1-1-2012

Effective semantic-based keyword search over relational databases for knowledge discovery

Sina Fakhraee
Wayne State University,

EFFECTIVE SEMANTIC-BASED KEYWORD SEARCH OVER RELATIONAL DATABASES FOR KNOWLEDGE DISCOVERY

by

SINA FAKHRAEE

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2012

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

I would like to dedicate this dissertation to my wonderful parents Dr. Seyed Hossein Fakhraee and Soraya Golrokh, who have supported me continuously not only during my PhD years but in every aspect of my life. Without their encouragement and support I would have not been able to pursue higher education, let alone obtain a PhD. I dedicate this dissertation to them in appreciation for the infinite support they have given me.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere appreciation to my academic advisor, Dean Farshad Fotouhi for all his support, guidance and help throughout my PhD years. Being the Chair of the Department of Computer Science and then Dean of the College of Engineering has kept him extremely busy, but he still manages to spend as much time with his graduate students as they need to advise them and make sure that they are all on track. I want to thank him for co-authoring all of my research papers with me, for his very helpful insights, comments and suggestions during my research as a PhD student. Being his student has taught me not only how to research and how to solve difficult problems, but has also taught me to have discipline in life and how to interact in a professional and at the same time friendly manner with my colleagues. I want to thank him for making sure that all of the graduate students, including myself were financially supported either by our department or by finding research/teaching assignments in other departments for us. It has been my great honor to work under the supervision of someone with such an excellence academic background, knowledge and experience in variety of different fields within computer science and even other disciplines. I will always look up to him as my academic advisor and as an excellent researcher.

I would like to thank Dr. Carol Miller, the chairperson of the Department of Environmental and Civil Engineering who is also serving in my PhD committee, for supporting me as a graduate research assistant in my 3rd year, getting me involved in a couple of multidisciplinary research projects, co-authoring a paper with me and sending me to a nationally well-known and prestigious conference.

I would like to thank my other PhD committee members, Dr. Chandan Reddy and Dr. Zaki Malik for their very helpful insights, comments and suggestions throughout my PhD years at Wayne State. Dr. Reddy has been always willing to help me with complex data mining

questions/problems in particular. Dr. Malik has always shared with me his extensive knowledge on web services and semantic web technologies in particular.

I would like to thank my supervisor at Ford Motor Company, Dr. Shuh-Yuan Liou, who despite his extremely busy schedule agreed to serve on my PhD committee. I want to thank him for supervising and coaching me on how to apply many aspects of my academic research work to real world and enterprise scenarios. It has been a pleasure working for him.

I would like to thank Dr. Loren Schwiebert, the Graduate Director of the Computer Science Department, for all of his support to all the graduate students and for making sure they are on track and are fulfilling the department and graduate school requirements.

I would like to thank all of the faculty members and staff of the computer science department and all of my friends and lab colleagues for their help, support and academic cooperation.

I would like to thank my wonderful sister, Salomeh Fakhraee, and her wonderful husband, Hamid Alavi, who were the first people in my life to inspire me to pursue a degree in computer science. Without their help and support and especially while I was pursuing an undergraduate degree as a computer science major, I may have given up and chosen a less challenging major. They have done a lot for me and I am very grateful to have them in my life.

I would like to thank my very talented brother, Sasan Fakhraee, for helping me with web design on a couple of projects and thus allowing me to focus more on writing the business logic of the software I was working on.

Lastly and most importantly, I want to thank my lovely, kind wife, Jennifer Gross, who has supported me every day of my PhD journey. It takes a lot of patience and sacrifice to live with and be in a relationship with a PhD student. There was not a single day that she has complained about my busy schedule or about assisting her in proofreading my research papers, despite her own busy schedule, being a law student and working at a law office. I am very grateful to have her in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND PROBLEM STATEMENT

1.1 Overview and Background

Keyword-based search has been popularized by Internet web search engines such as Google which is the most commonly used search engine to locate the information on the web. On the other hand while traditional database management systems offer powerful query languages such as SQL, they do not provide keyword-based search similar to the one provided by web search engines. The current amount of text data in relational databases is massive and is growing fast. This increases the importance and need for non-technical users to be able to search for such information using simple keyword search just as how they would search for text documents on the web. Keyword search over relational databases (KSRDBs) [32, 33, 34, 35, 36, 37, 38, 39, 40, 1, 2, 3, 4] enables ordinary users to query relational databases by simply submitting keywords without having to know any SQL or having any knowledge of the underlying structure of the data. Similar to a web search engine such as Google that requires the user to enter a set of keywords to find documents containing the keywords, in KSRDBs the user enters a set of keywords to find the inter-connected tuples joined on their primary-foreign key relationships containing the keywords which if found are presented to the user. Finding answers to KSRDBs is a very challenging task since good answers should be assembled by joining tuples from multiple relations across the databases. The effectiveness of KSRDBs is even more of a challenging task since unlike the text databases, relational databases have much richer structure and the search keywords are usually not just simply found in a single text attribute but they can be found in different text attributes of different relations each of which having different degree of relevancy to the query term(s).

Some of the challenging research issues in keyword search over relational databases are: search efficiency, search effectiveness, search recommendations, search over heterogeneous relational databases and semantic-based keyword search over relational databases.

Early research works [2, 4] in this area try to capture all the inter-connected tuples containing the exact keywords and then rank the results purely based on only the distance of the keyword-containing tuples from one another. This kind of approach is not very efficient since most users are only interested in the first top-k search results and also is not very effective because besides the distance of keyword-containing tuples from one another, other factors should be taken into account when ranking the answers.

Modern relational databases have incorporated the state-of-the-art information retrieval (IR) relevance ranking techniques at the attribute level. Recent works in KSRDBs have exploited this functionality to answer the keyword queries by identifying all database tuples that have a non-zero score for a given keyword search [1, 3, 6].

Once these tuples are found, the first top-k tuples containing the keywords from each relation, if joinable, are joined via their primary-foreign key relationships and the ones which collectively contain the search keywords are presented to the users as the search results. Taking advantage of IR relevance ranking strategies employed by modern relational database management systems (RDBMSs) has improved both the efficiency and the effectiveness of keyword search in relational databases to some extent.

1.2 Problem Statement and Research Contributions

The problem of finding keywords in the context of relational databases has been broadly researched and addressed by previous research works. In this research work our primary focus is to enhance the effectiveness of the keyword search over relational databases using semantic

web technologies. We have also addressed some the issues with the effectiveness of the current keyword search over relational databases. In particular we are addressing the followings:

1) To find more relevant search results to the users' intended search results. In many cases users' intended search results may have been hidden in the text attributes containing long string as supposed to short string. The current ranking functions applied on these text attributes need to be improved for enhancing the effectiveness.
2) To return similar search results by the search engine as the search results' recommendations. Preferably, making search recommendations should be done regardless of whether the user has searched that data repository (i.e. RDBs for our research) in the past or not, meaning the system should not always rely on making suggestions based on the users' past search history.
3) To bring semantics to keyword search over relational databases by considering not only the syntactic similarities between the query terms and the databases terms but also the semantic similarities of the two.
4) To present the relationships between the keywords in the search results which are crucial for the proper interpretation of the search results by the user and should be clearly presented in the search results.

I have made the following contributions during my PhD years which are presented in this dissertation:

1) I have studied the IR style techniques used in recent research works in the area of KSRDBs and have identified two important factors that can further improve the search effectiveness when incorporated into the IR relevance ranking strategies. These two factors are i) The query keywords' proximity, which is the overall distance of the keywords from one another in the value of the target text attribute and ii) The N-grams and, in particular, the quadgrams of the query keywords in both the query itself and in the text attributes' values. My experiments show that

incorporating these two factors into the existing state-of-the-art ranking functions can improve the ranking of KSRDBs by achieving significantly higher precision at standard recall points.

2) I have adapted a novel approach in making keyword search recommendations based on the text attributes in which the search terms were found without relying on the user's past search criteria. A proof of concept (POC) prototype system called TupleRecommender has been implemented based on this approach.

3) In (1), I identified two important factors which indeed improved the search effectiveness when incorporated into the IR relevance ranking strategies. However, one major issue with keyword search in general is its ambiguity which can ultimately impact the effectiveness of the search in terms of the quality of the search results. This ambiguity is primarily due to the ambiguity of the contextual meaning of each term in the query (e.g. each query term can be mapped to different schema terms with the same name or their synonyms). In addition to the query ambiguity itself, the relationships between the keywords in the search results are crucial for the proper interpretation of the search results by the user and should be clearly presented in the search results. To address these issues we have designed and implemented a proof of concept (POC) prototype system called database semantic search explorer (DBSemSXplorer) which can answer the traditional keyword search over relational databases in a more effective way with a better presentation of search results. We address the keyword search ambiguity issue by leveraging some of the existing techniques for keyword mapping from the query terms to the schema terms/instances. These techniques capture both the syntactic similarity between the query keywords and the schema terms as well as the semantic similarity (e.g. definition of the keywords) of the two and give better mappings and ultimately more accurate results. Finally, to address the last issue of lacking clear relationships among the terms appearing in the search results, our system has leveraged semantic web technologies in order to enrich the knowledgebase and to discover the relationships between the keywords. Our experiments show

that our system is more effective than the traditional keyword search approaches by enabling the users to find the search results which are more relevant to their keyword queries.

1.3 Dissertation Organization

The remaining of the dissertation is organized as follows: Chapter 2 discusses related work in KSRDBs, gives the basic concepts and definitions used in KSRDBs literature and gives an overview of query result generation. Chapter 3 presents my work: Effective Keyword Search over Relational Databases Considering Keywords Proximity and Keyword N-grams. Chapter 4 presents my work: TupleRecommender: A Recommender System for Relational Database Systems. Chapter 5 gives an overview of semantic search. Chapter 6 presents my work: DBSEMSXplorer: Semantic-based Keyword Search over Relational Databases for Knowledge Discovery. Chapter 7 builds on our approach presented in chapter 6 to further enhance the search using semantic web inference capability. Chapter 8 concludes my dissertation and gives research direction for future work.

CHAPTER 2

RELATED WORK

In this chapter, the basic terms and concepts of keyword search over relational databases are explained. This includes the basic definitions used in the literatures, search result generation and the ranking approaches used in the existing solutions.

2.1 Basic Concepts and Definitions

A relational database is consisted of a set of relations R_1, R_2,..,R_n and each relation R_i has m_i attributes $a_{i\text{-}1}$, $a_{i\text{-}2}$,…, $a_{i\text{-}m}$. A database schema graph is a directed graph *G(V,E)* where *V* represents the set of nodes corresponding to the database relations and *E* represents the set of edges corresponding to the primary-foreign key relationships among the relations. G_u is defined as the undirected version of *G*. Below are a few terms and notations defined in KSRDBs literatures [2,4] that we will be using trough out the paper:

Tuple tree: A tuple tree *T* also referred to as joining network of tuples or inter-connected tuples is a tree of tuples where for a tuple $t_i \in T$ adjacents to a tuple $t_j \in T$ where t_i is a tuple in R_i and t_j is a tuple in R_j , there is an edge from R_i to R_j in the schema graph of the database(and hence in G_u) and $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$.

Keyword Query: A keyword query *Q* is a set of keywords K_1, K_2,…, K_n entered by the user.

Query result: Query result *R* is a set of *tuple trees* where for each *tuple tree T* each keyword k_i must appear in at least one tuple t_i of *T*. This guaranties the totality of a tuple tree *T* which is, all the keywords are contained in *T.* Also, no tuple can be removed from *T* and still be total. This guaranties the minimality of a tuple tree *T*.

Master Index: Master index is an inverted list that relates each keyword that appears in the database with a list of locations in the database which are recorded as row-column pairs.

Basic tuple set: A basic tuple set of relation R_i with respect to a keyword k_j denoted by $R_i^{k_j}$ is a set of tuples in R_i that contain k_j in at least one of their text attributes.

Tuple set: A tuple set of relation R_i with respect to a K, a subset of keyword query Q, denoted by R_i^k is a set of tuples of R_i that only contain keywords in K and not any other keywords.

Free tuple set: A free tuple set of relation R_i denoted by R_i^F is the set of all the tuples of relation R_i, in other words it is the relation R_i itself.

Joining Network of Tuple Sets: A joining network of tuple sets J is a tree of tuple sets where for a tuple set $R_i^{k_1} \in J$ adjacents to a tuple set $R_j^{k_2} \in J$, there is a corresponding edge (R_i, R_j) in G_u.

Candidate Network: A Candidate network also referred to as join tree is a join expression involving tuple sets which are used to generate tuple trees that could be potential answers to the keyword query Q. In other words a candidate network is a joining network of tuple sets that is used to generate answers to a keyword query. Another definition for join tree states that a join tree $J(v, e)$, where $v \epsilon V$ and $e \epsilon E$, is a sub-tree of G such that each individual leaf node (relation) contains at least one of the keywords from Q and all the leaf nodes together contain all the keywords. This is shown in Figure 2.1 where on the left side the schema graph, G, of a database is depicted and relations containing the keywords, assuming Q={ *k1,k2,k3*}), are highlighted and on the right side all possible join trees of G are shown.

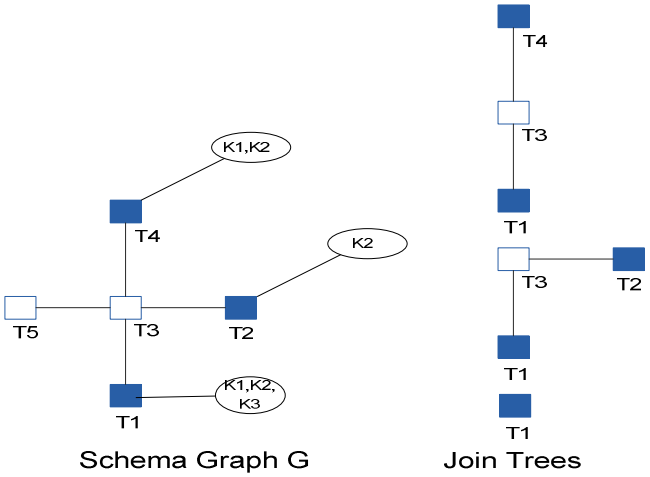

Figure 2.1: Schema graph of a database and all its possible join trees.

2.2 Keyword Search Result Generation

In this section I will briefly explain how the answers to keyword search over relational databases are generated, for detailed explanation please refer to the original papers [2,4] published in this area. To generate answers for a keyword query *Q* given by the user four modules are involved. The first module in the pipeline is the master index which inputs the keyword query *Q* and returns a set of basic tuple sets for each relation with respect to each keyword. The second module is the basic tuple set post-processor which inputs the basic tuple sets and generates the set of tuple sets for each relation with respect to all subsets of *Q*. The third module in the pipeline is the candidate network generator which inputs the keyword query *Q*, set of tuple sets, free tuple sets and the schema graph of the relational database and outputs the set of candidate networks. The last module in the pipeline is the execution engine which inputs the set of candidate networks and outputs the set of answers to the keyword query by executing the actual SQL statements corresponding to each candidate network.

The key component in the pipeline is the candidate network generator which generates the joining networks of tuple sets that are used to generate the actual query results which are joining networks of tuples containing the query keywords.

As mentioned in the previous section the two criteria that must be met by joining networks of tuples to be the answers to a keyword query are 1) Totality and 2) Minimality. Consequently this implies that the joining networks of tuple sets belonging to candidate networks must satisfy these two conditions in order to generate the inter-connected tuples meeting these two conditions. The condition for totality that must be met by a candidate network *J* is as follows:

$\forall\, k \in Q\ , \exists\ R_i^K \in J \mid k \in K$ (Condition 1)

This condition simply ensures that every keyword in *Q* is contained in at least one of the tuple sets of *J*. The two conditions for minimality that must be met by a candidate network *J* to ensure the generation of inter-connected tuples *T's* that are minimal with respect to the keyword query *Q* are as follows:

$\nexists\ R_i^F \in J \mid R_i^F\ is\ a\ leaf\ node$ (Condition 2) and $\forall\ t_i\ \in T, \nexists\ t_j \in T \mid t_i = t_j$ (Condition 3)

The second condition states that there must not be any tuple set containing no keyword as a leaf node or in other words candidate network must not contain a free tuple set as a leaf node. This ensures the generation of inter-connected tuples that do not have a tuple which contains no keyword. The third condition states that, the same tuple in an inter-connected tuples cannot repeat twice and this condition is carried to the joining networks of tuple sets *J's* by not allowing a candidate *J* to be of the form: $R_i^m - R_j^{korF} - R_i^n$. Together, these two conditions ensure the minimality of inter-connected tuples generated by a candidate network. Join-tree generation algorithm is shown in Figure 2.2.

Algorithm1: Candidate Network (Join-Tree) Generation Algorithm
Input: Keyword query *Q*, set of free and non-free tuple sets with respect to *Q* and the schema graph of the relational database
Output: Set of candidate networks, *J-Set*
1: *J-Set* = ∅ // set of candidate networks
2: $queue\ q$ // enqueue all the free and non-free tuple sets
3: *For each* $R_i^{KorF}\ in\ q$; $q\ is\ not\ empty$; *J = deque q*
4: *For each* $R_j^{KorF}\ in\ q$; R_j *is adjacent to* R_i ;*Nod e= deque q*
5: *If* (*J* is not Total && addition of *Node* to *J* does not violate conditions 3)
6: *J = J + Node*
7: *If* (*J* is Total) // (condition 1)
8: *J-Set = J-Set* ∪ *J*
9: *endif*
10: *endif*
11: *end for*
12: *end for*

Figure 2.2: Join-Tree Generation Algorithm.

