Effectively Interpreting Keyword Queries on RDF Databases with a Rear View

Haizhou Fu and Kemafor Anyanwu

Semantic Computing Research Lab, Department of Computer Science,
North Carolina State University, Raleigh NC 27606, USA
{hfu, kogan}@ncsu.edu

Abstract. Effective techniques for keyword search over RDF databases incorporate an explicit interpretation phase that maps keywords in a keyword query to structured query constructs. Because of the ambiguity of keyword queries, it is often not possible to generate a unique interpretation for a keyword query. Consequently, heuristics geared toward generating the top-K likeliest user-intended interpretations have been proposed. However, heuristics currently proposed fail to capture any user-dependent characteristics, but rather depend on database-dependent properties such as occurrence frequency of subgraph pattern connecting keywords. This leads to the problem of generating top-K interpretations that are not aligned with user intentions. In this paper, we propose a context-aware approach for keyword query interpretation that personalizes the interpretation process based on a user's query context. Our approach addresses the novel problem of using a sequence of structured queries corresponding to interpretations of keyword queries in the query history as contextual information for biasing the interpretation of a new query. Experimental results presented over DBPedia dataset show that our approach outperforms the state-of-the-art technique on both efficiency and effectiveness, particularly for ambiguous queries.

Keywords: Query Interpretation, Keyword Search, Query Context, RDF Databases.

1 Introduction

Keyword search offers the advantage of ease-of-use but presents challenges due to their often terse and ambiguous nature. Traditional approaches [1][2][8] for answering keyword queries on (semi)structured databases have been based on an assumption that queries are explicit descriptions of semantics. These approaches focus on merely matching the keywords to database elements and returning some summary of results i.e., IR-style approaches. However, in a number of scenarios, such approaches will produce unsatisfactory results. For example, a query like "*Semantic Web Researchers*" needs to be *interested* as a list of people, many of which will not have all keywords in their labels and so will be missed by IR-style approaches. For such queries, each keyword needs to be interpreted and the entire query needs to be mapped to a set of conditional expressions,

i.e., $WHERE$ clause and return clause. It is not often easy to find a unique mapping, therefore this problem is typically done as a top-K problem with the goal of identifying the K likeliest user intended interpretations.

Existing top-K query interpretation approaches [11] for RDF databases employ a cost-based graph exploration algorithm for exploring schema and data to find connections between keyword occurrences and essentially fill in the gaps in a keyword query. However, these techniques have the limitation of using a "*one-size-fits-all*" approach that is not *user-dependent* but rather more *database-dependent*. The heuristics used are based on the presumption that the likeliest intended interpretation is the interpretation that has the most frequent support in the database, i.e., the interpretation is related to classes of high-cardinality. Unfortunately, since such metrics are not user-dependent, the results generated do not always reflect the user intent.

In this paper, we address the problem of generating *context-aware query interpretations* for keyword queries on RDF databases by using information from a user's query history. The rationale for this is that users often pose a series of related queries, particularly in exploratory scenarios. In these scenarios, information about previous queries can be used to influence the interpretation of a newer query. For example, given a keyword query "*Mississippi River*", if a user had previously queried about "*Mortgage Rates*",then it is more reasonable to select the interpretation of the current query as being that of a *financial institution* "*Mississippi River Bank*". On the other hand, if a user's previous query was "*Fishing Techniques*", it may make more sense to interpret the current query as referring to a *large body of water*: the "*Mississippi River*". Two main challenges that arise here include (i) effectively capturing and efficiently representing query history and (ii) effectively and efficiently exploiting query history during query interpretation. Towards addressing these challenges we make the following ***contributions***:

i) Introduce and formalize the problem of *Context-Aware keyword query interpretation* on RDF databases.
ii) Propose and implement a *dynamic weighted summary graph model* that is used to concisely capture essential characteristics of a user's query history.
iii) Design and implement an efficient and effective top-K *Context-Aware graph exploration algorithm* that extends existing cost-balanced graph exploration algorithms, with support for biasing the exploration process based on context as well as with early termination conditions based on a notion of dominance.
iv) Present a comprehensive evaluation of our approach using a subset of the DBPedia dataset [3], and demonstrate that the proposed approach outperforms the state-of-the-art technique on both efficiency and effectiveness.

2 Foundations and Problem Definition

Let W be an alphabet of database tokens. An RDF database is a collection of subject-property-object triples linking RDF resources. These triples can be

represented as a graph $G_D = V_D, E_D, \lambda_D, \varphi_D$, where subject or object is represented as node in V_D while property is represented by edge in E_D. An object node can either represent another entity (RDF resource) or literal value. λ_D is a labeling function $\lambda_D : (V_D \cup E_D) \rightarrow 2^W$ that captures the *rdfs:label* declarations and returns a set of all distinct tokens in the label of any resource or property in the data graph. In addition, for any literal node $v_l \in V_D$, $\lambda_D(v_l)$ returns all distinct tokens in the literal value represented by v_l. φ_D is the incidence function: $\varphi_D : V_D \times V_D \rightarrow E_D$.

An RDF schema is also a collection of subject-property-object triples, which can also be represented as a graph: $G_S = (V_S, E_S, \lambda_S, \varphi_S, \pi)$, where the nodes in V_S represent classes and edges in E_S represent properties. λ_S is a labeling function $\lambda_S : V_S \cup E_S \rightarrow 2^W$ that captures the *rdfs:label* declarations and returns a set of all distinct tokens in the label of any class or property in the schema graph. φ_S is an incidence function: $\varphi_S : V_S \times V_S \rightarrow E_S$. $\pi : V_S \rightarrow 2^{V_D}$ is a mapping function that captures the predefined property *rdf:type* maping a schema node representing a class C to a set of data graph nodes representing instances of C. Nodes/edges in a schema can be organized in a subsumption hierarchy using predefined properties *rdfs:subclass* and *rdfs:subproperty*.

