A Semantic Approach to Keyword Search over Relational Databases

Zhong Zeng, Zhifeng Bao, Mong Li Lee, and Tok Wang Ling

School of Computing, National University of Singapore
{zengzh, baozhife, leeml, lingtw}@comp.nus.edu.sg

Abstract. Research in relational keyword search has been focused on the efficient computation of results as well as strategies to rank and output the most relevant ones. However, the challenge to retrieve the intended results remains. Existing relational keyword search techniques suffer from the problem of returning overwhelming number of results, many of which may not be useful. In this work, we adopt a semantic approach to relational keyword search via an *Object-Relationship-Mixed data graph*. This graph is constructed based on database schema constraints to capture the semantics of objects and relationships in the data. Each node in the ORM data graph represents either an object, or a relationship, or both. We design an algorithm that utilizes the ORM data graph to process keyword queries. Experiment results show our approach returns more informative results compared to existing methods, and is efficient.

Keywords: Keyword Search, Relational Databases, Semantic Approach

1 Introduction

The success of web search engines has made keyword search the most popular search paradigm for ordinary users. Given the rapid growth of structured data repositories, the ability to support keyword search over such repositories enables users to pose keyword queries easily without the need to have full knowledge of the database schemas or structured query languages.

Research in relational keyword search has been focused on the efficiency of computation of results from multiple tuples [9, 12, 8, 6, 4] as well as ranking strategies to improve the quality of results [16, 17, 20]. The works in [16, 17, 12, 14, 13] examine the effectiveness of relational keyword queries. However, the retrieval of informative results for relational keyword search remains a challenge.

We observe that when a user issues a keyword query, each keyword is usually directed at some object of interest, or relationship along with the associated objects. This motivates us to design a semantic approach to increase the effectiveness of relational keyword queries. In particular, we will construct an *Object-Relationship-Mixed data graph* (ORM data graph) of the database which consists of three types of nodes, namely object node, relationship node and mixed type node. In contrast to the traditional data graph where each node corresponds to a

Student

tupleid	sid	name	sex
s1	U054	John Williams	Male
s2	A005	Edward Martin	Male
s3	A021	Mary Smith	Female

Qualification

tupleid	lid	degree	major	university
q1	StnL	PhD	CS	University of Wisconsin-Madison
q2	StnL	Master	EE	University of Toronto
q3	JntK	PhD	CS	National University of Singapore

Course

tupleid	cid	title	credit	lid
c1	CS421	Database Design	4.0	Stnl
c2	CS526	Information Retrieval	3.0	JntK
c3	CS203	Java Programming	3.5	JntK

Enrol

tupleid	sid	cid	grade
e1	U054	CS203	A
e2	U054	CS421	B
e3	A005	CS421	A
e4	A005	CS526	B
e5	A021	CS526	A

Lecturer

tupleid	lid	Name	office	email
l1	StnL	Steven Lee	COM2 215	slee@yyy.zz
l2	JntK	Janet Kate	COM1 316	jkate@mmm.nn

Fig. 1. Example relational database

tuple, a node in an ORM data graph may correspond to a list of tuples. We will show that the ORM data graph can facilitate the retrieval of useful and relevant information for relational keyword queries.

The contributions of our work are summarized as follows:

1. We identify limitations of existing approaches for relational keyword search as they do not consider the objects and relationships represented by data instances.
2. We design an ORM data graph of the database to capture the semantics of objects and relationships in the data. Based on this graph, we develop an algorithm to process queries depending on the types of nodes that the keywords match.
3. We conduct comprehensive experiments to demonstrate the effectiveness and efficiency of processing keyword queries using our ORM data graph approach over existing approaches.

2 Motivating Example

Let us consider the sample relational database in Figure 1. The relations *Student* and *Lecturer* store the core information about students and lecturers respectively. The qualifications of a lecturer are captured in the relation *Qualification* since each lecturer could have more than one qualification. The relation *Course* stores both the core information about courses and the many-to-one relationship between courses and lecturers. This reflects the application constraints that each course is associated with only one lecturer. The relation *Enrol* captures the many-to-many relationship between students and courses. The schema of this database can be modeled as a schema graph [10, 9] where each node represents

