**Fuzzy Query over Ontologies based on Relational Databases**

Qiang Tong  
School of Software, Northeastern University  
Shenyang, China  
tongq@swc.neu.edu.cn

Fu Zhang, Jingwei Cheng  
School of Computer Science and Engineering, Northeastern University  
Shenyang, China

**Abstract**—With the increasing use of ontologies in the Semantic Web and various applications, it is critical to supply efficient methods to query the information in ontologies. Especially fuzzy query, which is closer to the nature language, has been paid more and more attention in recently. This paper presents a method of fuzzy queries for ontologies based on relational databases. We first present a storage model of ontologies in relational databases. On this basis, we further propose an approach for fuzzy queries over ontologies by transforming the fuzzy queries over ontologies into the SQL queries over the stored relational databases, and the query algorithms are also proposed. Finally, the case studies show that the approach is feasible.

**Keywords**—Ontology; Relational Database; Fuzzy Query

I. INTRODUCTION

Ontology, as a standard (W3C recommendation) for representing knowledge in the Semantic Web, has become a fundamental and critical component for developing applications in different real world scenarios [2]. In general terms, ontology, which is an explicit formal specification of a shared domain conceptualization, can be used to describe the objects, properties, concepts, and their relationships existing in the domain. A number of ontology definition languages, such as RDF(S), SHOE, OIL, and OWL, have been developed over the past years [13]. The most widely used among them is the W3C recommended standard language OWL (Web Ontology Language) [22] and its successor OWL 2 [9]. Today, there are lots of researches on ontologies, which are used widely in the context of the Semantic Web and other applications.

With the increasing use of ontologies in the Semantic Web and various applications, it is critical to supply efficient methods to query the information in the ontologies. That is, when clients access the application ontology, the ontology can be exploited to access the data and to answer queries taking into account the knowledge that is implicit in the ontology. To this end, there are today many proposals for answering queries over ontologies, and until now the literature on querying of ontologies has been flourishing (see [3], [14], [37] for survey). In particular, the database research community has successfully developed a wide theory corpus and a mature and efficient technology to deal with large and persistent amounts of information. In this case, some mature database techniques may be employed to query ontologies (see [40] for review).

However, when clients access the application ontology, their query requests are often imprecise and vague, and this is closer to the nature language. For example, one client wants to query the following information from the ontology LUBM [33] (LUBM is a widely used university domain ontology):

Q1: querying all of young professors?  
Q2: querying all of young students that choose the courses taught by old professors.

Therefore, one problem is considered that has arisen from practical needs: namely, fuzzy querying of ontologies. As we have known, the existing methods (see [3], [14], [37] for survey) only focused on answering classical queries over ontologies and did not consider the vague queries over ontologies. Please refer to the related work of this paper in detail. To this end, this paper presents a method of fuzzy queries for ontologies based on relational databases. In brief, the paper makes the following main contributions:

- After analyzing the semantic information characteristics of ontologies, we present a storage model of ontologies in relational databases;
- On this basis, we further propose an approach for fuzzy queries over ontologies by transforming the fuzzy queries over ontologies into the SQL queries over the stored relational databases, and the query algorithms are also proposed.
- Finally, we make some case studies, which show that the approach is feasible and efficient.

The remainder of this paper is organized: Section II introduces the related work. Section III presents a storage model of ontologies in relational databases. Section IV further presents a method of fuzzy query for ontology. Section V carries out experiments. Section VI shows conclusions.

II. RELATED WORK AND COMPARISON

As surveyed in our recent review work [40], in general, the query answering techniques for ontologies based on databases includes the conjunctive queries over ontologies based on databases [1], [7], [19], [34]; the SPARQL-to-SQL query over ontologies [8], [12], [16], [26], [28], [35]; and the query over ontologies with some special database techniques [20], [23], [27], [36]. The recent review work [40] summarizes the query...
answering techniques for ontologies based on databases. Please refer to [40], [29], [17] for a review on query answering techniques for ontologies.

