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

Sina Fakhraee
Dept. of Computer Science
Wayne State University
Detroit, MI 48202, USA
fakhraee@wayne.edu

Farshad Fotouhi
Dept. of Computer Science
Wayne State University
Detroit, MI 48202, USA
fotouhi@wayne.edu

ABSTRACT
Keyword search over relational databases has been broadly studied in recent years. Research works have been done to address both the efficiency and the effectiveness of the keyword search over relational databases. 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.
To address these issues we have designed and implemented a prototype system 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 adapting some of the existing approaches for keyword mapping from the query terms to the schema terms/instances. The approaches we have adapted 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, 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.

Keywords
Keyword Search, Relational Database, Semantic Web, RDF, SPARQL.

1. INTRODUCTION
Keyword search over relational databases has been broadly studied in the recent years. There have been many research works conducted to enhance the search in terms of both efficiency and effectiveness. One major issue with keyword search in general is its ambiguity and this could potentially impact the search effectiveness. 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.
Unfortunately, many of the existing approaches neither consider the semantic similarity nor the syntactic similarity (many only consider the exact match) 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 improve the quality of the search results and ultimately to enhance the search effectiveness, we have designed and implemented a prototype system called DBSemSXplorer (Database Semantic Search Explorer). DBSemXplorer addresses the above issues by leveraging semantic web technologies, as well as adapting some of the existing approaches used for keyword mapping from the query terms to the schema terms. These mapping approaches consider both the syntactic similarity and the semantic similarity between the query terms and schema terms.
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 relationships among 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 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 the clearly defined relationships among the terms appearing in the search results.

The remainder of this paper is organized as follows: Section 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 3 gives and overview of the DBSemSXplorer system architecture. Section 4 presents our approach in generating the knowledgebase from a relational database. Section 5 briefly discusses the different techniques used in keyword mapping. Section 6 explains our

KEYS'12, May 20, 2012, Scottsdale, Arizona, USA.

approach on constructing SPARQL query from the matched resources from section 5. Section 7 presents our system prototype and interface. Section 8 presents our experimental results and system evaluation. Section 9 presents the related work and section 10 concludes our work and gives direction for future work.

2. MOTIVATION and Brief Background on Keyword Query Results Generation

Traditionally keyword search over relational databases is performed by creating a master index which indexes all the terms in the relational database. 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. (Due to the lack of the space we refer the reader to [1,2] for detailed definitions of these terminologies and in depth explanation of the keyword search approach over relational databases).

To demonstrate the approach 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. Portion of IMDB Schema) depicted in Figure 1 and Table 1 respectively.

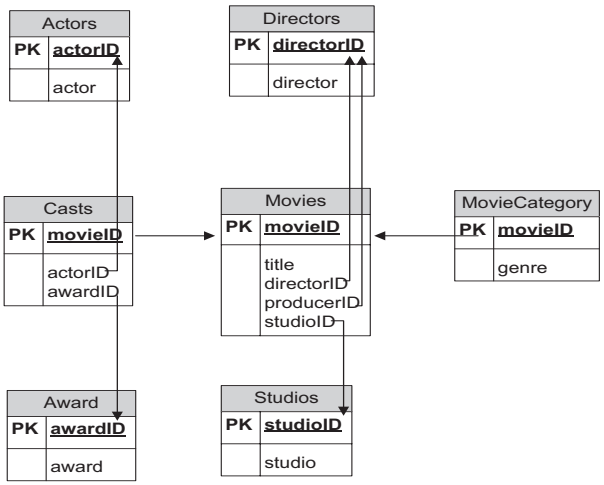

Figure 1. Portion of IMDB Schema

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" 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 an 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.

Table 1. An Instance of IMDB

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

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 Figure2.

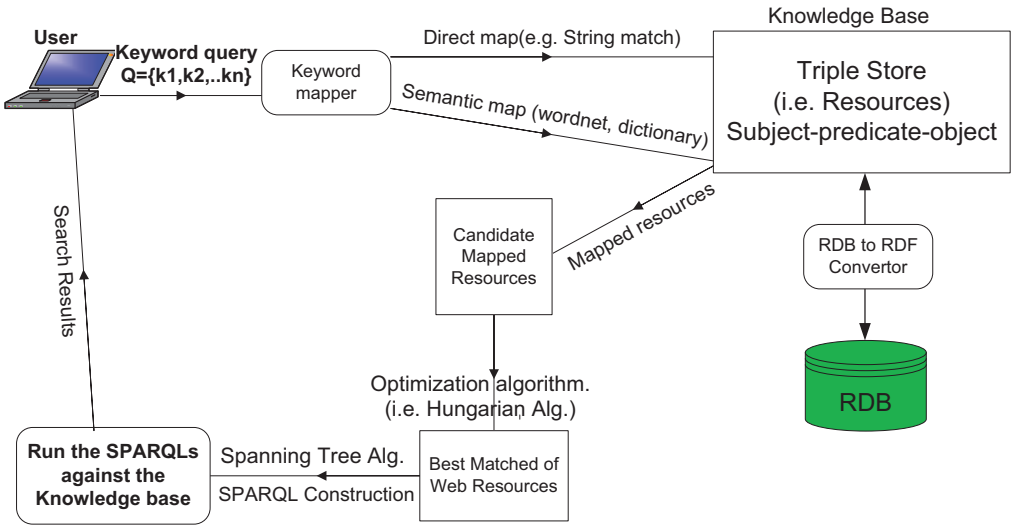

Figure 2. System Architecture

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.

