14778/3425879.3425888

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/RapidsAtHKUST/RapidMatch.

1 INTRODUCTION

A *subgraph query* is a basic operation on graphs [29], which finds all embeddings in a data graph G that are identical to a query graph Q. A common type of subgraph query is on labeled graphs G and Q, and G is much larger than Q. Consider the example query graph and data graph in Figure 1, $\{(u_1, v_2), (u_2, v_3), (u_3, v_4), (u_4, v_1)\}$ is an embedding of the query graph in the data graph, or a *match*. Subgraph query processing has been studied in a variety of work (e.g., [1, 23, 33, 37, 40, 42]). In this paper, we categorize existing work into *exploration-based* and *join-based* methods based on their modeling and implementation choices, and propose RapidMatch, a holistic approach to the problem. Table 1 lists a summary of characteristics of existing work as well as our RapidMatch.



(a) Query graph Q. **(b) Data graph G.**

Figure 1: Example query graph and data graph.

The exploration-based algorithms [6, 7, 12, 14, 15, 28, 33, 40, 42] adopt the backtracking framework proposed by Ullmann [37], which iteratively extends intermediate results by mapping *query vertices* (i.e., $V(Q)$) to *data vertices* (i.e., $V(G)$) along a *matching order* (i.e., a sequence of $V(Q)$) to find all matches. In order to reduce intermediate results, the latest algorithms such as CFLMatch [7], CECI [6] and DP-iso[12] execute a query with the *preprocessing-enumeration* paradigm. In particular, they first generate a candidate vertex set for each query vertex with specified pruning methods. Then, they optimize the matching order based on the statistics of candidates with greedy methods because they typically process up to several tens of vertices in the query graph. Finally, they enumerate results along the matching order over candidates. We call such query graphs of tens of vertices *large queries*. These large queries are common in social network analysis [41], computer aided design [10], and protein interaction understanding [40].

In contrast, the join-based algorithms [1, 23] model a subgraph query as a relational query and evaluate the query with relational operators such as selections and joins. The classical relational systems such as MonetDB and PostgreSQL implement a subgraph query as a sequence of pair-wise join operations, which can be viewed as extending intermediate results by an edge at each step. Consequently, the intermediate results can be more than the maximum output size $|OUT|$ of a query. Recently, the development of the *worst-case optimal join* (WCOJ) changes the landscape because its running time matches $|OUT|$ [24]. The previous research [4, 26] shows that the latest systems utilizing WCOJ significantly outperform the classical relational systems as well as native graph databases such as Neo4j. Recent join-based methods such as EmptyHeaded [1] and Graphflow [23] adopt the *direct-enumeration* paradigm, which directly enumerate results on data graphs, to be

Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425888

Table 1: A comparison of representative subgraph query processing algorithms.

Model	Representative Algorithms	Category of Techniques						
		Filter Candidates/Relation		Generate Matching Order/Join Plan			Optimize Data Layout	Accelerate Enumeration Procedure
		Method	Time Complexity	Cost Model	Plan Space	Time Complexity		
Exploration	CFLMatch[7]	Native	$O(\|E(Q)\| \times \|E(G)\|)$	Cardinality Estimation	Greedy	$O(\|V(Q)\|^2 \times \|E(G)\|)$	Tree	N/A
	DP-iso[12]	Native	$O(\|E(Q)\| \times \|E(G)\|)$	Cardinality Estimation	Greedy	$O(\|V(Q)\|^2 \times \|E(G)\|)$	Adjacency List	Failing Set Pruning
Join	EmptyHeaded[1]	Selection	$O(\|E(Q)\| \times \|E(G)\|)$	Cardinality Estimation	Optimal	Exponential to $\|V(Q)\|$	Trie	Set Intersection
	Graphflow[23]	Selection	$O(\|E(Q)\| \times \|E(G)\|)$	Cardinality Estimation	Optimal	Exponential to $\|V(Q)\|$	Adjacency List	Intersection Caching
	RapidMatch	Selection Semi-join	$O(\|E(Q)\| \times \|E(G)\|)$	Query Graph Density	Greedy	$O(\alpha(Q) + \|V(Q)\|^2)$ $\alpha(.)$ is the cost of nucleus decomposition	Encoded Trie	Failing Set Pruning Intersection Caching Set Intersection

easily integrated into database systems. They generate join plans by exhaustively searching the plan space as they target at query graphs of a few vertices. We call such query graphs *small queries*. Such small queries are commonly seen in detecting cycles in transaction networks to alert fraudulent activities, and searching clique/clique-like structures in social networks for recommendation [23].

Recently, the researchers on join-based methods [3, 23] find that WCOJ essentially corresponds to extending intermediate results by a vertex at each step in the graph exploration. We further find that the efficiency of current algorithms suffers either on small or large queries, due to their algorithmic design and implementation choices. The exploration-based algorithms lack optimizations on the enumeration methods [7, 12], so they are slow on small queries. In contrast, the join-based methods cannot handle large queries because (1) the number of join plans is exponential so it is prohibitively expensive to enumerate all plans of large queries [1]; and (2) the filtering methods simply based on vertex labels can result in a large number of false candidates [1, 23].

In this paper, we propose to study the two kinds of algorithms and design a holistic framework that takes advantage of both of them. Specifically, we compare them with respect to the time complexity and prove that the worst-case complexity of result enumeration in exploration-based methods matches that of WCOJ. Moreover, our detailed analysis shows that there is no fundamental difference between the result enumeration of exploration-based methods and join-based even though the latter is implemented with relational operators and the former is not. Therefore, a promising approach would be to adopt a join-based method and improve the join plan generation and execution for both small and large queries.

We propose a join-based subgraph query processing framework called **RapidMatch**, which utilizes graph structural information, as in graph exploration, for filtering and join plan generation. First, given Q and G, the *RelationFilter* component builds relations based on vertex labels, and then filters these relations with semi-joins to remove data edges that will not appear in results. Second, *JoinPlanGenerator* generates a join plan based on the statistics obtained in *RelationFilter* and the *nuclei forests* [31] of Q. Third, we develop *RelationEncoder* to optimize the relation data layout based on the join plan to accelerate the subsequent enumeration. We further extend a number of optimizations in query plan execution in *ResultEnumerator*, such as advanced set intersection methods [1, 13], the intersection caching [23] and the failing set pruning [12]. Our experiments show that RapidMatch outperforms the state of the art on both exploration and join based methods on a wide range of workloads. In summary, we make the following contributions.

- We study the exploration-based and join-based methods and bridge the gap between them.
- We propose RapidMatch, a join-based subgraph query processing engine that can efficiently evaluate both small and large queries.
- We design a relation filter based on semi-joins to reduce the cardinalities of relations. This filter maps state-of-the-art pruning techniques from exploration-based methods to join-based methods.
- We propose a join plan generator based on the nucleus decomposition of the query graph to reduce the search space of the result enumeration.
- We conduct extensive experiments with various kinds of workloads and demonstrate that RapidMatch outperforms both the state-of-the-art exploration-based and join-based approaches.

Paper Organization. Section 2 introduces the background. Section 3 compares the exploration-based method with the join-based. Section 4 gives an overview of RapidMatch. Sections 5, 6 and 7 present the relation filter, the join plan generator, and other implementation details such as relation encoder, respectively. We present the experiment results in Section 8 and conclude in Section 9.

2 PRELIMINARY AND RELATED WORK

2.1 Preliminary

This paper focuses on the undirected vertex-labeled graph $g = (V, E)$ where V is a set of vertices and E is a set of edges. For unlabeled graphs, we refer readers to recent studies [11, 20, 21, 34]. Given $u \in V$, $N(u)$ denotes the neighbors of u, i.e., $\{u' | e(u, u') \in E\}$. $d(u)$ denotes the degree of u, i.e., $|N(u)|$. L is a label function associating a vertex to a label in a label set Σ. Given $V' \subseteq V$, $g[V']$ is the vertex-induced subgraph of g on V'. Q and G denote the query graph and the data graph, respectively. They share the same label function L, and Q is connected. We call vertices and edges of Q (resp. G) query vertices and query edges (resp. data vertices and data edges), respectively. Q_C denotes the 2-core (Definition 2.1) of Q. Q_F is equal to $Q - Q_C$, i.e., $E(Q_F) = E(Q) - E(Q_C)$ and $V(Q_F)$ is the set of vertices incident to edges in $E(Q_F)$. Based on Definition 2.1, Q_C is connected, and Q_F is a set of trees. We call Q_C and Q_F the *core structure* and *forest* of Q, respectively. We summarize the frequently used notations in Table 2.

Definition 2.1. [32] A k-core of g is a maximal connected subgraph g' of g each vertex of which has at least degree k.

In the following, we briefly review the nucleus decomposition, the relational operators and the full reducer, which are building bricks in our proposed techniques.

Table 2: Notations.

Notations	Descriptions
g, Q, G and $\mathcal{Q}$	graph, query graph, data graph and hypergraph
$V, E, \mathcal{E}$ and Σ	vertex set, edge set, hyperedge set and label set
$d(u), L(u)$ and $N(u)$	degree, label and neighbors of u
e	edge or hyperedge
R_e or $R(u, u')$	relations corresponding to $e(u, u')$
$R(u : v, u')$	the neighbors of v in $\pi_{\{u'\}}R(u, u')$
$C(u)$ and $\mathcal{A}$	candidate vertex set of u and auxiliary structure
$\mathcal{A}^u_{u'}(v)$	neighbors of v in $C(u')$ where $v \in C(u)$
δ and φ	filtering order and matching order
$N^{\varphi}_{+}(u)$ ($N^{\varphi}_{-}(u)$)	backward (forward) neighbors of u given φ
$\bowtie$, $\ltimes$ and π	join, semi-join, and project
Q_C and Q_F	core structure and forest of Q
$\mathcal{T}_{r,s}$	r, s nuclei forest of Q

Nucleus Decomposition. Nucleus decomposition generalizes the k-core [32] and k-truss [8] decomposition and finds dense subgraphs at different levels of hierarchy [30]. We use K_r to denote an r-clique, and define the *nucleus* and the *nuclei forest* of a graph g in Definition 2.2. Intuitively, a k-(r, s) nucleus is a connected subgraph of g that satisfies the density and connectivity constraints and the nuclei forest $\mathcal{T}_{r,s}$ describes the hierarchies based on the (r, s) nucleus containment relationship. A k-$(1, 2)$ nucleus is exactly a k-core, and a k-$(2, 3)$ nucleus is a k-truss community [16], which requires the triangle connectivity in addition to the definition of k-truss. Efficient algorithms have been proposed to compute k-core and k-truss in $O(|E(g)|)$ and $O(|K_3(g)|) = O(|E(g)|^{1.5})$ time, respectively [30].

