ABSTRACT

FU, HAIZHOU. Efficient, Effective, and Scalable Personalized Keyword Query Interpretation for RDF Databases. (Under the direction of Dr. Kemafor Anyanwu.)

The ease of use of the keyword search has made it an increasingly popular paradigm for querying (semi) structured data. Existing techniques supporting the keyword search are based on information retrieval (IR) paradigms. IR-style search paradigms follow a “semantics as match” style where objects containing keyword matches are returned as results that are not necessarily equivalent to the exact user-intended meaning of a query. Using such techniques leads to answer sets with heterogeneous semantics, leaving the user with the burden of sifting through the query results to identify the intended answers. Recent approaches supporting the keyword search incorporate an “interpretation” phase as part of the query-answering process. In the interpretation of keywords, the latter are mapped to structured constructs of queries. However, keyword queries are often ambiguous because of their scanty and unstructured nature. Therefore, it is possible that there is no unique interpretation for a keyword query. Consequently, heuristics aimed to generate the top-K most likely user-intended interpretations have been proposed. Unfortunately, the heuristics used by existing techniques often are not geared toward capturing user-dependent characteristics. Instead, they depend on database-dependent properties, such as the frequency of occurrence of database terms. This leads to the problem of generating interpretations that are not aligned with the intensions of users.

In this thesis we present an efficient, effective, and scalable context-aware keyword query interpretation system. The apex of this research framework includes the following three principal components:

- **Context-Aware Keyword Query Interpretation Using Inter-Query Context.**

Users often pose a sequence of multiple queries to fulfill an information need. Given an ambiguous query, what is needed for personalized search is to be able to capture a user’s “ambient search context,” possibly by considering the queries in the immediate context
of the current query. In other words, we can look at other older queries in the querying context to infer the intention of the current query. In this component, we incorporate query history as “inter-query context” to infer user intent. Major challenges that need to be addressed include how to represent query history as well as how heuristics based on such contextual information can be developed to bias the query interpretation process.

- **Disambiguating Keyword Queries Using Intra-Query Context.** A key part of mapping keywords to a structured query is identifying which sub-sequences of keywords are closely related, that is belong to the same mapping. We hypothesize that users write queries such that closely related keywords are clustered together as semantic units representing specific meanings. In other words, a keyword query is usually not a random permutation of words but consists of segments of related keywords. Consequently, the meaning of a keyword can be inferred from the “intra-query context” of neighboring keywords in the same segment. In this component, we tackle the problem of keyword query segmentation to identify keyword segments.

- **Scaling Concurrency in the Context-Aware Keyword Query Interpretation Systems.** In the existing context-aware keyword query interpretation techniques, a single-user, single-machine environment is assumed. However, those techniques do not scale well with an increasing number of concurrent users. A bottleneck of the current interpretation approaches is the graph exploration-based search strategy, which is impractical for multi-tenant scenarios because separate context-aware graph exploration states need to be maintained for different user queries. This leads to significant memory overhead in situations where there are large numbers of concurrent requests. This limitation could negatively impact the possibility of achieving the ultimate goal of personalizing the keyword search. In this component we propose a light-weight interpretation approach that employs indexing to improve throughput and concurrency with much less memory overhead. It is also more amenable to distributed and partitioned executions.
© Copyright 2014 by Haizhou Fu
All Rights Reserved
Efficient, Effective, and Scalable Personalized Keyword Query Interpretation for RDF Databases

by

Haizhou Fu

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science

Raleigh, North Carolina 2014

APPROVED BY:

__________________________  __________________________
Dr. Rada Chirkova             Dr. Munindar Singh

__________________________  __________________________
Dr. Nagiza Samatova           Dr. Kemafor Anyanwu
Chair of Advisory Committee
Haizhou Fu received his BS degree in School of Software from Tsinghua University in 2005. Beginning in 2006, Haizhou enrolled in North Carolina State University in pursuit of his Ph.D. degree. In 2009, he began working Prof. Kemafor Anyanwu on work in Semantic Web and database system.
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents and my wife Fan Yang for their love, persistent support, motivation, guidance and inspiration throughout all my endeavors.

This work would not have been possible without the scientific advice and consistent motivation of my supervisor and mentor Dr. Kemafor Anyanwu. I would like to thank her for the opportunities and the scientific guidance she gave me. Furthermore, I would like to thank my colleagues in the semantic computing research lab: Sidan Gao, Padmashree Ravindra and HeyongSik with whom I had many inspiring and fruitful scientific and philosophical discussions.

The work has partially supported by NSF grant IIS- 0915865.
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
</tr>
</tbody>
</table>

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vii</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Ambiguity in Keyword Query Interpretation</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Related Work</td>
<td>6</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Existing Solutions for The Keyword Search</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Existing Solutions for Personalized Search</td>
<td>12</td>
</tr>
<tr>
<td>1.3</td>
<td>Key Components</td>
<td>14</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Component 1: Context-Aware Keyword Query Interpretation Using Inter-Query Context</td>
<td>14</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Component 2: Disambiguating Keyword Query Using Intra-Query Context</td>
<td>17</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Component 3: Scaling Personalized Keyword Query Interpretation</td>
<td>19</td>
</tr>
</tbody>
</table>

## Chapter 2 Context-Sensitive Keyword Query Interpretation Using Inter-Query Context

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Foundations and Problem Definition</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>Representing Query History Using A Dynamic Cost Model</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Top-K Context-Aware Query Interpretation</td>
<td>35</td>
</tr>
<tr>
<td>2.4.1</td>
<td>CoaGe</td>
<td>35</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Efficient Selection of Computing Top-k Combinations of Cursors</td>
<td>38</td>
</tr>
<tr>
<td>2.5</td>
<td>Evaluation</td>
<td>41</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Effectiveness Evaluation</td>
<td>42</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Efficiency Evaluation</td>
<td>45</td>
</tr>
<tr>
<td>2.6</td>
<td>Demonstration</td>
<td>45</td>
</tr>
<tr>
<td>2.6.1</td>
<td>System Interface</td>
<td>46</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Demonstration Scenarios</td>
<td>46</td>
</tr>
<tr>
<td>2.7</td>
<td>Related Work</td>
<td>48</td>
</tr>
<tr>
<td>2.8</td>
<td>Conclusion</td>
<td>50</td>
</tr>
</tbody>
</table>

## Chapter 3 Disambiguating Keyword Query Using Intra-Query Context

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.2</td>
<td>Deep Segmentation</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm</td>
<td>59</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Computing Optimal Deep Segmentation</td>
<td>59</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Avoiding Graph Exploration (AGE)</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluation</td>
<td>65</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Evaluation of Effectiveness</td>
<td>66</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Evaluation of Efficiency</td>
<td>68</td>
</tr>
</tbody>
</table>
3.4.3 Evaluation of Scalability ................................. 68
3.5 Related Work ............................................. 69
3.6 Conclusion and Future Work .............................. 70

Chapter 4 Scaling Personalized Keyword Query Interpretation ............................. 73
4.1 Introduction .............................................. 73
4.2 Problem Definition ....................................... 76
4.3 Overview of Our Approach and System Architecture ................................. 77
4.3.1 SKI System Architecture .............................. 79
4.4 Keyword Query Interpretation in SKI ........................................ 81
4.4.1 Foundations ............................................ 81
4.4.2 Identifying Candidate Roots and Productive hits ................................ 82
4.4.3 GeFree: A Graph Exploration Free Approach .................................... 89
4.5 Evaluation .................................................. 92
4.5.1 Data, Query Scalability, and Performance ........................................ 93
4.5.2 Impact of Memory Consumption on Concurrency .............................. 97
4.5.3 Impact of the Number of Slave Nodes on the Throughput .................... 98
4.5.4 Efficiency of Candidate Root Identification ...................................... 100
4.5.5 Other Factors that Impact the Performance of Exploration-Free and Exploration-Based Query Interpretation Algorithms .......................... 102
4.6 Related Work .............................................. 103
4.7 Conclusion .................................................. 104

Chapter 5 Conclusion .......................................... 105

References ..................................................... 107
LIST OF TABLES

Table 2.1 Complete cost model .................................................. 34
Table 2.2 Scenario B: Different contexts different rankings ............. 48
Table 2.3 Scenario C: The impact of length of query history on interpretation .... 48

Table 4.1 Memory consumption and concurrency ............................. 97
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example of keyword query interpretation</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Taxonomy of research and the focus of this thesis</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>RDF schema and data graph</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>Graph summarization</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>Architecture and workflow</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Efficiency and effectiveness evaluation</td>
<td>42</td>
</tr>
<tr>
<td>2.5</td>
<td>Graphical user interface</td>
<td>47</td>
</tr>
<tr>
<td>3.1</td>
<td>Different orderings of keywords lead to different semantics</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Pruning of not user-intended candidate interpretations by taking advantage of segmentation</td>
<td>53</td>
</tr>
<tr>
<td>3.3</td>
<td>Pruning of more candidate interpretations by taking advantage of deep segmentation</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of RDF schema and data graph</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of shallow/deep segment and construction</td>
<td>58</td>
</tr>
<tr>
<td>3.6</td>
<td>Examples of the matrix used for dynamic programming</td>
<td>60</td>
</tr>
<tr>
<td>3.7</td>
<td>A running example</td>
<td>62</td>
</tr>
<tr>
<td>3.8</td>
<td>Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td>Architecture of SKI</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecture and workflow for a single slave node</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>Example</td>
<td>82</td>
</tr>
<tr>
<td>4.4</td>
<td>Group reachability bitmap index</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Question 2) How to identify nodes that can reach at least one node in all groups</td>
<td>86</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of path index</td>
<td>90</td>
</tr>
<tr>
<td>4.7</td>
<td>Impact of query complexity on performance</td>
<td>94</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance of interpretation algorithms for same set of keyword queries on datasets with different scale</td>
<td>95</td>
</tr>
<tr>
<td>4.9</td>
<td>Impact of MPL on latency</td>
<td>96</td>
</tr>
<tr>
<td>4.10</td>
<td>Throughtput</td>
<td>99</td>
</tr>
<tr>
<td>4.11</td>
<td>Efficiency evaluation: a) to f)</td>
<td>101</td>
</tr>
<tr>
<td>4.12</td>
<td>Efficiency evaluation: g) to h)</td>
<td>102</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The keyword search is a popular paradigm for querying unstructured, structured and semi-structured data. It allows information needs to be expressed in fairly simple terms – a set of keywords. In contrast, expressing information needs using a structured query language imposes a significant burden on users by requiring them to understand how to write queries in a formal language, which in turn requires intimate familiarity with the schema of a database. There is a growing trend toward publishing Web data that have been “tagged” with metadata. The result is that an increasing proportion of data available on the Web is structured, and hence much more “machine processable” or “machine understandable.” The idea of making data on the Web more machine understandable is central to the Semantic Web vision and has led to the development of standard metadata languages. The Resource Description Framework (RDF) [1] is a foundational language in the framework for the Semantic Web. RDF is a set of specifications for encoding knowledge about resources and the relationships between them in the Web, which can be expressed as a set of RDF triples. An RDF triple consists of three parts: subject, predicate, and object. A set of such triples is called an RDF graph, which includes labels on both nodes and edges. Currently, a number of Web resources and data sets in this format have seen a steady increase from two billion RDF triples in 2007 to 30 billion RDF triples in 2012, which have been widely adopted by governments, academic institutions, and medical
institutions and so on. This is likely to become the dominant resource of structured knowledge on the Web. While there is also a standardized, structured query language for querying RDF data, called SPARQL\cite{2}, the complexity of such languages necessitates the development of easy-to-use query paradigms, such as the keyword search.

Earlier approaches to keyword searches on relational databases \cite{3,5,41,23} considered the meanings of keywords as matches in the space of attribute values. This style is reminiscent of information retrieval (IR) paradigms where the meanings of keywords are also considered as matches in documents. However, this “meaning-as-match” style results in “shallow” query interpretations and therefore leads to poor quality results, which consist of a set of answers with heterogeneous semantics, so users have to sift through query results to identify most intended answers. For example, the IR-style search results of the keyword query “Jaguar Speed” would contain documents or entities related to “Jaguar” as animals or cars or aircrafts. Users have to check each item in the list of results to identify those results that match their intention. For example, the user only intends to find the “speed of jaguar as an animal.” The problem is more pronounced in RDF databases where data and metadata are queried simultaneously. In RDF databases, a keyword can play dual roles, that is as the label of an entity or a conceptual term. For instance, a keyword (e.g., “River”) could be interpreted as the attribute value of an entity, such as an address “765 River Bank Rd.” On the other hand, it can also be interpreted as a concept indicating “a large body of water,” where the results may not necessarily contain matches of keywords, such as “The Amazon” or “The Nile,” which will be missed in IR-style approaches.

Recent studies on keyword search on RDF databases \cite{82,28,68,67,32} tried to overcome the limitation of the “shallow” interpretation style and improve the quality of keyword search techniques. They introduce an explicit “interpretation” phase prior to the query answering phase to “structurize” a keyword query into structured queries, such as SPARQL queries, each of which has fixed semantics. The “structurization” procedure is usually achieved by computing the smallest subgraphs that connect keyword terms on an extended schema graph structure.
There is a natural mapping from subgraphs to structured query components. Consequently, the role of each keyword needs to be interpreted, and the entire query needs to be mapped to a set of implied conditional expressions, such as the WHERE clause and the return clause. For example, a keyword query “Jaguar Speed” can be mapped to the following SPARQL query:

```
SELECT ?x, ?y 
WHERE { ?x rdf:type Animal. 
    ?x name "Jaguar". 
    ?y avg_speed ?y }
```

where “Jaguar” is interpreted as animals, and “Speed” is interpreted as a property “avg-speed.” The entire structured query can be interpreted as “Return the average speed of jaguars as animals.” Keyword query interpretation is the mapping of a keyword query to a graph pattern query, which is equivalent to a structured query. The graph pattern or structured query represents an interpretation of the keyword query. We show an example of an interpretation in Figure 1.1. Given the schema and data graphs as shown in Figure 1.1, for the keyword query “jaguar speed,” many graph pattern queries can be constructed according to the structure of the schema graph shown in this figure. For example, for the graph pattern A) in Figure 1.1, keyword “jaguar” hits the range of the property “:model,” and “speed” hits the property “:top-speed” per se. Based on the schema graph, the missing information associated with the model “jaguar” and the property “speed” are instances of the type “:Car” and the exact top speed values. Therefore, the two variables “?car” and “?speed” serve as return variables that expect instances of cars and their corresponding top speed values to be returned. The equivalent SPARQL query and its results are also shown under the graph pattern in Figure 1.1. The English interpretation of the graph pattern and the SPARQL query is “Find the top speeds of the Jaguar cars.”
Figure 1.1: Example of keyword query interpretation
1.1 Ambiguity in Keyword Query Interpretation

Because keywords are ambiguous, usually there is more than one interpretation, that is, graph pattern of a keyword query. For example, “jaguar speed” can also be interpreted as “Find the animal jaguars and the average speed of them,” which can be represented by graph pattern B) in Figure 1.1, where “jaguar” hits the species property of the class node “:Animal” and “speed” hits the property “:avg_speed” as shown in this figure. It is important to note that in this thesis the concept of the “ambiguity” of a particular keyword token represents different mappings from the data graph element that matches the keyword token to the schema graph elements, that is, classes or properties. This level of ambiguity is entirely different from counting the occurrences of keyword matches in the data graph level. Hence, a keyword that occurs several times in the data graph is not necessarily ambiguous if all of them can only be mapped to one class in the schema graph. For instance, in Figure 1.1, if “jaguar” only matches “JaguarXJ” and “JaguarXK,” it is not ambiguous because both are “Cars.” “Jaguar” is ambiguous only because another match of this keyword is mapped to “Animal,” an entirely different class in the schema graph.

Because keywords are often ambiguous, there could be multiple interpretations of a single keyword, and therefore, the combinations of the keyword interpretations of all keywords in a keyword query usually lead to a huge number of candidate query interpretations. Existing techniques employ heuristics that rely on the structural and statistical characteristics of the database (e.g., frequency distributions of properties or concepts) to identify the top-K interpretations. Unfortunately, those techniques are based on static database-dependent properties and therefore cannot always capture the user intent. In some situations, users are not interested in popular concepts or resources that have the most support from the database. For example, in the keyword query “Jaguar Speed,” if in the database, “Jaguar” as cars occurs more frequently than it does as animals, it is highly likely that the entities related to cars will always be ranked higher. This is not desired by users who intend to retrieve the information about the animals. To determine more accurately a user’s intended interpretation of a query, it is impor-
tant to capture and use information about the user’s current querying interests. User profiles are usually investigated to capture user-dependent properties. However, user profiles represent long-term search interests [60] [35] [40] [30]. For short-term interests, we focus on other queries, such as those a user or her friends asked recently. We also examine the queries themselves to provide contextual cues. There are extensive studies on the keyword search, keyword query interpretation, and personalized search. The related work is reviewed in the next sub-section.

1.2 Related Work

The keyword search has been extensively studied in unstructured, structured (relational databases) [3] [5] [23] [73] [72] [38] [64] [81], and semi-structured (XML [37] [76] [75] [77] [21] [83], RDF [13] [67] [68] [14] [16] [15]) databases. As illustrated in Figure 1.2, in terms of methodology, research on the keyword search can be generally categorized as the following: i) “meaning-as-match” (where keywords are only considered matches of documents or attribute values) and ii) keyword query interpretation (where an interpretation phase is incorporated before query executing). Traditional approaches to keyword segmentation provide only partial interpretation techniques. Moreover, for complete interpretation, the existing techniques are based on intensive user interactions or fixed data-driven methods, which cannot always generate high quality results. Our work focuses on incorporating context in a heuristics-based approach to improve the quality of keyword query interpretation. We will elaborate each branch of work in Figure 1.2 as follows.

Shallow Interpretation (“Meaning-as-match”)

Earlier studies on the keyword search on structured databases, such as BANKS[3], DBXplorer[5], DISCOVER[23], and SPARK[41], as well as some recent approaches to graph structured databases like STAR[28] and BLINKS[20] focused on efficiently computing subgraphs that connect keyword matches in the database. Those approaches made the implicit assumption that keywords can only be mapped to attribute values that contain keyword matches. DBXplorer and DID-COVER returned joining networks of tuples based on either primary keys or foreign keys. BANKS modeled the database as graphs and answers as Steiner trees that connect all keyword
matches. SPARK proposed a new simple but effective ranking formula by adopting existing IR techniques. BLINKS[20] showed a novel bi-level index that improved search efficiency. Since the indexing strategy needs a large amount of storage, a partition technique is introduced to partition the whole graph into several blocks based on the portal node, which is a cut node or separator in the graph. The indexes are built on both higher level blocks and inner-level blocks, which makes their backward search faster.