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.

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.

4. Generating the Knowledgebase from a Relational Database

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 another 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 directed 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 3 below.

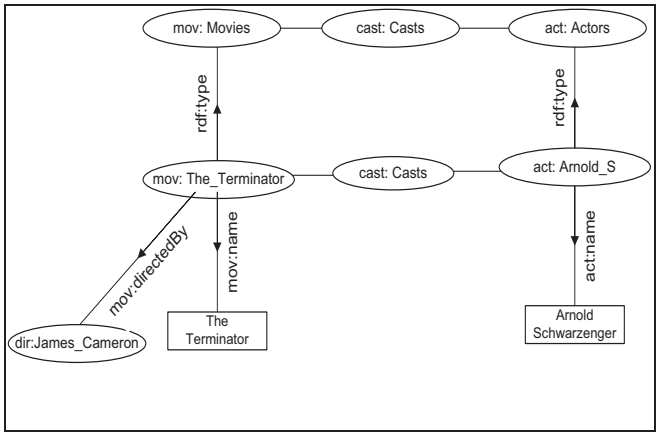

Figure 3. RDF graph for IMDB

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. RDBs) makes it less tedious to extract an ontology from its schema as opposed to non- or semi-structured data (e.g. text documents). The approach we have adapted to accomplish this task is similar to approaches employed by commercial RDB2RDF (i.e. relational database to rdf) systems and other related research literatures [3].