We define some special nodes and edges in the schema graph that are necessary for some of the following definitions:

- let $V_{LITERAL} \subset V_S$ be a set containing all literal type nodes (i.e., a set of literal nodes representing literal types such as "XSD:string");
- let $V_{LEAF\_CLASS} \subseteq (V_S - V_{LITERAL})$ be a set containing all leaf nodes (i.e. those nodes representing classes who do not have sub-classes) which are not literal type nodes;
- let $E_{LEAF\_PROPERTY} \subseteq E_S$ be a set containing all leaf edges (i.e. those edges representing properties which do not have sub-properties);
- let $V_{LEAF\_LITERAL} \subseteq V_{LITERAL}$ be a set containing all literal type nodes who are joined with leaf edges, for example, in Fig 1, literal type node $v_{string1}$ is in $V_{LEAF\_LITERAL}$ but $v_{string2}$ is not because the edge e_{name} connecting v_{Place} and $v_{string2}$ is not a leaf edge.

We define a keyword query $Q = \{w_1, w_2, \ldots, w_n | w_i \in W\}$ as a sequence of keywords, each of which is selected from the alphabet W. Given a keyword query Q, an RDF schema and data graphs, the traditional problem that is addressed in relation to keyword queries on RDF databases is how to translate an keyword (unstructured) query Q into a set of conjunctive triple patterns (structured query) that represents the intended meaning of Q. We call this process as ***keyword query "structurization"/interpretation***. To ensure that the structured query has a defined semantics for the target database, the translation process is done on the basis of information from the data and schema graphs. For example, given a keyword query "*Mississippi River Bank*", the schema graph and the data graph shown in Fig 1, we can find a structured query with conjunctive triple patterns listed at the top of Fig 1. Because of the class hierarchy defined in the schema, there could be many equivalent triple patterns for a given keyword query. For instance, in the schema graph of Fig 1, "*Organization*" is the

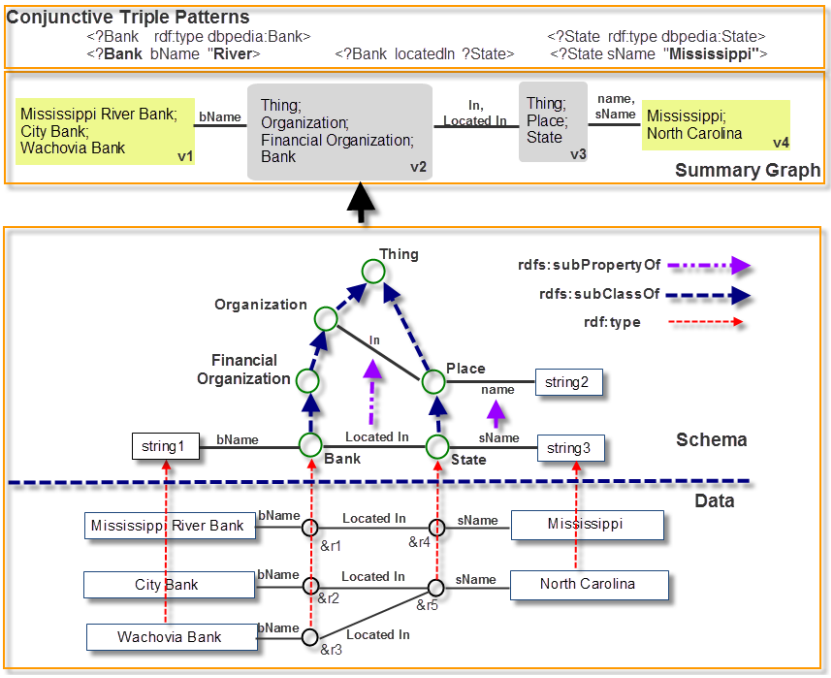

Fig. 1. Graph Summarization

super class of "*Bank*". Assuming that only "*Bank*" has the property "*bName*", thus, the two pattern queries:

⟨*?x bName "river"*⟩, ⟨*?x rdf:type Organization*⟩, and
⟨*?x bName "river"*⟩, ⟨*?x rdf:type Bank*⟩

are equivalent because the domain of the property "*bName*" requires that the matches of $?x$ can only be the instances of "*Bank*". To avoid redundancy and improve the performance, usually a summary graph structure is adopted that concisely summarizes the relationships encoded in the subsumption hierarchies and the relationships between tokens and the schema elements they are linked to.

Recall that our goal is to enable context-awareness for keyword query interpretation, we would also like this summary graph structure to encode information about a user's query history such as which classes have been associated with recent queries. This leads to a notion of a *context-aware summary graph* which is defined in terms of the concept of "*Upward Closure*":

DEFINITION 1 (***Upward closure***): Let v_C be a node in a schema graph G_S that represents a class C. The *upward closure* of v_C is $v_C^{\wedge}$, which is a set containing v_C and all the nodes representing super classes of C. For example, in Fig 1, the upward closure of the node v_{State} is: $v_{State}^{\wedge} = \{v_{Thing}, v_{Place}, v_{State}\}$. The *upward closure* of an edge $e_P \in E_S$ denoted by $e_P^{\wedge}$ is similarly defined.

DEFINITION 2 (***Context-aware Summary Graph***) : Given an RDF schema graph G_S, a data graph G_D and a query history QH: $QH = \{Q_1, \ldots, Q_T\}$, where Q_T is the most recent query, a *context-aware summary graph* can be defined as $SG = (V_{SG}, E_{SG}, \theta, \lambda_{SG}, \Psi_{SG}, \omega)$, where