**Complete Interpretation**

Some recent efforts, such as Keymantic[8], QUICK[79], SUITS[11], iSearch [78], Q2Semantics[68] and [67], focus on introducing an explicit keyword interpretation phase prior to answering
the query. The general approach is to find the “best” subgraphs (of the schema plus a data graph) connecting the given keywords and represent the intended meaning of the query. These approaches primarily use heuristics-guided graph exploration, in which the heuristics are designed to help converge on the “best” subgraphs as early as possible. Q2Semantic constructs a novel graph representation of the RDF data – RACK graph, which compresses the data to the schema graph such that the query can be done on the schema graph, which is relatively smaller. Q2Semantic applies a single-level search strategy (in contrast to the bi-level search strategy proposed by BLINKS[20]) to interpret keyword queries: to those hits of keywords scattered in the semantic graph. By expanding and detecting nodes reversely, their algorithms assign each newly detected node to the set which belongs to a keyword that can be reached by that node. The expansion will not stop until a node has been detected by every set. This node is returned as the root node and is the pivot for query construction. Q2Semantic adopts a ranking formula based on the popularities of classes or properties and the matching scores to distribute costs for nodes and edges. Thus, an optimal subgraph with the lowest cost can be returned. The most recent work on keyword query interpretation on RDF is [67], which improves the Q2Semantic. It applies an exploration algorithm very similar to backward search. During top-K computation, they borrow the threshold algorithm to find the top-K objects with the best scores, which is a combination of three factors, including path length, popularity, and matching scores. iSearch employs a similar backward search algorithm for all combinations of keywords, which is very expensive and focuses only on the effectiveness of the evaluation plan.

In addition, these approaches are based on database-dependent heuristics, such as the frequency distributions of node types and edge types in the data and/or the compactness or cohesiveness of the query structure. Variations on the above heuristics are presented in [24][66], which look beyond the mere structure and frequency distribution and try to predict the cohesiveness of relationships in schema based on correlation measures to capture the strength of relationships or their compactness within substructures. These techniques are based on fixed data-driven heuristics and do not adapt to varying user needs. Alternative approaches [11][79]
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.

**Partial Interpretation**

Other techniques [51][48][74][50] used to query relational databases offer some keyword query
interpretation but to a lesser degree (i.e., partial interpretation). For example, keyword segmen-
tation techniques identify segments of a keyword query. A valid segment should only contain
a “valid” database term. However, to be able to generate a structured query from a keyword
query, segments representing valid database terms can only be mapped to the conditional ex-
pressions of structured formal queries. Other aspects of query interpretation, such as identifying
which components are return expressions and how the different conditional expressions related
to one another are not addressed by these techniques. Nevertheless, these techniques do not
always identify the most relevant relationships between keywords, so the queries generated are
not often the user intended ones.

**Other Forms of Interpretation** Sawant et al. [56] proposed to bridging the gap be-
tween unstructured queries that do not have type information associated with keywords, and
the type of queries where type information is manually assigned to keywords by human be-
ings. To interpret queries with uncertain class/type information, they proposed a two-phase
approach, first to predict the entity-type mappings using a probabilistic model and then per-
form the type-restricted entity search. In our research we focus only on the latter form of
queries where type/class information is explicitly associated with database terms. Halder et al.
[19] proposed an approach to interpret the current query according to similar queries or “neigh-
borhood queries” using query transformation. Pound et al. [47] presented an approach that
interprets/translations keyword queries into structured representations over reference knowledge
base. They proposed to mine Web query logs in order to identify frequent query structures,
which are then mapped into the reference knowledge base. Probable semantic structures (de-
finned as semantically annotated queries) based on a statistical model learned from real user
queries are computed and mapped into the knowledge base. In contrast, instead of applying a probabilistic model, in this thesis we employ deterministic heuristics to infer user intended query structures. Gkorgkas et al. [17] proposed to interpret the intent of a keyword query after the answers are returned, which is entirely different from our approach because we interpret the intent before the query-answering phase. They extract features of keyword query results and classify the results based on similarities between querying results.

1.2.1 Existing Solutions for The Keyword Search

Keyword query interpretation takes place before query answering in the entire keyword query processing workflow. The general approaches used to generate top-K keyword query interpretations over a schema graph are similar to most existing keyword search techniques. Approaches supporting the keyword search can be categorized into schema-based approaches and graph-based approaches.

**Schema-based approaches.** These approaches answer keyword queries by utilizing schema information to construct structured queries, such as SQL queries and SPARQL queries, according to the keywords in a keyword query. Two key steps are usually used to answer a keyword query using schema-based approaches in relational databases. In the first step, all candidate networks (CN) [23] are enumerated. In the second step, the candidate networks are evaluated to return a set of minimal total joining network of tuples (MTJNT) [5][23]. However, because the number of MTJNTs is exponentially large, KRDBMS [52], DISCOVER-II [22], and SPARK [41][42] are approaches used to avoid enumerating all MTJNTs. However, the number of CNs increases exponentially with the size of the schema graph. Generating all CNs is inefficient even with proper pruning rules, especially in RDF databases with large schema graph. For more efficient query interpretation for RDF databases, CoSi [14][16] and Q2Semantics [68][67] introduce graph-based algorithms to identify top-K structured queries (i.e., SPARQL queries or conjunctive queries) progressively before answering the queries in RDF databases.

**Graph-based approaches.** Graph-based approaches to supporting the keyword search in
semi-structured or structured databases do not require the existence or assistance of a database schema. Those approaches aim at identifying subgraphs of the data graph that connects all keywords in a keyword query. According to the characteristics of the answers returned and the scoring functions used to rank the answers, the techniques can be grouped in three general categories:

I). Distinct Root Semantics (DRS). For DRS, an answer to a keyword query $Q$ is a sub-tree $T$ of the data graph $G_D$ rooted at $r \in V(G_D)$, and $T$ spans all keywords in $Q$. For any keyword node $v_k \in T$, there is an unique path between $v_k$ and $r$ in $T$, denoted by $<v_k,r>_T$, which is the shortest path between them in $G_D$. Hence, a root $r$ and a set of keyword nodes $\{v_{k1},v_{k2},...v_{km}\}$ uniquely identify an answer $T$ to a keyword query $Q = \{k_1,k_2,...k_m\}$. Therefore, given a set of keyword nodes, the search space of answer trees is $O(n)$, where $n = |V(G_D)|$; in other words, there are at most $n$ distinct roots and $n$ distinct answer trees. To answer queries with distinct root semantics, BLINKS [20] proposed a graph partitioning strategy and a shortest path based bi-level index to identify efficiently top-K answer trees with DRS. DPBF [12] proposed to the use of a best-first dynamic programming algorithm to find the optimal minimum Steiner tree. DPBF-k is a top-K algorithm of DPBF used to generate progressively approximate top-K Steiner trees rooted at $k$ different nodes.

II). Steiner Tree Semantics (STS). Compared with DRS, under STS, an answer to a keyword query is also a tree $T$ rooted at $r \in V(G_D)$, and $T$ spans all keywords in $Q$. However, the path between a keyword node $v_k$ and the root $r$ does not have to be shortest path. Therefore, given a root $r$ and a set of keyword nodes $\{v_{k1},v_{k2},...v_{km}\}$, the search space is $O(m\Delta^n)$, where $\Delta$ is the maximum degree of $G_D$, $n = |V(G_D)|$. Under STS the score of an answer $T$ is the total node/edge weights. Many approximate algorithms are proposed to find minimum Steiner trees. STAR [28] presents an efficient algorithm to identify top-K minimum Steiner trees. However, they did not consider the ambiguity of keywords. They assume that each keyword matches a single node in the graph. In the group Steiner tree problem where each keyword has multiple matches different types of graph-traversal algorithms are proposed. BANKS [3] applied
backward search to explore the graph from each keyword node. BANKS-II [27] improves upon BANKS by introducing a bidirectional expansion graph exploration algorithm.

III). Subgraph Semantics (SGS). Instead of the tree representation of answers, under SGS some approaches proposed returning subgraphs that span all keywords as answers. EASE [36] defined an \textit{r-radius Steiner graph} as a subgraph $G^r \subseteq G_D$ that spans all or a portion of keywords in $Q$ where the \textit{radius} of $G^r$ is exactly $r$, and radius $r$ is the \textit{centric distance} of the center of $G^r$, that is, the node with minimum centric distance. KRDBMS [52] proposed returning multi-centered subgraphs, in which each center and corresponding keyword nodes represent an answer under distinct root semantics. On the other hand, Ladwig et al. [33] argued that assuming a center of the answer graph restricts the search space. They introduce a \textit{d-length Steiner graph}.

In a \textit{d-length Steiner graph} $G^d$, any keyword node $v_k$ has at least one other keyword node $u_k \in V(G^d)$, such that the path between $v_k$ and $u_k$ in $G^d$ is of length $d$ or less.

1.2.2 Existing Solutions for Personalized Search

The same query made by different users may lead to entirely different query intentions. Personalized search systems aim at generating search results that match the current users’ search intention by using different strategies. These strategies can be categorized into four categories: i) Personalized search based on user behavior. Xiang et al. [69] proposed mining historic query logs for all users to identify those who perform similar Web search behavior, which can be used to identify further the current user’s search intent. In this research we are interested only in personalizing the search results for a particular user based on only his/her search behavior. Similar approaches that incorporate user behaviors, such as click-through data, particularly by users who perform similar tasks, include [4] and [25]. Park et al. [18] extended a user’s intention using a hierarchical phrase vector model (HPVM). Six through ten immediately clicked documents because explicit relevance feedback is used as information need. This kind of navigational information is used to determine a unique representation of a user’s intention. ii) Time-based personalization; [71] and Xu et al. [70] gathered the historic information of the dwell-times
derived from a user’s online reading or browsing activities. A mining algorithm is involved to re-rank the search results to achieve personalization. iii) Content-based personalization. Sopra et al. [10] took advantages of any available social information associated with Web content, such as tags, tweets, comments, rates, and they incorporated those social dimensions into the ranking function to personalize the search results. Shen et al. [57] proposed improving search quality by mining the long-term history of query logs. Query history has been utilized to dis-ambiguate and personalize keyword queries. However, in this thesis we propose to investigate short-term, user-specific query history. Compared with long-term history [61][44], short-term query history reflects the most recent search intentions. In addition, most query history-based personalization techniques are based on a query log containing all users’ search history. However, in this work, we focus on utilizing the most recent query history of each individual user. Lee et al. [34] conducted a series of behavior experiments to argue that mining users’ Web search logs is effective in predicting users’ search intentions. Lee classified a user’s search goal into two groups. In the first group, users have specific goals in mind, and in the second group, users are expected to learn from the search on a particular topic. The results of the experiment show that human participants can classify accurately about 90 percent of the queries. iv) Social context-based personalization. Mohamed et al. [9] proposed adding a social context layer to the textual content traditionally used for indexing. The additional social context layer and the social index are fed into a query expansion module to achieve personalization.

As alluded to above, studies have been conducted to understand the user’s intentions in making a personalized keyword search. None of them, however, apply those techniques to personalized Semantic keyword search, in particular, for Semantic keyword query interpretation. One of the reasons is that there are different challenges in answering keyword queries or in generating keyword query interpretations over (semi)structured databases. The challenges, research problems, and most importantly, key components will be illustrated in the following subsections.
1.3 Key Components

This research thesis consists of three components that improve the quality of keyword query interpretation in RDF databases by exploiting contextual information. There are two types of context: inter-query context and intra-query context. The inter-query context is represented by queries in the query history, and the intra-query context is represented by the latent substructures of a keyword query (i.e., keyword segments and in the last component, solutions concerning performance enhancement, including efficiency and scalability) are discussed. The three key components of utilizing contextual information to improve the quality of the interpretation of keyword queries will be introduced in the following subsections.

1.3.1 Component 1: Context-Aware Keyword Query Interpretation Using Inter-Query Context

Research Challenges and Problems Solved: We observe that users are prone to issues a sequence of related queries. In this component we propose to take advantage of the query history to infer user intentions. There are three key challenges in the problem of generating the most likely user-intended keyword query interpretations by exploiting the query history as inter-query context. The problem solved in this component is called the context-aware keyword query interpretation problem, which includes the following challenging subproblems:

- Efficient representation scheme for query history as inter-query context. Query history has been exploited in IR to help predict current queries (query completion) [65][7] or offer suggestions about other queries (query suggestions) [29][80] and to adjust the ranks of documents retrieved (result re-ranking) [63][58] based on text-similarity measures. The challenges of these problems differ from those addressed in this thesis. First, these tasks focus on reusing information (user clicks, additional keywords, or subsequent queries) from past “similar” searches. 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 textual patterns. Our problem requires unstructured queries, their intended interpretations (structured queries), and the ontology to be managed and exploited. Whereas in IR tasks, the query and answer models are unstructured and require less sophisticated computational machinery such as indexes. More importantly, we designed a representation model, which is a dynamic weighted graph model, to capture the evolving nature of the query history, and we developed a dynamic representation model.

- Develop heuristics to bias the selection of the most likely user intended candidate interpretations. Existing approaches employ graph exploration algorithms, such as bidirectional search [27] or backward search [3], to identify subgraphs that connect the keyword matches. However, the heuristics employed in these approaches were developed to guarantee the compactness of the subgraphs. In other words, they are only efforts to make sure the keyword matches are closely connected in the data graph. To capture user-dependent characteristics, which are represented by the representation model, we incorporated the query history into the database-dependent heuristics, such that both the relevance of a keyword query to the user’ query interpretation history and the compactness of subgraphs are considered as two important factors to guide the graph exploration process.

- Improve the efficiency of the graph exploration algorithms. The graph exploration algorithm is used to identify (top-K) minimum subgraphs that connect at least one keyword match for all keywords in a keyword query, which is equivalent to the group Steiner tree problem (GST). The GST problem is proved to be NP-complete[53]. The state-of-the-art graph exploration algorithms have some limitations. i) Some approaches [28] only solved the Steiner tree problem where each keyword is allowed to have only one match; ii) some approaches [20][12] required indices to be built based on a pre-computation of paths. This is not feasible for dynamic graph model, where the index needs to be rebuilt each time the graph weights are updated; iii) some approaches only return the top-1 minimum subgraph (or called GST-1 problem); iv) some approaches [68][82][67] did not provide solutions to
reduce the search space of all combinations of paths, which impedes the efficiency. To overcome these challenges, we developed an efficient algorithm to solve the GST-K problem, based on the dynamic representation model with maximum pruning of the search space.

**Overview of Technical Solutions**: This research introduces a context-aware summary graph model to represent the database and the query history. The context-aware summary graph model is a dynamic weighted and labeled graph that summarizes the class subsumption hierarchy and the data graph. The context-aware summary graph model is associated with a context-aware cost model that assigns weights to the nodes and edges in the summary graph. The weights represent the relevance of the concepts indicated by the nodes or edges in the summary graph to the query history. The weights will be updated each time a user issues a new query. We further propose a context-aware graph exploration algorithm, which is an algorithm used to solve the GST-K problem over the context-aware summary graph. Unlike techniques [20][12] that can be applied to only static graphs, our approach does not require the pre-computation of paths, and therefore can be adapted to dynamic environments. The heuristics employed by the proposed algorithm i) prefer small subgraphs (semantically closely associated), ii) prefer subgraphs with minimum weights (the most relevant to the query history); and iii) grant priority to keyword matches that are most relevant to the query history. We also introduce a new early stopping condition to the graph exploration algorithm to reduce significantly the search space and improve efficiency. Chapter 2 describes this research and provides a performance evaluation of this model.

**Major Contributions**: In addressing the challenges of solving the context-aware keyword query interpretation problem, we make the following contributions:

i) Introduce and formalize the problem of context-aware keyword query interpretation on RDF databases.

ii) Propose a model and implementation of a context-aware summary graph model that is used
to capture concisely the essential characteristics of a user’s query history.

iii) Develop an efficient and effective top-K context-aware graph exploration algorithm that extends the 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.

1.3.2 Component 2: Disambiguating Keyword Query Using Intra-Query Context

Research Challenges and Problems Solved: We hypothesize that users write queries such that closely related keywords are clustered as a keyword segment in a keyword query. The user’s intended meaning of a particular keyword can be inferred from the context of the neighboring keywords in a keyword segment that contains this keyword. The second issue that we address in this thesis is the problem of generating the top-K most likely user-intended interpretation of keyword queries by exploiting the intra-query context captured by keyword segmentation. Two challenges are tackled in this component:

- Identify the semantic units or keyword segments in a keyword query. In IR, the problem of keyword query segmentation to identify semantic units has been investigated. More recently in the context of relational databases, keyword segmentation has been introduced for solving query cleaning problems [48][50][74][49]. These approaches address the problem with using statistical models in which the optimal keyword query segmentation is the most probable sequence of database terms such as hidden Markov models and conditional random fields. This approach yields a segmentation into states, each of which is equivalent to a shallow segment that contains keywords in the same attribute value. However, the technique that identifies the interpretation of segments as database terms limits the potential to prune the search space. Moreover, a semantic unit representing the close
relationship of keywords in a keyword query and close semantic associations of keyword matches in the database cannot be limited to the keywords in the same database term. In contrast to such fine-grained segments, we identified coarser-grained keyword segments in the database to represent more general semantic units. We call this kind of segmentation “deep” segmentation.

- **Find optimal deep segmentations.** Because of the ambiguity of keyword queries, it is possible that there are many different segmentations for a given keyword query. Different segmentations correspond to different user information needs. It is necessary to develop a ranking scheme to evaluate the quality of a deep segmentation. In addition, the search space of deep segmentations is larger than traditional segmentations because fine-grained segments are a subset of the coarser-grained segments. To solve this problem, we propose an efficient algorithm to identify the optimal deep segmentations.

**Overview of Technical Solutions:** Our second approach exploits the intra-query context as keyword segmentation to identify users’ search intent. We propose and formalize the concept of deep segmentation. We develop an algorithm that extends traditional “shallow” segmentation techniques by aligning the sub-sequences of keywords of a keyword query to as many closely related structured query constructs as possible. This means that we used the structure of the database as evidence for validating segmentations. A scoring function is developed to assess the quality of deep segmentations based on their structural characteristics. We propose a dynamic programming-based algorithm to find the optimal deep segmentations.

**Major Contributions:** We propose a “deep segmentation” technique for identifying the coarse-grained segments of a keyword query that correspond to logical semantic units that are typically ignored by existing techniques. Specifically, this paper makes the following contributions:

i) We introduce and formalize the notion of the deep segmentation of a keyword query in RDF databases, which can better cope with the ambiguity of such queries than existing
techniques can. Furthermore, we formalize the problem of identifying the top-K interpretations of a keyword query by calculating its optimal deep segmentations using dynamic programming and a query-sensitive cost model.

ii) We present algorithms for computing deep segmentations and for top-K query interpretations that scale well for moderately sized schemas.

iii) We present extensive experimental results that show the effectiveness and performance of our approach in the context of significantly ambiguous queries, which previous studies have not considered.

1.3.3 Component 3: Scaling Personalized Keyword Query Interpretation

Research Challenges and Problems Solved: The previous discussion of research issues regarding enabling context awareness for keyword query interpretation are all based on the assumption of a single-user, single-machine search environment. However, in practice a search system should also provide scalable solutions to support a multiuser environment. The existing keyword query interpretation techniques are difficult to scale with the increasing number of users. The major reason is that keyword query interpretation techniques usually employ graph exploration-based algorithms, which suffer from high time and space complexity. The challenging research issues investigated, and resolved in this component include the following:

- **Eliminate the overhead of graph exploration.** For graph exploration-based algorithms, different keyword queries lead to different graph exploration search states. For each search instance, the large number of paths explored are maintained in the main memory, which incurs massive memory consumption and affects the scalability for supporting a large number of queries. It is even worse for achieving personalization of keyword query interpretations. Because different users have different weighted graphs with weights representing the query contexts for the same keyword query, the probing sequences of paths during graph exploration are also different for different users. Even for the same user
with the same keyword query, under different query contexts, the probing sequences of
tree can still be different. Some existing solutions have been proposed to pre-compute
graph exploration states to avoid graph exploration. However, their approaches are either
limited to much smaller search space requirements or cannot be applied to personalized
interpretation techniques that use dynamic weighted graphs. That is because the over-
head required by frequently updating the dynamic index is too costly. We tackled this
problem by investigating a new indexing scheme such that in runtime the expensive graph
exploration can be avoided.