With the emergence of imprecise and vague information in many applications, up to now lots of fuzzy extensions of ontologies (i.e., fuzzy ontologies) have been presented in the literature. Please refer to our recent survey [41] for a more comprehensive review on for ontologies. Accordingly, regarding the requirement of querying fuzzy ontologies, some efforts have been made, including query over fuzzy Description Logic ontologies [10], [21], [24], [32]; query fuzzy ontologies based on fuzzy relational databases [4], [5]; and query fuzzy ontologies with some special approaches [6], [11], [18], [39].

First we should be noted that our work in this paper focuses on fuzzy queries of ontologies based on relational databases. Here, both of the ontologies and relational databases are “classical” ones, but the query requests are “imprecise and vague”. Therefore, the differences between our work in this paper from the existing techniques are obvious:

- First, the main differences between our work in this paper from the existing techniques above are the differences of query requests. Our work can deal with the uses fuzzy query requests (i.e., the query conditions contain the imprecise and vague concepts such as young, old, and high), but the existing techniques cannot deal with such fuzzy query requests.

- Second, the main differences between our work in this paper from the existing techniques above are the differences of query sources. Both of the ontology and relational database are classical ones in our work. When users access the ontology, their query requests are imprecise and vague. But the existing techniques mainly focused on the fuzzy ontologies (i.e., the ontologies are imprecise ones in their work). As mentioned in [30], [41], there are many differences between classical ontologies and fuzzy ontologies, including the representation syntax, semantics, reasoning, query, and other aspects. The existing techniques cannot deal with the fuzzy queries over the classical ontologies based on the classical relational databases.

Based on the observations above, although there are some researches on query for ontologies, to our best knowledge, there is not a complete and detailed report on fuzzy query over ontologies based on relational databases. In the real-world applications, many users often want to retrieve the imprecise and vague information from the existing application ontologies. To this end, in this paper we aim at proposing an approach for fuzzy query over ontologies.

### III. STORAGE OF ONTOLOGIES IN RELATIONAL DATABASES

In this section, we propose a model for storing ontologies in relational databases. The storage model is the basic of fuzzy query over ontologies based on relational databases as will be presented in Section V.

#### A. Storage Model of Ontologies in Relational Databases

When we store ontologies in relational databases, we should let the storage reflects the characteristics of ontologies. In the following, we give a storage model of ontologies in relational databases (i.e., a series of rules).

1) **Storage of basic information of ontologies**

The basic information of an ontology includes classes, properties, and individuals. All of them are called resources. A resource is uniquely identified in an ontology with its URI (Uniform Resource Identifier), which is composed of a namespace and a name. Therefore, for querying easily and saving storage space, we use Namespace table, Resource table, Class table, Individual table, ObjectProperty table, and DatatypeProperty table to store the basic information of an ontology as shown in Rule 1-6.

**Rule 1: Namespace (nsID <INT, PK>, namespace <VARCHAR>)**

Remarks. The Namespace table is used to store namespaces in an ontology for saving storage space.

- The nsID field with the type of INT is the PRIMARY KEY (PK);
- The namespace field with the type of VARCHAR is the namespace used in the ontology.

**Rule 2: Resource (resID <VARCHAR, PK>, nsID <FK>, localname <VARCHAR>, type <VARCHAR, CHECK (type IN ('class', 'objectproperty', 'datatypeproperty', 'individual'))>)**

Remarks. The Resource table is used to store all of resources in an ontology.

- The resID field with the type of VARCHAR is the PRIMARY KEY (PK);
- The nsID field is the FOREIGN KEY (FK) that refers to the Namespace table;
- The localname field with the type of VARCHAR is the local URI of a resource;
- The type field with the type of VARCHAR is the category of a resource, where the CHECK constraint denotes that the value of the field type can be ‘class’, ‘objectproperty’, ‘datatypeproperty’, or ‘individual’, and no others.

**Rule 3: Class (classID <PK, FK>, className <VARCHAR>)**

Remarks. The Class table is used to store all of classes in an ontology.

- The classID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to the Resource table (since that a class is also a resource);
- The className field with the type of VARCHAR is the name of the class.

**Rule 4: Individual (indID <PK, FK>, indName <VARCHAR>)**

Remarks. The Individual table is used to store all of individuals in an ontology.

