Discovering Neighborhood Pattern Queries by Sample Answers in Knowledge Base

Jialong Han*, Kai Zheng†, Aixin Sun*, Shuo Shang‡, and Ji-Rong Wen§¶
*School of Computer Engineering, Nanyang Technological University, Singapore
†School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia
‡ China University of Petroleum, Beijing, China
§School of Information, Renmin University of China, Beijing, China
jialonghan@gmail.com, kevinz@itee.uq.edu.au, axsun@ntu.edu.sg, jedi.shang@gmail.com, jirong.wen@gmail.com

***Abstract*—Knowledge bases have shown their effectiveness in facilitating services like Web search and question-answering. Nevertheless, it remains challenging for ordinary users to fully understand the structure of a knowledge base and to issue structural queries. In many cases, users may have a natural language question and also know some popular (but not all) entities as sample answers. In this paper, we study the *Reverse top-k Neighborhood Pattern Query* problem, with the aim of discovering structural queries of the question based on: (i) the structure of the knowledge base, and (ii) the sample answers of the question. The proposed solution contains two phases: filter and refine. In the filter phase, a search space of candidate queries is systematically explored. The invalid queries whose result sets do not fully cover the sample answers are filtered out. In the refine phase, all surviving queries are verified to ensure that they are sufficiently relevant to the sample answers, with the assumption that the sample answers are more well-known or popular than other entities in the results of relevant queries. Several optimization techniques are proposed to accelerate the refine phrase. For evaluation, we conduct extensive experiments using the `DBpedia` knowledge base and a set of real-life questions. Empirical results show that our algorithm is able to provide a small set of possible queries, which contains the query matching the user question in natural language.**

I. Introduction

With the effectiveness of utilizing knowledge bases in various applications, the problems of harvesting, storing, and accessing structured knowledge are being actively investigated. Several real-world *knowledge bases* have been built to describe entities (*e.g.,* persons, locations, and books) and their relations (*e.g.,* born in, located in, and written by), including `DBpedia` [1], `Freebase` [3], `YAGO` [23], and `NELL` [6], to name a few. A knowledge base is commonly modeled as a directed labeled graph. Figure 1 depicts an example knowledge base, where entities are represented as nodes, their types as node labels, and relations as labeled edges.

Although a few structured query languages (*e.g.,* `SPARQL` for `RDF` data, `MQL` for `Freebase`, and `Cypher` for `neo4j` graph database) have been developed, it remains challenging for most users to formulate a query using these languages. Users have to learn the query language syntaxes and also be familiar with the structure of knowledge bases, *e.g.,* the types of entities and their relations. On the other hand, due to the inherent complexity and vagueness of natural language,

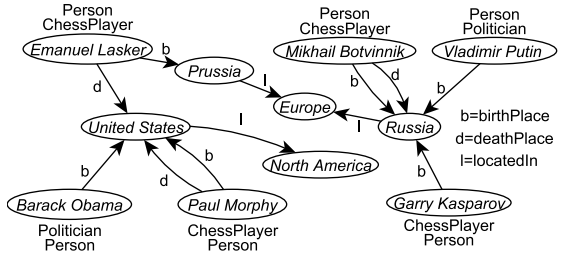


Fig. 1: A toy knowledge base. Strings inside nodes are IDs.

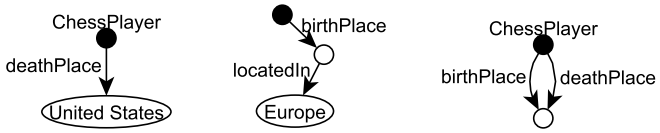
it is hard to fully map human language to structural query languages. According to the evaluation[1] of the third Question Answering over Linked Data contest (`QALD-3`), all six competing systems generally did well on natural language questions whose structural queries had simple shapes, like the ones shown in Figures 2a and 2b. However, for the question "which daughters of British earls died in *the same* place they were born in?" whose query resembles Figure 2c, only three systems managed to respond. Among them, only one survived from getting a zero F_1 score. In short, it is hard for users to directly formulate structural queries and it is also hard to fully map a natural language question to a structural query.

When a user raises a question whose answer consists of a list of entities, it is often the case that she can provide one or two entities as sample answers. Compared with other unknown and wanted entities, these sample answers may be more well-known or of higher popularity (*e.g.,* persons with more fame, or movies with better box-office records), thus are easier to recall. The following problem becomes interesting: *can we use the given examples as clues to discover the structured query*?

Motivating Example. Consider the toy knowledge base in Figure 1, where all person nodes are ranked in the first column of Table I by their popularity. When a user asks a question "which chess players died in the same place they were born in?", she lists Mikhail Botvinnik as an example answer. Based on this toy knowledge base, numerous structural queries may return persons as answers, as shown in Figure 2. Query 2a is invalid, because M. Botvinnik is not in its result set. Query 2b is a false positive, because if the user meant to ask for persons who was born in Europe, she would have probably come up with V. Putin as a sample answer who is more famous than

¶Corresponding author.

[1]http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/3/documents/qald3_results.pdf



(a) Chess players dying in the United States (b) (Persons) born (somewhere) in Europe (c) Chess players born and dying in the same (place)

Fig. 2: Three *neighborhood pattern queries*, each containing a solid node, called a *pivot*. When executed, they return all entities in the knowledge base that match their pivots.

TABLE I: Motivating example

Popularity order	Query 2a	Query 2b	Query 2c
B. Obama	E. Lasker	V. Putin	M. Botvinnik
V. Putin	P. Morphy	G. Kasparov	P. Morphy
G. Kasparov		E. Lasker	
E. Lasker		M. Botvinnik	
M. Botvinnik			
P. Morphy	Rank: $+\infty$	Rank: 4	$\surd$ **Rank: 1**

M. Botvinnik among persons born in Europe. An appropriate algorithm should return all queries like 2c, which not only cover the example answers, but also consider the popularity ranks of the sample answers. ■

In this paper, we formulate the above query discovery problem as the ***R**everse top-**k** **N**eighborhood **P**attern **Q**uery* problem (or RkNPQ for short). Here, *neighborhood pattern queries* refer to structural queries that look for entities by describing their neighborhood structures in the knowledge base. Formally, given some example entities I in a knowledge base D, we want all neighborhood pattern queries q such that: 1) when evaluated against D, the results $D(q)$ covers I; and 2) when entities in $D(q)$ are ranked according to their relative popularity, the given entities I all appear in the top-k positions.

Discovering SQL queries using full or partial result tables has been investigated in [21], [24], [30]. Hence, the problem of reverse engineering queries using results is not new. However, we note that their research differ from ours from three aspects.

1) Problem Setting. The solutions that require full result table as input [24], [30] do not fit in the knowledge base setting. If a user knows all answers of her question, there is no need to find the query. If user input is only regarded as a partial answer like in [21], the information that a user tends to list well-known entities as example answers is not exploited. Without considering entity ranking, many more queries would be returned. For instance, in the motivating example, both queries 2b and 2c will appear as discovered queries.

2) Solution Framework. Relational databases have well-defined schemas. Therefore, the schema of the result table may serve as clues to guide the query search. Tuples in the result table are only used to refine the search space. In our problem, a knowledge base often has weak (or even no) schema. The one-column results (*i.e.,* entities) provide little information on the potential shape of the query. To exploit such limited information, we map the problem to a frequent neighborhood pattern mining problem [9], establishing a preliminary search space for further refinement.

3) Candidate Refinement. In the full or partial result setting, to confirm that a candidate query is a true positive, the query itself need not be evaluated; the result tuples are verified against the query. In our problem, query evaluation is inevitable, because the top-k condition poses extra constraints on the ranking positions of the input entities in the answers. To deal with this, we identify *indicator answers* instead of full answers as the only factor affecting the top-k condition. We also propose two optimization techniques, *shared evaluation* and *partial evaluation*, to further reduce unnecessary overheads in evaluating indicator answers.

Reverse query problems have also been extensively investigated for other data types or settings. Examples include reverse nearest neighbor queries [12], reverse skyline queries [7], reverse top-k queries [26], and reverse top-k RWR (Random Walk with Restart) queries [29], to name a few. Compared with them, our problem is more related to graph pattern matching, which has a fundamentally different nature. To the best of our knowledge, our work is the first attempt to address reverse query problems for graph pattern matching queries. We summarize the contributions of this paper as follows:

1) We formulate the reverse top-k neighborhood pattern query problem, which helps users formulate structured queries in knowledge bases using popular partial answers.
2) We propose a filter-refine framework for solving RkNPQ, and show that the filter sub-problem is reducible to the problem of mining a class of frequent patterns.
3) We explore three tailored optimizations in the refine phase, namely shared evaluation, indicator answers, and partial evaluation, to accelerate reverse query processing.
4) We conduct extensive experiments to verify the effectiveness and efficiency of the proposed solution.

