GRAPH BY EXAMPLE: AN EXPLORATORY

GRAPH QUERY INTERFACE FOR RDF

DATABASES

by

CHENG YANG

Submitted in partial fulfillment of the requirements for the degree

of Master of Science

Thesis Advisor: Dr. Z. Meral Özsoyoğlu

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2016
CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Cheng Yang

Candidate for the degree of Master of Science*

Committee Chair

Dr. Z. Meral Ozsoyoglu

Committee Member

Dr. Mehmet Koyuturk

Committee member

Dr. Michael Lewicki

Date of Defense

October 22, 2015

*We also certify that written approval has been obtained for any proprietary material contained therein
# Table of Contents

List of Tables .................................................................................................................. vi

List of Figures .................................................................................................................. vii

Acknowledgement ......................................................................................................... ix

Abstract ......................................................................................................................... x

1. Introduction ............................................................................................................... 1

2. RDF and RDF querying with graph template ......................................................... 4

   2.1 RDF database .................................................................................................... 4

   2.2 Querying with graph templates .......................................................................... 7

      2.2.1 Definitions ................................................................................................. 8

      2.2.2 Indexes ....................................................................................................... 9

      2.2.3 Query Framework Process ...................................................................... 9

   2.3 Dataset and Queries Used in Examples: ......................................................... 10

      2.3.1 Dataset ..................................................................................................... 10

      2.3.2 List of Queries ......................................................................................... 11

3. Exploratory Querying ............................................................................................... 13

   3.1 Exploratory Query Models ............................................................................... 14

   3.2 Exploratory Query Methods ............................................................................. 15

   3.3 Exploratory querying feature in our interface ................................................. 15

      3.3.1 Exploratory Browsing Query ................................................................. 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Focused Searching query</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Advantages in using GBE interface for exploratory query</td>
<td>19</td>
</tr>
<tr>
<td>4.</td>
<td>Exploratory Graph Query Interface Design</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Interface Features</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Query Formulation</td>
<td>22</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Execution of Query</td>
<td>23</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Query Reuse</td>
<td>24</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Displaying Query Results</td>
<td>24</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Description of Query</td>
<td>25</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Usage of Namespace Prefix</td>
<td>25</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Multi-Thread Feature</td>
<td>28</td>
</tr>
<tr>
<td>4.1.8</td>
<td>Query Status Display</td>
<td>28</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Other Supplementary Features</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation details</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1</td>
<td>General Implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Query Template Drawing</td>
<td>31</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Internal Structures</td>
<td>32</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Querying Process</td>
<td>33</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Displaying Query Results</td>
<td>36</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Implementations details for additional features</td>
<td>39</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1 Example Triples in LUBM ................................................................. 11

Table 4.1 Supplementary Features ................................................................. 29

Table 4.2 Drawing Class Functions ................................................................. 32

Table 4.3 Internal Structure Class and Property ............................................. 33

Table 4.4 Input Variable Types ......................................................................... 34

Table 4.5 VertexLabel ....................................................................................... 35

Table 4.6 EdgeLabel ......................................................................................... 35

Table 4.7 CompPathList ................................................................................... 36

Table 4.8 Graduate Student Result IDLabel .................................................... 36

Table 4.9 Popular Namespace Prefixes ............................................................ 40

Table 4.10 Implementation of other functions ................................................ 41
List of Figures

Figure 1.1 SPARQL query ................................................................. 1

Figure 2.1 Linked Data Diagram ...................................................... 5

Figure 2.2 GBE query template with matching results ....................... 7

Figure 2.3 GBE query process .......................................................... 9

Figure 3.1 Exploratory Browsing Query Q1 ....................................... 16

Figure 3.2 Result set of different types of focused search refinement ....... 17

Figure 3.3 Focused Searching Query Q2 .......................................... 18

Figure 4.1 GBE Interface ................................................................. 21

Figure 4.2 Q4 with Full URLs ........................................................... 26

Figure 4.3 Using namespace prefix in Q4 ........................................ 27

Figure 4.4 Status Bar ....................................................................... 29

Figure 4.5 Q5 .................................................................................. 37

Figure 5.1 Query Process Diagram .................................................. 43

Figure 5.2 Q6 .................................................................................. 45

Figure 5.3 Query description for Q6 ................................................ 47

Figure 5.4 Results for Q6 ................................................................. 47

Figure 5.5 Browsing Query Q1 ......................................................... 49
Acknowledgement

I would like to express a special thank you for my advisor, Prof. Meral Ozsoyoglu. When I was hesitating about the future career and research direction in my first year at CWRU, she gave me guidance and leaded me to the decision of pursuing a Master of Science degree in computer science. Also, I would not be able to finish the thesis work without her encouragement, guidance and patience.

I would also like to thank Prof. Mehmet Koyuturk and Prof. Michael Lewicki to be in my thesis committee. I value the time you spent in reading my thesis and attending my thesis defense, along with the valuable questions you pointed out that helped me improve the thesis.

Along the process of the thesis work, Shi Qiao has given me a lot of guidance and we collaborated to make the GBE interface integrated well with the GBE query engine. He also gave me suggestions in daily life which greatly broadened my horizon. I really value the time spent with you.

At last, I want to express my appreciation for my friends and family, for their support for me during my graduate study. There’re hard times and you would be here to help me get through these, and there’re pleasant times and you would be here to share the joy with me. I cannot imagine a life without you.
Graph by Example: an Exploratory Graph Query Interface for RDF Databases

Abstract

by

CHENG YANG

Query interface is an important tool in accessing graph databases. Traditional text-based query interfaces only provide access to databases through text queries and results, without presenting the internal graph structure. In this thesis we present a graphical query interface for exploratory querying of graph structured data, called Graph by Example (GBE). Our interface introduces a Draw and Play feature, which enables users to query graph structured data intuitively using graph templates. GBE interface also provides exploratory querying features facilitating the editing and reuse of previous queries so that users can reformulate their queries and resubmit based on the results of the previous queries. These features makes the interface simpler and easier to use compared to traditional text based query interfaces. We implement this query interface utilizing an RDF querying framework in a previous research. We also demonstrate the interface’s functions and features through examples in real world databases and benchmarks.
1. Introduction

RDF databases have been widely used to represent the growing semantic web data due to their advantages in dealing with complicated and associative data sets. They are graph structured yet still have standard schemas defined, taking advantage of schema-less graph databases and standard schemas. Various data domains have applied RDF databases to store their data over the semantic web, including bioinformatics [1], image, videos, computer science [2] and social networks [3]. This has created a new topic: how to retrieve data and run queries in RDF databases. SPARQL came out as a W3C standard language intending to retrieve and manipulate RDF data. Figure 1.1 shows a typical SPARQL query [4]. The query wants to find the cancers that are associated with at least one protein that is also associated with “Breast Cancer” in the UniProt database [5], which is a hub of protein information integrated from biomedical datasets.

```sparql
SELECT 
?Cancer
WHERE
{
?Protein  <http://purl.uniprot.org/core/Protein>
   a <http://purl.uniprot.org/core/Annotation>
   ^annotation "disease_annotation1"
   ^annotation "disease_annotation2"
   disease_annotation1   ?disease1
   disease_annotation2   ?disease2
?disease1   "Breast Cancer"
?disease2   "Cancer"
FILTER regex(?Cancer, ".*Cancer.*")
}
```

*Figure 1.1 SPARQL query*

It can be seen that the syntax of SPARQL query is very complicated, especially for inexperienced users, and it requires extra efforts to write or understand such queries. In addition, the structure of RDF data is often very complex and hard to retain in the mind, and researchers often need to draw a draft graph to represent their query. These problems
call for new methods for dealing with RDF data, which can present queries more clearly and without having to use text based query language. In this thesis, we present a graphical query interface that utilizes graph template matching as the query method; this provides a good solution to the problems stated above.

Our query interface, Graph by Example (GBE), uses graph templates as the basic query units. We utilize the query engine presented in [6], which provides the framework for querying with graph template matching. We introduce the Draw and Play feature in this interface which allows users to draw the graph query template from scratch or by editing an existing query template; run the query directly with the graph template; and get the results displayed in the form of subgraphs of the database matching the input query graph template. Utilizing graphs reduces the difficulty in understanding the query. We further simplify it by attaching a description to each graph query, where users can also enter the description for later references.

By facilitating the editing and reuse of previous graph templates, and running them as new queries, our graphical query interface also lends itself to exploratory querying. That is, especially for scientific databases, researchers may like to revise their queries based on the results of their previous queries as they explore the data, and be more specific about what they need to query in the database [7]. With exploratory querying, they can use a general query at first and refine the query utilizing the query results. In this way they are more likely to formulate their desired query and achieve better results. In our interface users have access to the intermediate results and can run exploratory queries iteratively until the desired results are achieved. The GBE query interface has proven to be simple and useful
in our test using the LUBM benchmark [8], which will be illustrated with detailed examples in later chapters.