- $\theta : V_{SG} \cup E_{SG} \rightarrow 2^{(V_S \cup E_S)}$ is an injective mapping function that maps any node or edge in SG to a set of nodes or edges in G_S.
- $V_{SG} = \{v_i | \exists u \in V_{LEAF\_CLASS} \cup V_{LEAF\_LITERAL}$ such that $\theta(v_i) = u^{\wedge}\}$.
  For example, the context-aware summary graph in Fig 1 contains $\{v1, v2, v3, v4\}$ four nodes, each of which can be mapped to the upward closure of one of the leaf nodes in $\{v_{string1}, v_{Bank}, v_{State}, v_{string3}\}$ in the schema graph respectively.
- $E_{SG} = \{e_i | \exists u \in E_{LEAF\_PROPERTY}$ such that $\theta(e_i) = u^{\wedge}\}$.
  For example, the summary graph in Fig 1 contains $\{e1, e2, e3\}$ three edges, each of which can be mapped to the upward closure of one of the leaf edges in $\{e_{bName}, e_{locatedIn}, e_{sName}\}$ respectively.
- λ_{SG} is a labeling function: $\lambda_{SG} : (V_{SG} \cup E_{SG}) \rightarrow 2^W$.
  - $\forall v \in V_{SG}$ where $\theta(v) = v_C^{\wedge}$ and $v_C \in V_S$ representing class C,
    $\lambda_{SG}(v) = \{\bigcup_{v_i \in v_C^{\wedge}} \lambda_S(v_i)\} \cup \{\bigcup_{r_j \in \pi(v_C)} \lambda_D(r_j)\}$,
    i.e., union of all distinct tokens in the labels of the super classes of C and distinct tokens in labels of all instances of C. For example, in Fig 1,
    $\lambda_{SG}(v4) = \{$ *"Mississippi", "North", "Carolina"*$\}$;
    $\lambda_{SG}(v3) = \{$ *"Thing", "Place", "State"*$\}$.
  - $\forall e \in E_{SG}$ where $\theta(e) = e_P^{\wedge}$ and $e_P \in E_S$ representing property P,
    $\lambda_{SG}(e) = \bigcup_{e_i \in e_P^{\wedge}} \lambda_S(e_i)$,
    which is a union of all distinct tokens in the labels of all super classes of P. For example,
    $\lambda_{SG}(e2) = \{$ *"LocatedIn", " In"*$\}$.
- Ψ_{SG} is the incidence function: $\Psi_{SG} : V_{SG} \times V_{SG} \rightarrow E_{SG}$, such that if $\theta(v_1) = v_{C1}^{\wedge}, \theta(v_2) = v_{C2}^{\wedge}, \theta(e) = e_P^{\wedge}$, then $\Psi_{SG}(v_1, v_2) = e$ implies $\varphi_S(v_{C1}, v_{C2}) = e_P$.
- $\omega : (QH, V_{SG} \cup E_{SG}) \rightarrow \mathbf{R}$ is a query history dependent weighting function that assigns weights to nodes and edges of SG. For a query history QH_{T-1} and $QH_T = QH_{T-1} + Q_T$, and $m \in SG$, $\omega(QH_{T-1}, m) \geq \omega(QH_T, m)$ if $m \in Q_T$.

Note that, we only consider user-defined properties for summary graph while excluding pre-defined properties. Further, we refer to any node or edge in a context-aware summary graph as a *summary graph element*.

DEFINITION 3 (***Hit***): Given a context-aware summary graph SG and a keyword query Q, a *hit* of a keyword $w_i \in Q$ is a summary graph element $m \in SG$ such that $w_i \in \lambda(m)$ i.e., w_i appears in the label of m. Because there could be multiple hits for a single keyword w, we denote the set of all hits of w as $HIT(w)$. For example, in Fig 1, $HIT(\text{“}bank\text{”}) = \{v1, v2\}$.

DEFINITION 4 (***Keyword Query Interpretation***): Given a keyword query Q and a context-aware summary graph SG, a *keyword query interpretation*

QI is a connected sub-graph of SG that connects at least one hit of each keyword in Q.

For example, the summary graph shown in Fig 1 represents the interpretation of the keyword query "*Mississippi, River, Bank*" which means "*Returning those banks in the Mississippi State whose name contains the keyword 'River'* ". The equivalent conjunctive triple patterns are also shown at the top of Fig 1. Note that for a given keyword query Q, there could be many query interpretations due to all possible combinations of hits of all keywords. Therefore, it is necessary to find a way to rank these different interpretations based on a cost function that optimizes some criteria which captures relevance. We use a fairly intuitive cost function in the following way: $cost(QI) = \sum_{m_i \in QI} \omega(m_i)$,

which defines the cost of an interpretation as a combination function of the weights of the elements that constitute the interpretation. We can formalize the *context-aware top-k keyword query interpretation problem* as follows:

DEFINITION 5 (***Context-aware Top-k Keyword Query Interpretation Problem***): Given a keyword query Q, and a context-aware summary graph SG, let $[[Q]] = \{QI_1, \ldots, QI_n\}$ be a set of all possible keyword query interpretations of Q, the *context-aware top-K keyword query interpretation problem* is to find the top K keyword query interpretations in IS: $TOPK = \{QI_1, \ldots, QI_K\} \subseteq [[Q]]$ such that

(i.) $QI_i \in TOPK$ and $QI_j \in ([[Q]] - TOPK)$, $cost(QI_i) \leq cost(QI_j)$.

(ii.) If $1 \leq p < q \leq k$, $cost(QI_p) \leq cost(QI_q)$, where $QI_p, QI_q \in TOPK$.

This problem is different from the traditional top-k keyword query interpretation problem in that the weights are dynamic and are subject to the evolving context of query history. Because some queries are more ambiguous than others, keyword query interpretation problem requires effective techniques to deal with large interpretation space. We propose a concept called ***Degree of Ambiguity (DoA)*** for characterizing the ambiguity of queries: The DoA of the keyword query Q is defined as $DoA(Q) = \prod_{w_i \in Q} |HIT(w_i)|$, which is the number of all combinations of keyword matches. It will be used as a performance metric in our evaluation.

