ExpressQ: Identifying Keyword Context and Search Target in Relational Keyword Queries

Zhong Zeng
National University of Singapore
zengzh@comp.nus.edu.sg

Zhifeng Bao
HITLab Australia, University of Tasmania
zhifeng.bao@utas.edu.au

Thuy Ngoc Le
National University of Singapore
ltngoc@comp.nus.edu.sg

Mong Li Lee
National University of Singapore
leeml@comp.nus.edu.sg

Tok Wang Ling
National University of Singapore
lingtw@comp.nus.edu.sg

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ABSTRACT

Keyword search in relational databases has gained popularity due to its ease of use. However, the challenge to return query answers that satisfy users' information need remains. Traditional keyword queries have limited expressive capability and are ambiguous. In this work, we extend keyword queries to enhance their expressive power and describe an semantic approach to process these queries. Our approach considers keywords that match meta-data such as the names of relations and attributes, and utilizes them to provide the context of subsequent keywords in the query. Based on the ORM schema graph which captures the semantics of objects and relationships in the database, we determine the objects and relationships referred to by the keywords in order to infer the search target of the query. Then, we construct a set of minimal connected graphs called query patterns, to represent user's possible search intentions. Finally, we translate the top-k ranked query patterns into SQL statements in order to retrieve information that the user is interested in. We develop a system prototype called ExpressQ to process the extended keyword queries. Experimental results show that our system is able to generate SQL statements that retrieve user intended information effectively.

Categories and Subject Descriptors

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

Keywords

Keyword Query; Relational Database; Semantic Approach

1. INTRODUCTION

Designing effective query mechanisms for users to query large and complex databases easily has been one of the most elusive goals of the database research community. Traditional structured query models such as SQL for relational databases provide functionalities to query databases precisely. However, they require users to be knowledgeable about database schemas and query languages. Keyword queries have gained popularity due to their ease of use. However, they are inherently ambiguous and it is difficult to determine the users' search intention.

The traditional approach to evaluate a keyword query first materializes the database as a graph where each node represents a tuple and each edge represents a foreign key-key reference, and then finds the minimal connected subgraphs that contain all the keywords [8, 9, 6, 15]. But this is computationally expensive as the number of subgraphs is huge. Another approach translates a keyword query into a set of SQL statements, and leverages on relational DBMSs to evaluate the statements and retrieve answers [1, 7, 5, 12, 13]. However, these works do not analyze the users' search intention, and often returns an overwhelming amount of answers, many of which are complex and not easily understood.

Recently, [2] exploits the relative positions of keywords in a query along with auxiliary external knowledge to make an educated guess of the users' search intention. They measure the likelihood of mapping from a keyword to the database structure (a relation, an attribute, or a tuple), and generate the most probable mappings for the query keywords.

We observe that a relational database is essentially a repository of objects that interact with each other via relationships that are embedded in the foreign key-key references. When a user issues a query, s/he must have some particular search intention in mind. If we can determine the keywords that refer to the same object or relationship in the database, we would be able to infer the search target of the user. These keywords can also provide the context of subsequent query keywords that impose conditions on the search target.

Figure 1 shows a sample company database. Suppose a user issues a keyword query \{Employee Brown\}. The keyword Brown can refer to an employee name or a department address. However, since the keyword Employee matches the name of the Employee relation, we deduce that the user is more likely to be interested in an employee called Brown.
than the address of a department. In other words, a keyword that matches a relation name specifies a target object or relationship, while a keyword that matches an attribute name indicates the information that the user wants to retrieve for the target object/relationship. Keywords that match tuple values impose restrictions on the object/relationship.

<table>
<thead>
<tr>
<th>Employee</th>
<th>EmployeeSkill</th>
<th>EmpProj</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eid</td>
<td>Name</td>
<td>Pid</td>
</tr>
<tr>
<td>e1</td>
<td>Smith</td>
<td>p1</td>
</tr>
<tr>
<td>e2</td>
<td>Green</td>
<td>p2</td>
</tr>
<tr>
<td>e3</td>
<td>Brown</td>
<td>p3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>ProjDept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pid</td>
<td>Name</td>
</tr>
<tr>
<td>p1</td>
<td>RDB</td>
</tr>
<tr>
<td>p2</td>
<td>SQL</td>
</tr>
<tr>
<td>p3</td>
<td>Java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deptid</td>
</tr>
<tr>
<td>d1</td>
</tr>
<tr>
<td>d2</td>
</tr>
</tbody>
</table>

**Figure 1: Example company database**

In this work, we extend the expressive power of keyword queries so that users can better express their search intentions. We design a semantic approach to process these extended keyword queries. Our approach considers keywords that match meta-data e.g., names of relations and attributes, and utilizes them to provide the context of subsequent keywords in the query. We discover the various ways objects in a database interact with each other, and construct query patterns to denote user’s possible search intentions. We propose a ranking scheme that takes into account the search targets of the query as well as the number of objects captured in a query pattern. The top-k ranked query patterns are used to generate SQL statements. We develop a prototype system called ExpressQ to process extended keyword queries. Experimental results on two databases demonstrate the effectiveness of ExpressQ in generating SQL statements to retrieve relevant information for users.

2. **PRELIMINARIES**

The work in [16] classifies the relations in a database into object relations, relationship relations, mixed relations and component relations. An object (relationship) relation captures the information of objects (relationships), i.e., the single-valued attributes of an object class (relationship type). Multivalued attributes are captured in the component relations. A mixed relation contains information of both objects and relationships, which occurs when we have a many-to-one relationship.

In our example database in Figure 1, relations *Project* and *Department* are object relations as they contain single-valued attributes of project and department objects respectively. Relations *EmpProj* and *ProjDept* are relationship relations. *Employee* is a mixed relation as it captures information of employee objects and the many-to-one relationships between employee and department objects. The relation *EmployeeSkill* is a component relation containing the multivalued attribute of employees.