The main contribution of this thesis is the development of an exploratory graph query interface for RDF data. This makes use of the graph feature of the RDF databases and makes life easier for RDF database users. The main features include:

- Draw graph query templates using vertex, edge and pathEdge constructs from scratch or by reusing previous graph templates with or without editing
- Query RDF databases directly using graph templates.
- Display query results as graphs with navigation functionality.

The structure of this thesis is as follows: Chapter 2 presents a review of RDF databases and RDF querying with graph templates. Chapter 3 presents exploratory querying features. Chapter 4 introduces the design of the GBE graph query interface. Chapter 5 focuses on the implementation of the interface. Chapter 6 presents related work while Chapter 7 looks into future work and Chapter 8 gives conclusions on our work.
2. RDF and RDF querying with graph template

2.1 RDF database

Resource Description Framework (RDF) is a W3C standard model for storing and transmitting web data. It extends the linking structure of the web and forms a directed, labeled graph called RDF graph. It is now a commonly used method to model linked data and information implemented as web resources [9].

An RDF graph consists of a set of triples (s, p, o), which represent subject, predicate and object. Each triple forms a directed edge from one vertex to another. The subject, object and predicate can use unique identifiers called Uniform Resource Identifiers (URI) to uniquely refer to entities, while the object can also use literals. For example, in order to store the fact “Professor X acquired his PhD degree from Y University”, a triple (s, p, o) is used where s represents Professor X, p represents the relationship “PhD graduate from” and o represents “Y University”.

In the web databases, these entities and relationships are put in a standard way as web resources and have their unique URIs. The collection of URIs is called RDF vocabulary, which often begins with a common substring called namespace URI. The namespace prefix is often associated with certain namespace URI and can serve to assist readability. For example, if in the university namespace, Professor X has a unique URI of http://www.w3c.org/ontology/university/professorX, in which case the “http://.../university/” would be the namespace URI, and a namespace prefix uni is associated to it, making the abbreviated version of URI to be uni:professorX. Note that this is just to provide convenience to users, but is not a formal part of the RDF data model.
The usage of these standard web resources as URIs acts like schema for RDF databases, making it more standard compared to other NoSQL databases, yet still preserving its flexibility. This enables RDF to deal with all kinds of data sources, ranging from structured to semi-structured and even unstructured data, and the size of the datasets keeps growing. Figure 2.1 shows a collection of some current linked RDF databases [10].

Figure 2.1 Linked Data Diagram
Each node in the linked data diagram of Figure 2.1, corresponds to a dataset. The color of the node represents the domain of the dataset, which includes publications, life sciences, social networking, geographic, government, media, user-generated content, and linguistics and cross-domain datasets. The size of the node corresponds to the number of triples in the dataset, with the largest datasets consisted of more than one billion triples in the dataset. The arrow indicates that the datasets are linked, and the thickness of the arrow corresponds to the number of links between the datasets. Datasets include DBpedia, which contains the extracted structured information from Wikipedia [11], and FOAF, which contains people-related terms that can be used in structured data [3]. The datasets are highly connected to others and the size keeps growing dramatically.

Due to the size and complexity of RDF databases, the querying them involves several challenges, including, easy to use expressive query languages, user friendly query interfaces, efficient and scalable query evaluation. SPARQL emerged to be the standard query language for RDF databases, and researchers have been conducting work in improving SPARQL query performance [12]. However, although SPARQL makes use of the graph pattern of the RDF data structure, it uses text queries to represent the relation as well as the subjects and objects. It is complicated in its nature and requires extensive knowledge in SPARQL query language in order to write and understand queries. This is error-prone and reduces efficiency in researches using RDF databases. Another problem with SPARQL is that it fails to present the graph structure directly to users, thus making it hard to keep all of the relations in mind.
In order to query RDF data without having the disadvantages of SPARQL, some have focused on building new interfaces and frameworks. However, although trying to make use of the graph structure, most works are still limited to the use of text query.

### 2.2 Querying with graph templates

Querying RDF using graph query templates also draws attention from researches, and there are several results reported [4] [13] [14] [15]. In this section, we will provide a brief introduction to the GBE graph query framework that is used in our interface, which is based on graph template matching method for querying RDF data [6].

**Figure 2.2 GBE query template with matching results**

Figure 2.2 shows the idea of GBE graph query framework [6]. Figure 2.2(a) displays part of the RDF graph in DBLP dataset, which contains information on computer science publications listed in the DBLP Computer Science Bibliography. Figure 2.2(b) shows a query template example on the RDF graph. It tries to find the title of a paper authored by...
“Philip S. Yu”, published in VLDB after year 2000, and has a connection within 4 hops to a paper authored by “Jiawei Han” whose title begins with “Efficient and Effective” and is published in VLDB before year 2000. The query engine then matches the query template example with the RDF graph of DBLP and finds two matches, which are given in Figure 2.2(c-d).

The first result in Figure 2.2(c) is a paper named “A Framework for Clustering Evolving Data Streams”. It directly cites the “Jiawei Han” paper so it satisfy the requirement of within 4 hops. The second result shown in Figure 2.2(d) is a paper named “An Automated System for Web Portal Personalization”. Although it doesn’t directly cites the “Jiawei Han” paper, it cites another paper that does it, which makes the distance between the two papers to be 2, also satisfies the 4 hop requirement. A brief explanation of how the query engine works is presented below.

2.2.1 Definitions

RDF graph is a directed graph $G = \{V, E, I, f\}$ where $V$ is the vertex set, $E$ is the predicate set, $I$ is the label set and $f$ denotes mapping from vertices/edges to labels.

Connection edge ($\leftrightarrow$) represents a path between two vertices. It may consist of a number of edges and there’s a parameter $E$ that denotes the number of edges that the path can have.

A GBE Query Template is a small RDF graph that represents the query. There are partial keywords or missing keywords in its labels, and the query’s purpose is to find the subgraphs from the dataset matching the query template as query result.
That is, graph template matching process finds all the subgraphs of the dataset graph G that satisfy both the graph structure and the label constraints of query template Q based on graph isomorphism.

2.2.2 Indexes

There are two indexes used in the framework utilized in this interface [6]. The first one is IDMap, which maps vertices with their labels to numerical IDs. The second one is Neighborhood Interval (NI) index, which contains distance, interval, number of neighbor nodes and neighbor nodes IDs. Edge direction is also considered in building the NI index. The use of these two indexes enhances the overall performance of the framework. The process of building the two indexes from the database is explained in [6].

2.2.3 Query Framework Process

The process of GBE query framework is shown in Figure 2.3 [6]. It consists of several steps. First, it takes in the query template and divides the query to several separate components without connection edge. This is called query decomposition. Next, it tries to find matching results for each separate component using IDmap index, and performs Neighborhood check to prune the candidates. Then, it further dismantles each component to form 1 level D-trees, generates matching results for the small D-trees, and joins them to

![Figure 2.3 GBE query process](image-url)
find candidates for the components. Last, it checks the connectivity of the components using the NI index and eliminates the candidates that don’t satisfy the requirements. After these steps, only the results that satisfy all of the requirement remain, and are output out of the query framework.

The GBE query framework has proven useful in dealing with RDF datasets of different types and different sizes. Our interface utilizes this query framework and can be applied to various RDF databases to simplify the query process.

2.3 Dataset and Queries Used in Examples:

In this section we present the dataset and queries that are used in the examples throughout the rest of this thesis.

2.3.1 Dataset

In the examples for the rest of this thesis, we use the Lehigh University Benchmark (LUBM) [8]. The Lehigh University Benchmark is developed to evaluate the performance of extensional queries over a large data set that commits to a single realistic ontology. It provides a UBA (Univ-Bench Artificial data generator) that features random and repeatable data generation in university domain in OWL semantics. This makes the data in LUBM to be very standard without unexpected exceptions. Typical data entries in LUBM include literals, URIs and URLs. There are a total of 1316700 triples in the sample dataset we use, with the size of the whole dataset to be 863.38 MB. Table 2.1 shows some triples in LUBM, which include university, professor, student, course and publication. The example queries use entries similar to the ones in the table.
<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.Department2.University1.edu/FullProfessor0">http://www.Department2.University1.edu/FullProfessor0</a></td>
<td>#mastersDegreeFrom</td>
<td><a href="http://www.University186.edu">http://www.University186.edu</a></td>
</tr>
<tr>
<td><a href="http://www.Department5.University1.edu/GraduateStudent112">http://www.Department5.University1.edu/GraduateStudent112</a></td>
<td>#advisor</td>
<td><a href="http://www.Department5.University1.edu/AssociateProfessor2">http://www.Department5.University1.edu/AssociateProfessor2</a></td>
</tr>
<tr>
<td><a href="http://www.Department5.University1.edu/Course8">http://www.Department5.University1.edu/Course8</a></td>
<td>#name</td>
<td>&quot;Course8&quot;</td>
</tr>
<tr>
<td><a href="http://www.Department2.University1.edu/FullProfessor0/Publication2">http://www.Department2.University1.edu/FullProfessor0/Publication2</a></td>
<td>#publicationAuthor</td>
<td><a href="http://www.Department2.University1.edu/FullProfessor0">http://www.Department2.University1.edu/FullProfessor0</a></td>
</tr>
</tbody>
</table>

*Table 2.1 Example Triples in LUBM*

### 2.3.2 List of Queries

**Q1**: Find all the URL entries that are connected to University1.

**Q2**: Find the people who attend the same University as FullProfessor1.

**Q3**: Find the graduate students who got the undergraduate degree from University1 and are doing research about Database Optimization.
**Q4:** Find the graduate students in Department2 of University1 who have published at least one publication.