• *Improve concurrency.* The high memory consumption of each instance of keyword query
interpretation is the bottleneck that impacts the concurrency. To maintain intermediate
graph exploration search states for each query for a particular user, a memory of around
500 Mb is required to store the information in the dynamic weighted graph and inter-
mediate graph exploration in a schema graph that includes around 1.3 million nodes and
edges. Hence, for a server with 8G RAM, a maximum of only 17 user query requests can
be processed concurrently. In order to allow more users to issue queries concurrently, we
designed data structures that separate sharable user-independent information from dy-
namic non-sharable user-dependent information and keep the non-sharable information
to a minimum.

• *Maximize throughput.* First, the efficiency of every interpretation instance must be re-
duced. However, the performance of the graph exploration-based algorithm drops expo-
nentially according to the increasing degree of the graph and the degree of ambiguity of
the keyword queries. With the growth of data, both the complexity of the schema graph
and the number of matches of any keyword increase, which negatively affects the efficiency
and scalability of graph exploration-based keyword query interpretation algorithms. We
designed an algorithm that can significantly reduce the complexity of the keyword query
interpretation algorithm in large graphs. Second, to allow maximal throughput, we pro-
pose a system that is amendable to both distributed and parallel executions.
Overview of Technical Solutions: In this component we design and implement a multi-tenant indexing scheme such that sharable graph exploration states for all users are pre-computed with reasonable memory consumption and non-sharable user-dependent information is kept to a minimum. The sharable graph exploration states include a set of pre-computed paths in the graph of lengths that are less than or equal to a specified threshold and a compressed bitmap group reachability index. The resulting interpretations can be generated by the fast identification of candidate roots and assembled using those pre-computed paths. The proposed interpretation algorithm eliminates the costly overhead required by graph exploration, particularly in achieving a personalized search, such that both the CPU time and memory usage are significantly reduced, therefore improving the concurrency in the multi-tenant environment over big data. To maximize the throughput, a distributed multi-tenant keyword query interpretation system is designed and implemented.

Major Contributions We propose an approach called “SKI” to meet the goal of scaling the concurrency of personalized keyword query interpretation over large scale data in multiuser environments. The SKI approach comprises the following:

i) A dual indexing scheme that captures user-specific information about concepts and relations that is the most relevant to a user’s current querying context and data-specific information about substructures in data. The former is captured in an index - personalized query context map (PCM) and the latter is captured in two key indexes, dense path index (DPI) - an index of subgraph structures and Rabbit - a group reachability-based index. The data-specific indexes inform the decisions made by a graph exploration-free interpretation algorithm (GeFree) about which substructures to prune and how to assemble substructures into complete interpretations. GeFree avoids the need for graph exploration, and it is fast and memory-efficient, thus reducing both latency and memory requirements of query interpretation.

ii) Personalized query interpretation is achieved by using the PCM index to retrieve selectively the highest ranked substructures from the DPI index with respect to the relevance
weights associated with a user’s entry in the PCM. This ensures that the assembled interpretation is the most relevant to a specific user.

iii) Comprehensive evaluation of the SKI approach in comparison with graph exploration-based interpretation approaches using the BTC and DBpedia datasets, which demonstrate the superior latency and memory requirements of the SKI approach and its ability to enable high concurrency and throughput.
Chapter 2

Context-Sensitive Keyword Query Interpretation Using Inter-Query Context

2.1 Introduction

The keyword search on structured data such as RDF offers the advantage of ease-of-use but presents challenges due to their often terse and ambiguous nature. Traditional approaches for answering keyword queries 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 “US universities” needs to be interested as a list of universities, many of which will not have both keywords in their labels and so will be missed by IR-style approaches. For such queries, the role of each keyword needs to be interpreted, and the entire query needs to be mapped to a set of implied 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
The goal of identifying the $K$ likeliest user intended interpretations.

Existing top-$K$ query interpretation approaches [67] employed 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. Typically, 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. Unfortunately, since such metrics are not user-dependent, the results generated do not always reflect the user intent.

In this work, 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 serial of related queries, particularly in exploratory scenarios. In these scenarios, information about previous queries can be used to inform the interpretation of a newer query. For example, in Figure 2.1, given a keyword query “Missis-

![Figure 2.1: RDF schema and data graph](image-url)
sippi River Bank,” 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 is more reasonable to interpret the current query as referring to a large body of water: the “Mississippi River.” Two main challenges that arise here include (i) effective capture and efficient representation of query history, and (ii) effective and efficient exploitation of query history during query interpretation. Towards addressing the challenges of solving the context-aware keyword query interpretation problem, we make the following contributions:

i) Introduce and formalize the problem of Context-Aware keyword query interpretation on RDF databases.

ii) Propose a model and implementation of a Context-Aware summary graph model that is used to concisely capture essential characteristics of a user’s query history.

iii) 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) It presents a Context-Aware graph exploration algorithm to compute a ranked list of K query interpretations.

v) we present a comprehensive evaluation of our approach using a subset of the DBPedia dataset.

2.2 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. $\lambda_D$ is a labeling function $\lambda_D : (V_D \cup E_D) \to 2^W$ 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 $v_l$. $\varphi_D$ is the incidence function: $\varphi_D : V_D \times V_D \to E_D$.

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. $\lambda_S$ is a labeling function $\lambda_S : V_S \cup E_S \to 2^W$ 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. $\varphi_S$ is an incidence function: $\varphi_S : V_S \times V_S \to E_S$. $\pi : V_S \to 2^{V_D}$ is a mapping function that captures the predefined property rdf:type mapping 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 $V_{LITERAL} \subseteq V_S$ 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) who 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 who 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 Figure 2.2, 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 a 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 Figure 2.2, we can find a structured query with conjunctive triple patterns listed at the top of Figure 2.2. Due to 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 Figure 2.2, “Organization” is the super class of “Bank.” Assuming that only “Bank” has the property “bName,” thus, the two pattern queries:

$$\langle ?x \text{ bName} \text{ “river”} \rangle, \langle ?x \text{ rdf:type Organization} \rangle,$$

$$\langle ?x \text{ bName} \text{ “river”} \rangle, \langle ?x \text{ rdf:type Bank} \rangle$$

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 2.2.1. (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 Figure 2.2, 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
Definition 2.2.2. (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 $G_S$.
- $V_{SG} = \{v_i | \exists u \in V_{\text{LEAF\_CLASS}} \cup V_{\text{LEAF\_LITERAL}}$ such that $\theta(v_i) = u^\wedge\}$.
For example, the context-aware summary graph in Figure 2.2 contains \{v_1, v_2, v_3, v_4\} 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.

- \( E_{SG} = \{ e_i \mid \exists u \in E_{LEAF,PROPERTY} \text{ such that } \theta(e_i) = u^\wedge \} \).

  For example, the summary graph in Figure 2.2 contains \{e_1, e_2, e_3\} 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.

- \( \lambda_{SG} \) is a labeling function: \( \lambda_{SG} : (V_{SG} \cup E_{SG}) \to 2^W \).

  - \( \forall v \in V_{SG} \text{ where } \theta(v) = v_C^\wedge \text{ and } v_C \in V_S \text{ representing class } C, \lambda_{SG}(v) = \{ \bigcup_{v_i \in v_C^\wedge} \lambda_S(v_i) \} \cup \{ \bigcup_{r_j \in \pi(v_C)} \lambda_D(r_j) \}; \)

    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 Figure 2.2,

      \[ \lambda_{SG}(v_4) = \{ "Mississippi," "North," "Carolina" \}; \]

      \[ \lambda_{SG}(v_3) = \{ "Thing," "Place," "State" \}. \]

  - \( \forall e \in E_{SG} \text{ where } \theta(e) = e_P^\wedge \text{ and } e_P \in E_S \text{ representing property } P, \)

    \[ \lambda_{SG}(e) = \bigcup_{e_i \in e_P^\wedge} \lambda_S(e_i), \]

    which is a union of all distinct tokens in the labels of all super classes of \( P \). For example,

      \[ \lambda_{SG}(e_2) = \{ "LocatedIn," "In" \}. \]

- \( \Psi_{SG} \) is the incidence function: \( \Psi_{SG} : V_{SG} \times V_{SG} \to E_{SG} \) such that if \( \theta(v_1) = v_{C_1}^\wedge, \theta(v_2) = v_{C_2}^\wedge, \theta(e) = e_P^\wedge, \text{ then } \Psi_{SG}(v_1, v_2) = e \) implies \( \varphi_S(v_{C_1}, v_{C_2}) = e_P. \)

- \( \omega : (QH, V_{SG} \cup E_{SG}) \to R \) 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 2.2.3.** (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 Figure 2.2, $HIT(\text{"bank"}) = \{v1, v2\}$.

**Definition 2.2.4.** (Keyword Query Interpretation): Given a keyword query $Q$ and a context-aware summary graph $SG$, a keyword query interpretation $QI$ is a connected subgraph of $SG$ that connects at least one hit of each keyword in $Q$.

For example, the summary graph shown in Figure 2.2 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 Figure 2.2. 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 2.2.5.** (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.) \( Q I_i \in TOPK \) and \( Q I_j \in ([|Q|] - TOPK) \), \( cost(Q I_i) \leq cost(Q I_j) \).

(ii.) If \( 1 \leq p < q \leq k \), \( cost(Q I_p) \leq cost(Q I_q) \), where \( Q I_p, Q I_q \in TOPK \).

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 Figure 4.3:

- Find keyword hits using an inverted index for a given keyword query \( Q \). (Step (1)-(3)).

- The query interpreter takes the hits and utilize 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 Figure 4.3 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.

The cost model will be discussed in the next section, and the graph exploration algorithm will be discussed in section 4.
2.3 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 our weighting function has to achieve, consider the following scenario. Assuming that we have the following sequence of queries $Q_1 = \text{"Ferrari, price"}$ and $Q_2 = \text{"F1, calender,"}$ with $Q_2$ as the most recent query. If $Q_1$ 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 $Q_2$ 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 $Q_2$, their relevance scores should be decreased. Then ultimately, for a new query $Q_3 = \text{"Jaguar, speed,"}$ 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: $Q_{I_t}$. *Region factor* is defined as a monotonically decreasing function of the graph distance $d(m, Q_{I_t})$ between $m$ and $Q_{I_t}$:

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

$(rf(d(m, Q_{I_t})) = 0$ if $d(m, Q_{I_t}) \geq \tau)$ $\tau$ is a constant value, and $d(m, Q_{I_t})$ is the shortest distance between $m$ and $Q_{I_t}$, i.e., among all the paths from $m$ to ANY graph element in the subgraph $Q_{I_t}$, $d(m, Q_{I_t})$ is the length of the shortest path. $\alpha > 1$ is a positive integer constant.

The region factor represents the relevance of $m$ to $Q_{I_t}$. *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 $Q_{I_t}$ as $hif(t) * rf(d(m, Q_{I_t}))$. 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 $\gamma$:

$$\gamma(m, T) = \gamma(m, T-1)/\beta + 1/\alpha^{d(m, Q_{I_T})}$$

(2.1)

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

$$\gamma(m, T) = \gamma(m, T-1)/\beta + 1/\alpha^{d(m, Q_{I_T})}$$

(2.2)

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 $\beta$ then adding $1/\alpha^{d(m, Q_{I_T})}$. In practice, we use $d(m, Q_{I_T}) < \tau = 2$, 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.

![Table 2.1: Complete cost model](image)

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 $\gamma$.
\[ \omega(m, t) = 1 + \frac{1}{\gamma(m, t)} \]  

This implies that a summary graph element with a higher relevance value will be assigned a lower weight. The complete cost model is shown in Table 2.1. In the next section we will discuss how to find interpretations with top-k minimal costs.

2.4 Top-K Context-Aware Query Interpretation

The state of the art technique for query interpretation uses cost-balanced graph exploration algorithms [67]. Our approach extends such an algorithm [67] 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 Algorithm 1.

2.4.1 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 \( \text{TOPK} \) is used to maintain top-K interpretations and a min binomial heap \( \text{CQ} \) 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 \( c(\text{keyword}, \text{path}, \text{cost}, \text{topN}) \), where \( c.\text{keyword} = w \); \( c.\text{path} \) contains a sequence of summary graph elements in the path from \( m_w \) to the node that \( c \) just visited; \( c.\text{cost} \) is the cost of the path, which is the sum of the weights of all summary graph elements in \( c.\text{path} \); \( c.\text{topN} \) is a boolean value that indicates whether \( m_w \) is among the top-N hits of \( \text{HIT}(w) \). The Top-N hit list contains the \( N \) minimum weighted hits of all hits in \( \text{HIT}(w) \).
Algorithm 1 CoaGe

1: Input: Initialize priority queues $TOPK, CQ$.
2: Create cursor for each hit of each keyword;
3: Insert each cursor to $CQ$;
4: while $CQ$ is not empty do
5: \hspace{0.5cm} \text{c} = CQ.ExtractMin(); \quad \text{//get cheapest cursor}
6: \hspace{0.5cm} \text{v} = \text{cursor.path}[0]; \quad \text{//the visiting node}
7: \hspace{0.5cm} \text{if} \ v \ \text{is a root then} \ TopKCombination(TOPK, v.CL)
8: \hspace{1cm} \text{if} \ TOPK.count \geq K \land TOPK.Max() < CQ.Min() \ \text{then}
9: \hspace{1.25cm} \text{TERMINATE;}
10: \hspace{0.5cm} \text{end if}
11: \hspace{0.5cm} \text{end if}
12: \hspace{0.5cm} \text{if} \ c.depth \ \text{is less than the threshold then}
13: \hspace{1cm} \text{for all neighbor} \ n \ \text{of} \ v \ \text{do}
14: \hspace{1.25cm} \text{if} \ n \ \text{is not visited by} \ c \ \text{then}
15: \hspace{1.75cm} \text{create new cursor} \ \text{new\_cursor} \ \text{for} \ n;
16: \hspace{1.75cm} \text{if} \ c.topN == FALSE \ \text{then}
17: \hspace{2.25cm} \text{c.cost} = \text{penalty\_factor;}
18: \hspace{1.75cm} \text{end if}
19: \hspace{1.75cm} n.CL[c.keyword].Add(new\_cur);
20: \hspace{1.25cm} \text{end if}
21: \hspace{1cm} \text{end for}
22: \hspace{0.5cm} \text{end if}
23: \hspace{0.5cm} \text{end while}
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 word, 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 subgraph $QI$. However, computing all combinations of cursors as the traditional approach [67] 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 7) 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 8). 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 12). New cursors are generated (line 15) for unvisited neighbors of $c$ (not in $c.path$, line 14). New cursors will be added to the cursor manager $CL$ of $v$ (line 19). 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 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 $c_A \cdot \text{cost} \times \text{penalty\_factor} > c_B \cdot \text{cost}$. The space complexity is bounded by $O(n \cdot d^D)$, where $\sum_{w_i \in Q} |\text{HIT}(w_i)|$ 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.

### 2.4.2 Efficient Selection of Computing Top-k Combinations of Cursors

```
Algorithm 2 Pseudocodes for TopKcombination algorithm
1: Initialize the combination enumerator $Enum = CLCEnum$.
2: Initialize the threshold list $TL$;
3: while $cur\_comb = Enum\_current() \land cur\_comb \neq NULL$ do
4:   if $\exists h \in TL, \text{Dominate}(cur\_comb, h) == TRUE$ then
5:     if $Enum.DIRECT\_Next(h) == NULL$ then
6:       break;
7:     else
8:       if $\text{DuplicateDetection}(cur\_comb, TOPK) True$ then
9:         if $Enum.Next() == NULL$ then
10:          break;
11:        else
12:          continue;
13:        end if
14:     else
15:       //not a duplicate combination
16:       if $TOPK.count \geq K \land TOPK.Max() < cur\_comb\_cost$ then
17:         TL.Add(cur\_comb);
18:         if $Enum.DIRECT\_Next(v) == NULL$ then break;
19:       end if
20:     else
21:       if $TOPK.count \geq K$ then
22:         TOPK.ExtractMin();
23:         TOPK.Insert(cur\_comb);
24:       end if
25:     end if
26:   end if
27: end while
```
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 algorithm 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 ≻ Com_q if for all 1 ≤ i ≤ L = |Q|, p_i ≥ q_i, and exists 1 ≤ j ≤ L, p_j > q_j. Because every list CL[w_i] ∈ CL is sorted in a non-decreasing order, i.e., for all 1 ≤ s ≤ L, i ≥ j implies that CL[w_s][i].cost ≥ CL[w_s][j].cost. Moreover, because the scoring function for calculating the cost of a combination Com is a monotonic function: cost(Com) = ∑_{c_i ∈ 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])

≻

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

implies that for all 1 ≤ i ≤ L,

CL[w_i][p_i].cost ≥ CL[w_i][q_i].cost and cost(Com_p) ≥ 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 Algorithm 2. 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 an 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
\( \text{Com}_0 = (CL[w_1][0], CL[w_2][0], ..., CL[w_L][0]) \),

which is the “cheapest” combination in \( CL \). Let

\( \text{Com}_{\text{last}} = (CL[w_1][l_1], CL[w_2][l_2], ..., CL[w_L][l_L]) \),

be the last combination, which is the most “expensive” combination and \( l_i = CL[w_i].\text{length} - 1 \), 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

\( \text{Com}_{\text{current}} = (CL[w_1][s_1], CL[w_2][s_2], ..., CL[w_L][s_L]) \),

from 1 to \( L \), \( \text{Enum.Next()} \) locates the first index \( i \), where \( 1 \leq i \leq L \) such that \( s_i \leq l_i \), and returns the next combination as \( \text{Com}_{\text{next}} = (CL[w_1][0], ..., CL[w_{i-1}][0], CL[w_i][s_i + 1], ..., CL[w_L][s_L]) \),

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 \( CL[w_1].\text{length} \) equals to 10 and \( CL[w_2].\text{length} > 5 \), then, the next combination is \( (CL[w_1][0], CL[w_2][6]) \). The enumerator will terminate when \( \text{Com}_{\text{current}} == \text{Com}_{\text{last}} \).