The rest of this paper is organized as follows. We formally define the research problem in Section II. In Section III, we present the filter-refine framework for RkNPQ, and describe how the filtered search space is organized. Optimizations of the refine phase are discussed in Section IV, and the issue of redundant queries is addressed in Section V. Our experiments are reported in Section VI. After literature survey in Section VII, we conclude this study in Section VIII.

II. Problem Definition

In this section, we first introduce basic graph terminologies to model knowledge bases. By adopting the concept of *pivoted subgraph isomorphism* in [9], we use *neighborhood pattern queries* to model semantics of natural language questions which require lists of entities as sample answers. We then define the RkNPQ problem.

A. Preliminaries

Definition 1 (Directed labeled graph): A directed labeled graph is a 5-tuple $G = \langle V, L_V, E, \Sigma_V, \Sigma_E \rangle$, where 1) V is the set of all nodes; 2) Σ_V and Σ_E denote label names used to form node and edge labels, respectively; 3) $L_V \subseteq V \times \Sigma_V$ is the set of all node labels; 4) $E \subseteq V \times V \times \Sigma_E$ is the set of labeled edges.

When using directed labeled graphs to model knowledge bases, we uniquely map entities to nodes, and entity types to node labels. E.g., $(v_1, t) \in L_V$ indicates that entity v_1 has type t. Note that an entity may have multiple types, represented by multiple node labels. We use labeled edges to represent

relations between two entities. E.g., $(v_1, v_2, r) \in E$ indicates that relation r holds between entities v_1 and v_2. When multiple relations exist between two entities, we use parallel edges with one label on each to represent them, instead of an edge with multiple labels. Similar as node types, we also treat node IDs as node labels for a unified formulation, though each node may have only one ID label. We define the size of G as $|G| = |L_V| + |E|$, *i.e.,* the number of node labels and labeled edges. When the context is clear, we will mix the use of graph terminologies and knowledge base terminologies.

Definition 2 (Pivoted graph [9]): A pivoted graph is a tuple $\mathcal{G} = \langle G, v_p \rangle$, where G is a directed labeled graph, and $v_p \in V(G)$ is a chosen node in $\mathcal{G}$, called the pivot of $\mathcal{G}$.

Definition 3 (Pivoted subgraph isomorphism [9]): A pivoted graph $\mathcal{G}_1$ is pivoted subgraph isomorphic to $\mathcal{G}_2$, denoted by $\mathcal{G}_1 \subseteq_p \mathcal{G}_2$, if there exists an injective mapping $f : V(\mathcal{G}_1) \to V(\mathcal{G}_2)$ such that f preserves all node and edge labels, and maps the pivot of $\mathcal{G}_1$ to that of $\mathcal{G}_2$. Formally, 1) $\forall v_1 \neq v_2$, $f(v_1) \neq f(v_2)$ holds; 2) $\forall (v, t) \in L_V(\mathcal{G}_1)$ we have $(f(v), t) \in L_V(\mathcal{G}_2)$; 3) $\forall (v_1, v_2, r) \in E(\mathcal{G}_1)$ we have $(f(v_1), f(v_2), r) \in E(\mathcal{G}_2)$; 4) $f(v_p(\mathcal{G}_1)) = v_p(\mathcal{G}_2)$.

Compared with traditional subgraph isomorphism, pivoted subgraph isomorphism has a different node-oriented semantics. When seeking for an isomorphic mapping, the pivot of $\mathcal{G}_1$ should be mapped to that of $\mathcal{G}_2$. Later we will show that it helps depict the question/answer relationship between natural language questions and entities, and the sub-super relationship between questions.

Definition 4 (Neighborhood pattern query): A neighborhood pattern query q is a connected pivoted graph. Given a graph database D, the result set of q on D, denoted by $D(q)$, consists of all node instances from D that match pattern q's pivot, *i.e.,* appear at q's pivot position; $D(q) = \{v \in V(D) \mid q \subseteq_p \langle D, v \rangle\}$.

We further define the size of a neighborhood pattern query to be the size of the labeled graph it is based on, and its radius to be the largest distance (ignoring edge direction) between its pivot and any other node.

Figure 2 shows three example neighborhood pattern queries, where the three solid nodes stand for the three pivots. The result sets of them evaluated against the knowledge base shown in Figure 1 are listed in Table I. Note that a query may have *constant nodes*, *i.e.,* nodes with node IDs as labels, which indicates that they should only be bound to the corresponding nodes in the knowledge base.

B. Problem Statement

Definition 5 (Top-k neighborhood pattern query): Let $\prec$ be an order representing the relative popularity of nodes in D. $v_1 \prec v_2$ means that v_1 has the same type (*e.g.,* person, country, or movie) with, and is more popular than v_2. We assume that for a neighborhood pattern query q, $D(q)$ consists of entities with the same type. By abusing the notation of set, the top-k result set of a query q, denoted by $D_k(q)$, is the top-k subset of $D(q)$ when the entities in $D(q)$ is ranked by $\prec$.

In a knowledge base, it is natural that some entities are more well-known, and are accessed or queried more often than others. If a query returns a long list of entities, a user may be interested in the most famous ones. Conversely, when a user cannot formulate a structured query but can list several entities as sample answers, there is a high probability that these entities are among the most popular ones in the answer set. Based on this observation, we propose the *reverse top-k neighborhood pattern query* problem.

Problem Statement. Given a set of nodes $I \subset V(D)$, a reverse top-k neighborhood pattern query on I, denoted by $D_k^{-1}(I)$, returns all neighborhood queries q such that the top-k result set of q contains all nodes in I. Formally, $D_k^{-1}(I) = \{q \mid I \subseteq D_k(q)\}$. By definition, $|I| \leq k$.

The Expressivity of Neighborhood Pattern Queries. When mapped to `SPARQL`, neighborhood pattern queries are equivalent to a constrained fragment, where the `SELECT` clause contains only one variable and the `WHERE` clause contains no advanced operations such as regular expressions and range selections. For example, the query in Figure 2c is equivalent to the following `SPARQL` query (`URI` prefixes omitted):

```
SELECT ?uri WHERE
{
   ?uri :type :ChessPlayer .
   ?uri :birthPlace ?place .
   ?uri :deathPlace ?place
}
FILTER(?uri != ?place)
```

Note that the additional `FILTER` condition is caused by Definition 3, where different variables in the query are required to be bounded to different entities. In the following parts of the paper, we will drop it for brevity though its existence should not be neglected. Despite those limitations, we argue that neighborhood pattern queries are essentially a significant subclass of knowledge base queries for two reasons. First, unlike structured queries, real-world questions often require a list of entities as answers, rather than a list of entity tuples. Second, constraints on the type of entities and relations between entities could be sufficiently expressed by the triples in the `WHERE` clause.

The Popularity Order $\prec$. When ranking entities in $D(q)$ to obtain the top-k results of a neighborhood query q, we assume that entities in $D(q)$ are of the same type, and that their relative popularity is independent of q. These assumptions are practical for neighborhood pattern queries. The type of a query's pivot is decided explicitly by node labels on it, or implicitly by labeled edges associated to it. For example, all queries in Figure 2 return only person entities as answers. Moreover, the match between an entity and a query is binary, so the popularity rank of entities is not query-sensitive. We admit that, in practice, the popularity order may be subjective and contextual, and takes efforts to estimate (*e.g.,* based on query logs, entity related page viewing frequency). Considering the current scope of this paper, we adopt PageRank as a simple estimation and leave the ideal estimation for future study.