**Q5:** Find the graduate student whose advisor is FullProfessor2 and who is the TA for Course51.

**Q6:** Find all the courses that are taken by graduate students who have the same Full Professor advisor with GraduateStudent46.

**Q7:** Find the name of the publication in Department2 of University1 that is on the research topic "Research20" and is connected to University845.
3. Exploratory Querying

The semantic web has been using RDF databases to store massive data resources. This provides users flexibility and efficiency with storing and transmitting data. However, the increased flexibility has also made it harder for users to understand the structure of the data, due to the lack of knowledge, and now the querying of certain data is much harder than in traditional databases. Under these circumstances, having an exploratory query with a “trial-and-error” feature would be very useful [16], because users can make their first queries as trial and refine their queries when they find problems in the results. In addition to semantic web querying, exploratory query has also received much interest in other research communities, such as human-computer interaction [17], social networks, medicine [18], bioinformatics, and information retrieval, where it is often referred to as “exploratory search”.

Exploratory query describes an open-ended query between end users and databases, where users are primarily interested in learning more about the data itself and finding interesting topics, and refining their queries based on the results of their previous queries. There are several common scenarios in exploratory query, which includes [7]:

a) Users not being familiar with the information in the database

b) Users experiencing uncertainty of how they can achieve their goal

c) Users being confused about their goals in the first place

The uncertainties in these scenarios call for a special form of human-in-the-loop query application, which can help in formulating the final query. A typical application contains
the following three steps [19]: 1) Processing initial queries, 2) Reviewing returned results and 3) Refining queries accordingly. A good query interface should be able to carry out these steps and assist users in finding interesting results.

### 3.1 Exploratory Query Models

Based on the target of the querying, there’re two types of models for exploratory querying [7]. The first one is exploratory browsing and the second one is focused searching. A good exploratory query interface should be able to support both models effectively.

Exploratory browsing refers to circumstances where users are more interested in browsing types of information in the database. This can lead them to a general understanding of certain collections and improve knowledge about the data in the databases. Although this might not be as important in specific querying problems, this will help the researchers to conduct better queries for specific problems.

Focused searching is the process of querying when users have specific goals and certain knowledge about the database they are querying. This has been the most popular exploratory query model as most queries are targeted towards certain results. Focused searching requires more information from query results in order to make refinements towards certain directions more effectively.

These two models represent a general process of exploratory search over databases. Initially, users are not clear of the structure and data in the database, and need exploratory browsing to learn and discover useful information. As their knowledge over such databases grows, there is less uncertainty. Thus, focused searching plays an important role in helping users to have better refinements and find interesting topics.
3.2 Exploratory Query Methods

An exploratory query can be carried out in several different forms based on user needs and application. Query Reformulation [20] lets the user propose reformulations themselves on the initial query and gives the user some assistance in doing it. Query Prediction [21] provides more assistance than the basic query reformulation and gives the most possible queries that the user is going to ask based on previous query contents. Query Preview [22], however, has a different approach than query prediction. Query preview uses both the contents of previous similar queries and the results of those queries as materialized views to provide a preview of the query results.

Different methods are useful in different conditions. In cases where initial query results contain enough information, it is sufficient to provide the query results for performing query reformulation. On the other hand, when the query results are too complicated to understand and composing new query is a problem, it is better to have the query prediction and query preview for better assistance in doing exploratory querying.

3.3 Exploratory querying feature in our interface

The GBE query interface targets at querying of RDF databases with graph templates, making its integration with exploratory query different from traditional text-based queries. It provides all the results as graphs in the form of sub-graphs of the entire RDF database, which makes the results easier for users to understand and also shows, to some extent, the structure of the RDF database. This feature makes it easy to adapt to the exploratory query.
The following part gives two examples for the two models of exploratory querying. The first one is an exploratory browsing query, and the second one is a focused search query.

3.3.1 Exploratory Browsing Query

Consider the following query Q1:

**Q1**: Find all the URL entries that are connected to University1.

This is a typical exploratory browsing query, which doesn’t put much constraint on the query itself, but focuses on browsing the data structure in the dataset.

In traditional databases like Microsoft SQL Server, in order to run this query, users need to master special query languages and write the query. On the other hand, in the GBE interface, the query would be able to be simply represented by a graph with two vertices and one edge connecting them, one with the university1 URL and the other with general URL prefix and the query mark “*”, as shown in Figure 3.1. Users would be able to draw the graph simply and query directly with the graph.

![Figure 3.1 Exploratory Browsing Query Q1](image)

**Figure 3.1 Exploratory Browsing Query Q1**

After the query is executed, the interface would get the results that match the input query graph, and display them also as graphs, which have the same structure as the input query graph. Users can then refine the original query graph based on information revealed from the results and continue to do more browsing, such as by adjusting certain attributes. The whole exploratory browsing process in GBE interface don’t require users to master
complicated query languages, and is very helpful for quick browsing for information on the dataset. Section 5.3.2 in Chapter 5 shows a running example of an exploratory browsing query with GBE interface together with the results and refinements, which demonstrates its simplicity.

3.3.2 Focused Searching query

When it comes to focused searching, the graph feature can have even more advantages, as it shows the complicated graph structure directly to the user. There are three types of refinements in Focused Searching, classified by the relationship between original result set and refined query’s results set. The different relationships shown in Figure 3.2.

![Figure 3.2 Result set of different types of focused search refinement](image)

Compared to the original query, the refinement can be more relaxed when the ideal results are not in the result set; stricter when there are too many results; or stricter on some side
while more relaxed in others when the ideal results are overlapping with the result set. These types of refinements help the users to get more satisfied query results.

Consider the following query Q2:

**Q2: Find the people who attends the same University as FullProfessor1.**

Unlike the exploratory query Q1, which focuses more on the data structure side, Q2 tries to find some people that satisfy some constraints. This query may arise from the direct user interest in the people attending the same university as FullProfessor1, or from general interest in people related to FullProfessor1. In the second situation, refinements would be needed to adapt to more specific interests by the users after viewing the results of the initial query.

![Figure 3.3 Focused Searching Query Q2](image)

Figure 3.3 shows the graph template query for Q2 in GBE interface, with the right vertex being the FullProfessor1, the middle vertex being a university with query mark and the third vertex a URL prefix with query mark. The logic is to find all the other people that graduates from the same university as FullProfessor1. It is very easy for users to draw the query template and run the query. After execution of the query, the results will be displayed in graphs with the same structure of the input graph template, with the query marks replaced by the matching results.

Based on the new interest inspired by the query results of the original query, users can refine the original query to form new queries. They can make modifications on the middle
vertex to change the shared vertex to other things, or they can modify on the right vertex to add constraint to it. They may even add new vertices and delete some initial vertices to construct a query with new structure.

In Section 5.3.3 of Chapter 5, a detailed demonstration of the usage of GBE interface for a focused searching query with results and refinements is presented.

3.3.3 Advantages in using GBE interface for exploratory query

There are several advantages of our graph query interface in exploratory querying.

First, we provide the graph view of results, which let users have an intuitive impression of the results and helps them to find interesting results more easily.

Second, our drawing canvas allows users to modify directly on the query template, which is very convenient in the exploratory querying process. This eliminates the need to rewrite the query, and the graph template is also easier to modify as compared to traditional text-based query languages.

Next, we can provide intermediate results in different tabs while displaying different query results together in different tabs. So after users make refinements on the query based on the results and rerun the query, they have access to not only the current query results, but also the previous results that they referred to when making the refinements. This can help when they need to access previous results to make further refinements. In addition to displaying multiple queries’ results in different tabs, our interface also provides users with additional functionality by allowing them to directly print query results or save results as query files or as pictures for later reference.
Finally, we also provide an interface for many databases. Unlike typical interfaces that are integrated with certain databases, we let the users choose the database to load to the interface to run their query. The range of databases that can be loaded is not limited, and users can download different databases from different sources, preprocess using method in GBE query framework and load them into the interface.

Exploratory querying allows users to explore the database during the querying process and to refine their queries based on the answers of their previous queries which may not be precise and focused initially. Our interface provides exploratory querying features and brings many unique advantages. We believe, GBE query interface have potential to improve the efficiency and productivity of researchers using exploratory querying.
4. Exploratory Graph Query Interface Design

In this chapter, we describe the design of the Graph By Example (GBE) exploratory graph query interface. We begin with the features the interface offers, followed by details concerning implementation.