2.3 Background on Ranking the Search Results

As stated in the introduction, in keyword query over relational databases the user enters a set of keywords and the system tries to find the inter-connected tuples joined on their primary-foreign key relationships containing the keywords which if found are presented to the user. Finding relevant answers to keyword search over relational databases is a very challenging task since relevant answers should be assembled by joining tuples from multiple relations across the databases. Ranking the search results is even more challenging since unlike the text databases, relational databases have much richer structure and the search keywords are usually not just simply found in a single text attribute but they can be found in different text attributes of different relations each of which having different degree of the relevance to the query keyword(s).

The first couple of works [2, 4] in this area try to capture all the inter-connected tuples containing the exact Keywords and then rank the results purely based on only the distance of the keyword-containing tuples from one another. In other words they rank the results mostly based on the size of the answer tuple tree which is, given a query *Q* the score assigned to an answer tuple tree *T* is:

$$score(T, Q) = \frac{1}{size(T)} \tag{2.1}$$

Where, *size(T)* is the number of tuples in tuple tree *T*. The intuition behind this approach is that, the closer the keywords are to each other the more relevant the answer tuple tree is to the keyword query *Q*. This kind of approach is not very efficient since most users are only interested in the first top-k search results and also is not very effective because besides the distances of keyword-containing tuples from one another, other factors should be taken into account when ranking the answers. This simple approach only captures the size of the tuple trees and ranking the query results with this approach is only effective when tuple trees are consisted of tuples containing only one distinct keyword. As the current relational databases have many text-attributes within each tuple and each text-attribute contains large amount of text data and it is very possible for different search keywords to appear in the same text-attribute, this approach cannot be very effective in ranking answers from such relational databases.

Over the recent years IR community has developed state-of-the-art ranking techniques that can be leveraged when ranking results for keyword search in relational databases. Modern relational databases have incorporated the state-of-the-art information retrieval (IR) relevance ranking techniques at the attribute level. Recent works in KSRDBs have exploited this functionality to answer the keyword queries by identifying all database tuples that have a non-zero score for a given keyword search [1, 3, 6].

For instance [1] has exploited this ranking feature at the text attribute provided by the modern RDBs to define their ranking function which has two sub-functions, namely *Score* and *Combine* as defined below:

$$Score(d, Q) = \sum_{k \in Q \cap d} \frac{1 + \ln(1 + \ln(tf))}{(1 - s) + s\frac{dl}{avdl}} . ln\frac{N}{df} \tag{2.2}$$

where, *Score(d,Q)* is the relevance score with respect to the keyword query *Q* determined by an IR engine for a single text attribute *d* which is viewed as a text document. *k* is a single keyword in *Q*, *tf* is the term frequency of *k* in the value of text attribute *d*, *df* is the document frequency for keyword *k* which is the number of tuples in *d*'s relation with *k* appearing in *d*'s value, *dl* is the document length which is the number of characters of the value of *d*, *avdl* is the average length of the value of text attribute *d*, *N* is the number of tuples in *d's* relation and *s* is a constant number of value 0.2 usually. Now, let *D* be the set of all the text attributes of an answer tuple tree *T*. The score assigned to *T* with respect to the query *Q* is calculated using the aggregate function *combine* as follows:

$$Score(T,Q) = Combine\big(Score(D,Q), size(T)\big) = \frac{\sum_{d_i \in D} Score(d_i,Q)}{size(T)} \quad (2.3)$$

These two formulas together state that once the individual tuples are ranked, the first top-k tuples containing the keywords from each relation, if joinable, are joined via their primary-foreign key relationships and the ones which collectively contain the search keywords are presented to the users as the search results.

[3] has identified four important normalization factors that can further improve the keyword search effectiveness:

1) In Formula 2.3, using the raw *size(T)* can be sub-optimal and might not rank the relevant answers accurately. For example, consider a query $Q=\{k_1, k_2, k_3\}$ and answer tuple trees $T_1 = t_1—t_2—t_3$ and $T_2 = t_1$ where t_1 contains $\{k_1, k_2\}$ and t_3 contains $\{k_3\}$. Under *Or* semantics (i.e. retrieving answers containing a subset of keywords and not necessarily all the keywords) both T_1 and T_2 are the answers to the keyword query. Now assume $Score(t_1,"k_1k_2")=s_1$ and $Score(t_3,"k_3") = \frac{s_1}{2}$. Therefore, $Score(T_1,Q) = \frac{s_1+\frac{s_1}{2}}{3} = \frac{s_1}{2}$ and $Score(T_2,Q) = s_1$. As it can be seen, Formula 2.3 has assigned a higher score to T_2 even though T_1 is more relevant to the query. To resolve this issue, [3] proposed a normalized *size(T)* denoted by *Nsize(T)* as defined below:

$$Nsize(T) = (1 - s) + s * \frac{size(T)}{avgsize} \quad (2.4)$$

where *avgsize* is the average *T* size across all the answer tuple trees. *Nsize(T)* definition is similar to the document length normalization of the original *score* function in denominator of Formula 2.2.

2) The second issue addressed by [3] is with the average document length *avdl* in Formula 2.2. As discussed in that paper, the current definition of *avdl* only considers average text attribute length within local text attribute (i.e. intra-document collection). However, each document collection has its own *avdl* which could be very different from another document collection(i.e text attribute), consequently this could affect the relevance ranking of the answer tuple trees from different candidate networks. For example, two answer tuple trees T_1 and T_2 belonging to two different candidate networks, both containing all the keywords in two distinct text attributes which have very different *avdl's*, will be ranked differently (assuming other factors such as *tf*, *N* and etc are the same for both). To resolve this issue, [3] proposed a normalized *avdl* denoted by *Navdl* which considers average document length within both local and global document collection as defined below:

$$Navdl = \frac{avdl}{(1 + \ln(avdl))} \quad (2.5)$$

3) The third issue addressed by [3] is with the document frequency *df* and total number of tuples *N* in Formula 2. As discussed in [3], same keyword terms might have very different document frequency in different document collections. For example a keyword that is a person's name appears more frequently in tuples of a relation containing a text attribute "*Authors_Name* " versus in tuples of a relation containing a text attribute "*Books_Title* ". Also *N* for different relations could be very different. For example relation containing "*Authors_Name* " attribute has much less number of tuples than a relation containing "*Books_Title* " attribute since each author can have many books. To resolve these issues, [3] proposed global document frequency df^g and global document count N^g. df^g is the total number of documents (i.e. total number of text

attributes' values) in the entire database containing the keyword and N^g is the total number of all text attributes' values in the entire database.

Due to the lack of space we refer the user to read [3] for the fourth normalization factor. The final scoring function for tuple tree *T* with respect to query *Q* is obtained by replacing *size(T)* with *Nsize(T)*, *avdl* with *Navdl*, *df* with df^g and *N* with N^g. Comparing to the early works of [1,2], this ranking function has significantly improved the effectiveness of keyword search over relational databases.

Taking advantage of IR relevance ranking strategies employed by modern relational database management systems (RDBMSs) has improved both the efficiency and effectiveness of keyword search in relational databases to some extent.

In the next chapter, we present our proposed work in improving the keyword search effectiveness. We have identified two important factors namely keyword proximity and keyword N-grams, that once incorporated into the ranking function, they can significantly improve the effectiveness of the keyword search over relational databases.

CHAPTER 3

EFFECTIVE KEYWORD SEARCH OVER RELATIONAL DATABASES CONSIDERING KEYWORDS PROXIMITY AND KEYWOR N-GRAMS

3.1 Introduction

The original IR-style relevance ranking for an individual text-attribute and all the proposed normalization factors, were primarily based on the different keywords' statistics such as local and global text-attribute frequencies, inverse local and global text-attribute frequencies and text-attribute length.

In this research work we have studied and explored the IR style techniques used in recent research and have identified two important factors within the text attributes which once incorporated into the ranking function, can significantly improve the search effectiveness.

These two factors are 1) the query keywords proximity which is the overall distance of the keywords from one another in the value of the target text attribute and 2) the N-grams and in particular the quadgrams of the query keywords in both the query itself and in the text attributes' values. Our experiments show that incorporating these two factors into the existing state-of-the-art ranking function can significantly improve the effectiveness of KSRDBs.

3.2 Keyword Proximity

An important factor that should be incorporated into the existing relevance ranking function for relational databases is the keyword proximity which rewards a text attribute where the matched keywords are in the smallest proximity from one another. The intuition behind this is that users expect a document matching a set of query keywords to be ranked higher than a document

matching the same set of query keywords if the set of the keywords appear closer to each other in the first document as opposed to the second document.

[5] has proposed a few different definitions for the notion of keyword proximity such as: span-based proximity, minimum coverage proximity, minimum pair distance proximity, average pair distance proximity and maximum pair distance proximity. It is shown in [5] that minimum pair distance proximity is the best measure among all other measures when incorporating it into the relevance ranking functions for text documents. In other words by incorporating it into the state-of-the-art document retrieval models it can enhance the effectiveness of the ranking function by promoting scores of documents in which the matched query keywords appear close to one other.

Minimum pair distance proximity: [5] defines minimum pair distance proximity as the smallest distance of all the pairs of distinct matched query keywords. It is denoted by *MinDist*(*Q*,*D*) and read as minimum keyword pairs distance of document *D* with respect to query *Q*. The formal definition is as follows:

$$MinDist(Q, D) = min_{k1,k2 \in Q \cap D}\{Dist(k1, k2)\} \quad (3.1)$$

where $k1 \neq k2$ and $Dist(k1, k2)$is the length of the shortest segment between *k1* and *k2* (i.e. *k1-k2* segment). For example, assume we have a document D_1 consisting of keywords {k_1, k_2, k_1, k_3, k_2, k_1, k_4, k_5} and keyword query Q_1={k_1, k_2, k_4}. Then *MinDist*(Q_1,D_1) = *min*{*Dist*(k_1, k_2)=1, *Dist*(k_1, k_4)=1, *Dist*(k_2, k_4)=2}=1

3.2.1 Incorporation of Keyword Proximity into the Relevance Ranking Function

We cannot simply add proximity measure values to the values of ranking function (in order to reward the documents with small proximity of matched keywords) since these two quantities are not comparable as explained in [5]. Therefore, the proximity function (i.e. *MinDist* for our case) should be transformed to a function that produces values that are comparable with relevance

ranking scores and would reasonably impact the relevance values. Let γ be the transformation function and $\tau(Q, D)$ be the new transformed function also referred to as adjustment factor by [5]. Therefore, we will have: $\tau(Q, D) = \gamma(MinDist(Q, D))$. As proposed by [5] there are two criteria that must be met by the new transformed function$\tau(Q, D)$:

1) $\tau(Q, D)$ should positively impact the relevancy of the document to the query by promoting its relevance ranking score. (i.e. the smaller the *MinDist* the larger $\tau(Q, D)$).

2) The effect of *MinDist* on τ should drop quickly as the distance gets smaller past some point and its effect should become constant as the distance becomes larger beyond some point. These two constraints lead to the following definition for τ defined by [5]:

$$\tau(Q, D) = \log(\alpha + \exp(-MinDist(Q, D))) \quad (3.2)$$

I have adapted this definition for τ as our adjustment factor to be added directly to the ranking function, Formula 2.2. As we will see in the experiment section the new ranking function has improved the effectiveness of the keyword search over relational database.

3.3 Keyword N-Grams

Many users searching for information using keyword search might misspell the query term or might only know the partial spelling of the keyword(s). More importantly these keywords might have been misspelled in the target text attributes' values that the search is performed against. To demonstrate these two problems, consider the following three examples:

1) Assume a user is searching for action movies featuring actor *John Travolta* and is not sure about the correct spelling of the name of the actor. Therefore, he might perform his search by typing for example keyword sequences such as "*Actions Travelta*" or "*Actions Traveltha*" instead of "*Actions Travolta*". This could result in the failure of finding the intended inter-

connected tuples depending on which SQL predicate (such as *LIKE*, *CONTAINS* or *FREETEXT*) was used to implement the search system.

2) Assume a user is searching for thriller movies featuring a German actress *Martina Gedeck*. If the targeted movie database has inconsistencies in how the actress name has been spelled (e.g. *Gedack*, *Gedek* and etc.), this could negatively impact the effectiveness of the keyword search as the *tf* and *idf* factors of Formula 2.2 will not be accurate due to possible mismatching between query keywords and the keywords in the text attributes' values.

3) Similar to the second problem above, if the values of a target text attribute contains different variations of the same verb or noun (e.g. verb category and its variations such as categories, categorization and etc), this could also negatively impact the effectiveness of the keyword search as the *tf* and *idf* factors of Formula 2.2 will not be accurate due to possible mismatching between query keywords and the keywords in the text attributes' values.

To the best of our knowledge neither the search components nor the IR-style ranking schemes of the previous works on KSRDBs have addressed the mentioned issues. To address these issues we have computed the N-grams (in particular quadgrams) of the keywords both in the values of the target text attributes and in the query itself. We then incorporated the quadgrams of the keywords to the Formula 2.2 by updating the *tf* and *idf* not only when we encounter the exact search terms but also when we encounter the variations of the corresponding terms generated by the quadgrams. Our experiments show that the modified version of the Formula 2.2 will further enhance the keyword search effectiveness. The final ranking algorithm is shown in Figure 3.3.

Algorithm2: Relevance Ranking Algorithm
Input: Keyword query *Q*, a set of answer tuple trees *T*'s
Output: A ranked set of answer tuple tree.
1: *T-Set //* set of answer tuple trees *T*'s
2: *T-Ranked-Set //* a priority queue to store the set of
3: // ranked answer tuple trees *T*'s
4: *Q* // query keywords
5: *k_Quadgrams* //a set of all the quadgrams for keyword *k*
6: *For each* k_i in *Q*
7: computeQuadgrams(k_i)
8: k_i*_Quadgrams.update*
9: *end for*
10: *For each* k_i in each text_attribute
11: computeQuadgrams(k_i)
12: k_i*_Quadgrams.update*
13: *end for*
14: *For each T* in *T-Set*
15: *T-Ranked-Set* .push(*T, Score'*(*T*,*Q*)) // *Score'*(*T*,*Q*) is
16: // the modified version of Formula 3 after applying
17: // Quadgrams and $\tau(Q, d)$ to Formula 2.
18: *end for*
18: *Return T-Ranked-Set*

Figure 3.3: Relavance Ranking Algorithm.

3.4 Experimental Results

We used the IMDB [9] dataset to perform our experiments. We designed and implemented a relational database corresponding to the IMDB schema and populated the database with a small portion of raw text files downloaded from IMDB. In our schema we included *plot_summary* text attribute which has long string values in order to be able to create queries for evaluating the effect of *Keywords Quadrams* and *keywords proximity* which are both more effective on text attributes containing long strings. The IMDB schema we used is shown in Table 3.4.

Table 3.4: IMDB dataset.

IMDB Schema	# of Rows
Actors(actorID, actor)	2000
Directors(directorID, director)	1200
Movies(movieID, title, directorID, summaryID)	4000
Cast(movieID, actorID)	8345
MovieCategories(movieID, genre)	6126
PlotSummary(movieID, summary)	4000
ActorPlay(*actorID*, *character*, *movieID*)	7310

When populating our database we purposely misspelled the names of some of the actors/directors/characters/titles and in the plot summary of the movies, we changed some of the terms to different forms (but all generated from the original terms). We created two types of queries; 1) *Type I* queries, targeting both short and long text attributes. 2) *Type II* queries, only targeting short text attributes (Please see Table 3.4.1 below). We then formulated 50 queries, 25 per each type. To assess the effectiveness of our approach with comparison with the previous works we used two measures: 1) Number of top-1 search results that are relevant denoted by #Rel in Table 3.4.2. It shows how well the system retrieves one relevant answer. This metric is used for ranking tasks in which the user is looking for a single or a very small set of relevant answers in a large collection [6]. We chose this metric to evaluate *type I* queries since the user's primarily intention is to find a single movie. For example a user searching for a particular movie which has forgotten the name for, but remembers the genre of the movie and knows what the movie is about, would perform the search by entering the genre of the movie and few keywords describing the movie (e.g. query 4 in Table 3.4.1). 2) 11-point precision/recall (i.e. precision at recall level of 0.1). This measure shows the effectiveness of our system in retrieving *top10* answers. We chose this metric to evaluate *type II* queries since the user's primarily intention is to find a set of relevant movies. For example a user searching for a set of movies in a certain

category in which a particular actor has played, would perform the search by entering the genre of the movie and the name of the actor (e.g. query 7 in Table 3.4.1).

Table 3.4.1: Query Types.

	Type I queries		Type II queries
1	Summary	7	actor, genre
2	Summary, director	8	actor, director
3	Summary, actor	9	director, genre
4	Summary, genre	10	actor, director, genre
5	Summary, actor, genre	11	actor, character
6	Summary, character		

In order to identify the relevant answers in our database we use pooled relevance judgment used in [3] as follows; We ran all four algorithms for each query (*BA*: base algorithm Formula 2.2, *BA+KP*: base algorithm + keywords proximity, *BA+KQ* = base algorithm + keywords quadgrans and *BA+KP+KQ* = base algorithm + keywords proximity + keywords quadgrams) and merge their *top20* results. We then manually judged and selected relevant results for each query out of the 80 candidate results. We chose pooled relevance judgment as our standard for evaluation because only the users can determine if a search result satisfies the query's need or not.