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. The second triple states that the movie 5's director is "dir:4" which is a web resource. The third triple states that "dir:4" has a property "dir:name" with the 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".
```

The algorithm for transforming an RDB to an RDF is shown in Figure 4 below:

Algorithm1: 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 4. 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.

5. From the Query terms to the Knowledgebase terms

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. Many of the techniques we have used have been inspired by previous works of Spark and Kemantic [4, 5].

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 numbers of edit operations such as (insertion, deletion, replacement) required to transform one string into another one. To compute the syntactic similarity of two strings we use the following formula:

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

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 [1].
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.

Example 2: 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.

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 0.4 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.

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 5 below:

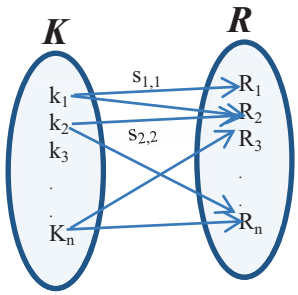

Figure 5. Bipartile graph modeling Keywords to Resources

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 [6] 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. SPARQL Query Construction

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.

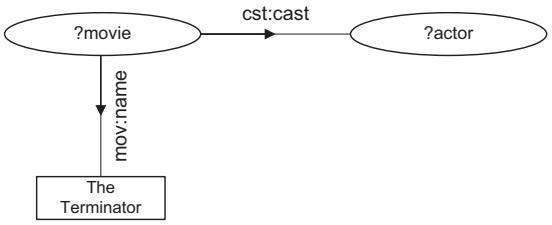

Figure 6. The Triple Pattern

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 7.

6.2 Construction of the SPARQL Query from the Matched Resources

After finding the best match of the set of the resources in section 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 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.

6.3 Ranking Scheme

When mapping from the query terms to the schema terms as explained in section 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 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} in Example 2. The SPARQL constructed from the web resources "scary movie" as a string literal and "stars" 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}).

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 to 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 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 7.

8. Experimental Results and System Evaluation

Evaluation of keyword search systems is a challenging task in general. 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 2 shows a list of some of the sample keyword queries (in generic form) along with the users' intended search results. There are two types of keywords in the queries; type1 keyword: the names of the actors, actresses, studios, etc which are indicated by $name_i$ in the table. type2 keyword: 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 (e.g.*) 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 [7] (a system which we implemented based on the previous works on KSRDBs with some improvements in the search results' ranking).

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 total relevant search results for that query.

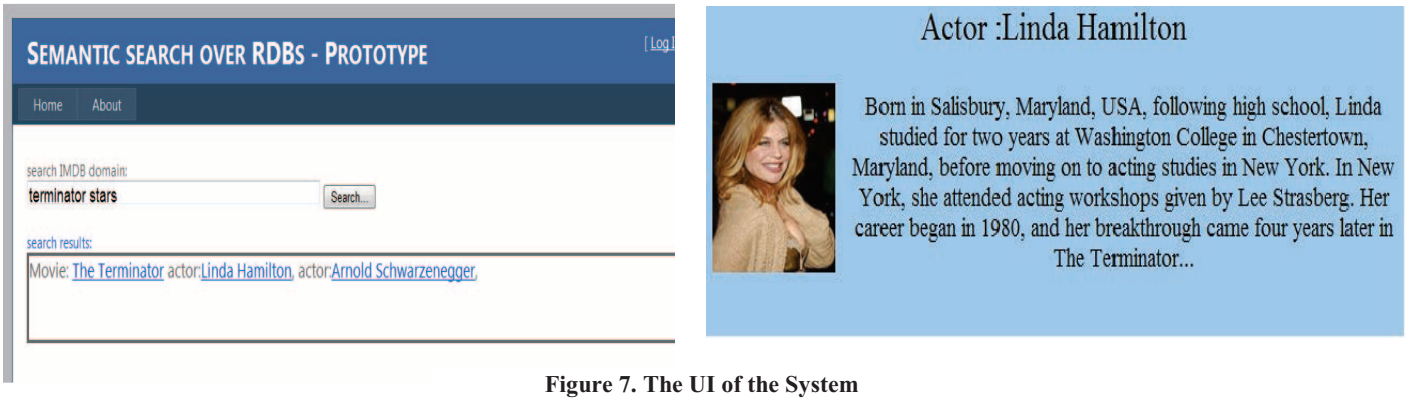

Figure 7. The UI of the System

Table 2. Sample Keyword Queries

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 type1 queries separately for each system. Figure 8 shows the precision/recall graph for both systems.

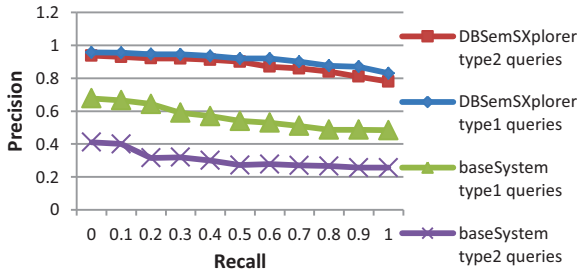

Figure 8. Precision/Recall Curve

We observed that DBSemSXplorer outperformed the baseSystem 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 outperforms the baseSystem for type1 queries is an indication of users leveraging concepts and relationships between them to find the relevant search results even thou the original queries were 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 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 baseSystem and in general based on our experiments MRR is higher for type1 queries than type2. We also observed that our system performs better in the range of 2 keywords to 4 on average

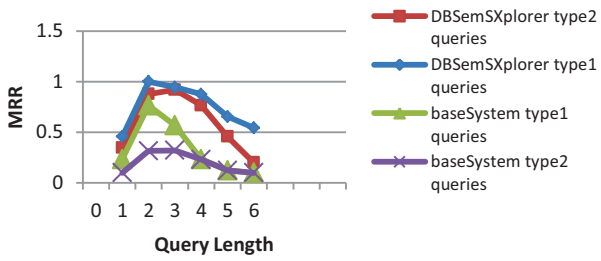

Figure 9. MRR-Query Length Plot

9. RELATED WORK

Keyword search has been researched over variety of different types of data sources such as unstructured data (e.g. text documents), structured data (e.g. relational databases) [1][2], semi structured data (e.g. XML) [8] and recently over semantic web(e.g. RDF)[4][5]. With keyword search, users are not required to have any knowledge of the underlying schema of the data structure, format, entity relationship model, etc. and they are not required to know any formal query languages (e.g. SQL, SPARQL, etc.) in order to search for the information. Rather, they can search for information by simply submitting keywords.

Our goal in this paper is not to solve the problem of keyword search over relational databases as it has been solved using few different approaches in the recent years. Our primary goal in this paper is to retrofit semantic search technologies onto relational databases in order to enhance the effectiveness of the search. There has not been much work in the area of semantic-based

keyword search in relational databases per se. SPARK transforms user's keyword query into SPARQL by mapping the query terms to the knowledgebase terms and then construct the formal queries. KEYMANTIC 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.
Many of our approaches and techniques used for keyword search over relational databases were inspired by previous works of SPARK, KEYMANTIC and EXPOSING RDB as RDF [4,5, 3].

10. CONCLUSION And The FUTURE WORK

In this paper we gave a brief overview of keyword search over relational databases and identified a couple of issues with finding the relevant results to the user's query and ranking them. 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 into the most semantically and syntactically similar knowledgebase 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 query. This improvement was specially observed when the users leveraged the concepts and the relationships between them when navigating for the relevant answers (regardless of which query type was performed).

In current approach we are only considering the attributes with short text such as actor's name, genre, studio's name, etc. For future work we are planning on adding tables' attributes with long string values such as movie description/summary, reviews etc. The challenges will be converting the values inside these columns into an RDF and incorporating them into the rest of the RDF generated from the RDB. The term mapping and SPARQL query construction techniques should be adjusted to consider the new generated knowledge base which contains the long string text attributes as new resource classes or relationships.

11. REFERENCES

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

[2] 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

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

[4] 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)

[5] 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

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

[7] 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

[8] Ziyang Liu , Jeffrey Walker , Yi Chen, XSeek: a semantic XML search engine using keywords, Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria