SoftFacts: A Top-k Retrieval Engine for Ontology Mediated Access to Relational Databases

Umberto Straccia
ISTI - CNR
Pisa, ITALY
straccia@isti.cnr.it

Abstract — We outline SoftFacts, an ontology mediated top-k information retrieval system over relational databases. An ontology layer is used to define (in terms of a OWL-QL like Semantic Web language) the relevant abstract concepts and relations of the application domain, while graded facts are stored into a relational database. Queries are conjunctive queries with ranking aggregates and scoring functions and the results of a query may be ranked according to user defined scoring functions. We will illustrate SoftFacts’ architecture, the representation and the query language, sketch the reasoning algorithms of the SoftFacts system and its Prot´eg´e plug-in.

Index Terms — Knowledge-based Systems, Ontologies, Fuzzy Description Logics, Top-k retrieval.

I. INTRODUCTION

Description Logics (DLs) [2] have gained popularity due to their application in the context of the Semantic Web. DLs play a particular role as they are essentially the theoretical counterpart of state of the art Semantic Web ontology languages of the OWL family.

Clearly, in the Semantic Web context, data is typically very large and dominate the intentional level of the ontologies. Hence, while in the above mentioned contexts one could still accept reasoning that is exponential on the intentional/ontological part, it is mandatory that reasoning is polynomial in the data size, i.e. in data complexity [24]. Recently efficient management of large amounts of data and its computational complexity analysis has become a primary concern of research in DLs and in ontology reasoning systems [1], [4], [8], [14], [15].

In this paper, we describe the salient features of the SoftFacts system, whose aim is to allow an ontology mediated access to relational databases for data intensive applications. Informally, data is stored into a database and a DL is used to define the relevant abstract concepts and relations of the application domain (see Fig. 1).

Main features of the SoftFacts system are: (i) The SoftFacts ontology language belongs to the family of DLR-Lite [4]. DLR-Lite is different from usual DLs as it supports n-ary relations (n ≥ 1), whereas DLs support usual unary relations (called concepts) and binary relations (called roles) only. The language is closely related to OWL QL (a so-called profile of the web ontology language OWL 2) 3. In fact, OWL QL can be managed by SoftFacts; (ii) At the query level, we consider Datalog like conjunctive queries with ranking aggregates and scoring functions [10], [11]. The results of a query may be ranked according to some scoring function. In fact, SoftFacts supports Top-k Query Answering [12], [18], [19], [20], [21], (find top-k scored tuples satisfying query), e.g. “find cheap hotels close to the train station”, where cheap and price are an user defined function of the distance and price, respectively. (iii) SoftFacts provides also an integrated graphical interface within the well-known Semantic Web ontology editor Prot´eg´e 4 (see Fig. 2).

Related work: While there are many works addressing the top-k problem for vague queries over databases (cf. [6], [7], [9], [10], [11]), little is known for the corresponding problem in knowledge representation and reasoning (see [22] for a survey), as e.g. [12], [16], [17], [18], [19], [20], [21], [25]. We also refer the reader to [5] for a crisp, main memory, DL-Lite implementation and application, and to [13] for a fuzzy DL-Lite language in the spirit of [20] (and less expressive language than ours) but not addressing specifically the top-k retrieval problem and ranking aggregates. However, to the best of our knowledge, SoftFacts is the unique system based on a OWL-like Semantic Web language, fully supporting top-k retrieval, and interfacing with several databases.

In the following, we will illustrate SoftFacts’ architecture,
the representation and the query language, sketch the reasoning algorithms of the SoftFacts system and its Protégé plug-in.

II. THE ARCHITECTURE

The SoftFacts architecture has two basic components: the DL-based ontology component (the intentional level) and the database component (the extensional level) as shown in Fig. 1. The DL-component supports both the definition of the ontology and query answering. In particular, it provides a logical query and representation language, which is an extension of the DL-Lite logic behind OWL QL (a profile of the web ontology language OWL 2) only. Please note that SoftFacts does not consider negation of atoms, as supported in DL-Lite, as these do not play any role at query answering time, the main task supported by SoftFacts. Negated atoms play a role only at knowledge base consistency checking time. In SoftFacts, any knowledge base is consistent. Therefore, to what concerns query answering, the drop of negation is harmless.

A. The representation language

A knowledge base $K = \langle F, O, A \rangle$ consists of a facts component $F$, an Ontology component $O$ and an abstraction component $A$, which are defined as follows (for a detailed account of the semantics, see [23]).

Facts Component. $F$ is a finite set of expressions of the form

$$R(c_1, \ldots, c_n)[s],$$

where $R$ is an $n$-ary relation, every $c_i$ is a constant, and $s$ is a degree of truth (or score) in $[0, 1]$ indicating to which extent the

III. THE REPRESENTATION AND QUERY LANGUAGE

The logic SoftFacts adopts is based on a fuzzy extension of the DLR-Lite [4] DL without negation. DLR-Lite is an extension of DL-Lite (which is the logic behind OWL QL (a profile of the web ontology language OWL 2) as it supports $n$-ary relations whereas DLs support usual unary relations (called concepts) and binary relations (called roles) only. Please note that SoftFacts does not consider negation of atoms, as supported in DL-Lite, as these do not play any role at query answering time, the main task supported by SoftFacts. Negated atoms play a role only at knowledge base consistency checking time. In SoftFacts, any knowledge base is consistent. Therefore, to what concerns query answering, the drop of negation is harmless.