To evaluate the effectiveness and impact of keyword proximity and keyword quadgrams, we purposely submitted 12 out of 25 queries containing misspellings for *type II* queries. For *type I* queries, we submitted 12 out of 25 queries containing misspellings and phrases with different keywords' variations. Table 3.4.2 shows the number of top-1 search results for *type I* queries for each algorithm and Figure 3.4 shows the 11-point precision/recall graph for *type II* queries for *BA* and *BA+KP+KQ* algorithms.

Table 3.4.2: Impact of keyword proximity and quadgrams on number of top-1 result

	BA	*BA+KP*	*BA+KQ*	*BA+KP+KQ*
#Rel	8	12	13	16

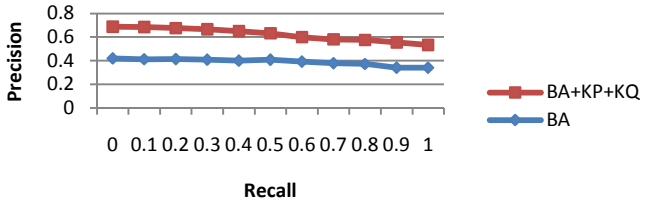

Figure 3.4: 11-Point Precision/Recall.

We can see from Table 3.4.2 that *BA+KP+KQ* algorithm outperformed the base algorithm *BA* for *type I* queries. We observed that many of the relevant answers to the queries targeting the *plot summary*, which contained misspelled keywords or keywords which were located apart from one another, were not in the top-1 search results for *BA*, and in fact they were not even in the top-10 results. We can also see from Figure 4 that *BA+KP+KQ* algorithm outperformed the base algorithm *BA* for type II queries as well.

Please refer to [41] for the full paper of this research work.

CHAPTER 4

TUPLERECOMMENDER: A RECOMMENDER SYSTEM FOR RELATIONAL DATABASESA

4.1 Introduction

An important and challenging task in any keyword-based search system in text documents or relational databases is the capability of the system to find additional search results besides the actual search results containing the searched keywords and present them to the user as search recommendations. This way the records that might be of interest to the user can be discovered. Most recommender systems such as *Amazon.com* and *IMDB.com* rely heavily on the users' ratings and previously learned patterns from the users and their selected items to find the search result recommendations.

In this research work we present a system called TupleRecommender which searches a relational database for a given keyword query and finds the search recommendations based on the similarity of the tuples with respect to the tables' attributes in which the search keywords were found , without relying on the previously learned patterns or users' ratings.

There has been an extensive research work in recommender systems using different approaches. The two most widely used state-of-the-art approaches are: 1) content-based filtering and 2) collaborative filtering. Many of these approaches have been very successful in terms of finding recommendations effectively. Our main goal in this work is to study a new approach in finding recommendations, which is purely based on what the user is currently searching for without using any information stored about the user. We then evaluate the effectiveness of this approach using a user-item rating matrix that we generated from the movie rating files downloaded from MovieLens movie recommender system [17].

4.2 Background on Recommender Systems

This section gives an overview of recommender systems in general (not particularly in the context of relational databases) so that we can compare our approach described in the next section with the existing approaches and point out its benefits.

The two widely used methods in recommender systems are content-based filtering [13,14] and collaborative filtering [15]. Content-based filtering, also called information filtering, uses content of the items that the user has searched for in the past in order to infer the types of the items that the user might be interested in. For example, keywords in a movie description, movie review or any other content of a movie could be used to find similar records as search recommendations. In contrast, collaborative filtering recommends items which are chosen by similar users. In collaborative filtering systems, similarities of the users are computed while in content-based filtering the similarities of the items are computed.

Both approaches have some shortcomings. Content-based recommendation systems could suffer from a situation called data sparsity which is referred to as the problem of insufficient data. This is due to the fact that these sorts of systems rely solely on the previously rated items of the user for whom recommendations are currently being made. Therefore, if there is insufficient data, the system will not be able to make accurate recommendations. Collaborative filtering systems resolve some of the issues that content-based filtering systems have. For instance, they usually don't suffer from data sparsity as often since an active user's (i.e. user for whom the recommendation is being made) behavioral data is compared against more data than in the content-based system. In the latter system, an active user's data is compared to other users' behavioral data to find items that like-minded users are interested in, rather than being limited in its recommendation to data specific to that active user.

However, both methods could suffer from a situation called cold-start, which is referred to a problem of having difficulty of recommending new items for which there are not enough ratings

available. To overcome these issues hybrid systems have been introduced which incorporate the advantages of both approaches while eliminating the shortcomings of both [16].

All of the above approaches could be biased towards a particular set of items rated by specific users. For example, users belonging to the same demographic regions, same gender or same age group could potentially have very similar ratings. This poses an important problem, as the recommendations made could be geared towards that specific set of users. For example a user from demographic region R_1 interested in item I_1 which is also liked by users from region R_2, may be getting recommendations which are not of interest to her, because the recommendations are based on the preferences of users of a different demographic region, R_2.

Our approach is different than the approaches above since we do not rely on historical data or previously learned patterns. If we have sufficient data presenting the database then the system identifies the similar items to the searched item(s) with respect to the features containing the query keywords. This approach does not create bias with respect to the demographic information. For example two users from totally different demographic locations who like the movie Terminator 1, will be both most likely interested in the movie Terminator 2, as our system would recommend. This is because our system takes into account the features of these two movies when computing the similarity (i.e. actor, director, film award, etc)

4.3 Proposed Approach

An effective technique to find search recommendations in a relational database is to use a clustering method to find similar results to the one(s) containing the query keywords and present them to the user as search recommendations. However, since the database features do not form a Euclidean space, traditional clustering methods cannot be used for this purpose. TupleRecommender adapts a data mining technique used in cross-relational clustering, called CROSSCLUS [12], which clusters the results based on their similarity with respect to the feature(s) as explained later in this chapter. TupleRecommender finds the interconnected tuples containing the query keywords using techniques for keyword search over relational databases

(KSRDBs) explained in chapter 2 and 3. Once the join-trees containing the keywords are found, features relevant to these keywords are identified and used for search recommendation purpose.

To find the query results containing the keywords we use techniques used in keyword search over relational databases. Similar to a web search engine such as Google that requires the user to enter a set of keywords to find documents containing the keywords, in keyword query over relational databases the user enters a set of keywords to find inter-connected tuples containing the keywords, which, if found, are presented to the user. Finding search keywords in relational databases proceeds in two steps. In the first step, join tree generator generates all possible join trees which are relational algebra join expressions. In the second step, Plan Generator generates plans to evaluate join trees and find the matching rows containing all the keywords using SQL. TupleRecommender uses this approach to compute the join trees and to find the inter-connected tuples containing the keywords before finding the search recommendations. Once the system locates the query results, it then identifies those features containing the query keywords and uses them to find similar results and present them to the user as search recommendations. As mentioned above, TupleRecommender does not rely on previously learned patterns from users' searches or their ratings to find recommendations. Instead it computes the similarity of two tuples with respect to the features that the query keywords were found in. The basic intuition behind this approach is that when a user searches a database for some information in the form of keyword query, the tables' attributes (i.e. features) containing those keywords should be used to find other similar records. For example, a user searching a movie database for some *action* movies featuring actor *Bruce Willis* would submit a query like: *"Bruce Willis Action,"* and would expect similar movies with respect to the actor and genre of the movie to be found, or if she searches for *documentary* movies directed by *Errol Morris,* she would expect similar movies with respect to the director and the genre to be returned.

4.4 Brief Summary of Cross-Relational Clustering with User's Guidance (CROSSCLUS)

A quick summary of CROSSCLUS is as follows (for detailed descriptions please refer to [12]): CROSSCLUS takes a user query containing a target table and one or more pertinent attributes. For example consider the database schema depicted in Figure 4.4 and the query "**cluster** *Directors* **with** *MovieCategory.genre*". In this query *Directors* is the target table and *genre* is the pertinent attribute. The goal is to cluster the records in Director's table with respect to the *genre* feature in MovieCategory's table.

There are a few terms and notations defined in CROSSCLUS [12] that we will make use in this research work and for that purpose we will restate them below.

A multi-relational feature f is defined as a path from the target relation to the relation D containing the pertinent feature via joins on the primary-foreign key relationships. For example in the query above the multi-relational feature *genre* is defined as (*Directors*⋈*Movies*⋈*MovieCategory.genre*).

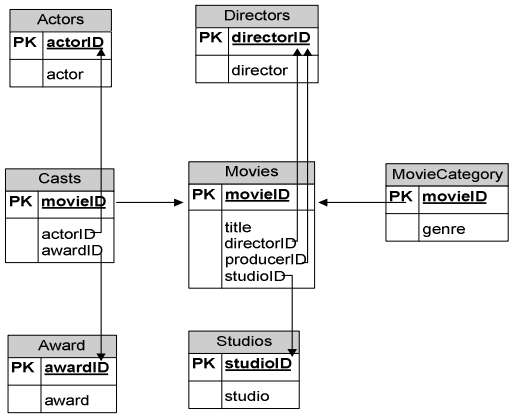

Figure 4.4: portion of IMDB Database Schema.

For a target tuple t, $f(t)$ is used to indicate t's value with respect to f. A set of Tuples in relation, D, joinable with t, via the join path of f, is represented by $D_f(t)$. As discussed in

CROSSCLUS, feature f can be either categorical or numerical. Let's assume feature f is a categorical feature (even though it is easy to adapt the method to consider numerical features) and t is a target tuple, then $f(t)$, value of t with respect to f, represents the distribution of t values among $D_f(t)$. To help understand the meaning of the terms defined above consider the summary of few instances of the IMDB database depicted in Table 4.4.

Table 4.4: An instance summary of IMDB database.

DID	SF	A	R	D
D1	20	5	10	5
D2	10	0	12	0
D3	0	0	2	0

DID=DirectorID SF= Science Fiction, A= Action R=Romance, D=Drama

The table shows the number of movies each director has directed in each category assuming categorical feature *genre* has four values, {Science Fiction, Action, Romance and Drama}. For example director D_1 has directed 10 romance movies and 5 action movies and so on. Therefore,

$$f(t = D_1) = (Science\ Fiction = 0.5, Action = 0.125,\ Romance = 0.25, Drama = 0.125).$$

4.4.1 Tuple similarity with respect to a feature

As described in [12], given two tuples t_1 and t_2 and multi-relational feature *f*, similarity between t_1 and t_2 with respect to *f* is represented by $sim_f(t_1,t_2)$ and defined in Formula 4.1 below:

$$sim_f(t_1,t_2) = \sum^{v}_{k=1} f(t_1).p_k \times f(t_2).p_k \quad (4.1)$$

where *k,* is the number of values a categorical feature *f* can have. For instance, feature *genre* from our example above has four different values (SF, A, R, D). $f(t).p_i$ is the proportion of tuples that join with *t* via join-path of *f* (defined earlier) that have value v_i on *f*. This similarity measure is very similar to the *cosine similarity* for two vectors.

To illustrate this equation consider calculating the similarity between tuples D_1 and D_2 with respect to *genre*:

$Sim_{genre}(D_1,D_2)$=(20/40x10/22)+(5/40x0)+(10/40x12/22)+(5/40x0) = 0.405

4.5 OVERVIEW OF THE TUPLERECOMMENDER SYSTEM ARCHITECTURE

TupleRecommender consists of two main components: the search component and the search results recommender component as depicted in Figure 4.5.

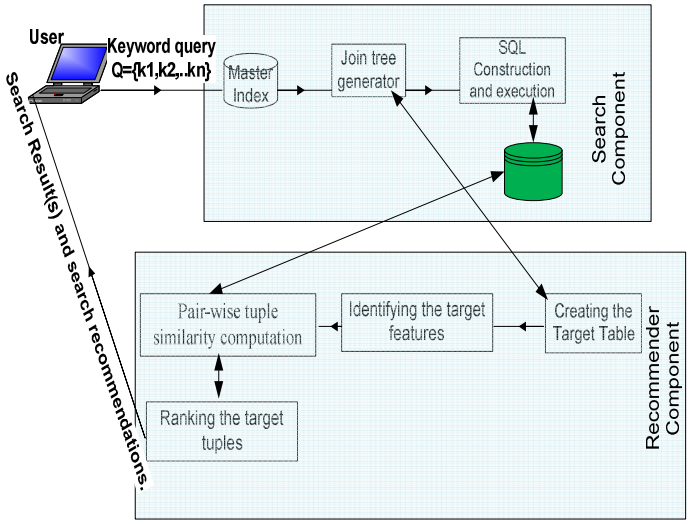

Figure 4.5: TupleRecommender System Architecture.

4.5.1 Search Component

Search component of TupleRecommender is very similar to DISCOVER [2] and DBXplorer [4]. A master index, also called symbol table, which is a table that stores keywords and the locations of the keywords in the database is built. Symbol tables have been implemented differently by different systems. Some store the column id in which the keywords are located and some store the cell id (row-column id). Each implementation has its own tradeoffs in terms of the time and space that it takes to build the master index versus the time it takes to look up a keyword in the

master index. Given a keyword query $Q = \{k_1, k_2, \ldots, k_n\}$ by the user, the search is performed in three steps. In the first step the keywords are looked up against the master index to identify the locations of the keywords in the database (i.e. tables, columns, rows, cells, depending on how the master index has been implemented). In the second step join trees are computed from *G*. In the third step, SQL statements are constructed and executed to identify and fetch all the matching rows which are essentially inter-connected tuples joined on their primary-foreign key relationship from different relations across the database containing all the keywords. Once these rows are fetched, they are pre-processed for the next step which is the search result recommendation step.

4.5.2 Search Result Recommender Component

A virtual table (a view), is created for each candidate network by joining the tables involved in that join tree. This view is essentially the target table denoted by T_t and its tuples are the target tuples denoted by t_t. In each join tree containing answer(s) to the search query, all the features containing query keyword(s) which are also connected to a *categorical* multi-relational feature f_c are identified. Let's call these features target features and denote them by f_t. Each of the target features f_t belongs to its own target relation denoted by R_t. Also, let's denote the value of the interconnected tuple(s) returned from the search component on the target feature f_t by f_v. Now, the pair-wise tuple similarity between f_v and every single tuple in R_t with respect to its corresponding f_c is calculated using Formula 1. This gives us a good similarity measure to find other interconnected tuples in the target table T_t similar to the one(s) returned from the search component.

To help understand this step, consider the database schema for the IMDB depicted in Figure 4.4 and an instance of the database depicted in Table 4.5.2. Assume the user is searching for movies featuring actress *Zoe Saldana* and directed by *James Cameron*. The keyword query entered by the user would be *Q*={"*James Cameron*", "*Zoe Saldana*"}. The join tree containing

these keywords would be *Actors-Casts-Movies*. Below is the search result returned by the system:

Title	Actor	Director	Studio	Genre
Avatar	Zoe Saldana	James Cameron	Lightstorm Entertainment	Science Fiction

Two target features that are connected to a multi-relational categorical feature are f_{t1}=*actor* and $f_{t\ 2}$=*director* that are both connected to categorical feature f_c =*genre* via *Directors-Movies-MovieCategory* and *Actors-Casts-Movies-MovieCategory* join-paths respectively. The values of the search result on target features are f_{v1}="*Zoe Saldana*" and f_{v2}="*James Cameron*".

Now, the pair-wise tuple similarity between f_v and every single tuple in R_t with respect to its corresponding f_c is calculated for both f_{v1} and f_{v2} using Formula 1 as shown below:

$Sim_{genre}(Z.S,A.S)=0$, $Sim_{genre}(Z.S,S.W)=1$,

$Sim_{genre}(Z.S,D.J)=0$, $Sim_{genre}(Z.S,L.D)=0$,

$Sim_{genre}(J.C,C.R)=0.25$, $Sim_{genre}(J.C,J.A)=0.5$

Lastly, the tuple similarity between the search result(s) and each tuple in the main target table T_t, is calculated by adding up the similarities for target features' values, f_v's, in both tuples, the search result tuple and the tuple that the similarity is being calculated for:

Sim(,1)=0.25, Sim(*,2)=1.5, Sim(*,3)=2,*

Sim(,4)=0.25, Sim(*,5)=1, Sim(*,6)=1*

As suggested by the system for this very small instance of the database, movie *"Aliens"* is the most similar one to the movie *"Avatar"*.

Table 4.5.2: An Instance of IMDB Database

RowId	Actor	Title	Studio	Director	Category
*	Zoe Saldana	Avatar	Lightstorm Entertainment	James Cameron	Science Fiction
1	Arnold Schwarzenger	Eraser	Kopelson Entertainment	Chuck Russel	Action
2	Zoe Saldana	Star Trek	Spyglass Entertainment	J. Abrams	Science Fiction
3	Sigourney Weaver	Aliens	Brandywine Productions	James Cameron	Science Fiction
4	Dwayne Johnson	The Scorpion King	Universal Production	Chuck Russel	Action
5	Arnold Schwarzenger	The Terminator	Hemdale Film	James Cameron	Action
6	Leonardo Dicaprio	Titanic	Lightstorm Entertainment	James Cameron	Romance

The recommendation algorithm is shown in Figure 4.5.1 below. Note that this algorithm is executed for each set of search results and their associated join-trees.