4.1 Interface Features

As a graph query interface, the GBE interface provides functions to draw query templates and run queries. The three main features are query formulation, query reuse and displaying query results. There are also many supplementary features in the interface, aiming to further reduce work amount and improve efficiency. These features include description of

Figure 4.1 GBE Interface
query, usage of URI prefix, multi-process query, query status display and other minor features. Figure 4.1 shows a screenshot of the query interface with brief explanation of some important parts. The query that is shown in the interface is Q3:

**Q3**: Find the graduate students who got the undergraduate degree from University1 and are doing research about Database Optimization.

We will explain the interface and the query in detail in the following sections. For further documentation, please refer to the appendix.

### 4.1.1 Query Formulation

In the query formulation stage, the query template canvas part provides the functionality to draw the graph query, as is shown in the bottom left of Figure 4.1. Before creating the query, the users first need to choose the database to query using the toolbar. The database chosen will be shown on the status bar. After the database is chosen, users can create a new query associated with the database, and begin to draw the query template.

The three main elements provided to draw the query template are vertex, edge, and pathEdge, which are also included in the graph elements part of the toolbar. The first element, vertex, corresponds to the subjects and objects in RDF data. The second element, edge, represents the predicate, and the third is the pathEdge that represents a path in the RDF graph, and may be composed of several directed edges. PathEdges may also have distance constraints indicating the limit on the number of edges in the pathEdge. All the elements have their labels, which show the contents of the elements. For a vertex or an edge, the label might be a URI or a literal, while the labels for a pathEdge should be distance constraints. The query information is provided through the labels of the vertices,
edges and pathEdges, with the full labels providing complete information and the vertex labels marking query targets with a query mark “*”.

Users can also use some buttons like cut/copy/paste in the toolbar, shown in the top left of Figure 4.1, to modify the query before it is completed. The query template in Figure 4.1 shows a complete graph query, Q3, which contains four vertices, two edges and one pathEdge. The two edges represent two attribute links of the vertex in the middle, showing that its type is GraduateStudent and this student graduates from University1. The pathEdge connects the middle node and the bottom node, with a distance constraint of “<3”, showing that the middle node is connected to “Database Optimization” with less than three hops. In the LUBM database, graduate students do not have an attribute “researchArea”, and they are connected to the research areas through their advisors. With the usage of pathEdge, the internal nodes on advisors can be skipped to simplify query process.

In addition to the simple drawing applications, where vertices and edges are just figures on the canvas, the elements on the GBE interface’s query template canvas have internal data structure and internal relationships. The data structure for query template contains all the information needed for the query engine to run the query and get the results, which allows users to query directly with the query templates. Users can also add the description of the query in the query description part shown at the bottom right of Figure 4.1.

4.1.2 Execution of Query

After drawing a graph query is completed, users first need to load the database that they want to query in the interface. This is done through selecting a database from the database dropdown menu and clicking the load database button on the Toolbar. Then, users need to
click the run query button and the interface will execute the graph query in the loaded database directly with the integrated query framework. At the current stage, the integrated query framework is the GBE framework, which supports the graph template matching [6].

Upon the click of the run query button, the interface will generate the data needed for the GBE query framework and send it to the query engine. After the background query is completed, the query engine will notify the interface and send the results to the interface, which will then display the query results.

4.1.3 Query Reuse

The reusability of graph template query is an important feature in our query interface design. After a query template is created, users can store them in association with the database being queried. The save/print/open buttons in the toolbar allow users to store and reuse the query. In addition, the query explorer component, shown in the right of Figure 4.1, provides the user access to the view and access of all the stored queries. All the queries in a database are stored together in one file, so it’s very convenient to access all the queries in one database.

When users need to formulate a new query, in addition to being able to create the query and draw all the elements from scratch, they can choose from the stored query templates in the query explorer and modify them to fit their needs. There’s also another way to reuse the queries. The interface keeps all the opened queries in the query page tab, so users can also go through different queries by clicking on different tabs in the query page tab.

4.1.4 Displaying Query Results
After the querying process, the matching results are returned from the query engine to the interface and are ready for display. The displaying of query results is also different from traditional interfaces. We display the results as graph instances from the database matching the given query template on a new canvas tab. The query results are in exactly the same structure as the query template, with the query mark “*” replaced by the full labels in the results.

In addition, users can choose the number of results to be displayed on one page. Currently, we support two to ten results being shown per page based on the result count of the queries. Users can choose the number per page and navigate all the results through the buttons in the result navigation part at top of Figure 4.1. As the query result are treated in the same way as the query template, users can modify the query result directly or, based on the query results, create a new query template.

4.1.5 Description of Query

To better assist the user with writing and understanding queries, we attach a description section to each query, as is shown in the bottom right part of Figure 4.1. Users can select to write anything that is related to the query in this section. Every time the users select a new query, the description section will be updated with the description associated with the new query.

4.1.6 Usage of Namespace Prefix

In RDF databases, the Universal Resource Identifier (URI) is often very complicated. URIs within the same ontology space often start with the same namespace URI, which is a prefix string for the full URI. We can use simple namespace prefix to represent these namespace
URIs, to simplify the work in writing the URIs. For example, “rdf” is a W3C standard namespace prefix that represents namespace URI “<http://www.w3.org/1999/02/22-rdf-syntax-ns#>”, so “rdf: type” is the short version for the full URI “<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>”.

Using of namespace prefix can greatly reduce the work needed in creating, understanding and modifying queries. Consider query Q4, which is a query about student and publication relationships in LUBM. Figure 4.2 and Figure 4.3 provide a comparison of the graph template versions of Q4 with full URL and with namespace prefix, which demonstrates the simplicity introduced in using namespace prefix.

**Q4: Find the graduate students in Department2 of University1 who have published at least one publication**

![Figure 4.2 Q4 with Full URLs](image)
Figure 4.2 provides the graph template for Q4 with full URLs. Two of the five vertex labels, together with all four edge labels are using complicated URLs. Users need to either remember clearly the complicated URLs or refer to the URL dictionaries during the query process. This requires additional work and can sometimes bring errors to the query when the full URLs are not entered correctly. Also, the long URL labels also brings problems to the layout of the query, which requires some additional spaces for the long edge labels to display properly.

Vertex Labels:

Edge Labels:
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>: #
<http://www.w3.org/1999/01/22-rdf-syntax-ns#type>: #type

(a) Namespace prefix table used for Q4

(b) Q4 graph template with namespace prefix

*Figure 4.3 Using namespace prefix in Q4*
The usage of namespace prefix in the query gives a good solution to the problems mentioned above. Figure 4.3(a) shows the namespace prefixes that can be used in this query, and Figure 4.3(b) shows the graph template query for Q4 with the namespace prefixes used. We can see that as compared to using the full URIs, the graph template query with namespace prefixes are more concise and requires less work. Namespace prefix is also less error-prone, especially when we have very complicated queries. Namespace prefix is most useful for representing RDF schema, as RDF data has many popular schema defined using URIs. However, as the GBE query engine doesn’t use the predicate’s information for the query, so in our interface the namespace prefix for edges are for demonstration usage only. In practical we only use namespace prefix for the ontology URIs.

4.1.7 Multi-Thread Feature

One key feature in current query interfaces is the support for multi-thread query, which prevents the program from becoming stuck and unresponsive. Our interface also uses the multi-thread feature to best serve the user. After testing, there are two things in our program that consumes time and is independent of other operations. The first one is loading the database and the second one is running query. We put the loading process and the querying process in another thread, so the users can manipulate the main interface while running these background tasks.

4.1.8 Query Status Display

Another feature we provide is the display of interface status and query status. The status bar is located at the bottom of the interface, as is shown in Figure 4.1. A more detailed
version of the status bar is shown in Figure 4.4, which shows the status bar after loading the Lehigh University Benchmark and the execution of a query in LUBM.

| Query | 06 Find Courses using full URL | Query Time | 410 ms | Result Count | 2 | Lehigh University Benchmark |

**Figure 4.4 Status Bar**

The right side of the status bar shows the name of the database that is currently loaded in the interface, which is the Lehigh University Benchmark. The left side of the status bar shows the status of the query after running, which contains the name of the query and information about the result count and the running time. This is useful for the analysis of query and performance monitoring.

### 4.1.9 Other Supplementary Features

In addition to the features listed in the previous sections, we will also discuss several supplementary features, which help to further simplify query work for the interface. These features are listed in Table 4.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoom</td>
<td>Allow user to zoom query to a maximum of 400%, a minimum of 20%</td>
</tr>
<tr>
<td>Alignment</td>
<td>Allow users to line up the vertices to make a better outlook</td>
</tr>
<tr>
<td>Font</td>
<td>Let users select the font, size, color and other options for the label text</td>
</tr>
</tbody>
</table>

**Table 4.1 Supplementary Features**

### 4.2 Implementation details

In this section, we talk about the implementation details of the GBE interface, including programming language, hardware, frameworks and detailed explanations of several important class functions and structures.
This section introduces the implementation details of some important functions and features in a concise way. For more detailed implementation of the interface, refer to the user manual, key functions and code structure in the Appendix.