III. Solution Framework

In this section, we adopt the filter-and-refine scheme to discuss a possible baseline solution to the reverse top-k neighborhood pattern query problem. Specifically, by dropping

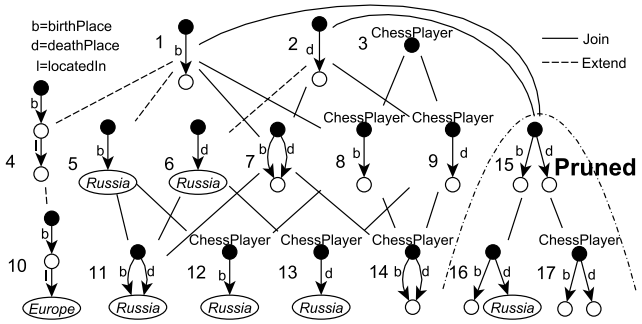


Fig. 3: The search space of $D^{-1}(\{M.\ Botvinnik\})$

Algorithm 1 The RkNPQ framework

```
1: function RkNPQ(D, I, k)
2:     Q_1 ← ⋂_{v∈I} Level_1_Queries(v); n ← 2;
3:     while Q_{n-1} ≠ ∅ do                                  ▷ Filter phase
4:         for all q_i, q_j ∈ Q_{n-1} do                     ▷ Join operation
5:             q ← Join(q_i, q_j);
6:             Discard q if does not cover I or is discovered before;
7:             Q_n ← Q_n ∪ {q};
8:         end for
9:         for all q_i ∈ Q_{n-1} do                          ▷ Extend operation
10:            q ← Extend(q_i); Q_n ← Q_n ∪ {q};
11:        end for
12:        n ← n + 1;
13:    end while
14:    return {q ∈ ⋃_{n≥1} Q_n | I ⊆ D_k(q)};               ▷ Refine phase
15: end function
```

the top-k constraint in the problem definition, we show that the relaxed problem, *i.e., reverse neighborhood pattern query* problem, is reducible to the frequent neighborhood pattern mining problem [9] with a constrained input. This reduction filters out the queries whose answers do not cover all input nodes in I, and provides an approach to organizing and searching the surviving candidate queries. We further refine those candidates by evaluating their answer sets to prune false positives and make up for the top-k constraint.

A. Filter

By dropping the top-k constraint in the problem definition, we have a relaxed version of the problem: Given a set of input nodes $I \subset V(D)$, the reverse neighborhood pattern query of I, denoted by $D^{-1}(I)$, returns all neighborhood pattern queries q whose result set covers I, *i.e.,* $D^{-1}(I) = \{q \mid I \subseteq D(q)\}$.

Proposition 1: Given a database D, for any $I \subseteq V(D)$, we have $D_k^{-1}(I) \subseteq D^{-1}(I)$.

We ignore the proof because it is obvious. This proposition actually enables a filter-and-refine approach to our problem. If we are able to search the space of $D^{-1}(I)$, it is possible to check each $q \in D^{-1}(I)$ on whether $q \in D_k^{-1}(I)$ holds. In fact, the FNM algorithm proposed in [9] for mining frequent neighborhood patterns in graph database can be utilized. Given a graph database D, a subset $V_0 \subseteq V(D)$, and a threshold τ, FNM outputs all neighborhood patterns, each of which matches at least τ nodes in V_0. Noticing that the neighborhood patterns in FNM are essentially equivalent to neighborhood pattern queries in this study, we constrain the input of FNM as follows: we set $V_0 = I$ and $\tau = |I|$, and run FNM on the given database D. Because this parameter configuration of V_0 and τ requires that all mined patterns match $|I|$ nodes in I, *i.e.,* all nodes in I, we simply get the exact $D^{-1}(I)$ after running FNM algorithm.

Through the above parameter configuration, FNM does a level-wise search on the query space of $D^{-1}(I)$ as a query lattice[2], defined as follow:

Definition 6 (Sub-query and query lattice): A query q_1 is a sub-query of q_2 if $q_1 \subseteq_p q_2$, and super-query vice versa. Given a set of queries, we organize them level-wisely according to the query size, and connect any two queries q_1 and q_2 if $q_1 \subseteq_p q_2$ and $|q_1| + 1 = |q_2|$. We call the level-wise arrangement a query lattice.

[2]Strictly speaking, the lattice discussed here does not meet the conditions of a mathematical lattice. However, we abuse the concept here only for reference within this paper.

In Figure 3, we show the $D^{-1}(\{M.\ Botvinnik\})$ lattice. For the sake of this example, not all queries are listed. Queries at the same level have the same size, *e.g.,* queries 1-3 have a size of one, and queries 4-9 have a size of two.

FNM generates this lattice in the following manner. It initializes with level-1 queries by scanning I and finding all common incoming edges, outgoing edges, and node labels. Then it goes to higher levels. At each level, it generates candidates belonging to the level by carrying out two operations, *join* and *extend*, on queries at the previous level. The joining operation is attempted between every pair of queries (self-join is possible, though it is not demonstrated in this example). For example, query 7 is obtained by joining queries 1 and 2, and query 14 could be generated by joining queries 7 and 9. Moreover, for path-like queries such as queries 1 and 4, FNM checks I and D to perform an additional extend operation to lengthen it or terminate it by a node label. In Figure 3, all extend operations are marked by dotted lines. For example, by extending query 1 we get queries 4 and 5, and we get query 10 by extending query 4. The extend operation is vital to the completeness of the results because queries like 4 and 10 cannot be obtained by joining two smaller queries.

We emphasize that the join operation may introduce false positives. Therefore, every query obtained by join should be checked on whether its answers indeed covers I by $|I|$ pivoted subgraph isomorphism checks (denoted by PSI checks for short). For example, in Figure 3, the join of queries 1 and 2 not only produces query 7 (persons born and dying in the same place) as Figure 3 shows, but also produces query 15 (persons born and dying in different places). However, after checking the database D in Figure 1 and confirming that it does not cover the node M. Botvinnik, FNM immediately discards query 15 to avoid checking its super-queries such as queries 16 and 17, which do not cover M. Botvinnik, neither.

Like all other frequent pattern mining algorithms, the result size and running time of FNM may be exponential in the worst case. However, it ensures a complete traversal of $D^{-1}(I)$, rendering the filter stage a safe one.

B. Refine

After obtaining $D^{-1}(I)$, it might be tempting to refine it by verifying $I \subseteq D_k(q)$ for every query $q \in D^{-1}(I)$ in the following manner: 1) evaluate $D(q)$ using SPARQL or some graph query engines like neo4j, gStore [34], and

Algorithm 2 RkNPQ−S (with **S**hared evaluation)

```
1: function RkNPQ-S(D, I, k)
2:     Q_1 ← ⋂_{v∈I} LEVEL_1_QUERIES(v); n ← 2;
3:     for all q ∈ Q_1 do
4:         q.RS ← D(q)                                   ▷ Initialize results;
5:     end for
6:     while Q_{n-1} ≠ ∅ do
7:         for all q_i, q_j ∈ Q_{n-1} do
8:             q ← JOIN(q_i, q_j);
9:             Discard q if does not cover I or discovered before;
10:            q.RS ← PSICHECKS(q, q_i.RS ∩ q_j.RS);      ▷ Maintain results
11:            Q_n ← Q_n ∪ {q};
12:        end for
13:        for all q_i ∈ Q_{n-1} do
14:            q ← EXTEND(q_i); Q_n ← Q_n ∪ {q};
15:            q.RS ← PSICHECKS(q, q_i.RS);               ▷ Maintain results
16:            Q_n ← Q_n ∪ {q};
17:        end for
18:        n ← n + 1;
19:    end while
20:    return {q ∈ ⋃_{n≥1} Q_n | I is ranked top-k in q.RS};
21: end function
```

JENA−TDB[3]; 2) rank nodes in $D(q)$ to get $D_k(q)$; 3) check whether $I \subseteq D_k(q)$ holds. This results in an preliminary algorithm like Algorithm 1. However, we note that such a refine phase contains duplicate computations, thus is less efficient. In the following section, we will discuss observations on such inefficiency, and propose three optimizations.

IV. Optimizations

We now discuss three optimizations of the refine phase, namely *shared evaluation*, *indicator answers*, and *partial evaluation*. The first two are orthogonal to and compatible with each other, while the third one depends on the first two.

A. Shared Evaluation

When refining $D^{-1}(I)$, the candidate queries are evaluated in a batch manner. Noticing that their shapes heavily overlap with each other, we propose the following observation:

Observation 1: Queries in $D^{-1}(I)$ are highly overlapped in shape, thus may share computations when evaluated on D.

For example, in Figure 3, after obtaining the answers of query 9 "chess players with a known death place", we know that G. Kasparov is not in its answers. Therefore, when checking any super-query of query 9 like queries 13 and 14, G. Kasparov could be ignored because it cannot appear in their answers. Based on this, we propose the following property of sub-super queries to reduce duplicate computations:

Proposition 2: $D(q_1) \supseteq D(q_2)$ if $q_1 \subseteq_p q_2$.

Proof: $\forall v \in D(q_2)$, $q_2 \subseteq_p \langle D, v\rangle$ holds. Based on the transitivity[4] of pivoted subgraph isomorphism, we have $q_1 \subseteq_p \langle D, v\rangle$, or $v \in D(q_1)$. Then $D(q_1) \supseteq D(q_2)$ is immediate. ■

Proposition 2 ensures that $D(q)$ for large queries needs not to be evaluated from scratch. Instead, the result sets of large queries can be obtained by checking those of their sub-queries. They are then kept for larger queries' use, detailed in Algorithm 2.

[3]http://jena.apache.org/documentation/tdb/index.html

[4]See proof in [9].