We can model the relational schema with an undirected graph called Object-Relationship-Mixed (ORM) schema graph \( G = (V, E) \) [16]. Each node \( v \in V \) comprises of an object/relationship/mixed relation and its component relations, and is associated with a \( v.type \in \{ \text{object, relationship, mixed} \} \). Two nodes \( u \) and \( v \) are connected via an edge \( e(u, v) \in E \) if there exists a foreign key-key constraint from the relations in \( u \) to that in \( v \).

**Figure 2: ORM schema graph of Figure 1**

Figure 2 shows the ORM schema graph of the database in Figure 1. It has two object nodes (rectangle), two relationship nodes (diamond) and one mixed node (hexagon). The node *Department* is an object node that comprises of the object relation *Department*, while the node *Employee* is a mixed node that comprises of the mixed relation *Employee* and component relation *EmployeeSkill*. These two nodes are connected via an edge because there is a foreign key-key constraint from the relation *Employee* to the relation *Department*.

Suppose a user issues the keyword query \( \{ \text{Smith Green} \} \). Both keywords match some employee name in the *Employee* relation. Based on the ORM schema graph in Figure 2, these two employees are possibly related via the relationship *EmpProj*, the many-to-one relationship with departments, and a combination of these relationships. Thus, some of the possible interpretations of the 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 works would consider all the above query interpretations and retrieve the corresponding information from the database. Consequently, the user is often overwhelmed by a huge number of answers, many of which are complex and not easily understood.

In order to reduce the ambiguity of keyword queries, we propose to allow users to explicitly indicate his/her search intention whenever possible. This can be achieved by augmenting the query with additional keywords that match the names of relations and attributes. For example, if the user would like to find the information on the department that both the employee *Smith* and the employee *Green* work in, s/he can express the query as \( \{ \text{Department Employee Smith Employee Green} \} \). The keyword *Department* matches the name of the *Department* relation, indicating that the user is interested in the information of the department. While the keyword *Employee* matches the name of the *Employee* relation, giving the context that the keywords *Smith* and *Green* refer to names of two employees.

**Definition 1.** An extended keyword query consists of a sequence of keywords \( Q = \{k_1, k_2, \ldots, k_n\} \) such that each keyword \( k \) matches a relation name, or an attribute name or a tuple value.
3. KEYWORD QUERY EVALUATION

Given an extended keyword query, we want to generate a set of SQL statements that best capture the user's search intention. This entails the following steps:

1. **Query analysis.** We parse each keyword in the query and utilize the ORM schema graph of the database to determine the object or relationship that a keyword refers to. The semantic information of each keyword is captured in a tag, and the tags that refer to the same object or relationship are grouped together.

2. **Query interpretation.** Based on the groups of tags, we generate a set of minimal connected graphs called query patterns, that represent the possible search intentions of the query, and rank these patterns.

3. **SQL statement generation.** The top-k ranked query patterns are used to generate SQL statements to retrieve results from the relational database.

The following sections give the details of each of these steps.

### 3.1 Query Analysis

Given an extended keyword query \( Q = \{k_1, k_2, \ldots, k_n\} \), we will determine the interpretation(s) of each keyword in \( Q \). We capture each keyword interpretation in a tag \( T = (\text{label}, \text{attr}, \text{cond}) \), where \( \text{label} \) is the name of the object or relationship, \( \text{attr} \) is the attribute name, and \( \text{cond} \) is the restriction on the object or relationship. The restriction occurs in the form of a value. The tag(s) for a keyword \( k \) is generated depending on the following type of matches:

a. \( k \) matches the name of some object/mixed/relationship relation.

   This indicates that \( k \) refers to some object or relationship. The name of the object is given by the corresponding object or mixed node in the ORM schema graph, while the name of the relationship can be obtained from the corresponding relationship node in the graph. We capture this keyword interpretation by creating a tag \((k, \text{null}, \text{null})\) for this keyword.

b. \( k \) matches the name of a component relation or an attribute name.

   This implies that \( k \) refers to the attribute of the some object or relationship \( l \), and we create a tag \((l, k, \text{null})\) for this keyword.

c. \( k \) matches some tuple value.

   Clearly, \( k \) refers to the value of some attribute \( a \) of an object or relationship \( l \), and we create a tag \((l, a, k)\) for this keyword.

**Example 1.** Consider the keyword queries in Table 1. Table 2 shows the sequence of tags generated for these queries. For query \( Q_1 \), we know that the keyword \textit{Department} matches an object relation, while the keyword \textit{Employee} matches a mixed relation from the ORM schema graph in Figure 2. Hence, these keywords refer to the names of department and employee objects in the database, and we capture their interpretations in the corresponding tags. On the other hand, the keywords \textit{Smith} and \textit{Green} match the Name attribute values of some tuples in the Employee relation, and we capture these information in their tags \( T_{13} \) and \( T_{15} \).

Query \( Q_2 \) contains the keyword \textit{Skill} that matches an attribute name. From the ORM schema graph, we see that this attribute belongs to the component relation EmployeeSkill which is associated with the mixed node Employee. Hence, we know that skill is a multivalued attribute of the employee object and we capture this interpretation in the tag \( T_{23} \). Further, the keywords \textit{Java} and \textit{Php} match the Skill attribute values of some tuples.

Note that \( Q_3 \) contains the keyword \textit{Brown} that matches the Name attribute value of some tuple in Employee relation, as well as the Address attribute value of some tuple in Department relation. In this case, we create a tag for each matching. The tag \( T_{34} \) in Table 2 captures the interpretation that keyword \textit{Brown} refers to an employee name, while the tag \( T_{34} \) captures the interpretation that \textit{Brown} refers to a department address. Hence, we see that \( Q_3 \) has two sequences of tags, denoting two different query interpretations.

After creating a sequence of tags for the keywords in the query, we group the tags that refer to the same object or relationship together. Clearly, tags that do not have the same \textit{label} are placed into different groups since their keywords refer to different objects/relationships. However, keywords with tags that have the same \textit{label} do not necessarily refer to the same object/relationship.

**Example 2.** Consider query \( Q_1 \) in Table 1 and its tags in Table 2. All these tags except \( T_{11} \) have the same label \textit{Employee}. However, their keywords actually refer to different objects: keywords \textit{Employee} and \textit{Smith} refer to the employee named \textit{Smith}, while keywords \textit{Employee} and \textit{Green} refer to the employee named \textit{Green}.

