Towards An Interactive Keyword Search over Relational Databases

Zhong Zeng[1], Zhifeng Bao[2], Mong Li Lee[1], Tok Wang Ling[1]
[1]National University of Singapore [2]RMIT University
{zengzh,leeml,lingtw}@comp.nus.edu.sg zhifeng.bao@rmit.edu.au

ABSTRACT

Keyword search over relational databases has been widely studied for the exploration of structured data in a user-friendly way. However, users typically have limited domain knowledge or are unable to precisely specify their search intention. Existing methods find the minimal units that contain all the query keywords, and largely ignore the interpretation of possible users' search intentions. As a result, users are often overwhelmed with a lot of irrelevant answers. Moreover, without a visually pleasing way to present the answers, users often have difficulty understanding the answers because of their complex structures. Therefore, we design an interactive yet visually pleasing search paradigm called ExpressQ. ExpressQ extends the keyword query language to include keywords that match meta-data, e.g., names of relations and attributes. These keywords are utilized to infer users' search intention. Each possible search intention is represented as a query pattern, whose meaning is described in human natural language. Through a series of user interactions, ExpressQ can determine the search intention of the user, and translate the corresponding query patterns into SQLs to retrieve answers to the query. The ExpressQ prototype is available at `http://expressq.comp.nus.edu.sg`.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—Search process

Keywords

Keyword Search; Relational Database; Interactive Approach

1. INTRODUCTION

The primary interaction between relational databases (RDB) and users starts from SQL queries, which assumes that users are familiar with database schemas and the query language. As databases increase in size and become more accessible to a diverse and less technically oriented audience, new forms of data exploration become increasingly attractive, and keyword search is one of them to explore information in RDB.

Existing search methods typically considers a keyword query as a set of user specified keywords that match tuple values. One traditional approach to evaluate the query is to materialize the database as a graph where each node represents a tuple and each edge represents a foreign key-key reference. The query answers are the minimal connected subgraphs that contain all the keywords [1]. Another approach translates the keyword query into a set of SQL statements, and leverages on relational DBMSs to retrieve answers [2, 5]. However, these works do not consider users' search intentions due to the intrinsic ambiguity of keyword queries, and often return an overwhelming amount of answers, many of which are complex and not easily understood. As a result, users usually have to re-formulate their queries multiple times before they can get the desired information.

Our goal is to design an exploratory search paradigm that actively involves the user in the search process in order to address the following challenges in relational keyword queries:

1. How to identify the search target of a user query;
2. How to identify the context of a keyword since keywords can match tuple values as well as meta-data such as the names of relation and attributes [6];
3. How to present the query answers such that they facilitate human understanding and convey the relationship between data items [7, 3].

To achieve our goal, we build a system called ExpressQ, which provides an interactive approach to relational keyword search. ExpressQ captures the semantics of objects and relationships in the database, and handles keywords matching the names of relations and attributes. Given a keyword query, ExpressQ infers the search target of the query by identifying the objects/relationships referred to by the keywords. Then it constructs query patterns to represent the user's possible search intention, and rank the patterns. The meanings of these patterns are described in human natural languages in order to facilitate users' understanding. Based on the user's choice of query patterns, ExpressQ will generate SQL statements to retrieve the answers.

The work in [4] presents a natural language interface for querying RDB. However, their focus is how to understand a loosely structured query, while we strive to interpret the search intention of a keyword query and interactively construct SQLs.

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742830.

2. PRELIMINARIES

The relations in a database can be classified into object relations, relationship relations, mixed relations and component relations [7]. Intuitively, an object (relationship) relation contains information of objects (relationships), namely, the single-valued attributes of an object class (relationship type). A mixed relation contains information of both objects and relationships, which occurs when there is a many-to-one relationship. The multivalued attributes of objects and relationships are stored in component relations.