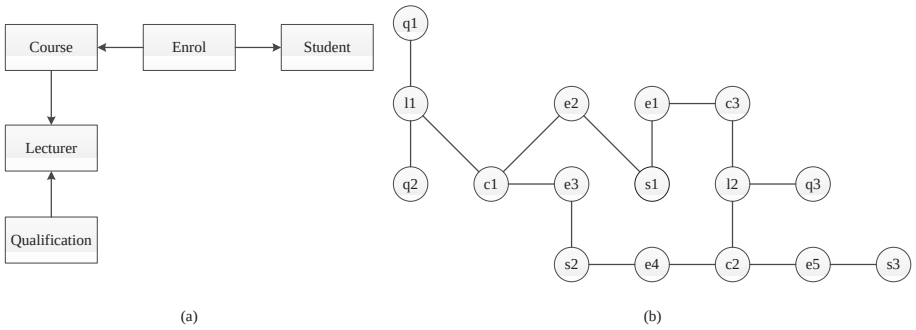


Fig. 2. The schema graph and the data graph for the example database in Fig. 1

a relation and each directed edge represents a foreign key-key constraint. Figure 2(a) shows the schema graph obtained. Correspondingly, the data instances of the database can be modeled as a data graph [11, 8] where each node represents a tuple and each undirected edge represents a foreign key-key reference. As Figure 2(b) shows, the data graph is undirected as direction is not a major concern for query processing.

Example 1. Suppose a user issues the keyword query "*Steven Lee*" to retrieve all the information about him. Existing works will only return his *lid*, *name*, *office* and *email*, that is, the first tuple in the *Lecturer* relation. However, information about the degrees and associated majors and universities of "Steven Lee", which are stored in the *Qualification* relation, is not retrieved.

Example 2. Suppose a user wants to know the information of the course where a student "Mary" obtains grade "A", and issues the keyword query "*Mary A*". Existing works will retrieve the third tuple in the *Student* relation and the last tuple in the *Enrol*, as the two query keywords occur in these tuples respectively and there exists a foreign key reference between them. This result is not informative as details such as the course id, title and credit is not retrieved.

In addition, relational keyword queries are also inherently ambiguous. Thus, existing works would consider all the possible interpretations of a keyword query and retrieve the corresponding information from the relational database. Consequently, a huge number of results are returned although many of them are probably not useful to the user.

Example 3. Suppose a user issues a keyword query "*John Mary*". Figure 3 shows two sample results obtained by existing works. Intuitively, the first result shown in Figure 3(a) indicates that student "John Williams" is enrolled in the course "Java Programming" and student "Mary Smith" is enrolled in the course "Information Retrieval". Both courses are taught by the same lecturer "Janet Kate". The second result shown in Figure 3(b) means that student "Edward Martin" is enrolled in the same course "Database Design" as the student "John Williams";

"Edward Martin" is also enrolled in the same course "Information Retrieval" as another student "Mary Smith". We observe that the first result is most likely more useful to the user.

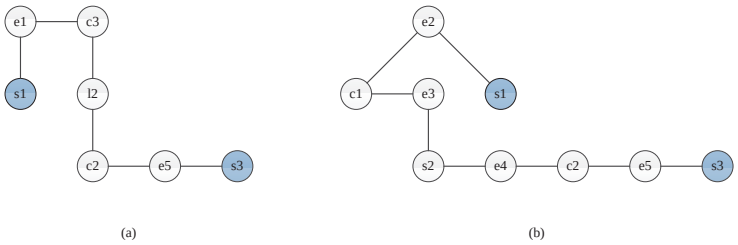


Fig. 3. Sample answers for query "*John Mary*" in Example 3

The above examples illustrate problems that arise when we do not consider the underlying semantics in the relational database. This motivates us to develop a semantic approach to answer keyword queries.

A relational database is typically designed using some conceptual model such as the ER diagram to capture the semantics in the real world in terms of entity and relationship types. Figure 4(a) shows the ER diagram for the relational database in Figure 1. The process of semantics discovery essentially reverses the translation from ER model to relational schema. There has been much research on discovering semantics from relational schema such as [19]. Here, we build upon these works and utilize primary key constraint and foreign key constraint to classify the relations in a relational schema.

Similar to [19], we have 4 types of relations, namely, *object relation*, *relationship relation*, *mixed relation* and *component relation*. Intuitively, an object (relationship) relation contains the majority of the attributes of an entity (relationship) type. A relation is a mixed relation if it encompasses both an entity type and a relationship type. A mixed relation occurs when there is a one-to-many relationship, e.g., the "Teach" relationship type in the ER diagram in Figure 4(a). A component relation represents a component part or the multi-valued attribute of an entity or relationship type, e.g, qualification is a multivalued attribute of Lecturer and is translated to the *Qualification* relation.

Based on the type of each relation in the database, we construct an *Object-Relationship-Mixed data graph* (ORM data graph) that consists of three types of nodes, namely object node (rectangle), relationship node (diamond) and mixed type node (hexagon). Each node typically includes some tuple in the corresponding relation. Tuples in the component relations are attached to their corresponding object (relationship or mixed) type nodes. In contrast to the traditional data graph where each node corresponds to a tuple in the database, a node in an ORM data graph may correspond to a list of tuples. Two nodes are connected via an edge if there exists a foreign key-key reference between tuples in the nodes.