This example demonstrates the need to process the tags of keywords in a keyword query in sequence, and examine the objects or relationships referred to by the current and preceding keywords.

Let \( T \) be the sequence of tags for a query \( Q \). We put a tag \( T_i \in T \) into a new group to denote a different object/relationship if one of these following cases is true:

- **Case 1.** \( T_i \) has a different \textit{label} from all the tags \( T_j \in T \), \( j \in [1, i - 1] \).

- **Case 2.** \( T_i \) has the same \textit{label} as \( T_j \in T \) for some \( j \in [1, i - 1] \), and the \textit{attr} and \textit{cond} of \( T_i \) are null.

- **Case 3.** \( T_i \) has the same \textit{label} and \textit{attr} as \( T_j \in T \) for some \( j \in [1, i - 1] \), and \textit{attr} is not a multivalued attribute.

A tag that satisfies Case 2 indicates that its keyword refers to a new object/relationship and provides the context for the next keyword in the query. Hence, we create a new group for this tag. On the other hand, a tag that satisfies Case 3 indicates that both its keyword and the preceding keyword refer to the same single-valued attribute of an object/relationship or its values. Since an object/relationship cannot have a single-valued attribute with two values, the keyword of this tag must refer to a new object/relationship.

**Example 3.** Let us consider the sequence of tags for \( Q_1 \). Tags \( T_{11} \) and \( T_{12} \) belong to two different groups \( g_1 \) and \( g_2 \).
since they have different labels. Since tag $T_{13}$ has the same label Employee as its preceding tag $T_{12}$, it is placed in the same group $g_2$. $T_{14}$ is put in a new group $g_3$ because its attr and cond are null and it refers to a different employee object (Case 2). Since $T_{15}$ has the same label as $T_{14}$, it is put in $g_3$. Note that we cannot put $T_{15}$ in $g_2$ because Name is a single valued attribute of employees, and it is not possible for the same employee to have two different names (Case 3). In other words, the keywords Smith and Green are the names of two different employees. Hence, the tags for $Q_1$ are grouped as follows:

- $g_{11} = \{T_{11}\}$ refers to some department object,
- $g_{12} = \{T_{12}, T_{13}\}$ refers to an employee named Smith,
- $g_{13} = \{T_{14}, T_{15}\}$ refers to an employee named Green.

The following example illustrates a query involving multi-valued attributes.

**Example 4.** Consider query $Q_2$ and its sequence of tags in Table 2. The tags $T_{22}, T_{23},$ and $T_{24}$ have the same label and are put in the same group. Note that $T_{24}$ and its preceding tag $T_{23}$ have the same label and attr. They are put in the same group because the attribute Skill is a multivalued attribute. In other words, the keywords Java and PhP refer to the values of the multivalued attribute Skill, and the user is interested in an employee who knows both Java and PhP. Hence, the tags for $Q_2$ are grouped as follows:

- $g_{21} = \{T_{21}\}$ refers to some project object,
- $g_{22} = \{T_{22}, T_{23}, T_{24}\}$ refers to an employee with skills Java and PhP.

### 3.2 Query Interpretation

After grouping the tags of a query, the next step is to generate query patterns. Each query pattern is a minimal connected graph that represents one possible search intention of the query. Intuitively, we construct a query pattern by creating a node to represent each object/relationship referred to by each group of tags. These tags will correspond to nodes in the ORM schema graph and we can connect them based on the edges in the graph.

Let $S = \{g_1, g_2, \ldots, g_m\}$ be a set of tag groupings, and $G = (V, E)$ be the ORM schema graph. A query pattern $P = (V', E')$ is constructed as follows. For each group of tags $g_i \in S$, $1 \leq i \leq m$, we create a node $v_i$ to denote the object/relationship referred to by $g_i$. The corresponding object class or relationship type is given by a node $v_i$ in $G$. We say that $v_i$ corresponds to $g_i$.

Let $D' = \{v_1, v_2, \ldots, v_n\}$ and $D = \{v_1, v_2, \ldots, v_n\}$. We first insert the nodes in $D'$ into the query pattern $P$. If $|D'| = 1$, then all the tags in the query are in one group which refers to the same object/relationship. Hence, the query pattern $P$ has only a single node. However, if we have $|D'| > 1$, then we need to use the schema graph $G$ to connect these nodes. We have two cases to handle:

**Case A.** The object class or relationship type of every object/relationship is distinct.

In this case, all the nodes in $D'$ correspond to distinct nodes in $D$, i.e., $|D'| = |D|$. We find a minimal subgraph $H$ of $G$ that connects all the nodes in $D$. For each intermediate node $x$ in $H$, we create a node $x'$ that corresponds to $x$ and insert it into $P$. For each edge $e(x, y)$ in $H$, we create an edge $e(x', y')$ in $P$.

**Case B.** The object class or relationship type of every object/relationship is not distinct.

In this case, some objects (or relationships) have the same object class (or relationship type). In other words, two or more nodes in $D'$ correspond to the same node in $G$, i.e., $|D'| > |D|$. We cluster the nodes in $D'$ according to their object classes, and connect the nodes between the clusters. We try to find a node $u \in D'$ such that $u$ can connect to the other nodes in $D'$ based on the paths between their corresponding nodes in the ORM schema graph. If no such node exists, we create a node $x'$ that corresponds to some node $x \in G$ to connect all the nodes in $D'$.

The following examples illustrate the above cases.

**Example 5.** Let us consider the set of tag groupings $S = \{g_{21}, g_{22}\}$ obtained for query $Q_2$ in Example 4. We create two nodes $u_1$ and $u_2$ to represent $g_{21}$ and $g_{22}$ respectively. Nodes $u_1$ and $u_2$ correspond to the nodes Project and Employee in the ORM schema graph in Figure 2 respectively (Case A). Since the Employee and Project nodes can be connected via the EmpProj node in the ORM schema graph, we create a new node $u_3$ that corresponds to the EmpProj node to connect $u_1$ and $u_2$, and output the graph as a query pattern (see Figure 3). This query pattern captures the user’s intention to find information on the project that involves the employees with both skills Java and PhP.