Overview of our approach. Having defined the problem, we start with an overview of our approach. It consists of the following key steps as shown in Fig 2:

- Find keyword hits using an inverted index for a given keyword query Q. (Step (1)–(3)).
- The query interpreter takes the hits and utilizes a graph exploration algorithm to generate a set of top-K interpretations of Q. (Step (4)–(5))
- The top-1 interpretation of the top-K interpretation is passed to a cost model to update the weights of the context-aware summary graph. (Step (6)–(7))
- Steps involved in Fig 2 only capture one of the iteration cycles of the interactions between user and our interpretation system. The new weights of context-aware summary graph will be used to bias the graph exploration in the next iteration when user issues a new query

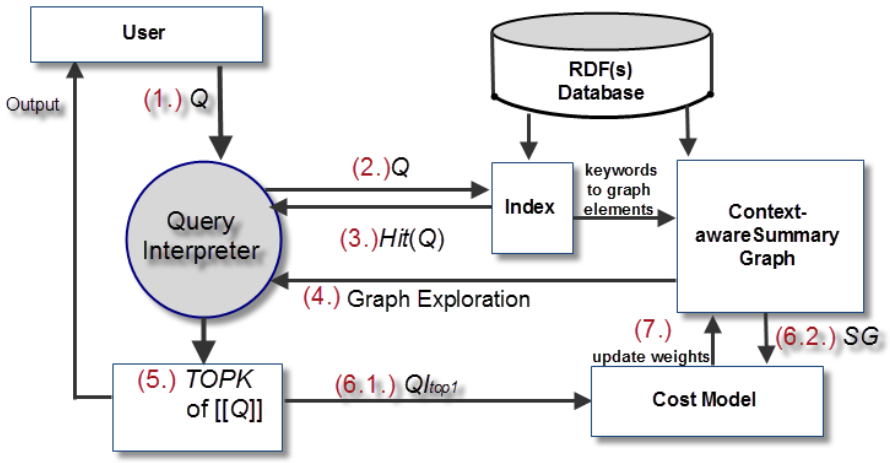

Fig. 2. Architecture and workflow

The cost model will be discussed in the next section and the graph exploration algorithm will be discussed in section 4.

3 Representing Query History Using A Dynamic Cost Model

The implementation of the dynamic cost model for representing query history consists of two main components: i) data structures for implementing a labeled dynamic weighted graph i.e., the context-aware summary graph; ii) a dynamic weighting function that assigns weights to summary graph elements in a way that captures their relevance to the current querying context. To understand what the weighting function has to achieve, consider the following scenario. Assuming that we have the following sequence of queries $Q1$ = "*Ferrari, price*" and $Q2$ = "*F1, calendar*", with $Q2$ as the most recent query. If $Q1$ is interpreted as "*Car*", the relevance score of the concept "*Car*" as well as related concepts (concepts in their immediate neighborhood such as "*Auto Engine*") should be increased. When $Q2$ arrives and is interpreted as "*Competition*", the relevance score for it should be increased. Meanwhile, since "*Auto Engine*" and all the other concepts that are not directly related to $Q2$, their relevance scores should be decreased. Then ultimately, for a new query $Q3$ = "*Jaguar, speed*", we will prefer the concept with higher relevance score as its interpretation, for example, we prefer "*Car*" than "*Mammal*".

To achieve this effect, we designed the dynamic weighting function to be based on a *relevance function* in terms of two factors: *historical impact factor* (hif) and *region factor* (rf).

Let T indicate the historical index of the most recent query Q_T, t be the historical index of an older keyword query Q_t, i.e., $t \leq T$, and m denote a

summary graph element. Assume that the top-1 interpretation for Q_t has already been generated : QI_t. *Region factor* is defined as a monotonically decreasing function of the graph distance $d(m, QI_t)$ between m and QI_t:

$rf(d(m, QI_t)) = \frac{1}{\alpha^{d(m,QI_t)}}$

$(rf(d(m, QI_t)) = 0 \; if \; d(m, QI_t) \geq \tau)$ τ is a constant value, and $d(m, QI_t)$ is the shortest distance between m and QI_t, i.e., among all the paths from m to *any* graph element in the sub-graph QI_t, $d(m, QI_t)$ is the length of the shortest path. Here, $\alpha > 1$ is a positive integer constant. The region factor represents the relevance of m to QI_t . *Historical impact factor* captures the property that the relevance between a query and a graph element will decrease when that query ages out of the query history. hif is a monotonically decreasing function: $hif(t) = 1/\beta^{T-t}$, where $\beta > 1$ is also a positive integer constant. We combine the two factors to define the relevance of m to query interpretation QI_t as $hif\,(t) * rf(d(m, QI_t))$. To capture the aggregate historical and region impacts of all queries in a user's query history, we use the combination function as the relevance function γ :

$$\gamma(m, T) = \sum_0^T hif\,(t) * rf\,(d(m, QI_t)) = \sum_0^T (1/\beta^{T-t})(1/\alpha^{d(m,QI_t)}) \quad (1)$$

To produce a representation of (1) for a more efficient implementation, we rewrite a function as recursive:

$$\gamma(m, T) = \gamma(m, T-1)/\beta + 1/\alpha^{d(m,QI_T)} \quad (2)$$

The consequence of this is that, given the relevance score of m at time $T-1$, we can calculate $\gamma(m, T)$ simply by dividing $\gamma(m, T-1)$ by β then adding $1/\alpha^{d(m,QI_T)}$. In practice, we use $d(m, QI_T) < \tau = 2$, so that, only m and the neighboring nodes and edges of m will be have their scores updated.

Boostrapping. At the initial stage, there are no queries in the query history, so the relevance score of the summary graph elements can be assigned based on the $TF - IDF$ score, where each set of labels of a summary graph element m i.e., $\lambda_{SG}(m)$ is considered as a document. User-feedback is allowed at every stage to select the correct interpretation if the top-1 query interpretation generated is not the desired one.

Since top-K querying generation is based on finding the smallest cost connected subgraphs of the summary, the definition of our weighting function for the dynamic weighted graph model is defined as the following function of the relevance function γ.

$$\omega(m, t) = 1 + 1/\gamma(m, t) \quad (3)$$

This implies that a summary graph element with a higher relevance value will be assigned a lower weight. In the next section, we will discuss how to find interpretations with top-k minimal costs.