Definition 2.2. [31] Let k, r, s be positive integers where $r < s$, and $\mathcal{S}$ be a set of K_ss in g.

- $K_r(\mathcal{S})$ is the set of K_rs contained in some $S \in \mathcal{S}$.
- The number of $S \in \mathcal{S}$ containing $H \in K_r(\mathcal{S})$ is the $\mathcal{S}$-degree of H.
- Two K_rs H, H' are $\mathcal{S}$-connected if there exists a sequence $H = H_1, H_2, ..., H_i, ..., H_x = H'$ in $K_r(\mathcal{S})$ such that for each i, some $S \in \mathcal{S}$ contains $H_i \cup H_{i+1}$.
- A k-(r, s) nucleus is the maximal union $\mathcal{S}$ of K_ss such that (1) $\mathcal{S}$-degree of any $H \in K_r(\mathcal{S})$ is at least k; and (2) Any $H, H' \in K_r(\mathcal{S})$ are $\mathcal{S}$-connected.
- The nuclei forest $\mathcal{T}_{r,s}$ is a set of trees in which nodes are (r, s) nucleus and the parent of a nucleus is the smallest (by cardinality) other nucleus containing it.

Relational Operators. The projection $\pi_I R$ selects the columns in I from R. A natural join $R \bowtie R'$ is the set of all combinations of tuples in R and R' that are equal on their common attributes. A left semi-join $R \ltimes R'$ is the set of all tuples in R each of which is equal to a tuple in R' on their common attributes.

A multi-way natural join can be represented by a *hypergraph* $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of hyperedges that are subsets of $\mathcal{V}$. Attributes and relations in the join correspond to vertices and hyperedges, respectively. In this paper, the join is interchangeably denoted by $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{Q} = \bowtie_{e \in \mathcal{E}} R_e$ where R_e is the relation corresponding to the hyperedge e.

Full Reducer. Given $\mathcal{Q}$, *dangling tuples* are the tuples in relations that do not appear in any join results. A *full reducer* is a finite sequence of semi-joins that can remove all dangling tuples in relations of $\mathcal{Q}$. $\mathcal{Q}$ is an acyclic query if and only if it has a full reducer [2]. When $\mathcal{Q}$ coincides with a simple graph Q, $\mathcal{Q}$ is acyclic if Q has no cycles. Algorithm 1 describes the full reducer, which takes an acyclic $\mathcal{Q}$ as the input and removes all dangling tuples. It utilizes $2 \times (|\mathcal{E}(\mathcal{Q})| - 1)$ semi-joins in total to eliminate dangling tuples.

Algorithm 1: FullReducer(an acyclic join $\bowtie_{e \in \mathcal{E}} R_e$)

Input: an acyclic query $\mathcal{Q} = \bowtie_{e \in \mathcal{E}} R_e$;
Output: $\mathcal{Q} = \bowtie_{e \in \mathcal{E}} R_e$ where R_e has no dangling tuples;
1 **if** $|\mathcal{E}| = 1$ **then return**;
2 $e \leftarrow$ an edge in $\mathcal{E}$ whose one end vertex is a leaf vertex of $\mathcal{Q}$;
3 $e' \leftarrow$ an edge in $\mathcal{E}$ sharing one end vertex with e ($e \neq e'$);
4 $R_{e'} \leftarrow R_{e'} \ltimes R_e$;
5 FullReducer($\bowtie_{e'' \in \mathcal{E} - \{e\}} R_{e''}$);
6 $R_e \leftarrow R_e \ltimes R_{e'}$;

2.2 Subgraph Query

Subgraph queries can be defined based on either: subgraph isomorphisms denoted by *ISO* or subgraph homomorphisms denoted by *HOM*. Their definitions are given in Definition 2.3. The only difference between the two is that ISO uses a injective function and HOM uses a mapping.

Definition 2.3. Subgraph Homomorphism (resp. Isomorphism): Given $g = (V, E)$ and $g' = (V', E')$, a subgraph homomorphism (resp. isomorphism) is a **mapping** (resp. **injective function**) $f : V \rightarrow V'$ such that (1) $\forall u \in V, L(u) = L(f(u))$; and (2) $\forall e(u, u') \in E, e(f(u), f(u')) \in E'$.

Exploration-based algorithms find all ISOs from Q to G, whereas join-based methods enumerate all HOMs. The HOM allows that a result contains duplicate data vertices, whereas the ISO does not. Therefore, an ISO must be an HOM, but not vice versa. A special case is when each query vertex has a distinct label, an HOM is an ISO because of the constraint on labels in the definition. Nevertheless, deciding whether there exists an ISO or an HOM is NP-complete [9]. We find that these two kinds of algorithms can enumerate the same results with simple modifications (see Sections 2.3 and 2.4).

2.3 Exploration-Based Algorithms

According to the execution paradigm, the exploration-based methods can be classified into three categories. The first category of algorithms such as QuickSI [33] follow the *direct-enumeration* paradigm, which directly explores G to enumerate matches. The second kind of algorithms such as GADDI [40], SPath [42] and SG-Match [28] utilize the *indexing-enumeration* framework, which constructs indexes on G and evaluates all queries with the assistance of the indexes. The third group of algorithms such as TurboIso [14], CFLMatch [7], CECI [6] and DP-iso [12] adopt the *preprocessing-enumeration* framework. Previous performance studies show that (1) the indexing-enumeration methods have severe scalability issues due to the index construction [18, 22, 35]; and (2) the preprocessing-enumeration methods generally perform the best among them [36]. Therefore, this paper uses the preprocessing-enumeration.

Algorithm 2 presents a sketch of most exploration-based algorithms [6, 7, 12, 14, 15, 28, 33, 37, 40, 42]. Line 1 first constructs a candidate vertex set $C(u) = \{v | v \in V(G) \wedge L(v) = L(u)\}$ for each $u \in V(Q)$, and then prunes $C(u)$ based on Proposition 2.4 [7, 12]. After that, it builds an auxiliary structure $\mathcal{A}$ maintaining data edges between candidate vertex sets. Given $e(u, u') \in E(Q)$, $\mathcal{A}^u_{u'}(v) = N(v) \cap C(u')$, i.e., the neighbors of $v \in C(u)$ in $C(u')$.

Algorithm 2: Exploration-Based Method

```
Input: a query graph Q and a data graph G;
Output: all subgraph isomorphisms from Q to G;
1 C, 𝒜 ← build candidate vertex sets and auxiliary structures;
2 φ ← generate a matching order;
3 Enumerate(C, 𝒜, φ, {}, 1);
4 Procedure Enumerate(C, 𝒜, φ, M, i)
5     if i = |φ| + 1 then Output M, return;
6     u ← φ[i], C_M(u) ← ⋂_{u'∈N_+^φ(u)} 𝒜_u^{u'}(M[u']);
7     foreach v ∈ C_M(u) do
          /* Remove if to find subgraph homomorphisms.        */
8         if v ∉ M then
9             Add (u, v) to M;
10            Enumerate(C, 𝒜, φ, M, i + 1);
11            Remove (u, v) from M;
```

Proposition 2.4. *Given $v \in C(u)$, if there exists $u' \in N(u)$ such that $N(v) \cap C(u') = \emptyset$, then v can be removed from $C(u)$.*

Next, Line 2 generates a matching order φ, which is a sequence of query vertices. The exploration-based methods generally adopt the greedy strategy, which first selects a start vertex and then iteratively adds query vertices u to φ based on the cardinality estimation on the number of embeddings in $\mathcal{A}$ identical to $Q[\varphi \cup \{u\}]$. Finally, the *Enumerate* procedure recursively finds all results along φ. M records the mappings between query vertices and data vertices. Line 6 computes the local candidate vertex set $C_M(u)$ where $N_+^{\varphi}(u)$ denotes the backward neighbors of u given φ (Definition 2.5). If $i = 1$, then $C_M(u) = C(u)$. During the enumeration, M containing i mappings ($1 \leqslant i \leqslant |\varphi|$) is a subgraph isomorphism from $Q[\varphi[1:i]]$ to G where $\varphi[1:i]$ denotes the vertices indexed from 1 to i in φ. *By removing Line 8, Algorithm 2 can enumerate all subgraph homomorphisms from Q to G.*

Definition 2.5. Backward (Forward) Neighbors: Given a matching order φ, the backward neighbors $N_+^{\varphi}(u)$ (forward neighbors $N_-^{\varphi}(u)$) of a query vertex u is the neighbors of u that are positioned before (after) u in φ.

Despite that there are a lot of algorithms, they share the same *Enumerate* procedure in Algorithm 2. Their major difference is on designing powerful filtering methods to reduce the sizes of candidate vertex sets and optimizing the matching orders to reduce the size of the search space.

2.4 Join-Based Algorithms

WCOJ. WCOJ is a class of join algorithms whose running time matches $|OUT|$ of Q. Researchers developed a tight bound called AGM on $|OUT|$ in terms of the fractional edge cover of Q (Definition 2.6) [5]. Specifically, the optimal fraction edge cover number ρ^* is the minimum number of all fractional edge covers of Q. $|OUT|$ is bounded by $O(|IN|^{\rho^*})$ where $|IN|$ is the input size [24].

Definition 2.6. Fractional Edge Cover: Given $Q = (\mathcal{V}, \mathcal{E})$, a fractional edge cover x of Q is a mapping from $\mathcal{E} \rightarrow [0, \infty)$ such that $\forall u \in \mathcal{V}, \sum_{e \in \mathcal{E}, u \in e} x(e) \geqslant 1$. The fractional edge cover number ρ is equal to $\sum_{e \in \mathcal{E}} x(e)$.