Algorithm1: Tuple recommender algorithm
Input: Set of search result(s), t_t's, and their associated join trees, T_t's.
Output: Sorted list of t_t's with the top one being the most similar one to the search result(s)
Step1: Identify the target features
1: $F_T[] = \emptyset$ //set of target features
2: *For each* $f_t \in T_t$ containing a keyword *do*
3: *if* (f_t connects to a f_c via a join-path) *then*
4: $F_T = F_T \cup f_t$
5: *end if*
6: *end for*
Step2: Pair-wise tuple similarity computation
8: simArray[][]= $\emptyset$ // 2-dimensional array for each
9: // target feature to store similarities.
10: for each $f_t \in F_T$
11: for each $t \in R_t$
12: simArray$_i$[][] = $sim_{fc}(f_v, t)$
13: end for
14: end for
Step3: Sorting the target tuples based on their similarities to the search result, $Result$.
15: for each $t_i \in T_t$
15: $sim(Result, t_i)$ = 0
16: for each f_t
17: $sim(Result, t_i)$ += simArray$_i$[][]
18: end for
19: List[] = $sim(Result, t_i)$
20: end for

Figure 4.5.1: TupleRecommender Algorithm.

4.6 Experimental Results and System Evaluation

We used a laptop pc of 2.26GZ CPU and 2GB of RAM with windows XP as our development machine. We used portion of IMDB dataset to test our algorithm. We downloaded the plain text data files from http://www.imdb.com/interfaces , designed and implemented IMDB relational database corresponding to the text data files and populated them with the data. The backend database used is MS SQL Server 2008. We used Visual Studio 2008 IDE as our development platform. ASP.NET and C# were used to create the frontend shown in Figure 4.6 below.

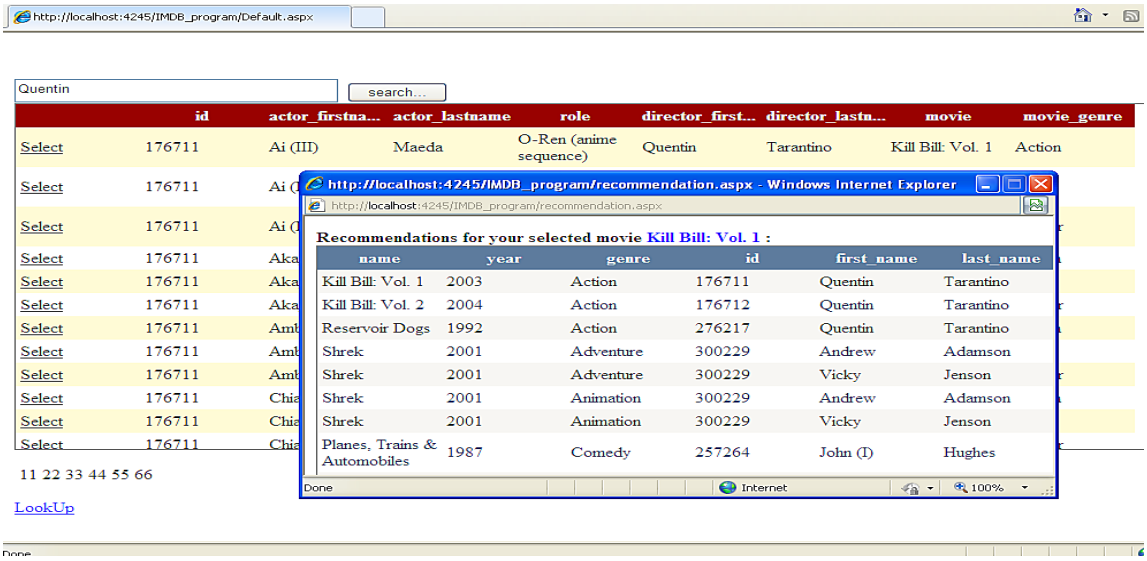

Figure 4.6: TupleRecommender frontend.

To evaluate our system we used datasets from IMDB [18] and MovieLens [17].

We used a subset of both datasets obtained by joining them on movie title. IMDB schema was used to create and populate our backend database. The rating scale in MovieLens dataset is from 0 to 5. We converted this scale to binary scale (i.e. 0-1) choosing 4 as the treshhold to generate a user-item binary rating matrix and used it as our ground truth. Rows represent the users and columns represent the movies. We used 100 records (i.e. users) of this matrix each of which with more than 20 rated movies to evaluate our system.

Four experimental methods called Given 2, Given 5, Given 10 and All but 1 were introduced by [15]. For the Given x method for each user *U*, x randomly chosen items are given to the recommender algorithm and the remaining items are withheld for the evaluation. We only need to give one item (i.e. movie) to our recommender algorithm which is chosen randomly and the rest of the movies are withheld for the evaluation purposes. The evaluation was performed as follows:

For each one of the 100 users if u_i likes the randomly chosen movie m_i (i.e. if in the binary rating matrix the entry for u_i-m_i is 1), we formulate a keyword query targeting m_i, using queries Q_1 to Q_6 shown below and then use it to query our database:

Q_1 = {actor, director, genre, studio}, Q_2 = {$actor_1$, $actor_2$, $actor_3$, director, genre}, Q_3 = {$actor_1$, $actor_2$, director, genre}, Q_4 = {actor, director, genre} Q_5 = {actor, director, genre, award}, Q_6 = {actor, director}

We then recommend the first ten movies returned by our system as the recommendations for the searched movie. True positive (TP) would be the intersection of the remaining of the movies which are rated 1 by user u_i (i.e. the u_i-m_i entries of 1's) and the set of movies returned by TuppleRecommende. Figure 4.6.1 shows histograms representing the average precision for the 100 users and for each query Q_i. We ran experiments for two different sets of movies, one with size of 200 and one with size of 400. As we expected the more movies representing the database the higher precision we get.

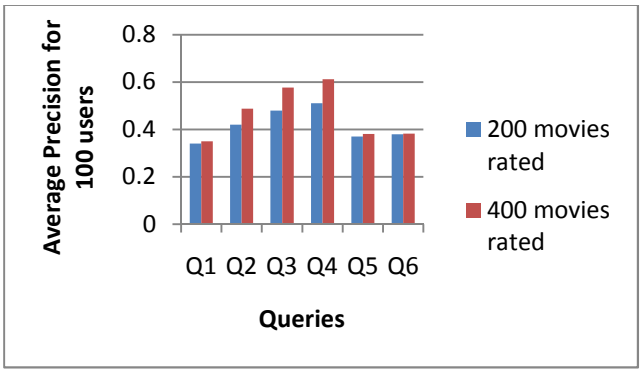

Figure 4.6.1: Average Precisions for 100 users using queries Q1 to Q6.

For the full paper of my work please refer to [42].

CHAPTER 5

OVERVIEW OF SEMANTIC SEARCH

5.1 Introduction

Semantic-based search promises to be more effective than traditional keyword-based search by providing more precise search results. This is because semantic-based search engines try to find and rank the search results based on the semantics of the keywords and their relationships with respect to one another. Whereas most traditional keyword search engines rank results purely based on the statistical information of the keywords (e.g. number of occurrences of the keywords in a document, etc.).

Different authors may refer to semantic search differently and it is important to be aware of its different meanings in the context in which it is being used. Semantic search is closely related to semantic web and used together in most literatures. Below is a definition of semantic search, from Wikipedia, in the context of semantic web: “authors primarily regard semantic search as a set of techniques for retrieving knowledge from richly structured data sources like ontologies as found on the Semantic Web. Such technologies enable the formal articulation of domain knowledge at a high level of expressiveness and could enable the user to specify his intent in more detail at query time.”[20]

This chapter gives a brief overview of Semantic web and semantic Search in general (not in the context of relational databases) and then discusses the incorporation of semantic search to keyword search. Next chapter presents our proposed research work in semantic-based keyword search over relational database.

5.2 Semantic Search

Let's explain what a semantic search is by comparing it with traditional keyword search using a simple example modified from the original example in [43] to illustrate what cannot be done using traditional keyword search.

Example 1: Assume we have a data repository which contains information about all of the students, faculty and staff of Wayne State University. This information includes the following: First name, last name, work address, home address, major pursued, classes taken, classes taught, faculty position/rank, GPA, etc. We are not concerned about where this information is stored. It could be in a relational database, an xml database, a text database, etc.

Assume there is a faculty member named "John Smith" who recently joined the computer science department. Also, assume there is a computer science student named "John Smith" who has been playing football for the WSU team for couple of years and has become very famous locally and nationally.

Now, assume a former friend of faculty member "John Smith" is searching for him to find out where he teaches. The only information he has about his friend besides his name is the fact that he has gotten a degree in computer science. Therefore, he performs the search using the traditional keyword search as follows: "John Smith computer science". Using the traditional keyword search and the ranking methods discussed in earlier chapters, the likelihood of not finding the intended John Smith is quite high. This is due to the fact that the statistical methods used in ranking function (Formula 2.2) would score the football player "John Smith" pretty high since he appears a lot more frequently in the text documents, database records, etc. versus the faculty member "John Smith".

On the other hand, semantic search allows the user to specify different properties (i.e. teaches a computer science course, isA a football player, isA faculty member, etc) of the entities (i.e. both

instances of “John Smith”) which he is searching for. Therefore, a semantic search for “John Smith” specified by the proper properties leads to the intended result(s).

Next section discusses what data structure is appropriate for data in order to enable semantic search.

5.3 RDF Model

Resource description model (RDF) has become the standard data model for storing and representing the data which is to be searched semantically (e.g. data on the semantic web). In an RDF data model, instances of people, places, etc. (which are called web resources or enteties) are represented as triples of the form subject-predicate-object. A subject is a web resource (e.g. John Smith from example above), a predicate is a property associated with the subject (e.g. teaches, isA, takes, lives, etc.) and an object which is either a web resource or a string literal (e.g. “CSC 2200”).

RDF model can be conceptually represented using a directed graph where the two nodes represent the subject and the object and the edge from the subject to the object denotes the relationship. To illustrate this, consider representing the scenario in example 1 using an RDF data graph depicted in Figure 5.3.

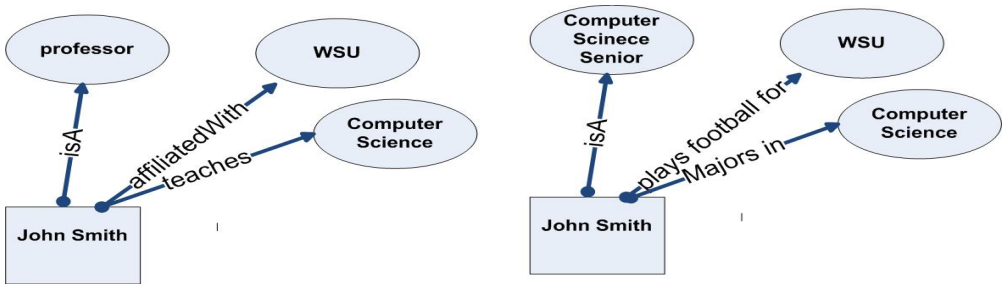

Figure 5.3: RDF data Model representing two different entities with their properties and their values.

In order to query an RDF data graph for a target web resource, a special query language called SPARQL, which is based on the notion of graph pattern matching, is used. With SPARQL query language users can begin their search for a web resource and iteratively refine their search by filtering out the web resources which do not have the properties they are looking for. [43]

For instance, in the example above when the user is searching for “John Smith”, he can refine the initial search results (i.e. entities) by choosing only the properties (from all available properties) associated with the John Smith that he is looking for, which in this case are: *teaches* with the value of *computer science*, *affiliatedWith* with the value of *WSU* and *isA* with the value of *professor*.

5.4 Semantic-based Keyword Search over Relational Databases

In order to perform semantic search over relational databases using keywords we have to incorporate semantic web technologies into keyword query in relational databases, since the techniques used in KSRDBs cannot search for the records semantically. There has not been much research work done to address this issue in relational databases. Few recent works in this area [21, 24, 25, 26, 27, 28, 29] have proposed approaches such as creating an ontological layer on top of the relational database’s schema or generating an RDF file from the relational database and then performing semantic search by using the SPARQL. Other works [30, 31] provide query interfaces over their backend RDF which support SPARQL. Some of the current approaches require the end user to know SPARQL in order to perform semantic search on their data which could be challenging for a non-technical user. SPARK [44] transforms user’s keyword query into SPARQL by mapping the query terms to the knowledgebase terms and then construct the formal queries. KEYMANTIC [45] tries to solve the problem of finding the best mapping from the query terms into the schema terms without a prior knowledge of the instances in the databases. “EXPOSING Relational Database as RDF” [46] has proposed an approach for exposing the data in an RDB into a virtual RDF. Some of the approaches and techniques that we have used in

semantic-based search over relational databases (presented in the next chapter) were inspired by SPARK and KEYMANTIC and [46].

Next chapter presents our proposed research work in semantic-based keyword search over relational database and shows how the proposed approach has significantly improved the search effectiveness versus the previous works.

CHAPTER 6

DBSemSXplorer: Semantic-based Keyword Search System over Relational Databases for Knowledge Discovery

6.1 Introduction

Our primary goal in this research work is to retrofit semantic search technologies onto relational databases in order to enhance the effectiveness of the keyword search. One issue with keyword search in general is its ambiguity which can ultimately impact the effectiveness of the search in terms of the quality of the search results. This ambiguity is primarily due to the ambiguity of the contextual meaning of each term in the query (e.g. each query term can be mapped to different schema terms with the same name or their synonyms). In addition to the query ambiguity itself, the relationships between the keywords in the search results are crucial for the proper interpretation of the search results by the user and should be clearly presented in the search results. Unfortunately, many of the existing approaches used in KSRDBs, neither consider the semantic similarity nor the syntactic similarity (in the case of exact matching) of the query keywords when mapping the keywords to the schema terms and, more importantly they may not present the relationships between the terms returned in the search results. Therefore, the user might not find the search result(s) she is looking for.

To address these issues we have designed and implemented a proof of concept prototype system called DBSemSXplorer which can answer the traditional keyword search over relational databases in a more effective way with a better presentation of the search results. We address the keyword search ambiguity issue by adapting some techniques for keyword mapping from the query terms to the schema terms/instances. The techniques we have used for term mapping capture both the syntactic similarity between the query keywords and the schema terms as well as the semantic similarity (e.g. definition of the keywords) of the two and give better mappings and ultimately more accurate results. Finally, to address the last issue of lacking clear

relationships among the terms appearing in the search results we have leveraged semantic web technologies to enable a faceted search interface and presenting the relationships between the keywords in the search results. Our system has also leveraged the inference/reasoning capability of the semantic web in order to enrich the knowledgebase (presented in chapter 7).

In short, given a schema and an instance of a relational database, we extract an ontology from the schema representing the tables as classes and the relationships between them. We then create and populate a knowledgebase (i.e. RDF) according to the extracted ontology and the instance of the relational database. The classes, the relationships between them, and classes' instances in the RDF are referred to as web resources. When a user inputs a keyword query, the similarity between each query term and each resource in the knowledgebase is calculated in order to find the overall best mapping from the query terms to the knowledgebase terms. Once we have the candidate resources in the RDF knowledgebase, we construct and execute the corresponding SPARQL query and present the search results to the user with clearly defined relationships among the terms appearing in the search results. We also present facets which represent different properties of each keyword which appear in the search result as explained in detailed in this chapter.

Our experiments show that the approach we have used in our system is more effective than the traditional keyword search approaches by enabling the users to find the search results which are more relevant to their keyword queries.

The remainder of this chapter is organized as follows: Section 6.2 outlines the motivation for our work by giving an example which demonstrates the problems with the current approaches in keyword search over relational databases. Section 6.3 presents the system architecture. Section 6.4 explains how we create an RDF knowledgebase from the RDB. Section 6.5 describes the techniques used in mapping query terms to the knowledgebase term. Section 6.6 describes the construction of the SPARQL from the matched resources. Section 6.7 presents the system

prototype and interface. Section 6.8 discusses the experimental results and system evaluation and section 6.9 concludes our work and gives direction for the future work.

6.2 Motivation

Traditionally keyword search over relational databases is performed by creating a master index which indexes all the relational database's attributes for which their values are to be searchable. When a keyword query is entered, it is looked up in the master index upon entry in order to locate the tuple sets across the database, which are the tables and their tuples (i.e. records) containing the query terms in at least one of their attributes. The tuple sets along with the schema graph of the relational database are then inspected to find the join trees. Join trees are the interconnected tuple-sets across the database joined on their primary key foreign key relationships. Lastly, each interconnected record from these join trees, which collectively contain all the query keywords, are returned as the search results please refer to [41] for detailed definitions of these terminologies and in depth explanation of the keyword search approach over relational databases.

To demonstrate the approach used in traditional keyword search mentioned above and to identify the issues with its effectiveness, consider the following example.

Example 1:

Consider the schema and an instance of the famous Internet Movie Database (i.e. IMDB) depicted in Figure 6.2.

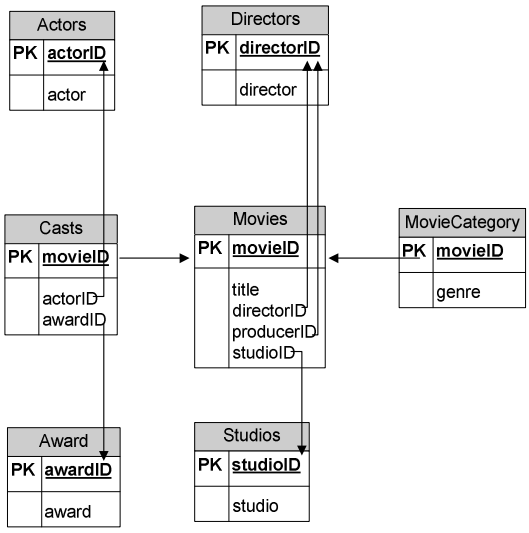

Figure 6.2.a: Portion of IMDB Schema.