- The indID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to the Resource table (since that an individual is also a resource);
• The indName field with the type of VARCHAR is the name of the individual.

Rule 5: ObjectProperty (oproID <PK, FK>, oproName <VARCHAR>, domain <FK>, range <FK>)
Remarks. The ObjectProperty table is used to store all of object properties in an ontology.
• The oproID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to the Resource table (since that an object property is also a resource);
• The oproName field with the type of VARCHAR is the name of the object property;
• The domain and range fields, which are the FOREIGN KEYS (FK) that refer to the Class table, describe the domain and range of an object property.

Rule 6: DatatypeProperty (dtproID <PK, FK>, dtproName <VARCHAR>, domain <FK>)
Remarks. The DatatypeProperty table is used to store all of datatype properties in an ontology.
• The dtproID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to the Resource table (since that a datatype property is also a resource);
• The dtproName field with the type of VARCHAR is the name of the datatype property;
• The domain field, which is the FOREIGN KEY (FK) that refers to the Class table, describes the domain of a datatype property. Since the range of a datatype property is a basic data type value (e.g., INTEGER or FLOAT), there is no need to create the range field of a datatype property.

2) Storage of basic information of ontologies
Besides the basic information of ontologies, some properties in the ontologies may have the additional characters (e.g., Functional, Transitive, and Symmetric) and restrictions (e.g., someValuesFrom, allValuesFrom, and cardinality). Rules 7-10 create several tables to store characteristics and restrictions of properties.

Remarks. The PropertyCharacter table is used to store the characters of a property in an ontology.
• The proID and ct fields together are the PRIMARY KEY (PK) of the table;
• The proID field is also the FOREIGN KEY (FK) that refers to the ObjectProperty and DatatypeProperty tables;
• The ct field with the type of VARCHAR is the characters of the property, and the CHECK constraint denotes that the value of the field ct can be ‘Functional’, ‘InverseFunctional’, ‘Transitive’, ‘Symmetric’, and no others.

Rule 8: InverseOfPropertyCharacter (proID1 <PK, FK>, proID2 <PK, FK>, ct <VARCHAR, CHECK (ct IN (‘InverseOf’))>)
Remarks. The InverseOfPropertyCharacter table is used to store the inverse relationship of two properties in the ontology.
• The proID1 and proID2 fields together are the PRIMARY KEY (PK) of the table. Moreover, both of them are also the FOREIGN KEYS (FK) that refer to the ObjectProperty table;
• The ct field with the type of VARCHAR is the character of the property, and the CHECK constraint denotes that the value of the field ct can be ‘InverseOf’ only, and no others.

Rule 9: ObjectPropertyRestriction (classID <PK, FK>, oproID <PK, FK>, valueRT <VARCHAR, CHECK (valueRT IN (‘someValuesFrom’, ‘allValuesFrom’))>, value1 <FK>, cardRT <VARCHAR, CHECK (cardRT IN (‘minCardinality’, ‘maxCardinality’))>, value2 <INT>)
Remarks. The ObjectPropertyRestriction table is used to store the restrictions of an object property in the ontology.
• The classID and oproID fields together are the PRIMARY KEY (PK) of the table. Moreover, both of them are also the FOREIGN KEYS (FK) that refer to the Class and ObjectProperty tables, respectively;
• The valueRT field with the type of VARCHAR is the restriction of the property, where the CHECK constraint denotes that the value of the field valueRT can be ‘someValuesFrom’, ‘allValuesFrom’, or ‘NULL’ and no others; and the value1 field, which is the value of the restriction valueRT, refers to the Class table;
• The cardRT field with the type of VARCHAR is the cardinality of the property, where the CHECK constraint denotes that the value of the field cardRT can be ‘minCardinality’, ‘maxCardinality’, or ‘NULL’, and no others; and the value2 with the type of INT is the value of the cardinality cardRT.