![Figure 3: Query pattern in Example 5](image)

**Example 6.** Consider the 3 groups of tags $g_{11}, g_{12}$ and $g_{13}$ obtained for $Q_1$ in Example 3. We create a set of nodes $D' = \{u_1, u_2, u_3\}$ for these groups, and $D = \{Employee, Department\}$. Note that the corresponding object classes of the nodes in $D'$ are not distinct (Case B). Node $u_1$ corresponds to the Department node in the ORM schema graph, while both nodes $u_2$ and $u_3$ correspond to the Employee node. We cluster the nodes in $D'$ according to their object classes, and connect the nodes between the clusters, i.e., $c_1 = \{u_1\}$ and $c_2 = \{u_2, u_3\}$, and we try to connect $u_1$ to $u_2$ and $u_3$.

Based on the ORM schema graph, the Department node can connect to the Employee node directly. Hence, we create two edges to connect $u_1$ to $u_2$ and $u_3$. Figure 4(a) shows the query pattern $P_1$ obtained which indicates that the user wants to find information on the department that both employees Smith and Green work in.
Further, we observe that the Department node can also connect to the Employee node via the path Department – ProjDept – Project – EmpProj – Employee in the ORM schema graph. By creating nodes $u_2$ and $u_3$ (correspond to ProjDept), $u_4$ and $u_5$ (correspond to Project), $u_6$ and $u_9$ (correspond to EmpProj), we obtain the query pattern $P_2$ in Figure 4(b). This pattern indicates that the user is interested in the department with projects involving both employees Smith and Green. □

### 3.3 Query Pattern Ranking

After generating the various query patterns, the next step is to rank them. The standard method typically ranks graphs based on the number of nodes, i.e., a smaller graph is more easily understood and is ranked higher than a larger complex graph. This approach does not consider the semantics of objects and relationships in the graphs. For example, a query to find an employee who have both skills Java and PHP will have a graph with 3 nodes (EmployeeSkill – Employee – EmployeeSkill), each of which denotes a relation tuple. However, all 3 nodes refer to the same employee object. It should not be ranked equally as a graph with 3 nodes such as Employee – EmpProj – Project, where nodes Employee and Project refer to two different objects.

We observe that when a user issues a query, s/he must have some particular search intention in mind. We refer to the objects/relationships that meet the user’s interest in the search intention as the search targets of the query. Our proposed ranking scheme aims to take into account the search targets of the query as well as the number of object/mixed nodes in the query patterns.

In order to identify the search targets of a query, we classify the nodes that correspond to the tag groups into target nodes and condition nodes since they denote the objects and relationships that the user is interested in. A target node specifies the search target of the query, while a condition node indicates the search conditions of the query. In our Example 6, $u_1$ is a target node since the user is interested in the information on a department. On the other hand, $u_2$ and $u_3$ are condition nodes as they specify two particular employees by their names. We define target nodes and condition nodes formally as below:

**Definition 2.** Let $u$ be an object/relationship node referred to by a tag group $g$. We say $u$ is a condition node if \( \exists \ T \in g \text{ such that } T.\text{cond} \neq \text{null} \). Otherwise, $u$ is a target node if:

1. $\forall \ T \in g, \text{ we have } T.\text{cond} = \text{null}$ or
2. $\exists \ T \in g \text{ such that } T.\text{attr} \neq \text{null} \text{ and } \exists \ T' \in g \text{ where } T'.\text{attr} = T.\text{attr}$

The first condition indicates that the user is interested in the information of an object/relationship, while the second condition indicates that the user is interested in obtaining information on an object/relationship attribute.

Note that a node can be both a target node and a condition node. Suppose we have the following group of tags:

- $T_1 = (\text{Department},\text{Name},\text{null})$
- $T_2 = (\text{Department},\text{Address},\text{Queen})$
- $T_3 = (\text{Department},\text{Name},\text{null})$

This group of tags refers to a department object node that is both a target and a condition node since the semantics of the tags indicates that the user is interested in the name of the department in Queen street.

Let $X$ be the set of target nodes, $Y$ be the set of condition nodes, and $N$ be the number of object and mixed nodes in a query pattern $P$. We compute a score for $P$ by counting the number of objects involved in the query pattern and the average distance between the target and condition nodes.

We define the score of a query pattern $P$ as:

$$
\text{score}(P) = \frac{1}{N} \sum_{u \in X, v \in Y} \frac{\text{dist}(u, v, P)}{|X| \cdot |Y|}
$$

where $\text{dist}(u, v, P)$ is the total number of object and mixed nodes in the path connecting two nodes $u$ and $v$ in $P$.

Query patterns with fewer object/mixed nodes, and a shorter average distance between target nodes and condition nodes will be scored higher.

**Example 7.** Figure 5 shows two query patterns $P_1$ and $P_2$ for the query $Q_2 = \{\text{Project Employee Green Brown}\}$. $P_1$ indicates that the user wants to find the information on the project that involves both the employees Green and Brown, while $P_2$ indicates that the user is interested in the project involving the employee Green who works in the department in Brown street. Both these query patterns have 3 object/mixed nodes. Besides, node $u_1$ is a target node while nodes $u_2$ and $u_3$ are condition nodes. We compute the average distance between the target node ($u_1$) and the condition nodes ($u_2$ and $u_3$) for both query patterns. $P_1$ has an average distance of $\frac{2+2}{2} = 2$, while $P_2$ has an average distance of $\frac{2+2+2}{3} = 2.5$. Thus, we have $\text{score}(P_1) = \frac{1}{3}$ and $\text{score}(P_2) = \frac{2}{15}$, and $P_1$ will be ranked higher than $P_2$.