Actor	Title	Studio	Director	Category
George Clooney	Ocean's Twelve	Village Roadshow Pictures	Soderbergh	Crime
George Clooney	Ocean's Eleven	Village Roadshow Pictures	Soderbergh	Crime
Zoe Saldana	Star Trek	Spyglass Entertainment	J. Abrams	Science Fiction
George Clooney	Leatherheads	Smokehouse Pictures	George Clooney	Comedy
Arnold Schwarzenegger	The Terminator	Hemdale Film	James Cameron	Action
R. Redford	Spy Game	Beacon Pics	T. Scott	Action

Figure 6.2.b: Portion of a sample Instance of IMDB

Now assume that the user wants to search for the movie(s) which are directed and stared in by George Clooney. She might formulate the keyword query as Q = {movies directed starred George Clooney}. With the traditional keyword search over relational databases there are two cases to consider: 1) If the logical OR is being used (i.e. not all the query terms are required to appear in the search result(s)), the search results would be rows 1, 2 and 4 (which is actually the intended search result) and 2) If the logical AND is being used (i.e. all the query terms are

required to appear in the search result(s)), then the user is presented with no search results. Even with case 1, the only keyword for which a term in the database has been found is “George Clooney” and all other keywords in the query are being neglected, since they are not found. Therefore, the user can potentially be presented with many results which are completely irreverent to her intended query. Also, depending on the ranking scheme used, the actual intended search result(s) might not even be in the top-k search results.

In our approach we have leveraged the semantic web techniques along with the keyword mapping techniques using both semantic and syntactic similarity of the keywords to improve its effectiveness by presenting more relevant search results. For instance, in the above query, our system will take advantage of the fact that “George Clooney” appears to be both a director and an actor with different properties such as “acting” and “directing” (which will be presented as search facets) and will try to find the results which are collectively relevant to the users’ keyword query. Please note that neither the keyword “directed” nor the keyword “starred in” may appear in any instance of the database (just as is the case in our example database above) and this is where the keyword mapping techniques and semantic web technologies have come into play.

The ideal search result(s) for the query example above would be the result(s) not only containing “George Clooney” but also presenting different properties of the keyword “George Clooney” (e.g. movies directed, movies played/starred in, etc.) that can help the user to pick the intended search results.

6.3 Overview of the DBSemSXplorer’s System Architecture

DBSemSXplorer consists of three main components: The Relational database to RDF convertor, the query Keyword to the knowledgebase resource mapper and the SPARQL query constructor as depicted in Figure 6.3.

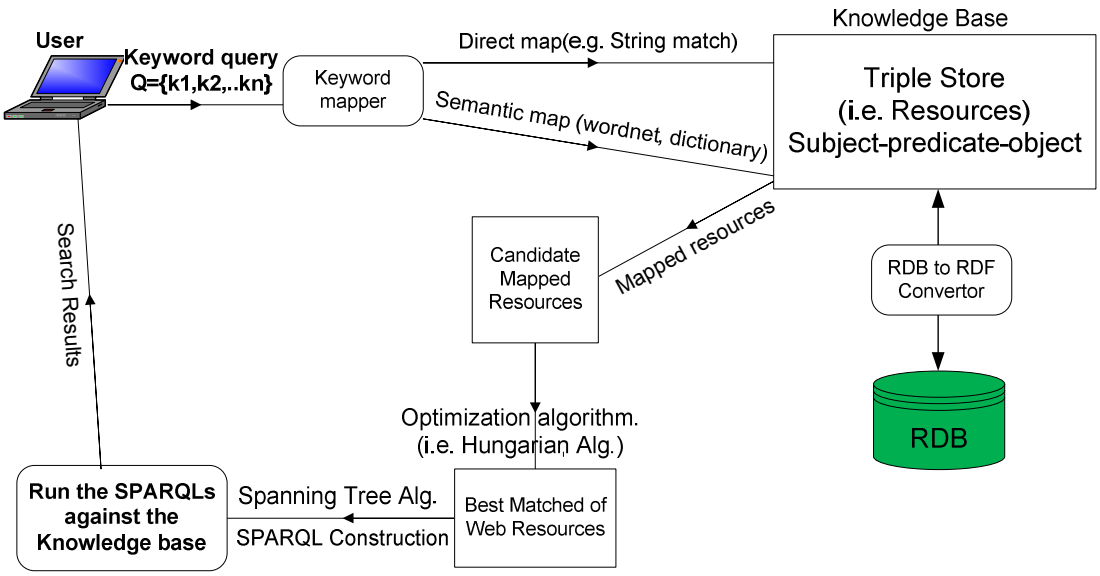

Figure 6.3: DBSemSXplorer System Architecture

6.3.1 Relational database to RDF convertor

The input to this module is a relational database's schema along with its instance. The module transforms the schema and the instance into a knowledgebase representing each relation as a class and the relation attributes as the class properties, as explained in the next section. The knowledgebase is essentially an RDF (i.e. a triple-store) in the form of subject-predicate-object in which SPARQL queries can be run against.

6.3.2 The Query Keyword to Knowledgebase Resource Mapper

The input to this module is the user's keyword query. The module uses different string comparison techniques to map each keyword to the most syntactically and semantically similar resource in the knowledgebase.

6.3.3 The SPARQL Query Constructor

The input to this module is a triple-store (i.e. RDF graph) along with the matched resources from the previous module. The module uses some graph theory algorithms to compute the minimum sub-graph containing the matched resource-nodes which are

connected. This sub-graph will be then used to formulate the SPARQL query which will be run against the underlying triple-store and the results are returned to the user.

6.4 Generating a Knowledgebase from the Relational Database

6.4.1 Basic Definitions and Concepts

In this section we give a few definitions and concepts used in semantic web technology which are essential to understand our approach.

Resource Description Framework (RDF): RDF is a standard Framework originally designed for representing information about the web resources. The model is based on the notion of triples, which are statements in the form of <*Subject*, *Predicate, Object*>. The **subject** of the statement is the resource that the statement is about, the **predicate** of the statement is the property or characteristic describing the semantics of the subject and the **object** of the statement is the value of that property. In other words any web resource can be described semantically using a set of triple statements. A **web resource** is anything that can be uniquely identified using a Uniform Resource Identifier (URI). Subjects and predicates of the triples are always URI web resources, whereas the objects can be either URI web resources or literals. Below is a simple RDF snippet describing the movie “The Terminator”.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix dir: <http://localhost:8080/IMDB/Directors#>
@prefix mov: <http://localhost:8080/IMDB/Movies#>
@prefix movcat: <http://localhost:8080/IMDB/MovieCategory#>
mov:The_Terminator rdf:type mov:movies;
mov:The_Terminator mov:name "The Terminator";
mov:The_Terminator mov:directedBy dir:James_Cameron;
mov:The_Terminator mov:category movcat:Action ;
        ……
```

There are four statements in this RDF snippet. Let's consider, for example, the second statement which has "The_Terminator" as its subject, "mov:name" as its predicate and "The Terminator" as its object. As you can see, the subject and the predicate of this statement are both web resources prefixed by "mov" (i.e. the abbreviation for the corresponding URI) and the object of the statement is a string literal. Now, consider the third statement, the subject and the predicate are web resources and, unlike the second statement, the object of the statement is also a web resource.

Resource Description Framework Schema (RDFS): RDF itself provides a set of vocabularies/terms such as rdf:type to describe the resources using simple statements. RDFS is a semantic extension to RDF which defines a broader set of vocabularies/terms such as rdfs:class, rdfs:domain, etc to enable the further enrichment of the knowledgebase. In other words, RDFS provides a way to semantically express resources and the relationships between them. Below is a simple RDF snippet with RDFS vocabularies added to describe the resource classes and relationships between them.

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix dir:   <http://localhost:8080/IMDB/Directors#>
@prefix mov: <http://localhost:8080/IMDB/Movies#>
mov:Movie               rdf:type       rdfs:class;
mov:The_Terminator   rdf:type       mov:Movie;
dir:Director              rdf:type       rdfs:class;
dir:James_Cameron    rdf:type       dir:Director;
mov:directedBy         rdf:type       rdf:Property;
mov:directedBy         rdfs:domain  mov:Movie;
mov:directedBy         rdfs:range     dir:Director;
```

Ontology: is a set of all the resource classes, their associated properties and the relationships between them which collectively define concepts and describe their interrelationships.

Knowledgebase: We define the knowledgebase as the collection of all the data instances represented by the triples along with the RDF/RDFS semantic vocabularies to describe the concepts and their relationships. We will use knowledgebase and triple-store interchangeably throughout the paper.

RDF Graph: RDF graph is a direct-graph G(V,E) representing the triple-store, where each vertex $v \in V$ represents either a resource or a string literal (ellipses for resources and boxes for literals) and each edge $e \in E$ represents the relationship between two resources or a resource and a literal. A small portion of the RDF graph of the IMDB example above is depicted in Figure 6.4.1 below.

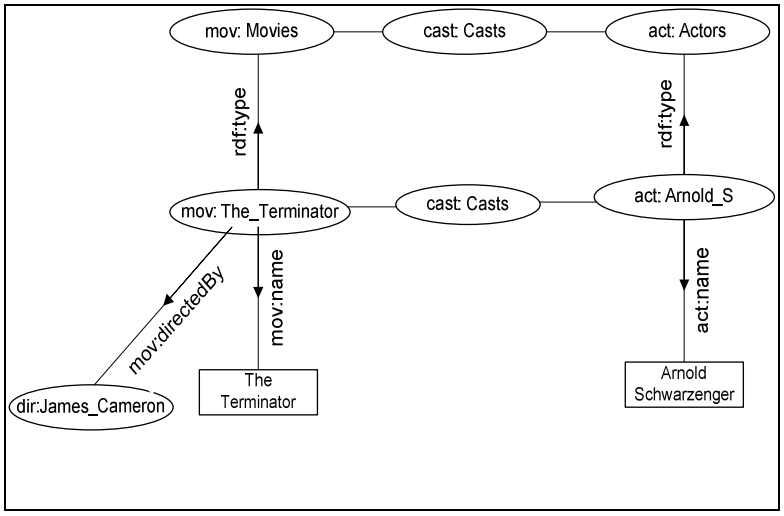

Figure 6.4.1: RDF graph for IMDB

6.4.2 Extracting an ontology from the database schema

Given a relational database schema, we want to extract an ontology which represents all the concepts and their associated properties along with the relationships which exist between them. Fortunately, the nature of the structured data (e.g. RDB) makes it less tedious to extract ontology from its schema as opposed to non or semi-structured data (e.g. text documents). Our ontology extraction approach is similar to approaches employed by commercial RDB2RDF (i.e. relational database to RDF) systems and other related research literatures [46].

6.4.2.1 From RDB schema to an RDF Model

We describe our approach through an example describing how the movie data can be modeled using a relational model. This example illustrates the similarities between the RDB model and RDF model.

Example: Consider how a relational database expert would model the scattered movie data into an Entity-Relationship model. She would start by extracting concepts from the data, finding the properties associated with each concept and, lastly, discovering the relationships between the concepts. For instance, she would create two entities (i.e. relations) for the concepts “Director”

and “Movie” respectively, with properties such as “firstName”, “lastName”, “DOB” for the “Director” concept and “movieTitle”, “year”, “genre”, “director” for the “Movie” concept. As we can see, some of the properties associated with “Movie” already exist as standalone concepts (e.g. Director Relation). In the E-R model, this is indicated by using a reference from “Movie” to “Director” (i.e. a foreign key). She will then start populating the database with respect to the extracted E-R model. Each instance (i.e. record) of each relation (i.e. table) should be identified uniquely by using a primary key.

The process of modeling the movie data into an RDF model is almost identical to this and that is what makes it natural to transform an RDB to an RDF. Each table in the relational model representing an entity becomes a class in the RDF model representing a concept and each table’s attribute becomes a class property. To transform the actual data stored in the RDB into an RDF model in the form of triples (i.e. S-P-O), we start by scanning each table row by row. Each row of a table which is identified uniquely by a primary key becomes a subject of a triple and, as we mentioned earlier, subjects are web resources which are identified uniquely by URIs. Therefore, we use the namespace indicating the table concatenated with the primary key of that row in order to identify the subject (e.g. http://localhost:8080/IMDB/Directors#1, where 1 is the primary key of a director). Each table’s column becomes a predicate which, as mentioned previously, has to be a web resource (e.g. http://localhost:8080/IMDB/Moviess#title, where title is the title of the movie). Lastly, the value of each column becomes an object which, as mentioned before, could be either a string literal or another web resource depending on whether the value is a foreign key (i.e. referring to another web resource) or a string. Below is an example showing three triples. The first triple states that the movie 5’s title is “The Terminator” which is a string literal and the second triple states that the movie 5’s director is “dir:4” which is a web resource itself. The third triple states that “dir:4” (from the second triple) has property “dir:name” with value of “James Cameron”.

Example:

```
@prefix dir:    <http://localhost:8080/IMDB/Directors#>
@prefix mov:  <http://localhost:8080/IMDB/Movies#>

<mov:5   mov:title “The Terminator”>
<mov:5   mov:director dir:4>
<dir:4   dir:name “James Cameron”>
```

Our algorithm for transforming an RDB to an RDF is shown in Figure 6.4.2.1 below:

Algorithm: RDB to RDF Transformation

Input: RDB schema, set of tables and the data stored in them
Output: an RDF Knowledgebase (triple store)
1: *S* // Relational database Schema
2: $table_i$ // A table in *Schema*
3: $attr_k$ // A column/attribute in a *table*
4: $table_{i\_PK}$ // The primary key for a $table_i$
5: $table_{i\_URI}$ // $table_i$ namespace
6: *KB* // Knowledgebase
7: *For each* $table_i \in S$
8: *For each* $attr_j \in table_i$
9: *if* ($attr_j$.*value* is *Literal*)
10: *SELECT* (“$table_{i\_URI}$” + $table_{i\_PK}$) , (“$table_{i\_URI}$” +”$attr_j$”), ($attr_j$)
11: *else if* ($attr_j$.*value* is *Foreign_Key*)
12: *SELECT* (“$table_{i\_URI}$” + $table_{i\_PK}$) , (“$table_{i\_URI}$” +” $attr_j$”) , (“$table_{r\_URI}$” + $attr_j$)
13: // $table_{r\_URI}$ is the URI for the referenced table.
14: *end for*
15: *end for*

Figure 6.4.2.1: RDB to RDF Algorithm.

Once we have the RDF graph generated, we index all of the web resources (e.g. classes, properties, instances) along with all of the string literals (e.g. names, labels, etc.) for quick access during the mapping phase described in the next section.

6.5 From the Query terms to the Knowledgebase terms

6.5.1 Basic Definitions and Concepts

After having the RDB transformed into a triple-store, we need to map the query keywords to the most syntactically and semantically similar terms in the knowledgebase. There are many different keyword comparison techniques such as stemming, N-grams, semantic comparison (e.g. synonyms) and Levenshtein distance used in a variety of applications. In particular, we have employed Levenshtein distance and semantic comparison in our system. Some of the techniques we have used have been inspired by previous works on Kemantic [45] and Spark [44].

Levenshtein distance: also referred to as string edit distance is a string comparison technique which measures the difference (e.g. distance) of two strings. Given two strings, it computes the minimum number of edit operations such as (insertion, deletion, replacement) required to transform one string into another one. We chose edit distance as our measuring metric since it is more sensitive to the sequential structure and phonetic segments in a word versus for instance the frequency per word method.

To compute the syntactic similarity of two strings we use the following formula:

$$similarity(str_1, str_2) = 1 - \frac{LevenshteinDist(str_1, str_2)}{maxLength(str_1, str_2)} \quad [6.5]$$

Semantic mapping: This type of mapping takes into account the definition of the query keywords when trying to find the corresponding knowledgebase terms by utilizing English dictionaries such as WordNet.

Assume that the query string consists of N terms; to compute the syntactic similarity we consider two cases when parsing the query string:

1) We tokenize the query string into N individual terms and then compute the syntactic similarity between each term in the query and each resource in the knowledgebase using Formula 6.5.

2) We generate a set of 2-term phrases/nouns from the query string by selecting the first and the second terms as the first phrase, the second and the third terms as the second phrase,.., and lastly the (N-1)-th and N-th terms as the (N-1)-th phrase. We then compute the syntactic similarity between each phrase and each resource in the knowledgebase.

For instance, consider the keyword query Q = {scary movie stars}. User's intention could be A) finding the "stars" of the movie called "scary movie" or B) finding the "stars" who have starred in scary movies. If we don't consider both 2 cases in which we parse the query string, we could miss the user's query intention depending on which one she means, whereas considering both cases would cover both query intentions:

Case 1 covers the query intention stated in B and case 2 covers the query intention stated in A. The choice of 2-term phrase versus 3-term, 4-term, etc. was based on our experiments and the observation of users' formulated keyword queries which were almost always string of individual terms or sometime mixture of individual terms and 2-term phrases/nouns. This can simply extended to compute different n-grams of the query terms. (i.e. n-gram of the entire query terms not within an individual term).