Rule 10: DatatypePropertyRestriction (classID <PK, FK>, dtproID <PK, FK>, valueDT <VARCHAR, CHECK (valueDT IN (‘someValuesFrom’, ‘allValuesFrom’))>, value1 <FK>, cardDT <VARCHAR, CHECK (cardDT IN (‘minCardinality’, ‘maxCardinality’))>, value2 <INT>)
Remarks. The DatatypePropertyRestriction table is used to store the restrictions of a datatype property in the ontology.
• The classID and dtproID fields together are the PRIMARY KEY (PK) of the table. Moreover, both of them are also the FOREIGN KEYS (FK) that refer to the Class and DatatypeProperty tables, respectively;
• The valueDT field with the type of VARCHAR is the restriction of the property, where the CHECK constraint denotes that the value of the field valueDT can be ‘someValuesFrom’, ‘allValuesFrom’, or ‘NULL’ and no others; and the value1 field with one of the BasicTypes in relational databases (e.g., VARCHAR and INT) is the value of the restriction valueDT;
• The cardDT field with the type of VARCHAR is the cardinality of the property, where the CHECK constraint denotes that the value of the field cardDT can be ‘minCardinality’, ‘maxCardinality’, or ‘NULL’, and no others; and the value2 with the type of INT is the value of the cardinality cardDT.

3) Storage of class and property axioms
The axioms in an ontology describe relationships among classes and properties at the conceptual level. In general, the
axioms mainly include the class constructor axioms (i.e., unionOf, intersectionOf, complementOf, disjointWith, and oneOf), the class hierarchy axioms (i.e., subclassOf and equivalentclass), and the property hierarchy axioms (i.e., subpropertyOf and equivalentproperty). Rules 11-13 create several tables to store these axioms.

Rule 11: Union (classID \(<PK, FK>,\) classMember \(<PK, FK>,\)) , Intersection (classID \(<PK, FK>,\) classMember \(<PK, FK>,\)) , Complement (classID \(<PK, FK>,\) classMember \(<PK, FK>,\)) , DisjointWith (classID \(<PK, FK>,\) classMember \(<PK, FK>,\)) , Enumeration (classID \(<PK, FK>,\) individualMember \(<PK, FK>,\))

Remarks. The tables are used to store the class hierarchy axioms mentioned above. The structures of the tables are similar.
- The PRIMARY KEY (PK) of each table is composed of the fields classID and classMember/individualMember;
- Both of the fields classID and classMember are also the FOREIGN KEYs (FK) that refer to the Class table;
- The field individualMember is also the FOREIGN KEY (FK) that refers to the Individual table.

Rule 12: SubClass (classID1 \(<PK, FK>,\) classID2 \(<PK, FK>,\)) , EquivalentClass (classID1 \(<PK, FK>,\) classID2 \(<PK, FK>,\))

Remarks. The tables are used to store the class hierarchy axioms mentioned above. The structures of the tables are similar.
- The classID1 and classID2 fields together are the PRIMARY KEY (PK) of each table;
- Both of the fields classID1 and classID2 are also the FOREIGN KEYs (FK) that refer to the Class table.

Rule 13: SubProperty ([proID1, proID2 \(<PK, FK>,\)]) , EquivalentProperty ([proID1, proID2 \(<PK, FK>,\)])

Remarks. The tables are used to store the property hierarchy axioms mentioned above respectively. The structures of the tables are similar.
- The proID1 and proID2 fields together are the PRIMARY KEY (PK) of each table;
- Both of the fields proID1 and proID2 are also the FOREIGN KEYs (FK) that refer to the ObjectProperty or DatatypeProperty tables.

4) Storage of individual assertions

The assertions in an ontology describe the information of individuals at the instance level. In general, the assertions mainly include the concept assertions describing all individual instances of classes, and the property assertions describing the relationships of individual instances. Rules 14-16 create several tables to store these assertions.

Rule 14: classID (indID \(<PK, FK>,\))

Remarks. The table is used to store the concept assertions in an ontology, i.e., the fact that an individual indID belongs to a class classID. The indID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to Individual table.

Rule 15: oproID (indID1 \(<PK, FK>,\) indID2 \(<PK, FK>,\))

Remarks. The table is used to store a relationship oproID between two individuals indID1 and indID2. The indID1 and indID2 fields together are the PRIMARY KEY (PK) and also the FOREIGN KEYs (FK) that refer to the field indID in Individual table.

Rule 16: dtproID (indID \(<PK, FK>,\) val \(<BasicType>,\))

Remarks. The table is used to store the datatype property assertions in an ontology, i.e., the value of the datatype property dtproID of an individual indID is “val”.
- The indID field is the PRIMARY KEY (PK) and also the FOREIGN KEY (FK) that refers to Individual table;
- The val field with one of the BasicTypes in the relational databases (e.g., VARCHAR, INT, and FLOAT) is the value.

IV. Fuzzy Query over Ontologies based on Relational Databases

On the basis of the storage model, in this section we further propose fuzzy query approach over ontologies based on relational databases. The basic idea of the approach is transforming the fuzzy query over ontologies into the SQL query over the stored relational databases. First, we present a query preprocessing algorithm for removing redundancies of the fuzzy query requests; Second, we further propose query transformation algorithms for fuzzy queries over ontologies by transforming the simple and complex fuzzy queries over ontologies into the SQL queries over the stored relational databases.

A. Query Preprocessing Algorithm

Before introducing the subsequent algorithms, we first give several notations and definitions concerning fuzzy query over ontology that will be used throughout the article.

As usual, the symbol “\(?x\)" is used to represent significant variable, i.e., the variable which stores the final result of a query. The symbol “\(!x\)" represents non-significant variable. For simplicity, the symbol \(a\) is used to represent either \(?x\), \(!x\) or a constant. On the basis of the ideas in [15], [25], a fuzzy query over an ontology can be formally defined as follows.

Definition 1 (Fuzzy Query FCQ): A fuzzy query FCQ is the conjunction of a classical query CQ and an imprecise query FQ, i.e., FCQ = CQ AND FQ, where:
- \(CQ = \{q_1, ..., q_n\}\) is the conjunction of query atoms \(q_i\), \(i \in \{1, ..., m\}\). A classical query atom \(q_i\) is either a concept assertion \(C(\alpha)\) or a property assertion \(R(\alpha, \alpha);\)
- \(FQ = \{f_1, ..., f_n\}\) is the conjunction of query atoms \(f_j\) = \(p(\alpha, \alpha, \tilde{t}_j), j \in \{1, ..., n\}\). The symbol \(p\) represents a property name that the user inputs, \(f\) represents a fuzzy term (e.g., young or old), and \(t_j \in [0, 1]\) is a user-defined threshold.

The definition is illustrated by the following example. A user wants to query “all students who are young and high”, which can be formalized as FCQ = [student(?x)] AND [age(?x, young, 0.8), height(?x, tall, 0.7)]. In the fuzzy query FCQ, there are two fuzzy conditions “age is young” and “height is tall”, where young and tall are fuzzy terms with their respective fuzzy sets that represent the terms. For example, the
membership functions of the fuzzy terms young and tall are as follows:

$$\text{young}(x) = \begin{cases} 
1 & 0 < x \leq 25 \\
1 + \left(\frac{x - 25}{5}\right)^2 & x > 25 
\end{cases}$$  \hspace{1cm} (1)

$$\text{tall}(x) = \begin{cases} 
1 + \left(\frac{185 - x}{10}\right)^2 & 0 \leq x < 185 \\
1 & x \geq 185 
\end{cases}$$  \hspace{1cm} (2)

There may be duplicate query items in a fuzzy query that user inputs. For example, when a fuzzy query contains the conjunction of two concept assertion items $C_1(\alpha)$ and $C_2(\alpha)$, we should first detect the relationship between the concept $C_1$ and the concept $C_2$. If $C_1$ is a subclass of $C_2$ or $C_1$ and $C_2$ are equivalent classes, we should remove $C_2(\alpha)$ from the query. Similarly, for property assertion items $P_1(\alpha, \alpha’)$ and $P_2(\alpha, \alpha)$, we should detect whether they are equivalent or $P_1$ is a subproperty of $P_2$, and then remove one of equivalent property assertion or the parent property assertion. The algorithm of removing redundancies applies to FCQ is given below.