Each time \( \text{Enum} \) move to a new combination \( \text{cur}_{\text{comb}} \), it is compared with every combination in \( TL \) to check if there exists a threshold combination \( h \in TL \) such that \( \text{cur}_{\text{comb}} \succ h \) (line 4). If so, instead of moving to the next combination using \( \text{Next()} \), \( \text{Enum.DirectNext()} \) 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

\( \text{Com}_{\text{threshold}} = (CL[w_1][s_1], CL[w_2][s_2], ..., CL[w_L][s_L]) \),

from 1 to \( L \), \( \text{Enum.DirectNext()} \) 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 \( \text{Com}_{\text{direct.next}} = (CL[w_1][0], ..., CL[w_i][0], ..., CL[w_{j-1}][0], CL[w_j][s_j + 1], ..., CL[w_L][s_L]) \)

where for all \( i \leq r < j \), \( s_r \) is changed to 0, and \( s_j = s_j + 1 \). For example, for
\( \text{com}_{\text{threshold}} = (CL[w_1][0], CL[w_2][6], CL[w_3][9], CL[w_4][2]), \)

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

\( \text{com}_{\text{direct, next}} = (CL[w_1][0], CL[w_2][0], CL[w_3][0], CL[w_5][3]). \)

In this way, some combinations that could be enumerated by “Next()” function and will dominate \( \text{com}_{\text{threshold}} \) will be ignored. For instance, \( \text{com}_{\text{next}} = \text{Next}(\text{com}_{\text{threshold}}) = (CL[w_1][1], CL[w_2][6], CL[w_3][9], CL[w_4][2]), \)

and the next combination after this one: \( \text{Next}(\text{com}_{\text{next}}) = (CL[w_1][2], CL[w_2][6], CL[w_3][9], CL[w_4][2]) \)

will all be ignored because they dominate \( \text{com}_{\text{current}}. \)

If a new combination is “cheaper” than the max combination in \( \text{TOPK} \), it will be inserted to it (line 24), otherwise, this new combination will be considered a new threshold combination, and inserted to \( TL \) (line 18) such that all the other combinations that dominate this threshold combination will not be enumerated. The time complexity of \( \text{TopKCombination} \) is \( O(K^k) \), where \( K = |\text{TOPK}| \) is the size of \( \text{TOPK} \), \( k = |Q| \) is the number keywords. Because, for any combination

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

\( \text{com}_K = (CL[w_1][K + 1], ..., CL[w_L][K + 1]) \succ \text{com} \)

In the worst case, any combinations that dominates \( \text{com}_K \) will be ignored and \( K^k \) combinations are enumerated. Consequently, the time complexity of \( \text{CoaGe} \) 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 [67] (we call this approach \( \text{TKQ2S} \)) is \( O(n \cdot d^D \cdot S^{D-1}) \), where \( S = |SG| \) is the number of nodes in the graph.

\[ \text{2.5 Evaluation} \]

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
Figure 2.4: Efficiency and effectiveness evaluation 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.

2.5.1 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. (Sample queries, questionnaires, experimental results and the statistics of the user feedback can be found from the following link: http://research.csc.ncsu.edu/coul/experiments.rar.)

**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, and $rel_i$ 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 research the relevance between a resultant interpretation $QI$ and a desired interpretation $QID$ for a given keyword query $Q$ cannot be simply characterized as either a match or not. $QI$ and $QID$ are both subgraphs of the summary graph and could have some degree of overlapping, which means $rel_i \in [0,1]$. Of course, if $QI == QID$, $QI$ should be a perfect match. In this experiment, we define the relevance between a candidate interpretation $QI$ and the desired interpretation $QID$ as: $rel_i = \frac{|QI \cup QID| - |QI \cap QID|}{|QI \cup QID|}$, 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 two subgraphs. 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:

$$P@K = \frac{|\text{relevant interpretations in TOPK}|}{|\text{TOPK}|},$$

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 Figure 2.4 (a) illustrates the quality of interpreting queries with varying DoA values. DCG is used to evaluate the quality. From (a), we can observe that TKQ2S does not generate good quality interpretations for queries with high DoA. That is because they prefer the popular concepts they 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. Figure 2.4 (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. Figure 2.4 (c) illustrates how different lengths of query history will affects results. In this experiment, 4 groups of queries are given, the $i$th 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 log. 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.

2.5.2 Efficiency Evaluation

From the result of the efficiency evaluation in Figure 2.4 (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 $DoA$ is high. The performance gain is due to the reduced search space enabled by early termination.

2.6 Demonstration

We developed and demonstrated a keyword query interpretation system called CoSi (COntext-Sensitive Keyword Query Interpretation) to demonstrate the improvement of the quality of the keyword query interpretation task by using query history as context information. The system is demonstrated using a very large real-world dataset DBPedia [6]. CoSi is a desktop application that will be run locally using a laptop or desktop PC. The end users can interact with several features of the system.
2.6.1 System Interface

Figure 2.5 shows the graphical user interface. By default, CoSi enables a context-sensitive interpretation mode. Users can also manually change the mode by clicking the “Search Mode” button. The users can start a new session by choosing “New Query Session” on the menu. By clicking the “New Search” button, user can start issuing keyword queries and browse the results. After user click the “Interpret” button, the system will generate a ranked list of up to five candidate interpretations. By selecting one of the candidate interpretations, the corresponding SPARQL query will be shown in the text box under the “Interpret” button. Users can also view any interpretation from the list and a subgraph that is equivalent to the interpretation of the keyword query will be shown on a panel in the main interface. Any information related to issued queries such as keywords, candidate interpretations (subgraphs and SPARQL queries) will be recorded. User can select to view the information of any query in the query history by clicking that query in the list shown on the top-right carousel panel control. The entire history of query interpretations can be saved by using the “Save Query Session” function on the main menu. CoSi also provides users a “weights viewer” to see the subgraph of the weighted summary graph. The size of the red circle in the node reflects the value of the weight. The subgraph contains nodes and edges related to all the queries in the query history. The users can observe the changes of subgraphs caused by the evolving context. Finally, the end users can choose to investigate the results of using context-agnostic mode, where the traditional approach is applied.

2.6.2 Demonstration Scenarios

The results of the demonstration of the CoSi system show some interesting features. Three interesting user cases are given:

**Scenario A:** The end users can choose to issue a sequence of related keyword queries under different search modes (context-sensitive and context-agnostic). In this way, they can observe how queries in recent query history influence the interpretation of newer queries.
Scenario B: Scenario B shown in Table 2.2 illustrates a more interesting feature of CoSi. Given a keyword query “Apple Manhattan,” assume that two possible interpretations are $I - A$ and $I - B$. Given two different query histories (The end users can load existing query logs by clicking “open query session” from the main menu), CoSi ranks $I - A$ higher by loading “Query History 1” because the conceptual scope of this search session is about “artist” and “album.” Alike, CoSi ranks $I - B$ higher if “Query History 2” is loaded because the context is all about “company” and “products.” Therefore, CoSi can detect the context which reflects the users’ focus and interests during the specific query session, while traditional methods (using context-agnostic mode) will always return the same ranking regardless of the context.

Scenario C: Another important capability of CoSi is demonstrated by scenario C shown in Table 2.3. The CoSi system is capable of learning the knowledge from query logs. The longer the query history the more information can be digested and utilized by the CoSi system, and therefore, the disambiguation process will be more accurate. In this scenario three query logs are given, and as is shown in Table 2.3, each log contains one more keyword query than the previous one. By looking at the target query “Rose,” there is no idea what the users want to ask, which is a usual case that the users themselves sometimes do not know how to issue a query for what they really want. Therefore, they may issue a sequence of queries for a clue.
Table 2.2: Scenario B: Different contexts different rankings

<table>
<thead>
<tr>
<th>Conceptual Scope of the Context:</th>
<th>Query History 1</th>
<th>Query History 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1: Tigal</td>
<td></td>
<td>Mac OS</td>
</tr>
<tr>
<td>Query 2: When the Pawn</td>
<td></td>
<td>Iphone App Store</td>
</tr>
<tr>
<td>Artist, Album</td>
<td></td>
<td>Company, Products</td>
</tr>
<tr>
<td>Apple Manhattan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-A: Singer named Fiona **Apple** who was born in **Manhattan**

I-B: Headquarters of **Apple Inc.** in **Manhattan**

<table>
<thead>
<tr>
<th>Query History 1</th>
<th>Query History 2</th>
<th>Query History 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1: Tool</td>
<td>Tool</td>
<td>Tool</td>
</tr>
<tr>
<td>Query 2: Model Database</td>
<td>Model Database</td>
<td></td>
</tr>
<tr>
<td>Query 3:</td>
<td></td>
<td>Entity Relationship</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Most Recent Query:</th>
<th>Ranking of Intended Interpretation</th>
<th>Intended Interpretation:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&gt;5</td>
<td>A software <strong>Rational Rose</strong> for ER diagramming</td>
</tr>
</tbody>
</table>

Table 2.3: Scenario C: The impact of length of query history on interpretation

But traditional interpretation systems are not helpful because they are not context-sensitive. Users still need to sift through the search results. But with CoSi, when people search in an exploratory way by issuing serial queries, CoSi will learn what they are really asking for and rank the intended interpretation higher such that the end users can find them more easily.

### 2.7 Related Work

There is a large body of work supporting the keyword search for relational databases [3][5][23] based on the interpretation as “match” paradigm. However, recent work [68][67] focused on providing meaningful results for keyword queries on databases. Some other approaches only
address the query interpretation problem partially. STAR [28] proposed an efficient approximate Minimum Steiner Tree algorithm to compute top-K optimal connecting trees. However, the problem addressed in this problem is more related to the Group Steiner Tree problem, which can be addressed using that technique. BLINKS [20], proposed a novel bi-level indexing strategy based on graph partitioning that improves the efficiency of graph exploration. However, its indexing is targeted at a statically weighted graph which to be adapted for our purposes will require re-indexing after every query.

The proposed approach is a context-aware interpretation engine for complete structurization of keyword query. Query logs have been exploited in IR to help predict current queries (query completion) [65][7] or offer suggestions about other queries (query suggestions) [29][80][39] and to adjust the ranks of documents received (result ranking) [63][58] based on which documents were selected by users when similar queries were issued in the past. However, traditional solutions for context-sensitive information retrieval consist of various n-gram language models, which are probabilistic models of text to mine the query logs. The problems tackled have different challenges from the ones addressed in this work. First, those tasks focus on reusing information (user clicks, additional keywords or subsequent queries) from past “similar” searches. But none of the approaches explore the concept similarity. Structured query interpretation problem requires concepts as variables to be returned by the SELECT clause in a formal query language like SPARQL. Therefore to identify the similarity between two concepts, we are not going to measure the textual similarity, instead, the paths in the schema graph between two concepts. Plus, traditional query logs contain only a set of keywords in past queries or query results history. However our query logs indirectly store the interpretation history which is a series of graph structured queries. The last but not the least, the traditional approaches do not consider the impact of ages of the past queries on the current query.
2.8 Conclusion

This research presents a novel and effective approach for interpreting keyword queries on RDF databases by integrating the querying context. We propose a novel approach to represent implicitly the context of query history. We also propose an efficient group Steiner tree (GST) based algorithm for fast identifying structural relationships among keywords.
Chapter 3

Disambiguating Keyword Query Using Intra-Query Context

3.1 Introduction

In Chapter 1, we discussed the general approach of keyword query interpretation used by current techniques, which are attempts to find the “best” subgraphs (of the schema plus data graph) that connect a set of keywords given in the query. These top subgraphs are supposed to represent the likeliest intended interpretations which can then be translated to structured queries in a straightforward way. The graph exploration algorithms for computing the subgraphs rely on fixed data-driven heuristic measures that allow converge on the “best” subgraphs early. The approach introduced in Chapter 2 provides a solution based on query history to overcome the limitation, where traditional approaches cannot always generate the most user-intended interpretations.

However, both the traditional approaches and proposed approaches in Chapter 2 make an assumption that a keyword query is represented by a “bag-of-words” model. We posit that there is often some partial structures that exist in keyword queries that can be exploited to capture the user intent. This viewpoint suggests that a keyword query is not a random permutation of
words (as in the “bag-of-words” view), but rather a set of logical word groupings. Each group represents a set of terms that are intended to have close associations in the knowledge base, we call each such group a semantic unit or keyword segment. For example, in Figure 3.1, given a set of words “Mississippi,” “River,” and “Bank,” the query “River Bank Mississippi” are more likely intended to mean “River banks located in Mississippi state” according to the knowledge base. On the other hand, the query “Bank Mississippi River” using the same set of words has a more likely intended meaning as, “Name of a bank as Mississippi river.” We can see that the difference between the earlier query, and the latter one is that in the earlier query, the intension of the concept of River Bank is reflected in the query, which places them close to each other in the query and also close in the knowledge base (representing a class name). In the other query, however, “Mississippi River” is more closely related in both query and the knowledge base. Consequently, it may be possible to more accurately capture the intended semantic units in a query using a “set of keyword sub-sequences,” where the keyword sub-sequence represents any associations suggested by the user query and the knowledge base.

Further, this “set of keyword sub-sequences” model will lead to a more efficient keyword query interpretation by reducing the possibility of search space explosion. Because the combinatorial explosion will be in terms of the number of keyword segments, which is better than in terms of the individual words. For example, in Figure 3.2, given a keyword query “Bob, Spring, 2007, Database, System,” “Database System” is grouped as a semantic unit representing a course named “Database System,” There, other candidate interpretation for “Database” such as “title of a publication” and other interpretation for “System” such as “keywords of a
Figure 3.2: Pruning of not user-intended candidate interpretations by taking advantage of segmentation

*paper* can be pruned, because they are not as closely related as the interpretation representing the course name. Consequently, interpretations based on database terms that contain those keywords further apart can be discarded or ranked lower.

However, although, such segmentation techniques can help improve the quality of interpretations by reducing the search space to some extent, the impact of such approaches is limited because the idea of traditional segmentation [51][48][74][50] is primarily applied at the level of database terms. In other words, segments can only consist of database literals. Consequently, how these database terms are related, and therefore, how their interpretations are to be combined remain a problem when keyword query is ambiguous. To achieve the full advantage of the segmentation technique, we propose a “*deep segmentation*” approach that exploits the keyword structure to its fullest extent. Here, rather than segmenting just into database terms, we allow segments to consist of closely related database terms (e.g. attribute values for the same entity)
Figure 3.3: Pruning of more candidate interpretations by taking advantage of deep segmentation resulting in coarser semantic units. Note that this deals with the problem of combining smaller subqueries (segments of database terms) into larger ones and achieves a multiplicative effect in terms of reducing the interpretation search space. For example in Figure 3.3, recall the example in Figure 3.2, for the same keyword query “Bob, Spring, 2007, Database, System,” a coarse-grained segment “Spring, 2007, Database, System” representing “a database course offered in spring 2007 semester” is identified, the four keywords are both close in keyword query and close in the knowledge base. Therefore, other candidate interpretations for “Spring” or “an address of a person,” “2007” as “year of a book” etc., are pruned.

In this work, we propose a “deep segmentation” technique for identifying coarse grained segments of a keyword query that correspond to logical semantic units typically ignored by existing techniques. Specifically, this research makes the following contributions:

i) We introduce and formalize the notion of deep segmentation of a keyword query over
RDF databases, which can better cope with the ambiguity of such queries than existing techniques. Further, we formalize the top-K keyword query interpretation problem that generates the K most likely intended interpretations of a query that relies on its deep segmentation and a query-sensitive cost model.

ii) We present algorithms for computing deep segmentations and top-K query interpretations that scale well for moderately sized schemas.

iii) We present extensive experimental results that show the effectiveness and performance of our approach in the context of significantly ambiguous queries, a consideration not made by previous efforts.

3.2 Deep Segmentation

The problem that we address in this work is how to generate the top-K list of “interpretations” or “structurizations” that represent the most likely meanings intended by the user. At a high level, the problem can be viewed as consisting of two parts: the first step is the identification of the basic components of the query i.e., the graph elements (edges / nodes) immediately relevant to the keywords in the query. The second step is the identification of the intended relationships necessary for connecting the basic graph elements i.e. the relevant connected subgraph. We begin the formalization of the problem with some basic definitions. The definition of RDF schema and data graph $G_S$ and $G_D$, the summary graph $SG$, a graph element $A$, keyword query $Q$, hit of a keyword $w$, i.e., $HIT(w)$ and keyword query interpretations of $Q$, i.e., $[[Q]] = \{QI_1, \ldots, QI_l\}$ are already defined in Chapter 1.

**Definition 3.2.1. (Hit Array)** Given a keyword query $Q = (w_1, w_2, \ldots, w_m)$, a hit array $H$ of $Q$ is an ordered sequence $H = (A_{w1}, A_{w2} \ldots A_{wm})$ of hits for each keyword $w_i$ such that for all $i$, $H(i) = A_{wi} \in HIT(w_i)$ i.e., the relative position of a keyword in a query is the same of that of its hit in a hit array. Since it is possible to have multiple hits for each keyword $k_i$ in the database i.e., $|HIT(w_i)| > 1$, multiple hit arrays are possible for a query $Q$.
Each hit array contains a different combination of hits for the keywords in \( Q \) and can be seen as a possible “interpretation” of \( Q \). For example, for a keyword query “database system” on the database model in Figure 3.4 <LN “Genetic entity... database”, LN “File system...” > is one possible hit array, and <CN “database”, LN “Database system...” > is another one, where LN indicates a literal node and CN indicates a class node. When several of the keywords in the query have multiple hits, the potential number of candidate interpretations becomes high due to the combinatorial explosion and reflects to a certain extent how ambiguous a query is.

**Definition 3.2.2. (Segment):** Given a summary graph \( SG \), a sub-query \( Q' \) of a keyword query \( Q \), and a hit array \( H \) of \( Q' \), a segment is a rooted subgraph \( T^r_H \) of \( SG \) connecting all hits in \( H \) with \( r \) as the root of \( T^r_H \). For example, in the schema graph in Figure 3.4, \( T^A_{\text{course}, \text{database}, \text{spring}, \text{2007}} \) is a segment rooted at the node \( A_{\text{course}} \) with hits for keywords “database, spring, 2007.” A hit \( A \) is trivial segment e.g. \( T^A_{\{A\}} = A \), the root of which is also the hit itself. A segment \( T^r_H \) must further satisfy the following restrictions either it is

1. a trivial segment or it is composition \( T^r_H = T^r_{H1} \sqcup T^r_{H2} \) of two segments \( T^r_{H1} \) and \( T^r_{H2} \) such that \( T^r_H = T^r_{H1} \sqcup T^r_{H2} = T^r_{H1} \cup T^r_{H2} \cup P1 \cup P2 \)
ii.) the concatenation of $H_1$ and $H_2$, i.e. $H_{12} = (H_1|H_2)$, must be a sub-sequence of some hit array $H$ of $K$;

iii.) $P_1$ and $P_2$ are the shortest paths between $r$ and $r_1$ and between $r$ and $r_2$ respectively that satisfy: $|P_1| \leq \pi$ and $|P_2| \leq \pi$, for some constant $\pi$.

**Definition 3.2.3. (Shallow Segment):** When $\tau = 0$, any valid segment is called a shallow segment, i.e. the segment contains only keywords which appear in the same database term. As mentioned earlier, this is equivalent to the traditional segmentation problem.

**Definition 3.2.4. (Deep Segment):** When $\tau = 1$, any valid segment is called a deep segment. More generally, a deep segment is a trivial segment or a segment constructed from two deep segments such that the distance between the two roots of the two child deep segments is less than or equal to $2\tau$ i.e. 2. A deep segment $T^r_H$ is a maximal segment if there does not exist a deep segment $T^r_{H|H'}$ such that $T^r_{H|H'} = T^r_H \uplus T^r_{H'}$, for any valid deep segment $T^r_{H'}$.

Figure 3.5 gives examples of shallow and deep segments and how they are constructed.