Each resource which has a similarity score of 0.4 or higher will be added to the corresponding keyword's matching set denoted by $Q_k^{\{\}}$ along with its similarity score. For example,

$$Q_{played}^{\{\}} = \{(\text{played in}, 0.9), (\text{acted}, 0.5), (\text{performed}, 0.5)\}$$

shows the set of the pairs of matching resources and their similarity scores for keyword "Played". We chose 0.4 as a threshold since about 80% of our test queries have shown that the resource with the syntactic similarity score of less than 0.6 was not intended by the query keyword.

We then look up each query keyword in a dictionary (e.g. WordNet) to find the semantically similar resources for that keyword, which if found will be added to the $Q_k^{\{\}}$ with a score of 0.5,

which is similarity score given to all the semantically similar resources to their corresponding query terms.

6.5.2 Keyword Mapping Techniques

As mentioned in the previous section, for each query keyword k_i its matching set $Q_{k_i}^{\{\}}$ contains the similar resources R_j's and their similarity scores $s_{i,j}$. This can be easily visualized as a bipartite graph whose nodes are divided into two disjoints sets *K* and *R* as depicted in the Figure 6.5.2 below:

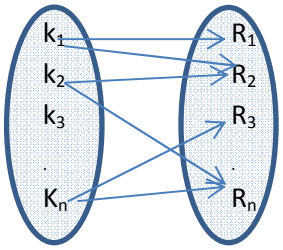

Figure 6.5.2: Bipartile graph modeling Keywords to Resources

Our goal is to find the best match from the keywords to the resources which would maximize the total similarity score over all keyword-resource pairs (i.e. (K_i,R_j)). This is very similar to the classical assignment problem in which an optimized matching (e.g. either maximum or minimum) is to be found. Please refer to Hungarian Algorithm [47] for a detailed explanation.

We create a similarity matrix with rows representing the resources and the columns representing the keywords. Each cell represents the similarity score for the corresponding resource- keyword pair at that row-column intersection. We run the optimization algorithm on the similarity matrix to find the best match to maximize the overall similarity score.

Below is an example to illustrate the assignment problem.

Consider the IMDB database from example 1 and assume that the user wants to know the actors playing in the movie "The Terminator". She formulates her search by submitting the

keyword query of Q={terminator stars}. The similarity matrix for the keywords and the corresponding knowledgebase terms (i.e. syntactically and semantically similar terms) is shown below:

	K_1=Terminator	K_2=Stars
R_1=The Terminator	0.71	X
R_2=Star Track	X	0.44
R_3=Actors	X	0.5

The optimized match for this example is $\{(\text{Terminator}, \text{The Terminator}), (\text{Stars}, \text{Actors}))\}$ with the first pair being the most syntactically similar pair and the second pair being the most semantically similar pair.

In the next section we explain how the formal query is constructed from the matched resources.

6.6 SPARQL Query Construction

6.6.1 SPARQL Query

SPARQL is a graph-matching query language for RDF datasets which enables the construction of queries consisting of variables and triple patterns. Syntax-wise, SPARQL is similar to the SQL query language containing SELECT and WHERE clauses. For instance, consider the keyword query from the previous example. An equivalent SPARQL query to return the actors of movie “The Terminator” is as follows:

```
SELECT ?actor
WHERE {
        ?movie   rdf:type     mov:Movies.
        ?movie   rdf:name   “The Terminator”.
        ?movie   cast:Casts  ?actor.
        ?actor    rdf:type      act:Actors.
       }
```

The SELECT clause indicates which variable(s) and their values are to be returned, which in this case is the variable “?actor”. The WHERE clause indicates which triple pattern(s) (i.e. subgraphs) are to be matched against the underlying RDF graph. There are two variables used in this query, “?actor” and “?movie”, representing the web resources which are either to be returned (e.g. ?actor) or to be matched against the underlying RDF graph in the WHERE clause (e.g. both ?actor and ?movie). The triple pattern for the WHERE clause of this example is depicted in Figure 6.6.1.

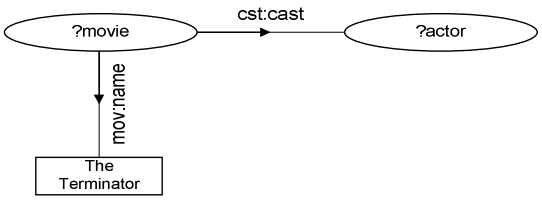

Figure 6.6.1: a sample Triple Pattern.

The SPARQL query returns the Actor resources “atc:Arnold_S”, “act:Linda_H”, “act:Michael_B” and “act:Paul_W” as the query result as shown in the table below. Please note that not all Actor resources are shown in the table due to the lack of space.

Actors
act:Arnold_S
act:Linda_H
act:Michael_B
act:Paul_W

DBSemSXplorer’s UI helps the user to drill down the returned resource(s) and to explore their different properties as illustrated in Section 6.7.

6.6.2 Construction of the SPARQL Query from the Matched Resources

After finding the best match of the set of the resources in section 6.5.2, we need to construct the equivalent SPARQL query to carry the user-intended keyword search. There are two sub-problems to this: 1) We need to find an interconnected subgraph which span all the matched

resources. This problem can be modeled as a minimum Steiner tree problem which is, given the set of all the vertices in RDF as V-vertices and the set of the matched resources as R-vertices, the goal is to interconnect the R-vertices by a subgraph of the shortest length. Returning back to the previous example, the optimized matched resources are $\{\text{The Terminator}, \text{Actors}\}$ where "The Terminator" is a string literal and "Actors" is a web resource class. The minimum Steiner tree spanning these resources is obtained by running the algorithm against the RDF supplying the matched resource vertices. 2) Once the spanning tree is obtained we construct the SPARQL query as follows: A) If the matched resource(s) are classes, such as "Actors" in this example, we treat them as wild cards by assigning variables (e.g. ?actor) to them which will hold the returned resources of those class types (e.g. Actors) satisfying the WHERE clause. B) Any resource class included on the Steiner tree path which was not necessarily matched during the mapping process, is treated as a variable (e.g. ?movie). C) If the matched resources are string literals such as "The Terminator" in this example, we use them as conditions inside the WHERE clause for the corresponding triple pattern.

6.6.3 Ranking Scheme

When mapping from the query terms to the schema terms as explained in section 6.5, for each set of query terms there could be more than one set of matched resources and therefore more than one constructed SPARQL. Each constructed SPARQL generates single result or a set of results which should be ranked differently. We don't rank the results for each SPARQL directly. Rather, we rank each SPARQL based on a few factors related to the matched resource set from which it was drawn as explain below:

1) The SPARQL constructed from a matched resource set which is more similar to the query terms, should be ranked higher. This is being done inclusively when we run the optimization algorithm (section 6.5.2) on the similarity matrix.

2) Our empirical results show that, a SPARQL constructed from a matched resource set which has more end-nodes (i.e. nodes containing string literals as opposed to variable nodes) should be ranked higher. For instance, consider the query Q = {scrary movie stars} given in an example earlier. The SPARQL constructed from the web resources “scary movie” as a string literal and “starts” as a variable should be ranked higher than the SPARQL constructed from the web resources “scary”, “movie” and “actors” all as variables. This is because often, an end user submitting a keyword query to a search engine formulates her query containing at least one known entity which leads to string literal during SPARQL construction.

(e.g. {Movies Arnold Schwarzenegger}, {Action Movies Travolta}).

6.7 System Prototype and Interface

We have implemented a web-based prototype for DBSemSXplorer using .NET technology. The backend database containing the IMDB data is MS SQL Server which is converted into a triple-store using our conversion algorithm. In the next versions of DBSemSXplorer, we are planning on providing web services to enable users to convert any relational database to an RDF by invoking the web service from a client program and have their RDB data semantically searchable via our system. We have used dotNetRDF, an open Source Library for .NET framework, as our SPARQL engine to issue queries against the target RDF. We have used ASP.NET and C# as code behind to implement the user interface using visual studio 2010. The development machine is a laptop pc with 4GB of RAM and CPU of 2.5 GHz. Figure 6.7 shows a couple of screen shots of the system UI and the presentation of the search results for the query of {Terminator stars}. Since the search results are generated from the triple(s) returned from a SPARQL query, they are in the form of subject-predicate-object. We translate this into a more human readable format but still having the web resources identified by underlining them. For example “Linda Hamilton” is a web resource which can be further explored for more information by clicking on it as shown in Figure 6.7b.

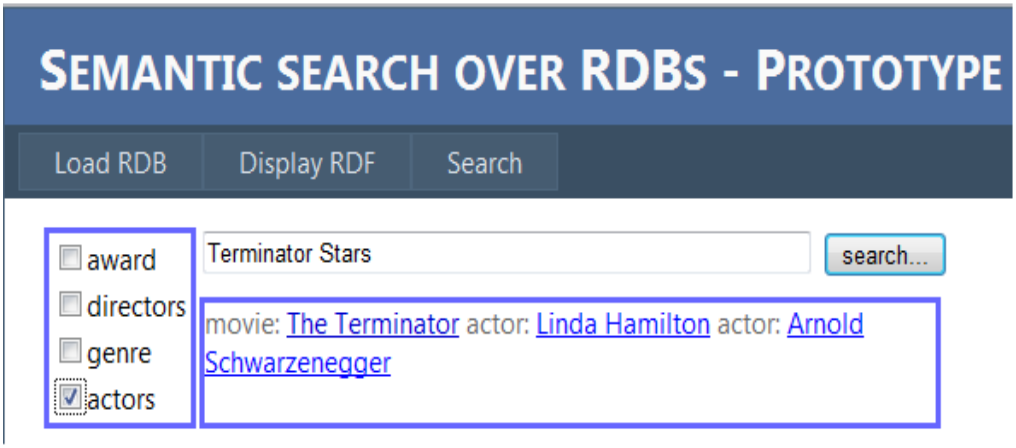

Figure 6.7.a.: search results for “Terminator Stars” with “actors” selected as a search facet

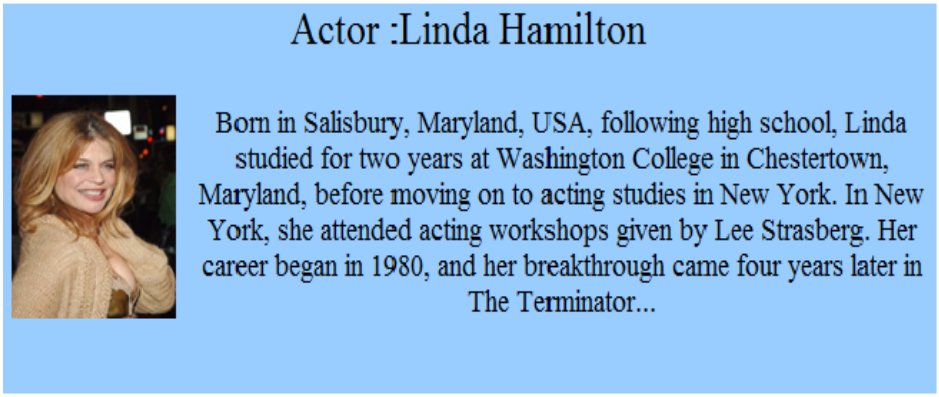

Figure 6.7.b.: clicking on “Linda Hamilton” web resource to get related info about her

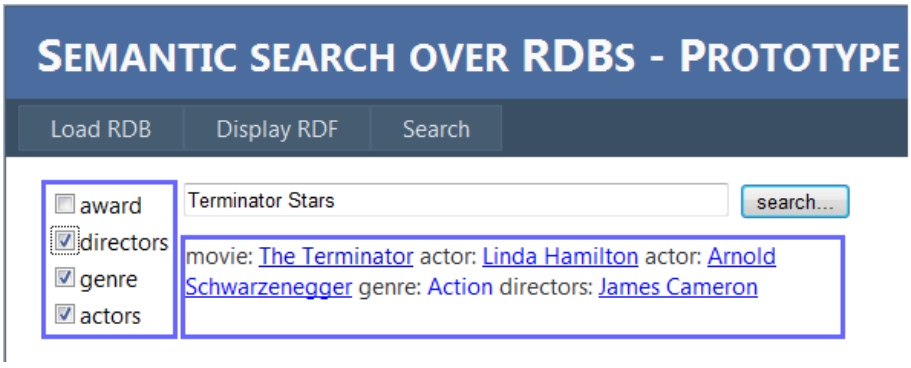

Figure 6.7.c.: selecting two more search facets: genre and directors

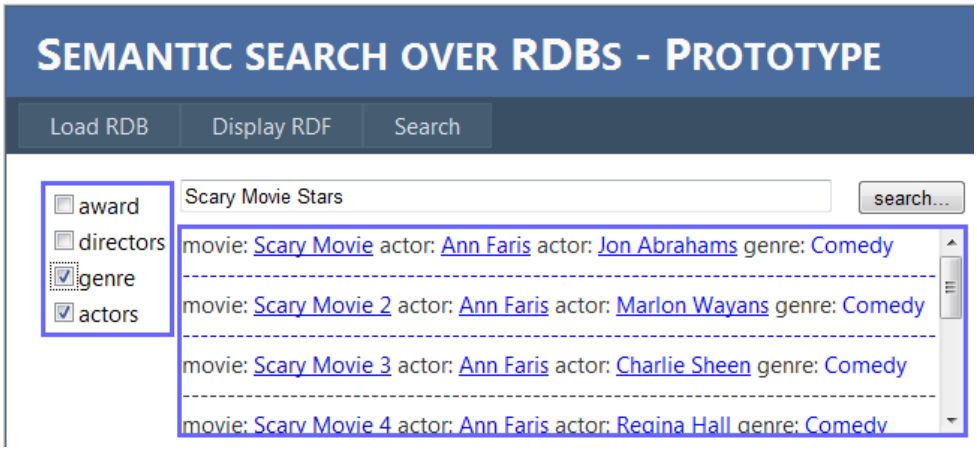

Figure 6.7.d.: search results for “Scary Movie Stars” with “actors” and “genre” selected as a search facets.

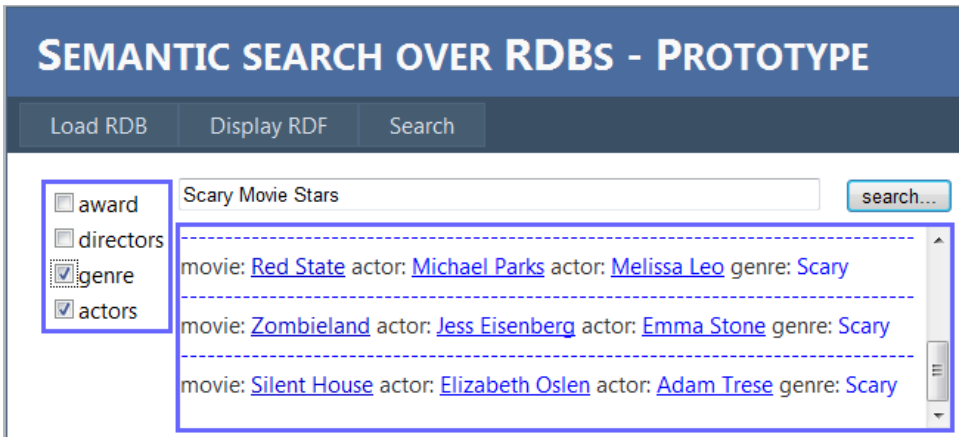

Figure 6.7.e.: search results for “Scary Movie Stars” with “actors” and “genre” selected as a search facets.

6.8 Experimental Results and System Evaluation

In general, evaluation of keyword search systems is a challenging task. In the context of relational databases, this is due to the fact that unlike SQL queries which are formal queries where the tables, tables’ attributes and their conditions are precisely specified, keyword queries are indefinite terms expressing user’s information needs. The accuracy and the correctness of a keyword query results are subjective to the users and it is almost impossible to find a true baseline for the evaluation of the keyword search and to formally prove the correctness of its results. To cope with the challenge of lacking a true baseline to which to compare our system and

to eliminate any bias towards our system, our experimental evaluation was performed by formulating the keyword queries by 3 non-technical users. The users didn't have any knowledge of the approaches we have employed in our system and formulated their keyword queries to retrieve the intended information. We asked the users to formulate 20 keyword queries each, targeting a variety of the information in the movie database

Table 6.8 shows a list of some of the sample keyword queries (in the generic form) along with the users' intended search results. There are two types of keywords in the queries: type1 keywords: which are the string literals representing the actual name of the actors, actresses, studios, genre, etc. which are indicated by str_i in the table. type2 keywords: which are the actual concepts, relationships (e.g. actor, director, starred in, etc). The queries which contain only type1 keywords are indicated by having a star next to them (i.e.*) in the table and are referred to as type1 queries. The queries which contain both type1 and type2 keywords are referred to as type2 queries.

To assess the effectiveness of our approach with comparison with the previous works on KSRDBs, we ran users' queries against both DBSemSXplorer and [41] (a system which we implemented based on the previous works on KSRDBs with some improvements in the search results' ranking). Please note that, we didn't consider the text attributes with long string values (i.e. unstructured text) such as movie summaries since, we only modeled the text attributes with short string values in the RDF triple-store. Incorporating a text attribute containing unstructured text into the RDF is left for future research work and the next version of DBSemSXplorer.

In order to identify the relevant answers to each query, we used pooled relevance judgment method as follows; after running each query against both systems, we merged their *top10* results and presented them to the users whom manually judged and selected the relevant results for that query. The aggregate of the relevant results from both systems chosen by the user was taken as the relevant search results for that query.

Table 6.8: Sample keyword queries in the generic form.