**Algorithm 1: Algorithm of removing redundancies in FCQ**

Input : A fuzzy query FCQ

Output : FCQ that removes all redundant query items

1. for (each $q_i \in FCQ$.i++) {
2. for (each $q_j \in FCQ$ and $j > i$, j++) {
3. if ($q_i = C(\alpha)$, $q_j = C(\alpha’)$, and $\alpha_i = \alpha_j$)
   then
5. if ($C_i = C_j$ or $C_i \subset C_j$)
6. else
   if ($C_i = C_j$ or $C_j \subset C_i$)
7. (switch $q_i$ and $q_j$, remove $q_j$);--
8. }/end if
9. else if ($q_i = R(\alpha, \alpha’)$, $q_j = R(\alpha, \alpha’)$, $\alpha_i = \alpha_j$, and $\alpha’_i = \alpha’_j$)
10. (switch $q_i$ and $q_j$, remove $q_i$);--
11. }/end if
12. }/end for $q_j$
13. }/end for $q_i$
14. return $FCQ$

After applying the algorithm 1 to an input query FCQ, the redundancies in FCQ may be deleted. Then, the output FCQ will be further handled in the following sections.

**B. Fuzzy Queries over Ontologies**

As mentioned in previous section, the basic idea of our approach is to transform a fuzzy query FCQ = $CQ$ AND $FQ$ over an ontology into a SQL query on the stored relational database. In general, two cases may occur in a fuzzy query FCQ over an ontology: one is called “simple FCQ”, i.e., the fuzzy terms only occur in fuzzy conditions that include significant variables “$2x$”; another one is called “complex FCQ”, i.e., the fuzzy terms may occur in fuzzy conditions that include significant variables “$2x$” or non-significant variables “$1x$”.

In this case, the corresponding algorithms for “simple FCQ” and “complex FCQ” are proposed in the following sections.

1) Algorithm for simple fuzzy queries over ontologies

For transforming a simple fuzzy query $FCQ = CQ$ AND $FQ$ over an ontology into a corresponding SQL query on the stored relational database, two parts in the $FCQ$ (i.e., the classic query $CQ$ and the imprecise query $FQ$) will be respectively transformed into SQL.

Regarding the transformation of classic query $CQ$ into SQL, the basic idea is: The query atoms and their relationships are transformed into SQL `select`, `from`, and `where` clauses. First, all of the query variables are added into SQL `select` clause; Second, the query atoms (i.e., concept assertions and property assertions) are added into SQL `from` clause, that is, the query atoms are transformed into SQL queries on the stored tables w.r.t. the assertions (i.e., the tables in rules 14-16); Finally, the shared variables among the query atoms are added into SQL `where` clause, and also the query atoms including the constants only are added into SQL `where` clause. In more detail, given a query atom $q_i \in CQ$, if $q_i = C(\alpha)$, then $q_i$ is transformed into SQL query on the table $C$ created following the rule 14. If $q_i = R(\alpha, \alpha’)$, then $q_i$ is transformed into SQL query on the table $R$ created following the rules 15-16. For example, if $q_i = C(0.5)$, then $q_i$ is transformed into $\text{CID}.\text{indID}$ (where $\text{CID}$ is the table corresponding to the class $C$ created following the rule 14, and $\text{indID}$ is the field in the table $\text{CID}$). If $q_i = R(\alpha, \alpha’)$, then $q_i$ is transformed into $\text{RID}.\text{indID1}$ or $\text{RID}.\text{indID2}$ (where $\text{RID}$ is the table corresponding to $R$ created following the rule 15 or 16, and $\text{indID1}$ and $\text{indID2}$ are the fields in tables $\text{RID}$).

Regarding the transformation of imprecise query $FQ$ into SQL, the basic idea is transforming the query atoms and their relationships into SQL `select`, `from`, and `where` clauses. First, all of the query variables are added into SQL `select` clause; Second, the query atoms (i.e., concept assertions and property assertions) are added into SQL `from` clause, that is, the query atoms are transformed into SQL queries on the stored tables w.r.t. the assertions in rules 14-16; Finally, the shared variables among the query atoms are added into SQL `where` clause, and also the query atoms including the constants only are added into SQL `where` clause. Moreover, according to the threshold in $FQ$, a fuzzy cut set is formed, and then an interval value can be calculated and added into the SQL `where` clause.

