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:
doi:10.14778/3425879.3425888

PVldb Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/RapidAtHKUST/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 and G, and Q 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.

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
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/clone-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, the 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 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., $\{u'|(u,u') \in E\}$, $d(u)$ denotes the degree of $u$, i.e., $|N(u)|$. $L$ is a label function associating a vertex to a label in a label set $\Sigma$. Given $V' \subseteq V$, $g(V')$ is the vertex-induced subgraph of graph $g$ 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 1: A comparison of representative subgraph query processing algorithms.**

<table>
<thead>
<tr>
<th>Model</th>
<th>Representative Algorithms</th>
<th>Filter Candidate/Relation</th>
<th>Generate Matching Order/Join Plan</th>
<th>Optimize Data Layout</th>
<th>Accelerate Enumeration Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploration</td>
<td>CFMatch[9]</td>
<td>Native</td>
<td>Greedy</td>
<td>$O(</td>
<td>V(G)</td>
</tr>
<tr>
<td></td>
<td>DP-iso[12]</td>
<td>Native</td>
<td>Greedy</td>
<td>$O(</td>
<td>V(G)</td>
</tr>
<tr>
<td>Join</td>
<td>RapidMatch[24]</td>
<td>Selection</td>
<td>Optimal</td>
<td>$O(\alpha(Q) +</td>
<td>V(G)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Failing Set Pruning</td>
</tr>
</tbody>
</table>

---

177
Table 2: Notations.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>g, Q, G and Q</td>
<td>graph, query, graph, data graph and hypergraph</td>
</tr>
<tr>
<td>V, E, F and S</td>
<td>vertex set, edge set, hyperedge set and label set</td>
</tr>
<tr>
<td>δ(u), 𝐸(𝑢) and 𝑁(𝑢)</td>
<td>degree, label and neighbor of u</td>
</tr>
<tr>
<td>e</td>
<td>edge or hyperedge</td>
</tr>
<tr>
<td>𝑅</td>
<td>relation corresponding to the hyperedge e</td>
</tr>
<tr>
<td>𝑅′, 𝑄′, 𝐺′</td>
<td>the nuclei forest of the neighbors of r, s</td>
</tr>
</tbody>
</table>

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 Kn to denote an r-clique, and define the nucleus and the nuclei forest of a graph g in Definition 2.2. Intuitively, a (r, s) nucleus is a connected subgraph of g that satisfies the density and connectivity constraints and the nuclei forest Tg 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(|Ks(g)|) = O(|E(g)|) time, respectively [30].

Definition 2.2. [31] Let k, r, s be positive integers where r < s, and S be a set of Kn,s in g.

- Kn,s(S) is the number of Sn,s contained in some Sn ∈ S.
- The number of Sn,s containing H ∈ Kn,s(S) is the S-degree of H.
- Two Kn,s, H, H′ are S-connected if there exists a sequence H = H1, H2, ..., Hi, Hi+1, H′ = H′ in Kn,s(S) such that for each i, some Sn ∈ S contains Hi ∪ Hi+1.
- A k-(r, s) nucleus is the maximal union Sn of Kn,s such that (1) S-degree of any H ∈ Kn,s(S) is at least k; and (2) Any H, H′ ∈ Kn,s(S) are S-connected.
- The nuclei forest Tg 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 πR selects the columns in I from R. A natural join R ∧ 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 ⊊ 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 Q = (V, E) where V is a set of vertices and E is a set of hyperedges that are subsets of V. Attributes and relations in the join correspond to vertices and hyperedges, respectively. In this paper, the join is interchangeably denoted by Q = (V, E) and Q := u∈E R, where Re is the relation corresponding to the hyperedge e.

Full Reducer. Given 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 Q. Q is an acyclic query if and only if it has a full reducer [2]. When Q coincides with a simple graph Q, Q is acyclic if Q has no cycles. Algorithm 1 describes the full reducer, which takes an acyclic Q as the input and removes all dangling tuples. It utilizes 2 × (|E(Q)| − 1) semi-joins in total to eliminate dangling tuples.

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 → V′ such that (1) ∀u ∈ V, L(u) = L(f(u)); and (2) ∀e(u, u′) ∈ E, ε(e), f(e(u), f(e(u′)) ∈ E′).