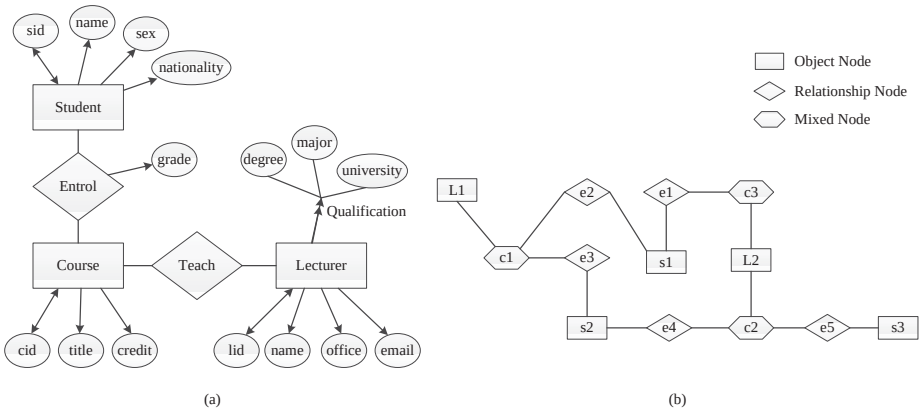


Fig. 4. The ER diagram and the ORM data graph for the database in Fig.1

Figure 4(b) shows the ORM data graph[1] for the database in Figure 1. Node $L1$ is an object node that includes the tuple $l1$ in the object relation *Lecturer*. In addition, both tuple $q1$ and $q2$ in the component relation *Qualification* are associated with $l1$ and attached to $L1$. Thus, node $L1$ corresponds to a list of tuples $\{l1, q1, q2\}$. Node $c1$ is a mixed type node that corresponds to the tuple $c1$. There is an edge between nodes $L1$ and $c1$ because of the foreign key-key reference between the tuples $l1$ and $c1$.

We say that a query keyword matches a node in the ORM data graph if the keyword occurs in some tuple in the node. We devise two ways to process the query depending on the types of nodes that the keywords match. Consider the query "*Steven Lee*" in Example 1. Since both the keywords match object node $L1$ in the graph, we will retrieve all the information about the lecturer object "Steven Lee", including his qualifications. In other words, the three tuples $\{l1, q1, q2\}$ are returned as the answer.

For the keyword query "*John A*" in Example 2, the keyword "A" matches relationship node $e1$, while the keyword "John" matches object node $s1$ that is directly connected with $e1$. Thus, we will return the information about the relationship along with all the participating objects, including the student object "John" and the course object "Java Programming".

Finally, for Example 3, since the keywords "John" and "Mary" both match object nodes $s1$ and $s3$ in the ORM data graph, we will return a tree of nodes $\{s1 - e1 - c3 - L2 - c2 - e5 - s3\}$. Note that the node $L2$ corresponds to a lecturer object which is common to both $s1$ and $s3$.

[1] We use uppercase to label a node in the ORM data graph if it corresponds to a list of tuples, and lowercase if a node corresponds to a single tuple.

3 Proposed Approach

A keyword query Q is defined as $Q = \{k_1, k_2, ..., k_n\}$, where $k_i, i \in [1, n]$ denotes a keyword. Each keyword is a term that specifies the user's search interest. Existing works consider that a keyword matches a tuple if the keyword is contained in the values of this tuple, and the goal of keyword query processing is to return the minimal number of tuples that collectively contain all the query keywords. While this approach retrieves all the tuples that contain the query keywords, the user will be overwhelmed with the large number of results. We observe that when a user issues a keyword query, it is usually directed at some object, or a relationship along with the associated objects.

Our proposed approach first utilizes key and foreign key constraints to classify the relations in a relational schema into $object, relationship, mixed$ or *component* relations as follows:

1. A relation R is an *object relation* if there exists some relation R' that references R, and R does not reference other relations.
2. A relation R is a *relationship relation* if the primary key of R comprises more than one disjoint foreign key.
3. A relation R is a *mixed relation* if (a) there exists two relations R' and R'' such that R' references R and R references R'', and (b) the primary key of R does not contain more than one disjoint foreign key.
4. A relation R_1 is a *component relation* if (a) no relation references R_1, (b) the primary key of R_1 does not contain more than one disjoint foreign key, and (c) the inclusion dependency $R_1[A_1] \subseteq R[K]$ holds, where A_1 is a subset of attributes in R_1 and K is a candidate key of R.