4 Top-K Context-Aware Query Interpretation

The state of the art technique for query interpretation uses cost-balanced graph exploration algorithms [11]. Our approach extends such an algorithm [11] with a novel context-aware heuristic for biasing graph exploration. In addition, our approach improves the performance of the existing algorithm by introducing an early termination strategy and early duplicate detection technique to eliminate the need for duplicate detection as a postprocessing step. *Context Aware Graph Exploration* (*CoaGe*) algorithm shown in Fig 3.

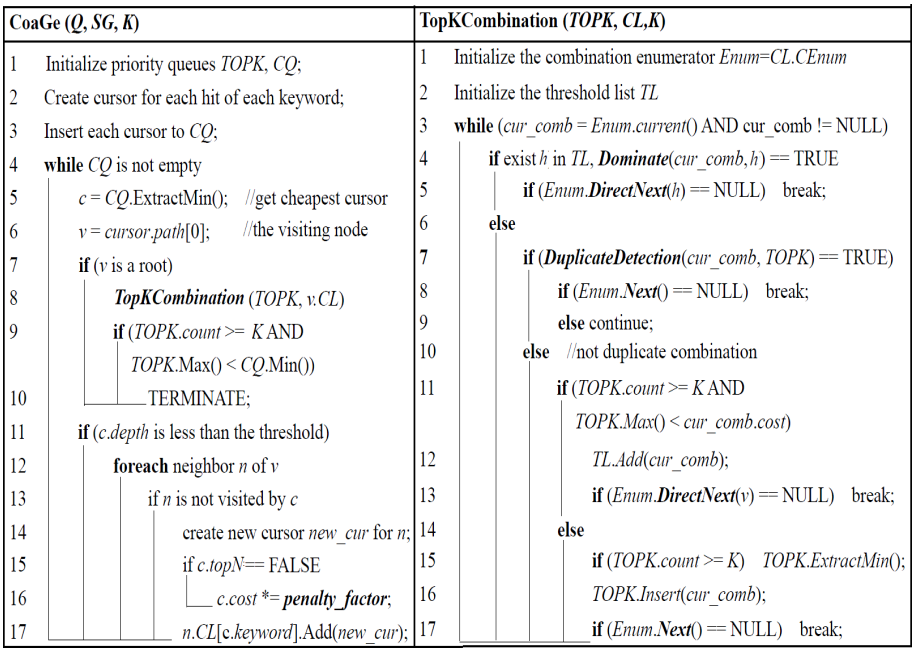

Fig. 3. Pseudocodes for CoaGe and TopCombination

4.1 CoaGe

CoaGe takes as input a keyword query Q, a context-aware summary graph SG and an integer value K indicating the number of candidate interpretations that should be generated. In *CoaGe*, a max binomial heap $TOPK$ is used to maintain top-K interpretations and a min binomial heap CQ is used to maintain cursors created during the graph exploration phase (line 1). At the initialization stage, for each hit of each keyword, *CoaGe* generates a cursor for it. A cursor originates from a hit m_w of a keyword w is represented as $c(keyword, path, cost, topN)$, where $c.keyword = w$; $c.path$ contains a sequence of summary graph elements

in the path from m_w to the node that c just visited; *c.cost* is the cost of the path, which is the sum of the weights of all summary graph elements in *c.path*; *c.topN* is a boolean value that indicates whether m_w is among the *top-N hits* of $HIT(w)$. The Top-N hit list contains the N minimum weighted hits of all hits in $HIT(w)$.

Each node v in the context-aware summary graph has a cursor manager CL that contains a set of lists. Each list in CL is a sorted list that contains a sequence of cursors for keyword w that have visited v, we use $CL[w]$ to identify the list of cursors for keyword w. The order of the elements in each list is dependent on the costs of cursors in that list. The number of lists in CL is equal to the number of keywords: $|CL| = |Q|$.During the graph exploration, the cursor with minimal cost is extracted from CQ (line 5). Let v be the node just visited by this "*cheapest*" cursor (line 6). *CoaGe* first determines whether v is a root (line 7). This is achieved by examining if all lists in $v.CL$ is not empty, in other words, at least one cursor for every keyword has visited v. If v is a root, then, there are $\prod_{w_i \in Q} |v.CL[w_i]|$ combinations of cursors. Each combination of cursors can be used to generate a sub-graph QI. However, computing all combinations of cursors as done in the existing approach [11] does is very expensive. To avoid this, we developed an algorithm *TopCombination* to enable early termination during the process of enumerating all combinations. *TopCombination* algorithm (line 8) will be elaborated in the next subsection. A second termination condition for the *CoaGe* algorithm is if the smallest cost of CQ is larger than the largest cost of the top-K interpretations (line 9). After the algorithm checks if v is a root or not, the current cursor c explores the neighbors of v if the length of *c.path* is less than a threshold (line 11). New cursors are generated (line 14) for unvisited neighbors of c (not in *c.path*, line 13). New cursors will be added to the cursor manager CL of v (line 17). The cost of new cursors are computed based on the cost of the path and if c is originated from a top-N hits.

Unlike the traditional graph exploration algorithms that proceed based on static costs, we introduce a novel concept of '*velocity*' for cursor expansion. Intuitively, we prefer an interpretation that connects keyword hits that are more relevant to the query history, i.e., lower weights. Therefore, while considering a cursor for expansion, it penalizes and therefore "*slows down*" the velocity of cursors for graph elements that are *not* present in the top-N hits (line 16). By so doing, if two cursors have the same cost or even cursor c_A has less cost than cursor c_B, but c_B originates from a top-N hit, c_B may be expanded first because the cost of c_A is penalized and $c_A.cost * penalty\_factor > c_B.cost$. The space complexity is bounded by $O(n \cdot d^D)$, where $n = \sum_{w_i \in Q} |HIT(w_i)|$ is the total number of keyword hits, $d = \Delta(SG)$ is the maximum degree of the graph and D is the maximum depth a cursor can explore.