Exploration-based algorithms find all ISOs from Q to G, whereas join-based methods enumerate all HOMs. The HOM allows that a graph with a dangling tuple can map to a HOM with a tuple missing. The third type of algorithms such as TurboIso [14], AFLMatch [7], CECI [6] and SG-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 ∈ V(G) ∧ L(v) = L(u)} for each u ∈ V(Q), and then prunes C(u) based on Proposition 2.4 [7, 12]. After that, it builds an auxiliary structure A maintaining data edges between candidate vertex sets. Given e(u, u′) ∈ E(Q), A(u, u′) = N(u) ∩ C(u′), i.e., the neighbors of v ∈ 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. \( C, \mathcal{A} \leftarrow \) build candidate vertex sets and auxiliary structures;
2. \( \varphi \leftarrow \) generate a matching order;
3. \( \text{Enumerate}(C, \mathcal{A}, \varphi, \emptyset, M, i) \)
4. \( \text{if } i = |\varphi| + 1 \text{ then Output } M, \text{ return}; \)
5. \( u \leftarrow \varphi[i], \mathcal{M}(u) \leftarrow \bigcap_{u \in \mathcal{N}^\varphi(u)} \mathcal{A}^\varphi(M[u]); \)
6. \( \text{foreach } \varphi \in \mathcal{C}_M(u) \)
   
   /* Remove if to find subgraph homomorphisms. */
7. \( \text{if } \varphi \notin M \text{ then } \)
8. \( \text{Add } (u, v) \text{ to } M; \)
9. \( \text{Enumerate}(C, \mathcal{A}, \varphi, M, i + 1); \)
10. \( \text{Remove } (u, v) \text{ from } M; \)

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 \( \varphi \), 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 \( \varphi \) 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 \( \varphi \). \( M \) records the mappings between query vertices and data vertices. Line 6 computes the local candidate vertex set \( \mathcal{C}_M(u) \) where \( \mathcal{N}^\varphi(u) \) denotes the backward neighbors of \( u \) given \( \varphi \) (Definition 2.5). If \( i = 1 \), then \( \mathcal{C}_M(u) = C(u) \). During the enumeration, \( M \) contains \( i \) mappings \((1 \leq i \leq |\varphi|)\) is a subgraph isomorphism from \( Q[\varphi[1 : i]] \) to \( G \) where \( \varphi[1 : i] \) denotes the vertices indexed from 1 to \( i \) in \( \varphi \). 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 \( \varphi \), the backward neighbors \( \mathcal{N}^\varphi(u) \) (forward neighbors \( \mathcal{N}^{\varphi}_F(u) \)) of a query vertex \( u \) is the neighbors of \( u \) that are positioned before (after) \( u \) in \( \varphi \).

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 \( \text{OUT} \) of \( Q \). Researchers developed a tight bound called AGM on \( \text{OUT} \) in terms of the fractional edge cover of \( Q \) (Definition 2.6) [5]. Specifically, the optimal fractional edge cover number \( \rho' \) is the minimum number of all fractional edge covers of \( Q \). \( \text{OUT} \) is bounded by \( O(IN[1]/N) \) where \( |N| \) is the input size [24].

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

Given \( Q \) and \( G \), we construct a join query \( Q \) as follows: (1) \( Q = (V, E) \) where \( V = V(Q) \) and \( E = E(Q) \); and (2) for each \( e(u, u') \in E, R(u, u') = \{(u, v') \in E \land L(u) = L(u) \land L(v') \} \).
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., \( \mathcal{R}_v(u) \)) 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 same matching order. The latest exploration-based algorithms [7, 12] enumerate results based on candidate vertex sets, whereas Algorithm 3 enumerates results based on subset of relations (i.e., \( \mathcal{R}_v(u) \)) or an auxiliary structure (i.e., \( \mathcal{R}_v(u) \)).