A mixed relation contains information about both objects and relationships. We will use semantic dependencies to differentiate the objects and relationships when processing the keyword query. For example, the mixed relation *Emp(eno, ename, birthdate, address, dno, joindate)* contains information about the employee and the date s/he joins a department. In this case, *joindate* is an attribute of the relationship between employee and department. This constraint can be captured by the *semantic dependency* [15] $\{eno, dno\} \xrightarrow{Sem} joindate$, indicating that the value of *joindate* will be updated when $\{eno, dno\}$ is updated. We consider the attributes *eno*, *ename*, *birthdate*, *address* the object part of the relation, and the attribute *joindate* the relationship part.

For each object relation R, we cluster the tuples in R and its component relations. Similarly, for each relationship (mixed) relation R, we also cluster the tuples in R and its component relations. Based on the clusters obtained, we construct an undirected *Object-Relationship-Mixed data graph (ORM data graph)* $G(V, E)$. Each node $v \in V$ corresponds to a cluster of tuples C. We have $v.label = C$, $v.tids$ is the list of tuple ids in cluster C, and $v.type \in \{object, relationship, mixed\}$ depending on whether tuples in the cluster are from an object relation, a relationship relation, or a mixed relation. An edge $e(u, v) \in E$ indicates a foreign key-key reference between tuples in u and v.

A query keyword k *matches* a node u in the ORM data graph G if k occurs in some tuple in u. Let $Obj(k)$ and $Rel(k)$ be the sets of object and relationship type nodes that match k respectively. Based on the semantic dependencies, if a keyword k matches the object part of a mixed type node u, then we add u to $Obj(k)$. Otherwise, if k matches the relationship part of u, we add u to $Rel(k)$.

If $Obj(k) \neq \emptyset$, that is, k matches some object type nodes and/or the object part of mixed type nodes, then we retrieve all the tuples associated with the nodes in $Obj(k)$. If $Rel(k) \neq \emptyset$, that is, k matches some relationship type nodes and/or the relationship part of mixed type nodes, then we retrieve the tuples associated with each node $v \in Rel(k)$, as well as the tuples in the object and mixed type nodes that are directly connected to v in the ORM data graph. The intuition is that when a keyword refers to some relationship, the user is either interested in the information about the relationship, or the information about the objects of the relationship. Thus, we will retrieve the information about the relationship, as well as the information about all the participating objects of this relationship.

After obtaining the tuples that match each keyword, we need to combine the results from different keyword matches. Given a keyword query Q, we have two main cases.

Case 1. $\exists k \in Q, Rel(k) \neq \emptyset$

For this case, the keywords in the query match either object, relationship or mixed type nodes. For each such k, we check each node $v \in Rel(k)$ whether the rest of the keywords match object and mixed type nodes that are directly connected to v in the ORM data graph. If so, then we return this result. We can view the result as a tree where the relationship node v is the root and the object and mixed type nodes are the leaves.

Recall the keyword query "*Mary A*" in Example 2. The keyword "Mary" matches object node $s3$, while keyword "A" matches relationship nodes $\{e1, e3, e5\}$ in the ORM data graph in Figure 4(b). Hence, we have $Obj(\text{“}Mary\text{”}) = \{s3\}$ and $Rel(\text{“}A\text{”}) = \{e1, e3, e5\}$. Since $s3$ and $e5$ are directly connected in the ORM data graph, we return the tuples associated with $e5$, as well as the tuples in $s3$ and $c2$ as the result. Intuitively, this result means that the student "Mary Smith" obtained grade "A" for the course "Information Retrieval".

Case 2. $\forall k \in Q, Rel(k) = \emptyset$.

For this case, all the keywords match only object and mixed type nodes and we generate all the possible combinations of nodes from $Obj(k_1)$, $Obj(k_2)$,..., $Obj(k_{|Q|})$. For each node combination, we apply the standard graph traversal method to find the set of Steiner trees that connects these nodes. For each Steiner tree, we will check whether there exists a node v such that the path from each keyword matched node to v comprises of nodes from different relations in the schema. If so, we output this tree as a query result.

Recall our query "*John Mary*" in Example 3. Our algorithm will output the Steiner tree in Figure 3(a) but not in Figure 3(b) as the the former contains node $l2$ such that both paths $l2 - c3 - e1 - s1$ and $l2 - c2 - e5 - s3$ comprises of nodes from different relations, while the latter does not contain a such node.