In Algorithm 2, we start with all level-1 queries that cover I. At Line 4, we calculate the result set $q.RS$ for every query q, and attach it to q. This only involves one look-up in the indices built on all node and edge labels. Then the algorithm goes to higher levels of the query lattice, which involves joining and extending smaller queries to generate larger ones.

At higher levels, according to Proposition 2, it is sufficient to generate $q.RS$ by scanning the results of its sub-queries and picking out those that q matches. This is done by the function $PSIChecks(q, V)$, which sequentially performs PSI checks between q and nodes in V and returns those that pass the check. At Line 10, when a query q is generated by joining two of its sub-queries, we only need to pick out $q.RS$ from the intersection of the two sub-queries' results. Similarly, at Line 15, $q.RS$ is generated by calling function $PSIChecks$ on that of its only sub-query. At Line 20, the top-k condition for each q is verified by ranking and checking $q.RS$.

B. Indicator Answers

In Section III-B, we discussed evaluating the whole result set $D(q)$ using SPARQL or graph query engines. However, this approach does not exploit the following fact:

Observation 2: Nodes that are less popular than the least popular node in I do not affect the rank of all nodes in I, thus may be ignored when evaluating the results of q.

For example, assume we are processing the reverse query $D_1^{-1}(\{M.\ Botvinnik\})$, *i.e.,* finding all queries where M. Botvinnik is ranked at the first position. According to Table I we have $M.\ Botvinnik \prec P.\ Morphy$, therefore for any query q, whether $P.\ Morphy \in D(q)$ holds will not have any influence on the rank of M. Botvinnik in $D(q)$, since it is always ranked below M. Botvinnik. Based on this observation, we propose evaluating the *indicator answers* instead of the full answers of each query, defined below:

Definition 7 (Indicator Answers): For any $q \in D^{-1}(I)$, we call $IA(q)$ the indicator answers of q, which consists of all the nodes v satisfying: 1) v appears in the answer set of q, 2) v is more popular than the least popular node in I (denoted by $inf(I)$, which is short for infimum), and 3) v does not appear in I. Formally, $IA(q) = \{v \mid v \in D(q) \wedge v \prec inf(I) \wedge v \notin I\}$.

Intuitively, the indicator answers of q are nodes that are ranked above or between nodes of I in $D(q)$. In other words, they are precisely nodes that may affect the rank positions of I in $D(q)$. Therefore, if $|IA(q)|$ is small enough, it may enable nodes in I to be ranked high in $D(q)$, and cause $q \in D_k^{-1}(I)$. The following proposition shows how small $|IA(q)|$ should be to make $q \in D_k^{-1}(I)$ hold.

Proposition 3: For any $q \in D^{-1}(I)$, $q \in D_k^{-1}(I)$ if and only if $|IA(q)| \leq k - |I|$.

Proof: $(|IA(q)| + |I|)$ is in fact the rank position of $inf(I)$ in $D(q)$. Therefore, the sufficiency and necessity of the proposition is clear. ■

The indicator answer optimization is orthogonal to the shared evaluation optimization. To implement it alone, we need to customize SPARQL or graph query engines. The additional constraints in Definition 7 need to be pushed down to the early stages of query evaluation in order to reduce intermediate

Algorithm 3 Partial evaluation

```
1: procedure PARTIALEVAL(q, V)
2:     for all v ∈ V do
3:         V ← V \ {v}
4:         if q ⊆_p ⟨D, v⟩ then
5:             q.IA ← q.IA ∪ {v}
6:             if |q.IA| > k − |I| then
7:                 break
8:             end if
9:         end if
10:    end for
11:    q.ĨA ← V
12: end procedure
```

results. However, the following proposition points out that indicator answers share similar properties with ordinary result sets, and need not be evaluated from scratch:

Proposition 4: Given I, for any $q_1 \subseteq_p q_2$, we have $IA(q_1) \supseteq IA(q_2)$.

Proof: For any $q_1 \subseteq_p q_2$, we have $D(q_1) \supseteq D(q_2)$. Therefore, $D(q_1) \cap \{v \mid v \prec inf(I)\} \setminus I \supseteq D(q_2) \cap \{v | v \prec inf(I)\} \setminus I$ holds. Then $IA(q_1) \supseteq IA(q_2)$ is immediate. ■

To implement indicator answers on top of the shared evaluation optimization, only a few modifications to Algorithm 2 are needed: At Lines 4, 10, and 15, we need to initialize or maintain indicator answers $q.IA$ for each q in the same manner as $q.RS$. Meanwhile, Line 20 should be substituted by a verification of the condition in Proposition 3.

C. Partial Evaluation

Even if the indicator answer optimization is implemented, the major overhead of Algorithm 2 still lies in Lines 10 and 15, where the indicator answers of each candidate query is maintained. To obtain $q.IA$, the algorithm has to perform a series of PSI checks between q and its sub-queries' indicator answers. For a relatively large graph and less popular example nodes, the size of $IA(q)$ for most queries in $D^{-1}(I)$ may be large, causing the number of PSI checks to be high. Recall that in Proposition 3, a threshold $k - |I|$ on the size of $IA(q)$ is given. If $|IA(q)|$ exceeds the threshold, q is confirmed as a false positive. This implies the following observation:

Observation 3: To reject a false positive query q, we need not know the exact size of $IA(q)$. A lower bound of it is enough to reject q, as long as the bound is larger than $k - |I|$.

For example, in the motivating example of Table I, if we know that V. Putin appears in the indicator answers of query 2b (*i.e.,* it is ranked above M. Botvinnik in the answers of query 2b), we immediately know that the rank position of M. Botvinnik must be two at best. So we reject query 2b without spending computations on the ranking position of G. Kasparov and E. Lasker. Motivated by this observation, we propose the following strategy: when checking the indicator answers of q's sub-queries to obtain $q.IA$, we only evaluate a subset of $q.IA$ which is sufficient to reject q. We call this strategy *partial evaluation*. Note that all following discussions in this section will be based on Algorithm 2 with indicator answer optimization.

At Line 4, we initialize all level-1 queries q with the exact indicator answers $IA(q)$, for this involves no PSI checks. However, for all queries q at level-2 and above, instead of $IA(q)$ itself, we maintain two disjoint node sets $\underline{IA}(q)$ and $\widetilde{IA}(q)$. $\underline{IA}(q)$ consists of the nodes that are confirmed to match q, while $\widetilde{IA}(q)$ contains the nodes whose membership to $\underline{D}(q)$ is unknown. It is obvious that the union of $\underline{IA}(q)$ and $\widetilde{IA}(q)$ covers $IA(q)$, which ensures that all nodes that contribute to $|IA(q)|$ are not missed and the lower bound of $|IA(q)|$ is not under-estimated:

Proposition 5: Let $\overline{IA}(q) = \underline{IA}(q) \cup \widetilde{IA}(q)$, and we have $IA(q) \subseteq \overline{IA}(q)$.

We maintain $\underline{IA}(q)$ and $\widetilde{IA}(q)$ as follows. In Algorithm 2, the calls of function $PSIChecks(q, V)$ and updates of $q.IA$ are replaced by function $PartialEval(q, V)$ in Algorithm 3. Specifically, after joining queries q_i and q_j, we call $PartialEval(q, V)$ where V is specified using $q_i.\overline{IA} \cap q_j.\overline{IA}$. Similarly, after extending query q_i, $PartialEval(q, q_i.\overline{IA})$ is called instead. When performing partial evaluation between query q and node set V, we scan nodes v in V and check whether q matches v. The matching Nodes v are added to $q.\underline{IA}$, otherwise v is discarded. Note that as soon as the size of $q.\underline{IA}$ exceeds the threshold $k - |I|$, we confirm that q is a false positive and end the check. Before the function returns, nodes in V that are not scanned are added to $q.\widetilde{IA}$ because it remains unknown whether q matches them, and they should be taken into consideration when checking the super-queries of q. Finally, the check of whether $|q.IA|$ exceeds $(k - |I|)$ at Line 20 should be performed on $q.\underline{IA}$ instead.

We note that, being a heuristic-based greedy optimization, partial evaluation may not always reduce the number of PSI checks globally. However, in our experiments, partial evaluation improves the efficiency for almost all questions. More details about the experiments are in Section VI.

V. Redundant Query Elimination

Though modeled as directed labeled graphs, knowledge bases are more complicated than graphs. One thing that differentiates a knowledge base is that there are various kinds of dependencies among the occurrences of its nodes, edges, and labels. For example, an edge labeled `birthPlace` can only link a `Person` node and a `Location` node. Therefore, when formulating the query "persons whose birth place is Russia", the node label `Person` can be safely dropped, being implied by the edge label `birthPlace`. As a result, dependencies in a knowledge base may cause queries with different shapes to be equivalent to each other.