**Definition 3.2.5. (Deep Segmentation):** Given a keyword query $Q$ and the summary graph $SG$, a deep segmentation $\Pi^\pi_H = (T^r_{H1}, T^r_{H2}, \ldots T^r_{Hn})$ is an ordered sequence of maximal segments such that $H = (H1|H2|\ldots|Hn)$ and $Hi$ is a hit array for a subquery $w_i$ of $Q$. For example: given a keyword query $K = (Bob, database, system, spring, 2007)$, and the data model in Figure 3.4, a deep segmentation is

$$\Pi^\pi_H = (T^A_{Bob}, T^A_{database}, T^A_{system}, T^A_{spring}, T^A_{2007})$$

where

$\pi: (Bob)||(database, system)||(spring, 2007)$

$H: (A_{Bob})||(A_{database}, A_{system})||(A_{spring}, A_{2007})$

**Definition 3.2.6. (Score):** Given a segment $T^r_H$, its score $s$ can be defined as:

$$s(T^r_H) = \begin{cases} 
0 & \text{if } T^r_H \text{ is a trivial segment and } |H| = 1 \\
 s(T^r_{H1}) + s(T^r_{H2}) + SR(r_1, r_2), & \text{if } T^r_H = T^r_{H1} \uplus T^r_{H2} 
\end{cases}$$
Figure 3.5: Example of shallow/deep segment and construction
where $SR(r_1, r_2) = \tau - |path[r_1][r_2]|/2$.

Intuitively, the closer the two roots are the higher the score is. For example, to be a deep segment, we have $\tau = 2$, $SR(A_{spring}, A_{2007}) = 2 - 2/2 = 1$, if $A_{spring}, A_{2007}$ are both leaf nodes for the class node $A_{semester}$ in the schema graph shown in Figure 3.4. Moreover, the score of a segmentation is the sum of the scores of all its maximal segments: $s(\Pi_H) = \sum s(T_{H_i}^R)$.

**Definition 3.2.7.** (Optimal Deep Segmentation): Given the set of segmentations for a keyword query, an optimal deep segmentation is a deep segmentation with the highest score among all possible deep segmentations. The problem addressed here focuses on finding the top-K optimal deep segmentations.

### 3.3 Algorithm

In this section, we present a dynamic programming (DP)-based algorithm for computing optimal deep segmentation. Further, we propose an algorithm to avoid graph exploration (AGE) for capturing compact associations among segments. Before we present the algorithm, we first describe the index employed for identifying keyword matches on the schema graph. As shown in Figure 3, given an RDF schema and data graphs, a standard Trie is adopted to index all the database literals in both the data and the schema graphs. Each node (the key of this Trie node is a token w) in the Trie contains a list of pointers pointing to a set of corresponding schema elements $\{A_i\}$ such that $HIT(w) = \{A_i\}$.

#### 3.3.1 Computing Optimal Deep Segmentation

Optimal Deep Segmentation can be solved using a DP-based algorithm. According to Definition 3.2.4 and Definition 3.2.6, deep segments and their scores can be constructed and computed in a recursive way. Therefore, given a sequence of keywords $Q = \langle w_i \ldots w_j \rangle$, the optimal segmentation $\Pi$ over $Q$ should satisfy:

i.) $s(\Pi) = \max_{i < m < j} [s(\Pi_{H1}) + s(\Pi_{H2}) + score(\Pi_{H1}, \Pi_{H2})]$
ii.) $\mathcal{H}_1 = (A_{w_1}, \ldots, A_{w_m})$; $\mathcal{H}_2 = (A_{w_{m+1}}, \ldots, A_{w_j})$ are hit arrays for keyword sub queries $<w_1, \ldots, w_m> \subseteq Q$ and $<w_{m+1}, \ldots, w_j> \subseteq Q$ respectively

iii.) $\text{score}(\Pi_{\mathcal{H}_1}, \Pi_{\mathcal{H}_2}) = \begin{cases} s(T_{\mathcal{H}_1} \uplus T_{\mathcal{H}_2}); & \text{if } \Pi_{\mathcal{H}_1} = \{T_{\mathcal{H}_1}^1\} \land \Pi_{\mathcal{H}_2} = \{T_{\mathcal{H}_2}^2\} \\ 0; & \text{otherwise} \end{cases}$

This scoring function indicates that if $\Pi$ contains only one segment that is a composition of other two deep segments, then extra score which is $s(T_{\mathcal{H}_1} \uplus T_{\mathcal{H}_2})$ should be added, otherwise, only the sum of the scores of two deep segmentations $\Pi_{\mathcal{H}_1}$ and $\Pi_{\mathcal{H}_2}$ is taken into consideration. The rationale behind this scoring function is that we prefer to build larger deep segments.

The pseudo codes in Algorithm 3 present the algorithm. The algorithm employs a matrix $M$ to compute optimal deep segmentation. As shown in Figure 3.6, the algorithm starts with examining entries in the first layer which is the entries in the diagonal. After that, new roots and scores for new deep segments will be detected and stored in entries in higher levels. Best scores and back tracing information are recorded so that at the highest level which contains
only one entry $M[N][N]$, the roots of deep segments in an optimal deep segmentation can be returned. Figure 3.6 also shows an example of identifying the best solution for the entry $\mathcal{H}$, where four different sub-routines should be analyzed.

\begin{algorithm}
\caption{Optimal deep segmentation}
\begin{algorithmic}[1]
\State \textbf{for} each level $\text{lev} \in [1,N]$ \textbf{do}
\For{$i \in [0,N-\text{lev}]$} //consider the units at the $i$th level
\State $m = M[i][i+\text{lev}]$; //target entry
\For{$j \in [0,\text{lev}]$} //iterate all possible sub-routines
\State $L = M[i][i+j]$;
\State $R = M[i+j+1][i+\text{lev}]$;
\State $s = L.s + R.s$;
\ForAll{node $r1 \in L.roots$}
\ForAll{node $r2 \in R.roots$}
\If{$\exists r$ such that $|\text{path}(r,r1)| \leq 1 \land |\text{path}(r,r2)| \leq 1$}
\State $s_k = SC(r1,r2)$
\If{$s + s_k > m.s$}
\State $m.s = s + s_k$;
\State $m.roots = \{r\}$;
\If{$s$ is the maximum score}
\State $M.left = L; M.right = R$; //Store backtracing information
\EndIf
\EndIf
\EndFor
\EndFor
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

Figure 3.7 gives a running example of this algorithm. The schema graph is shown in the first rectangle with nodes labeled from $A$ to $R$, which is the same schema graph shown in Figure 3.4. The first matrix in the example is an initialized matrix, where the entries of the diagonal contain lists of hits of some keywords. Assuming that there are four keywords “Bob, Database, Spring, 2007,” According to the data in Figure 3.4, “Bob” hits node $G$, “Database” hits nodes $D,A,L,P$, “Spring” hits $I,M,C,H,P$, “2007” hits $N$ and $Q$. At the end, the last
matrix is returned and by backtracking algorithm, we can return the roots \{F\}, where F is the root of segment \(T_{MNG}^F\), the corresponding optimal deep segmentation is then \(\Pi = (T_{MNG}^F)\), the English interpretation of which is “the faculty Bob who teaches a database class in semester spring 2007.”

The worst case time complexity of this algorithm is \(O(N^3 \cdot DOTA)\). This is because the time complexity of the standard DP algorithm is \(O(N^3)\), and in the worst case, all combinations of hits can be candidate conceptual components, which means the time complexity of enumerating all of them is \(DOTA\). It should be noted that the complexity of generating a graph pattern from a hit array \(H\) is bounded by \(|H||G|\), where \(|G|\) is the scope of graph exploration required (in the worst case it is equal to the size of the graph) and \(|H|\) is the size of the hit array. Intuitively, in the worst case, each cursor corresponds to each hit in the hit array explores the entire graph each time before a connecting graph is found. The overall complexity of the whole problem is \(O(|G| (\sum_{i=0}^{n} |Hi|))\), where \(n\) is the number of all possible hit arrays that are identified, or called.
the size of the search space. Therefore, two main factors are critical to the efficiency: $|H_i|$ and $n$. So, we introduce query segmentation to group hits in the hit array such that the actual input to the query exploration algorithm is the number of deep segments $= m < n$. This reduces the complexity to $O(|G| (\sum_{j=0}^{n-m} |H_j| + m))$. Because there are $n-m$ keywords that do not belong to any deep segments, we still need to consider all their hits. But for the $m$ segments, each segment has only one root. Further, a segmentation with fewer segments will also decrease the search space $n$. Considering an extreme situation, when no deep segmentations are detected, for a given keyword query, we need to enumerate all combinations of hit arrays, and the search space is $n = DOTA$. But if we can group some hits while pruning other candidate hits that are not grouped, the search space could be reduced.

3.3.2 Avoiding Graph Exploration (AGE)

After deep segmentation, the next step is to identify the associations among those segments. The root of a deep segment is used as hit for keywords covered by that segment. For example, if a deep segment $T$ covers keywords $w1$ and $w2$, without deep segmentation, all hits in $HIT(w1)$ and $HIT(w2)$ should be considered to find the final answer graph connecting $w1$ and $w2$ and other keywords. But with deep segmentation, only the root is considered as the only hit for $w1$ and $w2$. In this way, other hits can be pruned which leads to smaller search space. Usually a graph exploration algorithm such as bidirectional expansion is applied to find top-K connecting subgraphs. However, for GE algorithm like [67], the worse case time complexity is $O(|G||number of cursors|)$ because in the worst case, before the top one connecting node is found, every cursor could have explored the entire graph. Deep segmentation is an attempt to reduce the number of cursors by grouping and locking some semantic units. We propose an algorithm that can avoid graph exploration, and the time complexity is $\Theta(|G| (K^{1/n} n))$, where $K$ is the number of top-K queries generated, and $n$ is the number of keywords. In practice, on average, $n$ is usually from 2 to 3, and for query generation, we do not have to generate too many candidate queries, so $K$ is usually a small number. Graph exploration is a solution.
to identify close association among graph elements, but if the graph is small enough, we are able to pre-compute the association information and store it in the index. By doing so, we can construct an approximate Steiner tree connecting all matches in the graph in constant time without exploring the graph. AGE is such an algorithm that takes advantages of the facts that the schema graph is always a small graph, and it is feasible to store all shortest path for pair wise schema nodes in the index, and $K(\text{top } K)$ and $n$ (number of keywords) are usually small number.

\begin{algorithm}
\textbf{Algorithm 4} AGE Top-K query generation
\begin{algorithmic}[1]
\STATE for each class node $C$ in the schema graph do
\STATE \hspace{1em} for each keyword $k_i$ do
\STATE \hspace{2em} get ordered hit list $H_i = \text{trie.search}(k_i, C)$;\label{alg:hit_list}
\STATE \hspace{2em} get top $m$ hits from $H_i$, denoted by $X_j$;\label{alg:top_m}
\STATE \hspace{2em} for each matching $M \in \{X_j\}$ do
\STATE \hspace{3em} for each hit $h \in M$ do
\STATE \hspace{4em} $\text{sub-query} + = \text{Path}(C, h)$;\label{alg:sub_query}
\STATE \hspace{3em} end for
\STATE \hspace{2em} end for
\STATE \hspace{1em} if $|\text{sub-query}| \leq |\text{kth-query}|$ then
\STATE \hspace{2em} $\text{Top-K-Query}.\text{remove}(k)$;\label{alg:remove}
\STATE \hspace{2em} $\text{Top-K-Query}.\text{insert}(\text{sub-query})$;\label{alg:insert}
\STATE \hspace{1em} end if
\STATE \hspace{1em} end for
\STATE end for
\end{algorithmic}
\end{algorithm}

Further, we need an extended Trie data structure for top-K query generation. Assume that there are $n$ class nodes in the schema : $< C_1, C_2 \ldots C_n >$, then, for each token, there are $n$ different lists that correspond to $n$ class nodes. Every list contains the same set of hits for that token, but in different order. For example, $H = \text{Trie.Search}(\text{token}, C_2)$, $H$ is an ordered list of four hits of the token: $< h_1, h_2, h_3, h_4 >$. The ordering of these hits is based on the distance from $C_2$. So, $\text{Dist}(C_2, h_1) \geq \text{Dist}(C_2, h_2) \geq \text{Dist}(C_2, h_3) \geq \text{Dist}(C_2, h_4)$. By storing the distances and orderings information, we are able to find the top-K nearest hits of a specific token from $C_2$, we only need to pick the first $m$, $m = \lceil \frac{K}{n} \rceil^n$ hits from each hit list of each
token. For example, if $K = 4$, $n = 3$ (3 keywords, top-4 queries), we only need to pick first 2 hits for each keyword, and there are $2^3 = 8$ possible candidates, for each candidate, we can look up the path table to calculate the size of the connecting tree and find the top-4 queries from all of these 8 candidates. The pseudo codes is shown Algorithm 4.

### 3.4 Evaluation

In this section, we will discuss the experiments we have performed to evaluate the effectiveness and efficiency of our approach. (Sample queries for questionnaires and experimental results including effectiveness, scalability and efficiency evaluations can be found from the following link: [http://research.csc.ncsu.edu/coul/mosaic_new.htm](http://research.csc.ncsu.edu/coul/mosaic_new.htm)). To assess the effectiveness, we extract part of the schema and data from YAGO [59] database, which is a real life dataset that contains more than 2M entities such as persons, organizations, cities, etc. We extracted 1.5M triples for 9 classes and 20 properties. To evaluate the efficiency, we also use a synthetic graphs dataset which is a set of artificial schema graphs with user specified number of class nodes (for scalability) and random number of leaves.

**Platform.** The experiments are conducted on a machine with Intel duel core 1.86GHz and 3GB memory running on Windows Seven Professional.

**Sample queries.** For effectiveness, a set of 10 queries with different DOTA values were crafted along with 4 candidate interpretations based on the literals in the dataset and their different occurrence contexts.

**Human subjects.** 25 students randomly selected on a college campus were asked to select the likeliest intended interpretation from the candidate interpretations given for each query, and a consensus interpretation was chosen by the user for each query.

**Metrics.** We compared the *Weighted Aggregated Precision (WAP)* and *Weighted Aggregated Recall (WAR)* of our Deep Segmentation technique against that of the variant of the Q2Semantic [68] described in [67] (denoted as $TKQ2S$), where $WAP$ and $WAR$ are defined as: given the top-$K$ structured keyword query $\{Q_i| i = 1...K\}$, and the desired query interpretation
\[ QI_D, WAP = \frac{\sum_{i=0}^{K} \text{precision}(QI_i) \cdot \text{top}(i)}{\sum_{i=0}^{K} \text{top}(i)}; \quad WAR = \frac{\sum_{i=0}^{K} \text{recall}(QI_i) \cdot \text{top}(i)}{\sum_{i=0}^{K} \text{top}(i)}, \]

where \( \text{precision}(QI_i) = |QI_i \cap QI_D| / |QI_D| \) and \( \text{recall}(QI_i) = |QI_i \cap QI_D| / |QI_i| \), and \( \text{top}(i) \) is a fixed constant assigned for position \( i \) if the consensus interpretation is placed at position \( i \) by the query generation algorithm. In our experiments, we considered only the top 4 queries returned by the algorithms, and the weights are \( \text{top}(i) = 10, 5, 2, 1 \) for \( i = 1, 2, 3, 4 \). The intuition behind the metric is that the algorithm that ranks an answer with better quality higher is a better algorithm. For example, if the precision of the output of the technique utilizing deep segmentation is: top-1: 1.0, top-2 0.5, while the output for the algorithm without deep segmentation is like this: top-1: 0.5, top-2: 1.0. Though both of them include the best answer with 100% precision generated, but, the algorithm with deep segmentation ranks it higher, which leads to higher \( WAP \).

### 3.4.1 Evaluation of Effectiveness

In the Figure 3.8, we use “DS” to indicate the interpretation algorithm using deep segmentation technique. “Nor-Seg” represents the one with traditional segmentation (only shallow segments are detected). \( TKQ2S \) is the interpretation technique without any segmentation detection procedures. From charts a) and b), the queries generated by \( DS \) have the highest \( WAP/R \) in most of the cases. There are a few cases when the deep segmentation did not yield the consensus answer e.g. for \( Q7 \). Specifically, the \( WAR \) of \( Q7 \) for \( DS \) is lower than \( TKQ2S \). This indicates that the aggressive pruning of candidate interpretations eliminated a desired interpretation. However, its precision for the same query is much better than the other approaches. This indicates that although the pruning is aggressive, the retained segmentations are good quality ones. Also, it still outperformed the traditional segmentation for recall for that query. Moreover, we can observe that the algorithm uses normal segmentation, which only considers keywords in the same database term as segment, the quality is between \( DS \) and \( TKQ2S \). That is because, sometimes when only deep segmentation can be detected, which means no shallow segments can be found or there does not exist adjacent keywords in the same database term, then the
Figure 3.8: Evaluation
Nor−Seg performs as bad as TKQ2S; but while there are only shallow segments, which means no deep segments can be detected, then DS will generate a segmentation exactly the same as what Nor−Seg will generate, then Nor−Seg performs as well as DS.

3.4.2 Evaluation of Efficiency

Efficiency evaluations were performed on both real and synthetic graphs. On YAGO dataset, three different experiments were performed (charts c, e and f). The first experiment shown in chart c) demonstrates the extra time the algorithm needs to compute deep segmentation to trade off for improvement of the quality. Keyword queries with different DOTA values are used as input, the y-axis indicates how much percentage of increase is there. The experimental result shows that in most of the cases, with a little increase of extra time, the quality can be significantly increased (up to 600%). Therefore it is worthwhile to spend a reasonable more time for guarantee of better effectiveness.

On the other hand, our efficiency evaluation also tested the performance of the two algorithms using YAGO data set. Different test cases are provided (time against the number of keywords, and time against how many top-K queries are generated, and DOTA=100). The experimental results shown in charts e) and f) demonstrate that AGE algorithm is more efficient for the given YAGO dataset.

3.4.3 Evaluation of Scalability

To evaluate scalability, on synthetic data, as shown in chart d) we compared the time consumed for the 5 algorithms. The result shows that when DOTA=1000, the AGE-based algorithm (with or without DS) performs better than the GE-based algorithm when the schema size is less than 75, then, the AGE-based algorithm is not so efficient to deal with schema that is large enough. But it is reasonable to sacrifice a little bit more time for DS but have quality guarantee as illustrated in chart c). From the experiment, although AGE does not scale as well as GE, but when the schema is small or in a reasonable range, where in practice, most existing schemas
are, AGE is still a good choice.

3.5 Related Work

In the area of computational linguistics, the problem of word sense disambiguation (WSD) has been widely studied [45][26][55][31] and also focused on trying to identify the intended meanings/senses of words. Disambiguating keyword queries applying the idea of keyword segmentation is a similar task that they all try to identify concepts consisting of more than one consecutive words in the input sequence. However, NLP approaches cannot be applied directly to query segmentation problem because keywords are an abstraction of natural language. Furthermore, the WSD problem focuses on interpreting groups of words that are complete in some grammatical sense which can be parsed and its components mapped to grammatical structures. On the other hand, keyword queries often do not contain complete grammatical structures. In addition, we require that the output of the interpretation of a keyword query be a structurization that can be mapped easily to a structured formal query.