Given Q and G, we construct a join query Q as follows: (1) $Q = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = V(Q)$ and $\mathcal{E} = E(Q)$; and (2) for each $e(u, u') \in \mathcal{E}, R(u, u') = \{(v, v') | e(v, v') \in E(G) \wedge L(v) = L(u) \wedge L(v') = L(u')\}$. $R(u : v, u')$ denotes the neighbor set of v in $R(u, u')$, i.e., $N(v) \cap \pi_{\{u'\}} R(u, u')$. In the following, we simply use Q to denote a join query for brevity. Correspondingly, a subgraph Q' of Q represents a join query $\bowtie_{e \in E(Q')} R_e$.

Algorithm 3: Leapfrog Triejoin (LFTJ)

```
1 Procedure LFTJ(Q = (V, E), φ, M, i)
2     if i = |φ| + 1 then Output M, return;
3     I ← {u ← φ[i]}, X_M(I) ←⋈_{e∈E_I} π_I R_e, J ← V − I;
4     foreach v ∈ X_M(I) do
5         Bind u to v in M;
6         Q' ←⋈_{e∈E_J} π_J(R_e ⋉ R(u)) where R(u) ← {v};
7         LFTJ (Q', φ, M, i + 1);
8         Unbind u from v in M;
```

Leapfrog Triejoin (LFTJ). Ngo et al. proposed GenericJoin [25], which is an abstraction of WCOJ algorithms including NPRR [24] and LFTJ [38]. We use LFTJ in our paper because it is an efficient instantiation of GenericJoin [4, 26]. LFTJ (i.e., Algorithm 3) evaluates Q with a matching order φ, and recursively finds all results by binding query vertices to data vertices along φ. Initially, $M = \{\}$ and $i = 1$. E_I denotes $\{e | e \in E \wedge e \cap I \neq \emptyset\}$. $\pi_I R_e$ obtains a set of data vertices and LFTJ computes $X_M(I)$ with set intersections (Line 3). When $i = 1$ and $u = \varphi[i]$, $X_M(I) = \bigcap_{e(u,u') \in E} \pi_{\{u\}} R(u, u')$. Line 6 constructs a new query Q' by removing u from V. Specifically, for each $e(u, u') \in E_J$, $\pi_J(R(u, u') \ltimes R(u))$ is equal to $R(u : M[u], u')$. If R_e has no common attributes with $R(u)$, then R_e will be unchanged. Therefore, when $i \geqslant 2$ and $u = \varphi[i]$, the relation R_e for $e \in E_I$ at Line 3 has two cases: (1) for $u' \in N_-^{\varphi}(u)$, $R_e = R(u, u')$ (i.e., u' is still in V); and (2) for $u' \in N_+^{\varphi}(u)$, $R_e = R(u' : M[u'], u)$ (i.e., u' has been removed from V). Then, $X_M(I)$ is equal to $X_1 \cap X_2$ where $X_1 = \bigcap_{u' \in N_-^{\varphi}(u)} \pi_{\{u\}} R(u, u')$ and $X_2 = \bigcap_{u' \in N_+^{\varphi}(u)} R(u' : M[u'], u)$. As M can contain duplicate data vertices, Algorithm 3 finds all subgraph homomorphisms. *Adding the check that is the same as Line 8 in Algorithm 2 can make LFTJ find all subgraph isomorphisms.*

To guarantee the worst-case optimal running time, the set intersections at Line 3 should satisfy the *min property*: the running time of the intersection algorithm has an upper bound limited by the length of the smaller input size [1].

Existing Methods. The latest systems such as LogicBlox [4], EmptyHeaded [1] and Graphflow [17, 23] employ WCOJ. Graphflow is the latest join-based subgraph query algorithm [23]. It prunes relations based on labels and optimizes the join plan based on a cost model with a variety of metrics (e.g., the cost of set intersections). Specifically, it enumerates all plans by considering the generation of a sub-query Q_k of Q with two alternative methods: (1) perform the pair-wise join on two smaller sub-queries; or (2) extend from a sub-query by adding one vertex. It picks the one with the minimum cost based on its cost model. When query graphs have more than ten vertices, Graphflow adopts the greedy method to generate join plans. Additionally, Graphflow can support edge-labeled and directed graphs. While focusing on vertex-labeled and undirected graphs, RapidMatch can be easily extended to support edge-labeled and directed graphs by using the edge label and the edge direction as pruning conditions, i.e., the data edge must have the same label and direction as the query edge in addition to vertex labels when generating relations for query edges in *RelationFilter*.

3 EXPLORATION VERSUS JOIN

Exploration-based and join-based approaches have been studied separately in most previous work. Since they solve very similar or the same problem, an important question is whether the enumeration procedure of an exploration-based algorithm is inherently better than a join-based algorithm. In the following, we present our analysis and algorithms (Sections 3 to 7) for subgraph homomorphism, since subgraph homomorphism can be achieved by Algorithm 2 without the check at Line 8. Also, we refer to Algorithm 2 for its *Enumerate* procedure, and we assume that retrieving the neighbors of v in a relation (i.e., $R(u : v, u')$) or an auxiliary structure (i.e., $\mathcal{A}^u_{u'}(v)$) takes $O(1)$ time.

Algorithm 2 enumerates results based on candidate vertex sets, whereas Algorithm 3 computes joins on a set of relations. For fair comparison, we give the two algorithms equivalent input and the same matching order. The latest exploration-based algorithms [7, 12] prune candidate vertex sets along a spanning tree of Q based on Proposition 2.4. Although these algorithms repeat the procedure a limited number of rounds in balance of efficiency and effectiveness, the resulting candidate vertex sets $C(u)$ after pruning are close to the *steady state*: given $v \in C(u)$, $\forall u' \in N(u), N(v) \cap C(u') \neq \emptyset$. Therefore, we assume the input candidate vertex sets of Algorithm 2 are at the steady state in the following analysis. Moreover, we build the input relations of Algorithm 3 as follows: given $e(u, u') \in E(Q)$, $R(u, u') = \{(v, v') | e(v, v') \in E(G) \wedge v \in C(u) \wedge v' \in C(u')\}$. These input satisfies Proposition 3.1.

Proposition 3.1. *Properties (1) Given an edge $e(u, u') \in E(Q)$, $\pi_{\{u\}} R(u, u') = C(u)$; and (2) Given an edge $e(u, u') \in E(Q)$, $R(u : v, u') = N(v) \cap C(u')$ where $v \in C(u)$.*

Proof. Given $e(u, u')$, $\pi_{\{u\}} R(u, u') \subseteq C(u)$ based on its construction method. According to the property of $C(u)$, for each $v \in C(u)$, $N(v) \cap C(u') \neq \emptyset$, i.e., there exists $e(v, v') \in E(G)$ where $v' \in C(u')$. Therefore, v must belong to $\pi_{\{u\}} R(u, u')$. As such, $C(u) \subseteq \pi_{\{u\}} R(u, u')$, and Property (1) is proved. Given $v \in C(u)$ and $v' \in N(v)$, the construction method of $R(u, u')$ ensures that if $v' \in C(u')$, then (v, v') belongs to $R(u, u')$; otherwise, $(v, v') \notin R(u, u')$. Therefore, Property (2) is proved. □

Next, we prove that Algorithm 2 satisfies Proposition 3.2.

Proposition 3.2. *Given matching order φ, Algorithm 2 generates the same intermediate results as Algorithm 3 at each step if their input satisfies Proposition 3.1.*

Proof. Without loss of generality, assume that $\varphi = (u_1, u_2, ..., u_i, ..., u_{|V(Q)|})$. To prove this proposition, we only need to prove that $C_M(u_i)$ in Algorithm 2 is equal to $X_M(\{u_i\})$ in Algorithm 3 at each step because both of them extend M by mapping v in $C_M(u_i)$ or $X_M(\{u_i\})$ to u_i to generate new intermediate results. We prove this by induction. Initially, $i = 1$ and $M = \{\}$. Based on the analysis in Sections 2.3 and 2.4, $C_M(u_1) = C(u_1)$ and $X_M(\{u_1\}) = \bigcap_{e(u_1,u) \in E(Q)} \pi_{\{u_1\}} R(u_1, u)$. Based on Proposition 3.1, for each $e(u_1, u) \in E(Q)$, $\pi_{\{u_1\}} R(u_1, u) = C(u_1)$. As a result, $X_M(\{u_1\}) = C(u_1)$ and $C_M(u_1)$ is equal to $X_M(\{u_1\})$. Therefore, the proposition is true for the initial step.

Assume that the proposition is true when $i = k - 1$ ($2 \leqslant k \leqslant |\varphi|$). We next show that it holds when $i = k$. In Algorithm 2, $C_M(u_k) = \bigcap_{u \in N^\varphi_+(u_k)} \mathcal{A}^u_{u_k}(M[u]) = \bigcap_{u \in N^\varphi_+(u_k)} (N(M[u]) \cap C(u_k))$. On the other hand, $X_M(\{u_k\}) = X_1 \cap X_2$ where $X_1 = \bigcap_{u \in N^\varphi_-(u_k)} \pi_{\{u_k\}} R(u_k, u)$ and $X_2 = \bigcap_{u \in N^\varphi_+(u_k)} R(u : M[u], u_k)$ (see the description of Algorithm 3 in Section 2.4). According to Proposition 3.1, $X_1 = C(u_k)$ and $X_2 = \bigcap_{u \in N^\varphi_+(u_k)} (N(M[u]) \cap C(u_k))$. As such, $C_M(u_k)$ is equal to $X_M(\{u_k\})$. Thus, the induction completes. As the proposition holds for both the initial and inductive steps, it is proved by induction. □

Algorithm 4: RapidMatch

Input: a query graph Q and a data graph G;
Output: all subgraph homomorphisms (or isomorphisms);
1 $Q_C, Q_F, \mathcal{T}_{1,2}, \mathcal{T}_{2,3}, \mathcal{T}_{3,4} \leftarrow$ nucleus decomposition on Q;
2 `RelationFilter`(Q, G, Q_C, Q_F);
3 $\varphi \leftarrow$ `JoinPlanGenerator`$(Q, \mathcal{T}_{1,2}, \mathcal{T}_{2,3}, \mathcal{T}_{3,4})$;
4 `RelationEncoder`(φ, Q_C, Q_F);
5 `ResultEnumerator`(φ, Q_C, Q_F);

According to Proposition 3.2, Algorithm 2 satisfies the following proposition.