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 \( e(v, v') \in E(Q) \) 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 \( e(u, u') \) 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 \( \phi \), 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 \( \phi = (u_1, u_2, \ldots, u_i, \ldots, u_{|\phi(Q)|}) \). 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 \( M = \emptyset \). Based on the analysis in Sections 2.3 and 2.4, \( C_M(u_1) = C(u_1) \) and \( X_M(\{u_1\}) = \bigcap_{e(u, u') \in E(Q)} \pi_{(u')} R(u, u') \). Based on Proposition 3.1, for each \( e(u, u') \in E(Q) \), \( \pi_{(u')} R(u, u') = C(u_1) \). 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 \( \leq k \leq |\phi| \)). We next show that it holds when \( i = k \). In Algorithm

Algorithm 4: RapidMatch

Input: a query graph \( Q \) and a data graph \( G \).
Output: all subgraph homomorphisms (or isomorphisms);
\( Q_C \cdot Q_T \cdot T_2 \cdot T_3 \cdot T_4 \) ← nucleus decomposition on \( Q \);
RelationFilter(\( Q, Q_C \cdot Q_T \cdot T_2 \cdot T_3 \cdot T_4 \));
\( \phi \leftarrow \) JoinPlanGenerator(\( Q, T_2 \cdot T_3 \cdot T_4 \));
RelationEncoder(\( \phi \cdot Q_C \cdot Q_T \));
ResultEnumerate(\( \phi \cdot Q_C \cdot Q_T \));
2. \( C_M(u_k) = \bigcap_{u \in N_v^\top(u_k)} \mathcal{R}_v(u_k) (M[u]) = \bigcap_{u \in N_v^\top(u_k)} (N(M[u]) \cap C(u_k)) \) . On the other hand, \( X_M(\{u_k\}) = X_1 \times X_2 \) where \( X_1 = \bigcap_{u \in N_v^\top(u_k)} \pi_{(u_k)} R(u_k, u) \) and \( X_2 = \bigcap_{u \in N_v^\top(u_k)} R(u : M[u], u_k) \) (see the description of Algorithm 3 in Section 2.4). According to Proposition 3.1, \( X_1 = C(u_k) \) and \( X_2 = \bigcap_{u \in N_v^\top(u_k)} (N(M[u]) \cap C(u_k)) \). 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.

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 \( X_M(\{u_k\}) = \bigcap_{u \in N_v^\top(u_k)} \pi_{(u_k)} R(u_k, u) \) 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 \( Q_T \) and the nuclei forests \( T_2, T_3, T_4 \) of \( Q \) to identify dense subgraphs of \( Q \) as well as their hierarchical relationships. We adopt the nuclei forests \( T_2, T_3 \) and \( T_4 \) because (1) \( T_2 \) contains all query vertices; and (2) previous research on \( T_2 \) [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 \( \phi \) 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 \( G \), \( \phi \) 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
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: $R(u, u') = \{(v, v') | (e(v, v')) \in E(G) \land L(v) = L(u) \land L(v') = L(u')\}$ [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 $Q = \forall e \in E(Q)$, $R_e$ 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 ($\delta$). 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 $V(Q_C)$ to $\delta$ (Line 14) and then select the vertices in $V(Q_C)$ with most neighbors in $\delta$ as the next vertex (Lines 15-16). Lines 7-9 apply the full reducer to the trees along the order of $\delta$. This procedure utilizes the pruning results on $S_u$ to filter relations in $S_{u'}$ where $u$ is positioned before $u'$ in $\delta$ 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 $\delta$.

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 $S_{u_1}, S_{u_2}$ and $S_{u_3}$ respectively along the order of $\delta$, 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 $Q_T \in Q_T$ takes $2 \times (|E(Q_T)| - 1)$ semi-joins. Hence, Lines 4-5 takes $\sum_{Q_T \in Q_T} 2 \times (|E(Q_T)| - 1) \leq 2 \times |E(Q_T)|$ semi-joins. We omit the cost of generating $\delta$ as it is trivial. The full reducer on a tree $S_R$ has $2 \times |d(u) - 1|$ semi-joins. Therefore, Lines 7-11 take $4 \times \sum_{u \in V(Q)} (|d(u) - 1|) \leq 8 \times |E(Q)|$ semi-joins. A semi-join on relations $R$ and $R'$ takes $|R| + |R'|$ time. Then, the time complexity of Algorithm 5 is $O(|E(Q)| \times |E(G)|) + (2 \times |E(Q)|) \times |E(Q)|) = O(|E(Q)| \times |E(G)|)$.