Research on keyword query segmentation can be categorized in two types of work: the keyword search in the Web context [62][54] and in the context of relational databases [51][48][74][50]. Because of the fundamental differences between unstructured data in the Web context and the structured data, the approaches are different and cannot be applied to each other. [62] proposed an unsupervised approach to calculate query segmentation using a generative model with its parameters estimated by an expectation-maximization algorithm. That model requires a partial corpus, or a probability distribution $P_c$ be given before parameter estimation so that the mutual information can be calculated automatically from textual data. They use a dynamic programming algorithm to avoid segmentation enumeration. However, because of all approaches for segmentation problem in the realm of computational linguistics or information extraction rely hardly on the corpus, for a query segmentation problem on structured database like relational databases, a meaningful segment should have all its words appeared in a specific database term while a probabilistic model relies on corpus cannot guarantee that.
On the other hand, for the problem of keyword query segmentation that has been investigated for relational database, [74] proposed a segmentation on keyword query based on a technique called conditional random field. It is said to outperform the segmentation approach using hidden Markov model proposed in [48]. Because the primary advantage of CRF over HMM is their conditional nature, resulting in the relaxation of the independence assumptions required by HMMs in order to ensure tractable inference. Additionally, CRF can avoid the label bias problem, a weakness exhibited by maximum entropy Markov models (MEMMs) and other conditional Markov models based on directed graphical models. CRF outperforms both MEMMs and HMMs on a number of real-world sequence labeling tasks. The segment they defined is also limited to database terms, and both statistical model based approaches need a training data, which is the query logs that containing the correct labels of the query. CRF is used only as a scoring function. The next step is looking for an optimal segmentation among all possible segmentations. They first calculated all “maxterms” using a $O(n)$ algorithm, where $n$ is the number of terms in the query), then the optimal segmentation for each such invalid segment is computed through a tree search procedure. [51], [48], and [74] are all extended versions of [50]. However, query segmentation over semi-structured data, like RDF, adds more challenges. An RDF data model is a highly connected graph model, which emphasize connectivity and relationship between entities. A desired segment is no longer restricted to database terms but more complex structures, which leads to the “Deep” segmentation problem.

3.6 Conclusion and Future Work

We present a technique called “Deep Segmentation” to disambiguate the interpretation of a keyword query and capture the user intent by exploiting the intra-query context as keyword segmentation. Deep segmentation is based on “coarse-grained” semantic units i.e., keyword segments in contrast to traditional keyword segmentation approaches where “fine-grained” semantic units are used. This approach achieves a more aggressive pruning of irrelevant interpretations from the space of interpretations considered and, therefore, produces better quality
query interpretations even in the presence of significant query ambiguity. Experimental results on both synthetic and real life datasets show the superiority of our approach over existing approaches. Utilizing query history as discussed in Chapter 2 and intra-query context as elaborated in this Chapter achieves promising results separately. But it is possible to further extend the two components by combining them with each other:

**Context-Aware Keyword Query Segmentation**

Due to the large search space of candidate deep segmentations, the dynamic programming algorithm proposed in this work is still very expensive. To be able to prune irrelevant deep segmentations more aggressively without losing the optimal segmentation, we propose to incorporate the cost model introduced in Chapter 2 to assign weights to the summary graph. In this way, the a segments based on the context-aware summary graph is a weighted tree. We propose a new scoring function based on the original scoring function in Definition 3.2.6 that evaluates the quality of segments:

\[
s'(T_r^H, t) = \alpha \cdot s(T_r^H) + (1 - \alpha) \cdot \sum_{m \in H} \omega(m, t),
\]

where \( \alpha < 1 \) is a constant, \( T_r^H \) is a segment with hit array \( H \) at root \( r \), \( t \) is the historical index of, \( \omega(m, t) \) is the weight of the graph element \( m \). This composite scoring function ensures that two segments with same scores using original scoring function are likely to have different new scores according to the proposed context-aware scoring function, and the one with lower score will be pruned.

**Exploiting the Integration of Inter-query Context and Inter-query Context**

The experimental results shown in Chapter 1 and Chapter 2 shows significant improvements of the quality of keyword query interpretation respectively. It is a very interesting problem of integrating intra-query context with inter-query context to further improve the interpretation effectiveness. The segmentation of a keyword query contains segments, each of which is just a partial interpretation of a keyword query. The structural relationship between keyword segments still needs to be identified. We propose to extend the current approach to adapt to context-aware summary graph, where, for different query history, associations among segments
should be different. We propose to compare the effectiveness of the combined approach with the approaches with intra-query context and the inter-query context alone.
Chapter 4

Scaling Personalized Keyword Query Interpretation

4.1 Introduction

In the previous chapters, context-awareness has been thoroughly studied for keyword query interpretations. Most of the keyword query interpretation techniques including the previous work in this dissertation proposed for structurizing a keyword query involve superimposing keywords over a summarized graph representation of data and schema, and then exploring the summarized graph to compute connected subgraphs of keyword hits. While these techniques have shown reasonable performance in the context of a single-user architecture, work addressing scalability challenges of these approaches in a multi-user search system is lacking. Scalability challenges here can be viewed from two perspectives: in the context of big Semantic Web data e.g., “Billion Triple Challenge datasets (BTC 2009)” \(^1\), the number of hits for each keyword and the density of the summary graph increases. For example, the keyword “conference” has 19 occurrences in DBPedia 3.6 \(^2\) including 272 class nodes and around 3000 properties, but 1785 hits in BTC 2009 whose schema graph contains 150232 class nodes and around 1.3 mil-

---

\(^1\)http://km.aifb.kit.edu/projects/btc-2009/
\(^2\)http://blog.dbpedia.org/2011/01/17/dbpedia-36-released/
lion edges. This results in a significantly increased exploration search space and consequently an increased latency of the interpretation process. In addition, memory requirements for the exploration-based interpretation techniques increases sharply for larger, denser graphs because of the increase in number of possible paths in the graph that must be maintained in memory as potential partial results. Given that multi-user environments require maintaining search states for each users’ queries, exploration-based approaches are unlikely to be able to support high levels of concurrent users which is characteristic of real-world search workloads. As a concrete example, our experiments show that the amount of memory required to maintain the summary graph for a single interpretation process over the BTC datasets is around 500MB memory. Therefore, a server with 8GB RAM will only be able to support around 17 concurrent user queries! The combination of increased latency and reduced concurrency in big data environments makes exploration-based techniques impractical for achieving reasonable throughput in multi-tenant Semantic Web search systems.

**Contributions and Scope.** In this chapter, we focus on strengthening the advantage of “interpretive” search by scaling it to large scale data and multi-user environments. Specifically, we aim to realize increased throughput by increasing concurrency and reducing latency of the interpretation process. We posit that integrating our proposed scalable query interpretation approach with many ongoing-efforts on scalable query-answering techniques, will result in scalable Semantic Web search architectures. Our work is motivated by the goal of personalizing search on the Semantic Web, which is achieved by personalizing the interpretation process to identify the specific query intent for a user. In some previous work [14], we presented a context-aware interpretation model that represents a user’s query history in the summary graph as dynamic weights. The weights on edges and nodes of the graph are updated with each query, thus reflecting the increased on decreased relevance of that concept node or relationship to the current querying context and is used to bias the graph exploration i.e. interpretation process. However, even with superior performance in quality and performance over existing interpretation approaches, being a graph exploration-based interpretation process, it is still plagued
with concurrency limitations highlighted earlier. Here, we propose an approach called “SKI” that overcomes these challenges to meet the goal of scaling concurrency of personalized keyword query interpretation over large scale data in multi-user environments. The SKI approach comprises

i). a dual indexing scheme that captures user-specific information about concepts and relations most relevant in a user’s current querying context and data-specific information about substructures in data. The former is captured in an index - *personalized query context map* (PCM) and the latter in two key indexes, *dense path index* (DPI) - an index of subgraph structures and *Rabit* - a group reachability-based index. The data-specific indexes inform a graph exploration-free interpretation algorithm’s (*GeFree*) decisions about which substructures to prune and how to assemble substructures into complete interpretations. *GeFree* avoids the need for graph exploration and is fast and memory-efficient reducing both latency and memory requirements of query interpretation.

ii). Personalized query interpretation is achieved by using the PCM index to selectively retrieve the highest ranked substructures from the DPI index with respect to relevance weights associated with a user’s entry in the PCM. This ensures that the assembled interpretation is most relevant to a specific user.

iii). Comprehensive evaluation of the SKI approach in comparison with graph exploration-based interpretation approaches using the BTC and DBpedia datasets demonstrate the superior latency and memory requirements of the SKI approach, and its ability to enable high concurrency and throughput.

The rest of this chapter is organized as follows, we define the problem in Section 4.2, in Section 4.3, we overview the approach and architecture for the keyword query interpretation system; in Section 4.4, we present detailed discussions on the indexes and algorithms for query interpretation; Section 4.5 presents experimental results. We discuss the related work in Section 4.6 and conclude in Section 5.
4.2 Problem Definition

Let $\Sigma$ be a universe of words. An RDF schema graph is a labeled graph $G_S = (V_S, E_S, \lambda_S)$ where the nodes represent classes and edges represent properties. $\lambda_S$ is a labeling function that maps a graph element (node or edge) to $l \subseteq \Sigma$ that contains words that make up the label of the class or property. An RDF data graph is also a labeled graph $G_D = (V_D, E_D, \lambda_D, \pi_D)$ that has $V_D$, $E_D$ and $\lambda_D$ are similarly defined with $\lambda_D$ mapping a graph element to set of words contained in the string literal associated with that element. $\pi_D$ maps each data graph element (node or edge) $g_D$ to a set of schema graph elements: those to which $g_D$ is connected by a “rdf:type” relation.

Given a schema graph $G_S$ and data graph $G_D$, an annotated schema graph $G_A$ is a tuple $(G_S, \lambda_A)$ where $\lambda_A$ maps each graph element $g_S$ such that $\lambda_A(g_S) = \{ \lambda_S(g_S) \cup \lambda_D(g_D) | g_S \in \pi_D(g_D) \}$. In other words, the function $\lambda_A$ lifts labels on elements of data graph to the schema elements that they are instances of, thereby annotating the schema graph with keywords from the data graph.

A keyword query $Q$ is a set of words $\{k_1, k_2, \ldots, k_{|Q|}\}$ from the universe $\Sigma$. An element $g_i$ of an annotated schema graph $G_A$ is called a hit for keyword $k$ if $k \in \lambda_A(g_i)$. The set of all hits for a keyword $k$ is denoted as $\text{hit}(k)$. An interpretation $[[Q]]$ of a keyword query $Q$ is a subtree of $G_A$, that contains one hit for each keyword in $Q$. Since there are potentially multiple interpretations of a keyword query (due to different possible combinations of hits for all keywords), we focus on a specific (top) interpretation for $Q$ which we denote $[[Q]]^*$, but ignore for now, how to choose $[[Q]]^*$ from the set of alternatives for $Q$.

**Definition 4.2.1.** A context-aware annotated graph or CAG for short, associates weights with an annotated schema graph based on the sequence of queries that have been posed on that graph. A sequence of queries $Q_1, \ldots, Q_m$ produces a sequence of weighted graphs $G_0 \rightarrow G_1 \ldots \rightarrow G_m$ where $G_t \ (0 \leq t \leq m)$ is the weighted annotated schema graph produced after $Q_t$. $G_t$ is defined as $(G_A, \omega_t)$ where $\omega_t \in W$, $W$ is a family of weighting functions s.t:
• \( \omega_0 \in W \) is an init. function that assigns weights to all elements of the annotated schema graph prior to \( Q_1 \);

• \( \omega_t \in W \) is defined s.t. for \( g \in [[Q_t]]^* \), i.e. in the top interpretation for \( Q_t \), \( \omega_t(g) = \theta(\omega_{t-1}(g)) \).

In other words, after \( Q_t \), the weights of a graph elements in the top interpretation of \( Q_t \), is a function of their weights after \( Q_{t-1} \). This leads to the definition of an annotated schema graph as a dynamically weighted graph whose elements’ weights change with queries.

As mentioned earlier, the details of how to select \( [[Q]]^* \) – the top interpretation for a keyword query \( Q \), are discussed in Chapter 2.

The problem we address in this chapter is supporting personalized CAGs for a large number of concurrent users, i.e. for two users \( X \) and \( Y \) with query sequences \( QS_X \) and \( QS_Y \) resp., \( CAG_X \), and \( CAG_Y \) are two (possibly different) CAGs associated with \( X \) and \( Y \) respectively. The conceptual model of CAGs extends very naturally to multiple users. However, the key challenge is how to scale up the number of concurrent users, since a different CAG has to be managed for each user? Particularly, in the presence of big annotated schema graphs. The following section discusses our approach.

### 4.3 Overview of Our Approach and System Architecture

Using our previous exploration-based, context-aware interpretation approach, each user \( u_i \) will need to have a context-aware summary graph [14] which captures relative context of the concepts and relations to \( u_i \)’s current querying context. So for \( u_i \), and a summary graph \( G \) for a database, after the \( t^{th} \) query \( Q_t \), \( u_i \)’s context-aware graph is \( G^t_{u_i} = (G, \omega^t_{u_i}) \), in which \( \omega^t_{u_i} (n, H^t_{u_i}) = x \), where \( n \in G^t_{u_i} \), \( x \in R^+ \) and \( H^t_{u_i} \) is the set of hits of keywords in query \( Q_t \) in \( G \) for a particular user \( u_i \). The key observation is that for different users, \( G \) is fixed but \( \omega^t_{u_i} \) is varied. \( \omega^t_{u_i} \) represents the mappings from schema elements to weights for a particular user. Therefore, our approach reduces the overhead of supporting multiple concurrent interpretations by using an
index on the summary graph $G$ that can be shared across multiple interpretation instances. Then, for each user, our approach only needs to maintain the mappings defined by $\omega_{ui}^t$. This information is maintained in an index called \textit{personalized query context map} (PCM). After each query has been interpreted, PCM is updated to reflect this new information about query context i.e. $\omega_{ui}^t \rightarrow \omega_{ui}^{t+1}$ for user $i$. Another component of our approach is to avoid the overhead of maintaining different exploration states for different interpretation instances by eliminating the need for graph exploration altogether. Rather, it uses an approach of reassembling pre-computed substructures guided by appropriate indexes. Information about users $\omega_{ui}^t$ is used to bias the reassembly process so that only the most relevant substructures for a specific user are created. In the following we describe our architecture.
4.3.1 SKI System Architecture

SKI’s architecture consists of 3 layers: user layer, interpretation layer and query processing layer. The overall process (shown in Figure 4.1) begins with queries being sent from the user layer to the interpretation layer, followed by the generation of top-K query interpretations in the form SPARQL queries. These queries are sent to both the user as well as to the query answering layer which begins processing queries in order defined by top-K. However, if a user disagrees with the top-1 (indicated by selecting a different interpretation from the list, an interrupt is issued to the query processing layer, and the processing begins for the user-selected query, and an update is issued to the PCM (step 2) and propagated to all replicas. For the query processing layer, we assume any RDF database like RDF-3X[46], Sesame3, and for the rest of the discussion we will focus on the interpretation layer.

The interpretation layer employs a master-slave cluster architecture that consists of a master node and a set of interpretation slave clusters, or interpretation clusters for short. The master node is responsible for scheduling user query requests to clusters and exchanging interactive information such as resulting interpretations and query answers between users and interpretation clusters. The master assigns users to an interpretation cluster. Each interpretation cluster in turn has an interpretation cluster (IC) tracker that maintains information about the load for each of its constituent slave node. An IC tracker assigns a user query to a specific slave node as primary and along with two replica slaves. Each slave node runs multiple threads representing multiple interpretation instances. Within each interpretation instance there are 3 key steps (A – C) as shown in Figure 4.2:

- A). Initializing hits by looking up the inverted index that maps keywords to summary graph elements;

- B). Candidate roots identification: a bitmap-based algorithm for fast identification of candidate roots and eliminating keyword hits that cannot be used as part of the final subgraph construction;

---

3http://www.openrdf.org/
• C). GeFree: an algorithm that assemble and generate top-K interpretations from candidate roots.

On each slave node there are key indexes that are used to support these algorithms and are shared across the interpretation instances: **KS-Map** maps keywords to schema graph elements; **Rabbit Index**: a bitmap-based group reachability index for (i) rapid identification of root nodes of candidate connecting trees; (ii) pruning of hits that cannot reach those candidate root nodes in $\Delta$ hops. **Dense Path Index**: for fast construction of interpretations, i.e., interpretation without the need for cost-balanced graph exploration. We propose **GeFree**, which uses DPI and Rabbit to enable rapid assembly of the top-K minimum connecting trees representing the top-K interpretations using materialized path information. **Partition of PCM**: a partition of personalized query context map for maintaining dynamic user dependent contextual information. Before we discuss the algorithms, in the next section, we will define some key concepts
used in the proposed algorithm: $\Delta$-path covers, and $\Delta$-graph exploration graphs defines the structural information maintained for DPI and candidate root is a key concepts for Rabit and GeFree.

### 4.4 Keyword Query Interpretation in SKI

#### 4.4.1 Foundations

**Definition 4.4.1.** A **$\Delta$-path cover** of a node $r$ in graph $G$ is denoted by $\pi(\Delta, r, G) = \{p_i\}$, where $p_i$ is a path such that $\forall p_i, p_i \in \pi(\Delta, r, G)$ if and only if

- $r \in G$ is the source node of $p_i$ and,
- $|p_i| \leq \Delta$.

In other words, the path cover $\pi(\Delta, r, G)$ contains all reachable paths of length less than or equal to $\Delta$ from $r$. For example, a 2-path cover of node $D$ in the graph shown in Figure 4.3 is $\{D, D \rightarrow A, D \rightarrow B, D \rightarrow E, D \rightarrow X, D \rightarrow X \rightarrow N, D \rightarrow X \rightarrow I, D \rightarrow H \rightarrow M, D \rightarrow H \rightarrow I, D \rightarrow H \rightarrow E, D \rightarrow E \rightarrow H, D \rightarrow E \rightarrow I, D \rightarrow E \rightarrow F\}$.

**Definition 4.4.2.** A **$\Delta$-Exploration Graph** of a node $r$ in graph $G$ (denoted by $g_r^\Delta$) is a rooted subgraph of $G$, which contains all edges in $\pi(\Delta, r, G)$. For example in Figure 4.3, let $G_S$ be the whole graph, the subgraph in the box is a 2-exploration graph rooted at node $Z$ : $g_Z^2$, which contains all edges in $\pi(2, Z, G_S) = \{Z, Z \rightarrow Q, Z \rightarrow Q \rightarrow M, Z \rightarrow Q \rightarrow S, Z \rightarrow Q \rightarrow R\}$.

**Definition 4.4.3.** Given a keyword query $Q$ and a schema graph $G_S$, a **candidate root** of $Q$ is the root node $r$ of a $\Delta$-exploration graph $g_r^\Delta$, which contains at least one hit of each keyword in $Q$. For example, in Figure 4.3, $I$ is a candidate root for keyword query $k1, k2, k3$, because the 2-exploration graph $g_I^2$ contains a hit $P$ (matches $k1$), $K$ (matches $k2$) and $Y$ (matches $k3$). Those hits (such as $P, K, Y$ in $g_I^2$) that are covered by the $\Delta$-exploration graph $g_r^\Delta$ rooted at a candidate root $r$ are called **productive hits**. Productive hits can always reach a candidate root following a path of length less than or equal to $\Delta$. 

81
4.4.2 Identifying Candidate Roots and Productive Hits

Different from the graph exploration-based approaches, where roots are identified by exploring paths originated from each hits, in this section, we introduce a fast candidate roots identification algorithm. Given a keyword query $Q = \{k_i\}$, in order to identify a group of candidate roots, a “group reachability” question that needs to be answered is: Given a set of groups of nodes, whether every node in a group of nodes $CR$ can reach at least one node in ALL the other groups within $\Delta$ hops.