Proposition 3.3. *The time complexity of Algorithm 2 matches the maximum output size of a query if the set intersection satisfies the min property.*

Proof. Based on the proof of Proposition 3.2, we can show that: (1) the computation of $X_1 = \bigcap_{u \in N^\varphi_-(u_k)} \pi_{\{u_k\}} R(u_k, u)$ can be removed because $\pi_{\{u_k\}} R(u_k, u) = C(u_k)$; and (2) the set intersections computing $X_M(\{u_k\}) = X_2$ and $C_M(u_k)$ take the same input. Thus, Algorithm 2 achieves the same worst-case optimality as LFTJ if the set intersection has the min property. □

4 AN OVERVIEW OF RAPIDMATCH

Algorithm 4 outlines the processing flow of RapidMatch. Specifically, it first performs the nucleus decomposition to obtain the core structure Q_C, the forest Q_F and the nuclei forests $\mathcal{T}_{1,2}, \mathcal{T}_{2,3}, \mathcal{T}_{3,4}$ of Q to identify dense subgraphs of Q as well as their hierarchical relationships. We adopt the nuclei forests $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,3}$ and $\mathcal{T}_{3,4}$ because (1) $\mathcal{T}_{1,2}$ contains all query vertices; and (2) previous research on $\mathcal{T}_{r,s}$ [30] showed that setting $r = 3$ and $s = 4$ is a sweet spot, which obtains dense subgraphs with comparable densities and can be computed efficiently. After nucleus decomposition, RapidMatch proceeds to subgraph query processing. Figure 2 illustrates the processing flow of the four components of RapidMatch.

First, given the input, *RelationFilter* builds a relation for each query edge based on vertex labels. Then, it decomposes Q into a set of tree-structured sub-queries because the full reducer cannot handle cyclic queries, and applies the full reducer to each of the sub-queries to eliminate data edges that will not appear in any join results. Next, *JoinPlanGenerator* generates a join plan φ based on the statistics of relations obtained in *RelationFilter* and the nuclei forests of Q. The join plan is to first evaluate Q_C with LFTJ and then perform a sequence of hash joins on subsequent queries to find final results. Because dense subgraphs (e.g., 4-clique) generally appear less frequently than sparse ones (e.g., 4-cycle) in G, φ prioritizes query vertices in dense regions of Q. Additionally, all joins are to be executed in a pipeline to avoid materializing intermediate results.

After join plan generation, *RelationEncoder* optimizes the data layout of relations based on the join plan at runtime to accelerate

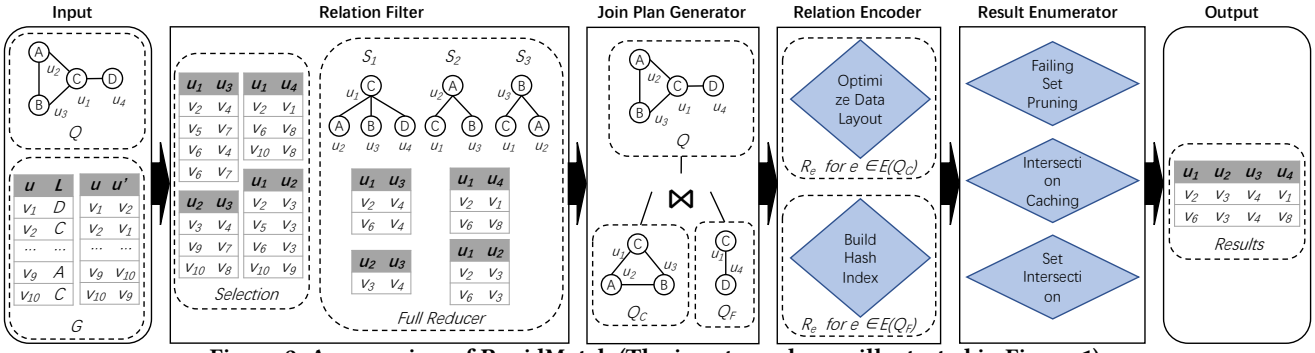


Figure 2: An overview of RapidMatch (The input graphs are illustrated in Figure 1).

the subsequent enumeration. It encodes the *ID*s of data vertices in relations in Q_C, and stores them in a *trie* structure to assist the computation of LFTJ. It also builds hash indexes for relations in Q_F to serve hash joins. Finally, *ResultEnumerator* executes the join plan on the encoded relations. Additionally, RapidMatch adopts several optimizations including advanced set intersection methods [1, 13], the intersection caching [23] and the failing set pruning [12]. In the following, we present the four components in detail.

5 RELATION FILTER

We present the relation filter in this section.

5.1 General Idea

Given Q and G, a simple method of generating a relation $R(u, u')$ for each $e(u, u') \in E(Q)$ is based on labels: $R(u, u') = \{(v, v') | e(v, v') \in E(G) \wedge L(v) = L(u) \wedge L(v') = L(u')\}$ [1, 23]. However, this method ignores the underlying graph structure, which can result in a large number of *dangling tuples* in relations, i.e., data edges that will not appear in any final result. This issue directly degrades the performance of join operations. Additionally, dangling tuples negatively affect the effectiveness of the join plan generation because the statistics of relations are important factors in the plan optimization [1, 7, 12, 23]. However, removing all dangling tuples of Q is an NP-hard problem due to the hardness of the subgraph homomorphism (or isomorphism) problem.

Proposition 5.1. *Given Q and G, removing all dangling tuples from the query $Q = \bowtie_{e \in E(Q)} R_e$ is NP-hard.*

Due to the hardness, we develop a heuristic taking advantage of graph structural features to remove dangling tuples. Specifically, we decompose Q into a set of tree-structured sub-queries Q' and apply the full reducer on each Q' to remove dangling tuples. After executing the full reducer, each data edge in relations R_e where $e \in E(Q')$ appears in a subgraph homomorphism from Q' to G.

5.2 Implementation Details

Algorithm 5 illustrates *RelationFilter*, which builds relations for each query edge and prunes dangling tuples. Lines 2-3 generate a relation for each query edge based on labels. Then, Lines 4-5 remove dangling tuples of each $Q_T \in Q_F$ with the full reducer. Next, we decompose Q_C into a set of trees S_u with $u \in V(Q_C)$ as the root and $N(u)$ as leaves and apply the full reducer to each of them. Specifically, Line 6 generates an order of query vertices to determine the processing sequence of the trees. We call this order the *filtering order* (δ). In the full reducer on S_u, we want to involve as many as possible relations that have been processed to utilize the pruning results from previous steps. Therefore, we first add the vertices exclusively belonging to $V(Q_F)$ to δ (Line 14) and then select the vertices in $V(Q_C)$ with most neighbors in δ as the next vertex (Lines 15-16). Lines 7-9 apply the full reducer to the trees along the order of δ. This procedure utilizes the pruning results on S_u to filter relations in $S_{u'}$ where u is positioned before u' in δ and u' is a neighbor of u. To filter relations in S_u with the pruning results on $S_{u'}$, Lines 10-11 conduct the same operations along the reverse order of δ.

Algorithm 5: RelationFilter

```
1  Procedure RelationFilter(Q, G, Q_C, Q_F)
2      foreach e(u, u') ∈ E(Q) do
3          R(u, u') ← {(v, v') | e(v, v') ∈ E(G) ∧ L(v) = L(u) ∧ L(v') =
             L(u')};
4      foreach Q_T ∈ Q_F do
5          FullReducer(Q_T);
6      δ ← GenerateFilteringOrder(Q, Q_C, Q_F);
7      foreach u ∈ V(Q_C) along the order of δ do
8          S_u ← the tree rooted at u with N(u) as leaves;
9          FullReducer(S_u);
10     foreach u ∈ V(Q_C) along the reverse order of δ do
11         The same as Lines 8-9;
12 Function GenerateFilteringOrder(Q, Q_C, Q_F)
13     δ ← ();
14     Add u ∈ V(Q_F) − V(Q_C) into δ with an arbitrary order;
15     while |δ| ≠ |V(Q)| do
16         Add arg max_{u∈V(Q_C)−δ} |N(u) ∩ δ| into δ;
17     return δ;
```

Example 5.2. In Figure 2, Q_F contains only one edge, $e(u_1, u_4)$, and the full reducer on the forest does not update Q_F. Suppose $\delta = (u_4, u_1, u_2, u_3)$. Algorithm 5 applies the full reducer on S_1, S_2 and S_3, respectively along the order of δ, and then the reverse order. The relations after filtering are listed in the figure.

5.3 Analysis of Relation Filter

Space and time. We build a relation for each query edge, which contains data edges that can be mapped to the query edge. Therefore, the space complexity is $O(|E(Q)| \times |E(G)|)$. Lines 2-3 in Algorithm 5 take $O(|E(Q)| \times |E(G)|)$ time since we scan $E(G)$ for each

query edge. The full reducer on $Q_T \in Q_F$ takes $2 \times (|E(Q_T)| - 1)$ semi-joins. Hence, Lines 4-5 takes $\sum_{Q_T \in Q_F} 2 \times (|E(Q_T)| - 1) \leqslant 2 \times |E(Q_F)|$ semi-joins. We omit the cost of generating δ as it is trivial. The full reducer on a tree S_u has $2 \times (d(u) - 1)$ semi-joins. Therefore, Lines 7-11 take $4 \times \sum_{u \in V(Q_C)} (d(u) - 1) \leqslant 8 \times |E(Q)|$ semi-joins. A semi-join on relations R and R' takes $|R| + |R'|$ time. Then, the time complexity of Algorithm 5 is $O(|E(Q)| \times |E(G)| + (2 \times |E(Q_F)| + 8 \times |E(Q)|) \times |E(G)|) = O(|E(Q)| \times |E(G)|)$.