**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 of Algorithm 5 the relations can be updated, then $R(u, u') \bowtie R(u, u'')$ will be equal to $R(u, u')$ given $u \in V(Q)$ and $u'))) \bowtie R(u, u')$. Therefore, $C(u)$ will be equal to $\pi(u)R(u, u')$. For each $u \in V(Q)$, 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 $(v, u') \in R(u, u')$ 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 $\varphi$ that positions the core vertices $V(Q_C)$ before the non-core vertices (i.e., $V(Q) - V(Q_C)$). Moreover, $\varphi$ is connected, i.e., $\forall 1 \leq i \leq |\varphi|, Q(\varphi[1 : i])$ is a connected graph. Then, a non-core vertex has exactly one backward neighbor in $\varphi$ based on Definition 2.1. RapidMatch first executes $Q_C := \bigcup_{e \in E(Q_C)} R_e$ with LFTJ taking $\varphi[1 : |V(Q_C)|]$ as the matching order, and then evaluates $Q_C :\bowtie Q_T$ where $Q_T := \bigcup_{e \in E(Q_T)} R_e$ 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 $\varphi$ based on the cost model $\text{cost}(\varphi) = \sum_{i=1}^{|\varphi|}|R(Q(\varphi[1 : i]))|$, 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 estimator can result in very ineffective join plans. Due to the difficulty of the cardinality estimation, we optimize $\varphi$ 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 $\varphi$ by terminating such invalid search paths at an early stage. $X_M(\{u\})$ is computed by $\bigcap_{u' \in N_M(\{u\})} R(u') : M[u'[1 : i]]$ as discussed in Section 2.4. Then, the join plan $\varphi$ putting query vertices with more backward neighbors at the beginning tends to perform better, and we define the utility function as $\text{utility}(\varphi) = \sum_{i=1}^{|\varphi|} \gamma_{i,j}^\varphi |N[i]|$. 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 $\gamma_{i,j}^\varphi |N[i]|$ 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 $\varphi$ 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.

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 $\varphi$ via the Traverse function. We add vertices in $e(u, u') \in E(X)$ 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 $T \in T_X$, we set the parent of the root node of $T$ as the smallest (by the vertex
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 \( \{u_5, u_4, u_3, u_2\} \) 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 \( X \) has been visited, then return (Line 17). Otherwise, mark it as visited. If all vertices in \( X \) exist in \( \varphi \), then jump to its parent (Line 19). As the density of \( X \) is generally lower than its children, Lines 20-24 first consider its children. We prioritize the child \( X' \) having stronger connection with \( \varphi \), which is defined in Equation 1. \( |\text{SP}(\varphi, X')| \) is the length of the shortest path from \( X[\varphi] \) to \( X' \) through vertices in \( X \). The child that has more common vertices with \( \varphi \) or is close to \( \varphi \) scores a high value.

\[
\text{connection}(\varphi, X') = |\varphi \cap V(X')| + \frac{1}{1 + |\text{SP}(\varphi, X')|}. 
\]

Example 6.3. Continue Example 6.2 and we traverse from \( \mathcal{X}_6 \) of \( T \). We first add \( u_5, u_6 \) to \( \varphi \) since \( e(u_5, u_6) \) has the maximum degree sum in \( \mathcal{X}_6 \). We add all vertices in \( \mathcal{X}_6 \) to \( \varphi \) and traverse to its parent \( \mathcal{X}_5 \). \( \mathcal{X}_5 \) is the child of \( \mathcal{X}_6 \) and the connection between \( \varphi \) and \( \mathcal{X}_5 \) 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 \( \varphi \). Based on rules breaking ties, we add \( u_7 \) to \( \varphi \) since it has a higher vertex degree than \( u_6 \). Next, we add \( u_1 \) to \( \varphi \) before \( u_6 \) because \( u_6 \) has more neighbors in \( \varphi \). After adding all vertices in \( \mathcal{X}_7 \) to \( \varphi \), the traversal procedure leaves \( \mathcal{X}_7 \) and continues the process until all query vertices are added into \( \varphi \).