A. The representation language

A knowledge base $K = \langle F, O, A \rangle$ consists of a facts component $F$, an Ontology component $O$ and an abstraction component $A$, which are defined as follows (for a detailed account of the semantics, see [23]).

Facts Component. $F$ is a finite set of expressions of the form

$$R(c_1, \ldots, c_n)[s],$$

where $R$ is an $n$-ary relation, every $c_i$ is a constant, and $s$ is a degree of truth (or score) in $[0, 1]$ indicating to which extent the
tuple \(<c_1, \ldots, c_n>\) is an instance of relation \(R\). Facts are stored in a relational database. We may omit the score component and in such case the value 1 is assumed.

**Example 1** ([17]): Suppose we have Curricula Vitæ. Some basic information is stored into the Profile relation. An excerpt is shown in Fig. 3.

Note the difference to so-called fuzzy relational databases (see, e.g., [3], [26]), in which values of a column may be fuzzy (e.g., the person’s age is “young”), whereas this is not the case here.

**Ontology Component.** The ontology component is used to define the relevant abstract concepts and relations of the application domain by means of axioms. Specifically, \(O\) is a finite set of *axioms* having the form

\[ Rl_1 \cap \ldots \cap Rl_m \subseteq Rr, \]

where \(m \geq 1\), all \(R_l\) and \(R_r\) have the same arity and each \(R_l\) is a so-called left hand relation and \(R_r\) is a right hand relation (note that recursive axioms are allowed). We assume that relations occurring in \(F\) do not occur in axioms (so, we do not allow that database relation names occur in \(O\)). The intuitive semantics is that if a tuple \(e\) is instance of each relation \(R_l\) to degree \(s_i\) then \(e\) is instance of \(R_r\) to degree \(\min(s_1, \ldots, s_m)\).

As illustrative purpose, a simple ontology axiom may be of the form

\[ \text{ItalianCity} \sqsubseteq \text{EuropeanCity} \]

with informal reading “any Italian city is an European city”. Here ItalianCity and EuropeanCity are unary relations (i.e., concepts) with signature ItalianCity(id) and EuropeanCity(id), respectively. Similarly, the ontology axiom

\[ \text{ItalianCity} \sqcap \text{BigCity} \sqsubseteq \text{BigEuropeanCity} \]

has informal reading “any Italian city, which is also big, is a big European city” (here BigCity and BigEuropeanCity are again unary relations (concepts) with signature BigCity(id) and BigEuropeanCity(id), respectively.

The exact syntax of the relations appearing on the left-hand and right hand side of ontology axioms is specified below (where \(h \geq 1\)):

\[
\begin{align*}
R_l & \quad \rightarrow \quad A \mid R[i_1, \ldots, i_k] \\
R_r & \quad \rightarrow \quad A \mid R[i_1, \ldots, i_k] \\
\text{Cond} & \quad \rightarrow \quad (i < v) \mid (i \leq v) \mid (i > v) \mid (i \geq v) \\
& \quad \rightarrow \quad (i = v) \mid (i \neq v)
\end{align*}
\]

where \(A\) is an atomic concept (an unary predicate), \(R\) is an \(n\)-ary relation with \(n \geq 2\), \(1 \leq i_1, i_2, \ldots, i_k \leq n\), \(1 \leq i \leq n\) and \(v\) is a value of the appropriate type. Here \(R[i_1, \ldots, i_k]\) is the projection of the relation \(R\) on the columns \(i_1, \ldots, i_k\) (the order of the indexes matters). Hence, \(R[i_1, \ldots, i_k]\) has arity \(k\). For instance, Profile[1, 3] is the binary relation that is the projection on the first and third column of the Profile relation. Note that Profile[1, 3] is different from Profile[3, 1] (columns are inverted).

On the other hand, \(R[i_1, \ldots, i_k].(\text{Cond}_1, \ldots, \text{Cond}_k)\) further restricts the projection \(R[i_1, \ldots, i_k]\) according to the conditions specified in \(\text{Cond}_i\). For instance, \((i < v)\) specifies that the values of the \(i\)-th column have to be less or equal than the value \(v\) and similarly, for the other conditions. We assume that the comparison occurs among values with a comparable type. For instance,

\[\text{Profile}[1, 5].(\langle 5 \geq 1979\rangle)\]

coresponds to the set of tuples \((\text{profID}, \text{BirthDate})\) such that the fifth column of the relation \(\text{Profile}\), i.e. the person’s birth date, is equal or greater than 1979.