**Naive Approach.** Intuitively, a node-to-node reachability index can be used to answer this question. For example, in Figure 4.3, given a keyword query $Q = \{k_1, k_2, k_3\}$, $KE = \{A, P, K, Y, R, T\}$ is the set of hits of $Q$. Let $\Delta = 2$, which means we are only interested in those paths between root and any hit that are of length less than or equal to 2. To determine if the node $I$ is a candidate root, we can examine every node in $KE$ and check the reachability between each hit and $I$. In this way, we can identify $KE' = \{P, K, Y\}$, where each element can reach $I$ within 2 hops. Because $KE'$ contains at least one hit of all keywords in $Q$, $I$ is a candidate root. To generate all candidate roots, we need to apply the exact same steps for
all nodes in the graph. However, the steps discussed above to identify candidate roots are not efficient enough, the time complexity is $\Theta(nm)$, where $n$ is the number of nodes in the graph, and $m$ is the total number of hits for the keyword query $Q$. For reachability indexes like the R2R reachability index proposed by Markowetz et al., [43], where pairwise nodes reachability information is stored in a hashtable, the candidate roots can only be identified in those steps discussed above in $\Theta(nm)$. Same time complexity is expected for productive hits identification if we choose to use reachability index that maintains pairwise reachability information.

Productive hits are those hits that can reach candidate roots. We only keep productive hits and prune those unproductive hits because those hits will never reach a root node within $\Delta$ hops. Eliminating those unproductive hits can significantly reduce the search space, and for traditional graph exploration algorithm, exploration from those hits is wasted. For naive approach using pairwise reachability information, productive hits can be identified in the following steps after candidate roots are identified. Recall the example in the last paragraph where we showed how to identify candidate roots, for the set of all hits of $Q = \{k1, k2, k3\}$: $KE = \{A, P, K, Y, R, T\}$ and the set of candidate roots: $CR = \{I\}$, for each candidate root in $CR$, we iterate all hits in $KE$ and check their reachability. Those hits that can reach at least one candidate root in $CR$ will be considered as a productive hits, which for this example is $PKE = \{P, K, Y\}$. The time complexity of this method is $\Theta(|CR|*|KE|)$.

**Group Reachability Bitmap Index.**

We propose a group reachability bitmap index (Rabit) to avoid pairwise reachability checks and identify candidate roots and productive hits more efficiently. For each node $u_i$ in the graph, we propose to use a *bit vector* denoted by

$$\beta_i = <b^i_1, b^i_2, ..., b^i_{|V(G_S)|}>$$

to represent the nodes that can reach $u_i$ within $\Delta$ hops, i.e., for all the nodes in $V(g^\Delta_{u_i})$, $b^i_j = 1$ implies that $u_j \in V(g^\Delta_{u_i})$ and $b^i_j = 0$ implies that $u_j \notin V(g^\Delta_{u_i})$. For example, in Figure 4.4, each row in the matrix represents the bit vector for the corresponding node in the graph shown
in Figure 4.3. $\beta_A = < 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 >$, implies that $\{A, B, D, E, H, X\} \in V(g^2_A)$ are nodes that can reach $A$ within 2 hops. Notice that, each bit vector is 26 bits long, and we can compress each bit vector to an 32-bit integer such that we can use one integer to represent the $\Delta$ exploration graph for each node. For example, for the matrix shown in Figure 4.4, we can use an a integer to represent the bit vector for node $A$, the corresponding integer is 56885252 according to the bit vector $\beta_A$. In the rest of the chapter, we use node ID instead of column index to identify a specific bit vector or bit. For instance, $\beta_A$ corresponds to $\beta_0$ because the index of node $A$ is 0, so as the index of a bit in a bit vector: $b_A$ is equivalent to $b_0$. In addition, the bit vector can not only represent the nodes in a $\Delta$-exploration graph of a certain node, in general, it can also represent any set of nodes and naturally represent a single node. For example, a node set contains only $A$ can be represented using a bit vector $\gamma(\{A\}) = < 1, 0, 0, ... , 0 >$, where only $b_A = 1$, and $\gamma$ is a function that maps a set of nodes $U = \{u_1, ... u_m\}$ to bit vector such that only $b_{u_i} = 1$ iff $u_i \in U$. Now, we can check the reachability of two nodes using bitwise AND (i.e., & ) in the following way: check
if $\beta_u \& \gamma({v})$ equals to $\gamma({v})$. Take a simple example, we can check if node $H$ can reach $A$ by calculating $\beta_A \& \gamma({H}) = 56885252 \& 262148 = 262148$, which means, $H$ can reach $A$ within 2 hops. In practice, the number of nodes in a graph may exceed 32, and a sequence of 64-bit integers need to be used to represent a single compressed bit vector:

$<n_0, n_1, ..., n_{\lceil N/64 \rceil}>$, where $N = |V(G_S)|$, $n_i = <b_{64i+0}, b_{64i+1}, ..., b_{64i+(N\%64)}>$.

Alike, the $\gamma$ function can also map a node set to the compressed representation of bit vector. Particularly, $\gamma(\Phi) = <0, ..., 0>$ is a sequence that contains $\lceil N/64 \rceil$ 0s, which represents the bit vector for empty set $\Phi$; alike $\gamma(\cup) = <1, ..., 1>$ is a sequence that contains $\lceil N/64 \rceil$ 1s, which represents the bit vector for universal set $\cup$.

**Algorithms.**

Having Rabit, we can now answer some questions on group reachability easier, which are key questions for candidate roots and productive hits identification:

1. Given a set of nodes $U = \{u_i\}$, how to identify another set of nodes $X = \{v_i\}$ such that any node in $X$ can reach at least one of the node in $U$ within $\Delta$ hops, i.e., $\forall v_i \in X$, $\exists u_j \in U$ such that $v_i \in V(g^\Delta_{u_j})$

2. Given several groups of nodes $\{G_i = \{u^i_j\}\}$, how to identify another set of nodes $Y = \{v_k\}$ such that $v_k$ can reach at least one of the node in all groups within $\Delta$ hops, i.e., $\forall v_k \in Y$ such that $\forall i \leq |\{G_i\}|$, $\exists j$ such that $v_k \in V(g^\Delta_{u^i_j})$.

To answer question 1), instead of pairwise reachability checks for all combinations of nodes in $U$ and $V(G_S)$, with Rabit, we can simply follow the GetNeighbors algorithm shown in Algorithm 5:

We can consider the answer to the question 1) as the union of (direct and indirect) neighbors of the nodes in $U$. In the GetNeighbors algorithm, a bit vector $x$ is initialized representing an empty set (line 2). For each node $u_i \in U$, the bit vector representing the nodes in $g^\Delta_{u_i}$ is $\beta_{u_i}$. The bitwise OR operator (i.e., |) is used to calculate the a new vector $x$ that represents the union
Algorithm 5 GetNeighbors

1: Input: $U = \{u_i\}; \Delta$
2: $x = \gamma(\phi)$
3: for all $u_i \in U$ do
4: $x = x \mid \beta_{u_i}$
5: end for
6: return $x$

Figure 4.5: Question 2) How to identify nodes that can reach at least one node in all groups of nodes that can reach at least one node in $U$ that has been iterated in the for loop (line 4).

Finally, this algorithm returns $x$, which is a bit vector that represents the union of neighboring nodes of all nodes in $U$. For example, in Figure 4.3, $x = \text{GetNeighbors}(\{A, C\}, 2) = \beta_A \mid \beta_C = 56885252 \mid 12255232 = 67043332 = 11111111100000000000100_2$, which means the set of node $\{A, B, C, D, E, F, G, H, I, J, X\}$ is the union of neighbors of $\{A, C\}$.

Question 2), which we call group reachability problem, is the basis for identifying candidate roots. Intuitively, question 2) can be answered in the following way: as shown in Figure 4.5, give two node groups $\{u_1, u_2\}$ and $\{v_1, v_2\}$, $X1, X2, Y1, Y2$ represent the neighbors of $u_1, u_2, v_1, v_2$ respectively. To identify the nodes that can reach at least one node in both groups,
we need to first find the union of all the neighbors of the first group \{u_1, u_2\}, which is \(X_1 \cup X_2\); similarly, we need to find the union of all the neighbors of the second group \{v_1, v_2\}, which is \(Y_1 \cup Y_2\). The next step is to identify those common nodes in the two groups of neighbors, which are nodes in \(Z = (X_1 \cup X_2) \cap (Y_1 \cup Y_2)\).

Therefore, there are two steps: i) Identify the neighbors of each group of nodes. In other words, given a set of node groups \(\{G_i\}\), a set of neighbor groups \(\{NG_i\}\) should be computed where \(NG_i\) can be represented by the bit vector \(x = GetNeighbors(G_i, \Delta)\); ii) Identify the common of nodes in all the neighbor groups, i.e., a set of nodes \(CN\) such that \(\forall i, CN \subseteq NG_i\). The two steps are illustrated by the following algorithm.

### Algorithm 6 GetRoots

1: Input: \(G = \{G_i\}; \Delta\).
2: \(y = \gamma(\cup)\)
3: for all \(G_i \in G\) do
4: \(x = GetNeighbors(G_i, \Delta)\);
5: \(y = x \& y\);
6: end for
7: return \(y\).

In Algorithm 6, a bit vector \(y\) is initialized representing an universal set (line 2). For each node group \(G_i\) in \(G\), a bit vector \(x\) representing neighbors of \(G_i\) (line 2) is returned using \(GetNeighbors\). The bitwise AND operator is applied to \(x\) and \(y\) to compute the common node set of all the node groups (line 5).

By answering the question 1) and 2), we are now able to identify candidate roots by using Algorithm 6, where the inputs are \(\Delta\) and \(|Q|\) groups of hits: \(\{G_i \mid G_i = \{u_1^{w_i}\}\}\), and \(G_i\) is the group contains all hits for \(w_i\). \(GetRoots(\{G_i\}, \Delta)\) returns a bit vector representing the set of candidate roots. For example, for keyword query “\(k1,k2,k3\)” in Figure 4.3, the candidate roots can be calculated in the following steps: the 3 groups of hits for each keyword are \(G_1 = \{A, P\}\), \(G_2 = \{K, T\}\), \(G_3 = \{Y, R\}\).
For $k1$, $x1 = GetNeighbor(G1,2) = \beta_A \mid \beta_P = 56885252 \mid 138244 = 57023492$;

For $k2$, $x2 = GetNeighbor(G2,2) = \beta_K \mid \beta_T = 1294338 \mid 736 = 1295074$;

For $k3$, $x3 = GetNeighbor(G3,2) = \beta_Y \mid \beta_R = 1294338 \mid 9089 = 1303427$.

$x1 \& x2 \& x3 = 131072 = 000000001000000000000000000002$, which means, only the node $I$ is the candidate root.

After the candidate roots are identified, we can use ProductiveKE shown in Algorithm 7 to identify productive hits. For every hit $u^j_i$ (i.e., the jth hit for keyword $w_i$), the common nodes between the neighboring nodes of $u^j_i$ and the candidate roots (represented by $z$ line 3) are calculated, which is represented by a bit vector $tmp$ (line 6). If $tmp$ is not zero, which means $u^j_i$ can reach at least one candidate root within $\Delta$ hops (line 7), this node will be considered as one of the productive hits. For example, in Figure 4.3, we have already know the candidate roots, which can be represented by a bit vector $z = 131072$. We can determine if $A$ is a productive hit by calculating $\beta_A \& z = 56885252 \& 131072 = 0$, which means, $A$ is not a productive hit and therefore, can be pruned.

---

**Algorithm 7 ProductiveKE**

1: Input: $|Q|$ groups of hits: $\mathcal{G} = \{\mathcal{G}_i\}; \Delta$.
2: Initialize an empty set $ANS$;
3: $z = GetRoots(\mathcal{G}, \Delta)$
4: for all $\mathcal{G}_i \in \mathcal{G}$ do
5:     for all $u^j_i \in \mathcal{G}_i$ do
6:         $tmp = z \& \beta_{u^j_i}$;
7:         if $tmp \neq 0$ then
8:             $ANS.Add(u^j_i)$
9:         end if
10:     end for
11: end for
12: return $ANS$;
4.4.3 GeFree: A Graph Exploration Free Approach

In the previous sub-section, we discussed how to identify candidate roots and productive hits, which are the first step towards generating top-K interpretations without exploring the graph. To achieve the final goal, in this sub-section, we will present another important index called dense path index (DPI). Having unproductive hits pruned, given a set of candidate roots \( CR = \{r_i\} \), for keyword query \( Q \) and each root \( r_i \), there are many possible interpretations rooted at \( r_i \). However, only knowing the roots is not enough to identify the top-K interpretations among all possible interpretations that rooted at each root in \( CR \). The paths from a root node to all hits need to be computed and assembled to construct an interpretation.

**Dense Path Index.** The dense path index is a two-level hierarchical index. The first layer is called node-exploration graph map, or NE map, which is organized as a hash table that maps any node \( r \) in \( G_S \) to a \( \Delta \)-exploration graph \( g_r^\Delta \). \( NE(r) = g_r^\Delta \) denotes such mapping. The second layer of the dense path index is called destination-path map, or DP map that maps a destination node to a set of paths: For each \( \Delta \)-exploration graph, the \( \Delta \)-path cover of \( r \), i.e., \( \pi(\Delta, r, g_r^\Delta) \) is pre-computed, and those paths in \( \pi(\Delta, r, g_r^\Delta) \) are grouped by destination nodes \( \pi(\Delta, r, g_r^\Delta) = \{P_{v_0} \cup P_{v_1} \cup ... \cup P_{v_m}\} \), where \( P_{v_i} \) is called a path group of \( v_i \), where \( v_i \in V(g_r^\Delta) \) is a destination node, and \( \forall i, j, P_{v_i} \cap P_{v_j} = \phi \). \( DP(v, g_r^\Delta) = P_v \) denotes the mapping for \( g_r^\Delta \) from a destination node \( v \) to the path group \( P_v \) of \( v \). The \( \Delta \)-path cover can be organized as a hash table that maps any destination node \( v_j \in V(g_r^\Delta) \) to the path group of \( v_j \), i.e., \( P_{v_j} \). The two maps are illustrated as follows:
Figure 4.6: Example of path index

**DensePathIndex**:

\[
v_i \to g^\Delta_{vi}
\]

\[
\pi(\Delta, v_i, g^\Delta_{vi}):
\begin{bmatrix}
\vdots \\
\end{bmatrix}
\]

\[
v_j \to P_{vj} = \{p_{v_1 \rightarrow v_j}^{v_i \rightarrow v_j}, \ldots p_{v_L \rightarrow v_j}^{v_i \rightarrow v_j}\}
\]

\[
\vdots
\]

Figure 4.6 shows an example of the dense path index for the graph shown in Figure 4.3. For each root-keyword pair, at most the top-K minimum weighted paths are necessary for computing top-K interpretations. For example, given a keyword query \(Q = <w_1, w_2>\), considering a candidate root \(r\), assuming that we are only interested in top-2 interpretations, let \(P_1 = <p_1^1, p_2^1, p_3^1>\) be the top-3 paths from \(r\) to any hits of \(w_1\); \(P_2 = <p_1^2, p_2^2>\) be the top-2 paths from \(r\) to any hits of \(w_2\), any combinations of paths including \(p_3^1\) will never form a top-2 interpretation and should never be investigated during the processing of generating top-K interpretations. The GeFree algorithm is illustrated in Algorithm 8.

**GeFree Algorithm.** For each keyword \(w_i\), only productive hits are left and stored in \(H[i]\) for \(w_i\) (line 4). For each candidate root \(r_i\), the corresponding \(\Delta\)-exploration graph is returned
Algorithm 8 GeFree

1: Input: $Q = \{w_i\}; CR = \{r_i\}; PKE = \{u_i\}.$
2: $TOPK = \phi$ is a priority queue for maintaining top-K interpretations
3: \textbf{for all} $w_i \in Q$ \textbf{do}
4:  $H[i] = \text{hit}(w_i) \cap PKE$;
5: \textbf{end for}
6: \textbf{for all} $r_i \in CR$ \textbf{do}
7:  $g^\Delta_{r_i} = NE(r_i)$;
8: \textbf{for all} $w_j \in Q$ \textbf{do}
9:  $PQ_j; / / \text{Initialize a priority queue for maintaining top-K paths}$
10: \hspace{0.5cm} \textbf{for all} Hit $v_{jk} \in H[j]$ \textbf{do}
11:  $PG = DP(v_{jk}, g^\Delta_{r_i});$
12: \hspace{1cm} Calculate $P_k : \text{TopK paths in PG as a heap}$;
13: \hspace{1cm} $P_k \rightarrow PQ_j;$
14: \hspace{1cm} Sorted Path Group $S_j = PQ_j.\text{HeapSort}();$
15: \textbf{end for}
16: \textbf{end for}
17: $TOPK = TopKCombinations(\{S_j\});$
18: \textbf{end for}
19: \textbf{return} $TOPK$;

using $NE$ map (line 7). Then, for each keyword $w_j$, $H[j]$ contains a set of productive hits for $w_j$; for each productive hit $v_{jk}$, $DP$ map returns a path group $P_k$ (line 11). For a candidate root $r_i$ and a keyword $w_j$, Top-K root-keyword paths are computed by merging topK-heaps from each path groups (sorted by costs) of all productive hits of $w_j$ into a priority queue $PQ_j$ of size $K$ (line 12 and line 13). In line 14, for the specific root $r_i$, a sorted list $S_j$ containing $K$ paths is computed for each keyword. $|Q|$ lists are computed in total. $TopKCombinations(\{S_j\})$ is an algorithm to quickly generate top-K combinations of paths from $|Q|$ sorted path lists. The algorithm is based on the $TopKCombinations$ algorithm proposed in [14]. This algorithm suggests an early termination strategy such that top-K combinations will be generated without enumerating all combinations of paths and has guaranteed $O(K|Q|)$ time complexity. The time complexity of GeFree is $O(|CR|(|K|Q| + K|PKE|\lg(K)))$, where $CR$ is the set of candidate roots, and $PKE$ is the set of all productive hits, $K$ is the number of top-K interpretations generated. Because in the worst case, $CR$ may contain all nodes in the graph, and $PKE$ is
equal to a set of all possible hits. The worst case time complexity is \( O(n(K^{|Q|} + Km \log(K))) \),
where \( n = |V(G_S)| \), and \( m = \sum |hit(w_i)| \) is the total number of hits. In comparison, the time complexity for graph exploration-based algorithm as reported in COSI [14] is \( O(nd^\Delta K^{|Q|}) \), where \( d \) is the degree of the graph.

### 4.5 Evaluation

The focus of our evaluation was to test the scalability of the proposed multi-tenant query interpretation system (SKI) with respect to latency, concurrency, and throughput.

**Testbed:** Our testbed includes two real-life datasets: DBPedia 3.6 and the “Billion Triple Challenge 2009” (BTC) collection. DBPedia 3.6 has 300 classes, 3K property types, 3.5 million entities represented in 678 million RDF triples. BTC 2009 contains 1.19 billion triples, from which we created the schema graph that includes 150232 classes and around 1.3M properties (i.e., a schema graph with 150232 nodes and 1.3M edges).