Algorithm 1 (ORMSearch) shows the details. The input is a keyword query Q, ORM data graph G and parameter K. We initialize two priority queues PQ_o and PQ_r to store candidate result trees ordered by the number of nodes in the tree (Line 1). For each keyword k, we find the set of nodes in G that match k. We partition the nodes into two sets: $Obj(k)$ and $Rel(k)$. For each node $v \in Rel(k)$, we create a tree $T_{v,k}$ that consists of v and its neighboring nodes in the ORM data graph G. $T_{v,k}$ is associated with the keyword k to denote that k matches some node in the tree. If the tree already exists in queue PQ_r, we update the associated keywords of the tree by adding k. Otherwise, we insert the tree into queue PQ_r (Lines 4-6). Similarly, for each node $v \in Obj(k)$, we create a tree $T_{v,k}$ that consists of a root node v. If the tree exists in the queue PQ_o, we update the associated keywords of the tree by adding k. Otherwise, we insert the tree into PQ_o (Lines 8-10).

Next, we combine the results from different keyword matches. Lines 11-24 process the trees in PQ_r (Case 1). We initialize a variable *count*, and iteratively dequeue a tree T from PQ_r. We obtain the set of keywords W associated with T (Lines 13-14). For each query keyword k that does not appear in W, we check whether k matches some node in T. If so, we put k into W (Lines 15-17). Finally, if every query keyword matches some node in T, we will put T into *Result* and increase *count* (Lines 19-20). This process terminates when *count* equals to K, i.e., we have already found K number of results (Lines 21-22).

Lines 25-48 process the trees in PQ_o (Case 2). For each iteration, we dequeue a tree T from PQ_o. Let v be the root of T and W be the set of keywords associated with T. If every keyword matches some node in T, we put T into *Result* and increase *count* (Lines 27-30). If *count* equals to K, we exit the loop. Otherwise, we traverse the ORM data graph G to find the set of Steiner trees that associate all the query keywords. We use *tree grow* and *tree merge* strategies in [6] to expand Steiner trees associated with partial keywords to those associated with all query keywords.

For each node u that is directly connected to v in G, we create a new tree T' from T by adding u as the new root of T' (Lines 36-37). This process is called tree growing. We first check whether there exists a node y in the new tree T' such that every path from y to a leaf node consists of nodes from distinct relations. If so, then we check if PQ_o already contains a tree with root u and associated with keywords W. If yes, then we update PQ_o with the smaller tree, else we insert T' into PQ_o (Lines 38-40).

For each set of keywords W' such that W' is a subset of Q and W' has no common keywords with W, we check whether we have found a tree T' with root v and associated with keywords W' in previous iterations. If T' exists, we create a new tree T'' by merging T' and T. T'' is rooted at v and associated with keywords $W \cup W'$ (Lines 42-44). This process is called tree merging. After that, we check whether there exists a node y in the new tree T'' such that every path from y to a leaf node consists of nodes from distinct relations. If so, we update queue PQ_o with T'' (Lines 45-47). Finally, we return the top-K trees in *Result* (Line 49).

Algorithm 1: ORMSearch

input : keyword query $Q = \{k_1, ...k_n\}$, K, ORM data graph G,
output: result set *Result*

```
1  Result ← ∅; PQ_o ← ∅; PQ_r ← ∅;
2  for i = 1 to n do
3      Let Rel(k_i) be the set of relationship/mixed nodes in G that match k_i;
4      foreach node v ∈ Rel(k_i) do
5          create a tree T_{v,k_i} that consists of v and its neighboring nodes in G;
6          update PQ_r with T_{v,k_i};
7      Let Obj(k_i) be the set of object/mixed nodes in G that match k_i;
8      foreach node v ∈ Obj(k_i) do
9          create a tree T_{v,k_i} with root v;
10         update PQ_o with T_{v,k_i};
11 count = 0;
12 while PQ_r ≠ ∅ do
13     T = dequeue PQ_r;
14     Let W be the set of keywords that are associated with T;
15     foreach keyword k ∈ Q − W do
16         if k matches some node in T then
17             W = W ∪ {k};
18
19     if W == Q then
20         add T to Result; count++;
21         if count = K then
22             break;
23
24
25 count = 0;
26 while PQ_o ≠ ∅ do
27     T = dequeue PQ_o;
28     let v be the root of T and W be the set of keywords associated with T;
29     if W == Q then
30         add T to Result; count++;
31         if count = K then
32             break;
33
34     else
35         //Tree growing process
36         foreach node u that is directly connected to v in G do
37             create a new tree T' from T by adding u as the new root;
38             if ∃ node y ∈ T' s.t. every path from y to a leaf node consists of nodes from
               distinct relations then
39                 update PQ_o with T';
40
41         //Tree merging process
42         foreach set of keywords W' ⊂ Q s.t. W ∩ W' = ∅ do
43             if ∃ tree T' s.t. v is the root of T' and W' is the set of keywords associated
               with T' then
44                 merge T' with T to form T'';
45                 if ∃ node y ∈ T'' s.t. every path from y to a leaf node consists of nodes
                   from distinct relations then
46                     update PQ_o with T'';
47
48
49     return the top-K trees in Result;
```

4 Performance Study

In this section, we evaluate the effectiveness and the efficiency of our semantic approach. We adopt the traditional data graph approach as the baseline because our approach also performs search directly on the data. We use the well-established Steiner tree and the state of the art DPBF [6] implementation. Since the ranking of results is orthogonal to this work, we will output query results ordered by the number of nodes in the result.