4.2.1 General Implementation

We use C# with .Net 4.0 framework and Microsoft Visual Studio 2010 to develop the interface.

The main Graphical User Interface is a windows form application. We use the basic functions in NClass framework [23] as reference. NClass is a free open source tool to create UML class diagrams. We refer to the drawing functions and basic design in NClass and create custom drawing functions to fit our needs in graph querying. For the query engine, we use the GBE framework provided in [6]. This framework supports graph template matching.

We support all kinds of RDF databases in the interface. However, normal RDF databases would require some pre-processing to build the IDMap and Neighborhood Interval indexes described in section 2.2.2 of Chapter 2. For the detailed process of generating the required indexes from the RDF databases provided, refer to [6].

The interface is made up of four individual projects, each with its own functionality. The first one is the TemplateDesigner Project, which implements the interface’s template drawing-related functions. Next is the Core Project, which implements the internal structures of different elements in the interface including vertices, edges, pathEdges, query templates and database projects. The third one is the GUI project, which implements the main GUI and different sections on the GUI. It also integrates the other three projects and
provides the main interface functions. The last one is the Query engine, which comes from the GBE query framework and is integrated in the interface for the execution of query.

### 4.2.2 Query Template Drawing

Query template drawing functions are provided in the TemplateDesigner project. The canvas of the query template is provided by the canvas class, which contains a white canvas with the query name as tab and allows users to draw the three different kinds of template elements on it. There are three main classes for the drawing of query template, which are vertexShape, edgeConnection and pathConnection. VertexShape implements the operations on vertex like drawing, adding label, moving, resizing and selecting, while edgeConnection/pathConnection implements the drawing, label and layout of the edge/pathEdge.

When users click the vertex button on the toolbar, there will be a virtual vertex square following the cursor, allowing them access to any position on the canvas. The users can then select a position on the canvas and right click the mouse, which will draw the vertex at the specified location with empty label.

When the users click the edge or the pathEdge button on the toolbar, the interface will allow users to select the starting vertex and the end vertex to form a directed edge or pathEdge. The layout of the edge or pathEdge line will be automatic. Based on the layout of the staring vertex and the end vertex, the starting point and end point of the edge/line will be different with a total of 9 variations. In addition, when users move and resize the vertex, the edge or pathEdge line will change automatically based on the relative location of vertices to ensure a clear layout.
For the label of the vertex, users can input the labels by click the vertex’s label area or by double click the vertex. In the case of edge or pathEdge, the edit label dialog is opened when users double click on the center of the lines.

Related classes and some functions supported are shown in Table 4.2.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Property</th>
<th>Label</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canvas</td>
<td></td>
<td>Name of Tab</td>
<td>Drawing Canvas</td>
</tr>
<tr>
<td>VertexShape</td>
<td>Solid Rectangle</td>
<td>In Rectangle</td>
<td>Resize/Move</td>
</tr>
<tr>
<td>EdgeConnection</td>
<td>Solid Line with Arrowhead</td>
<td>Align with line</td>
<td>Reshape/Move</td>
</tr>
<tr>
<td>PathConnection</td>
<td>Dotted Line with Arrowhead</td>
<td>Align with line</td>
<td>Reshape/Move</td>
</tr>
</tbody>
</table>

*Table 4.2 Drawing Class functions*

### 4.2.3 Internal Structures

With the drawing functions, users are able to draw the graph template query on the canvas to represent RDF queries. However, the elements created are not simple shapes but also have internal structures, which are implemented in the Core project. The internal structure of the query template and template elements is used for the storing and execution of the query.

Some implementation relationships between the shapes and their internal structures are shown in Table 4.3. A template class stores the graph drawn on the canvas, with all the vertex, edge, and pathEdge information.

A vertex class contains information about the vertex itself, and the list of vertices and edge and pathEdges that is connected to this vertex. In addition, every vertex in one query template has a unique ID, which helps the query process.
Edge and pathEdge classes contain the information about themselves with the ID of the starting and ending vertex they connected. There’re some constraints on edge and pathEdge classes. First, they must have one starting vertex and an ending vertex. Without either vertex, the edge or pathEdge would not able to be formed. And if either vertex is deleted, the edge or pathEdge will also be deleted. Second, the label of the edge must be a string while the label of pathEdge must be a distance constraint in form of “<=I” or “<I” where “I” is a positive integer.

<table>
<thead>
<tr>
<th>Class</th>
<th>Shape Class</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template</td>
<td>Canvas</td>
<td>Query Template file; List of Vertices; List of Edges and PathEdges.</td>
</tr>
<tr>
<td>Vertex</td>
<td>VertexShape</td>
<td>List of Edges and PathEdges that link this Vertex; List of Vertices that is connected to this vertex. Vertex ID.</td>
</tr>
<tr>
<td>Edge</td>
<td>EdgeConnection</td>
<td>Starting Vertex ID; Ending Vertex ID.</td>
</tr>
<tr>
<td>Path</td>
<td>PathConnection</td>
<td>Starting Vertex ID; Ending Vertex ID.</td>
</tr>
</tbody>
</table>

*Table 4.3 Internal Structure Class and Property*

### 4.2.4 Querying Process

The querying process finds all the subgraphs in the RDF database graph that match the query template. Although RDF databases represent graphs, they are often stored as RDF triples. Thus, the basic query process is the matching of the vertices and edges connecting the vertices in the query template to those corresponding to the RDF graph of the dataset.

Upon clicking of the run query button, the interface first groups the query template into separate components, which is defined in Definition 4.1.
**Definition 4.1:** *Component:* A component is part of the graph template that contains vertices and edges. Each vertex contained in a component must be reachable through edges regardless of the direction and vertices in different components must not be reachable through edges. Different components are connected via pathEdges. Every query graph template contains at least one component (when all the vertices are reachable through edges) and may contain several components (when there’re pathEdges connecting different components to form the whole query template).

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexLabel</td>
<td>Dictionary that maps Vertex ID to its label</td>
</tr>
<tr>
<td>EdgeLabel</td>
<td>List of Edge ID triples, includes Starting vertex ID, ending vertex ID and label.</td>
</tr>
<tr>
<td>CompVertexList</td>
<td>Dictionary that maps component ID to the list of vertex IDs in it.</td>
</tr>
<tr>
<td>CompEdgeList</td>
<td>Dictionary that maps component ID to the list of edge IDs in it.</td>
</tr>
<tr>
<td>CompPathList</td>
<td>Starting component ID, ending component ID, Starting vertex ID, ending vertex ID, distance constraint</td>
</tr>
</tbody>
</table>

**Table 4.4 Input Variable Types**

Then, the execution of query is classified into two groups: The first is query without pathEdge and the second is query with pathEdge. Each of them have their own query execution plan and input variables.

In the first group, there’re two variables needed for the execution of the query, which includes a list of the EdgeID and a dictionary of the vertex’s ID to its Label. These two variables are executed through a RunQueryWithoutPath method which calls the functions in the GBE query engine.
The second execution plan needs three different variables for the execution of the query. The first one is CompVertexList, which is a dictionary that maps the component ID to the vertexID list it contains. The second one is the CompEdgeList, which is also a dictionary, mapping the component ID to the edgeID list it contains. The third one is CompPathList, which is a list of the CompPathIDs that contain the information regarding the pathEdges, including the starting and ending component IDs and vertex IDs.

Table 4.4 shows the five input variable types during the query process. With the first two variables for the querying without pathEdge and the last three variables for the querying with pathEdge. An example is given from the data generated from Q3 in Figure 4.1, Table 4.5 shows the VertexLabel generated from it. Table 4.6 shows the edgeLabel triple list generated and Table 4.7 shows the CompPathList generated from query Q3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>University1</td>
</tr>
<tr>
<td>1</td>
<td>GraduateStudent</td>
</tr>
<tr>
<td>2</td>
<td>&lt;http://*</td>
</tr>
<tr>
<td>3</td>
<td>“Database Optimization”</td>
</tr>
</tbody>
</table>

*Table 4.5 VertexLabel*

<table>
<thead>
<tr>
<th>Starting Vertex ID</th>
<th>Ending Vertex ID</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>#type</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>#graduateFrom</td>
</tr>
</tbody>
</table>

*Table 4.6 EdgeLabel*
<table>
<thead>
<tr>
<th>Starting Comp ID</th>
<th>Ending Comp ID</th>
<th>Starting Vertex ID</th>
<th>Ending Vertex ID</th>
<th>Distance Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>&lt;3</td>
</tr>
</tbody>
</table>

**Table 4.7 CompPathList**

When the users click the query button, all the variables are refreshed to make sure that they are up to date. Then the variables are sent to GBE framework based on the type of the query (with or without pathEdge) for the execution of the query. The returned results are stored in a list of IDLabel table, with all query marks “*” replaced by the matching vertex’s label.