Pruning power. Next, we compare the pruning power with CFLMatch and DP-iso, the latest exploration-based algorithms. CFLMatch and DP-iso utilize Proposition 2.4 to filter the candidate vertex set $C(u)$ for each $u \in V(Q)$ based on $C(u')$ where $u' \in N(u)$ along a spanning tree of Q. Repeating such a procedure till no candidate vertex sets $C(u)$ can be updated reaches the steady state: given $v \in C(u)$, $\forall u' \in N(u), N(v) \cap C(u') \neq \emptyset$. However, CFLMatch and DP-iso set the number of rounds as two and three, respectively, to balance effectiveness and efficiency.

If repeating the procedure at Lines 7-9 in Algorithm 5 till no relations can be updated, then $R(u, u') \ltimes R(u, u'')$ will be equal to $R(u, u')$ given $u \in V(Q_C)$ and $u', u'' \in N(u)$. Therefore, $\pi_{\{u\}} R(u, u')$ will be equal to $\pi_{\{u\}} R(u, u'')$. For each $u \in V(Q_C)$, let $C(u)$ be $\pi_{\{u\}} R(u, u')$ where u' is an arbitrary neighbor of u. Then, $C(u)$ will satisfy the steady state because given $u' \in N(u)$ and $v \in C(u)$, there exists $(v, v') \in R(u, u')$ where $v' \in C(u')$. Therefore, the pruning power of Algorithm 5 is competitive with the native filtering methods in exploration-based algorithms. To balance the execution time and the pruning power, we set the times of executing the pruning procedure to two.

6 JOIN PLAN GENERATOR

This section elaborates the join plan generator.

6.1 General Idea

The join plan in RapidMatch is a matching order φ that positions the core vertices $V(Q_C)$ before the non-core vertices (i.e., $V(Q) - V(Q_C)$). Moreover, φ is *connected*, i.e., $\forall 1 \leqslant i \leqslant |\varphi|, Q[\varphi[1 : i]]$ is a connected graph. Then, a non-core vertex has exactly one backward neighbor in φ based on Definition 2.1. RapidMatch first executes $Q_C = \bowtie_{e \in E(Q_C)} R_e$ with LFTJ taking $\varphi[1 : |V(Q_C)|]$ as the matching order, and then evaluates $Q_C \bowtie Q_F$ where $Q_F = \bowtie_{e \in E(Q_F)} R_e$ with a sequence of hash joins. In practice, RapidMatch executes all joins in a pipeline manner, i.e., the results generated by one join are immediately emitted to the subsequent joins, to avoid materializing intermediate results.

Example 6.1. In Figure 2, suppose that $\varphi = (u_1, u_2, u_3, u_4)$. RapidMatch first evaluates $Q_C = R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3)$ with LFTJ along (u_1, u_2, u_3), and then performs a hash join $R(Q_C) \bowtie R(u_1, u_4)$ to find results of Q.

Existing methods [7, 12] optimize φ based on the cost model $cost(\varphi) = \sum_{i=1}^{|V(Q)|} |R(Q[\varphi[1 : i]])|$, which is the total number of intermediate results generated during the enumeration. However, the *cardinality estimation* on the output size of sub-queries of Q is very challenging, especially when Q is large. The bias of the estimation can result in very ineffective join plans. Due to the difficulty of the cardinality estimation, we optimize φ from the perspective of graph structures. When $X_M(\{u\})$ is empty in Algorithm 3, an intermediate result M cannot be further extended, i.e., it cannot appear in any final result. We optimize φ by terminating such invalid search paths at an early stage. $X_M(\{u\})$ is computed by $\bigcap_{u' \in N_+^\varphi(u)} R(u' : M[u'], u)$ as discussed in Section 2.4. Then, the join plan φ putting query vertices with more backward neighbors at the beginning tends to perform better, and we define the utility function as $utility(\varphi) = \sum_{i=1}^{|V(Q)|} \sum_{j=1}^{i} |N_+^\varphi(\varphi[j])|$. However, maximizing the equation by listing all permutations of $V(Q)$ is prohibitively expensive when Q is large. We propose to optimize the utility function by prioritizing vertices in dense regions of Q since $\sum_{j=1}^{i} |N_+^\varphi(\varphi[j])|$ is equal to $|E(Q[\varphi[1 : i]])|$, i.e., the number of edges in the vertex-induced subgraph of Q on the first i vertices. In this paper, we generate φ with the nucleus decomposition because it can efficiently find high quality dense regions with detailed hierarchies [30]. For the sequence of hash joins, we prefer first joining with the relation having a small cardinality.

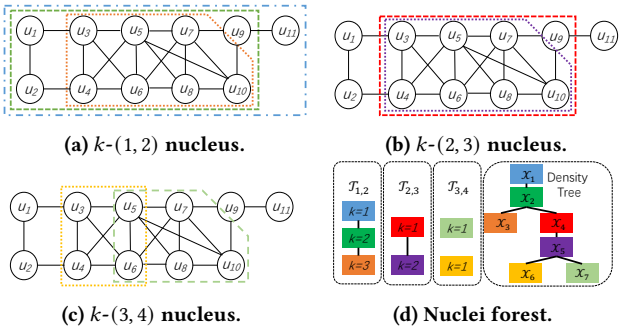
Algorithm 6: JoinPlanGenerator

```
1  Function JoinPlanGenerator(Q, T_{1,2}, T_{2,3}, T_{3,4})
2      T ← ConstructDensityTree(T_{1,2}, T_{2,3}, T_{3,4});
3      Ω ← {};
4      foreach X ∈ T.leaves do
5          e*(u,u') ← arg min_{e(u,u')∈E(X)} (d(u) + d(u'));
6          φ ← (u,u') where we suppose that d(u) ⩾ d(u');
7          Traverse(Q, T, φ, X);
8          Add φ to Ω;
9      return arg max_{φ∈Ω} utility(φ);
10 Function ConstructDensityTree(T_{1,2}, T_{2,3}, T_{3,4})
11     foreach T ∈ T_{3,4} do
12         Set the parent of T.root as the smallest X in T_{2,3} containing it;
13     foreach T ∈ T_{2,3} do
14         Set the parent of T.root as the smallest X ∈ T_{1,2} containing it;
15     return the tree in T_{1,2};
16 Procedure Traverse(Q, T, φ, X)
17     if X is visited then return;
18     Mark X as visited;
19     if V(X) − φ = ∅ then Go to Line 29;
20     while X.children ≠ ∅ do
21         X* ← arg max_{X'∈X.children} connection(φ, X');
22         if V(X*) ∩ φ = ∅ then
23             Add vertices u in SP to φ if u ∉ φ;
24         Traverse(Q, T, φ, X*);
25         Remove X* from X.children;
26     while V(X) − φ ≠ ∅ do
27         u* ← arg max_{u∈V(X)−φ} |N(u) ∩ φ|;
28         Add u* to φ;
29     if X ≠ T.root then Traverse(Q, T, φ, X.parent);
```

6.2 Implementation Details

Algorithm 6 presents our join plan generator based on the nucleus decomposition. Line 2 first constructs a *density tree* T of Q (via the *ConstructDensityTree* function), in which the nodes are nuclei. Lines 4-8 traverse T from the leaf node to put vertices in dense regions of Q at the beginning of φ via the *Traverse* function. We add vertices in $e(u, u') \in E(\mathcal{X})$ with the maximum degree sum, and start the traversal (Lines 5-7). In the following, we present more details about the two functions: *ConstructDensityTree* and *Traverse*.

In the *ConstructDensityTree* function, for each nuclei tree $T \in \mathcal{T}_{3,4}$, we set the parent of the root node of T as the smallest (by the vertex



(a) k-$(1,2)$ **nucleus.** **(b)** k-$(2,3)$ **nucleus.**

(c) k-$(3,4)$ **nucleus.** **(d) Nuclei forest.**

Figure 3: Example of a density tree.

cardinality) nucleus $\mathcal{X}$ in $\mathcal{T}_{2,3}$ containing it (Lines 11-12). Lines 13-14 conduct a similar operation for each $T \in \mathcal{T}_{2,3}$. As a result, the *ConstructDensityTree* function generates a single tree T, because (1) the 1-$(1,2)$ nucleus contains all query vertices; and (2) given a k-(r,s) nucleus, there must be a k'-$(r-1,s-1)$ nucleus containing it where $k' \geqslant k$.

Example 6.2. The query graph Q is illustrated in Figure 3a where we omit the vertex label for brevity. Figures 3a, 3b and 3c present the k-$(1,2)$, k-$(2,3)$ and k-$(3,4)$ nucleus decomposition, respectively. The corresponding nuclei forests are illustrated in Figure 3d. Take $\mathcal{X}_6$ containing $\{u_3, u_4, u_5, u_6\}$ as an example. The smallest nucleus in $\mathcal{T}_{2,3}$ containing it is $\mathcal{X}_5$. We set $\mathcal{X}_5$ as the parent of $\mathcal{X}_6$. The density tree of Q is presented in Figure 3d.

In the *Traverse* function, if $\mathcal{X}$ has been visited, then return (Line 17). Otherwise, mark it as visited. If all vertices in $\mathcal{X}$ exist in φ, then jump to its parent (Line 19). As the density of $\mathcal{X}$ is generally lower than its children, Lines 20-24 first consider its children. We prioritize the child $\mathcal{X}'$ having stronger *connection* with φ, which is defined in Equation 1. $|SP(\varphi, \mathcal{X}')|$ is the length of the shortest path from $\mathcal{X}[\varphi]$ to $\mathcal{X}'$ through vertices in $\mathcal{X}$. The child that has more common vertices with φ or is close to φ scores a high value.

$$connection(\varphi, \mathcal{X}') = |\varphi \cap V(\mathcal{X}')| + \frac{1}{1 + |SP(\varphi, \mathcal{X}')|}. \quad (1)$$

After visiting all children of $\mathcal{X}$, Lines 26-28 add the remaining vertices in $\mathcal{X}$ to φ. At each step, we pick the vertex u with the maximum $|N(u) \cap \varphi|$ to optimize the utility function. We break ties with (1) the high vertex degree; and (2) the small relation cardinality. Line 29 traverses to the parent of $\mathcal{X}$. Finally, Line 9 returns φ with the maximum utility value as the join plan.