ExpressQ models the relational schema with an undirected graph called *Object-Relationship-Mixed (ORM) schema graph* $G = (V, E)$. Each node $v \in V$ comprises of an object/ relationship/ mixed relation and its component relations, and is associated with a $v.type \in \{object, relationship, mixed\}$. Two nodes u and v are connected via an edge $e(u, v) \in E$ if there exists a foreign key-key constraint between the relations in u and that in v.

Consider the sample company database in Figure 1. Figure 2 shows the corresponding ORM schema graph of the database. Note that node `Employee` is a mixed node because of the many-to-one relationships between employees and departments.

Employee

Eid	Name	Salary	Deptid	JoinDate
e1	Smith	3.5k	d1	2010
e2	Green	4.2k	d1	2009
e3	Brown	5.5k	d1	2006

EmployeeSkill

Eid	Skill
e1	Java
e2	SQL
e3	Java
e3	PhP

EmpProj

Eid	Pid	JoinDate
e1	p1	2010
e2	p1	2009
e2	p2	2010
e3	p2	2007
e3	p3	2008

Project

Pid	Name	Budget
p1	XML	40k
p2	RDB	50k
p3	Survey	30k

ProjDept

Pid	Deptid
p1	d1
p2	d1
p3	d2

Department

Deptid	Name	Address
d1	computing	Brown Street
d2	marketing	Queen Street

Figure 1: Sample company database

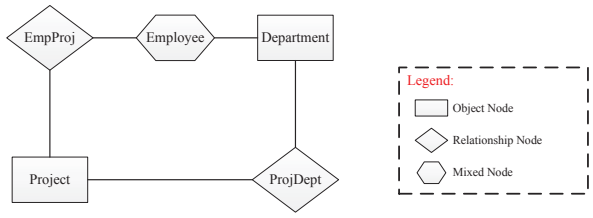

Figure 2: ORM schema graph of Figure 1

Suppose a user issues the keyword query `{Smith Green}`, where both the keywords match the names of several employees. Based on the ORM schema graph in Figure 2, some of the possible interpretations of this query are:

a. Find information on the project in which both employees `Green` and `Smith` are involved.

b. Find information on the department in which both employees `Green` and `Smith` work.

c. Find information on the department which conducts a project that involves the employee `Green` and the employee `Smith` works in.

Existing RDB keyword search engines [1, 2, 5] would consider all the above query interpretations and retrieve the corresponding information from the database. Consequently, the user is often overwhelmed by a lot of answers.

ExpressQ extends the keyword query language in order to reduce the interpretations of keyword queries. An extended keyword query consists of a sequence of keywords such that each keyword matches a relation name, or an attribute name or a tuple value. For instance, if the user is interested in the information on the department that both the employee `Smith` and the employee `Green` work in, s/he can issue the query as `{Department Employee Smith Employee Green}`. The keyword `Department` indicates that the user is interested in the information of the department, while the keyword `Employee` gives the context that `Smith` and `Green` refer to names of two employees.

3. SYSTEM ARCHITECTURE

Figure 3 depicts the architecture of ExpressQ. The system takes as input a keyword query, and generates a set of SQL statements that best capture the user's search intention. During the query processing, it interacts with the user in the front end and communicates with the database and its corresponding schema in the back end, in order to retrieve the information that the user is interested in. There are four main components in ExpressQ, namely, *Query Analyzer*, *Query Interpreter*, *Ranker*, and *SQL Generator*. The following sections discuss the functions of these components.

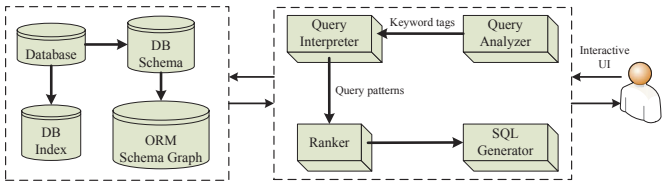

Figure 3: System Architecture

3.1 Query Analyzer

The Query Analyzer parses each keyword in the query and obtains the possible matches of the keywords. Based on the ORM schema graph of the database, the Query Analyzer determines the object/relationship that a keyword refers to, and creates tags for the keyword. A tag is given by $T = (label, attr, cond)$, where *label* is the name of the object/relationship, *attr* is the attribute name, and *cond* is the restriction on the object/relationship. A keyword may have multiple tags as it matches different objects or relationships. This results in multiple sequences of tags for a query. After obtaining a sequence of tags for the keywords in the query, the Query Analyzer groups the keywords that refer to the same object or relationship together.