### 4.2.5 Displaying Query Results

As mentioned in the last section, the query results returned from the query engine is a list of IDLabel data tables. Each data table consists of the IDs and their corresponding matching labels. An example of the returning IDLabel data table is given in Table 4.8. In this table, we can see that the “*” tickers in Table 4.5 are replaced by the complete labels.

<table>
<thead>
<tr>
<th>ID</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>University1</td>
</tr>
<tr>
<td>1</td>
<td>GraduateStudent</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.Department3.University7.edu/GraduateStudent52">http://www.Department3.University7.edu/GraduateStudent52</a></td>
</tr>
<tr>
<td>3</td>
<td>“Database Optimization”</td>
</tr>
</tbody>
</table>

**Table 4.8 Graduate Student Result IDLabel**

To display the result graphically, we use the graph structure of the query template and replace each label with the result’s labels. Figure 4.4 on the next page is an example of a query Q5 with both query and query result shown.
**Q5**: Find the graduate student whose advisor is FullProfessor2 and who is the TA for Course51.

![Figure 4.5(a) Screenshot for query Q5](image)

**Figure 4.5(a) Screenshot for query Q5**

![Figure 4.5(b) Screenshot for result of query Q5](image)

**Figure 4.5(b) Screenshot for result of query Q5**

Figure 4.5(a) shows the screenshot of query template for query Q5, which is consisted of four vertices and three edges. The query name and query target are shown in red rectangles,
with the query name shown as a tab in the tab bar and query target vertex labeled with query mark “*”.

Figure 4.5(b) shows the screenshot of query result for query Q5. The differences between query template and the result is marked in red rectangle. For the query result’s name, the interface creates a new tab with name as the query name with “result” to indicate that this is the result for the query. For the query vertex, it replaces the query mark “*” with the matching result in the database. And in this case, it is “GraduateStudent46” that replaces the “*”. Another section in the result is the status bar, which shows the status of the query and the query result, including query name, query time and query result count. In the example we can see the query name is “Find graduate student”, the query time is 294ms and the query result count is 1.

As is mentioned earlier, we create a new tab with the name “X result” to indicate that this tab shows the result for the query with Name X. In many circumstances, there are multiple results from the query, so we support the display of multiple results on one page. We do this by creating multiple graph instances with the query template’s structure and putting them into two columns, where each cell will receive the information of each result and be updated with that result’s information. When navigating through the results, we just move the index forward or backward for the number of results and update the labels. Although there’s no order for the results, we store them in their original order passed from the GBE framework for navigation convenience.

Certain special conditions may occur when the results are displayed. The first condition is when the index is at the start or end of the result list. In that case, we would disable the previous button and next button in order. The second condition is when there are not
enough results to fill all the cells. For example, what should we do when there are five results but only ten cells per page? In this situation, we would only fill the first cells with the results.

In addition to the result content, there are also some other information related to the result, like the query time and count of result numbers. We count the query time using the timeframe between sending the data tables to the query engine to receiving the list of data tables. The result count is the list length. We show this two information on the status bar after the query.

4.2.6 Implementations details for additional features

We present details of implementations for GBE interface in addition to those described earlier in this section.

1. Query storing and loading

The query files, along with the query template elements, are stored and loaded using XML serializing functions.

2. Creating a new query from existing template

Users can create new query from an existing template and modify it to fit their needs. This feature is implemented by creating a copy of the current query template in a new tab with name “X new”. The new query template will automatically be stored under the same directory with the original query.

3. Namespace prefix
<table>
<thead>
<tr>
<th>Namespace Prefix</th>
<th>Namespace URI represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdf</td>
<td>&quot;<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a>&quot;</td>
</tr>
<tr>
<td>foaf</td>
<td>&quot;<a href="http://xmlns.com/foaf/0.1/">http://xmlns.com/foaf/0.1/</a>&quot;</td>
</tr>
<tr>
<td>dbo</td>
<td>&quot;<a href="http://dbpedia.org/ontology/">http://dbpedia.org/ontology/</a>&quot;</td>
</tr>
<tr>
<td>owl</td>
<td>&quot;<a href="http://www.w3.org/2002/07/owl#">http://www.w3.org/2002/07/owl#</a>&quot;</td>
</tr>
<tr>
<td>rdfs</td>
<td>&quot;<a href="http://www.w3.org/2000/01/rdf-schema#">http://www.w3.org/2000/01/rdf-schema#</a>&quot;</td>
</tr>
<tr>
<td>ub</td>
<td>&quot;<a href="http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#">http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#</a>&quot;</td>
</tr>
</tbody>
</table>

*Table 4.9 Popular Namespace Prefixes*

The usage of namespace prefix is implemented by matching the namespace prefix to namespace URI. We first store the namespace prefix of some popular RDF databases. Table 4.9 shows some of the popular URIs that we pre-store or plan on pre-storing [24]. After users click running query, the namespace prefixes will be detected and automatically transferred to the namespace URI. For example, if the vertex’s label is ub:ProfessorX, the interface will translate the namespace “ub” into namespace URI “http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#” which will leads to the label “http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#ProfessorX”. And this full URI is sent to the GBE framework to execute the query.

For the query results, we don’t use namespace prefixes as users may need to learn detailed knowledge of the result’s full URI. So all the results’ labels use full URI.

4. Query Description
Query description is a string attribute of each query file. We have a query description section on the bottom right of the interface that displays this information. When users update the query description section, it will also be updated in the query file.

5. Other functions in a table

There are some other implementations on features like attribute of the text, multi-thread query, and query explorer. These functions with the class associated to their implementations are listed in Table 4.10.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Class Associated</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fonts/Color/Size</td>
<td>Style</td>
<td>Set the attribute of the text</td>
</tr>
<tr>
<td>MultiThread</td>
<td>BackgroundWorker</td>
<td>Run query and load database in background.</td>
</tr>
<tr>
<td>Query Explorer</td>
<td>queryExplorer</td>
<td>View and access query files</td>
</tr>
</tbody>
</table>

*Table 4.10 Implementation of other functions*
5. GBE Interface Usage

In this chapter, we will be discussing the usage of the GBE exploratory graph query interface. We begin with an introduction to the complete querying process, followed by demonstrations of different types of queries along with comparisons between our graph query method and SPARQL query language using the Lehigh University Benchmark (LUBM) [8]. LUBM provides standard RDF entries and is good for query evaluation. A brief introduction of LUBM can be found in section 2.3.1 of Chapter 2.

5.1 Querying Process

In this section we detail the complete querying process of the interface. Figure 5.1 shows the query process step by step. Typically, there are 6 steps in total:

Step 1: Choose and Load Database. Users need to configure the database that they are going to query and load the database into the interface.

Step 2: Create Query. There are two ways to create a query. The first way is to create a new empty graph query and expand it through adding vertices, edges and pathEdges. The second way is to choose a query template from the stored query templates and modify it to fit the need.

Step 3: Modify Query. A general step to complete the writing of a new query or modifying an existing query. Note that the description for the query also needs updating. An optional step here is to store the modified query.

Step 4: Run Query. Once the query graph is complete, users can click the “query” button and run the query with the integrated query engine.
Step 5: Analyze Results. In some circumstances there will be many results in the result set, and users can navigate through these results to make analyses. In the exploratory query
process, this step is also required before refining the query. After this step, if users are not satisfied with the results, they can go back into the loop and modify the query. Otherwise, they can proceed to the last step.

Step 6: Get Final Results. When users find their ideal results after query attempts and result analysis, they can choose to save the results or print them and finish the query process.

5.3 Query Examples

In this section we will give examples of querying under different targets and also demonstrate the use of the exploratory query feature. We provide the following types of query examples:

1. Query with full label/query with namespace prefix
2. Exploratory Browsing
3. Focused Searching
4. Query with pathEdge

These types of query cover most query conditions for normal query and exploratory query using our interface. On the other hand, these types of queries are also popular with other databases like DBLP and UniProt. Demonstration of usage in LUBM can also apply to conditions when querying with other databases.

5.3.1 Query with Full Label/URI Prefix

The first example utilizes query Q6. It makes comparison among three versions of Q6, which are SPARQL query, graph template query using full URI and using namespace prefix.
Q6: Find all the courses that are taken by graduate students who have the same FullProfessor advisor with GraduateStudent46.

Select
?C
Where
{
  <http://www.Department2.University1.edu/GraduateStudent46> #advisor ?P
  ?P #type <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#FullProfessor>
  ?S #advisor ?P
  ?S #type <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
  ?S #takeCourse ?C
  ?C #type <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Course>
}

Figure 5.2(a) SPARQL version for Q6

Figure 5.2(b) Graph template query using full URI version for Q6

Figure 5.2(c) Graph template query using namespace prefix version for Q6
The three versions of Q6 are shown in Figure 5.2, with (a) SPARQL query version, (b) graph template query using Full URI version and (c) graph template query using namespace prefix.