We see that this ranking complies with human intuition that both the employees Green and Brown are “closely” related to the target project in $P_1$. In contrast, the department in Brown street is related to the target project just because it has some employee (Green) involves in the project in $P_2$. On the other hand, using standard method which ranks graphs based on the number of nodes will give $P_1$ a lower rank than $P_2$, as $P_1$ consists of 5 nodes while $P_2$ consists of 4 nodes. □
Note that a query may not contain keywords that explicitly indicate the search targets. For example, none of the keywords in the query \{Project XML Project RDB\} indicate the search targets explicitly. In this case, we will need to infer the target nodes.

The work in [11] defines the centric distance of a node \( u \) as the longest distance between \( u \) and any node in the graph. Further, the radius of the graph is the minimal value among the centric distances of every node. A query answer is a graph whose radius is not larger than a specified value. Here, we use the radius of query patterns to determine the target nodes.

We define the centric distance of a node \( u \) in \( P \) as the longest distance between \( u \) and any node \( v \) in \( P \), that is,

\[
\text{centric}(u, P) = \max_{v \in P} \text{dist}(u, v, P)
\]

Then the radius of \( P \) is given by the shortest centric distance among all the nodes in \( P \). We infer that a node \( u \) is a target node if its centric distance is equal to the radius of \( P \).

**Example 8.** Figure 6 shows a query pattern for the query \{Project XML Project RDB\} with 3 object nodes. Both the nodes \( u_1 \) and \( u_2 \) correspond to the Project node in the ORM schema graph and are condition nodes. There is no target node in the pattern according to Definition 2. Thus, we will look for a node whose centric distance is equal to the radius of the query pattern. This gives us \( u_4 \) as the target node, indicating that the user is interested in the department that conducts both projects XML and RDB. We compute the average distance between the target node \( u_4 \) and the condition nodes \( u_1 \) and \( u_2 \), and obtain a score of \( \frac{1}{2} \) for this pattern.

![Figure 5: Query patterns in Example 7](image)

![Figure 6: Query pattern in Example 8](image)

### 3.4 SQL Statement Generation

Finally, we generate a set of SQL statements based on the top-k query patterns. These SQLs statements are used to retrieve results from the relational database. The results are then returned as answers to the extended keyword query.

For each query pattern \( P \), we generate an SQL statement as follows:

**SELECT clause.** For each target node \( u \) in \( P \), if \( u \) specifies a search target via the object or relationship name (Condition 1 in Definition 2), then we include all the attributes of the relations of \( u \) in the SELECT clause. Otherwise, if \( u \) specifies a search target via an attribute name (Condition 2 in Definition 2), then we include only the corresponding attribute of the relations of \( u \) in the SELECT clause. If \( u \) is inferred from the radius of \( P \), then we assume that user is interested in all of information of \( u \) and include all the attributes of the relations of \( u \) in the SELECT clause.

**FROM clause.** The FROM clause contains the relations of all the nodes in \( P \).

**WHERE clause.** The WHERE clause joins the relations in the FROM clause based on the foreign key-key constraints. Further, for each condition node \( u \) in \( P \), we check the group of tags that refer to the object/relationship denoted by \( u \). For each tag \( T \) such that \( T\text{cond} \neq \text{null} \), we include the condition “\( T\text{cond}.\text{label}.\text{attr} \) contains \( T\text{cond}.\text{val} \)” in the WHERE clause.

**Example 9.** Consider the query pattern \( P_1 \) in Figure 4(a). Node \( u_1 \) is a target node and denotes a department object, while nodes \( u_2 \) and \( u_3 \) are condition nodes that denote the employee objects named Smith and Green respectively. We will generate the following statement for the query pattern:

```sql
SELECT D.Deptid,D.Name,D.Address
FROM Department D,Employee E1,Employee E2
WHERE D.Deptid=E1.Deptid AND D.Deptid=E2.Deptid AND
E1.Name contains 'Smith' AND E2.Name contains 'Green'
```

### 4. EXPRESSQ SYSTEM PROTOTYPE

We design a system called ExpressQ which enables users to query the database using extended keyword queries. Figure 7 shows the main components in ExpressQ: (a) Query Analyzer, (b) Query Interpreter, and (c) SQL Generator.

Given a query \( Q \) and the ORM schema graph \( G \), we first call the Query Analyzer to produce a list of sets of tag groupings. Since a keyword in an extended keyword query may be ambiguous, and may be associated with multiple tags, we enumerate the different sequence of tags for a query. For each sequence of tags, the Query Analyzer produces a set of tag groupings. Then we call the Query Interpreter to generate a list of query patterns for each set of tag groupings. We compute the scores of the query patterns and output the top-k patterns. Finally, we generate the SQL statements for the top-k ranked query patterns. The results of these SQLs are returned as the answers of \( Q \).
Algorithm 1: QueryAnalyzer

Input: Q = {k1, ..., kn}; ORM schema graph G
Output: list of sets of tag groupings L
1. L ← ∅;
2. for i = 1 to n do
3.    TagListi ← createTags(ki, G);
4.    foreach tag sequence (T1, ..., Tn), Ti ∈ TagListi, do
5.        S ← ∅;
6.        Let g = (T1); glabel = T1.label, newg = false;
7.        for i = 2 to n do
8.            if Ti.label ≠ glabel then
9.                newg = true;
10.           else if Ti.attr = null ∧ TiCOND = null then
11.                newg = true;
12.           else if Ti attr is not multivalued ∧ 3j ∈ [i − |g|, i − 1]
13.              such that Tj.attr = Ti.attr then
14.                newg = true;
15.           if newg is true then
16.                S = S ∪ {g}; g ← ∅;
17.           g = g ∪ (Tj); glabel = Tj.label;
18.          else
19.              S = S ∪ {g};
20.         Insert S into L;

Algorithm 1 describes the details of the Query Analyzer. We first create a list of tags for each keyword in the query (Lines 2-3). For each sequence of tags, we group the tags that refer to the same object or relationship. We initialize S and put the tag of the first keyword T1 into a group g (Lines 5-6). Next, we check whether the tags of the subsequent keywords can be put into the same group g (Lines 8-13), and create new groups if needed (Lines 14-19). Finally, we insert S into the list L (Line 20).