For example, given an imprecise query $FQ = \text{[age}(?x, \text{young}, 0.8), \text{height}(?x, \text{tall}, 0.7)]$, where the membership functions of the fuzzy terms have been given above, according to the basic idea above, two fuzzy cut sets are formed, i.e., $\text{young}_{0.8} = \{u | \text{A}(u) \geq 0.8\}$ and $\text{tall}_{0.7} = \{v | \text{A}(v) \geq 0.7\}$. Then, we can calculate two interval values $u \in (0, 27]$ and $v \in [179, \infty)$. Therefore, the imprecise query $FQ$ can be transformed into the following SQL query:

```sql
select \text{indID} from \text{ageID}, \text{heightID} //where \text{ageID} and \text{heightID} are the tables created following the rules 15-16 where \text{ageID}.\text{val} \leq 27 and \text{heightID}.\text{val} \geq 179
```

Based on the idea above, the following give the transformation algorithm from simple FCQ to SQL query.

By the algorithm above, a simple FCQ over an ontology can be transformed into a SQL query over the stored tables in relational database.
Algorithm 2: Algorithm for simple fuzzy queries over ontologies

Input: simple FQ
Output: SQL
1. Initializing three string variable: S1, S2, S3
2. for (each qj ∈ CQ, i++) {
3.   if (qj, ∈ CQ) {
4.     S1 += “CID.indID = ?”;
5.   } else if (qj = R(α, α))
6.     S2 += “RJD.indID”;
7.   else if (qj = R(α, α)) S3 += “CID.indID = ?”;
8.   } // end for
9.   if (α = “α”)
10.  S1 += “CID.indID = ?”;
11.  S2 += “RJD.indID”;
12.  S3 += “CID.indID = ?”;
13. } // end if
14.  if (α = “α”)
15.  S1 += “CID.indID = ?”;
16.  S2 += “RJD.indID”;
17.  S3 += “CID.indID = ?”;
18. } // end for
19.  if (qj, ∈ CQ, j > i) {
20. } // shared variables in CQ
21.  if (qj = C(α) ⊂ qj(R(α, α)) or qj = R(α, α, α))
22.   S1 += “CID.indID = RJD.indID1”;
23.  S2 += “CID.indID = RJD.indID2”;
24.  S3 += “CID.indID = RJD.indID3”;}
25.  // end for
26.  for (each fq = p(α, f, t, w) ∈ FQ, k++) {
27. } // shared variables in FQ
28.  S1 += “pα”;
29.  // adding the property name to the SQL from clause
30.  if (qj = C(α) and fq = p(α, f, t, w))
31.   S1 += “CID.indID = RJD.indID4”;
32.  S2 += “CID.indID = RJD.indID5”;}
33.  // end for
34.  for (fq = p(α, f, t, w) ∈ FQ, k++)
35.  // where Function is the membership functions of the fuzzy term f.
36.  S1 += “α ≤ pJD.val ≤ β”;
37. } // end for
38.  SQL = “select” + S1 + “from” + S2 + “where” + S3;
39. Return SQL.

Algorithm 3: Algorithm for complex fuzzy queries over ontologies

Input: complex FQ
Output: SQL
1. Initialize conditions:
2. FQ = CQ AND FQ
3. fexpr = false; if fexpr is used to denote whether the FCQ will be layered.
4. if (fexpr contains non-significant variables)
5. } // end if
6. if (fexpr == true)
7. Following the steps mentioned above to generate the new queries FCQ’ and subFCQ.
8. FQ = CQ AND FQ // storing the new generated outer layer FQ.
9. subFCQ = subFCQ AND subFQ // storing the new generated next layer FQ.
10. for (qj, ∈ CQ) {
11. } // end for
12. if (qj, ∈ CQ) {
13. } // end if
14. (subCQ.add(qj); or subFQ.add(qj);)
Based on the algorithm above, a complex FCQ over an ontology can be transformed into a nested SQL query over the stored tables in relational database.