Consider the keyword query `{Project Employee Green Brown}`. Table 1 shows two sequence of tags created for the query. Based on the tags, we can tell that the keywords `Project` and `Employee` refer to the names of projects and employees in the database, while the keyword `Green` refers to an employee name. Note that the keyword `Brown` has two tags. Tag T_{14} captures the information that `Brown` refers to

Table 1: Sequence of tags created for the query {Project Employee Green Brown}

T_{11} = (Project,null,null), T_{12} = (Employee,null,null), T_{13} = (Employee,Name,Green), T_{14} = (Employee,Name,Brown)
T_{11} = (Project,null,null), T_{12} = (Employee,null,null), T_{13} = (Employee,Name,Green), T'_{14} = (Department,Address,Brown)

an employee name, while tag T'_{14} captures the information that `Brown` refers to a department address.

Figure 4 shows the screenshot of the interface where ExpressQ lists the different matches of the keywords in this query for the user to select.

3.2 Query Interpreter

The Query Interpreter constructs a set of query patterns to represent the possible search intention of the user. A query pattern is a minimal connected graph derived from the ORM schema graph. Intuitively, the Query Interpreter creates a node to denote the object/relationship referred to by each group of keywords. These nodes will correspond to the nodes in the ORM schema graph and the Query Interpreter connects them based on the edges in the graph.

For example, given the first sequence of tags for the keyword query `{Project Employee Green Brown}` in Table 1, ExpressQ creates three nodes to denote a project object, the employee named `Green` and the employee named `Brown`. Based on the ORM schema graph in Figure 2, the `Project` node can connect to the `Employee` node via the `EmpProj` node. Hence, ExpressQ connects these three objects by creating two `EmpProj` relationships between the employees and the project. The query pattern P_1 obtained is shown in Figure 5, indicating that the user is interested in projects that involve both the employees `Green` and `Brown`.

Further, the `Project` node can also connect to the `Employee` node via the path `Project` − `ProjDept` − `Department` − `Employee` in the ORM schema graph. By creating nodes (`ProjDept` and `Department`) according to this path, we obtain the query pattern P_2 in Figure 5. This pattern depicts the user's intention to find projects which involves `Green` and are conducted by the department where `Brown` works.

3.3 Ranker

Since ExpressQ may generate multiple query patterns for a keyword query, it is necessary to rank these patterns. The Ranker in ExpressQ takes into account how many objects are involved in the query patterns. This is captured by the number of object/mixed nodes in the patterns. The Ranker also identifies the target nodes and the condition nodes by their tags. A target node indicates the output object of the query, and is typically specified by a keyword matching the name of a relation or an attribute. On the other hand, a condition node indicates the restrictions for the output object, and is specified by a keyword matching some tuple value. Consequently, query patterns with fewer object/mixed nodes, and a shorter average distance between target nodes and condition nodes will be ranked higher.

In Figure 5, the query pattern P_1 contains 3 object/mixed nodes while the query pattern P_2 contains 4 object/mixed nodes. Both P_1 and P_2 have one target node (`Project`) and two condition nodes (`Employee`). We compute the average distance between the `Project` node and two `Employee` nodes. P_1 has an average distance of 2 while the P_2 has an average distance of 2.5. Thus, P_1 is ranked higher than P_2.