Keyword Query	**Intended Search Results**
$movie_i$ actors	*list of the actors in $movie_i$*
$movie_i$ director	*director of $movie_i$*
$movie_i$ director actors	*list of the director and the actors in $movie_i$*
** $movie_i$*	*list of the info about $movie_i$ (actors,director,genre,etc)*
** $actor_i$ $genre_i$*	*list of the $genre_i$ movies where $actor_i$ starred in*
$movie_i$ stars	*list of actors in $movie_i$*
$movie_i$ year	*the year in which $movie_i$ was made*
$genre_i$ movies $director_i$	*$genre_i$ movies directed by $director_i$*
movies directed starred $name_i$	*movies directed and starred in by $name_i$*
** $actor_i$ $year_i$*	*movies made in $year_i$ featuring $actor_i$*
**$actor_i$ $director_i$*	*movies directed by $director_i$ featuring $actor_i$*
$genre_i$ movies $director_i$ $actor_i$ $year_i$	*$genre_i$ movies made in $year_i$ directed by $director_i$ featured $actor_i$*

To evaluate and compare the systems we used two metrics:

1) Average precision/recall (i.e. average precision at recall level of 0.1):

We calculated the average precision at each recall level of 0.1 across type1 and type2 queries separately for each system. Figure 6.8 shows the precision/recall graph for both systems.

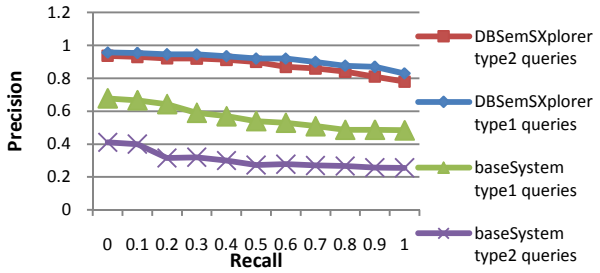

Figure 6.8: Precision/Recall Curve

We observed that DBSemSXplorer significantly outperformed the base system for both type1 and type2 queries. Both systems performed better on type1 queries in our experiments which is the indication of users finding more relevant results when using only type1 keywords as opposed to when using type2 keywords which represent relationships, concepts, etc. DBSemSXplorer outperforming the base system for type1 queries is an indication of users leveraging the

relationships between the keywords and actual concepts to find the relevant search results even thou when the original queries initiated from type1 keywords.

2) Mean Reciprocal Rank (MRR):

Reciprocal rank for a query is the inverse of the rank of the first correct (i.e. relevant) result for that query. We calculated the average reciprocal rank for type1 and type2 queries separately for each system. MRR indicates how soon in the search result set the first relevant result appears. Figure 6.9 shows the MRR for both systems along with the query lengths (i.e. # of keywords used) in order to draw some conclusions.

We observed that DBSemSXplorer outperformed the base system. Based on our experiments, for both systems, MRR is higher for type1 queries than type2. We also observed that on average our system performs better in the range of 2 keywords to 4 keywords.

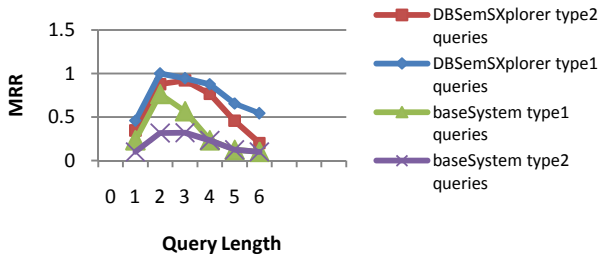

Figure 6.9: MRR-Query Length Plot

CHAPTER 7

ENHANCING THE SEMANTIC-BASED KEYWORD SEARCH IN RELATIONAL DATABASES BY LEVERAGING THE INFERENCE CAPABILITY OF SEMANTIC WEB

7.1 Introduction

In the last two chapters we introduced the semantic web technologies and presented our approach and system architecture in which we have incorporated and leveraged semantic web into keyword-based search in RDBs. We observed that our approach has significantly improved the effectiveness of the KSRDBs in terms of the relevancy of the search results. Not only that we gained a much better precision/recall after incorporating the semantic web, we were also able to answer more of the natural language processing (NLP) queries formulated by keyword queries (e.g. give me the actors of movie Terminator, which was expressed as Q={Terminator Actors}).

In this chapter we introduce Inference capability of semantic web which could potentially enrich the knowledge base (e.g. more facts can be inferred from the existing facts and the rules added to the RDF) and consequently improve the keyword search. We use our approach from chapter 6 to generate the knowledge base from a RDB, add rules to the knowledge base for enriching the knowledgebase and then perform a keyword query using the keyword mapping and SPARQL query construction technique we used in chapter 6.

7.2 Motivating Business Application Scenario

Large enterprise relational databases could have complex schemas with changing data and complex inner-relationships between the relations. There may be potential facts and knowledge in the database which are not exclusively encoded in the RDB and are thus less visible to the end user. Consequently, these facts are not found via traditional keyword search. For instance, consider the hypothetical schema depicted in Figure 7.2a which represents an auto industry's

vehicle production's relational database's schema including plants, assembly lines, parts, vehicle names/years/models, problems discovered in parts, etc. This schema is very similar to a small portion of an entity-relationship schema used in real world auto industries.

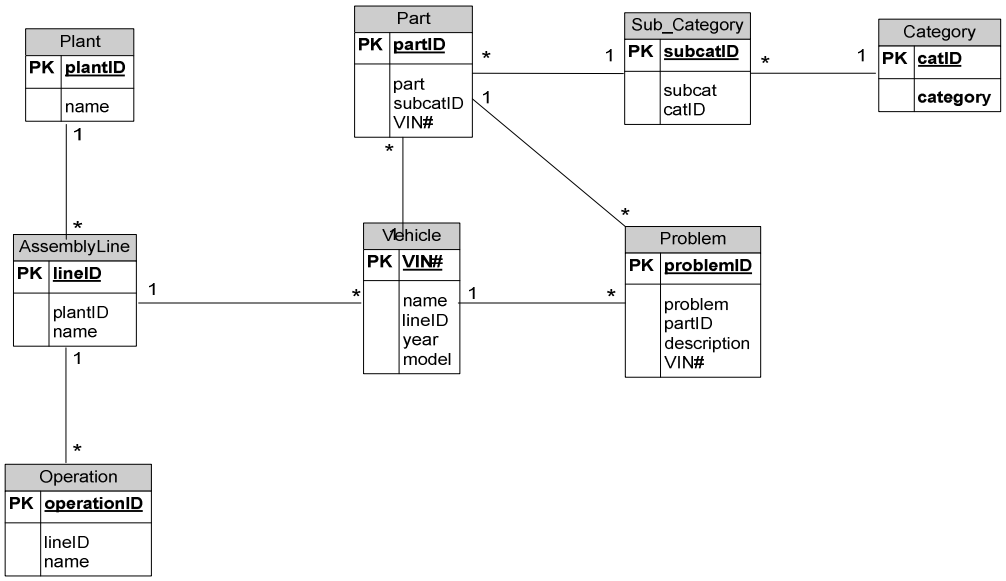

Figure 7.2a: An auto industry's vehicle production's relational database's schema.

For the data-confidentiality purpose we populated the database with hypothetical data depicted in Figure 7.2b.

Row#	Plant	Assembly Line	Operation	Vehicle	Part	Sub_Category	Category	Problem
1	PlantA	Line A	Installing Powertrain	CarA	Engine	Powertrain	Mechanical	none
2	PlantA	Line A	Installing Powertrain	CarA	Transmission	Powertrain	Mechanical	"doesn't shift smoothly"
3	PlantA	Line A	Installing Break	CarA	Rotor(2134)	Break System	Mechanical	"grinding noise"
4	PlantA	Line A	Installing Break	CarB	Rotor(1111)	Break System	Mechanical	"grinding noise"
5	PlantA	Line B	Installing lights	Car A	Head light(123)	Lights	Electrical	None
6	PlantA	Line B	Installing lights	Car B	Tail light(333)	Lights	Electrical	None

Figure 7.2b: An instance of the vehicle production's relational database.

Couple of notes about the data: 1) We can think of each distinct value of relation sub-category as a subclass of one distinct value of relation category.(e.g. break system is a subclass of mechanical class) 2) The name for part Rotor(2134) has changed to Rotor(1111) recently.(please note that the part number is not an identification for an individual part which is obviously can only be used once in one vehicle. The part number indicates the specifications of a group of parts (size, diameter, type, etc.)).

Now assume that an engineer wants to find all of all the vehicles where there is an issue with Rotor(2134) parts and further she wants to know the plants where the installation took place. The typical keyword query would look something like Q={vehicles Rotor(2134) issues plants}. By now, we know that the traditional KSRDB approaches covered in the earlier chapters cannot answer this question primarily because of the two reasons we discussed in the last chapter: 1) the keyword “issue” may not have been used in the database which is also the case in this example 2) even if we use the term “problem” instead of “issue”, traditional approaches do not treat keywords “problem”, “vehicles” and “plants” as wild cards (i.e. variables) to retrieve their values which are fulfilling the search criteria.

In the last chapter after identifying these issues, we came up with an approach of extracting the underlying ontology from a RDB and then generating an RDF representing the KB (knowledgebase). We then found the best mapping from the query terms to the KB terms (i.e. web resources) using an optimization algorithm. We finally, construct the corresponding SPARQL from the matched resources for the keyword search and executed it against the KB. This approach took into account the possible synonyms for the query terms when mapping to the KB terms, also using the semantic web technologies we were able to answer queries containing both string literals and variables.

Returning back to our example above, our approach from last chapter maps the term “issues” to the web resource “problem” and also treats the terms “problem”, “vehicles” and “plants” as variables. Therefore, using our approach from last chapter we are able to find row#3 as the only

answer to the query. Whereas row#4 should also be returned as a relevant answer to the query since part Rotor(2134) is the same as part Rotor(1111). If the end user is not aware of the fact that, the same part has been called Rotor(2134) up until some point in time and then the name has changed to Rotor(1111), she will not find any of the records where the new name is used in.

What if the user wants to find all of the assembly lines where the installation for the faulty Rotors(2134) took place. Does she need to perform another keyword search with the keyword “assembly lines”? Lastly, let’s say she wants to find all the mechanical parts with issues, does she need to perform separate keyword queries for “break system”, “engine and transmission” , etc.?

The above queries cannot be efficiently answered by the approach presented in the last chapter. For instance, a) User must know that the name for Rotor(2134) has changed to Rotor(1111) and even if she knows she has to perform two queries. b) She has to perform separate queries for finding the assembly lines where the faulty rotors were installed. c) She has to perform multiple queries to find all the “mechanical parts” with issues (one query per each subclass of the class “mechanical part” such as “break system”, “engine”, etc.) and then aggregate the results.

To cope with the issues mentioned in this example, we decouple the system/application from the knowledgebase by extracting the semantics of these relationships (e.g. relationships between Rotor(2134) and Rotor(1111), plants and assembly lines, break system and mechanical parts, etc.) out of the user’s queries (i.e. application level) and place them in the knowledge base (i.e. RDF generated by our algorithm from last chapter) and then we use the inference capability of semantic web to infer and discover many facts which are not exclusively encoded in the KB.

In the next section we give an overview of the concepts/definitions defined in the semantic web community. In particular we review the following properties: *class*, *subclassOf*, *type* and *sameAs* defined in *RDF*, *RDFS* and *OWL* which are necessary to know before presenting our approach.

7.3 Semantic Web Vocabulary and Inference

7.3.1 RDFS and OWL Properties

In chapter 6 we used Resource Description Framework (RDF) to define the web resources in our knowledgebase and to represent user's defined relationships between them. RDF can also be used to describe RDFS and OWL vocabularies which are the semantic web predefined properties used to define the relationships between the web resources and between the web resources and the string literals. Below are the definitions of the four semantic web vocabulary terms that we will use in next section to explain our approach.

1) rdfs:class: In semantic web in order to group similar objects together(web resources where share common features) they are placed into the same category or class which is represented by rdfs:Class. For instance consider the schema from example above, triple below states that category is a class.

ex:category rdf:type ex:class

2) rdfs:SubclassOf: In semantic web in order to state that all the instances of one class are the instances of another class, property rdfs:SubclassOf is used. For instace, triple below states that class subcategory is a subclass of class category.

ex:subcategory rdfs:SubclassOf ex:category

3) owl:sameAs: In semantic web in order to assert that two web resources with differents URIs are the same, property owl:sameAs is used. For instace, triple below states that web resource ex:Rotor(2134) is the same as web resource ed:Rotor(1111).

ex:Rotor(2134) owl:sameAs ex:Rotor(1111)

7.3.2 Semantic Web Inference

Semantic web provides inference mechanisms done by the reasoning engines (i.e. reasoners) which apply the semantics of the vocabulary and the properties added to the to the actual triple

statements of the knowledgebase (section 7.3.1) to infer additional data which are not exclusively encoded as triples and hence less visible to the user. This is also called enrichment of the knowledgebase using inference.

Many semantic web frameworks such as Jena which use rule-based reasoners. Rule-based reasoners perform inference by adding the new assertions (i.e. rules) to the existing assertions contained in the knowledgebase to imply/infer new facts (i.e. entailments). For the rest of this chapter we use rules and assertions interchangeably. (Please note that in semantic web literatures the term *rule* can also refer to the conditional statements which are in the form of if-then clauses).

In next section we present the modified version of the system architecture presented in chapter 6. The modified architecture integrates the reasoning engine of Jena framework. We have written a semi-automated rule extractor component which extracts new assertions from the RDB and incorporate them into the Jena Model which is instantiated from the KB generated from the RDB (using our algorithm from chapter 6). We will then report on the recall of the keyword search results obtained after performing the inference on the enriched KB after new assertions have been added.

7.4 System Architecture and Rule Extraction Algorithm

The new version of the system architecture is depicted in Figure 7.4 below. For the space limitation only the modified portion of the architecture is depicted. The right hand side of the Figure, the arrow labeled with “constructed sparql queries”, comes from the best-matched-of-web-resources module shown in Figure 6.3. The two main modules added are the Jena model and the rule extractor.

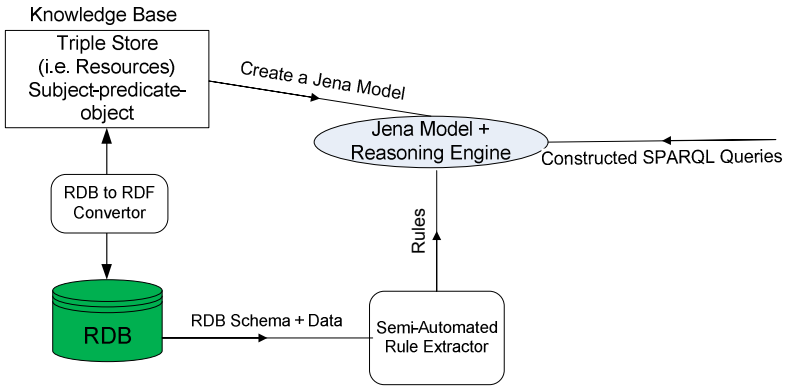

Figure 7.4: Modified System Architecture

7.4.1 Jena Model

Jena is a Java framework developed by the Apache Software Foundation. Jena framework is used to facilitate building semantic web applications by providing java libraries, components and tools such as java APIs to read/write RDF files, create RDFS and OWL ontologies. Most importantly Jena provides SPARQL query engine and rule-based reasoner which our system heavily relies on. The Jena framework and the Java technologies used in it are beyond the scope of this chapter and we invite the reader to read and learn more about it at [48]. Below is a small snippet of the Java code program we wrote to create a Jena model using the KB rdf file and then bind it to the Jena’s rule-based reasoner. (Please note there are few helper functions that needed to be implemented which are not shown in this snippet).

```
// Importing Jena libraries
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.reasoner.Reasoner;

//instantiating a Jena model
Model jenaModel = new Model();

// creating an ontology and reading our KB rdf file
// into the Jena model
jenaModel = ModelFactory.createOntologyModel();
……..
……..
……..
jenaModel.read("KB.rdf");

// adding rules to the model
jenaModel.addRule("ex:subcategory rdfs:SubclassOf ex:category);
jenaModel.addRule(ex:Rotor(2134) owl:sameAs ex:Rotor(1111));

//instantiating jena reasoned and binding it to the model
com.hp.hpl.jena.reasoner.Reasoner reasoner =
ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(jenaModel);

// run the SPARQL query constructed in chapter 7
jenaModel.runSPARQL();
```

Figure 7.4.1: Jave code snippet showing how to create Jena model and biding it to rule-based reasoner

7.4.2 Rule Extractor

As mentioned earlier in this chapter, the primary goal of this chapter is to see if we can outperform the semantic-based keyword search over RDB presented in chapter 6. We introduced the inference capability of the semantic web which if incorporated into the KB can help to discover potential relevant answers to the user's keyword search. Adding rules has to be done by a domain expert who is very well familiar with the schema of the RDB and also knows the content very well. In reality and practically no automated rule extractor can possibly replace a human agent (i.e. domain expert) in identifying rules which need to be added to the KB as new assertions to infer further entailments in the KB. On the other hand identifying rules can be a

tedious and time consuming task especially for very large relational databases with complex schemas. To facilitate this task, we have written our rule extraction algorithm which can effectively extract intuitive rules.

