Efficient Query Answering in Probabilistic RDF Graphs

Xiang Lian and Lei Chen
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
{xlian, leichen}@cse.ust.hk

ABSTRACT

In this paper, we tackle the problem of efficiently answering queries on probabilistic RDF data graphs. Specifically, we model RDF data by probabilistic graphs, and an RDF query is equivalent to a search over subgraphs of probabilistic graphs that have high probabilities to match with a given query graph. To efficiently process queries on probabilistic RDF graphs, we propose effective pruning mechanisms, structural and probabilistic pruning. For the structural pruning, we carefully design synopses for vertex/edge labels by considering their distributions and other structural information, in order to improve the pruning power. For the probabilistic pruning, we derive a cost model to guide the pre-computation of probability upper bounds such that the query cost is expected to be low. We construct an index structure that integrates synopses/statistics for structural and probabilistic pruning, and propose an efficient approach to answer queries on probabilistic RDF graph data. The efficiency of our solutions has been verified through extensive experiments.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.3 [**Information Systems**]: Information Storage and retrieval—*Information search and retrieval, Search process*

General Terms

Algorithms, Design, Experimentation, Performance, Theory

Keywords

probabilistic RDF graph, probabilistic data, RDF

1. INTRODUCTION

The RDF (Resource Description Framework) data model is a W3C standard to describe resources on the Web and capture their relationships. RDF has been proposed for a decade, and nowadays it has been widely used in many real applications such as the Semantic Web [1]. In RDF, data items are represented in the form of triples, (*subject*, *predicate*, *object*), also known as (*subject*, *property*, *object*) triples. As an example, information about two persons with identities (IDs), pid_1 and pid_2, could include the triples as shown in Figure 1 (note: not all triples are depicted).

SIGMOD'11, June 12–16, 2011, Athens, Greece.

RDF triples from data source A:
(1) ("Zoe", <givenNameOf>, pid_1);
(2) ("Panos", <familyNameOf>, pid_1);
(3) (pid_1, <type>, "scientist");
(4) (pid_1, <bornInLocation>,"Athens");
(5) (pid_1, <studyIn>,"Athens");
(6) ("Athens", <bornStudyRel.>, "Athens");
(7) ("Athens", <locationIn>, "Greece");
(8) (pid_1, <hasDoctoralAdvisor> pid_2);
...
(9) (pid_2, <bornInLocation>, "Athens");
(10) ("Athens", <locationIn>, "Greece");
...

Figure 1: RDF triple data.

From the triples, the first person with ID pid_1 is a scientist (observed from triple (3)) called "Zoe Panos" (from triples (1) and (2)), born and studying in Athens, Greece (triples (5)-(7)), and her PhD advisor is the person with ID pid_2 (triple (8)). Similarly, the second person with ID pid_2 was also born in Athens, Greece (obtained from triples (9) and (10)).

RDF triples can be equivalently viewed from a graph perspective [3]. That is, for each RDF triple (S, P, O), *subject*, S, and *object*, O, can be considered as labels of two adjacent vertices in a graph, and *predicate*, P, is treated as the label of a directed edge from S to O. Figure 2(a) presents the equivalent graph representation of RDF triples above. In the previous example, the top-left vertex in Figure 2(a) has label "Zoe", connecting with a vertex pid_1 through the directed edge labeled "givenNameOf", and it exactly corresponds to the subject of triple (1), ("Zoe", <givenNameOf>, pid_1).

In practice, RDF data can be highly unreliable [13, 15] for reasons such as data errors or expirations. For example, in the application of information extraction (IE) [28] from unstructured Web data, IE methods will naturally infer top-k possible (uncertain) data, associated with extraction accuracies, in the probabilistic RDF form.

Similarly, during the data integration [11, 12], we need to incorporate RDF data from various data sources into a database. Here, the uncertainty/inconsistency in the integrated data can result from either the extraction accuracy or the reliability of sources [11, 12]. For example, assume that another data source, B, has collected the same set of information about the two persons, pid_1 and pid_2, however, with a few different triples from data source, A, listed below.

Different RDF triples in data source B from that in data source A:
(4') (pid_1, <bornInLocation>, "Aarau");
(5') (pid_1, <studyIn>,"Aarau");
(6') ("Aarau", <bornStudyRel.>, "Aarau");
(9') (pid_2, <bornInLocation>, "Berlin");
(10') ("Berlin", <locatedIn>, "Germany");
(11') ("Aarau", <locationIn.>, "Switzerland");

Taking triples (4) and (4') as an example, the two data sources A and B report inconsistent places where person pid_1 was born, that is, "Athens" and "Aarau", respectively.

In order to model/resolve such inconsistencies [6, 10], we can assign each possible birthplace with a probability to indicate its

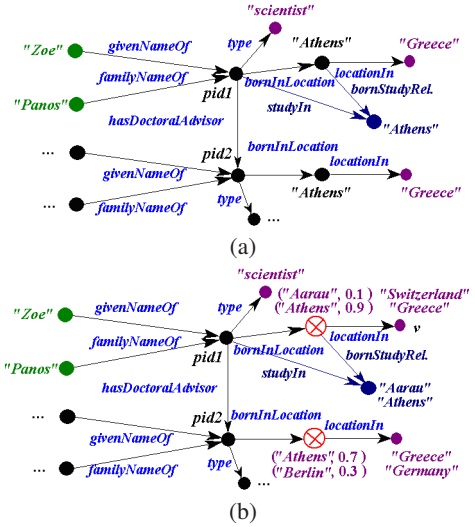


Figure 2: Graph representation of certain and probabilistic RDF data. (a) Certain RDF data graph. (b) Probabilistic RDF data graph.

confidence to be true. Accordingly, in the graph of Figure 2(b) for such integrated RDF data, the top vertex "⊗" is associated with a probability table containing labels for possible birthplaces, that is, "Athens" with probability 0.9 and "Aarau" with probability 0.1. Here, probabilities can be inferred from the reliability of data from different sources [11]. Similarly, the vertex v adjacent to top vertex "⊗" via an out-going edge labeled "locationIn" is associated with a (conditional) probability table, containing entries for labels "Switzerland" and "Greece". That is, since "Aarau" is a city of "Switzerland", rather than "Greece", vertex v takes label "Switzerland" with probability 1, given that vertex "⊗" takes label "Aarau"; moreover, v takes value "Greece" with probability 1, given that "⊗" takes "Athens". Therefore, this graph representation (via conditional probability tables in vertices) is generic enough to capture the correlations/constraints among labels of vertices [20].

The SPARQL query is a standard language for querying RDF data, and supports conjunctions (disjunctions as well) of triple patterns. However, it usually assumes that RDF data are certain and accurate. To give an example, the SPARQL query below retrieves the given name and family name of a scientist, who was born in Greece, and whose doctoral supervisor was also born in Greece.

```
select ?gn ?fn
where { ?gn <givenNameOf> ?p. ?fn <familyNameOf> ?p.
        ?p <type> "scientist". ?p <bornInLocation> ?city.
        ?city <locatedIn> "Greece".
        ?p <hasDoctoralAdvisor> ?a.
        ?a <bornInLocation> ?city2. ?city2 <locatedIn> "Greece"}
```

Note that, within the "where" clause of this SPARQL query, triple patterns contain either variables or literals, and they are concatenated by dots (indicating conjunctions). The SPARQL query is to find bindings of variables (e.g., $?gn$ and $?fn$) that satisfy the conjunctions in the "where" clause.

Each SPARQL query can be also represented by a graph [3], which contains directed edges corresponding to triple patterns. Figure 3 shows the query graph of our SPARQL example mentioned above. As a result, any SPARQL query can be considered as a subgraph pattern matching problem, which identifies subgraphs in RDF data graph matching with structures/labels of the query graph.

In the case of data integration, RDF data graph integrated from different data sources can be considered as a probabilistic graph. Thus, in this paper, our goal is to efficiently and effectively conduct a SPARQL query over probabilistic RDF data graphs.

In order to guarantee the confidence of the returned query answer, we re-define the matching probability between a subgraph

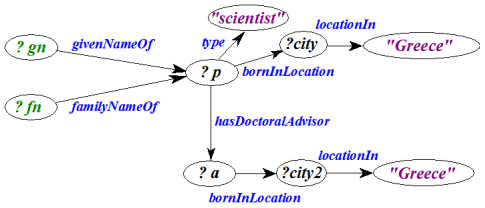


Figure 3: A SPARQL query on (probabilistic) RDF data.

and a query graph in probabilistic RDF graphs. However, the query answering in such probabilistic RDF graph is challenging, in terms of the query efficiency. First, the subgraph matching probabilities need to consider uncertainties and correlations of labels for vertices, which is costly to compute. That is, the computation of subgraph matching probabilities requires to take into account the *possible worlds* semantics [10] for probabilistic RDF graph, where each possible world is a materialized instance of the data graph. However, since the number of possible worlds can be exponential, it is not time- and space- efficient to materialize and query on all possible worlds. Thus, existing solutions to querying certain RDF data do not easily apply to probabilistic RDF data well, in terms of query efficiency. Second, RDF data graphs have their own properties, for example, some vertices in the graph have very large (mega-scale) degrees, or labels with skewed distributions. Therefore, such properties may incur low query efficiency, if we use indexes for usual probabilistic graphs that do not have these properties.

To tackle the challenging problems mentioned above, in this paper, we first formalize the problem of answering SPARQL queries over probabilistic RDF data, in the presence of label correlations. Then, we propose effective pruning techniques for RDF queries, with respect to both structural and probabilistic pruning. For the structural pruning, we design novel filtering synopses that are adaptive to label distributions, in order to improve the pruning power. The rationale behind our synopses is due to an observation, that highly frequent labels are less likely to help prune false alarms. Our synopses take into account this observation, and assign less frequent labels with higher weights. This way, we can reduce the chances of hashing confliction in synopses, and the pruning power is expected to be higher. We derive a cost model to decide the parameter settings in our hashing functions for synopses. For the probabilistic pruning, we propose another cost model to formalize the query cost, and adaptively store pre-computed probability upper bounds to facilitate the pruning such that the query cost is minimized in light of our cost model.

To summarize, we make the following contributions.

- We give a general framework for answering RDF queries in the probabilistic RDF data graph.
- For the structural pruning, we propose effective methods to increase the pruning ability of graph synopses w.r.t. labels and graph structures, by adaptively considering the distribution of hashed labels based on our proposed cost model.
- For the probabilistic pruning, we design a cost model to formalize the query cost w.r.t. probability upper bounds. Then, we adaptively determine the number of pre-computed bounds that we store for the pruning, so as to achieve low query cost.
- We propose an effective index to summarize/store the pre-computed data in synopses, and present an efficient approach for query answering over probabilistic RDF data graph.

Section 5 shows through extensive experiments the efficiency and effectiveness of our approaches. Section 6 reviews previous works on query answering in certain RDF databases, as well as that in probabilistic databases. Finally, Section 7 concludes this paper.