Algorithm 2 gives the details of the Query Interpreter. For each group of tags gi in S, we create a node ui to denote the object/relationship referred to by gi. We add ui into set D′ and its corresponding node vi in the ORM graph into set D. Then we insert ui into the query pattern P (Lines 3-7).

If D′ has only one node, we simply add P into Plist (Lines 8-9). Otherwise, we compare the number of nodes in D′ and D. If |D′| = |D|, we find the minimal subgraph H of G that connects all the nodes in D. For each intermediate node x in H, we create a node x′ that corresponds to x and insert it into P. For each edge e(x, y) in H, we create an edge e(x′, y′) in P. Then we add P into Plist (Lines 10-16). If |D′| is larger to |D|, we divide the nodes in D′ into clusters c1, c2, ..., cm such that the nodes in each ci corresponds to a node vi in D, and |ci| ≤ |ci+1|, ∀i ∈ [1, m − 1]. If the smallest cluster c1 has only one node u1, then we connect u1 to the nodes in c2, c3, ..., cm. Let H be the path that connect v1 and vi in G. We connect u1 to the nodes in c1 based on H (Lines 17-25).

On the other hand, if c1 has more than one node, then we will use a node x′ that corresponds to some node x ∈ G − D to connect all the nodes in c1, c2, ..., cm. For each object or mixed node x in G − D, we first create a copy P′ of P, and insert a node x′ that corresponds to x into P′. Again, let H be the path that connect x and vi in G. We connect x′ to the nodes in ci based on H. After all the nodes in D′ are connected, we add P′ into Plist (Lines 26-34).

5. PERFORMANCE STUDY

In this section, we evaluate the effectiveness and efficiency of the ExpressQ system. We implement the algorithms in Java and carry out experiments on an 3.40GHz CPU with 8GB RAM. We use two relational databases in our experiments: the TPC-H database (TPCH) and the ACM Digital Library publication (ACMDL).

We construct 7 queries for each database. Tables 3 and 4 show the queries and the corresponding descriptions (or search intentions). The keywords of these queries may match relation names, attribute names and tuple values.

5.1 Effectiveness Experiments

One of the advantages of ExpressQ is its ability to identify the context of keywords and the search target of a query to retrieve user’s intended information. We verify its effectiveness by comparing ExpressQ with Spark [13], an existing relational keyword search engine that does not consider keyword contexts or search targets.

Spark finds the relations whose tuples matches the query keywords, and constructs a set of minimal connected graphs called candidate networks based on these relations. The candidate networks are ranked according to their size, and SQL statements are generated from the top-k networks.

5.1.1 Results for the TPCH Database

Table 5 shows the generated SQL statements that best match the descriptions of the queries for the TPCH database. We see that although both ExpressQ and Spark generate the same SQL statement for query T1, they differ greatly for the rest of the queries.

Queries T2 to T4 show that ExpressQ is more selective in its retrieval of information as it identifies the search target in the query. ExpressQ retrieves only the retail price of the part
The results of this set of experiments clearly indicate that ExpressQ has identified \textit{retailprice} as the search target in \textit{T2}. Similarly, ExpressQ retrieves only the customer phone and order date information for \textit{T3} and \textit{T4} respectively, while Spark retrieves all the attributes of the relations in the FROM clause.

Queries \textit{T5} to \textit{T7} demonstrate that by considering the context of keywords, ExpressQ is able to generate SQL statements to retrieve information that the user is interested in. ExpressQ uses the context provided by the keyword \textit{supplier} in both \textit{T5} and \textit{T6} to correctly generates SQL statements that retrieve supplier information.

In contrast, Spark generates SQL statements to retrieve information on the nation ‘Canada’ for \textit{T5}, and part information for \textit{T6}. We see that this does not match the query descriptions of \textit{T5} and \textit{T6}. Similarly, for \textit{T7}, the SQL statement obtained from ExpressQ retrieves the intended customer information, while Spark retrieves item information instead.

### 5.1.2 Results for the ACM DL Database

Table 6 gives the results for the ACM DL database. Both ExpressQ and Spark generate the same SQL statements for queries A1 and A2 because these are relatively straightforward keyword queries. However, for query A3, ExpressQ correctly retrieves the affiliation of the author ‘Jagadish’ while Spark retrieves all the attributes of the author. This is because Spark is unable to identify that affiliation is the search target of a query.

Queries A4 to A7 show that the context of keywords is important and enables ExpressQ to correctly generate SQL statements that retrieves the intended information: publisher information for A4, paper information for A5, editor information for A6, and author information for A7. On the other hand, we observe that Spark retrieves information that clearly do not match the query descriptions, e.g., proceedings for A4, and authors for A5.

The results of this set of experiments clearly indicate that identifying the keyword context and search target of queries greatly enhances the evaluation of keyword queries and leads to the retrieval of appropriate information.

### 5.2 Efficiency Experiments

Finally, we compare the time taken by ExpressQ and Spark to generate SQL statements for the queries. Figure 8 shows the results for both TPCH and ACM DL queries in Tables 3 and 4 on cold cache.

We observe that Spark is faster than ExpressQ when the number of nodes in the candidate network/query pattern is small. This is because Spark does not analyze the search intention of the queries but only finds candidate networks containing all the keywords that match tuple values.

However, Spark is slower than ExpressQ for queries T4, T7, A6 and A7. This is because the number of nodes in the candidate networks for these queries is large, and Spark needs more time to enumerate the networks in a breadth-first traversal manner. For example, Spark generates 339 intermediate graphs before finding the top 3 candidate networks for query T7. In contrast, ExpressQ finds the path \textit{customer-lineitem-orders} in the ORM schema graph, and builds the query pattern based on this path directly.

Figure 9 compares the time taken by ExpressQ to generate SQL statements, and the time needed to execute these statements over the databases. We see that the execution of SQL statements dominates the overall processing time (in seconds), indicating that the extra time taken by ExpressQ to analyze the queries (in ms) to identify the search intention of the user is a good tradeoff. We note that the execution time of SQL statements varies significantly with the number of results retrieved and the number of joins. For example, the SQL statement for query T7 takes more than 30 minutes to join 5 relations, while the SQL statement for query T5 takes only 78 ms to retrieve 412 results.