V. EXPERIMENTS

In order to verify the validity of algorithms, we take the LUBM ontology [33] as test data (LUBM has widely been accepted as a standard of evaluating ontology system performance). The basic steps include: Firstly, in order to test our approach we generate three different scale datasets by means of the LUBM Ontology Generator UBA. And then we store the datasets in Oracle relational databases following the proposed storage mechanism. The following Table I shows the numbers of individuals, concept assertions, and property assertions in the three datasets.

Secondly, we performed a series of fuzzy queries following the proposed algorithms. Further, based on query response time, the validity of the method is analysed. Here, the query response time is the time of receiving and answering the queries. Regarding to the query response time, we construct a series of fuzzy queries, and the following Table II shows several typical query requests $Q_1$-$Q_4$, and their corresponding FCQ forms. In Table II, $Q_1$ and $Q_2$ are simple fuzzy queries; $Q_3$ and $Q_4$ is complex fuzzy queries; $t_i \in [0, 1]$ is a user-defined threshold.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Individuals</th>
<th>Concept Assertions</th>
<th>Property Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>85</td>
<td>110</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>500</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>1000</td>
<td>1500</td>
</tr>
</tbody>
</table>

**TABLE II. SEVERAL QUERY EXAMPLES.**

**Query** | **FCQ** |
---|---|
$Q_1$ | Check all of young professors; \([\text{Professor}(\xi_1)] \text{ AND } [\text{age}(\xi_1, \text{young}, t_1)]\); |
$Q_2$ | Query all of young students that choose the Course1 course; \([\text{Student}(\xi_2), \text{takeCourse}(\xi_2, \text{Course1})] \text{ AND } [\text{age}(\xi_2, \text{young}, t_2)]\); |
$Q_3$ | Query all of young students that choose the courses taught by professors; \([\text{Student}(\xi_3), \text{FullProfessor}(\xi_3), \text{teachOf}(\xi_3, \text{\xi}_5), \text{takeCourse}(\xi_5, \text{\xi}_3)] \text{ AND } [\text{age}(\xi_3, \text{young}, t_3)]\); |
$Q_4$ | Query all of young students that choose the courses taught by young professors. \([\text{Student}(\xi_4), \text{Professor}(\xi_4), \text{teachOf}(\xi_4, \text{\xi}_5), \text{takeCourse}(\xi_5, \text{\xi}_4)] \text{ AND } [\text{age}(\xi_4, \text{young}, t_4), \text{age}(\xi_4, \text{young}, t_4)]\); |

The following Fig. 1 shows the query results of $Q_1$-$Q_4$ (i.e., the returned numbers of individuals satisfying the query conditions) on the dataset “3”. The other results on the datasets “1” and “2” are similar and thus are omitted here.

Further, Fig. 2 shows the query response time of the queries $Q_1$-$Q_4$ on the three datasets in Table I. In Fig. 2, $d = 1, d = 2$ and $d = 3$ denote the datasets in Table I, respectively. And all of the tests show and verify the validity and feasible of the approach.

VI. CONCLUSIONS

In this paper we presented a method of fuzzy query for ontology based on relational database. After analyzing the semantic information characteristics of ontologies, we gave a storage model of ontologies in relational databases. On this basis, we further developed an approach for fuzzy query over ontologies by transforming the fuzzy query over ontologies...
into the SQL query over the stored relational databases, and the query algorithms were also proposed. Finally, on the basis of the proposed approach, we made some case studies, which show that the approach is feasible.

However, we realize that lots of fuzzy queries in some real applications are different. Therefore, it is difficult to give a unified pattern which is enough to express all of the users' requests. In our future work, we will further investigate the forms of fuzzy queries in depth, test and improve the SQL query method. Also some optimization techniques may be introduced. In addition, extending an existing database system with reasoning capabilities for supporting fuzzy queries of ontologies stored in databases is an important direction.

ACKNOWLEDGMENT

The work is supported by the National Natural Science Foundation of China (61672139), the Fundamental Research Funds for the Central Universities (N151704001), and the Natural Science Foundation of Liaoning Province (CN, 2015200348).

REFERENCES