We emphasize that such equivalent queries should not appear simultaneously in the results of a reverse neighborhood pattern query search. The reasons are two-fold. First, equivalent queries make the final results redundant to users. Second, the equivalent queries incur unnecessary computational costs (*e.g.,* query generation and indicator answer maintenance). To this end, we denote longer queries in which some elements are implied by the others as redundant queries, and use known dependencies in the knowledge base to avoid searching for them. In the following, we propose three types of utilizable dependencies, and show example redundant queries in Table II.

Edge-node Label Dependency. In a knowledge base, some relations only exist between entities of certain types. For example, in `DBpedia`, the relations `birthPlace` and `gender`

TABLE II: Dependencies and example redundant queries

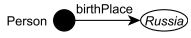
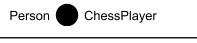
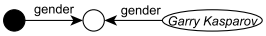
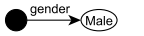
Dependency Name	Example Query	Redundant Form	Simpler Form
Edge-node label	Persons born in Russia		
Intra-node label	All chess players		
Cross-node ID	Persons having the same gender with G. Kasparov		

TABLE III: Statistics of DBpedia dataset

	Size
$\|V\|$	3,995,882
$\|L_V\|$	10,784,441
$\|E\|$	12,724,403
$\|\Sigma_V\|$	388
$\|\Sigma_E\|$	678

must start from Person entity. An example is shown in row 1 of Table II.

Intra-node Label Dependency. In a knowledge base, there may be subsumption between certain entity types. For example, in DBpedia, all ChessPlayer entities also have the type of Person. Therefore, in row 2 of Table II, when the algorithm generates a query where a node contains label ChessPlayer, the node should not have any super type of ChessPlayer as additional node labels, including Person. Moreover, constant nodes (*i.e.,* nodes with node IDs as labels) should not carry any other type labels, because they are actually implied by the IDs.

Cross-node ID Dependency. In a query, when a constant node appears, it is bounded to a certain node in the knowledge base. This may cause some adjacent node to have a definite bind. For example, in row 3 of Table II, given that G. Kasparov has only gender Male, the middle node is uniquely bound to Male when the query is evaluated against the knowledge base. Therefore, the query may be simplified by explicitly adding a Male label to the middle node, and deleting the other unnecessary nodes and edges.

The implementation of redundant query elimination is simple but effective. First, the redundancy check does not involve query execution. In our current implementation on the DBpedia knowledge base, we adopt the corresponding OWL (Web Ontology Language) metadata[5] which explicitly describes the first two dependencies in the data. Specifically, the edge-node type dependency is given by the ⟨rdfs:domain⟩ and ⟨rdfs:range⟩ fields of each relation record, while the hierarchical node type dependency is given by the ⟨rdfs:subClassOf⟩ field of each type record. For constant node dependency, we build an index on the graph, which returns all ending nodes a given node links to via a given type of edge. Second, we do not need to generate every redundant query before eliminating it. Because any super-query of a redundant query is also redundant, we directly block a redundant query from entering higher levels of $D^{-1}(I)$ once it is found, to avoid generating its super-queries. In experiments, the three types of dependencies help pruning redundant queries in $D^{-1}(I)$ by almost halving its size. We will demonstrate the details in Section VI-B.

VI. Experiments

In this section, we extensively evaluate our solution using real-life datasets. First, we introduce the knowledge base, questions, and experimental terminology used in the experiments. Second, we verify the necessity of eliminating redundant queries by showing their proportions in the search space. Based on the reduced search space, we investigate the effectiveness of our algorithm in terms of the number of candidate queries it returns. Third, we study the optimizations proposed in Section IV and compare the efficiency of our algorithm with other methods. Finally, we discuss the impact of parameter k on the effectiveness and efficiency of our algorithm.

A. Experimental Settings

Knowledge Base. We used DBpedia 3.9 as our knowledge base. It consists of several sub-datasets. Specifically, our entity type information was from the "Mapping-based Types" sub-dataset, while relation information was from the "Mapping-based Properties" sub-dataset. Note that DBpedia also provides an "Infobox Properties" sub-dataset containing relation information. However, we did not involve this sub-dataset because it is noisy and inconsistent with the ontology according to manuals from DBpedia.

To perform redundant query elimination, we used the OWL metadata mentioned in Section V. We ignored all relationships which are marked as "DatatypeProperty" by the metadata, because they point to numerical/datetime/coordinate type of data instead of entities, which falls out of the scope of this paper. The statistics of the processed knowledge base are shown in Table III.

Ideally, the relative popularity of entities in a knowledge base shall be estimated from the number of times the entities are accessed/viewed by users. Without the luxury of accessing such data, we simulate the relative popularity $\prec$ of entities by using PageRank algorithm on the corresponding labeled graph. The damping factor d is set to 0.85 as default in [18].

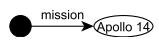
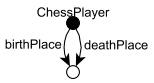
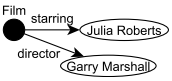
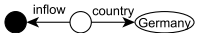
Questions. For real-life questions, we used the QALD-4-Task-1[6] dataset (denoted by QALD-4 for short). This dataset was first introduced by a contest on natural language question answering over linked data. It includes 250 questions in natural language, each with a SPARQL query and an answer set.

Note that some questions cannot be expressed by neighborhood pattern queries, thus are beyond the scope of this paper. Therefore, we discarded questions meeting any of the following criteria. 1) It is marked as "OUT OF SCOPE" by the dataset, which means it cannot be answered by DBpedia. 2) It contains operators not expressible in neighborhood pattern queries, such as UNION, COUNT, FILTER, and ORDER BY. 3) Its answers are not entities; *e.g.,* a question may require dates, decimals, or booleans as answers. 4) It involves datasets

[5]http://downloads.dbpedia.org/3.9/dbpedia_3.9.owl.bz2

[6]http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/index.php?x=task1&q=4

TABLE IV: Question groups with examples. Groups are named after the size and radius of corresponding queries.

Group	Question Number	Example Question	SPARQL Query	Top-2 Answers	Neighborhood Pattern Query
$G(2,1)$	24	Give me all Apollo 14 astronauts.	SELECT DISTINCT ?uri WHERE {?uri :mission :Apollo_14 .}	Alan Shepard Edgar Mitchell	
$G(3,1)$	18	Which chess players died in the same place they were born in?	SELECT DISTINCT ?uri WHERE {?uri :type :ChessPlayer . ?uri :birthPlace ?x . ?uri :deathPlace ?x .}	Mikhail Botvinnik Vladimir Alatortsev	
$G(4,1)$	4	Give me all people that were born in Vienna and died in Berlin.	SELECT DISTINCT ?uri WHERE {?uri :birthPlace :Vienna . ?uri :deathPlace :Berlin .}	Richard Thurnwald Alfred Halm	
$G(5,1)$	3	In which films directed by Garry Marshall was Julia Roberts starring?	SELECT DISTINCT ?uri WHERE {?uri :type :Film . ?uri :starring :Julia_Roberts . ?uri :director :Garry_Marshall.}	Valentine's Day Runaway Bride	
$G(*,2)$	3	Which rivers flow into a German lake?	SELECT DISTINCT ?uri WHERE {?x :inflow ?uri . ?x :country :Germany .}	Rhine Havel	

not used in our experiments, such as the "Infobox Properties" sub-dataset, YAGO2, and FOAF. Moreover, we discarded questions whose answers consist of only one entity, because users do not need more answers for such questions.

After pre-processing, we got 52 questions, whose sizes range from 2 to 5, and radiuses from 1 to 2. We grouped all questions by the shapes of their SPARQL queries. Questions with radius-1 queries were grouped by their sizes, and were denoted by $G(2,1)$, $G(3,1)$, $G(4,1)$, and $G(5,1)$, respectively. All questions with radius-2 queries were in one group denoted by $G(*,2)$, because there were only three of them. In Table IV, an example is provided for each group. Besides persons, films, and locations involved in the table, the 52 questions also cover many other types like countries, languages, books, etc.

Experimental Protocols. We treated all questions uniformly. For each question, we regarded the corresponding SPARQL query provided by QALD-4 as ground truth. We constructed the input entity set I using one or two entities with the largest PageRank scores in the answer set. When varying $|I|$, we required that $k = |I|$. In other words, the returned queries should promote all entities in I above other entities. When studying the effects of parameter k, we fixed $|I| = 1$. Because entities with large PageRank scores tend to lie in the dense areas of the labeled graph, it is impossible to enumerate all queries that match these entities. Therefore, we employed techniques in [9], [10], and used the size and radius of the ground truth query to bound the search space. Removing this limitation is part of our future work.