6.3 Analysis of Join Plan Generator

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

**Proposition 6.4.** \( \varphi \) 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 \( O(|E(Q)| \times \sum_{X \in T} |X.children|^2) = O(|E(Q)| \times |T|^2) \) 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 disjointment property \cite{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 \( \mathcal{T}_{2,3}, \mathcal{T}_{3,4} \).

**Time of joins.** RapidMatch evaluates \( R(Q_C) \cong (\forall e \in E(Q_F) R_e) \) with a sequence of hash joins along \( \varphi \). Benefiting from RelationFilter, the joins satisfy Proposition 6.5. Therefore, RapidMatch has the worst-case optimality because it evaluates \( Q_C \) with LFT, 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 \cite{39}.

**Proposition 6.5.** Each join of \( R(Q_C) \cong (\forall e \in E(Q_F) R_e) \) along \( \varphi \) increases the size of intermediate results.

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

**Discussions.** CFLMatch \cite{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 \((u_1 \cdots u_{10})\) in Figure 3a, but \( Q_F \) has only one vertex \((u_{11})\). The decomposition method of CFLMatch can put \( u_{11} \) after \( u_{10} \) in \( \varphi \) but cannot obtain the dense structures of the \( Q_C \). In contrast, we generate \( \varphi \) 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 $u \in N^+_v (u')$, 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 $(\text{pos}(v), \text{pos}(v'))$ into $R'(u, u')$; otherwise, skip $(v, v')$. Furthermore, given $v \in R(u) - \pi(u) R(u, u')$, add $(\text{pos}(v), \emptyset)$ 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 $e(u, u') \in E(Q_T)$, we build a hash index of $R(u, u')$ to serve the hash join.

The time complexity of optimizing the relation data layout is $O(\sum_{e \in E(Q_T)} |R_e| + \sum_{e \in E(Q_c)} |R_e| \times \log |R_e|)$, and stores it as a trie takes $O(\log |R|)$ 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 IDs 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 Rapid-Match in this section.

8.1 Experimental Setup

Algorithms Under Study. We compare the performance of Rapid-Match (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/HeartGraph/ceci-release, Last accessed on 2020/08/15.

Table 3: Properties of real-world datasets.

| Dataset    | Name   | |V| | |E| | |t| |d| |
|------------|--------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Biology    | Human  | 36               | 9,074           | 36,282          | 41               | 38              | 10              |
| Lexical    | WordNet| 76,597           | 238,399         | 71               | 30               | 14              |
| Citation   | US Patents| 5,774,768       | 16,315,942      | 20               | 5.6             |
| Social     | LiveJournal| 4,367,571        | 62,512,248      | 4                | 3.75            |
|            | Youtube | 2                | 1,134,880       | 2,987,624        | 24               | 3.3             |
|            | DBLP   | 317,080          | 1,094,808       | 13               | 3.6             |
| Web        | eu2005 | 662,664          | 16,135,408      | 4                | 37.4            |

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 compiled by g++ 7.3.1 with the -O3 flag enabled and the JAVA code is compiled 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 $\lambda$ 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 $\Sigma$ 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 $\Sigma$ 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 $\Sigma$ 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\(^4\).


\(^4\)Figure 4: Query graphs.
set (i.e., $d(Q) > 3$) 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 $H_D$ and $H_S$, respectively.

**Metrics.** To keep consistent with previous research, we find all results for small queries [1, 23], while terminate the query after finding $10^3$ 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 encoding 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 $\sum_{r \in Q} |r|$, 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 $Q_5$ 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.

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., $\frac{1}{|Q|} \sum_{Q \in H} t(Q)$ 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 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-iteration 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 pre-processing-iteration 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 $H_D$ against $yt$ because the overhead of the pruning in the pre-processing-iteration methods can offset the benefit on the query with a short running time. However, GF runs much slower than competing algorithms on $H_D$ against $yt$ because it generates ineffective join plans for a number of queries.

**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 sub-graph 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.
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 Q₃ 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 dh, 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.

**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 Q₄ 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.](image1)

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 Q₂.

**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.

Moreover, we collect the detailed metrics of RM-O in Figure 13. We relabel the IDs 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.

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.

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 ℋ). 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