First we talk about the knowledge needed to compose the queries. For all the queries, users need to know the structure of the LUBM database and the labels of typical subjects, objects and predicates. For the SPARQL query, users need to master the grammar of SPARQL to write an accurate SPARQL query. For the graph template query using full URI, users need to know how to draw the vertices and edges to compose the query graph. For the graph template query using namespace prefix, in addition to the previous one, users need to know the namespace prefixes and the namespace URIs that they represented.

Next we focus on the comparison between the SPARQL query version and the graph template query version. We make the comparison on three aspects: labels, grammar/structure and method in dealing with partial labels. For the normal labels, both versions need to input all the related labels needed to query. For the grammar/structure, SPARQL’s grammar is stricter than the graph drawing method of graph template query and is more error-prone. In addition, the structure of graph template query is easy to understand due to its graph expression, while the SPARQL query requires some effort to understand. When dealing with partial labels, SPARQL needs to use the filter regex expression while graph template query just needs to put in the partial label in the vertices. It can be seen that graph template query requires less work than SPARQL query to create and enjoys a more concise structure.

The next concern we have is on the comparison between the full label version and prefix version of the graph template query. Both versions are similar in structures, with the only
difference being the labels for some vertices. For the “GraduateStudent”, “FullProfessor” and “Course” vertices in the LUBM database, the usage of namespace prefix can help make the labels more concise and easier to understand as the full URIs are often very complicated.

Find all the courses that are taken by graduate students who have the same FullProfessor advisor with GraduateStudent46.

**Figure 5.3 Query description for Q6**

In real world RDF databases like DBLP, this can have even more advantages as the usage of URIs is more common under those conditions.

**Figure 5.4 Results for Q6**

to write and understand. In real world RDF databases like DBLP, this can have even more advantages as the usage of URIs is more common under those conditions.

Figure 5.3 shows the query description section for this query. This section can help the users understand this query more conveniently. Users also have access to modify this section when they make changes to the query.
Next, we present the results from the full label version and the prefix version. Figure 5.4 shows the results page from the prefix version, which is the same as the results from the full label version. This is because we use full label for the results regardless of the query type. There are two results in total, lined up horizontally as we choose to display results by two per page. In each result, we can see that the nodes with URI prefix are replaced with corresponding full URIs, and the nodes with the query mark “*” are replaced by the results’ full labels. Statistics on the results is shown on the status bar, with a query time of 644ms and 2 results in total.

5.3.2 Exploratory Browsing

The second example here is exploratory browsing. Exploratory browsing is the process of browsing through the database to have a better general understanding of the types and relations in the database. This is very useful when users first access some databases and don’t know much about the data structures and entries in the database.

Consider a scenario where users are given the LUBM database with a very limited knowledge of what is stored in the database. Table 5.1 in section 5.2 shows an example of the brief information shown to the new users. In addition to that, users have access to the GBE query interface. In order to learn more about the data structures and entries in the database, users have to use some browsing queries to browse the database.

The information contained in section 5.2 and Table 5.1 shows that the entries are standard URLs and there’re universities, professors, students, publications, courses and some other types of data in the database. However, the brief information doesn’t fully reveal all the data types and how they are related, which is useful for the users when running queries. In
order to know how the different types of data are related, they can begin with any vertex and browse the URL entries related to them. Q1 is one of the initial browsing queries they can begin with, with the graph template query shown in Figure 5.5.

**Q1**: Find all the URL entries that are related to University1.

![Figure 5.5 Browsing Query Q1](image)

**Figure 5.6 Query Result for Q1**

Q1 browses all the URL entries related to University1, which has the label "<http://www.University1.edu>". This query will help the users learn more about the data.
related to University1, and will reveal general data structure information related to university URLs.

Figure 5.6 gives a screenshot of the results of this exploratory query. There are 58 results in total, and the users can navigate through the results 10 per page. The screenshot provides a partial idea of the data structure around University1 and how University1 is related to other URL entries, which include departments, professors and students. After going through all the results, users can have a better understanding of the stored data that is related to a university. They can then keep browsing the database with the information found in the results, and repeat this process until enough knowledge on the database is obtained.

Figure 5.7 gives an example of using the knowledge acquired from the original query result to form new browsing queries and further explore the database. The results shown in Figure 5.6 shows that there’re many professors and students entries related to a university. So it is reasonable to use browsing queries related to the professor entries and student entries to see the data related to them. Q1* and Q1** are two variants of the refinement and are shown in Figure 5.7(a) and Figure 5.7(b) respectively.

![Figure 5.7(a) Browsing Professor Query Q1*](image1)

![Figure 5.7(b) Browsing Student Query Q1**](image2)
**Q1**: Find all the URL entries that are related to FullProfessor3.

**Q1**: Find all the URL entries that are related to GraduateStudent58.

The refined Q1* and Q1** will further explore the data structure around the professor and student entries and improve the users’ knowledge of the database. Figure 5.8 and Figure 5.9 present the query results for Q1* and Q1**, respectively.
5.9 show part of the results of the two new browsing queries. The first result shows that Fullprofessor3 is connected to the department “<http://www.Department10.University5.edu>“, which he belongs to; the graduate course “<http://www.Department10.University5.edu/GraduateCourse4>“, which he taught; and the universities “<http://www.University613.edu>“/<http://www.University1.edu>“, which he graduated from. The second result shows that GraduateStudent58 has an advisor AssociateProfessor4, takes courses GraduateCourse7/GraduateCourse27/GraduateCourse49, attends Department100 at University8 and graduates from University1. As we can see from the results, these two queries further reveal the data structure around the query entries through some instances (Course/Department/Advisor). These information will be very useful for new users to the LUBM and they can further explore the database using this technique.

Currently, the query engine doesn’t support the labels on the edges, so we don’t have the edge labels shown on the graphs. This makes it a little harder to understand what the relationship means. However, in standard RDF databases, the relationships are often standardized and are easy to recognize. The future introduction of edge labels will also help to produce better understanding of the relationships.

5.3.3 Focused Searching

The next example we present here is the focused searching model in the exploratory querying process. Focused Searching is very useful when users have certain knowledge about the database and want the query to find some interesting topics and results. This is the common type in exploratory querying and an interface with this feature can often increase productivity of the researches [7].
The example we provide here features Q2, which is a query about FullProfessor1. In Q2, the users already know that person type is related to the universities that they graduate from, and wants to know the information about the alumni of FullProfessor1. The graph template is shown in Figure 5.10.

**Q2: Find the people who attend the same University as FullProfessor1.**

![Figure 5.10 Focused Search Query Q2](image)

After running the query, the users will get the basic information of the alumni of FullProfessor1. Figure 5.11 shows a screenshot of the results page. The result set shows that there are 139 results in total. The users can then refine the query based on the results. The refinement can be either of the three types shown in Figure 3.3 in Chapter 3. The
refinement examples we demonstrate here make the query more restricted and thus reduce the result set and produce a more focused result set.

**Figure 5.12(a) Focused Search Refined Query Q2***

The first refinement focuses more on a certain university. Among the results, certain university that FullProfessor1 graduate from might arose more interest than others, so a refinement to fix the university and find only alumni within this university can satisfy the new interest. This refinement example fixes University298 and the graph template is shown in Figure 5.12(a).

The second refinement focuses more on a certain person. The users might be more familiar with certain alumnus/alumna and want to learn more about the shared features between FullProfessor1 and the alumnus/alumna. In this case the refinement will keep the two people’s vertices but only change the shared vertex using query mark to query the shared property. The graph template query is shown in Figure 5.12(b), which keeps the nodes of FullProfessor1 and Lecturer1 and use query mark for their shared vertex in the middle.

In addition to the refinements listed above, there can also be other refinements on the query graph which concerns about the university they attended or search for more information on
certain alumnus/alumna. The graph-intuitive interface provides a good platform for these modifications with reference to previous query results.

This example shows how users can use this interface to perform a focused search and exploratory query. A good usage of such features can help users find interesting and important topics more efficiently.

5.3.4 Query with pathEdge

The last example we give here concerns about querying with pathEdge. The introduction of pathEdge in GBE graph template query can hugely reduce the work in composing new queries and eliminate the need to deal with complicated internal nodes. The example we give here features query Q7 on a publication:

**Q7:** Find the name of the publication in Department2 of University1 that is on the research topic "Research20" and is connected to University845.

In circumstances that use basic graph query template, if users want to find a publication on research topic “Research20” and is connected to University845, they would need to know how the publication is connected to the research topic and the university. This brings new problems and often requires additional work to obtain additional information before ready to query. For example, in query Q7, users would not be able to draw the query graph if they are not clear of the internal relationships, and they would have to get some more information. They would only be able to draw the query graph after they find the publication is connected to the research topic through a professor, and is connected to the university through a graduate student. This leads to the graph query template shown in Figure 5.13.
In the query template, only the publication vertex, the research topic vertex and the university vertex are information known before the original query, while the graduate student and full professor information both requires additional effort to learn. In addition, it makes the query template more complicated than what the original query expects, bringing four extra vertices, two related to the graduate student and two related to the full professor.