### 6. RELATED WORK

Existing works in relational keyword search typically use a schema graph or a data graph to process a keyword query. The schema graph approach models the database schema as an undirected graph where each node represents a relation and each edge represents a foreign-key key constraint. Early works such as DBXplorer [1] and DISCOVER [7] propose a breadth-first traversal on the schema graph to generate a set of SQL statements. The output tuples of each SQL contain

---

**Table 3: Queries for the TPCH database**

<table>
<thead>
<tr>
<th>#</th>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>part type nickel</td>
<td>find the information of the parts with type ‘nickel’</td>
</tr>
<tr>
<td>T2</td>
<td>part retailprice name rose</td>
<td>find the retail price of the part ‘rose’</td>
</tr>
<tr>
<td>T3</td>
<td>customer phone mktsegment automobile</td>
<td>find the phone of the customers who are in ‘automobile’ market segment</td>
</tr>
<tr>
<td>T4</td>
<td>orders date priority high China</td>
<td>find the date of the orders that have ‘high’ priority and come from ‘China’</td>
</tr>
<tr>
<td>T5</td>
<td>supplier Canada</td>
<td>find the information of the suppliers in ‘Canada’</td>
</tr>
<tr>
<td>T6</td>
<td>supplier part cornflower</td>
<td>find the information of the suppliers who supply part ‘cornflower’</td>
</tr>
<tr>
<td>T7</td>
<td>customer name lineitem ship rail</td>
<td>find the name of the customers who order lineitems by ‘ship’ and ‘rail’</td>
</tr>
</tbody>
</table>

**Table 4: Queries for the ACM DL database**

<table>
<thead>
<tr>
<th>#</th>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>author “Tok Wang Ling”</td>
<td>find the information of the author “Tok Wang Ling”</td>
</tr>
<tr>
<td>A2</td>
<td>paper “keyword search”</td>
<td>find the information of the papers on “keyword search”</td>
</tr>
<tr>
<td>A3</td>
<td>author Jagadish affiliation</td>
<td>find the affiliations of the author ‘Jagadish’</td>
</tr>
<tr>
<td>A4</td>
<td>publisher code proceeding SIGMOD</td>
<td>find the code of the publisher for the ‘SIGMOD’ proceedings</td>
</tr>
<tr>
<td>A5</td>
<td>paper title author Hristidis</td>
<td>find the title of the papers authored by ‘Hristidis’</td>
</tr>
<tr>
<td>A6</td>
<td>editor name proceeding EDBT ICDT</td>
<td>find the names of the common editors for the proceedings ‘EDBT’ and ‘ICDT’</td>
</tr>
<tr>
<td>A7</td>
<td>author name paper “query processing” “data integration”</td>
<td>find the name of the authors with both papers “query processing” and “data integration”</td>
</tr>
</tbody>
</table>
Table 5: SQL statements generated for the TPCH database

<table>
<thead>
<tr>
<th>#</th>
<th>ExpressQ</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>select R1.date</td>
<td>select R1.date</td>
</tr>
<tr>
<td></td>
<td>from orders R1, customer R2, nation R3 where R1.custkey=R2.custkey and</td>
<td>from orders R1, customer R2, nation R3 where R1.custkey=R2.custkey</td>
</tr>
<tr>
<td></td>
<td>R2.nationkey=R3.nationkey and match(R1.priority) against ('high' in</td>
<td>and R2.nationkey=R3.nationkey and match(R1.priority) against ('high'</td>
</tr>
<tr>
<td></td>
<td>boolean mode) and match(R3.name) against ('China' in boolean mode);</td>
<td>in boolean mode) and match(R3.name) against ('China' in boolean mode);</td>
</tr>
<tr>
<td>75</td>
<td>select R2.supkey, R2.name, R2.address... from nation R1, supplier R2</td>
<td>select R2.supkey, R2.name, R2.address... from nation R1, supplier R2</td>
</tr>
<tr>
<td></td>
<td>where R2.supkey=R3.supkey and R2.partkey=R11.partkey and R1.name</td>
<td>where R2.supkey=R3.supkey and R2.partkey=R11.partkey and R1.name</td>
</tr>
<tr>
<td></td>
<td>against ('Canadá' in boolean mode);</td>
<td>against ('Canadá' in boolean mode);</td>
</tr>
<tr>
<td>76</td>
<td>select R3.supkey, R3.name, R3.address... from part R1, part-supplier</td>
<td>select R3.supkey, R3.name, R3.address... from part R1, part-supplier</td>
</tr>
<tr>
<td></td>
<td>R2, supplier R3 where R2.supkey=R3.supkey and R2.partkey=R11.partkey</td>
<td>R2, supplier R3 where R2.supkey=R3.supkey and R2.partkey=R11.partkey</td>
</tr>
<tr>
<td></td>
<td>and R1.name against ('cornflower' in boolean mode);</td>
<td>and R1.name against ('cornflower' in boolean mode);</td>
</tr>
<tr>
<td>77</td>
<td>select R1.name from customer R1, orders R2, lineitem R3, orders R4,</td>
<td>select R1.name from customer R1, orders R2, lineitem R3, orders R4,</td>
</tr>
<tr>
<td></td>
<td>lineitem R5 where R2.custkey=R1.custkey and R3.orderkey=R2.orderkey</td>
<td>lineitem R5 where R2.custkey=R1.custkey and R3.orderkey=R2.orderkey</td>
</tr>
<tr>
<td></td>
<td>and R4.orderkey=R4.orderkey and R3.shipmode and R5.shipmode</td>
<td>and R4.orderkey=R4.orderkey and R3.shipmode and R5.shipmode</td>
</tr>
<tr>
<td></td>
<td>against ('rail' in boolean mode);</td>
<td>against ('rail' in boolean mode);</td>
</tr>
</tbody>
</table>