Compared Methods. We compared four variants of RkNPQ distinguished by their candidate query refinement approaches: 1) RkNPQ-gStore: evaluate full answers using gStore[7]; 2) RkNPQ-S: **S**hared evaluation of full answers; 3) RkNPQ-SI: **S**hared evaluation of **I**ndicator answers; 4) RkNPQ-SPI: **S**hared **P**artial evaluation of **I**ndicator answers. All four methods have identical outputs and only differ in efficiency.

The experiments were conducted on two identical PCs, each with two Xeon 2.50GHz processors and 64GB memory. The RkNPQ framework and the last three variants were implemented in C# and run on one PC running Windows system. All query evaluations involving gStore were done on another PC running Linux system. All experiments ran on a single core, and we report efficiency results in seconds.

[7]We also experimented with the neo4j graph engine and JENA-TDB RDF engine. We only report gStore here because in our experiments it has the best performance among all three query engines.

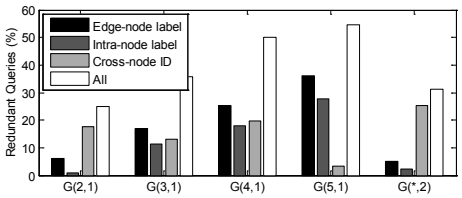
B. Effectiveness Evaluation

In this section, we demonstrate two empirical observations related to effectiveness. First, redundant query elimination can halve the preliminary search space at most. Second, our algorithm returns a reasonable number of possible queries for most questions, even if limited examples are provided. Note that, for every question, we ensure that its ground truth query (those in the last column of Table IV) always appears in the output. This leads to perfect *recall* of the output, and *precision* to be the inverse of the output size. Therefore, we only need to care about the output size and do not consider qualitative metrics like *precision*, *recall*, and *F-measure*.

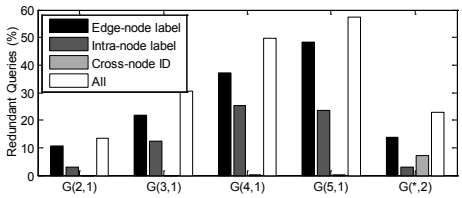
Redundant Query Elimination. In Section V, we proposed three types of redundant queries. We also discussed how to eliminate them to avoid duplicate results and reduce running time. The amount of time to be saved depends on the proportions of redundant queries in the search space $D^{-1}(I)$.

In Figure 4a, we show the average proportion of redundant queries within each group, when only one example entity was given as input. The respective and overall percentages of different types of redundant queries are both reported. Note that the three redundancy types may overlap, so the overall percentage is smaller than the sum of all three percentages. From the figure, it is clear that by eliminating redundant queries, the search space is reduced by 25%-50% in size.

In Figure 4b, when two example entities were provided, the percentages of redundant queries follow a similar trend as in Figure 4a. The only difference is that the percentage of the third redundancy type drops significantly. Recall that this type of queries involve constant nodes. When more than one example entities were provided, it is harder to find constant nodes that connects to all input entities in the same manner. Therefore, such queries are less likely to appear. From now on, we enable redundant query elimination by default, and report all results based on this setting.

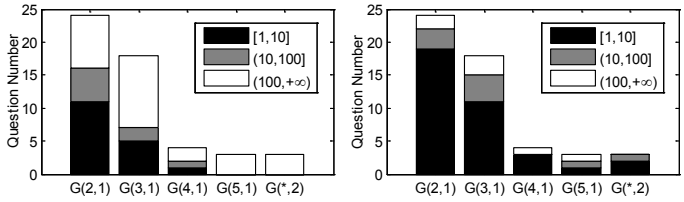


(a) $|I| = k = 1$



(b) $|I| = k = 2$

Fig. 4: Redundant queries: respective & overall percentages



(a) $|I| = k = 1$ (b) $|I| = k = 2$

Fig. 5: Distribution of questions w.r.t. size of their query sets

Number of Returned Queries. After redundant queries are removed from $D^{-1}(I)$, our algorithm refines queries in $D^{-1}(I)$ to return the final query set $D_k^{-1}(I)$. The user then looks through $D_k^{-1}(I)$ to find the query she wants. By checking the returned queries of every question, we confirmed that all ground truth queries were included in the corresponding query sets. Therefore, the size of the returned query set is the only matter to the user. The smaller the query set is, the easier it is for the user to look through them. We treated the difficulty to view query sets whose sizes were in [1,10], (10,100], and (100,+∞) as easy, moderate, and hard, respectively.

In Figure 5, we report the number of corresponding questions w.r.t. their query sets' difficulty. The two figures show that, for questions with simple query shapes, it is relatively easy for users to browse the returned queries. For example, the group $G(2,1)$ contains 24 questions. When the user provided one example, our algorithm returned no more than 10 queries for 11 questions (45.8%). When the input consists of two examples, the number of easy questions grows to 19 (79.2%).

However, for questions whose query shapes are complicated, our algorithm returned too many queries with only one input entity. For example in Figure 5a, for all questions whose queries are of size 5 or radius 2 (*i.e.,* groups $G(5,1)$ and $G(*,2)$), an intolerable query set of over 100 queries was returned. We note that complicated ground truth queries and too few input entities are both the causes of large query sets. By comparing Figure 5a and 5b, we find that, in all five groups, if two examples were given instead of one, the number of easy questions increased significantly, and much fewer hard questions remained.

We exemplify the above observations with an example question "in which films directed by Garry Marshall was Julia Roberts starring" of Group $G(5,1)$ in Table IV. When only "Valentine's Day" is provided as input, the user may be asking for the above question. But she may also be asking for films directed by Garry Marshall and starred by Anne Hathaway. In fact, our algorithm returned more than two thousand queries including the above two. However, when the user gave a second example "Runaway Bride", the question about Anne Hathaway was ruled out because she did not starred in this film, and the algorithm returned only 6 queries.

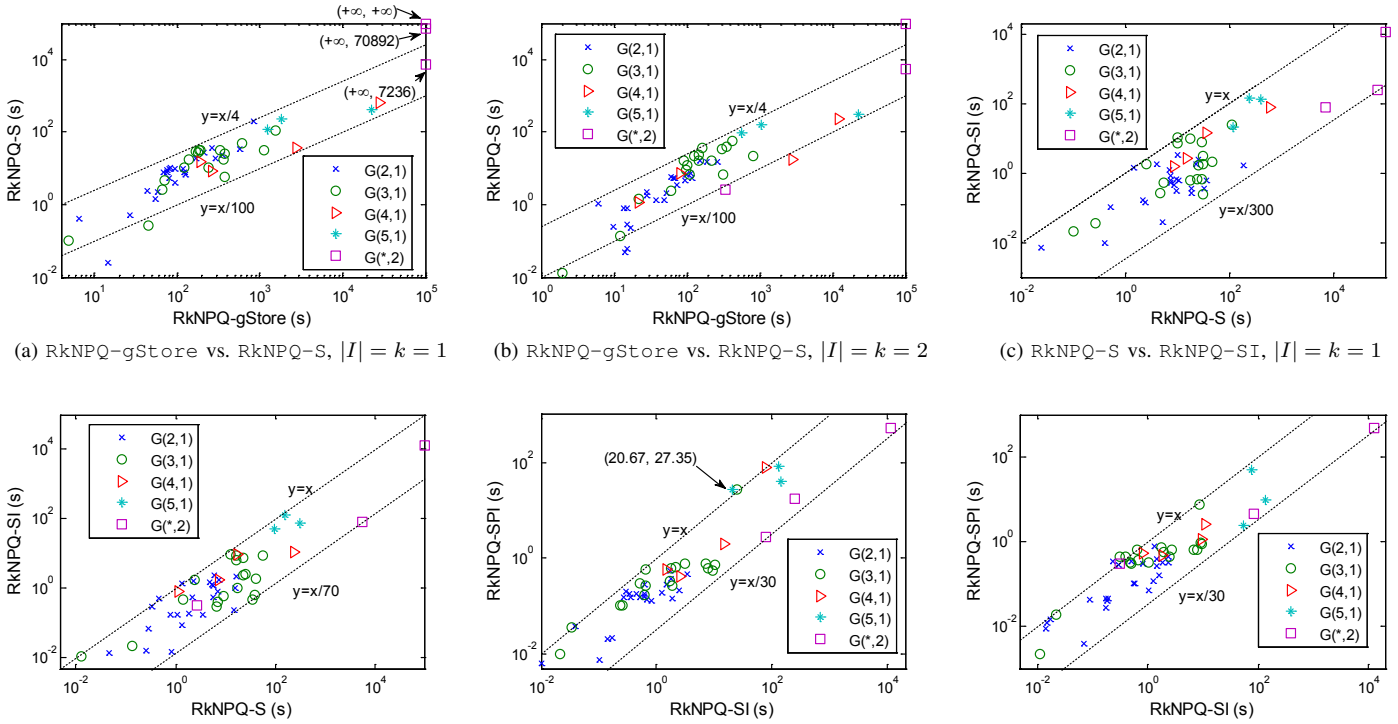
C. Efficiency Evaluation

In this section, we validate the observations and optimizations proposed in Section IV. We compare four variants of RkNPQ, namely RkNPQ-gStore, RkNPQ-S, RkNPQ-SI, and RkNPQ-SPI. Each time we competed the running time of two adjacent methods in the list to support one observation/optimization. As Figure 6 shows, we compared two methods by plotting each question as a data point, where the running time of the two methods were treated as x and y coordinates. Note that the refine phase of RkNPQ-gStore was much slower than the filter phase, causing the filter phase to be negligible. Therefore only refine time is reported for RkNPQ-gStore. For all other variants, we report end-to-end running time.