Our query collection comprises of 28 queries with varying degrees of ambiguity. **Degree of term ambiguity (DOTA)** is a metric introduced in our previous work [14] that characterizes the ambiguity of a query as a reflection of how much work (i.e., exploration search space) is needed for its interpretation. Clearly, if a query is not ambiguous, then the problem of interpretation is trivial. Therefore, we consider this an important metric for the evaluation of the keyword search. DOTA is defined as a function of the number of combinations of all keyword hits. 8 of the queries are the list queries from Semantic Search Challenge 2011 [4] with stopwords removed. The remaining 20 queries were created by randomly selecting a list of “ambiguous” tokens from databases (tokens associated with multiple unrelated classes and properties implying different meanings). Then, queries of length from 3 to 5 were crafted by manually assembling the selected keywords into meaningful queries. The DOTA values of those 20 queries range from 2 to \( 1 \times 10^{10} \) (most ambiguous). Among the 20 queries, 10 of them are selected from DBPedia (identified by \( Q = q_1...q_{10} \)), and the other 10 queries (identified by \( Q' =

---

q1'...q10') are selected from BTC (note that randomly selected keywords from BTC in those queries may not exist in DBPedia). All queries, experimental results, and data can be found in the following link\(^5\).

**Setup:** We evaluated the performance of our keyword query interpretation algorithms in SKI with TKQ2S[67] and COSI[14] in terms of execution time and memory consumption in megabytes. The evaluation of concurrency and throughput were conducted on a cluster of up to 30 slave nodes, each of which is a PC with dual core 2.0 GHz CPU and 8G memory running Windows 8 Server. On the other hand, the experiments on efficiency and scalability were conducted on a machine with an Intel i5 2.5 GHz CPU and 8G memory PC running Windows 7. All results recorded were averaged over at least 5 trials. All algorithms were implemented using Visual Studio 2012 using C#.

### 4.5.1 Data, Query Scalability, and Performance

The SKI approach was compared against TKQ2S and COSI over both datasets using the two groups of queries Q and Q'. The purpose of the experiments was to investigate the relative performance advantage of the index-based interpretation approach over exploration-based approaches with increasing data and increasing degree of ambiguity. Results are shown in Figure 4.7) (A) and (B) for DBPedia and BTC datasets respectively. SKI outperforms both COSI and TKQ2S on all cases. We see a general trend in both datasets where the SKI approach scales better with increasing DOTA and outperforms the other approach COSI. COSI performs better than TKS2QS because COSI employs an early termination condition that is not used by T2KQS. It is important to note that the DOTA metric doesn’t capture all of the ambiguity inherent in keywords. It doesn’t reflect the degree of connections between keywords, but only how much differentiation there is in keywords’ occurrences. This may explain some of the spikes in the graph. For example, for q6’, the performance of COSI in BTC drops considerably because the number of cursors generated during graph exploration increases a lot (from 33469

\(^5\)http://research.csc.ncsu.edu/coul/SKI/experiments.xlsx
to 70047), which implies a longer search period. However, we note that the performance of SKI is relatively stable with increasing DOTA. That is because the performance of SKI does not rely on the cursors for graph exploration, but depends on the number of candidate roots and the productive keyword hits identified. Usually the number of candidate roots increase with the growing of DOTA, but the growth rate is relatively slow. Due to the space limitation, for each algorithm, additional experiments concerning the impact of factors other than DOTA (such as the number of cursors generated, number of candidate roots, search depth, top-K, number of keywords) to the performance can be found using the following link\(^6\). In Figure 4.7 (A), there

\(^6\) [http://research.csc.ncsu.edu/coul/SKI/experiments2.xlsx]
is no obvious trend for SKI before the query q7. This is because, the DOTAs for the first 7 queries do not vary too much. Higher DOTA usually implies more candidate roots because the search space is larger, and the possibility that more class nodes are identified as candidate roots should increase. But a little bit higher DOTA cannot guarantee more candidate roots to be generated for example, for the query with DOTA=1740, the number of identified candidate roots is 39 where are 73 candidate roots are identified for the query with DOTA=658.

Figure 4.8: Performance of interpretation algorithms for same set of keyword queries on datasets with different scale
Given, the clear superiority of COSI over TK2QS both in terms of data and query ambiguity, the rest of the experiments use COSI as the representative graph explorative technique. We compare SKI and COSI using the 8 queries from the “Semantic Search Challenge 2011” over both the BTC and DBPedia datasets. All keywords appear in both datasets but with varying frequencies since BTC is a superset of the DBPedia dataset. Thus, the queries tend to have much higher DOTA in the BTC dataset than in DBPedia. Figure 4.8 (B) shows the different DOTAs for each query in both datasets, and Figure 4.8 (A) shows the comparative performance of interpretation on both datasets for the 8 queries (note that the Y-axis is on a log scale). SKI outperforms COSI for all queries over both datasets. The performances of both COSI and SKI drop when interpreting the same queries on a dataset with larger scale since the degree of ambiguity increases for the same keyword query. For queries Q6 to Q8, the DOTA increases considerably, SKI performs much stable on both BTC and DBPedia while the performance of COSI drops considerably because the search space increases significantly with the growth of DOTA. Note that the queries are ordered by the DOTAs of queries over DBPedia, not BTC. The DOTA of Q2 over BTC is larger than the DOTAs of Q1 and Q3, therefore, the time spent by COSI for Q2 on BTC is higher than that for Q1 and Q3.

![Figure 4.9: Impact of MPL on latency](image_url)
4.5.2 Impact of Memory Consumption on Concurrency

In this experiment, we use the multi-programming level (MPL) (i.e. the number of concurrent programs running concurrently) as the means of comparing the degree of concurrency supported by SKI and COSI. Due to the resource contention between multiple programs, the performance and memory consumption of the interpretation algorithm impact the response latency of concurrently running queries. Figure 4.9 shows the impact of MPL on the latency (in milliseconds) for interpretation algorithms. Although all approaches experience increased latency as MPL increases, the rate of increase is higher with the exploration-based approaches than the index approach of SKI. This suggests that the concurrency overhead for COSI and TKQ2S is higher than SKI. Note that, SKI supports much higher multi-programming level while keeping low latency. However, for COSI or TKQ2S, due to high memory consumption and high CPU overhead, they are difficult to maintain reasonable latency for MPL greater than 5. As an explanation for this behavior, we report the memory consumption for SKI, COSI, and TKQ2S in Table 4.1 and estimate the maximum number of concurrent user queries they can support.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Approaches</th>
<th>Graph</th>
<th>PCM</th>
<th>KS-MAP</th>
<th>DPI</th>
<th>RABIT</th>
<th>EstMaxU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTC</td>
<td>SKI</td>
<td>337mb</td>
<td>2.8mb</td>
<td>221.9mb</td>
<td>155.6mb</td>
<td>25.6mb</td>
<td>2857</td>
</tr>
<tr>
<td></td>
<td>COSI</td>
<td>445mb</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>TKQ2S</td>
<td>453mb</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>17</td>
</tr>
<tr>
<td>DBPedia</td>
<td>SKI</td>
<td>1.1mb</td>
<td>0.07mb</td>
<td>38.8mb</td>
<td>6.8mb</td>
<td>1.3mb</td>
<td>114286</td>
</tr>
<tr>
<td></td>
<td>COSI</td>
<td>1.4mb</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5714</td>
</tr>
<tr>
<td></td>
<td>TKQ2S</td>
<td>1.4mb</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5714</td>
</tr>
</tbody>
</table>

In this table, the memory consumption for maintaining schema graph, PCM for each user, and indexes including KS-Map, DPI, Rabit are reported. Note that, the reported memory usage of graph for graph exploration-based algorithms includes the overhead of maintaining intermediate graph exploration states such as paths explored and cursors generated. In the last
column, we estimate the maximum number of user queries that can be running concurrently based on the memory consumption of each approach. According to this table, for SKI, the PCM for each user consumes only 2.8MB RAM for BTC while other indexes are shared by all users. However, for COSI or TKQ2S, each user requires around 445MB for maintaining graph exploration states associated with the summarized schema graph. Therefore, given a PC with 8GB RAM, SKI can process query requests for up to 2857 users concurrently on BTC, while COSI or TKQ2S can only support up to 17 users.

4.5.3 Impact of the Number of Slave Nodes on the Throughput

In this experiment, we evaluate the performance and throughput of our distributed multi-tenant keyword query interpretation system, i.e., SKI by varying the number of user queries and the number of slave nodes in a slave cluster. The master node simulates up to 10000 random user queries from 1000 different users and continuously send those queries to the slave nodes in a slave cluster. We measure the time a slave cluster spent from the master node issues the first query to the master node receives the last result from the slave cluster. As you can see in Figure 4.5.3 (A), ten groups of experiments with different number of queries were conducted. For each group, we compare the performances of slave clusters with different numbers of slave nodes in a cluster (SC5 represents a slave cluster with 5 nodes in it). As expected, for a fixed number of queries, a cluster with more slave nodes in it gives rise to high performance; in addition, with the increasing number of queries, a slave cluster needs more time to process them. Note that, a cluster with more slave nodes in it is more scalable with the growing number of queries. For example, the increase rate of SC5 is much higher than SC30. That is because the cluster performance equals to \( \frac{m}{n} t + T \), where \( m \) is the total number of queries, \( n \) is the number of slave nodes, \( t \) is the average time for interpreting one query, and \( T \) is the time for transmitting queries from master node to all slave nodes. That means, the increasing rate is inversely proportional to the number of nodes in a slave cluster. On the other hand, the throughput is measured using the same sets of experiments as the previous one, the evaluation
results are illustrated in Figure 4.5.3 (B). In figure (B), experimental results are grouped based on the sizes of the slave clusters. In each group, the throughput (i.e., the number of queries per second) is measured for different query streams with varying number of queries. As expected, the throughput is increasing with the growing number of slave nodes in a slave cluster. For each group, the throughput is also increasing when more user queries are issued concurrently. That is because the throughput equals to the number of queries divided by the total time: \[ \frac{m}{n \cdot t + T} = \frac{1}{\frac{n}{m} + \frac{T}{m}}. \] That implies that with fixed \( n \), \( t \), and \( T \), the throughput is increasing with the
growing number of \( m \), i.e., the total number of slave nodes.

### 4.5.4 Efficiency of Candidate Root Identification

We compared the efficiency of identifying candidate roots and productive hits using the *Rabit* and naive reachability indexing schemes using the 11 queries. Figure 4.11 a) shows the performance gain of *Rabit* over the naive reachability index. As expected, the *Rabit* index consistently outperforms the naive reachability index because it avoids the pairwise reachability checks. Its performance is stable with increasing DOTA until about 37413464. A key reason for this is that the key factors that affect the performance of *Rabit* are the number of hits. However, although there is not a direct correlation between these factors and DOTA, it possible that an increased DOTA increases the likelihood of number of hits. We see that in Figure 4.11 a). The naive index has an overall increasing trend with increasing DOTA with some spikes (DOTA = 36 and 6880). The reason is that the naive index depends on both the number of hits and the number of candidate roots generated. For example, for query with DOTA=6880, the number of candidate roots identified is 159 while for the query with DOTA=1740 and 30975, the number of candidate roots are 119 and 114 respectively. Figure 4.11 b) shows some insight into the impact of efficient candidate root generation (using *Rabit*) on the graph exploration-based algorithms. It shows the percentage amount of time saved with respect to the number of cursors saved (which correlates with the search space). As shown in Figure 4.11 c) to f), the execution times for CoSi+Rabit and the CoSi+naive reachability index outperform CoSi (alone) in most cases. But in a few cases, CoSi outperforms CoSi+Rabit or CoSi+naive reachability index, especially for queries with small search depth (Figure 4.11 c)), small number of keywords (Figure 4.11 d)) or small DOTA (Figure 4.11 e)). That is because for the latter cases, less needless hits are pruned, and therefore, less search space is reduced. For example, for one of the keywords in the query with DOTA=2 (“Porsche 922”), only one hit of the keyword “Porsche” is pruned. Therefore, not a lot of time reduction is achieved from the small reduction in exploration space. On the other hand, in Figure 4.11 e), the execution time for interpretation using CoSi+Rabit
for query with $\text{DOTA}=30975$ drops significantly because, although its $\text{DOTA}$ is large, one of the keywords appears only once in the database. This means that most of the hits for the other keywords will be unproductive because they don’t reach that single hit within $\Delta$ hops. However, we observe in the query with $\text{DOTA}=525$, that $\text{DOTA}$ is not the only factor that affects the performance which is why its execution time is longer than other queries with higher $\text{DOTA}$. This is because structural properties of the graph such as the degree of nodes also have an impact.

![Efficiency evaluation: a) to f)](image_url)

Figure 4.11: Efficiency evaluation: a) to f)
4.5.5 Other Factors that Impact the Performance of Exploration-Free and Exploration-Based Query Interpretation Algorithms

In this experiment we compare the GeFree algorithm with TKQ2S, CoSi, CoSi+Rabit and Cosi+Naive reachability index. The purpose of this group of experiments is to show the performance gain achieved by avoiding graph exploration. The comparisons were done by varying DOTA, top-K, search depth and the number of keywords. The results are shown in (Figure 4.11 c) to f)). In Figure 4.11 e) execution times for query interpretations for 9 different queries with different DOTAs are shown. GeFree is relatively unimpacted by DOTA but rather by search depth, number of keywords, and $K$ of top-$K$. Figure 4.11 f) shows that all approaches are not significantly impacted by $K$. Search depth $\Delta$ is a critical factor that impacts the performance significantly. As shown in Figure 4.11 c), the time is exponentially increased due to the growth of search depth. However, the proposed approaches including GeFree and Cosi+Rabit degrade more gracefully with increased search depth. Figure 4.11 d) shows that the execution times grow with increasing number of keywords because large number of keywords usually leads to higher DOTA. For GeFree, the performance for the first keyword query with 4 keywords drops because for that query, a large number of hits are pruned, while for the other 4-keyword query right after the first 4-keyword query, the DOTA increases from to 30975 to 37413464, resulting in a significant increase in execution time. We observe that CoSi+Rabit’s performance
is very close to that of GeFree because they both do not need to explore a large proportion of search space (GeFree - none at all) even in the presence of high DOTAs. The performance gain of GeFree over Cosi+Rabit is correlated with the remaining unpruned search space that Cosi+Rabit still needs to explore. Figure 4.12 g) shows the relationship between unpruned search space and execution time. The X-axis is the actual search space measured by the number of cursors generated after pruning. The percentages of times saved by GeFree against Rabit are on the Y-axis. As expected is shows that with a larger amount of search space left, the more time Cosi+Rabit needs for query execution and consequently more time is performance advantage for GeFree. The last but not the least, search depth $\Delta$ is a critical factor that impact the performance significantly. As shown in Figure 4.12 h), the time exponentially increased due to the growth of search depth. However, our approaches including GeFree and Rabit grow more smoothly than other approaches. Because usually, users are not likely to issue very complex queries with a large number of joins in it, every top-K interpretation consists of relatively small number of nodes and edges, and it is not necessary to search with depth larger than 2.

4.6 Related Work

In some previous work COSI [14][16], we formalized the problem of context-aware query interpretation and proposed techniques for modeling ambient search context and generating context-aware query interpretations. The idea is to model ambient context as a dynamic weighted schema graph in which weights of nodes (represents classes) and edges (represents properties) represent the degree of remanence of those classes and properties to the query context. This dynamic weighted graph model is then used as the substrate for the graph exploration step of query interpretation where the top-K minimum connecting trees are found, each representing a specific structured query. COSI focus on effectiveness, however suffer from the overhead of expensive graph exploration. As to approaches that do not require graph exploration, BLINKS [20] proposed a similar idea to pre-compute i) node to keyword index and ii) keyword to node index. However, BLINKS employs a distinct root semantics where a root node identifies an
unique subgraph that connects all keywords by shortest paths. While the distinct root semantics heuristic is very useful for significantly reduces search space in data graphs, it is too limiting for query interpretation which is done on schema graphs because it would prune many closely related interpretations. In this research we propose to save and manage richer path information (not only shortest path) and generate answers with Steiner tree semantics, which implies larger search space. In addition, BLINKS maintains mappings from keyword to lists of nodes ordered by costs; EASE [36] proposed EI-index, where ordered weighted r-radius graphs for pair wise tokens are pre-computed. However, adapting these kinds of indexes to personalized query interpretation such as COSI[14] has some challenges. Because after every query, the classes and properties associated with the query, as well as related concepts and properties, will need to have their weights increased to reflect the impact of the query on the context model, the ordering of all indexes containing such classes and properties will need to be updated. For existing techniques this may require a complete rebuild after each query which is impractical.

### 4.7 Conclusion

This work presents an approach for “interpretive” search that scales well under high concurrent workloads. The approach achieves this by relying on a suitable set of indexes rather than expensive graph exploration-based algorithms. Personalization was integrated by devising light-weight indexing scheme for information about user querying context. Comprehensive evaluation over real world datasets showed very promising results. Future work will explore closer integration to query answering systems.
Chapter 5

Conclusion

This thesis has presented techniques for efficient, effective, and scalable keyword query interpretations for knowledge bases that are organized in RDF databases. We propose approaches for integrating context-awareness into traditional systems used to interpret keyword queries. To achieve context-aware keyword query interpretation, we devised models and algorithms for two types of contextual information: the inter-query context and the intra-query context. For the inter-query context, we propose capturing a user’s “ambient search context,” possibly by considering the queries in the immediate context of the current query, that is, the query history. We formalized the problem of context-aware query interpretation and propose techniques for modeling an ambient search context and generating context-aware query interpretations. The idea is to model ambient context as a dynamic, weighted, extended schema graph in which the weights of nodes (which represent classes) and edges (which represent properties) represent the degree of relevance of those classes and properties to the query context. This dynamic weighted graph model is then used as the substrate in the graph exploration step of query interpretation, where the top-K minimum connecting trees are found, each representing a specific structured query. On the other hand, we investigate another type of contextual information that is also useful for disambiguating a keyword query – the intra-query context. We observe that the complete interpretation of a keyword query includes the terms given in the query as well as the
relationships found between those terms in the database. Consequently, an important part of
the interpretation process is to identify the close relationships between keyword terms. However,
it is also important to pay close attention to the relationships implied by the query structure.
Hence, in future research, we would like to limit our search for close associations between key-
words that are also close in the query. In addition, we would like to broaden the scope of our
association-finding in a query as much as possible so that we decompose or segment a query
into semantic units of closely related terms that are as large as possible. In this part of our
research, we propose a “deep segmentation” technique for identifying the coarse-grained seg-
ments of a keyword query that correspond to logical semantic units typically ignored by existing
techniques. However, all the approaches to interpretations proposed so far are based on graph
exploration-based algorithms. An important point to note about all these techniques is that the
graph exploration process used in computing the subgraphs requires substantial costs, and it is
difficult to support a multi-tenant environment and scale with the growth of data. Therefore, if
we can eliminate the need for graph exploration by using appropriate indexes and algorithms,
we can significantly improve the efficiency of query interpretation. In the last component, we
proposed a multi-tenant keyword query interpretation that is designed to maximize sharable
computations from graph exploration states such that all user-independent computation and
information can be reused and shared across all users. Hence, both concurrency and throughput
are significantly improved.
REFERENCES


[38] Li, L., Petschulat, S., Tang, G., Pei, J., and Luk, W.-S. Efficient and effective aggregate keyword search on relational databases. IJDWM 8, 4 (2012), 41–81.