Two real world datasets are used in our experiments: the Internet Movie Database (IMDB)[2] and the DBLP data (DBLP) [5]. For the IMDB dataset, we convert a subset of its raw text files into 8 relations. The total number of tuples is 2,168,813. For the DBLP dataset, the schema consists of 6 relations and the data consists of 881,867 tuples. Table 1 shows the keyword queries used.

The experiments were performed on a Intel(R) Core(TM) i7-2600 CPU 3.40GHz with 8GB of RAM. All the algorithms were implemented using JDK 1.7 and JDBC. The inverted indices are built using MySQL v5.5 fulltext index.

Table 1. Queries used in experiments

DBLP		**IMDB**	
DQ1	Keyword Search	IQ1	Christopher Nolan
DQ2	SIGMOD Jeffrey	IQ2	Woody Allen
DQ3	Jim Gray Alexander	IQ3	Johnny Depp Jack
DQ4	PageRank Computing research	IQ4	Jamie Paul Jones
DQ5	Query optimization Yannis Papakonstantinou	IQ5	Steven Horse drama
DQ6	Conceptual design relational database	IQ6	Peter Parker comedy
DQ7	Ling Tok Wang Object Relationship	IQ7	American Comedy Page Ellen

4.1 Effectiveness Experiments

We first compare the query results returned by ORMSearch and DPBF. Table 2 shows a sample of the results for the IMDB dataset. We observe that the results obtained by DPBF is not as informative as those obtained by ORMSearch. $Q1$ is a query about the movie "Inception". ORMSearch retrieves all the information about this movie but DPBF does not retrieve the genre information. $Q2$ is a query about the movie "Intouchables" and the character name "Nouvel". Compared to ORMSearch, DPBF provides no information about the actor who played the character "Nouvel" in "Intouchables". For $Q3$, ORMSearch retrieves movies where Jeremy plays the character Cruise, as well as movies where both Jeremy and Cruise act in. In contrast, DPBF retrieves 174 results, many of which are not useful. The results for DBLP is similar and we omit it due to space limit.

Figure 5(a) and 5(b) show the number of results retrieved for each query on both datasets when we set the result size to 7 and 9 respectively. We see that DPBF typically produces more results than ORMSearch. Moreover, when the result size increases from 7 to 9, the number of results returned by DPBF increases significantly.

[2] http://www.imdb.com/interfaces

Table 2. Results of queries for IMDB dataset

Query	ORMSearch	DPBF
Q1: Inception	1. Movie: Inception 2010 Action Adventure Mystery	1. Movie: Inception 2010
Q2: Intouchables Nouvel	1. Movie: Intouchables 2011 Comedy Drama Character: Nouvel auxiliaire 2 Actor: Cayrey, Jean Fran	1. Movie: Intouchables 2011 Character: Nouvel auxiliaire
Q3: Cruise Jeremy	1. Movie: Car Jack 2008 Action Adventure Crime Character: Cruise Actor: Anus, Jeremy 2. Movie: August 2008 Drama Actor: Bobb, Jeremy Actor: Cruise, Tom 3. Movie: Mission: Impossible-Ghost Protocol 2011 Actor: Renner, Jeremy Actor: Cruise, Tom	1. Character: Cruise Actor: Anus, Jeremy 2. Character: Cruise Guy Actor: Palko, Jeremy 3. Movie: Knocked Down 2008 Character: Irving Cruise Character: Cab Driver Actor: Aimone, Jeremy ...

To further verify that our approach can achieve a better search quality than the base line, we carried out a survey where we show the queries to 6 users and collect the possible search intentions (at most 5) of each query. For each particular search intention, we generate an SQL statement and take the SQL execution results. Results of all the SQLs form the ground truth for us to determine the precision of the results obtained by ORMSearch and DPBF. Figure 5(c) and 5(d) show that ORMSearch is able to achieve a much higher precision than DPBF for most of the queries. Both ORMSearch and DPBF has a precision of 1.0 for query $DQ2$ as it has only one possible search intention. The precision of DPBF is low for query $DQ6$ as it is inherently ambiguous with a large number of possible search intentions. However, ORMSearch is still able to improve the precision by retrieving more informative and useful results.