As mentioned earlier in this chapter we are referring to the assertions added to the KB for the purpose of inference, as rules. We are only trying to capture two types of assertions from the RDB schema and its associated tuples. These two types of assertions are 1) Assertions containing the *sameAs* property to infer that two web resources are the same. 2) Assertions containing the *SubclassOf* property to infer that the members of a subclass are also the valid members of its superclass and the properties of the superclass are inherited by its subclasses. Our algorithm for rule extraction is shown in Figure 7.4.2 below:

Input: 1)RDB schema (i.e. RDB relations,
relationships between the relations) 2) data stored in database.
3) RDF knowledgebase (KB)
Output: a set of rules in tiple (s-p-o) format
1: *S* // Relational database Schema
2: $table_i$ // A table in *Schema*
3: $attr_k$ // A column/attribute in a *table*
4: $table_{i\_PK}$ // The primary key for a $table_i$
5: $table_{i\_FK}$ // The foreign key for a $table_i$
6: $table_{i\_URI}$ // $table_i$ namespace
7: *KB* // Knowledgebase
8: *Rules* // set of rules extracted to be added to *KB*
9: // finding the *SubclassOf* relationship between relations
10: *For each* $table_i \in S$
11: *For each* $table_j \in S$
12: if ($table_i$!= $table_j$ && $table_{i\_PK}$ == $table_{i\_FK}$ &&
13: ($table_{i\_PK}$)^1 → ($table_{i\_FK}$)*)
14: If(*Taxonomy*($table_i$, $table_j$)→ ($table_j$ SubclassOf $table_j$))
15: *Rules*.add($table_j$ SubclassOf $table_j$)
16: *end for*
17: *end for*
18: // finding the *sameAs* relationship between the web resources
19: *For each* $table_i \in S$
20: *For each* $attr_i \in table_i$
21: *For each* $attr_j \in table_i$
22: if ($attr_i$!= $attr_j$ && $attr_i$ →functional_dependency $attr_j$)
23: *For each* ($value_i \in attr_i$ && $value_j \in attr_j$)
24: if($value_j$ == $value_j'$ && $value_i$!= $value_i'$)
25: *Rules*.add($value_i$ sameAs $value_i'$)
26: *end for*
27: *end for*
28: *end for*

Figure 7.4.2: Semi-Automated Rule Extractor

This algorithm shows the semi-automated rule extraction from the RDB. The input is the schema of the RDB, the data stored in the RDB along with the KB generated from it. In lines 10 to 14 the algorithm checks all the relations in the RDB to find any relation whose primary key is a foreign key of another relation and there is a 1→* (one-to-many) relationship from the parent relation to the child relation. If these criteria are met then the names for both of the relations (parent and child) are fed as an input to a dictionary (e.g. WordNet's hyponym relationship) and/or a taxonomy previously built for the RDB, to check whether there exist a class/subclass relationship between the two relations. At this point (line 15 in the algorithm) a domain expert has to check whether there indeed exist a class-subclass relationship between the two relations and if so it will be added to the *Rules* which is eventually added to the KB. In lines 19 to 24 the algorithm checks the relations to find the pairs of attributes (could be extended to find the pairs of the sets of attributes) in which there exist a *functional dependency*(*FD*) relationship. If the *FD* relationship exist between a pair of attributes let's say *X* and *Y* (e.g. *X* is functionally dependent on *Y*) and if there is a change in the value of *X* (x' found among all other equal *x*'s) then at this point the domain expert inspect *x*' and if it refers to the same resource as referred by *x* then the rule (x' sameAs *x*) is added to the *Rules*.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Summary

In this dissertation work we gave an overview of keyword search over relational databases. We explained how the answers to the users' keyword queries are generated. We reviewed the approaches by other research works in finding the most relevant results. We identified some of the issues with finding the relevant results to the users' keyword queries and with ranking them. We incorporated two important factors namely, keyword N-grams and keyword proximity to the ranking function and observed a significantly higher recall/precision.

We gave an overview of the two mainly approaches used in recommendation systems namely content-based filtering and collaborative filtering. We identified the shortcomings faced by both approaches and gave our approach which leveraged a data mining method which doesn't rely on the users' historical data in order to make recommendations.

We then gave an overview of semantic web technologies such as RDF, RDFS and SPARQL. We proposed an approach and implemented a system based on that, DBSemSXplorer, to answer keyword search in RDBs by transforming the RDBs into an RDF knowledgebase, mapping the query terms to the most semantically and syntactically similar knowledge base resources and finally constructing equivalent SPARQL from the mapped terms. We observed that our approach outperformed the previous approaches in finding the most relevant answers to the keyword queries.

8.2 Contributions

I have made the following contributions during my PhD years which were presented in this dissertation:

1) I have studied the IR style techniques used in recent research works in the area of KSRDBs and have identified two important factors that can further improve the search effectiveness when incorporated into the IR relevance ranking strategies. These two factors are i) The query keywords' proximity, which is the overall distance of the keywords from one another in the value of the target text attribute and ii) The N-grams and, in particular, the quadgrams of the query keywords in both the query itself and in the text attributes' values. My experiments show that incorporating these two factors into the existing state-of-the-art ranking functions can improve the ranking of KSRDBs by achieving significantly higher precision at standard recall points.

2) I have adapted a novel approach in making keyword search recommendations based on the text attributes in which the search terms were found without relying on the user's past search criteria. A proof of concept (POC) prototype system called TupleRecommender has been implemented based on this approach.

3) In (1), I identified two important factors which indeed improved the search effectiveness when incorporated into the IR relevance ranking functions. However, one major issue with keyword search in general is its ambiguity which can ultimately impact the effectiveness of the search in terms of the quality of the search results. This ambiguity is primarily due to the ambiguity of the contextual meaning of each term in the query (e.g. each query term can be mapped to different schema terms with the same name or their synonyms). In addition to the query ambiguity itself, the relationships between the keywords in the search results are crucial for the proper interpretation of the search results by the user and should be clearly presented in the search results. To address these issues we have designed and implemented a proof of concept

(POC) prototype system called database semantic search explorer (DBSemSXplorer) which can answer the traditional keyword search over relational databases in a more effective way with a better presentation of the search results. We address the keyword search ambiguity issue by leveraging some of the existing techniques for keyword mapping from the query terms to the schema terms/instances. These techniques capture both the syntactic similarity between the query keywords and the schema terms as well as the semantic similarity (e.g. definition of the keywords) of the two and give better mappings and ultimately more accurate results. Finally, to address the last issue of lacking clear relationships among the terms appearing in the search results, our system has leveraged semantic web technologies in order to enrich the knowledgebase and to discover the relationships between the keywords. Our experiments show that our system is more effective than the traditional keyword search approaches by enabling the users to find the search results which are more relevant to their keyword queries.

8.2 Future Research Work

Our primary focus in this research work was to effectively find the most relevant results to the keyword search performed over RDBs. Our last approach was leveraging semantic web technologies in RDBs with short string text attributes. However, the unstructured text data stored in a variety of text attributes across the RDB could also be modeled as an RDF along with the short string attributes. Further research work has to be done to take advantage of the semantic web technologies and to apply them to the unstructured text data within RDB (e.g. memos, descriptions, etc.) and measure the enhancement of the search result relevancy.

Another important research area is to look into how leveraging semantic web technology can improve the efficiency of the keyword search over RDBs.

APPENDIX

Publications

[1] Fakhraee, S., Fotouhi, F.: Effective Keyword Search over Relational Databases Considering keywords proximity and keywords N-grams. In: 8th International Workshop on Text-based Information Retrieval – TIR'11

[2] Fakhraee, S., Fotouhi, F.: TupleRecommender: A Recommender System for Relational Databases. In: International Workshop on Recommender Systems Meet Databases – RsmeetDB'11

[3] Miller, C., McElmurry, S., Fakhraee, S., Fotouhi, F., Pitts, D.K., Bristol, C., Dereski, M.: Digital Library to Facilitate Urban Environmental Management. Watershed Management Conference: Innovations in Watershed Management Under Land Use and Climate Change, Madison Wisconsin, August 2010

[4] Fakhraee, S., Fotouhi, F.: DBSemSXplorer: Semantic-based Keyword Search System over Relational Databases for Knowledge Discovery. Submitted to KEYS 2012

BIBLIOGRAPHY

[1] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Effcient ir-style keyword search over relational databases. In *VLDB*, pages 850-861, 2003.

[2] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[3] Fang Liu, Clement Yu Weiyi Meng “Effective Keyword search in RDBMS” SIGMOD 2006 Chicago, Illinois

[4] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases," icde, pp.0005, 18th International Conference on Data Engineering (ICDE'02), 2002

[5] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *SIGIR '07*, pages 295–302, 2007

[6] Luo, Y., Lin, X., Wang, W., and Zhou, X. Spark: top-k keyword query in relational databases. In Proc. SIGMOD (New York, NY, USA, 2007), ACM, pp. 115-126.

[7] R. Goldman, N. Shivakumar, S. Venkatasubramanian and H. Garcia-Molina: Proximity Search in Databases. VLDB, 1998

[8] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan.Keyword searching and browsing in databases using BANKS. In Proc. of ICDE’02, 2002.

[9] IMDB datasets, http://www.imdb.com/interfaces

[10] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan.Keyword searching and browsing in databases using BANKS. In Proc. of ICDE’02, 2002.

[11] L. Qin, J. Yu and L. Chang. Keyword Search in Databases: The power of RDBMs. In SIGMOD 2009.

[12] X. Yin, J. Han, and P. S. Yu. Cross-Relational Clustering with User’s Guidance. In KDD’05.

[13] Adomavicius, G., Sankaranarayanan, R., Sen, S., and Tuzhilin, A., 2005. Incorporating contextual information in recommender systems using a multidimensional approach. ACM TOIS, 23, 1, 103-145.

[14] N. Mirzadeh, F. Ricci, and M. Bansal. Feature selection methods for conversational recommender systems. In Proceedings of the IEEE International Conference on e Technology, e-Commerce and e-Services, Hong Kong, 29 March - 1 April 2005. IEEE Press.

[15] Breese, Heckerman, and Kadie: Empirical analysis of predictive algorithms for collaborative filtering. Proc of 14th Conf of Uncertainty in Artificial Intelligence, pp 43-52, 1998

[16] Burke, R. Hybrid recommender systems: Survey and experiments. User Modeling and User-Adapted Interaction, 2002.

[17] MovieLens datasets, http://www.grouplens.org

[18] IMDB datasets, http://www.imdb.com/interfaces

[19] http://amazon.com

[20] http://en.wikipedia.org/wiki/Semantic_search

[21] M. Saleh. Semantic-Based Query in Relational Database Using Ontology. Canadian Journal on Data, Information and Knowledge Engineering Vol. 2, No. 1, January 2011

[22] H. Wang, K. Zhang, Q. Liu, T. Tran, and Y. Yu, "Q2semantic: A lightweight keyword interface to semantic search," in ESWC, 2008, pp. 584-598.

[23] Bergamaschi S, Domnori E, Guerra F, OrsiniM, Lado RT, Velegrakis Y (2010) Keymantic: Semantic keyword based searching in data integration systems. Proceedings of VLDB, vol 3(2), pp 1637–1640

[24] Atsutoshi Imai, and Shuichi Yukita, “RDF Model and Relational Metadata”, 17th International Conference on Advanced Information Networking and Applications (AINA'03)

[25] Shufeng Zhou. “Exposing Relational Database as RDF”, 2010 2nd International Conference on Industerial and Information Systems

[26] I. Astrova, "Reverse Engineering of Relational Databases to Ontologies" The Semantic Web: Research and Applications:pp. 327-341.

[27] C. Bizer and A. Seaborne, "D2RQ : Treating Non-RDF Databases as Virtual RDF Graphs". Proceedings of the 3rd International Semantic Web Conference (ISWC2004)

[28] O. Erling, "Declaring RDF Views of SQL Data". Position paper for the W3C RDFRDB Workshop on RDF Access to Relational Databases

[29] A. Ezzat, M. Hausenblas and H. Halpin (2009). "W3C RDB2RDF Working Group." World Wide Web Consortium (W3C). from http://www.w3.org/2001/sw/rdb2rdf/.

[30] Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: ISWC, pp. 54–68 (2002)

[31] Lu, J., Ma, L., Zhang, L., Brunner, J.S.,Wang, C., Pan, Y., Yu, Y.: Sor: A practical system for ontology storage, reasoning and search. In: VLDB, pp. 1402–1405 (2007)

[32] Luo, Y., Wang, W., Lin, X.: SPARK: A keyword search engine on relational databases. In: ICDE, pp. 1552–1555 (2008)

[33] Li, G., Feng, J.-H., Lin, F., Zhou, L.-z.: Progressive ranking for efficient keyword search over relational databases. In: Gray, A., Jeffery, K., Shao, J. (eds.) BNCOD 2008. LNCS, vol. 5071, pp. 193–197. Springer, Heidelberg (2008)

[34] Wang, S., Zhang, K.: Searching databases with keywords. J. Comput. Sci. Technol. 20(1), 55–62 (2005)

[35] Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)

[36] Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: ACM SIGMOD, pp. 903–914 (2008)

[37] Sarda, N.L., Jain, A.: Mragyati: A system for keyword-based searching in databases. CoRR cs.DB/0110052 (2001)

[38] Balmin, A., Hristidis, V., Papakonstantinou, Y.: ObjectRank: Authority-based keyword search in databases. In: VLDB, pp. 564–575 (2004)

[39] Hristidis, V., Hwang, H., Papakonstantinou, Y.: Authority-based keyword search in databases. ACM Trans. Database Syst. 33(1), 1–40 (2008)

[40] Sayyadian, M., LeKhac, H., Doan, A., Gravano, L.: Efficient keyword search across heterogeneous relational databases. In: ICDE, pp. 346–355 (2007)

[41] Fakhraee, S., Fotouhi, F.: Effective Keyword Search over Relational Databases Considering keywords proximity and keywords N-grams. In: 8th International Workshop on Text-based Information Retrieval – TIR'11

[42] Fakhraee, S., Fotouhi, F.: TupleRecommender: A Recommender System for Relational Databases. In: International Workshop on Recommender Systems Meet Databases – RsmeetDB'11

[43] Haidarian, H.: Semantic Search in Linked Data: Opportunities and Challenges

[44] Zhou, Q., Wang, C., Xiong, M., Wang, H., Yu, Y.: Spark: Adapting keyword query to semantic search. In: A berer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ISWC 2007. LNCS, vol. 4825, pp. 694-707. Springer, Heidelberg (2007)

[45] Sonia Bergamaschi , Elton Domnori , Francesco Guerra , Mirko Orsini , Raquel Trillo Lado , Yannis Velegrakis, Keymantic: semantic keyword-based searching in data integration systems, Proceedings of the VLDB Endowment, v.3 n.1-2, September 2010

[46] S. Zhou, "Exposing Relational Database as RDF," 2nd International

Conference on Industrial and Information Systems, 2010.

[47] http://en.wikipedia.org/wiki/Hungarian_algorithm

[48] http://incubator.apache.org/jena/

ABSTRACT

EFFECTIVE SEMANTIC-BASED KEYWORD SEARCH OVER RELATIONAL DATABASES FOR KNOWLEDGE DISCOVERY

by

SINA FAKHRAEE

May 2012

Advisor: Dr. Farshad Fotouhi

Major: Computer Science

Degree: Doctor of Philosophy

Keyword-based search has been popularized by Internet web search engines such as Google which is the most commonly used search engine to locate the information on the web. On the other hand while traditional database management systems offer powerful query languages such as SQL, they do not provide keyword-based search similar to the one provided by web search engines. The current amount of text data in relational databases is massive and is growing fast. This increases the importance and need for non-technical users to be able to search for such information using simple keyword search just as how they would search for text documents on the web. Keyword search over relational databases (KSRDBs) enables ordinary users to query relational databases by simply submitting keywords without having to know any SQL or having any knowledge of the underlying structure of the data. In this research work our primary focus is to enhance the effectiveness of the keyword search over relational databases using semantic web technologies. We have also addressed some the issues with the effectiveness of the current keyword search over relational databases. In particular we are addressing the followings:

We have improved (gained significantly higher precision/recall curve) the existing state-of-the-art ranking functions by incorporating the query keywords' proximity and query keywords' quadgrams of the text attributes with long string into the scoring function.

We have adapted a novel approach in making keyword search recommendations based on the text attributes in which the search terms were found without relying on the user's past search criteria. A proof of concept (POC) prototype system called TupleRecommender has been implemented based on this approach.

We have designed and implemented a proof of concept (POC) prototype system called database semantic search explorer (DBSemSXplorer) which can answer the traditional keyword search over relational databases in a more effective way with a better presentation of search results. This system is based on semantic web technologies and is equipped with faceted search, inference capability to ease the task of knowledge discovery for the end user.

AUTOBIOGRAPHICAL STATEMENT

SINA FAKHRAEE

I was born in Saint Louis Missouri when my father was doing his residency there. I grew up in Tehran the capital city of Iran. I moved to United States in 1999 after graduating from Alborz high school (i.e. ranked as top five high school at the time I successfully passed the its entrance exam). Upon arrival to Michigan I attended Washtenaw Community College in Ann Arbor for two years before transferring to Michigan State University where I obtained both my BS and MS degrees in Computer Science and Engineering. I attended Wayne State University to pursue a PhD degree in Computer Science and successfully finished it in March of 2012. I have experience teaching many under graduate CS, Physics and Math courses and a couple of graduate level CS courses. I have worked as a software developer in a couple of companies during the time I was pursuing my PhD. Currently, I am working at Ford Motor Company as a researcher and a software engineer in the Office of Technical Fellow (OTF) in Knowledge on Demand (KOD). In my spare time, I love to workout, run and play soccer.