Table 6: SQL statements generated for the ACMDL database

<table>
<thead>
<tr>
<th>#</th>
<th>ExpressQ</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>select R1.author_id, R1.name from author R1 where</td>
<td>select R1.author_id, R1.name from author R1 where</td>
</tr>
<tr>
<td></td>
<td>match(R1.type) against ('Tok Wang Ling' in boolean mode);</td>
<td>match(R1.type) against ('Tok Wang Ling' in boolean mode);</td>
</tr>
<tr>
<td>A2</td>
<td>select R1.paper_id, ..., R1.title... from paper R1 where match(R1.title)</td>
<td>select R1.paper_id, ..., R1.title... from paper R1 where match(R1.title)</td>
</tr>
<tr>
<td></td>
<td>against ('keyword search' in boolean mode);</td>
<td>against ('keyword search' in boolean mode);</td>
</tr>
<tr>
<td>A3</td>
<td>select R11.affiliation from author R1, author_aff_history R11</td>
<td>select R11.affiliation from author R1, author_aff_history R11</td>
</tr>
<tr>
<td></td>
<td>where R11.author_id=R1.author_id and match(R1.name) against ('Jagadish'</td>
<td>where R11.author_id=R1.author_id and match(R1.name) against ('Jagadish'</td>
</tr>
<tr>
<td></td>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>A4</td>
<td>select R3.code from proceeding R1, publisher R2 where</td>
<td>select R3.code from proceeding R1, publisher R2 where</td>
</tr>
<tr>
<td></td>
<td>R1.publisher_id=R2.publisher_id and match(R1.acronym) against</td>
<td>R1.publisher_id=R2.publisher_id and match(R1.acronym) against</td>
</tr>
<tr>
<td></td>
<td>('SIGMOD' in boolean mode);</td>
<td>('SIGMOD' in boolean mode);</td>
</tr>
<tr>
<td>A5</td>
<td>select R3.title from author R1, author_paper R2, paper R3</td>
<td>select R3.title from author R1, author_paper R2, paper R3</td>
</tr>
<tr>
<td></td>
<td>where R2.publisher_id=R3.publisher_id and</td>
<td>where R2.publisher_id=R3.publisher_id and</td>
</tr>
<tr>
<td></td>
<td>R2.author_id=R1.author_id and match(R1.name) against</td>
<td>R2.author_id=R1.author_id and match(R1.name) against</td>
</tr>
<tr>
<td></td>
<td>('Hristidis' in boolean mode);</td>
<td>('Hristidis' in boolean mode);</td>
</tr>
<tr>
<td>A6</td>
<td>select R1.name from editor R1, edit_proceeding R2, proceeding R3,</td>
<td>select R1.name from editor R1, edit_proceeding R2, proceeding R3,</td>
</tr>
<tr>
<td></td>
<td>edit_proceeding R4, proceeding R5 where</td>
<td>proceeding R3 where</td>
</tr>
<tr>
<td></td>
<td>R2.proc_code=R3.proc_code and R2.editor_id=R1.editor_id and</td>
<td>R2.proc_code=R3.proc_code and R2.editor_id=R1.editor_id and</td>
</tr>
<tr>
<td></td>
<td>R4.proc_code=R5.proc_code and R4.editor_id=R1.editor_id and</td>
<td>R4.proc_code=R5.proc_code and R4.editor_id=R1.editor_id and</td>
</tr>
<tr>
<td></td>
<td>match(R3.acronym) against ('EDBT' in boolean mode) and match(R5.acronym)</td>
<td>match(R3.acronym) against ('EDBT' in boolean mode) and match(R5.acronym)</td>
</tr>
<tr>
<td></td>
<td>against ('ICDT' in boolean mode);</td>
<td>against ('ICDT' in boolean mode);</td>
</tr>
<tr>
<td>A7</td>
<td>select R1.name from author R1, author_paper R2, paper R3, author_paper</td>
<td>select R1.name from author R1, author_paper R2, paper R3, author_paper</td>
</tr>
<tr>
<td></td>
<td>R4, paper R5 where R2.publisher_id=R3.publisher_id and</td>
<td>R4, paper R5 where R2.publisher_id=R3.publisher_id and</td>
</tr>
<tr>
<td></td>
<td>R2.author_id=R1.author_id and</td>
<td>R2.author_id=R1.author_id and</td>
</tr>
<tr>
<td></td>
<td>match(R3.title) against ('query processing' in boolean mode)</td>
<td>match(R3.title) against ('query processing' in boolean mode)</td>
</tr>
<tr>
<td></td>
<td>and match(R5.title) against ('data integration' in boolean mode);</td>
<td>and match(R5.title) against ('data integration' in boolean mode);</td>
</tr>
</tbody>
</table>

The data graph approach models the database as an undirected graph where each node represents a tuple and each edge represents a foreign key-key reference. BANKS [8] proposes a backward expansion search to find the Steiner trees that contain all the keywords, and evaluates the relevance of a Steiner tree based on its root and leaf nodes. [9] improves the efficiency of BANKS by using a bidirectional expansion search to reduce the search space. [3] employs a dynamic programming technique by identifying top-k minimal group Steiner trees. CI-Rank [15] considers the importance of nodes, as well as the cohesiveness of the tree structure.

To improve the usability of keyword search, [10] summarizes the results of keyword queries so that users can refine their search based on these summaries. [14] finds co-occurring terms of keywords to provide users relevant information to refine the answers. [4] generates a list of object summaries. None of these works consider the context of keywords and search target in the queries.

7. CONCLUSION

In this paper, we have examined the problem of enhancing the expressive power and evaluation of relational keyword queries. This is achieved by extending the keyword queries in two aspects. First, we consider keywords that match meta-data e.g., names of relations and attributes, and utilize them to provide the context of subsequent keywords in the query. Second, we use the ORM schema graph to enrich the semantics of the keywords, and identify sets of keywords that refer to the same object/relationship in the database, in order to infer the search target of the query. The proposed approach is implemented in a prototype system called ExpressQ that analyzes keyword query to identify user’s search intention and generates SQL statements to retrieve relevant information. Experimental results demonstrate that the effectiveness of ExpressQ.

8. REFERENCES