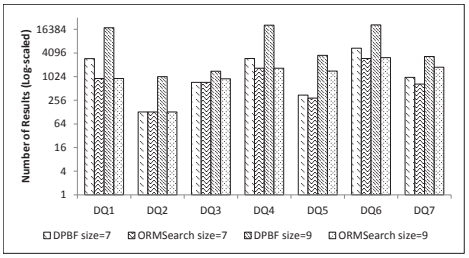
4.2 Efficiency Experiments

Finally, we compare the execution time of the two approaches. Figure 6(a) and Figure 6(b) show the results. As we can see, ORMSearch is about 2∼3 times faster than DPBF, especially when the maximum result size is 9.

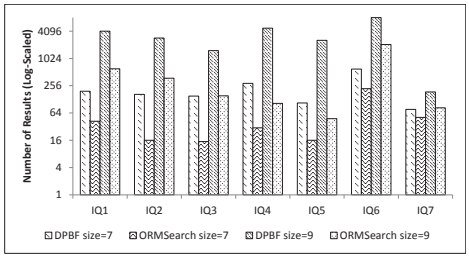
Besides the queries in Table 1, we also randomly generate 40 queries for each dataset whose lengths vary from 2 to 5 keywords, with 10 queries for each query size. For each query, we test the execution time of ORMSearch and DPBF for retrieving first output 10, 50 and 200 results respectively. The average execution time on cold cache is recorded in Figure 6(c) and Figure 6(d). On average, ORMSearch is about 6∼8 times faster than DPBF. Further, the time required by ORMSearch to retrieve 10, 50, and 200 results are almost the same, while the execution time for DPBF increases. The gap between ORMSearch and DPBF widens as the number of keywords increases. This is because our ORM data graph has fewer nodes compared to the traditional data graph.

5 Related Work

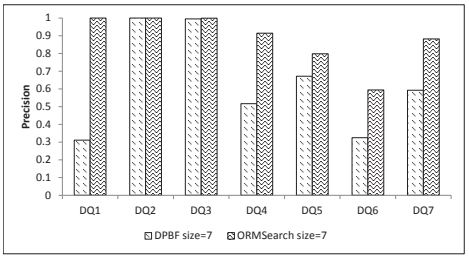
Relational keyword search can be broadly classified into two categories: (a) schema graph approach and (b) data graph approach. In the schema graph



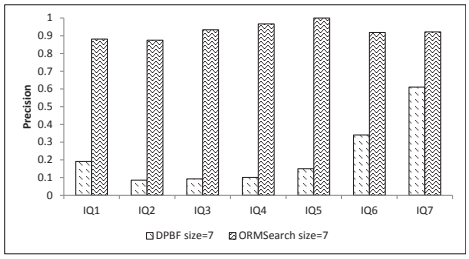
(a) Number of Results (DBLP)



(b) Number of Results (IMDB)

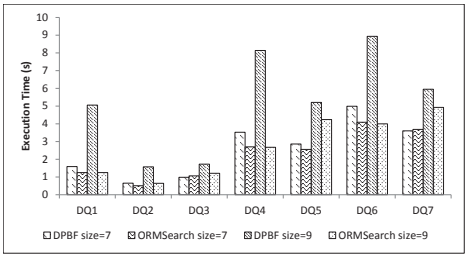


(c) Precision (DBLP)

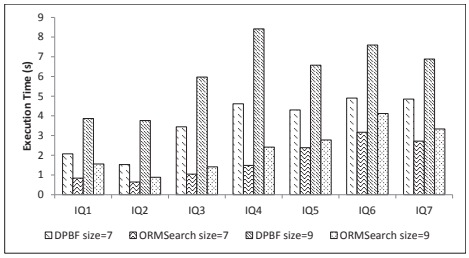


(d) Precision (IMDB)

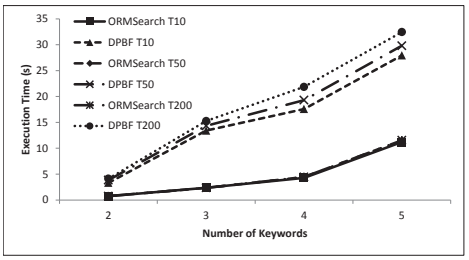
Fig. 5. Effectiveness of ORMSearch vs DPBF