Example 6.3. Continue Example 6.2 and we traverse from $\mathcal{X}_6$ of T. We first add u_5, u_6 to φ since $e(u_5, u_6)$ has the maximum degree sum in $\mathcal{X}_6$. We add all vertices in $\mathcal{X}_6$ to φ and traverse to its parent $\mathcal{X}_5$. $\mathcal{X}_7$ is the child of $\mathcal{X}_5$ and the connection between φ and $\mathcal{X}_7$ is 3 because they have two common vertices u_5, u_6 and the shortest path between them is 0. Then, we traverse to $\mathcal{X}_7$. Both u_7 and u_8 have two neighbors in φ. Based on rules breaking ties, we add u_7 to φ since it has a higher vertex degree than u_8. Next, we add u_8 to φ before u_{10} because u_8 has more neighbors in φ. After adding all vertices in $\mathcal{X}_7$ to φ, the traversal procedure leaves $\mathcal{X}_7$ and continues the process until all query vertices are added into φ.

6.3 Analysis of Join Plan Generator

The join plan generated by Algorithm 6 satisfies Proposition 6.4, which meets the requirement of RapidMatch. We omit the proofs of Propositions 6.4 and 6.5 for brevity.

Proposition 6.4. *φ generated by Algorithm 6 is connected and core vertices are before non-core vertices in it.*

Time of Algorithm 6. The time complexity of constructing the density tree T is $O(|V(Q)| \times |T|^2)$. The cost of computing Equation 1 is at most $O(|E(Q)|)$. Then, the entire traversal procedure takes $O(|E(Q)| \times \sum_{\mathcal{X} \in T} |\mathcal{X}.children|^2) = O(|E(Q)| \times |T|^2)$ time to compute Equation 1. Lines 26-28 take $O(|V(Q)|^2)$ time during the entire traversal procedure of T. As we start a traversal from each leaf, the time complexity of Algorithm 6 is $O(|T| \times |V(Q)| + |T|^3 \times |E(Q)| + |T| \times |V(Q)|^2)$. Because Q only contains tens of vertices and the nucleus in a nuclei forest have the *disjointness property* [31], $|T|$ is very small in our experiments and we regard it as a constant value. Then, the time complexity of Algorithm 6 is $O(\alpha(Q) + |V(Q)|^2)$ where $\alpha(Q)$ denotes the total cost of the nucleus decomposition obtaining $\mathcal{T}_{1,2}, \mathcal{T}_{2,3}, \mathcal{T}_{3,4}$.

Time of joins. RapidMatch evaluates $R(Q_C) \bowtie (\bowtie_{e \in E(Q_F)} R_e)$ with a sequence of hash joins along φ. Benefiting from *RelationFilter*, the joins satisfy Proposition 6.5. Therefore, RapidMatch has the *worst-case optimality* because it evaluates Q_C with LFTJ, which is worst-case optimal. In particular, when Q is a tree, RapidMatch first applies the full reducer on Q in *RelationFilter*, and then evaluates Q with a sequence of pair-wise joins, which is the same as the Yannakakis algorithm and has the *instance optimality* [39].

Proposition 6.5. *Each join of $R(Q_C) \bowtie (\bowtie_{e \in E(Q_F)} R_e)$ along φ increases the size of intermediate results.*

Space of Joins. RapidMatch does not materialize intermediate results, while other approaches [1, 17, 23] do if their join plans contain pair-wise joins on two sub-queries.

Discussions. CFLMatch [7] proposes to decompose Q into Q_C and Q_F, and prioritizes core vertices to start the enumeration from the dense part of Q. However, CFLMatch cannot further take advantage of the dense structures of Q_C, which can have more vertices than Q_F, for example, Q_C has ten vertices (u_{1-10}) in Figure 3a, but Q_F has only one vertex (u_{11}). The decomposition method of CFLMatch can put u_{11} after u_{1-10} in φ but cannot obtain the dense structures of the Q_C. In contrast, we generate φ by constructing the density tree based on the nucleus decomposition. It can efficiently find high-quality dense subgraphs with a multi-level hierarchy as shown in Figure 3d.

Our ordering method is a greedy approach based on the heuristic rule that the dense sub-structures of Q generally appear less frequently in G. The matching order prioritizes the dense sub-structure of Q with the basic cardinality estimation to break ties. This heuristic method is effective in practice, but it also has some limitations. First, there is no guarantee that the optimal join plan must be within our plan space. Second, the ordering method can generate ineffective matching orders on the workloads where the assumption in the heuristic rule does not hold. For example, we may generate an ineffective matching order if the dense part of Q appears much more frequently than the sparse part in G due to the distributions of labels.

7 OTHER IMPLEMENTATION DETAILS

This section introduces the relation encoder and result enumerator.

7.1 Relation Encoder

LFTJ stores a relation $R(u, u')$ as a two-level *trie* [38]. The first level stores the values v of u, and the second records the neighbors, which are sorted. We optimize the relation data layout by encoding vertices in relations, and call it *encoded trie*. Specifically, for each $u \in V(Q_C)$, we first generate a relation $R(u)$ containing candidate data vertices of u by $\pi_{\{u\}}R(u, u')$ where u' is an arbitrary neighbor of u. $R(u)$ is stored as an array. Note that given $u \in V(Q_C)$, all relations containing the attribute u share the same $R(u)$. $pos(v)$ denotes the position of v in $R(u)$. For each $e(u, u') \in E(Q_C)$ where $u \in N^{\varphi}_{+}(u')$, we generate $R'(u, u')$ by looping over $(v, v') \in R(u, u')$: if $v \in R(u)$ and $v' \in R(u')$, then add $(pos(v), pos(v'))$ into $R'(u, u')$; otherwise, skip (v, v'). Furthermore, given $v \in R(u) - \pi_{\{u\}}R(u, u')$, add $(pos(v), \emptyset)$ into $R'(u, u')$ to indicate that v has no neighbors in $R'(u, u')$. After that, we store $R'(u, u')$ as a trie. For each $e(u, u') \in E(Q_F)$, we build a hash index of $R(u, u')$ to serve the hash join.

The time complexity of optimizing the relation data layout is $O(\sum_{e \in E(Q_F)} |R_e| + \sum_{e \in E(Q_C)} |R_e| \times \log |R_e|)$, and stores it as a trie takes $O(|R| \times \log |R|)$ time. Retrieving the neighbors of a vertex in the encoded trie takes $O(1)$ time, while $O(\log |R|)$ time in the trie. Additionally, the set intersection method storing input sets in compact layouts [13] can benefit from the encoding because the domain of vertex *ID*s is reduced to $[0, |R(u)|)$.

7.2 Result Enumerator

In the result enumeration, we adopt three optimizations to improve its efficiency. Firstly, we adopt two set intersection (SI) methods in LFTJ. We use QFilter [13], which encodes sets in a compact layout, for small queries whose neighbor sets in relations are large. For large queries whose neighbor sets are small after filtering, we use a hybrid method on integer arrays. The hybrid SI method denoted by *Hybrid* handles the cardinality skew of sets by integrating the merge-based method denoted by *Merge* with the *Galloping* algorithm [1]. We further accelerate *Merge* and *Hybrid* with the AVX2 instruction set (256-bit width) denoted by *M+AVX2* and *H+AVX2*, respectively. QFilter [13] is implemented with the SSE instruction set (128-bit width). Secondly, we use the intersection caching [23] for small queries [23]. Thirdly, we utilize the failing set pruning [12] to accelerate large queries.

8 EXPERIMENTS

We conduct experiments to evaluate the effectiveness of RapidMatch in this section.

8.1 Experimental Setup

Algorithms Under Study. We compare the performance of RapidMatch (RM) with CFLMatch (CFL) [7], DP-iso (DP) [12] and Graphflow (GF) [23]. GF is the state-of-the-art join-based algorithm. CFL and DP are the latest exploration-based algorithms. Because the open-source version[1] of CECI [6] failed on a number of queries, we do not include CECI in our experiments.

[1]https://github.com/iHeartGraph/ceci-release, Last accessed on 2020/08/15.

Table 3: Properties of real-world datasets.

Graph Category	Dataset	Name	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d
Biology	Human	*hu*	4,674	86,282	44	36.9
Lexical	WordNet	*wn*	76,853	120,399	5	3.1
Citation	US Patents	*up*	3,774,768	16,518,947	20	8.8
Social	LiveJournal	*lj*	4,847,571	42,851,237	4	17.7
	Youtube	*yt*	1,134,890	2,987,624	25	5.3
	DBLP	*db*	317,080	1,049,866	15	6.6
Web	eu2005	*eu*	862,664	16,138,468	4	37.4

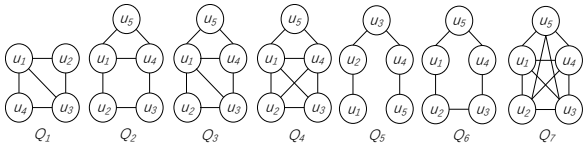


Figure 4: Query graphs.

Implementation and Experiment Environment. We obtain the source code of CFL from its original authors. The source code of GF[2] and the binary file of DP[3] are publicly available at GitHub. RM, CFL and DP are implemented in C++, and GF is programmed in JAVA. The C++ code is complied by g++ 7.3.1 with the -O3 flag enabled and the JAVA code is complied by JAVAC 1.8.0. We conduct experiments on a Linux machine with two Intel Xeon E5-2650 v3 CPUs and 64GB RAM. As the binary of DP does not support for finding homomorphisms, we use our homegrown implementation of DP-iso to answer small queries. As GF considers edge-labeled and directed graphs, we set edges of data graphs and query graphs as the same label and store each edge of data graphs in both directions. We use the default settings of GF to build catalogues for data graphs other than the Youtube dataset on which we set the parameter h as two to avoid the out-of-memory issue. GF can build the catalogues within ten minutes. We omit the detailed figures for brevity.

Datasets. The details of the data graphs are listed in Table 3. We use two kinds of workloads that are widely used in the previous studies[1, 7, 12, 23]. The first kind uses the small queries in Figure 4, which are mostly used in join-based approaches [1, 23]. For consistency with those previous studies, those queries perform homomorphism. We select eu2005 and LiveJournal as the data graphs, and use the same experimental setting as previous studies [23]: (1) as the graphs originally have no label, we choose a label from a set Σ of distinct labels uniformly at random and assign it to the vertex; (2) we set the number of distinct labels as four, because the workload will be trivial to be evaluated if Σ has a large number of distinct labels.

The second kind adopts large queries having tens of vertices each, which perform subgraph isomorphism and are widely used in the experiments of exploration-based algorithms [7, 12, 19]. We select DBLP, Youtube, US Patents, Human and WordNet as the data graphs. Human and WordNet originally have labels. For other datasets, we choose a label from Σ uniformly at random and assign it to the vertex. We vary the size of $|\Sigma|$ from 10 to 30 at a step of 5, and pick the size such that we can demonstrate the capabilities of competing algorithms without breaking most of them. We generate query graphs using the same approach as [7, 12], which randomly extracts subgraphs from the data graphs. We generate a dense query

[2]https://github.com/queryproc/optimizing-subgraph-queries-combining-binary-and-worst-case-optimal-joins, Last accessed on 2020/08/15.

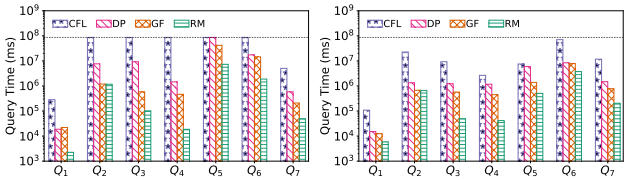
[3]https://github.com/SNUCSE-CTA/DAF, Last accessed on 2020/08/15.

set (i.e., $d(Q) \geqslant 3$) and a sparse query set (i.e., $d(Q) < 3$) for each data graph. Each set contains 200 connected query graphs with the same number of vertices. We set $|V(Q)| = 20$ for Human and WordNet, but $|V(Q)| = 32$ for other graphs, because Human is dense and most of vertices in WordNet and Human have the same label, which makes them challenging. The dense and sparse query sets with i vertices are denoted as H_{iD} and H_{iS}, respectively.

Metrics. To keep consistent with previous research, we find all results for small queries [1, 23], while terminate the query after finding 10^5 results for large queries [7, 12, 19, 22]. We measure the *query time* in milliseconds (ms) to process a query on a data graph, which consists of the *filtering time* (i.e., the time spent on the filtering), the *ordering time* (i.e., the time spent on generating the join plan), the *encoding time* (i.e., the time spent on the encoding) and the *enumeration time* (i.e., the time spent on enumerating results). The *preprocessing time* consists of the filtering time, the ordering time and the encoding time. To compare the pruning capability of the filtering methods, we examine the *relation cardinality*, which is $\frac{\sum_{e \in E(Q)} |R_e|}{|E(Q)|}$, and the *memory consumption* on storing relations. We set the time limit for a small query as 24 hours and terminate the large query if it cannot be completed within 5 minutes so that our experiments can be finished within a reasonable time.

8.2 Comparison with Existing Algorithms

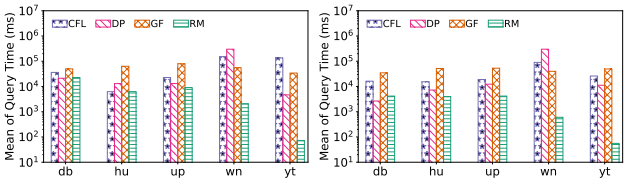
Figure 5 presents the experiment results on small queries. GF generally runs faster than exploration-based algorithms. RM significantly outperforms competing algorithms. In particular, RM completes Q_5 on *eu* within around 2 hours while other algorithms spend more than ten hours, which demonstrates the advantage of RM on acyclic queries. GF is competitive with RM on Q_2 since its join plan considers the impact of the intersection caching on the enumeration. CFL performs worse than other algorithms because its local candidate vertex set computation, which depends on G, is much slower than the set intersection based methods.



(a) Vary Q on *eu*. **(b) Vary Q on *lj*.**

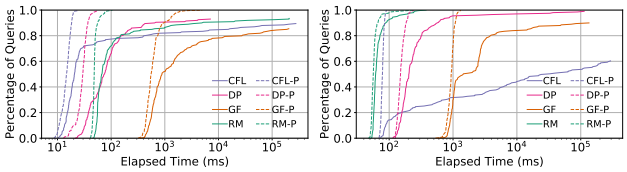
Figure 5: Overall performance on small queries.

We next evaluate the performance of competing algorithms on large queries. Figure 6 presents the results on the *mean of query time* on a query set, i.e., $\frac{1}{|H|} \sum_{Q \in H} t(Q)$ where $t(Q)$ is the query time on a query Q. RM runs more than one order of magnitude faster than competing algorithms on *wn* and *yt*, while DP and RM are competitive on *db*. To further examine the performance of competing algorithms on individual query, we report the *cumulative distribution function* of the preprocessing time (the dashed line) and query time (the solid line) of queries in a query set in Figure 7. The preprocessing time dominates the query time on the short running queries (i.e., the query with a short query time). As the preprocessing time of RM is longer than that of CFL on *db*, the



(a) Vary G on dense query sets. **(b) Vary G on sparse query sets.**

Figure 6: Overall performance on large queries.

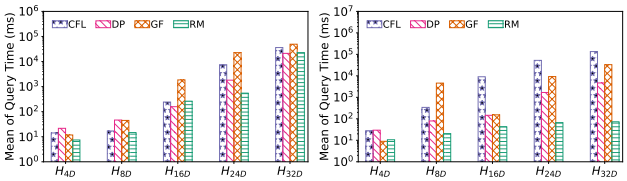


(a) H_{32D} on *db*. **(b) H_{32D} on *yt*.**

Figure 7: Cumulative distribution on large queries.

query time of RM is more than that of CFL on a number of queries. In contrast, RM performs much better than CFL for long running queries on *db* and *yt*. As a result, RM spends less time on processing a query set than CFL. Because GF is a direct-enumeration method, its ordering time is equal to the preprocessing time. We can see that its ordering time is much longer than the preprocessing time of the preprocessing-enumeration algorithms although it adopts a greedy join plan generation method for large queries. As a result, GF is slower than other algorithms on a number of queries.

Furthermore, we evaluate competing algorithms with different query sizes in Figure 8. RM performs well on queries with different sizes. GF runs faster than the other algorithms on H_{4D} against *yt* because the overhead of the pruning in the preprocessing-enumeration methods can offset the benefit on the query with a short running time. However, GF runs much slower than competing algorithms on H_{8D} against *yt* because it generates ineffective join plans for a number of queries.



(a) Vary query sizes on *db*. **(b) Vary query sizes *yt*.**

Figure 8: Overall performance with different query sizes.

Finding (1): (A) Join-based algorithms outperform exploration-based on small queries, and RM outperforms all other counterparts by up to one order of magnitude; (B) Although no algorithm can dominate competing algorithms on each large query, RM performs very well in most tested workloads; and (C) The overhead of the pruning can offset the benefit on queries with a short running time.

8.3 Evaluation of Individual Techniques

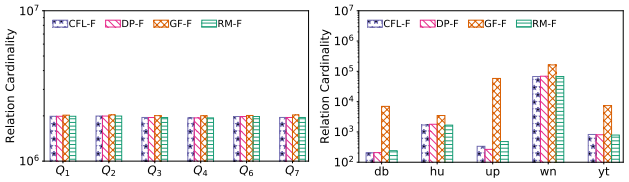
We evaluate individual techniques listed in Table 4 within our subgraph query processing framework. We first compare the pruning power of the filtering methods. Next, we evaluate the effectiveness of join plans with and without the optimization techniques

Table 4: A summary of individual techniques under study.

Category	Method	Description
Filtering relations or candidate vertex sets	RM-F	Filtering relations with full reducers
	GF-F [23]	Filtering relations with selection on labels
	CFL-F [7]	Native candidate vertex sets filtering method based on Proposition 2.4
	DP-F [12]	Native candidate vertex sets filtering method based on Proposition 2.4
Generating matching order or query plan	RM-O	Generating join plans with the nucleus decomposition
	GF-O [23]	Generating join plans with sampling
	CFL-O [7]	A path-based ordering method
	DP-O [12]	A path-based ordering method
Optimizing enumeration	IC [23]	Intersection caching method in GF
	FSP [12]	Failing set pruning method in DP
Optimizing data layout	Encoded	Storing relations as trie with vertex IDs encoded
	Trie [38]	Storing relations as trie
	Hash [38]	Storing relations as hash table
Accelerating Set Intersection(SI)	Merge	Merge-based SI
	M+AVX2	Merge-based SI with AVX2
	Hybrid	Integrate Merge-based with Galloping SI
	H+AVX2	Hybrid SI with AVX2
	QFilter [13]	Sparse bitmap based SI with SSE

including the intersection caching and the failing set pruning. Finally, we evaluate the data layout optimizations with different set intersection (SI) methods. By default, the filtering method, the join plan generation method and the data layout are RM-F, RM-O and *Encoded*. We adopt QFilter and the intersection caching for small queries, while use *H+AVX2* and the failing set pruning for large queries. Due to space limit, we omit the experiment results on the filtering time, join plan generation time and encoding time because the absolute value is very small. RM takes at most 600MB memory space to store relations in our experiments, and we do not report detailed results on memory consumption as well. Additionally, we omit the path query Q_5 since Section 8.2 has shown the efficiency of RM on it.

8.3.1 Filtering Methods. As CFL-F and DP-F focus on pruning candidate vertex sets, we count the number of data edges between candidate vertex sets as their relation cardinality. Figure 9 presents the results. RM-F is competitive with CFL-F and DP-F, and outperforms GF-F. RM-F, DP-F and CFL-F significantly reduce relation cardinalities over GF-F on *db*, *up* and *yt*. In contrast, they reduce relation cardinalities by 5%-10% on *eu* compared with GF-F, and the results of the four methods are close on *hu* and *wn*. This is because *eu* only has four distinct labels, and most vertices of *hu* and *wn* have the same label. Consequently, data vertices in these datasets are more likely to pass the filtering, since RM-F, DP-F and CFL-F prune the input based on the neighborhood information of vertices.



(a) Vary Q on *eu*. **(b) Vary G on dense query sets.**

Figure 9: Comparison of filtering methods.

Finding (2): (A) The pruning power of RM-F is competitive with the native methods CFL-F and DP-F; and (B) the effectiveness of filtering methods is limited when the data graph has only a few distinct labels because the overhead of filtering offset the benefit.

8.3.2 Join Plans and Enumeration Optimizations. Figure 10 presents the enumeration time of small queries on *eu*. The bars with grey and white background colors represent the time without and with the intersection caching (called IC for short), respectively. When disabling the intersection caching, the enumeration time of the four methods is close to each other except Q_4 on which RM-O and DP-O significantly outperform CFL-O and GF-O. Enabling the intersection caching reduces the enumeration time on some queries.

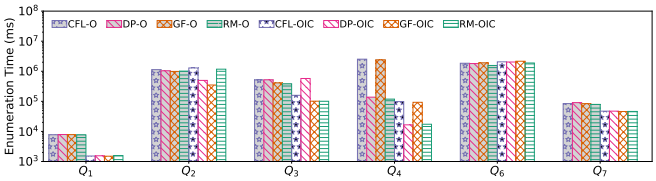
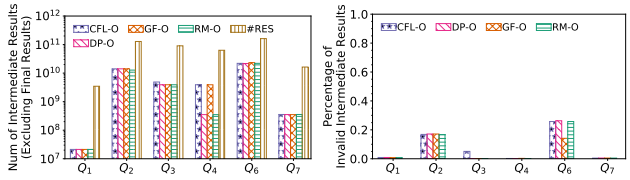
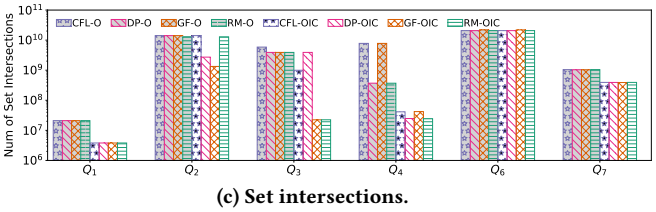


Figure 10: Vary Q on *eu* without/with IC.

We collect more detailed metrics in Figure 11 to further interpret the experiment results. Figure 11a reports the number of intermediate results and final results (#RES) generated during the enumeration. We can see that final results are much more than intermediate results. Figure 11b presents the percentage of invalid intermediate results, which are the intermediate results not contained in any final results. The invalid intermediate results account for a small portion of intermediate results. We count the number of set intersections during the enumeration in Figure 11c. Its trend is the same as the enumeration time in Figure 10. Based on the detailed metrics, we can see that the cost of set intersections is a key performance factor. The intersection caching improves the performance on some queries because it reduces the number of set intersections. However, RM-O does not consider its impact on the enumeration. Consequently, it performs worse than GF-O on Q_2.



(a) Intermediate results. **(b) Invalid intermediate results.**



(c) Set intersections.

Figure 11: Detailed metrics of small queries on *eu*.

Finding (3): For small queries, (A) Final results of a query are much more than intermediate results, and generally only a small portion of intermediate results are invalid; (B) The cost of set intersections is the key performance factor of the enumeration; and (C) RM-O can be slower than GF-O on some queries because RM-O has not taken the cost of set intersections and the impact of the intersection caching into consideration.

Figure 12 presents the mean of enumeration time on large dense queries without and with the failing set pruning (called FSP for short), which are represented by bars with the grey and white background colors, respectively. RM-O runs slightly faster than competing algorithms on *db* and *up*, but slower than CFL-O and GF-O on *hu*, which is very dense. In contrast, RM-O achieves up to one order of magnitude speedup over the other three algorithms on *wn* and *yt*, which are sparse. Enabling FSP improves the performance on some datasets, e.g., *yt*.

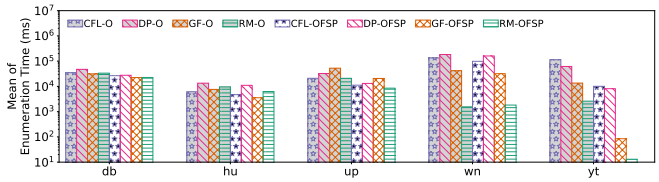
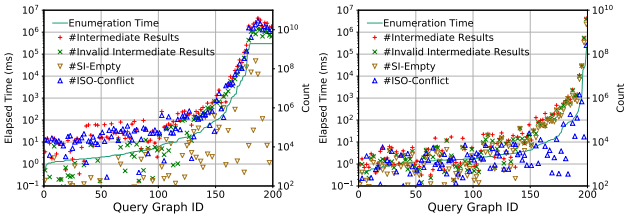


Figure 12: Vary G on large dense queries without/with FSP.

Moreover, we collect the detailed metrics of RM-O in Figure 13. We relabel the *ID*s of query graphs based on the enumeration time, and count the number of intermediate results and invalid intermediate results for each query. An intermediate result is invalid in Algorithm 2 due to either the local candidate vertex set is empty or the data vertex has been mapped. We count the number of the two kinds of failures denoted by *SI-Empty* and *ISO-Conflict*, respectively. As shown in the figure, the enumeration time grows with the increase of intermediate results, and invalid intermediate results dominate intermediate results for long running queries. We observe that the failures of long running queries on *db* are mainly due to the *ISO-Conflict*, while *SI-Empty* on *yt*. This is because (1) a data vertex in *db* can be candidates of more query vertices than *yt*, which results in *ISO-Conflict*, since *db* has fewer distinct labels than *yt*; and (2) data vertices in *yt* are more likely to have no common neighbors, which causes *SI-Empty*, because *yt* is sparser than *db*. RM-O performs much better on *yt* than *db* because RM-O optimizes the join plans by reducing the invalid search paths caused by *SI-Empty* but does not consider *ISO-Conflict*.



(a) H_{32D} on *db*. **(b) H_{32D} on *yt*.**

Figure 13: Detailed metrics of RM-O on large queries.

Finding (4): For large queries, (A) RM-O generally performs better than competing algorithms, especially, for sparse graphs; (B) CFL-O and GF-O slightly win RM-O on very dense graphs; (C) The long running queries suffer from the large number of invalid intermediate results; and (D) The *ISO-Conflict* can result in a large number of invalid intermediate results as well, which is ignored by all the competing join plan generation methods.

(a) Q_6 on *eu*. **(b) H_{32D} on *yt*.**

Figure 14: Effectiveness of relation structures and set intersections on the enumeration time.

8.3.3 Relation Structures and Set Intersections. Figure 14 presents the speedup achieved with different combinations of relation structures and SI methods. The baseline is the *Trie* structure with the *Merge* set intersection. *Encoded* outperforms *Hash* and *Trie*. QFilter can benefit from *Encoded*. The speedup with advanced SI methods is limited on *yt* because the neighbor sets are small after filtering. For the same reason, the overhead of QFilter offsets its benefit compared with the methods on uncompressed neighbor sets.

Finding (5): (A) *Encoded* outperforms *Hash* and *Trie*, and accelerates QFilter, the set intersection method on the compact layout; and (B) Advanced set intersection methods significantly improve the performance on small queries, while the effect is limited on large queries compared with the merge-based method.

9 CONCLUSION & FUTURE WORK

In this paper, we study both exploration based and join based subgraph query processing algorithms and propose RapidMatch, a holistic join-based approach with optimizations in filtering, matching order generation, and enumeration. We show that the time complexity of the enumeration in the exploration-based methods can match the maximum output size of a query, which is the same as the methods based on the worst-case optimal join. RapidMatch is based on relational operators and utilizes graph structural information to optimize relation filtering and join plans. We conduct extensive experiments with various kinds of workloads, and show that RapidMatch outperforms both the state-of-the-art join-based and exploration-based methods.

Nevertheless, our work has several limitations, which lead to some interesting research directions. First, our density based join plan generation method can be enhanced by combining with advanced cardinality estimation methods [27], taking other metrics (e.g., the cost of set intersections and the impact of ISO conflicts) into consideration, and extending the plan space (e.g., considering binary joins between sub-structures of Q). Second, we consider parallelizing the end-to-end execution of RapidMatch. It is more challenging than parallelizing direct-enumeration methods [1, 23], because synchronization barriers will be needed between parallel pruning and parallel enumeration. Third, implementing and integrating our proposed techniques in an actual system will make our work more impactful.

ACKNOWLEDGMENTS

The work of Shixuan Sun and Bingsheng He was supported by the grant "Asian Institute of Digital Finance" awarded by National Research Foundation, Singapore and administered by the Infocomm Media Development Authority under its Smart Systems Strategic Research Programme in 2020 (R-703-001-034-279).

REFERENCES

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

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018).

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

[5] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 739–748.

[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] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–1.

[9] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[10] Brian Gallagher. 2006. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching.. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*. 45–53.

[11] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.

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

[13] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*. 1587–1602.

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

[15] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.

[16] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

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

[18] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1566–1577.

[19] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. (2017).

[20] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*. 1231–1245.

[21] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[22] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[23] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[24] Hung Q Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 111–124.

[25] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.

[26] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*. 1–8.

[27] 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 ACM SIGMOD International Conference on Management of Data*. 1099–1114.

[28] Carlos R Rivero and Hasan M Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge and Information Systems* 51, 1 (2017), 61–87.

[29] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[30] Ahmet Erdem Sariyüce, C Seshadhri, and Ali Pinar. 2018. Local algorithms for hierarchical dense subgraph discovery. *Proceedings of the VLDB Endowment* 12, 1 (2018), 43–56.

[31] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. 2015. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*. 927–937.

[32] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

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

[34] 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.

[35] Shixuan Sun and Qiong Luo. 2019. Scaling up subgraph query processing with efficient subgraph matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 220–231.

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

[37] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[38] Todd L Veldhuizen. 2012. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).

[39] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

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

[41] Shijie Zhang, Shirong Li, and Jiong Yang. 2010. SUMMA: subgraph matching in massive graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*. 1285–1288.

[42] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.