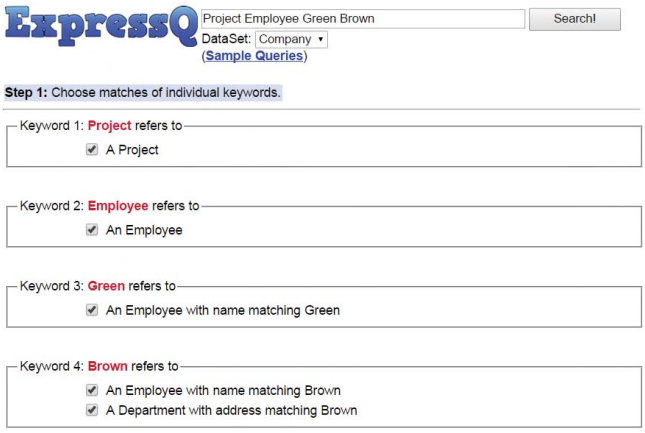

Figure 4: Screenshot of possible keyword matches

3.4 SQL Generator

The SQL Generator translates a query pattern into an SQL statement to retrieve the result from the database. Existing RDB keyword search engines typically generate SQLs that project every attribute of joining relations [2, 5]. As a result, many irrelevant attributes are projected, which makes the output overwhelming and difficult to understand.

In contrast, ExpressQ only projects the information on the target node. In particular, if the target node specifies the output object by its name, then the SQL Generator will include all the attributes of the object in the SELECT clause. Otherwise, if it specifies the name of an attribute of the output object, then the system will include the corresponding attribute in the SELECT clause.

For example, ExpressQ will translate the query pattern P_1 in Figure 5 to the following SQL statement:

```
SELECT P.Pid,P.Name,P.Budget
FROM Project P, EmpProj EP1, Employee E1, EmpProj EP2, Employee E2
WHERE P.Pid=EP1.Pid AND P.Pid=EP2.Pid AND
      EP1.Eid=E1.Eid AND EP2.Eid=E2.Eid AND
      E1.Name contains 'Green' AND E2.Name contains 'Brown'
```

Note that ExpressQ only outputs the relevant project information. In contrast, existing works will project the attributes of 5 relations in the FROM clause, and the output will contain a lot of irrelevant information.

3.5 User Interaction

One key feature of ExpressQ is its friendly interaction with the user to understand his/her search intention so that it can be selective in its generation of SQL statements and subsequent retrieval of relevant answers for the user. A keyword query is inherently ambiguous for the following reasons:

1. a query keyword can have multiple matches in the database, and

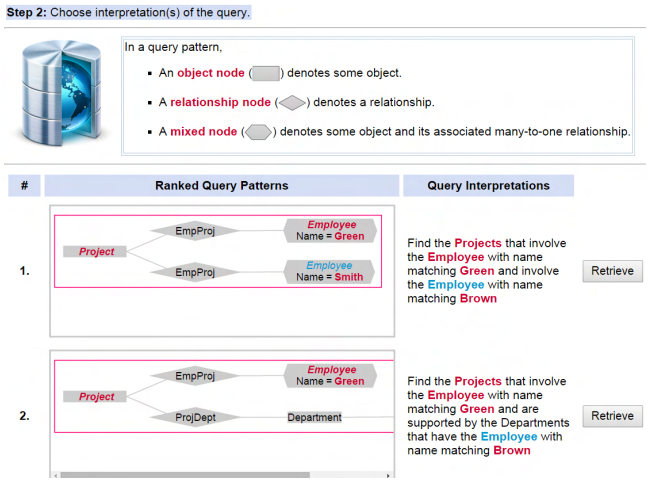

Figure 5: Screenshot of query interpretations

2. the keyword match objects can be connected via various relationships and form various interpretations.

As the user often has some particular search intention in mind, ExpressQ actively involves the user in the query evaluation process. In particular, if a keyword is associated with more than one tag, the user is offered the opportunity to choose the tag(s); if the Query Interpreter constructs more than one query pattern, the user is again allowed to select his/her intended query pattern and retrieve the corresponding answers. This interactive approach has the advantage of systematically leading the user to obtain answers that satisfy his/her search intention. This approach also gives the user insight into how the query is interpreted by the system and the results that can be expected.