2. PROBLEM DEFINITION

2.1 Probabilistic RDF Data Graph

Probabilistic RDF Graph Databases: First, we define a probabilistic RDF graph database $\mathcal{G}$ that contains probabilistic graphs G, modeled by *Bayesian networks* [29] in the probability theory.

DEFINITION 2.1. (*Probabilistic RDF Data Graph*) *A probabilistic RDF data graph $G \in \mathcal{G}$ is represented by a triple $(V(G), E(G), S(G))$. Here, we have:*

- *$V(G)$ is a finite set of vertices v_i with possible labels $l(v_i)$;*
- *$E(G)$ is a finite set of directed edges e_{ij} with labels $l(e_{ij})$;*
- *$S(G)$ is a finite set of conditional probability tables (CPTs), $T(v_i|pa(v_i))$, associated with vertices $v_i \in G$, which describe probabilities that v_i take labels $l(v_i)$, given that vertices $v_j \in pa(v_i)$ take some labels $l(v_j)$, where $pa(v_i)$ contains parent vertices that point to v_i via directed edges.*

In Definition 2.1, each vertex $v_i \in V(G)$ of graph G has one or multiple possible labels, $l(v_i)$, corresponding to either *subjects* or *objects* in RDF triples collected from data sources. Moreover, $e_{ij} \in E(G)$ is a directed edge from vertex v_i to vertex v_j, with a deterministic edge label $l(e_{ij})$ that corresponds to the *predicate* in RDF triples.

Furthermore, any CPT, $T(v_i|pa(v_i))$, in $S(G)$ stores the correlations of possible labels for vertex v_i and its parents in $pa(v_i)$. In particular, a CPT contains 3 types of attributes: labels $l(v_i)$, $l(v_j)$, and conditional probabilities $Pr\{l(v_i)|l(v_j)\}$, where v_j is the parent of v_i (i.e., $v_j \in pa(v_i)$).

In our previous example of Figure 2(b), the top vertex "$\otimes$" in the probabilistic RDF data graph is associated with a CPT, which contains 2 entries: $Pr\{$"Aarau"$|\ pid_1\} = 0.1$ and $Pr\{$"Athens" $|\ pid_1\} = 0.9$. Intuitively, this CPT stores the probability that top vertex "$\otimes$" takes a possible birthplace (e.g., "Aarau" or "Athens"), conditioned on its parent, that is, a particular person ID pid_1.

Remarks: Different from certain RDF graphs where every vertex has a deterministic label, the probabilistic RDF data graph in Definition 2.1 contains vertices with uncertain labels. In particular, set $V(G)$ includes vertices that can take different possible labels with some confidences, whereas set $S(G)$ has CPTs that store correlations of uncertain labels assigned to vertices. Note that, by storing different conditional probabilities among variables in CPTs, we can capture arbitrary correlations (including, but not limited to, mutually exclusive or independent relationship) for labels in vertices.

Moreover, in our model, each vertex of probabilistic RDF graphs is associated with uncertain labels, which is quite different from the model assumption in [15] that considers the existence probabilities of edges to appear in reality (with certain vertex labels).

Possible Worlds of Probabilistic RDF Data Graphs: In the literature of probabilistic databases [10], we usually consider the *possible worlds* semantics, where each possible world is a materialized instance of the database that can appear in the real world. Similarly, we also consider the possible worlds semantics in a probabilistic RDF data graph. As will be described later in Section 2.2, RDF queries are defined over such possible worlds.

DEFINITION 2.2. (*Possible Worlds of a Probabilistic RDF Data Graph, $pw(G)$*) *The possible world, $pw(G)$, of a probabilistic RDF graph $G \in \mathcal{G}$ is an instance of probabilistic graph G such that each vertex $v_i \in V(G)$ is assigned with a deterministic label $l(v_i)$.*

The certain RDF data graph shown in Figure 2(a) is exactly one possible world of the probabilistic RDF data graph in Figure 2(b), where each vertex is associated with a deterministic label (e.g., both vertices "$\otimes$" in Figure 2(b) take labels "Athens" in Figure 2(a)).

Each possible world $pw(G)$ of a probabilistic RDF graph G has its own appearance probabilities given below.

DEFINITION 2.3. (*The Appearance Probability of a Possible World, $Pr\{pw(G)\}$*) *Given a probabilistic RDF graph $G \in \mathcal{G}$, the appearance probability, $Pr\{pw(G)\}$, of possible world $pw(G)$ is:*

$$Pr\{pw(G)\} = \prod_{\forall v_i \in V(pw(G))} Pr\{l(v_i) \mid l(v_j), \forall v_j \in pa(v_i)\}, \quad (1)$$

where $pa(v_j)$ is a set of parent vertices of v_j.

Intuitively, Eq. (1) calculates the (joint) probability of one possible label binding (i.e., $l(v_i)$) for all vertices v_i in the probabilistic graph G, which can be obtained by multiplying conditional probabilities, $Pr\{l(v_i) \mid l(v_j), \forall v_j \in pa(v_i)\}$, in CPTs of $S(G)$.

In the previous example of Figure 2(b), assume that all vertices except for two "$\otimes$" have CPTs with conditional probabilities equal to 1. Then, one of its possible worlds, $pw(G)$, shown in Figure 2(a) has the appearance probability $Pr\{pw(G)\} = Pr\{$"Athens"$|pid_1\}$ $\cdot Pr\{$"Athens" $|\ pid_2\} = 0.9 \times 0.7 = 0.63$.

2.2 Subgraph Matching on Probabilistic RDF Data Graph

Queries Over Probabilistic RDF Graph Databases: As mentioned earlier, we can transform any SPARQL query to a query graph q (containing vertices with or without labels specified by the query issuer), and then retrieve those subgraphs, g, in probabilistic RDF data graphs G that match with q. In the probabilistic RDF graph database, we formally define RDF queries below.

DEFINITION 2.4. (*RDF Queries in the Probabilistic RDF Graph Database*) *Given a probabilistic RDF graph $G \in \mathcal{G}$, a query graph q, and a user-specified probabilistic threshold $\alpha \in [0, 1)$, a subgraph matching query obtains those subgraphs $g \in G$ and their label bindings $l(g_i)$ for vertices $g_i \in V(g)$, such that (1) g is isomorphic to q, and (2) $Pr\{g\} > \alpha$ holds.*

In Definition 2.4, a subgraph g in G matches with query graph q, if and only if two conditions are satisfied. The first condition indicates that subgraph g and query graph q should be isomorphic to each other. Note that, similar to SPARQL queries on certain RDF graph where it is rare to let predicates (edge labels) be variables (unspecified), we assume that all edge labels are specified in query graph q, and variables only appear in vertex labels of q[1]. Thus, g and q are isomorphic, if and only if they structurally match with each other, have the exact matching of edge labels, and have the same vertex labels in g as that specified in q. Finally, the second condition adds a probabilistic constraint to guarantee high confidences of query answers, and only returns those subgraphs g that appear in reality with probabilities $Pr\{g\}$ greater than α.

2.3 Major Challenges

According to the definition of subgraph matching on probabilistic RDF data graph, one straightforward method is to compare every possible subgraph g and its label bindings with the query graph q, and return those subgraphs that satisfy the two conditions given in Definition 2.4. Clearly, this method is not efficient, since there are exponential number of subgraphs $g \in G$, the cost of checking the isomorphism is high, and the computation of matching probabilities $Pr\{g\}$ (in the second condition) is also costly by considering exponential number of possible worlds $pw(g)$ of g. Therefore, this raises a serious problem about the query efficiency. Most importantly, a probabilistic RDF data graph has its own features, for example, its labels/degrees of vertices can be highly skewed, which may lead to low pruning power of query answering through indexes, and moreover, the correlated label information may cause the appearance probabilities difficult to compute.

[1]The case where an edge has a label variable can be easily handled by treating this edge as a special vertex with a label variable, connecting to its two adjacent vertices.

Symbol	Description
$\mathcal{G}$	a probabilistic RDF graph database
$V(G)$ $(E(G))$	a set of vertices (edges) in probabilistic RDF graphs G
$S(G)$	a set of conditional probability tables (CPTs) in probabilistic RDF graphs G
$pw(G)$	a possible world of probabilistic RDF graphs G
$Pr\{pw(G)\}$	the appearance probability of a possible world of G
q	an RDF query graph containing $\|q\|$ vertices
v_i, q_i, or g_i	the name of a vertex
$pa(v_i)$	a set of parent vertices of vertex v_i
e_{ij} (or e_g)	the name of a directed edge $\overrightarrow{v_i v_j}$ (or that in graph g)
$l(v_i)$ (or $l(e_{ij})$	the label of a vertex v_i (or a directed edge e_{ij})
α	a probabilistic threshold for RDF subgraph matching

Table 1: Symbols and descriptions.

Observing the challenges above, in this paper, we follow a filter-and-refine framework to first filter out those false alarms of subgraph candidates and then refine the remaining candidates. The detailed steps will be discussed in Section 3.1. Table 1 summarizes the commonly used symbols.

3. SUBGRAPH MATCHING OVER PROBABILISTIC RDF GRAPHS

3.1 Framework

Figure 4 illustrates the pseudo-code of the general framework for answering queries in a probabilistic RDF graph database, namely Prob_RDF_QA_Framework. In particular, this framework consists of 4 phases, *indexing*, *query pre-processing*, *pruning*, and *refinement phases*. The first indexing phase builds a tree index, $\mathcal{I}$, for synopses of subgraphs g in a probabilistic RDF graph database $\mathcal{G}$ (line 1). Next, for any RDF query with a query graph q, we access index $\mathcal{I}$ and obtain those matching subgraphs g (lines 2-5).

Specifically, upon the query request, the query pre-processing phase evaluates query graph q via estimations and chooses a good query plan that can achieve the expected low query cost (line 3). Then, we traverse index $\mathcal{I}$ in the pruning phase by applying the structural and probabilistic pruning on nodes/subgraphs in $\mathcal{I}$ (line 4). After the index traversal, we can combine and refine the remaining candidate subgraphs based on Definition 2.4 (line 5). Finally, we return actual RDF query answers to the query issuer (line 6).

3.2 RDF Query Pre-Processing

As illustrated in procedure Prob_RDF_QA_Framework (Figure 4), the query pre-processing phase needs to identify a good query plan to traverse the index $\mathcal{I}$ with low query cost.

In fact, our query procedure (discussed later in Section 4) first starts with vertices $q_i \in V(q)$ in the query graph q that have the user-specified labels $l(q_i)$, and then traverses index $\mathcal{I}$ to find their matching vertex candidates v_i in G in parallel, satisfying $l(v_i) = l(q_i)$ (filtering with other information, if possible). Meanwhile, we combine candidates v_i to obtain subgraphs g that match with q. To enable the pruning of candidate subgraphs, we can utilize the information such as the shortest path distances between two matching candidates v_i and v_j (w.r.t. q_i and q_j, respectively).

In particular, as will be illustrated in Section 3.3, we will check whether or not neighbors within several hops from the two candidates share any common labels, in order to prune with the graph distance. Thus, with a larger graph distance from q_i to q_j, we will consider more neighbor hops, and thus neighbors of candidates v_i and v_j are more likely to share common labels, which incurs lower pruning power and produces more intermediate results. Therefore, in order to enhance the pruning power, our query plan selection aims to find a number of close vertex pairs (q_i, q_j) in q (note: both $l(q_i)$ and $l(q_j)$ are specified by q), such that only a small number of false candidates (in intermediate results) are kept for further processing (join), and low query cost can be achieved.

Based on the intuition above, we aim to online obtain a set, *QPlan*, containing pairs of labeled vertices (q_i, q_j) $(i \neq j)$ in q

Procedure Prob_RDF_QA_Framework {
  Input: a probabilistic RDF graph database $\mathcal{G}$, a query graph q, and a probabilistic threshold α
  Output: subgraphs g from $\mathcal{G}$ that match with q with probability $Pr\{g\} > \alpha$
  (1) construct an index $\mathcal{I}$ for synopses of subgraphs g in $\mathcal{G}$
  // indexing phase
  (2) for any RDF query with query graph q
  (3) obtain a good query plan via estimations on q
  // query pre-processing phase
  (4) perform structural and probabilistic pruning while traversing index $\mathcal{I}$
  // pruning phase
  (5) combine and refine candidate subgraphs
  // refinement phase
  (6) return the matching subgraphs g
}

Figure 4: Framework for probabilistic RDF query answering.

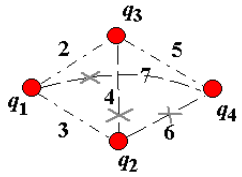


Figure 5: Example of obtaining query plan (*QPlan* =$\{(q_1, q_3), (q_1, q_2), (q_3, q_4)\}$).

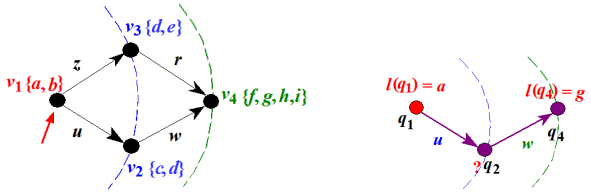
with small graph distances, such that we can use the graph distance pruning to filter out false alarms of candidates during the index traversal. In particular, we conceptually view each labeled vertex $q_i \in V(q)$ in q as a node in a conceptual graph K, and the shortest path length between pairwise labeled vertices, q_i and q_j, as the weight of their connecting edge in K. This way, we can obtain such a set *QPlan* by finding a *minimum spanning tree*, T, in the graph K, and storing their edges $E(T)$ in *QPlan*. Intuitively, the minimum spanning tree has edges with small weights (i.e., with small number of intermediate candidates) and has the minimum number of edges (i.e., $|V(T)|-1$) to connect all nodes in $V(T)$.

Algorithm of Obtaining a Query Plan: We illustrate the procedure of obtaining such a minimum spanning tree T (equivalent to a query plan), by using an example in Figure 5, where 4 vertices q_i $(1 \leq i \leq 4)$ in query graph q have user-specified labels (note: $|V(q)| > 4$). We first construct a conceptual graph K from the query graph q. In particular, graph K contains 4 labeled vertices q_i $(1 \leq i \leq 4)$ in q. Between any vertex pair (q_i, q_j) in K, there is a virtual edge, whose distance weight equals to the shortest path distance between q_i and q_j (computed by classical Dijkstra's algorithm). Note that, here we only obtain the weight without considering the path direction, which can facilitate the graph distance pruning discussed later. In Figure 5, (q_1, q_3) has distance weight 2, and in K we draw a dashed line between q_1 and q_3 to indicate the virtual edge.

To obtain a minimum spanning tree T from K, our algorithm starts from a virtual edge that has the smallest distance weight in K, and add it to *QPlan*. Then, the algorithm keeps adding edges to *QPlan* in non-descending order of weights and without introducing loops, until all edges in K are added/discarded. In the example of Figure 5, the initial edge in *QPlan* is (q_1, q_3) with the minimum weight 2. Next, we include edge (q_1, q_2) with the second smallest weight 3 in *QPlan*. Since the third shortest edge, (q_2, q_3), introduces a loop in *QPlan*, we discard it, and instead add the fourth shortest one (q_3, q_4). This way, we can obtain a final query plan *QPlan*, containing 3 virtual edges (q_1, q_3), (q_1, q_2), and (q_3, q_4).

3.3 Structural Pruning

In the sequel, we will illustrate the essential idea of our structural pruning, which uses properties of the graph structure to filter out false alarms that do not match with query graph q. Note that, the structural pruning has been extensively studied in the literature of certain graph databases [34]. Our problem can simply adopt the pruning techniques proposed in these works. Nevertheless, due to



(a) (b)

Figure 6: An example of structural pruning. (a) Probabilistic RDF data graph g. (b) RDF query graph q.

special features of probabilistic RDF graphs, these techniques may incur low filtering power. For example, the distribution of label frequencies in probabilistic RDF graphs can be quite skewed, which may produce many conflictions in synopses, leading to low query efficiency. Thus, after introducing our synopses to facilitate the structural pruning, we will propose adaptive synopses that consider label distributions in the probabilistic RDF graph.

3.3.1 Synopsis Design

Basically, since an RDF query specifies all edge labels and partial vertex labels, our synopses should be able to encode these edge and vertex labels for probabilistic RDF data graphs.

Synopses for Vertex Labels: Specifically, for each vertex $v_i \in V(G)$ and each of its possible labels $l(v_i)$, we perform a breadth-first search through out-going and in-coming edges separately[2]. For each direction, we can obtain labels within k-hop neighbors of v_i, for all $1 \leq k \leq d_{max}$, where d_{max} is the maximum possible diameter of query graph q.

Then, for each k value, we maintain a bit vector $BV_{\leq k}(v_i)$, in which all bits are initialized to 0. For simplicity, in this paper, we use bit vectors of the same size B, and we would like to leave the topic of using different vector sizes that are adaptive to k as our future work. We use a hash function, $H(z)$, to map the label of each v_i's neighbor with hop-counter not greater than k to a position in $BV_{\leq k}(v_i)$, and set its $H(l(v_i))$-th position to 1. Note that, to facilitate the pruning, we can select m (≥ 1) hash functions, denoted as $H^{(m')}(z)$ (for $1 \leq m' \leq m$).

Figure 6(a) shows an example of probabilistic RDF data graph, where possible (uncertain) labels of each vertex v_i are given in brackets. Starting from vertex v_1, 1-hop neighbors of v_1 include v_2 and v_3; similarly, the 2-hop neighbor of v_1 is v_4. As a result, for vertex v_1, we maintain two bit vectors, $BV_{\leq 1}(v_1)$ encoding labels $a \sim e$, and $BV_{\leq 2}(v_1)$ encoding labels $a \sim i$.

Synopses for Edge Labels: Similar to vertex labels, we can also construct synopses for edge labels. That is, for each vertex $v_i \in V(g)$, we traverse the graph from v_i in a breadth-first manner, and obtain edge labels within k hops from v_i, where $1 \leq k \leq d_{max}$. Then, for each k value, we construct a bit vector $BV^e_{\leq k}(v_i)$ of size B by hashing ($\leq k$)-hop edge labels to some positions in $BV^e_{\leq k}(v_i)$ via hash function $H(z)$.

In the example of Figure 6(a), bit vector $BV^e_{\leq 1}(v_1)$ encodes labels z and u, whereas $BV^e_{\leq 2}(v_1)$ contains hashed labels $\{z, u, r, w\}$.

3.3.2 Pruning With Synopses

We now illustrate how to utilize synopses designed above to prune false alarms for probabilistic RDF subgraph matching (in Definition 2.4). In brief, for any online query graph q, we can also construct synopses for vertex/edge labels (e.g., $BV_{\leq k}(q_i)$ or $BV^e_{\leq k}(q_i)$). Then, by comparing synopses and statistics between data and query graphs, we can obtain three pruning methods below, label pruning, graph distance pruning, and degree/counter pruning.

[2]Note that, to distinguish directions of edges will increase the pruning power. In the following discussions, however, to keep the notation simple, we will not explicitly mark distinct directions in symbols, if the context is clear.

Label Pruning: The label pruning method checks the existence of vertex/edge labels in data and query graphs, and prunes those candidate subgraphs g that do not correctly contain labels in query graph q. We first give the rationale of label pruning for vertices.

Pruning with Vertex Labels: Similar to the data graph, for any labeled vertex q_i in query graph q, we can obtain a bit vector $BV_{\leq k}(q_i)$. Then, intuitively, assuming that q matches with a subgraph $g \in G$ with $l(q_i) = l(v_i)$, it must hold that: for any position $1 \leq j \leq B$, if $BV_{\leq k}(q_i)[j] = 1$ holds, we have $BV_{\leq k}(v_i)[j] = 1$. We rewrite this matching property via bit-AND operation in the lemma below.

LEMMA 3.1. *Given a query graph q and a subgraph candidate g, if q structurally matches with g, then there must exist two vertices $q_i \in V(q)$ and $v_i \in V(g)$ with the same label (i.e., $l(q_i) = l(v_i)$), such that for any $1 \leq k \leq d_{max}$, we have:*

$$BV_{\leq k}(v_i) \wedge BV_{\leq k}(q_i) = BV_{\leq k}(q_i). \quad (2)$$

In the example of Figure 6, since query graph q in Figure 6(b) is subgraph matching to that in Figure 6(a), bit vector $BV_{\leq 1}(v_1)$ encodes labels $\{a \sim e\}$, which is a superset of that encoded by $BV_{\leq 1}(q_1)$ (i.e., $\{a\}$). The case where $k = 2$ is the same.

From Lemma 3.1, we can immediately have the following corollary to prune a subgraph g that does not structurally match with query graph q.

COROLLARY 3.1. (*Label Pruning*) *If for any vertex $v_i \in V(g)$ satisfying $l(v_i) = l(q_i)$, Eq. (2) does not hold for some $k \in [1, d_{max}]$, then subgraph g can be safely pruned.*

Intuitively, if Eq. (2) does not hold for some k value, then it indicates that there exists a label, $l(q_i)$, of some vertex within k hops from q_i, such that $l(q_i)$ does not appear in k-hop neighbors of v_i in subgraph g. This implies that at least v_i should not be matched to q_i. Thus, if this case happens for all vertices $v_i \in V(g)$ with the same label as q_i, then it is guaranteed that q does not match with g.

Pruning with Edge Labels: For the pruning with edge labels, we can simply replace the bit vector, $BV_k(v_i)$, for vertex labels in Corollary 3.1 with $BV^e_{\leq k}(v_i)$ for edge labels.

Graph Distance Pruning: The synopses in Section 3.3.1 can be also used to prune false alarms by considering the graph distance of vertex pair (v_i, v_j).

LEMMA 3.2. (*Graph Distance Pruning*) *Assume that we have $(q_i, q_j) \in QPlan$ ($i \neq j$), and it holds that $dist(q_i, q_j) = len$, where $dist(x, y)$ is the shortest path distance between vertices x and y in the graph. For vertices $v_i, v_j \in V(g)$, if it holds for some $k \in [1, len)$ that:*

$$BV_{\leq k}(v_i) \wedge BV_{\leq len-k}(v_j) = O, \quad (3)$$

then the candidate pair (v_i, v_j) can be safely pruned, where all bits in bit vector O equal to 0.

Intuitively, if a vertex pair (v_i, v_j) in G matches with (q_i, q_j) in q, then the shortest path distance $dist(v_i, v_j)$ from v_i to v_j in G should not be greater than that from q_i to q_j in q (i.e., $dist(q_i, q_j) = len$). In other words, there must exist some common labels between ($\leq k$)-hop neighbors of v_i and ($\leq len-k$)-hop neighbors of v_j. Thus, when we have bit-AND on their corresponding bit vectors (as given on the LHS of Eq. (3)), $BV_{\leq k}(v_i)$ and $BV_{\leq len-k}(v_j)$, there should exist some nonzero position.

For example, in Figure 6, since we have $dist(q_1, q_4) = len = 2$, (≤ 1)-hop neighbor of v_1 ($k = 1$) and (≤ 1)-hop neighbor of v_4 ($len - k = 1$) have common vertices v_2 and v_3 (labels as well).

In the case where Eq. (3) in Lemma 3.2 holds, it indicates that vertices v_i and v_j have their shortest path distance longer than len. Thus, candidate pair (v_i, v_j) does not match with (q_i, q_j), and can be safely pruned.

To further enhance the pruning power, we can take into account the directions of edges. That is, we construct bit vectors, denoted as $BV^+_{\leq k}(v_i)$ and $BV^-_{\leq k}(v_i)$, for $(\leq k)$-hop vertices that can reach v_i or that v_i can reach (i.e., considering incoming and out-going edges), respectively. Assume that $dist(v_i, v_j) = len_1$ and $dist(v_j, v_i) = len_2$. The pruning condition in Eq. (3) can be rewritten as:

$$BV^-_{\leq k}(v_i) \wedge BV^+_{\leq len_1 - k}(v_j) = O,$$

or

$$BV^-_{\leq k}(v_j) \wedge BV^+_{\leq len_2 - k}(v_i) = O.$$

Degree/Counter Pruning: Apart from the pruning with labels and graph distances, there are some other structural metrics to facilitate the pruning. For example, we can record the degree, $deg(v_i)$ of each vertex v_i, the total number, $cnt_k(v_i)$ (or $cnt^e_k(v_i)$), of vertices (or edges) within $(\leq k)$-hop neighbors of v_i, where $1 \leq k \leq d_{max}$. Correspondingly , for query graph q, we also have statistics such as $deg(q_i)$, $cnt_k(q_i)$, and $cnt^e_k(q_i)$ for vertex $q_i \in V(q)$. We use these metrics to prune false alarms in the following lemma.

LEMMA 3.3. (*Degree/Counter Pruning*) *For a query graph q and a subgraph candidate g with $l(v_i) = l(q_i)$, g can be safely pruned, if either of the following conditions holds: $deg(v_i) < deg(q_i), cnt_k(v_i) < cnt_k(q_i)$, or $cnt^e_k(v_i) < cnt^e_k(q_i)$.*

The reason that we use degree/counter pruning is that probabilistic RDF data graphs can sometimes be very sparse. That is, they may contain vertices of low degrees and a small number of vertices/edges within k hops of a vertex v_i. Thus, such structural statistics can be used as an effective means to filter out false alarms in probabilistic RDF data.

3.3.3 Adaptive Hashing for Synopses

The label pruning and graph distance pruning discussed in Section 3.3.2 utilize synopses, which are bit vectors summarizing labels via some hashing functions $H(z)$. Below, we will discuss how to choose good hash functions $H(z)$ that can achieve high filtering effect of our structural pruning.

Uniform Hashing Function, $H_{uni}(z)$: We first consider a straightforward uniform hashing function $H_{uni}(z)$. That is, for each label $l(v_i)$, we map $l(v_i)$ to a bit vector $BV(\cdot)$ such that each position between 1 and B has equal probability to be selected. With this uniform hashing function, although the resulting synopses can work correctly, the pruning power may not be high. This is because uncertain vertex/edge labels in RDF graphs can be quite skewed. While our structural pruning is essentially to check the existence of a label in a synopsis, those frequently appearing labels may have lower pruning power than those infrequently appearing ones. The hashing function $H_{uni}(z)$, however, maps them to positions with equal chance, regardless of their high or low frequencies. As a consequence, those infrequent labels may have confliction with frequent ones, that is, they are hashed to the same position in the bit vector, which makes infrequent labels unable to prune false alarms.

Adaptive Hashing Function, $H_{ada}(z)$: Based on heuristics above, we design an adaptive hashing function $H_{ada}(z)$, which takes into account label frequencies. Our basic idea is to hash those labels of low (high) frequencies to a position in the bit vector $BV(\cdot)$ with higher (lower) probabilities (inversely proportional to frequencies). This way, an infrequent label in a query graph q is more likely to prune a subgraph candidate g that does not contain this label (due to the lower confliction probability with frequent ones).

Let l_1, l_2, ..., and l_L be L labels with frequencies $w(l_1)$, $w(l_2)$, ..., and $w(l_L)$, respectively. We next aim to obtain m hashing functions $H^{(m')}_{ada}(z)$ that map these L labels l_i to some positions in $BV(\cdot)$ with probability inversely proportional to their frequencies, $w(l_i)$. In particular, each hashing function takes a random order of labels, denoted as $l_1^{(m')}$, $l_2^{(m')}$, ..., and $l_L^{(m')}$, as input. For brevity, we omit superscripts (m') below, when the context is clear.

Denote $W = \sum_{i=1}^{L} w(l_i)$. For each label l_i, we first randomly generate a real number, ran_i, within $[0, w(l_i))$. Then, our adaptive hashing function $H_{ada}(z)$ will map label l_i to a position:

$$H_{ada}(l_i) = \left\lfloor B \cdot \frac{\sum_{j=1}^{i-1} w(l_j) + ran_i}{W} \right\rfloor$$

in a bit vector $BV(\cdot)$ of size B. Intuitively, $H_{ada}(z)$ maps label l_i to a random position within $\left[\lfloor B \cdot \sum_{j=1}^{i-1} w(l_j)/W \rfloor, \lfloor B \cdot \sum_{j=1}^{i} w(l_j) /W \rfloor\right)$, in which each position contains this label with a probability proportional to $1/w(l_i)$. This way, a label with large $w(l_i)$ would have lower chance to conflict with an infrequent one at a position, which can achieve high pruning power.

For m different hashing functions $H^{(m')}_{ada}(l_i)$ (for $1 \leq m' \leq m$), we record m hashed positions for each label l_i, respectively, in a label dictionary, which is used for online look-up.

Analysis of Adaptive Hashing Function: The only remaining issue to address now is how to decide parameters in adaptive hashing functions, including the number, m, of hashing functions and the size, B, of bit vectors. In brief, we design a cost model which computes the maximum confliction probability, and then select good parameters (m, B) to achieve low confliction probability according to the cost model. Specifically, the confliction probability, $CP(l_i)$, that label l_i is expected to conflict with others is given by:

$$CP(l_i) = \frac{1}{L-1} \sum_{j \neq i \wedge j=1 \sim L} CP(l_i, l_j)$$

where $CP(l_i, l_j)$ is the confliction probability that labels l_i and l_j are mapped to the same position in $BV(\cdot)$.

Moreover, let X_r be a real random number independently and identically drawn from a variable following a probability density function $pdf(l_i) = w(l_i)/W$ $(1 \leq i \leq L)$, with mean μ and variance σ^2. We can derive the maximum confliction probability, that is, the upper bound of $CP(l_i, l_j)$, as follows.

$$\begin{aligned} & CP(l_i, l_j) \qquad (4) \\ &= Pr\left\{ \lfloor B \cdot (\sum_{r=1}^{i-1} X_r + ran_i/W) \rfloor = \lfloor B \cdot (\sum_{r'=1}^{j-1} X_{r'} + ran_j/W) \rfloor \right\} \\ &\leq Pr\left\{ \left| B \cdot (\sum_{r=1}^{i-1} X_r + ran_i/W) - B \cdot (\sum_{r'=1}^{j-1} X_{r'} + ran_j/W) \right| < 1 \right\}. \end{aligned}$$

Intuitively, the hashed position of a label l_i is given by $\lfloor B \cdot (\sum_{r=1}^{i-1} X_r + ran_i/W) \rfloor$. Eq. (4) derives an upper bound of probability $CP(l_i, l_j)$ that l_i and l_j are mapped to the same position, which can be obtained by applying central limit theory (CLT) [31].

Next, we consider our problem of using m hashing functions (w.r.t. m random orders of labels) in synopses. Let $kHop(v)$ be a set of labels for k-hop neighbors of vertex v. The confliction probability, $U(q_i, m, B)$, that a label $l(q_i)$ in query graph q is conflicting with labels of v_i's k-hop neighbors can be given by:

$$\begin{aligned} U(q_i, m, B) &= [1 - (1 - CP(l(q_i))^{|kHop(v)|}]^m \\ &\approx (1 - e^{-|kHop(v)| \cdot CP(l(q_i)}{})^m, \qquad (5) \end{aligned}$$

where $|kHop(v)|$ is the average number of labels in set $kHop(v)$.

Therefore, from Eq. (5), if we require that the confliction probability should be smaller than or equal to a threshold ε. Then, we can have the inequality: $U(q_i, m, B) \leq \varepsilon$, that is,

$$(1 - e^{-|kHop(v)| \cdot CP(l(q_i))})^m \leq \varepsilon, \qquad (6)$$

or equivalently,

$$m \geq \frac{ln(\varepsilon)}{ln(1 - e^{-|kHop(v)| \cdot CP(l(q_i))})}. \qquad (7)$$

As a result, we can take different possible values of B, and obtain the smallest value of m that satisfy Eq. (7). Our goal is to consider

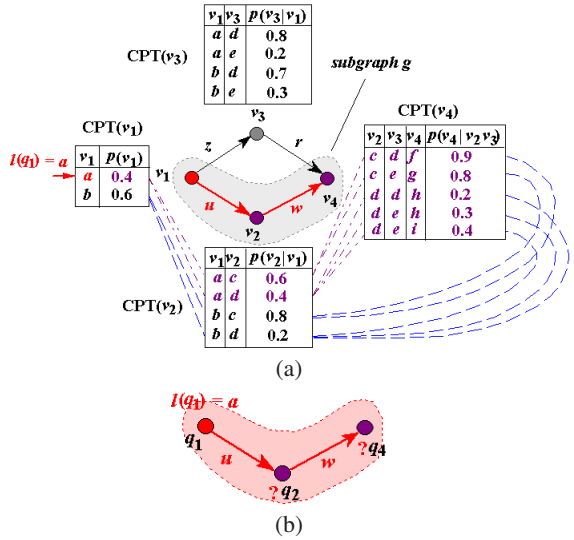


Figure 7: An example of probabilistic pruning. (a) Probabilistic RDF data graph G containing a subgraph g. (b) RDF query graph q.

different combinations of value pair (m, B), and then select one such that the required space $(m \cdot B)$ is minimized.

3.4 Probabilistic Pruning

In this subsection, we will consider unique probabilistic information in probabilistic RDF graphs, and propose an effective probabilistic pruning methods to filter out false alarms. From Definition 2.4, if a subgraph g matches with a query graph q, then it holds that its appearance probability, $Pr\{g\}$, should be above a probabilistic threshold α. However, the computation of $Pr\{g\}$ (in Eq. (1)) is costly, due to label correlations in an exponential number of possible worlds in probabilistic RDF data graphs.

Figure 7 illustrates an example of subgraph matching in a probabilistic RDF graph G, where a subgraph $g \in G$ and a query graph q are shown in Figures 7(a) and 7(b), respectively. In G, each vertex v_i (for $1 \leq i \leq 4$) is associated with a conditional probability table (CPT), CPT(v_i), which records correlations between vertex label $l(v_i)$ and that of its parent vertices $pa(v_i)$. One straightforward method to compute the appearance probability $Pr\{g\}$ of subgraph g is as follows. We first join all the CPTs, CPT(v_1) $\sim$ CPT(v_4), and meanwhile multiply conditional probabilities of joining tuples in CPTs, where (partial) join relationships of tuples between CPTs are represented by edges. Then, in the join result, we project on vertex labels needed in q, and eliminate duplicates by summing up their existence probabilities. Finally, we return those joining tuples with probabilities greater than α. However, this method is clearly time inefficient for the sake of join, duplicate elimination, and probability multiplication.

In order to efficiently answer RDF queries, in the sequel, we will propose an effective probabilistic pruning method to reduce the search space of finding matching subgraphs g. Specifically, our basic idea is to derive an upper bound, $UB\_P(g)$, for the appearance probability $Pr\{g\}$ of probabilistic subgraph g at a low cost. Then, if the probability upper bound is already smaller than or equal to threshold α, we can safely prune subgraph g. We summarize this pruning method in the following lemma.

LEMMA 3.4. (*Probabilistic Pruning*) *Given a subgraph g, let $UB\_P(g)$ be an upper bound of the appearance probability $Pr\{g\}$ (i.e., $UB\_P(g) \geq Pr\{g\}$). Then, for a given threshold $\alpha \in [0, 1)$, if it holds that $UB\_P(g) < \alpha$, we can safely filter out subgraph g.*

Intuitively, in Lemma 3.4, if $UB\_P(g) < \alpha$ holds, then we can infer that $Pr\{g\} < \alpha$, and safely discard g from the candidate list.

v_1	v_2	v_4	$Pr\{g\}$
a	c	f	$0.4 \times 0.6 \times 0.9$
a	c	g	$0.4 \times 0.6 \times 0.8$
a	d	h	$0.4 \times 0.4 \times (0.2 + 0.3)$
a	d	i	$0.4 \times 0.4 \times 0.4$

Table 2: Join results for subgraph g in the example of Figure 7.

Derivation of Probability Upper Bound $UB\_P(g, v_i)$: We now address the issue of deriving the upper bound, $UB\_P(g, v_i)$ of appearance probability $Pr\{g\}$. Our idea is based on the process of computing probability $Pr\{g\}$ among CPTs. In the sequel, we will use an example in Figure 7 to illustrate how to derive this upper bound from our observation.

As shown in Figure 7(b), query graph q contains a vertex q_1 with the user-specified label "a". Thus, in probabilistic RDF graph G (depicted in Figure 7(a)), we can first find vertex $v_1 \in V(g)$, which also has label "a". Then, we start from the first entry in CPT(v_1) (i.e., with label "a"), and join it with CPT(v_2), followed by CPT(v_4), through dash-dot lines (in Figure 7(a)) among CPTs.

In particular, when v_1 takes label "a", v_2 can take label "c" or "d" with conditional probability 0.6 or 0.4, respectively. If $l(v_2)$ ="c" holds, vertex v_4 has labels "f" and "g" with probabilities 0.9 and 0.8, respectively; otherwise, if $l(v_2)$ ="d" holds, then v_4 can take label "h" or "i" with probability 0.5 (=0.2+0.3) or 0.4, respectively. The final join results for $(v_1, v_2, v_4, Pr\{g\})$ are listed in Table 2, where the appearance probability, $Pr\{g\}$, of each possible subgraph g is given by multiplications of probabilities in CPTs.

Based on this observation, our rationale to obtain an upper bound of $Pr\{g\}$ is to overestimate the conditional probability in each CPT that participates in the calculation of $Pr\{g\}$ (as depicted in Table 2). In our example, when we start from label "a" of vertex v_1, we take probability 0.4 associated with "a" in CPT(v_1). Then, for CPT(v_2), we take the maximum probability 0.6 (instead of 0.4) for entries that can join with label "a" in CPT(v_1). Next, for CPT(v_3), we will take the largest probability 0.9 (among 4 possible probabilities in Table 2). This way, we can obtain a probability upper bound $UB\_P(g, v_1) = 0.4 \times 0.6 \times 0.9$ for subgraph g.

Formally, within each CPT(v_i), we project on each variable, $v_j \in pa(v_i)$, and v_i, and remove duplicate (label) tuples by summing up their existence probabilities. Then, for each label $l(v_j)$ of variable v_j, we can obtain a maximum existence probability $p_{max}(l(v_j))$, which can be used as a multiplicative factor to compute the probability upper bound for a subgraph g.

Our basic idea of calculating this upper bound in a probabilistic RDF data graph $G \in \mathcal{G}$ is as follows. For each entry in a CPT of vertex $v_i \in V(G)$, we aim to pre-compute a probability upper bound $UB\_P(g, v_i)$ for any subgraph $g \in G$ that contains (the entry in) vertex v_i, where the size of g can vary from 1 to n_{max}. Here, n_{max} is the maximum possible size of query graph q (i.e., the number of vertices, $|V(q)|$). Specifically, we follow the same procedure as joining the initial entry in CPT(v_i) with other CPTs in a breadth-first manner, but use upper bounds w.r.t. $p_{max}(l(v_j))$ (rather than exact existence probabilities in CPTs) in order to compute $UB\_P(g, v_i)$.

Let a function $P_{max}(n, i)$ represent the probability upper bound for any subgraph g (containing the initial entry in CPT(v_i)) of size n (i.e., $|V(g)| = n$), when we have accessed the i-th level of CPTs, denoted as S_i (note: "level" is the breadth-first search level). Then, we can obtain a recursive function for $P_{max}(n, i)$ below:

$$P_{max}(n, i) = \max \left\{ \max_{\forall v_l \in S_i} \left\{ P_{max}(n-1, i-1) \cdot \max_{\forall v_j} \{p_{max}(l(v_j))\} \right\}, P_{max}(n, i-1) \right\}. \quad (8)$$

Intuitively, within the outer $\max\{\cdot\}$ of Eq. (8), the first term computes a probability upper bound such that subgraph g contains

$(n-1)$ vertices within $(i-1)$ levels of CPTs (i.e., $S_1 \cup ... \cup S_{i-1}$), and moreover one vertex from the i-level CPTs (i.e., S_i); the second term, $P_{max}(n, i-1)$, is the probability upper bound that all n vertices are from $S_1 \cup ... \cup S_{i-1}$.

This way, by the recursive function in Eq. (8), we can offline pre-compute the probability upper bound $P_{max}(n, d_{max})$ for each possible n value, where $1 \leq n \leq n_{max}$ and d_{max} is the largest diameter of the query graph q. For any RDF query with a query graph of size n, to compare with a candidate subgraph g w.r.t. an entry in CPT(v_i), we only need to check whether or not $P_{max}(n, d_{max}) \leq \alpha$ holds for this entry (from the probabilistic pruning in Lemma 3.4). If the answer is yes, then we can safely prune this subgraph g.

Note that, although the above methods can pre-compute probability upper bounds for any query size n from 1 to n_{max}, it requires $O(n_{max})$ space cost for each entry in CPTs, which incurs much I/O cost. We observe that, when n becomes larger, the probability upper bound would decrease dramatically. Thus, it is not necessary to store all the n_{max} pre-computed bounds, so as to save the I/O cost. Inspired by this, in the sequel, we will first formalize the pruning power (I/O cost as well) for probabilistic pruning by a cost model, and then propose a novel approach to adaptively decide how many pre-computed values are necessary to guarantee low query cost.

Analysis of Probabilistic Pruning: To evaluate the effectiveness of our probabilistic pruning method (as mentioned in Lemma 3.4), below we provide a cost model to measure the probability that the pre-computed probability upper bounds can be used for pruning (i.e., $\leq \alpha$).

Specifically, assume that for each vertex $v_i \in G$, we offline pre-compute probability upper bounds $P_{max}(\cdot, d_{max})$ for subgraphs g with size from 1 to n_i. In the case where the actual query size $n > n_i$, we will simply use the pre-computed value w.r.t. n_i as the upper bound, since $P_{max}(n, d_{max}) \leq P_{max}(n_i, d_{max})$ holds for $n > n_i$. Furthermore, we assume that two parameters in users' RDF queries, the query size n and the probabilistic threshold $\alpha_n \in [\alpha_{min}, \alpha_{max}]$, follow a probability density function $pdf(n, \alpha_n)$ (which can be inferred from historical queries).

Then, we have the formula of pruning power as follows:

$$PP = \sum_{i=1}^{N} \left(\sum_{n=1}^{n_{max}} \int_{\alpha_{min}}^{\alpha_{max}} pdf(n, \alpha_n) \cdot \chi(F(n, n_i) < \alpha_n) d\alpha_n \right) \quad (9)$$

where $\chi(true) = 1$, $\chi(false) = 0$, and

$$F(n, n_i) = \begin{cases} P_{max}(n, d_{max}) & \text{if } n \leq n_i; \\ P_{max}(n_i, d_{max}) & \text{otherwise.} \end{cases} \quad (10)$$

In Eq. (9), the pruning power of each vertex v_i is given by:

$$PP_i = \sum_{n=1}^{n_{max}} \int_{\alpha_{min}}^{\alpha_{max}} pdf(n, \alpha_n) \cdot \chi(F(n, n_i) < \alpha_n) d\alpha_n$$

When we increase the number of pre-computed probability upper bounds n_i for vertex v_i, the pruning power is also expected to increase (due to the tighter bound); on the other hand, however, more bounds need to be stored on the disk, which incurs higher I/O costs to access them. Therefore, in the sequel, we will model such I/O cost for each vertex v_i, with which we can later select an optimal n_i value to achieve low query cost.

In particular, we denote the I/O cost of accessing v_i's pre-computed probability bounds as $G(v_i)$. We also assume that if v_i is a false alarm, then it can be quickly filtered out by probabilistic pruning via the index (discussed in Section 4)) without accessing the data, and thus its cost can be ignored. Therefore, we have:

$$G(v_i) = (1 - PP_i) \cdot (C_{IO} \cdot n_i) \quad (11)$$

where C_{IO} is a unit I/O cost for a pre-computed probability upper bound. In Eq. (11), the first term for $G(v_i)$ is the probability that v_i cannot be pruned by our probabilistic pruning, and the second term is the I/O cost that we have to access the pre-computed probability upper bounds.

Adaptive Pre-Computations Based on Cost Model: From our derived cost model $G(v_i)$ in Eq. (11) for the query cost of a vertex v_i, our goal is to adaptively obtain an optimal value of n_i (i.e., the number of pre-computed bounds we maintain for v_i), such that the query cost is the smallest. In fact, this goal can be achieved, when n_i satisfies the equality below:

$$\frac{\partial G(v_i)}{\partial n_i} = \frac{\partial((1 - PP_i) \cdot (C_{IO} \cdot n_i))}{\partial n_i} = 0, \quad (12)$$

which, by letting $J = 1 - PP_i$, can be rewritten as:

$$C_{IO} \cdot J + C_{IO} \cdot n_i \cdot \frac{\partial J}{\partial n_i} = 0. \quad (13)$$

As a result, for each vertex v_i, we will select an optimal value for n_i, such that by Eq. (13), $J + n_i \cdot \frac{\partial J}{\partial n_i} = 0$ holds, where J and $\frac{\partial J}{\partial n_i}$ are given in Appendix A.

4. PROBABILISTIC RDF QUERY ANSWERING

4.1 Index Construction

In this subsection, we illustrate the details of constructing a tree index, $\mathcal{I}$, for synopses of probabilistic RDF data graphs. As mentioned earlier in Section 3.3.1, we present the structural pruning methods that use structural information such as synopses for vertex/edge labels, degrees, and the number of edges/vertices to facilitate the pruning. Moreover, in Section 3.4, we provide the pre-computed probabilistic upper bound to enable probabilistic pruning. Therefore, we will integrate these synopses/parameters for our proposed pruning techniques into the index $\mathcal{I}$.

Data Format: In the index $\mathcal{I}$, we store possible labels of each vertex v_i in probabilistic RDF graphs G, as well as their surrounding structural/probabilistic information. In particular, for each label $l(v_i)$ of vertex v_i, we collect its synopses $BV_{\leq k}(v_i)$ and $BV^e_{\leq k}(v_i)$, degrees/counters $deg(v_i)$, $cnt_k(v_i)$, $cnt^e_k(v_i)$, and probabilistic upper bounds $F(n, n_i)$, where $1 \leq k \leq d_{max}$ and $1 \leq n \leq n_i$ for adaptive choice of n_i.

Index Structure: To index the probabilistic RDF data mentioned above, we will build a B$^+$-tree-like data structure, $\mathcal{I}$, whose primary sorting keys are (possible) labels of vertices in probabilistic RDF graphs. In leaf nodes, we store structural/probabilistic information of vertices in the aforementioned data format. On the other hand, each intermediate node N_a stores a summarization of synopses and parameters for all data under N_a. Specifically, for synopses, we have:

$BV_{\leq k}(N_a) = \bigvee_{\forall v_i \in N_a} BV_{\leq k}(v_i)$ and
$BV^e_{\leq k}(N_a) = \bigvee_{\forall v_i \in N_a} BV^e_{\leq k}(v_i)$.

For degree/counters, we have:

$deg(N_a) = \max_{\forall v_i \in N_a} deg(v_i)$,
$cnt_k(N_a) = \max_{\forall v_i \in N_a} cnt_k(v_i)$, and
$cnt^e_k(N_a) = \max_{\forall v_i \in N_a} cnt^e_k(v_i)$.

For probabilistic upper bound, we have:

$F(n, n_a) = \max_{\forall v_i \in N_a} \{F(n, n_i)\}$,

where $n_a = \max_{\forall v_i \in N_a} \{n_i\}$.

Note that, the pruning techniques for intermediate nodes are similar to that for vertex v_i, as described in Section 3. We only need to replace synopses/parameters for vertices with that for nodes .

Insert: We insert the label $l(v_i)$ of a vertex into $\mathcal{I}$ by a variant of "insert" operator in the B$^+$-tree. That is, we start with the tree root, $root(\mathcal{I})$, and always descend to subtrees that contain the label $l(v_i)$. When multiple subtrees contain the same label $l(v_i)$, we will choose one branch to descend, which is determined by bit vector

Procedure Prob_RDF_Processing {
Input: a probabilistic RDF graph database $\mathcal{G}$, a tree index $\mathcal{I}$ over $\mathcal{G}$, a query graph q with some , and a probabilistic threshold α
Output: subgraphs g that match with q with confidence α
(1) pre-process q to obtain the query plan, *QPlan* // pre-processing phase
(2) let *QS* be a set of vertices in q with user-specified labels
(3) obtain synopses and parameters for each $q_i \in QS$, and initialize an empty candidate list $cand(q_i)$ for each q_i
(4) for each entry N_a in $root(\mathcal{I})$
(5) for each $q_i \in QS$
(6) if N_a cannot be pruned by q via label/degree/counter/probabilistic pruning
(7) add N_a to $cand(q_i)$
(8) for $(q_i, q_j) \in QPlan$
(9) apply graph distance pruning to filter out false alarms of pairwise nodes between $cand(q_i)$ and $cand(q_j)$, and obtain candidate set $cand(q_iq_j)$
(10) while ($cand(\cdot)$ contains nodes)
(11) obtain children nodes/vertices via index $\mathcal{I}$ for each $N_i \in \bigcup_{\forall i} cand(q_i)$
(12) for each $(N_i, N_j) \in cand(q_iq_j)$
(13) use structural/probabilistic pruning to prune (N_a, N_b) for $N_a \in N_i$ and $N_b \in N_j$
(14) add (N_a, N_b) to $cand^{new}(q_iq_j)$ if it is not pruned
(15) $cand(q_iq_j) = cand^{new}(q_iq_j)$
(16) join candidate lists among $cand(q_iq_j)$ for (q_i, q_j) in *QPlan*
(17) refine candidate subgraphs in the join results and return the actual answers
}

Figure 8: Query answering in the probabilistic RDF graph database.

synopses, probabilistic upper bounds, degrees, and counters. For synopses $BV_{\leq k}(N_a)$, we consider the hamming distance between bit vectors before and after including new data of v_i; the case of $BV^e_{\leq k}(N_a)$ is similar. This way, we will select a branch with the minimum hamming distance to descend. In the case of tie, we consider the smallest increase of probability upper bounds, degrees, and counters in order.

Split: When a tree node N_a is full (i.e., overflows), we will split it into two new nodes N_{a1} and N_{a2}, which are of approximately equal size. The criteria of splitting are similar to the insertion. That is, the splitting should follow the order of vertex labels, $l(N_{a1}) \leq l(N_{a2})$. Moreover, with the same vertex label, two split groups should have small (summed) hamming distances between pairwise bit vectors, and small increases of probability upper bounds, degrees, or counters.

Delete: Since we assume static probabilistic RDF graphs without any deletions, we will not discuss it in detail. Nevertheless, it can be easily handled by removing the corresponding vertex labels from the index, and merging synopses/parameters of two nodes into one in the case of underflow.

4.2 Query Procedure

We next illustrate the pseudo code of our query procedure, namely Prob_RDF_Processing, for answering RDF queries through the tree index in Figure 8. Specifically, given a query graph q, we first pre-process g by obtaining a query plan, *QPlan*, for using graph distance pruning and joining pairs of candidate vertices during the index traversal (line 1). Moreover, for each vertex $q_i \in QS$ that has a user-specified label, we obtain its synopses and parameters used later for pruning methods, and also initialize an empty candidate list $cand(q_i)$ for candidate vertices v_i in RDF data (lines 2-3).

To start the index traversal, we first insert into candidate list $cand(q_i)$ those entries N_a in the root, $root(\mathcal{I})$, of index $\mathcal{I}$ that cannot be pruned by label/degree/counter/probabilistic pruning methods w.r.t. vertex $q_i \in QS$ (lines 4-7). Moreover, for pair (q_i, q_j) in *QPlan*, we obtain a set, $cand(q_iq_j)$, of candidate pairs via graph distance pruning (lines 8-9). Then, each time we access children nodes/vertices of nodes that appear in candidate lists $cand(\cdot)$, and apply our proposed structural/probabilistic pruning methods to obtain candidate pairs $cand(q_iq_j)$ for nodes/vertices on lower level of the index (lines 10-15). The iteration stops when the leaf nodes are visited (line 10). After that, we join candidate lists $cand(q_iq_j)$ for (q_i, q_j) in *QPlan*, and obtain a set of candidate subgraphs g (line 16). Finally, we refine candidate subgraphs g by checking their structures and appearance probabilities, and return the actual RDF query answers (line 17).

Parameters	Settings
α	$0.1, 0.2, \mathbf{0.5}, 0.8, 0.9$
$deg(v_i)$	$\mathbf{2}, 3, 5, 8, 10$
n_{max}	$2, 3, \mathbf{5}, 8, 10$
N	$10K, 20K, \mathbf{50K}, 80K, 100K$

Table 3: The experimental settings.

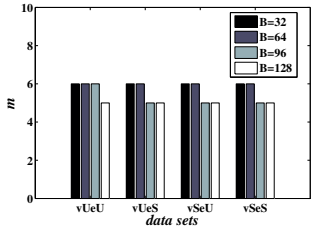
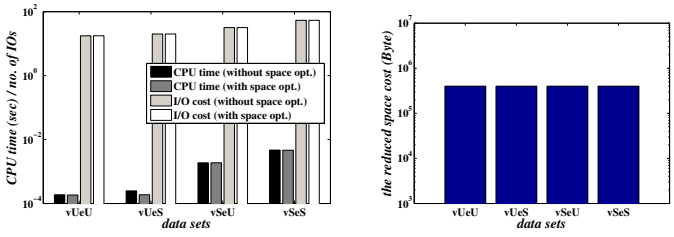


Figure 9: Choosing (B, m) parameter pairs with our cost model.

5. EXPERIMENTAL STUDY

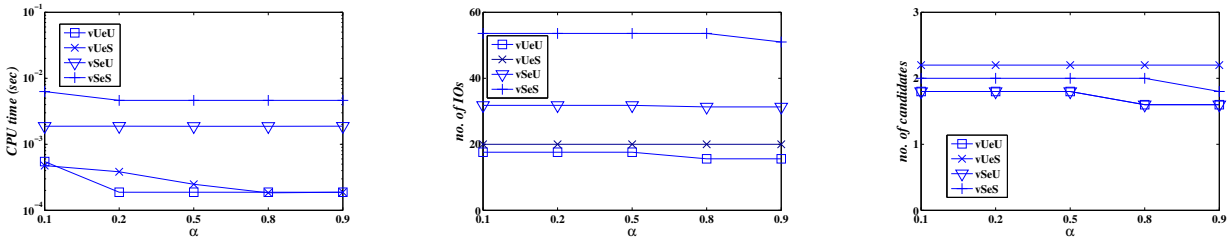
In this section, we evaluate the query performance of our proposed approaches for answering queries in probabilistic RDF graphs. Specifically, we use both real and synthetic data sets. For synthetic data, we generate an RDF graph by randomly connecting each vertex v_i with others in the graph with average in- and out-degrees $deg(v_i)$, and then for each vertex v_i (or edge e_{ij}), generating $L \in [L_{min}, L_{max}]$ possible labels, $l(v_i)$ (or one label $l(e_{ij})$ for e_{ij}), following either Uniform or Skew distribution. By considering Uniform/Skew (denoted as U and S) distributions of vertex/edge (v and e for short) labels, we can obtain 4 types of data sets, $vUeU$, $vUeS$, $vSeU$, and $vSeS$. In addition, for each vertex v_i, we construct a conditional probability table (CPT), CPT(v_i), which contains probabilities (randomly generated) that v_i has label $l(v_i)$, given the labels of its parent vertices. For real data, we used "directed_CheckedFactExtractor" in the YAGO [26] data set, which contains RDF data associated with the confidence among facts (i.e., subjects/objects). We construct data graphs from such real RDF data such that each triple corresponds to an edge (with predicate as the label) that connects two vertices (i.e., subject/object). Then, for each vertex, we assume 2 possible labels, one (i.e., object/subject name) is specified in data (with a confidence), and the other one is randomly produced from existing vertex labels. Similarly, we create a CPT for vertex v_i, such that in CPT the summed probability for each label of $v_j \in pa(v_i)$ (specified in the data) follows its own confidence. For other data sets with different parameter settings, the results are similar and thus omitted.

To evaluate the performance of RDF queries, we extract 50 query graphs q from probabilistic RDF data graphs G, by starting from any random vertex in G and then traversing the graph through connecting incoming / outgoing random edges, where the maximum number, n_{max}, of vertices in query graph q is 5 by default, and the diameter, d_{max}, of q is also set to 5. This way, for each vertex v_i we encounter, we obtain a corresponding vertex q_i in q, and set



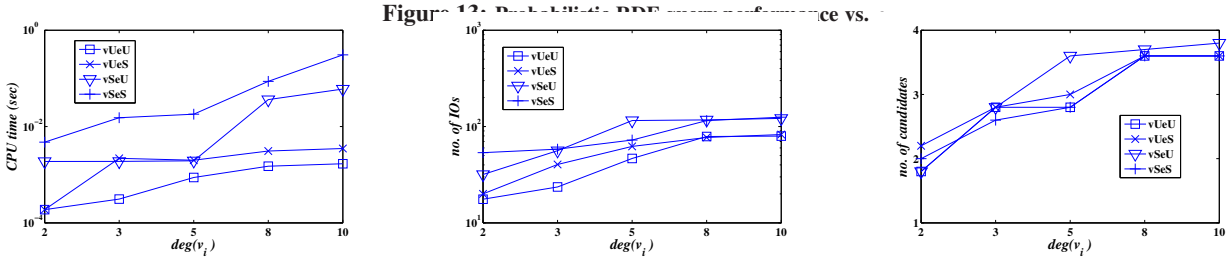
(a) (b)

Figure 10: Performance with and without space optimization.



(a) CPU time (b) No. of I/Os (c) No. of candidates

Figure 13: Probabilistic RDF query performance vs. α.



(a) CPU time (b) No. of I/Os (c) No. of candidates

Figure 14: Probabilistic RDF query performance vs. $deg(v_i)$.

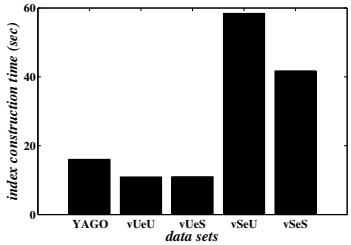
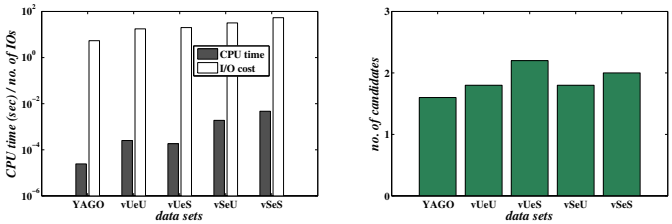


Figure 11: Index construction time for both real/synthetic data.



(a) (b)

Figure 12: Probabilistic RDF query efficiency on real/synthetic data.

label $l(q_i)$ to one random label of v_i. Note that, a straightforward baseline method to process RDF queries is to compare query graph q with at least N subgraphs of G, where $N = |V(G)|$. Clearly, since this baseline does not use any pruning methods, our proposed approach outperforms the baseline by orders of magnitude. Thus, we would not report comparison results with the baseline approach. Moreover, the work in [15] assumes the RDF model of with (independent) edge existence probabilities, whereas our problem considers a different one with correlated uncertainties in RDF labels. We thus do not compare with this work either, since our problem cannot be solved using techniques in [15].

We measure the performance of RDF queries in terms of the number of remaining subgraph candidates, CPU time (filtering via the index), and the number of I/Os. Table 3 summarizes experimental settings in our experiments, where the numbers in bold font are default values. In subsequent experiments, each time we vary one parameter, while setting others to default values.

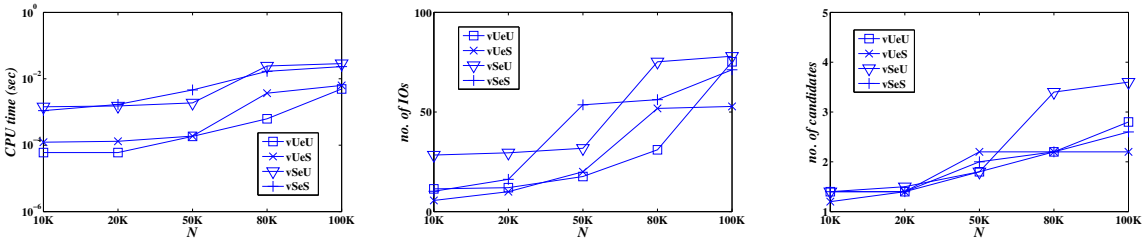
Effect of Adaptive Hashing Function: In the first set of experiments, we first decide 2 parameters B and m for the adaptive hash function H_{ada}, where B is the size of a bit vector, and m is the number of bit vectors for each synopsis (Section 3.3). We assume that the smallest synopsis unit is an integer of 4 Bytes (i.e., 32 bits). Thus, we will select a B value that is a multiple of 32. In our experiments, we test B values from 32 to 128, and set a smallest m value such that Inequality (7) holds, where we require the conflict probability be below 0.1 (i.e., $\varepsilon = 0.1$). Figure 9 shows the plot of possible (B, m) pairs by Inequality (7). When B increases in each data set, the change of m is not large (only decreases from 6 to 5). To achieve small synopsis size ($B \cdot m$)) yet guarantee good pruning power, below we will select the pair $(B, m) = (32, 6)$ with the minimum space consumption in experiments.

Effect of Adaptive Pre-Computations for Probabilistic Pruning: Figure 10(a) shows the time and I/O costs for using and without using adaptive pre-computations mentioned in Section 3.4. In the figure, the performance of using our adaptive pre-computations does not degrade much (in some cases, it performs even better due to less I/O cost with the reduced space). In Figure 10(b), we report the benefit from our adaptive pre-computations. That is, with a probabilistic RDF graph data containing $50K$ vertices, we can reduce the space cost of about $400K$ Bytes from index synopses.

Index Construction Time: Figure 11 reports the time cost of index construction over real/synthetic data sets, where default parameters are used for synthetic data sets. Here, the index construction time not only includes the time cost of building the tree index, but also include that of pre-computing synopses/statistics for indexing probabilistic RDF data graphs. In the figure, the required time is small, that is, from several seconds up to a minute.

Performance vs. Data Sets: We now show the performance of subgraph matching on probabilistic RDF graph data sets in Figure 12. From the figure, we can see that our filtering time through the index for real/synthetic data sets is low (i.e., below 0.01 second), and the number of page accesses is similar (only 5.4 and 15-54 for real and synthetic data, respectively). After filtering, the number of candidate graphs is small (i.e., 1-3), which can greatly save the refinement time (i.e., the isomorphism checking and computation of matching probabilities). In the sequel, we will report the robustness of our approach on synthetic data with different parameter settings.

Performance vs. Probabilistic Threshold α: Figure 13 varies the probabilistic threshold α from 0.1 to 0.9, under the default parameter settings. From figures, when α increases, the I/O cost and the number of candidates are non-increasing. This is because larger α threshold can possibly remove more false alarms with low confidences, which confirms the effect of our probabilistic pruning w.r.t. α (via probability upper bounds). Thus, larger α will reduce the I/O cost and lead to fewer candidates for refinement. Note



(a) CPU time (b) No. of I/Os (c) No. of candidates

Figure 15: Probabilistic RDF query performance vs. N.

that, the curves in figures are somewhat flat, for example, when $\alpha = 0.1, 0.2$, and 0.5, because probability upper bounds of most candidates are above the thresholds. Moreover, the time cost of our approach is small (i.e., below 0.1 second).

Performance vs. Degree $deg(v_i)$: In Figure 14, we vary the average degree of vertices for probabilistic RDF graphs from 2 to 10. From figures, higher degrees of vertices in probabilistic RDF graphs lead to the increase of both time and I/O costs. This is because RDF graphs with higher degrees would have more possible candidates when pruning with synopses. Further, the pre-computed probability upper bound is less tighter w.r.t. higher degrees, which incurs lower pruning power. Nonetheless, in figures, the CPU and I/O costs for probabilistic RDF data with different degrees are low, and the number of remaining candidates is small (i.e., around 2-4).

The performance trends w.r.t. parameter n_{max} values (i.e., the number of vertices in the query graph q) are similar to that of $deg(v_i)$ in Figure 14. Thus, we omit it due to the space limitation.

Scalability vs. Data Size N: Finally, Figure 15 reports the scalability of our approach against data size N varying from $10K$ to $100K$, with default values of other parameters. In figures, the time and I/O costs of our proposed approach smoothly increase with the increase of data size N. This is expected, since in a larger RDF graph, more subgraphs would match with query graph q. Nonetheless, the number of candidates is small, and the time cost for all data sizes remains low (i.e., below 0.1 second). This confirms the efficiency and effectiveness of our query answering approach in probabilistic RDF graph databases.

6. RELATED WORK

RDF is a W3C standard to represent information, which has nowadays gained much attention in real applications such as the Semantic Web. RDF data have several equivalent formats, for example, triple store [30, 19, 5], column store [25, 2, 23], property tables [33, 32], or graphs [3, 27]. Many existing works focus on improving the performance of answering SPARQL join queries in systems such as C-Store [2], RDF-3X [18], MonetDB [23], and Hexastore [30]. These works usually utilize indexes or join plans to speed up the RDF query efficiency. Tran et al. [27] explored the keyword search problem in a large RDF data graph, which retrieves combinations of keywords contained in subgraphs with the highest scores. This is different from our work in that the structure of the query graph is known in our problem. Thus, the direct usage of keyword search techniques to tackle our problem would incur many candidates (due to the unspecified structure among keywords).

In certain graph databases, Zhao and Han [34] designed a neighborhood signature to directly store labels within k hops from each vertex. In contrast, our work uses synopses to compactly store uncertain labels, which can efficiently filter out false alarms by the structural pruning via bit operators (e.g., bit-AND). Most importantly, our synopses can be also used to enable the graph distance pruning, and the synopsis design is based on an adaptive hashing (instead of uniform one) to increase the pruning power.

In XML databases, XSKETCH [21] and DescribeX [9] use summary subgraphs or paths in trees to represent XML features. Thus, they differ from synopses which encode uncertain labels of k-hop neighbors, and are specifically designed by considering probabilistic RDF features. Moreover, Gutierrez et al. [14] considered certain RDF data with additional temporal information, whereas our work studies RDF with uncertainty which involves label correlations and joint probabilities in possible worlds, and is thus more challenging.

All the aforementioned works assume that the underlying (RDF) graph data are reliable and precise. However, as mentioned in Section 1, during the data integration [11, 12], the integrated RDF data from different sources may often contain inconsistent or imprecise information. Therefore, they can be modeled as probabilistic RDF data. Due to the uncertainty in probabilistic RDF data, previous techniques for answering queries on certain RDF cannot guarantee high probabilistic confidence of query answers, and they are thus not directly applicable to our probabilistic RDF problem.

Some previous works [15, 13] in the Semantic Web considered the uncertainty in the RDF data. Huang and Liu [15] modeled such RDF data by a probabilistic database. However, this model assumes that RDF triples (corresponding to edges in RDF graphs) have independent existence probabilities to appear in reality. Based on this assumption, their query processing method is to decompose the query graph into triple patterns (edges), compute probabilities of candidates for each triple pattern, and finally join candidates by calculating their actual probabilities. In contrast, our problem involves a different model of probabilistic RDF graphs, which have correlated and uncertain vertex labels (but certain edge labels), rather than uncertain edges (certain labels). Therefore, techniques in [15] (proposed for RDF data with the edge independence and edge existence uncertainty) cannot be applied to solve our problem. Furthermore, Fukushige [13] extended the RDF vocabulary to allow the probabilistic confidence existing in RDF data (via a graphical model). However, it more focused on how to represent probabilistic RDF data, rather than how to efficiently answer probabilistic RDF queries, which is the focus of our problem in this paper.

In the literature of probabilistic databases [10], there are many existing systems that manage probabilistic and uncertain data, for example, MystiQ [7], Orion [8], TRIO [6], MayBMS [4], MCDB [16], and BayesStore [29]. The query evaluation usually considers possible worlds semantics, where each possible world is a possible instance of the probabilistic database in reality. That is, the query can be answered by obtaining query answers in each possible world, and then aggregating the returned answers. By using different aggregation methods, we can have distinct definitions of query semantics in probabilistic databases, for example, probabilistic top-k query [24, 17]. Potamias et al. [22] considered approximate approaches to obtain k-nearest neighbors of a query point in a probabilistic graph (considering the shortest path) by sampling possible worlds. In contrast, our work takes into account special features of probabilistic RDF graphs (w.r.t., label frequency/distributions and probabilistic correlations), and designs efficient approaches to improve the efficiency of RDF queries on probabilistic graph data.

7. CONCLUSION

Probabilistic RDF graphs pervasively exist in real applications such as data integration, where data often exhibit uncertainties. In this paper, we study the problem of retrieving subgraphs from probabilistic RDF graph databases that match with a given query graph with high confidence. To efficiently tackle this problem, we propose effective structural pruning methods with the help of synopses that are adaptively designed according to label distributions in RDF graphs. Further, we also utilize the probabilistic information in RDF graphs to enable the pruning, which is also adaptively designed according to a formal cost model, minimizing the query cost. We build a tree index structure that incorporates synopses and pre-computed data to facilitate online structural/probabilistic pruning. Extensive experiments have been conducted to demonstrate the efficiency and effectiveness of our approaches.

Acknowledgments

Funding for this work was provided by RGC NSFC JOINT Grant under Project No. N_HKUST61 2/09, HKUST RPC10EG13, and NSFC Grant No. 60736013, 60803105, 60873022, and 60903053.

8. REFERENCES

[1] W3C: Resource description framework (RDF). In *http://www.w3.org/RDF/*.
[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
[3] R. Angles and C. Gutierrez. Querying RDF data from a graph database perspective. In *The Semantic Web: Research and Applications*, 2005.
[4] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, 2007.
[5] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, 2010.
[6] O. Benjelloun, A. Das Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.
[7] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *SIGMOD*, 2005.
[8] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *VLDB*, 2005.
[9] Mariano P. Consens, Renée J. Miller, Flavio Rizzolo, and Alejandro A. Vaisman. Exploring XML web collections with describex. *TWEB*, 4(3), 2010.
[10] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4), 2007.
[11] X. L. Dong, L. Berti-Equille, and D. Srivastava. Integrating conflicting data: The role of source dependence. *PVLDB*, 2(1), 2009.
[12] X. L. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. *VLDBJ*, 18(2), 2009.
[13] Y. Fukushige. Representing probabilistic relations in RDF, 2005.
[14] C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing time into RDF. *TKDE*, 19(2), 2007.
[15] H. Huang and C. Liu. Query evaluation on probabilistic RDF databases. In *WISE*, 2009.
[16] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.
[17] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1), 2009.
[18] T. Neumann, M. Bender, S. Michel, R. Schenkel, P. Triantafillou, and G. Weikum. Distributed top-k aggregation queries at large. *Distributed and Parallel Databases*, 26(1), 2009.
[19] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1), 2008.
[20] A. Nierman and H. V. Jagadish. Protdb: Probabilistic data in XML. In *VLDB*, 2002.
[21] N. Polyzotis and M. Garofalakis. XSKETCH synopses for XML data graphs. *TODS*, 31(3), 2006.
[22] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-nearest neighbors in uncertain graphs. *PVLDB*, 3(1), 2010.
[23] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.
[24] M. A. Soliman, I. F. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.
[25] M. Stonebraker, D. J. Abadi, A. Batkin, X.Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, 2005.
[26] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semant.*, 6(3), 2008.
[27] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.
[28] D. Z. Wang, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein. Querying probabilistic information extraction. *PVLDB*, 3(1), 2010.
[29] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. Bayestore: Managing large, uncertain data repositories with probabilistic graphical models. In *VLDB*, 2008.
[30] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.
[31] E. W. Weisstein. Central Limit Theorem. *http://mathworld.wolfram.com/CentralLimitTheorem.html*.
[32] K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.
[33] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, and J. Database. Efficient RDF storage and retrieval in Jena2. In *ESWDB*, 2003.
[34] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1), 2010.

Appendix

A. Derivation of J and $\partial J/\partial n_i$ in Eq. (13).

We expand formulae of J and $\partial J/\partial n_i$ as follows.

$$
\begin{aligned}
J =& \ 1 - PP_i \qquad (14)\\
=& \int_{\alpha_{min}}^{\alpha_{max}} \sum_{n=1}^{n_{max}} pdf(n,\alpha_n)\cdot\chi(F(n,n_i)\geq\alpha_n)d\alpha_n\\
=& \int_{\alpha_{min}}^{F(n_i,n_i)} \sum_{n=1}^{n_{max}} pdf(n,\alpha_n)\cdot 1\, d\alpha_n\\
&+\int_{F(n_i,n_i)}^{\alpha_{max}} \left(\left(\sum_{n=1}^{n_{\alpha:F(n_\alpha,n_i)=\alpha_n}} pdf(n,\alpha_n)\cdot 1\right)\right.\\
&\left.+\left(\sum_{n=n_\alpha}^{\alpha_{max}} pdf(n,\alpha_n)\cdot 0\right)\right) d\alpha_n\\
=& \int_{\alpha_{min}}^{\alpha_{max}}\sum_{n=1}^{n_\alpha} pdf(n,\alpha_n)d\alpha_n + \int_{\alpha_{min}}^{F(n_i,n_i)} \sum_{n=n_\alpha+1}^{n_{max}} pdf(n,\alpha_n)d\alpha_n
\end{aligned}
$$

For probability upper bound function $f(n_\alpha) = P_{max}(n_\alpha, d_{max}) = \alpha$, we denote its inverse function $n_\alpha = f^{-1}(\alpha)$. Then, from Eq. (14), while the first term in J is not with respect to n_i, we have:

$$
\begin{aligned}
\frac{\partial J}{\partial n_i} &= \frac{d}{dn_i}\int_{\alpha_{min}}^{F(n_i,n_i)} \sum_{n=n_\alpha+1}^{n_{max}} pdf(n,\alpha)d\alpha\\
&= \frac{d}{dn_i}\int_{\alpha_{min}}^{f(n_i)} \sum_{n=n_\alpha+1}^{n_{max}} pdf(n,n_\alpha)\cdot df(n_\alpha)\\
&= \frac{d}{dn_i}\int_{f^{-1}(\alpha_{min})}^{n_i} \sum_{n=t+1}^{n_{max}} pdf(n,t)\cdot f'(t)dt\\
&= \sum_{n=t+1}^{n_{max}} pdf(n,t)\cdot f'(t)\,|_{f^{-1}(\alpha_{min})}^{n_i} \qquad (15)
\end{aligned}
$$

where $f'(\cdot)$ is the derivation of function $f(\cdot)$.