4.2 Efficient Selection of Computing Top-k Combinations of Cursors

The *TopCombination* algorithm is used to compute the top-K combinations of cursors in the cursor manager CL of a node v when v is a root. This al-

gorithm avoids the enumeration of all combinations of cursors by utilizing a notion of *dominance* between the elements of CL. The ***dominance relationship*** between two combinations of cursors $Com_p = (CL[w_1][p_1], ...CL[w_L][p_L])$ and $Com_q = (CL[w_1][q_1], ...CL[w_L][q_L])$ is defined as follows: Com_p *dominates* Com_q, denoted by $Com^p \succ Com^q$ if for all $1 \leq i \leq L = |Q|$, $p_i \geq q_i$ and exists $1 \leq j \leq L$, $p_j > q_j$. Because every list $CL[w_i] \in CL$ is sorted in a non decreasing order, i.e., for all $1 \leq s \leq L$, $i \geq j$ implies that $CL[w_s][i].cost \geq CL[w_s][j].cost$. Moreover, because the scoring function for calculating the cost of a combination Com is a monotonic function: $cost(Com) = \sum_{c_i \in Com} c_i.cost$, which equals to the sum of the costs of all cursors in a combination, then we have:

$Com_p = (CL[w_1][p_1], CL[w_2][p_2], ..., CL[w_L][p_L])$

$\succ$

$(Com_q = CL[w_1][q_1], CL[w_2][q_2], ..., CL[w_L][q_L])$

implies that for all $1 \leq i \leq L$,

$CL[w_i][p_i].cost \geq CL[w_i][q_i].cost$ and $cost(Com_p) \geq cost(Com_q)$.

In order to compute top-k minimal combinations, given the combination Com_{max} with the max cost in the top-k combinations, we can ignore all the other combinations that dominate Com_{max}. Note that, instead of identifying all non-dominated combinations as in line with the traditional formulation, our goal is to find top-K minimum combinations that require dominated combinations to be exploited.

The pseudocodes of the algorithm $TopKCombination$ is shown in Fig 3. $TopKCombination$ takes as input a max binomial heap $TOPK$, a cursor manager CL and an integer value K indicating the number of candidate interpretations that should be generated. The algorithm has a combination enumerator $Enum$ that is able to enumerate possible combinations of cursors in CL (line 1). TL is initialized to contain a list of combinations as thresholds (line 2). The enumerator starts from the combination

$Com_0 = (CL[w_1][0], CL[w_2][0], ..., CL[w_L][0])$,

which is the "*cheapest*" combination in CL. Let

$Com_{last} = (CL[w_1][l_1], CL[w_2][l_2], ..., CL[w_L][l_L])$, be the last combination, which is the most "*expensive*" combination and $l_i = CL[w_i].length - 1$, which is the last index of the list $CL[w_i]$.

The enumerator outputs the next combination in the following way: if the current combination is

$Com_{current} = (CL[w_1][s_1], CL[w_2][s_2], ..., CL[w_L][s_L])$,

from 1 to L, $Enum.Next()$ locates the first index i, where $1 \leq i \leq L$ such that $s_i \leq l_i$, and returns the next combination as $Com_{next} =$

$(CL[w_1][0], ..., CL[w_{i-1}][0], CL[w_i][s_i + 1], ..., CL[w_L][s_L])$,

where, for all $1 \leq j < i$, s_j is changed from $l_j - 1$ to 0, and $s_j = s_j + 1$. For example, for $(CL[w_1][9], CL[w_2][5])$, if $CL[w_1].length$ equals to 10 and $CL[w_2].length > 5$, then, the next combination is $(CL[w_1][0], CL[w_2][6])$. The enumerator will terminate when $Com_{current} == Com_{last}$.

Each time $Enum$ move to a new combination cur_comb, it is compared with every combination in TL to check if there exists a threshold combination $h \in TL$

such that $cur\_comb \succ h$ (line 4). If so, instead of moving to the next combination using $Next()$, $Enum.DirectNext()$ is executed (line 5) to directly return the next combination that does not dominate h and has not been not enumerated before. This is achieved by the following steps: if the threshold combination is

$Com_{threshold} = (CL[w_1][s_1], CL[w_2][s_2], ..., CL[w_L][s_L])$,

from 1 to L, $Enum.DirectNext()$ locates the first index i, where $1 \leq i \leq L$ such that $s_i \neq 0$, and from $i+1$ to L, j is the first index such that $s_j \neq l_j - 1$, then the next generated combination is $Com_{direct\_next} =$

$(CL[w_1][0], ..., CL[w_i][0], ..., CL[w_{j-1}][0], CL[w_j][s_j + 1], ..., CL[w_L][s_L])$

where for all $i \leq r < j$, s_r is changed to 0, and $s_j = s_j + 1$. For example, for

$com_{threshold} = (CL[w_1][0], CL[w_2][6], CL[w_3][9], CL[w_4][2])$,

assume that the length of each list in CL is 10, then its next combination that does not dominate it is

$com_{direct\_next} = (CL[w_1][0], CL[w_2][0], CL[w_3][0], CL[w_5][3])$.

In this way, some combinations that could be enumerated by "*Next()*" function and will dominate $com_{threshold}$ will be ignored. For instance, $com_{next} = Next(com_{threshold}) =$

$(CL[w_1][1], CL[w_2][6], CL[w_3][9], CL[w_4][2])$,

and the next combination after this one: $Next(com_{next}) =$

$(CL[w_1][2], CL[w_2][6], CL[w_3][9], CL[w_4][2])$

will all be ignored because they dominate $com_{current}$.

If a new combination is "*cheaper*" than the max combination in $TOPK$, it will be inserted to it (line 16), otherwise, this new combination will be considered a new threshold combination, and inserted to TL (line 12) such that all the other combinations that dominate this threshold combination will not be enumerated. The time complexity of $TopKCombination$ is $O(K^k)$, where $K = |TOPK|$ is the size of $TOPK$, $k = |Q|$ is the number keywords. Because, for any combination

$com = (CL[w_1][s_1], ..., CL[w_L][s_L])$, where for all s_i, $1 \leq i \leq L$, $s_i \leq K$

$com_K = (CL[w_1][K+1], ..., CL[w_L][K+1]) \succ com$

In the worst case, any combinations that dominates com_K will be ignored and K^k combinations are enumerated. Consequently, the time complexity of $CoaGe$ is $O(n \cdot d^D \cdot K^k)$, where n is the total number of keyword hits, $d = \Delta(SG)$ is the maximum degree of the graph, D is the maximum depth. The time complexity of the approach in [11] (we call this approach $TKQ2S$) is $O(n \cdot d^D \cdot S^{D-1})$, where $S = |SG|$ is the number of nodes in the graph.