RkNPQ-gStore vs. RkNPQ-S (Shared Evaluation). Figure 6a compares RkNPQ-gStore and RkNPQ-S when only one example entity was provided. We observed that both methods spent too much time on some questions in $G(*,2)$. Therefore, we terminated the search when it took more than 10^5 seconds (approximately three hours), and reported the running time as infinity (+∞). In Figure 6a, three observations are made. First, almost all points are located below the dotted line $y = x/4$. This indicates that when computations were shared among evaluations of similar queries, our algorithm could be accelerated by at least four times. Second, there are two outliers on the upper-right corner. They were actually caused by timeout and terminated runs, and are not against the previous observation. Third, most data points lie above the dotted line $y = x/100$, which indicates that shared evaluation could achieve an efficiency gain of approximately two orders of magnitude at best. When two example entities were provided (see Figure 6b), the above observations hold.

RkNPQ-S vs. RkNPQ-SI (Indicator Answers). In Section IV-B, we discussed out that only indicator answers are responsible for the ranking positions of example entities. We also emphasized that indicator answer evaluation could benefit from shared evaluation as well. Here we only compare RkNPQ-SI with RkNPQ-S to demonstrate that indicator answers can reduce computations when shared evaluation is applied.[8] Figure 6c shows that, when the input consisted of only one example answer, the algorithm could be accelerated by up to 300 times if only indicator answers were considered. When two examples were given (see Figure 6d), RkNPQ-SI was 70 times faster than RkNPQ-S at best, which was less significant compared with Figure 6c. This is because when more examples are given, the indicator answer set $IA(q)$ tends

[8]This is because it takes time to modify gStore to implement an RkNPQ-I method which only employs the indicator answer optimization.



(a) RkNPQ-gStore vs. RkNPQ-S, $|I| = k = 1$ (b) RkNPQ-gStore vs. RkNPQ-S, $|I| = k = 2$ (c) RkNPQ-S vs. RkNPQ-SI, $|I| = k = 1$

(d) RkNPQ-S vs. RkNPQ-SI, $|I| = k = 2$ (e) RkNPQ-SI vs. RkNPQ-SPI, $|I| = k = 1$ (f) RkNPQ-SI vs. RkNPQ-SPI, $|I| = k = 2$

Fig. 6: Pairwise running time comparisons among RkNPQ-gStore, RkNPQ-S, RkNPQ-SI, and RkNPQ-SPI.

to converge to $D(q)$ for all surviving candidate queries q, which makes RkNPQ-SI less beneficial.

RkNPQ-SI vs. RkNPQ-SPI (Partial Evaluation). In Figure 6e, we demonstrate the benefit of partially evaluating the indicator answer sets. For most questions, this heuristic reduces about 30 times of extra computations. By checking each question, we found that the number of PSI checks was successfully reduced except for the example question of $G(5, 1)$. This caused RkNPQ-SPI to be seven seconds slower than RkNPQ-SI, as is marked by the outlier in Figure 6e. The same situation also happens for some data points in Figure 6f. We note that being heuristic-based, partial evaluation does not strictly ensure a reduction of PSI checks. We will consider optimizations with theoretical guarantees in future work.

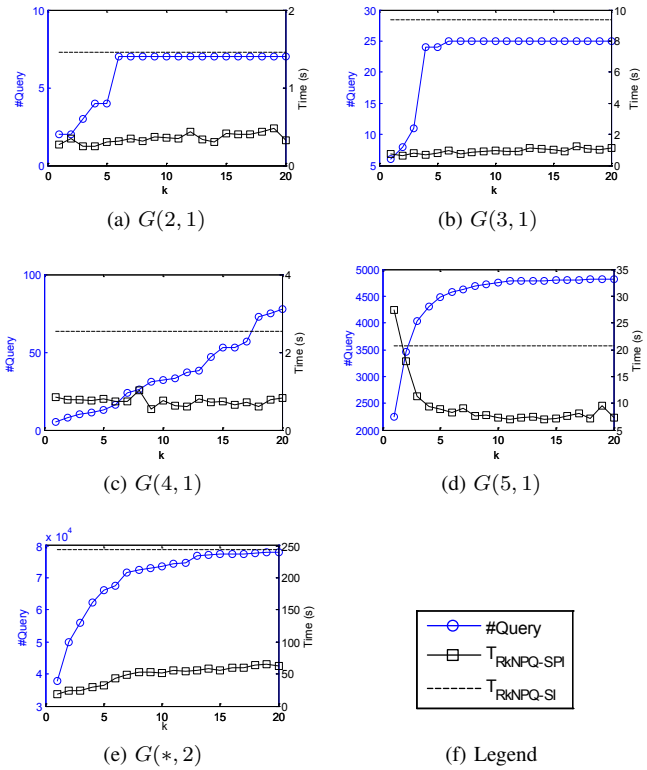
D. Analysis of Parameter k

In this section, we analyze the impact of parameter k. Recall that our algorithm only returns queries whose top-k results cover the example entities I. Therefore by simple reasoning, the impact of increasing k are two-sided. Because different users may have different judgements on the relative popularity of entities, a larger k makes the algorithm more tolerable on the ranking positions of I. This avoids the situation where the user cannot find a satisfactory query. However, a larger k also causes more queries to be returned.

Currently we do not have real input data of I, so we limit our empirical analysis to the number of returned queries and running time, when I consists of the top entities in the answer set. Because similar observations are made for both $|I| = 1$ and $|I| = 2$, we only report results for $|I| = 1$ due to space limitation. In Figure 7, we report the number of returned queries and running time of RkNPQ-SPI of the five questions in Table IV when k varied between 1 and 20. We also use dotted lines to denote the running time of RkNPQ-SI. Note that the running time of RkNPQ-SI does not depend on k.

Impact on Effectiveness. The above results show that our algorithm returned more queries with increasing k. Recall that in Section I, we mentioned differences between the partial answers setting [21] and our setting. In fact, the setting where entities in I are only treated as partial answers is equivalent to our setting when $k \to \infty$. In Figure 7, we follow the trend to see what may happen when k approaches infinity. In Figure 7a and 7b, the questions enjoyed a small search space. The query numbers quickly converge to 7 and 25, which indicates an easily or moderately readable set, respectively, according to the criteria in Section VI-B. However, in Figure 7c, the query number only demonstrates a linear growth when k is below 20, which implies that the query set is at most moderately readable when it converges. In this situation, an appropriately set k may help balance the query set size and the possibility that the ground truth query is returned. Unfortunately, in Figures 7d and 7e, the query sets are both large and grow quickly when k is small. The only way to reduce them may be to ask the user to give more example answers.

Impact on Efficiency. Figure 7 demonstrates three observations. First, the running time of RkNPQ-SPI does not necessarily grow monotonically when k increases. Second, for most cases the line for RkNPQ-SPI is below the dotted line, indicating that the partial evaluation optimization is feasible for a broad range of k. Third, in Figure 7d when $k = 1$, the



(a) $G(2,1)$ (b) $G(3,1)$ (c) $G(4,1)$ (d) $G(5,1)$ (e) $G(*,2)$ (f) Legend

Fig. 7: Impacts of k on the number of returned queries and running time for the example questions in Table IV.

line for `RkNPQ-SPI` is above the dotted line. We note that this case corresponds to the discussed outlier in Figure 6e.

VII. Related Work

Our research is related to reverse engineering structured queries, query by example entities/tuples, question-answering on knowledge bases, and query optimization techniques.