In order to simplify process under similar conditions, the GBE query framework proposed a way to skip the internal vertices and query with connected edge (pathEdge) [6], which represents a path in the database. In GBE interface we facilitate this function of the framework and let users use the interface with pathEdge in addition to the basic vertex and edge.

Figure 5.13 Q7 graph query template with full path

In the query template, only the publication vertex, the research topic vertex and the university vertex are information known before the original query, while the graduate student and full professor information both requires additional effort to learn. In addition, it makes the query template more complicated than what the original query expects, bringing four extra vertices, two related to the graduate student and two related to the full professor.

In order to simplify process under similar conditions, the GBE query framework proposed a way to skip the internal vertices and query with connected edge (pathEdge) [6], which represents a path in the database. In GBE interface we facilitate this function of the framework and let users use the interface with pathEdge in addition to the basic vertex and edge.
For example, when drawing the graph query template for Q7, users would be able to use just the original information provided and draw the new version with pathEdge (dotted line), shown in Figure 5.14(a). Instead of dealing with the internal nodes, users can use pathEdge.
and connect the publication directly with the university and research topic. There can be a distance constraint on the pathEdge, as is shown on the pathEdges in Figure 5.14(a). The distance constraint is to ensure that the pathEdge make sense in the context, as a long path may lose its meaning in querying in LUBM. The default distance constraint varies by dataset.

Users can then treat the query in the same way as the basic query template and query directly with the pathEdge query. The results will be in the form of the original query structure, with the query marks replaced by the matching results’ labels, as is shown in Figure 5.14(b). The result display and navigation functions are the same as the case of basic graph query template.

In the above sections, we have demonstrated usage of the GBE exploratory query interface for several different querying types in the LUBM database. We have shown that it is simple and easy to use and helps users in finding interesting topics and important results. As LUBM is a generated standard dataset and real-life databases are often more complicated, our interface would have more advantages when dealing with real-life databases.
6. Related Work

In the mid-1970s, Moshe M. Zloof proposed a database query language for relational databases called Query by Example (QBE) [25]. Query by Example is the first graphical query language that converts user actions to database manipulation language, inspiring new ideas for future query interface designs. Since then, researchers have made great progress in developing query interfaces for traditional databases. Early query interfaces include Living in a Database System (LID) and Prototype Personal Database (PDB), which concerns the basic graphical operations and display of traditional databases. In recent years, research has also been done to design smarter interfaces for traditional databases. Magesh et al. presented a way to generate form-based database query interface in an efficient way automatically [26], and Fei et al. proposed an interactive natural language in interface NaLIR for querying RDBMS [27].

On the other hand, with the development of graph databases over web data, especially RDF databases in the twenty-first century, many researchers have been dedicated to designing new interfaces for graph databases. Mario et al. presented a web-based query interface called Structured Advanced Query Page (SAQP) for the querying of bioinformatics web data, which allows users to build up a wide range of query constructs while setting various query constraints to form the query. The result is displayed in forms. FreeQ for Freebase [28] and Uniprot for uniprot [5] database are other examples of database specific query interface that provides form and text-based query interface for users to select and enter simple texts. Users can use the resulting information to construct and run the query. This process allows users to query and manipulate the specific database, without having to master the SPARQL query language.
In Blueprint Titan interface, researchers provide an interface along with a graph-centered query language to query common graph databases [14]. The use of query language effectively utilizes the graph structure and is much simpler than SPARQL. These interfaces have made querying of RDF much easier than using SPARQL, yet they still have one problem to solve. They are all text based and unable to present the graph structure directly to users.

Meanwhile, Shi et al. proposed the idea of graph template matching in a Graph by Example (GBE) framework [6]. Inspired by this framework and the Query by Example idea, we decided to design a graph query interface that allows users to draw graph templates, query them directly and get the results displayed as graphs, leading to the graph feature of the GBE interface.

SPARQLGraph interface developed by Dominik et al. also provides the functionality of graphically querying web databases [15]. There are some similarities and some differences between our GBE interface and SPARQLGraph interface. The comparison between the interfaces are as follows:

1. The idea for query graph databases is the similar. Both interfaces let users draw the vertices and edges and query directly with the graph template query.
2. Both interfaces provide template queries in multiple databases and let the users choose from them to query directly.
3. The domain of databases the interfaces support are different. SPARQLGraph targets at querying biological semantic web databases, while GBE interface targets at general RDF databases including multiple domains.
4. The integration of the databases to the interface is different. In the SPARQLGraph interface, the databases are integrated to the interface internally and no other databases are supported. In GBE interface, there are some databases integrated, but users can also configure an outside database. For either integrated or outside databases, users need to configure and load them in before querying.

5. GBE interface supports the usage of connection edge in query template, allowing users to skip complicated internal vertices when they want, while in SPARQLGraph users need to draw every vertex along the path to make the query work.

6. The platform of the interface is different. SPARQLGraph interface is a web based application based on JavaScript and uses Meteor framework, while GBE interface is a windows based application utilizing the NClass Framework. Further, SPARQLGraph is based on web users and need registration before use, with different profile for different users, while GBE interface is based on the computer and don’t have user profiles associated to it.

7. The output of the interface is in different format. In SPARQLGraph interface, the results are displayed in a tabular form or exported in CSV format. In GBE interface, the results are displayed as graph instances in the RDF database matching the graph template, and can be saved as template result or exported as pictures.

Another aspect of our work is the design of interfaces that supports exploratory querying, especially in web data querying. We will list several of the works in this section.
Eyal et al. developed an exploratory interface for RDF data using the faceted browsing techniques [29]. They partitioned the semi-structured RDF data into facets and set constraints to the data to allow exploration of an unknown dataset.

David et al. built a web-based system which allows users to assess the ways they can improve their searching [30]. It features pre-determined sets of information and tasks to help users better determine ways of analyzing records of query refinement behavior.

Haggai et al. focused on the exploratory search over social and medical data [18]. The IBM Patient Empowerment System presented features an interactive query interface, which provides various types of question capabilities. Their query interface starts with a free text query, followed by query language, faceted search, or entity-relationship graph navigation methods to refine the query.

Gene et al. presented a design for general exploratory search [31]. Their multi-session exploratory search system records people’s search activity as historical metadata, and utilizes this data to help improve the search activity.

The current works in exploratory querying mainly focuses on the idea itself combined with traditional interfaces. The method they use to do exploratory querying is often related to faceted browsing. The combination of graph template query and exploratory querying would bring advantages to querying in graph databases.
7. Future Work

Currently, the GBE exploratory graph query interface is the first windows query interface that allows users to query using graph templates. As we focused more on the basic interface features needed to do graph query and exploratory query, we did not yet cover the more advanced features, wider usage or optimized user experience. In this chapter, we discuss some of the future works that can be done in the future to further improve the GBE interface.

First, we want to talk about advanced features, beginning with the query auto completion feature. Current web search engines provide the feature to show some possible full search entries when users input the first few words. This feature would also be beneficial to our graph query interface. When users draw the first few vertices with labels, the interface with this feature would show the possible complete query templates. Completing this step will help in creating and reformulating new queries.

The next possible future feature is the support for natural language query conversion. Currently, the interface has a description section which allows the users to write down a description of the query. A useful feature would be translating between the query description and the query template. That is, users can create new query templates by typing in natural language query, while letting the interface translate the natural language into query templates. On the other hand, when users create new query templates through drawing, the interface can automatically generate the natural language description for it. This feature will provide more alternatives in writing and modifying queries.

Exploratory query is an important feature in the GBE interface, so adding new features that further assist exploratory query would be very helpful. The current version only provides
the query result’s view for the query reformulation. A query preview feature or query anticipation based on previous user searches, as is discussed in chapter 3, will provide more assistance to users in exploratory querying.

The next topic we concern about is the platform of the interface. Currently, the GBE interface is a windows program that can only be run on the Windows platform, which restricts its usage. A web application version for GBE interface would help the interface spread more efficiently, while providing users with increased convenience and access. The web application will be a new project that requires major work.

Lastly, we want to look into ways to optimize user experience. As GBE interface is in its first version with a smaller user base, it is difficult to make a prediction on user feedback. In order to optimize user experience, user experiments and surveys need to be done in order to understand user behaviors and obtain user feedback. The use of experiments can help us improve to the GBE interface for better user experience.
8. Conclusion

The GBE exploratory graph query interface is the first windows query application that allows users to query RDF databases with graph templates. GBE utilizes the graph feature of the RDF data, featuring graph templates as the basic query units while displaying the graph instances in the result set. Compared to SPARQL and other text based query languages and interfaces, GBE is simpler and easier to use. The intuitive way of using graph templates also gives users a better grasp of the data structure, along with a better experience in querying RDF databases.

In addition, GBE interface also support exploratory querying, which helps researchers better refine their queries to obtain important results. The results that are presented in internal graph instances, along with other supplementary features, can greatly help users in analyzing results and making appropriate refinements.

However, there is still much work to be done in order polish the interface and expand its application. With appropriate improvements, the GBE interface can further bring benefits to graph database related searches.
Bibliography