5 Evaluation

In this section, we discuss the experiments including efficiency and effectiveness of our approach. The experiments were conducted on a machine with Intel duel core 1.86GHz and 3GB memory running on Windows 7 Professional. Our test bed includes a real life dataset DBPedia, which includes 259 classes and over 1200 properties. We will compare the efficiency and effectiveness with $TKQ2S$.

5.1 Effectiveness Evaluation

Setup. 48 randomly selected college students were given questionnaires to complete. The questionnaire contains 10 groups of keyword queries (To minimize the cognitive burden on our evaluators we did not use more than 10 groups in this questionnaire). Each group contains a short query log consisting of a sequence of up to 5 keyword queries from the oldest one to the newest one. For each group, the questionnaire provides English interpretations for each of the older queries. For the newest query, a list of candidate English interpretation for it is given, each interpretation is the English interpretation representing a structured query generated by either $TKQ2S$ or $CoaGe$. Therefore, this candidate interpretation list provided to user is a union of the results returned by the two algorithm. Then users are required to pick up to 2 interpretations that they think are the most intended meaning of the newest keyword query in the context of the provided query history. A consensus interpretation (the one that most people pick) was chosen as the desired interpretation for each keyword query.

Metrics. Our choice of a metric of evaluating the query interpretations is to evaluate how relevant the top-K interpretations generated by an approach is to the desired interpretation. Further, it evaluates the quality of the ranking of the interpretations with respect to their relative relevance to the desired interpretation. Specifically, we adopt a standard evaluation metric in IR called "Discounted cumulative gain (DCG)" with a refined relevance function: $DCG_K = \sum_{i=1}^{K} \frac{2^{rel_i}-1}{\log_2(1+i)}$, where K is the number of top-K interpretations generated, rel_i is the graded relevance of the resultant interpretation ranked at position i. In IR, the relevance between a keyword and a document is indicated as either a match or not, rel_i is either zero or one. In this paper, the relevance between a resultant interpretation QI and a desired interpretation QI_D for a given keyword query Q cannot be simply characterized as either a match or not. QI and QI_D are both sub-graphs of the summary graph and could have some degree of overlapping, which means $rel_i \in [0, 1]$. Of course, if $QI == QI_D$, QI should be a perfect match. In this experiment, we define the relevance between a candidate interpretation QI and the desired interpretation QI_D as: $rel_i = \frac{|QI_i \cup QI_D| - |QI_i \cap QI_D|}{|QI_i \cup QI_D|}$, where QI_i is the interpretation ranked at position i. rel_i returns the fraction of those overlapping summary graph elements in the union of the the two sub-graphs. Large overlapping implies high similarity between QI_i and QI_D, and therefore, high relevance score. For example, if QI_i has 3 graph elements representing a class "*Person*", a property "*given name*" and a class "*XSD:string*". The desired interpretation QI_D also has 3 graph elements, and it represents class "*Person*", a property "*age*" and a class "*XSD:int*". Therefore, the relevance between QI_i and QI_D is equal to $1/5 = 0.2$ because the union of them contains 5 graph elements and they have 1 common node.

On the other hand, we use another metric precision to evaluate the results. The precision of a list of top-K candidate interpretation is:

$P@K = |$ relevant interpretations in $TOPK|/|TOPK|$,

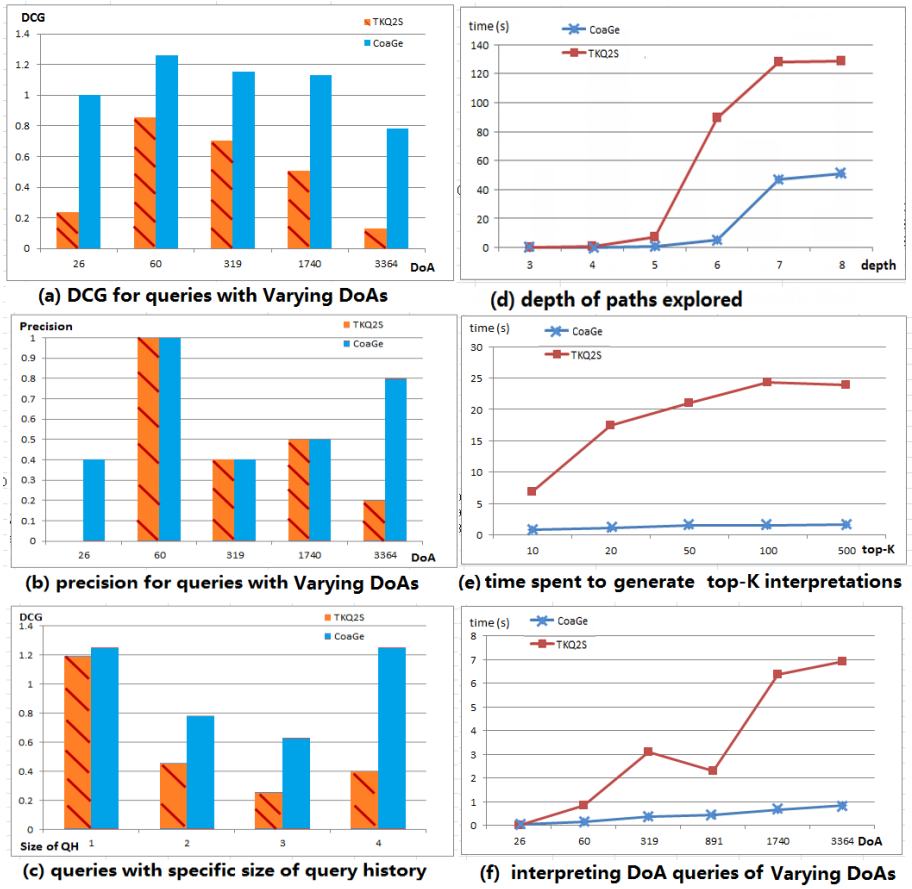

Fig. 4. Efficiency and effectiveness evaluation

which is the proportion of the relevant interpretations that are generated in $TOPK$. Because sometimes, when the user votes are evenly distributed, the consensus interpretation cannot represent the most user intended answer, our evaluation based on DCG may not be convincing. The precision metric can overcome this limitation by consider the candidate interpretations that over 10% people have selected as desired interpretations.