Reverse Engineering Structured Queries. The problem of reverse engineering queries using answers has been extensively studied in the relational database and `SQL` setting. In [24], Tran et al. proposed the problem of query by output. Given the output table of an unknown `SQL` query q on a database D, they explored techniques to find all queries q' having exactly the same result with q. To reverse engineer the `WHERE` clauses, *i.e.,* conjunctions and disjunctions of constraints on the attributes, they built decision trees with the relational data, and interpreted the trees into `WHERE` clauses. Zhang et al. [30] pointed out that [24] posed an impractical restriction on the structure of q', *i.e.,* no multiple instances of a single relation are allowed in the `FROM` clause. They dropped the restriction and proposed a sound and complete searching approach to discover all complex join queries. Shen et al. [21] addressed the problem in a slightly different setting. They allow the input table to have empty cells. Moreover, the results of the discovered queries need not match the input exactly, but only inclusion is required.

In [5], Bonifati et al. studied an interesting variation of the reverse engineering query problem. Instead of providing the answer herself, the user only needs to judge each tuple presented by the system on whether it appears in the answers to her query. Therefore, their major concerns were on identifying informative tuples to minimize the number of interactions. Authors of [5] also investigated inferring twig queries on XML data [22] and path queries on graph databases [4] with theoretical guarantees. Their queries support expressive features such as descendant edges and regular expressions. However, they are limited to tree and path shapes. We note that neither of them subsumes or is a fragment of neighborhood pattern queries studied in this paper.

Reverse Query Problems for Vector Data. In [26], Vlachou et al. studied the problem of reverse top-k queries with an application on product recommendation. They modeled products as property vectors, and buyers as preference vectors on the same dimensions. The potential interest of a buyer for a product was measured by the inner product of the two vectors. Given a product, the algorithm returns every buyer for whom this product is among the top-k ones he is interested in. The manufacturer may then find promotional targets based on the results. Other reverse query problems like reverse nearest neighbor queries [12] and reverse skyline queries [7] have also been studied in previous work. Due to the inherent difference between vector-based and graph-based data, our algorithmic details are significantly different.

Query by Example Entities & Tuples. The input of our problem are example entities, which is literally similar to the well-known Query by Example [32] (QBE) framework of specifying queries in relational databases. However, the "examples" of QBE are actually joins and constraints expressed via manipulating tables, rather than real tuples from the database.

Recently, various studies [11], [14], [16], [17] were conducted on querying with real examples like entities and tuples. They all required a mixed list of similar entities (tuples) as output, instead of structural queries as in this paper. Specifically, Lim et al. [14] represented entities as feature vectors, where the features are distances between the entities and concepts in a concept hierarchy. Metzger et al. [16] measured the similarity between entities by their overlapping *aspects*. Mottin et al. [17] required returned tuples to have an isomorphic embedding to, and a good connectivity with, the input tuple. Jayaram et al. [11] generated a *maximal query graph* by exploring the neighborhood of the input tuple. Subgraphs of the maximal query graph were treated as queries, and tuples were ranked by heuristical scores on the queries they match and the quality of the match. We note that this line of work and ours are based on different assumptions of user intentions. If a user only wants results broadly similar to her examples, our technique may not help because she may not have a clear and describable question. On the contrary, if her information need is precise, it may be inappropriate to return her a list like the first column of Table I, which is a mixture of answers from three queries.

Natural Language QA over Knowledge Bases. As more large knowledge bases [1], [3], [6], [23], [27] are becoming available, researchers are trying to re-address the traditional question answering problem using knowledge bases instead of unstructured text data. To translate a natural language question into an executable structured query, Unger et al. [25] proposed a template-based approach. Yahya et al. [28] formulated the translating task as an integer liner program problem. Berant et al. [2] studied a learning approach using question-answer

pairs. Zou et al. [33] suggested unifying the query translating and evaluating stages for some ambiguity in the question may be resolved better at the second stage. Currently, we do not need the textual question to be provided as extra input, though it may greatly reduce the number of possible queries. We plan to address this new setting in our future research.

Multi-query-based and View-based Query Optimization. The problem of multi-query optimization ([19], [20], [31], to name a few) has been studied for decades. Specifically, Le et al. [13] attacked the problem in the `RDF` and `SPARQL` setting. The proposed solutions first identified common substructures residing in the input queries, and then used them to share computations. View-based query optimization (see [8] for a comprehensive survey) aims at answering queries using materialized views. For a new query, the proposed solutions generate execution plans which only access the pre-computed answers to some given views. In our problem, after building the query lattice, the algorithm is actually aware of the containment relationship between candidate queries. This information makes common structure detection unnecessary. It also allows us to directly reduce duplicate computations for a query by intersecting and filtering results of its sub-queries instead of generating more complicated execution plans.

VIII. Conclusion and Future Work

In this paper, we formulated the reverse top-k neighborhood pattern query problem to help discover knowledge base queries using popular partial answers. A solution framework based on filter-refine scheme was introduced. We reduced the filter phase to the frequent neighborhood pattern mining problem, and proposed three optimizations on the refine phase, namely shared evaluation, indicator answers, and partial evaluation. Experiments on real-life datasets demonstrated the effectiveness of our problem setting, and the efficiency of our approach. We would like to address the following future works. First, we will investigate utilizing the question text to cut the search space. Second, we are interested in discovering queries with two pivots, which is related to meta-path queries studied in [15]. Finally, numerical nodes and range selections may be introduced to neighborhood pattern queries, and tailored searching techniques may be challenging to explore.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. This work was partially supported by Australian Research Council Grant DE140100215, the National Key Basic Research Program (973 Program) of China under grant No. 2014CB340403, Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-066, and the National Natural Science Foundation of China (NSFC. 61402532).

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "Dbpedia: A nucleus for a web of open data," in *ISWC/ASWC*, 2007.

[2] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on freebase from question-answer pairs." in *EMNLP*, 2013.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *SIGMOD*, 2008.

[4] A. Bonifati, R. Ciucanu, and A. Lemay, "Learning path queries on graph databases," in *EDBT*, 2015.

[5] A. Bonifati, R. Ciucanu, and S. Staworko, "Interactive inference of join queries," in *EDBT*, 2014.

[6] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *AAAI*, 2010.

[7] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *PVLDB*, 2007.

[8] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[9] J. Han and J.-R. Wen, "Mining frequent neighborhood patterns in a large labeled graph," in *CIKM*, 2013.

[10] J. Han, J.-R. Wen, and J. Pei, "Within-network classification using radius-constrained neighborhood patterns," in *CIKM*, 2014.

[11] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying knowledge graphs by example entity tuples," *TKDE*, 2015.

[12] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *SIGMOD*, 2000.

[13] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for sparql," in *ICDE*, 2012.

[14] L. Lim, H. Wang, and M. Wang, "Semantic queries by example," in *EDBT*, 2013.

[15] C. Meng, R. Cheng, S. Maniu, P. Senellart, and W. Zhang, "Discovering meta-paths in large heterogeneous information networks," in *WWW*, 2015.

[16] S. Metzger, R. Schenkel, and M. Sydow, "Qbees: query by entity examples," in *CIKM*, 2013.

[17] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: Give me an example of what you need," *PVLDB*, 2014.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." 1999.

[19] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," *SIGMOD Record*, 2000.

[20] T. K. Sellis, "Multiple-query optimization," *TODS*, 1988.

[21] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, "Discovering queries based on example tuples," in *SIGMOD*, 2014.

[22] S. Staworko and P. Wieczorek, "Learning twig and path queries," in *ICDT*, 2012.

[23] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW*, 2007.

[24] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, "Query by output," in *SIGMOD*, 2009.

[25] C. Unger, L. Bühmann, J. Lehmann, A.-C. Ngonga Ngomo, D. Gerber, and P. Cimiano, "Template-based question answering over rdf data," in *WWW*, 2012.

[26] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag, "Reverse top-k queries," in *ICDE*, 2010.

[27] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: A probabilistic taxonomy for text understanding," in *SIGMOD*, 2012.

[28] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum, "Natural language questions for the web of data," in *EMNLP*, 2012.

[29] A. W. Yu, N. Mamoulis, and H. Su, "Reverse top-k search using random walk with restart," *PVLDB*, 2014.

[30] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava, "Reverse engineering complex join queries," in *SIGMOD*, 2013.

[31] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla, "Simultaneous optimization and evaluation of multiple dimensional queries," in *SIGMOD Record*, 1998.

[32] M. M. Zloof, "Query by example," in *AFIPS NCC*, 1975.

[33] L. Zou, R. Huang, H. Wang, J. X. Yu, W. He, and D. Zhao, "Natural language question answering over rdf: a graph data driven approach," in *SIGMOD*, 2014.

[34] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: Answering SPARQL queries via subgraph matching," *PVLDB*, 2011.