Recall the query `{Project Employee Green Brown}`. Suppose the user issues this query to find the project that involves both the employees `Green` and `Brown`. Since the keyword `Brown` can refer to an employee named `Brown` or a department at `Brown` street, the ExpressQ shows the possible matches for the user to choose from (see Figure 4). If the user selects `Brown` as refering to an employee name, then ExpressQ will show how the employees `Brown` and `Green` can relate to a project in terms of query patterns (see Figure 5). Note that these patterns are ordered by their ranking scores.

Another feature of ExpressQ is it depicts the query interpretations and answers in human natural language to facilitate users' understanding. For instance, the tag T_{14} = (`Employee`, `Name`, `Brown`) in Table 1 is described as "`Brown` refers to an employee with name matching `Brown`". The query pattern P_1 in Figure 5 is represented as a tree annotated with the semantics of objects and relationships. The root of the tree denotes the output object while the leaves denote the restrictions on the output object. The meaning of this pattern is to "Find the projects that involve the employee with name matching `Green` and involve the employee with name matching `Brown`". Thereby, the user can easily identify the intended query pattern by the tree structure, and verify its meaning by the description. After the user selects a query pattern, ExpressQ retrieves the answers and represents them according to the corresponding search intention. Figure 6 shows the screenshot of the interface which displays the answers w.r.t. the query pattern P_1 in Figure 5.

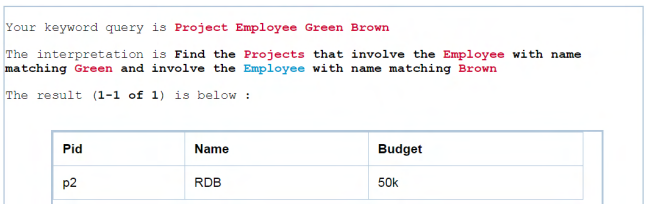

Figure 6: Screenshot of answers retrieved

4. DEMONSTRATION

In our demonstration, we will present a web-based browsing interface of ExpressQ, which communicates with the main Java based server. The system is available at `http://expressq.comp.nus.edu.sg`. We intend to show the use of ExpressQ against a number of real application scenarios such as the IMDB database (`www.imdb.com`), and the ACM Digital Library (`dl.acm.org`).

The demonstration will consist of two parts. First, we will run a number of sample keyword queries against these sources. We will demonstrate how ExpressQ exploits the semantics of objects/relationships in the database and utilize the keywords that match meta-data in the query to infer the search intention for these queries. Next, the user will be free to run their own queries. We will demonstrate how ExpressQ interactively leads the user to retrieve the intended answers effectively. During the query processing, the system will present different interpretations of the queries. The user will be able to choose the ones he/she is interested in, and thus obtain the answers that satisfy the search intention.

Through this demonstration, we will highlight three keypoints to the audience. First, the interpretation of the user's search intention is critical to keyword search over relational database. This requires the keyword search system to be knowledgeable about the semantics of objects and relationships in the database. Second, keywords that match the meta-data are helpful to infer the search intention of the user since they provide the context of subsequent keywords in the query. Third, the presentations of the query interpretations and query answers are important to facilitate user understanding and subsequent interaction with the system.

5. REFERENCES

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan. BANKS: Browsing and keyword searching in relational databases. In *VLDB*, 2002.

[2] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, and X. Yu. MeanKS: Meaningful keyword search in relational databases with complex schema. In *SIGMOD*, 2014.

[3] F. Li and H. V. Jagadish. Usability, databases, and HCI. *IEEE Data Eng. Bull.*, 35(3):37–45, 2012.

[4] F. Li and H. V. Jagadish. NaLIR: An interactive natural language interface for querying relational databases. In *SIGMOD*, 2014.

[5] Y. Luo, W. Wang, and X. Lin. SPARK: A keyword search engine on relational databases. In *ICDE*, 2008.

[6] Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and W. T. Ling. ExpressQ: Identifying keyword context and search target in relational keyword queries. In *CIKM*, 2014.

[7] Z. Zeng, Z. Bao, M. L. Lee, and T. W. Ling. A semantic approach to keyword search over relational databases. In *ER*, 2013.