Discussion. We compared the DCG of the results returned by our approach and $TKQ2S$. Top-8 queries are generated for each algorithm for each keyword query. The result shown in Fig 4 (a) illustrates the quality of interpreting queries with varying DoA values. From (a), we can observe that $TKQ2S$ does not generate good quality interpretations for queries with high DoA. The reason is that they prefer the popular concepts and rank the desired interpretation which is not popular but is higher relevant to the query history at a low position. It also

does not rank higher relevant interpretations higher. Fig 4 (b) illustrates the precision of the top-5 queries generated by each algorithm. In most of cases, $TKQ2S$ generates same number of relevant queries as $CoaBe$, but it fails to generate enough relevant interpretations for the last query with $DoA = 3364$. For the first query in (b), $TKQ2S$ does not output any relevant interpretations, therefore, the precision is 0. Fig 4 (c) illustrates how different lengths of query history will affect results. In this experiment, 4 groups of queries are given, the ith group contains i queries in the query history. Further, the ith group contains all the queries in the $(i-1)$th group plus a new query. Given the 4 different query histories, the two algorithms are to interpret another query Q. (c) illustrates the quality of interpreting Q given different query histories. We can observe that our approach will do better with long query history. But for the first group, both algorithms generate a set of interpretations that are very similar to each other. Both the DCG values are high because user have to select from the candidate list as the desired interpretation, even though they may think none of them is desired. For the third group, that difference in performance is due to a transition in context in the query history. Here the context of query changed in the 2nd or 3rd query. This resulted in a lower DCG value which started to increase again as more queries about new context were added.

5.2 Efficiency Evaluation

From the result of the efficiency evaluation in Fig 4 (d)–(f), we can see that, our algorithm outperforms $TKQ2S$ especially when the depth (maximum length of path a cursor will explore) and the number of top-K interpretations and the degree of ambiguity DoA are high. The performance gain is due to the reduced search space enabled by early termination strategy using the $TopKCombination$ algorithm.

6 Related Work

There is a large body of work supporting keyword search for relational databases [1][2][7][8] based on the interpretation as "match" paradigm. Some recent efforts such as Keymantic[5] QUICK[13], SUITS[6], Q2Semantics[12] and [11] have focused on introducing an explicit keyword interpretation phase prior to answering the query. The general approach used is to find the "*best*" sub-graphs (of the schema plus a data graph) connecting the given keywords and represent the intended query meaning. However, these techniques are based on fixed data-driven heuristics and do not adapt to varying user needs. Alternative approaches [6][13] use techniques that incorporate user input to incrementally construct queries by providing them with query templates. The limitation of these techniques is the extra burden they place on users.

Query history has been exploited in IR [4][9][10]. These problems have different challenges from the ones addressed in this work. The similarity measures are based on mining frequent query patterns. However, we need to exploit the

query history to identify similar search intent, which may not necessarily be the most frequent query patterns. Our problem requires unstructured queries, their intended interpretations (structured queries) and the ontology to be managed.

7 Conclusion and Future Work

This paper presents a novel and effective approach for interpreting keyword queries on RDF databases by integrating the querying context. In addition to the techniques proposed in this paper, we plan to explore the idea of exploiting the answers to a priori queries for query interpretation.

Acknowledgement. The work presented in this paper is partially funded by NSF grant IIS-0915865.

References

1. Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, and Sudarshan, S. Banks: Browsing and keyword searching in relational databases. In *VLDB* (2002), pp. 1083–1086.
2. Agrawal, S., Chaudhuri, S., and Das, G. Dbxplorer: enabling keyword search over relational databases. In *SIGMOD Conference* (2002), p. 627.
3. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. G. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC* (2007), pp. 722–735.
4. Bar-Yossef, Z., and Kraus, N. Context-sensitive query auto-completion. In *WWW* (2011), pp. 107–116.
5. Bergamaschi, S., Domnori, E., Guerra, F., Lado, R. T., and Velegrakis, Y. Keyword search over relational databases: a metadata approach. In *SIGMOD Conference* (2011), pp. 565–576.
6. Demidova, E., Zhou, X., Zenz, G., and Nejdl, W. Suits: Faceted user interface for constructing structured queries from keywords. In *DASFAA* (2009), pp. 772–775.
7. He, H., Wang, H., Yang, J., and Yu, P. S. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference* (2007), pp. 305–316.
8. Hristidis, V., and Papakonstantinou, Y. Discover: Keyword search in relational databases. In *VLDB* (2002), pp. 670–681.
9. Kelly, D., Gyllstrom, K., and Bailey, E. W. A comparison of query and term suggestion features for interactive searching. In *SIGIR* (2009), pp. 371–378.
10. Pound, J., Paparizos, S., and Tsaparas, P. Facet discovery for structured web search: a query-log mining approach. In *SIGMOD Conference* (2011), pp. 169–180.
11. Tran, T., Wang, H., Rudolph, S., and Cimiano, P. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE* (2009), pp. 405–416.
12. Wang, H., Zhang, K., Liu, Q., Tran, T., and Yu, Y. Q2semantic: A lightweight keyword interface to semantic search. In *ESWC* (2008), pp. 584–598.
13. Zenz, G., Zhou, X., Minack, E., Siberski, W., and Nejdl, W. From keywords to semantic queries - incremental query construction on the semantic web. *J. Web Sem.* 7, 3 (2009), 166–176.