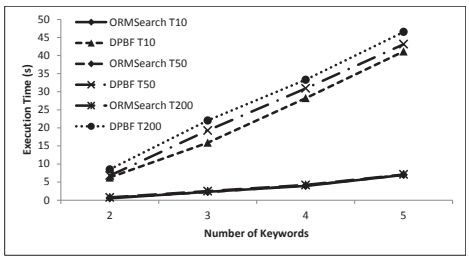
(a) Sample Queries on DBLP data



(b) Sample Queries on IMDB data



(c) Random Queries on DBLP data



(d) Random Queries on IMDB data

Fig. 6. Efficiency of ORMSearch vs DPBF

approach, the database schema is modeled as an undirected graph where each node represents a relation and each edge represents a foreign key-key constraint. To answer a keyword query, DBXploer [1] proposes join trees such that each leaf relation covers some keyword with tuples containing that keyword, and all the leaf relations collectively cover all keywords of the query. Thus, by joining all the relations in a join tree, the output tuples will contain all keywords specified in the query. Discover [10] tries to find join trees without any redundant leaf relations that cover the same keywords as others. In addition, it allows duplicate relations in join trees because a relation can join itself via a many-to-many relationship with other relations. [9] is a variant of Discover, which relaxes the requirement that the output tuples should contain all the keywords in a query.

In the data graph approach, the relational database is modeled as a graph where each node corresponds to a tuple and each edge corresponds to a foreign key-key reference. Banks [11] proposes a backward expansion search to find the common node which connects a keyword node for each keyword via the shortest path. An answer to a query is an Steiner tree with the common node as the root and each keyword node as a leaf. [12] improves the performance of [11] by using a bidirectional expansion technique to reduce the size of the search space. [6] proposes dynamic programming to identify the top-k minimal group Steiner tree in time exponential in the number of keywords. [8] proposes a bidirectional index to improve query performance. [14] studies how to calculate the radius of a graph and defines an answer to a keyword query as a subgraph which has a user-specified radius and is relevant to each keyword. These works are focused on the efficiency of relational keyword search and do not consider the quality of the search results.

To improve the search quality, [9] adopts an IR-style ranking strategy to evaluate the relevance of an answer. [16] further improves the search effectiveness by normalizing the ranking formulae in [9]. Spark [17] considers all the tuples in the answer as a visual document to avoid the side effect of overly rewarding contributions of the same keyword. Banks [11] evaluates the relevance of an answer tree by investigating its root and each leaf nodes. [20] measures the importance of not only the root and leaf nodes, but also the intermediate nodes. However, none of these works addresses the problems raised in Section 2.

Objectrank [2] considers the database as a set of objects. However, it is not clear how database objects are detected. [18] proposes to infer the basic, independent semantic unit of information in a database, but does not consider the relationship between semantic units. [7] defines a query result as an object summary (OS) about a particular data subject. The relationship between data subjects is not considered. [3] proposed a statistical way to find the promising search target(s) for an XML keyword query.

6 Conclusion

In this paper, we have examined the limitations of existing relational keyword search methods, and proposed a semantic approach to address the problems of

retrieving informative and useful results. This is achieved by constructing an ORM data graph to capture the semantics of objects and relationships in the database. Compared to the traditional data graph, each node in the ORM data graph is associated with a type and may correspond to a list of tuples. Based on the ORM data graph, we devised an efficient algorithm to process keyword queries. Experiments on two real world datasets verify the effectiveness and efficiency of our approach.

References

1. S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.
2. A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: authority-based keyword search in databases. In *VLDB*, 2004.
3. Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, 2009.
4. S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *SIGMOD*, 2011.
5. R. Cyganiak. D2RQ benchemarking. `http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2rq/benchmarks/`.
6. B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.
7. G. J. Fakas, Z. Cai, and N. Mamoulis. Size-l object summaries for relational keyword search. In *VLDB*, 2011.
8. H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.
9. V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.
10. V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB*, 2002.
11. A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.
12. V. Kacholia, S. Pandit, and S. Chakrabarti. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
13. M. Kargar and A. An. Keyword search in graphs: finding r-cliques. In *VLDB*, 2011.
14. G. Li, B. C. Ooi, and J. Feng. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.
15. T. W. Ling and M. L. Lee. Relational to entity-relationship schema translation using semantic and inclusion dependencies. *Integr. Comput.-Aided Eng.*, 1995.
16. F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, 2006.
17. Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.
18. A. Nandi and H. V. Jagadish. Qunits: queried units for database search. In *CIDR*, 2009.
19. L.-L. Yan and T. W. Ling. Translating relational schema with constraints into OODB schema. In *Database Semantics Conference*, 1933.
20. X. Yu and H. Shi. CI-Rank: Ranking keyword search results based on collective importance. In *ICDE*, 2012.