**Example 2** (Example 1 cont.): An excerpt of the domain ontology is described in Fig. 4 and partially encodes an ontology used to describe Curricula Vitæ. We assume that we have relations \(\text{hasDegree}(\text{profID, degID, marks})\), \(\text{knowsLanguage}(\text{profID, lanID})\) and an atomic concept \(\text{Degree(degID)}\). For instance, axiom (4) states that the languages known by profile \(\text{profID}\) should be languages.

**Abstraction Component.** The abstraction component (similarly to [5], [15]) is a set of “abstraction statements” that allow to connect atomic concepts and relations to physical relational tables. Essentially, this component is used as a wrapper to the underlying database and, thus, prevents that relational table names occur in the ontology. As illustrative purpose, assume that we have a relation \(\text{Jobs}\) in a database with signature \(\text{Jobs(jobID, name)}\), where the first column is of type \(\text{int}\), while the second is of type \(\text{string}\). Then, an example of abstraction statement is

\[
\text{jobName} \mapsto \text{Jobs(jobID[int], name[string])},
\]

by means of which we state that the relation \(\text{jobName}\) occurring in the ontology component, has arity two and has to be mapped into the relation \(\text{Jobs}\) occurring in the database.

Formally, let \(R\) be a relation symbol and let \(T\) be an \(m\)-ary table in the database. Let \(c_1, \ldots, c_n\) be column names of table \(T\) each of which of type \(t_i\) (\(n \leq m\)). We assume that \(R\) occurs in \(O\), while \(T\) occurs in \(F\). We also assume that the score of an instance of \(T\) is stored in the column \(c_{\text{score}}\). Then

![Fig. 3. The profile table.](image-url)

![Fig. 4. Excerpt of a CV ontology.](image-url)
an abstraction statement is of the form
\[ R \rightarrow T(c_1[t_1], \ldots, c_n[t_n])[c_{score}] \],

stating that \( R \) is an \( n \)-ary relation of the ontology component, that is mapped into the projection on columns \( c_1, \ldots, c_n \) of table \( T \). The score of the tuples of \( R \) is provided by column \( c_{score} \) of table \( T \). The score column \( c_{score} \) may be omitted and in that case the score 1 is assumed for the tuples. Finally, we assume that there is at most one abstraction statement for each abstract relational symbol \( R \).

B. The query language

Concerning queries, a query consists of a “conjunctive query”, with a scoring function to rank the answers. We first present ranking queries such as presented in [12] and then add to them ranking aggregates [10].

A ranking query [12] is of the form
\[ q(x)[s] \xrightarrow{\exists y} R_1(z_1)[s_1], \ldots, R_i(z_i)[s_i], \]
\[ \text{OrderBy}(s = f(s_1, \ldots, s_i, p_1(z_1^1), \ldots, p_k(z_k^1))), \]
\[ \text{Limit}(k) \]

where
1) \( q \) is an \( n \)-ary relation, every \( R_i \) is an \( n_i \)-ary relation. \( R_i(z_i) \) may also be of the form \((z < v), (z < v), (z > v), (z > v), (z = v), (z \neq v)\), where \( z \) is a variable, \( v \) is a value of the appropriate concrete domain;
2) \( x \) are the distinguished variables;
3) \( y \) are existentially quantified variables called the non-distinguished variables. We omit to write \( \exists y \) when \( y \) is clear from the context;
4) \( z_1, z_1^1 \) are tuples of constants or variables in \( x \) or \( y \);
5) \( s, s_1, \ldots, s_i \) are distinct variables and different from those in \( x \) and \( y \);
6) \( p_j \) is an \( n_j \)-ary fuzzy predicate assigning to each \( n_j \)-ary tuple \( c_j \) a score \( p_j(c_j) \in [0, 1] \). Such predicates are called expensive predicates in [6] as the score is not precomputed off-line, but is computed on query execution. We require that an \( n \)-ary fuzzy predicate \( p \) is safe, that is, there is not an \( m \)-ary fuzzy predicate \( p' \) such that \( m < n \) and \( p = p' \). Informally, all parameters are needed in the definition of \( p \). Note that concerning fuzzy predicates, we may use the so-called left shoulder, right shoulder, triangular and trapezoidal functions (see Fig. 5), which are well known fuzzy membership functions in fuzzy set theory;
7) \( f \) is a scoring function \( f: ([0, 1])^{l+h} \rightarrow [0, 1] \), which combines the scores of the \( l \) relations \( R_i(c_i^1) \) and the \( n \) fuzzy predicates \( p_i(c_i^1) \) into an overall score to be assigned to the rule head \( R(c) \). We assume that \( f \) is monotone, that is, for each \( v, v' \in ([0, 1])^{l+h} \) such that \( v \leq v' \), it holds \( f(v) \leq f(v') \), where \( (v_1, \ldots, v_{l+h}) \leq (v'_1, \ldots, v'_{l+h}) \) iff \( v_i \leq v'_i \) for all \( i \). We also assume that the computational cost of \( f \) and all fuzzy predicates \( p_i \) is bounded by a constant;
8) \( \text{Limit}(k) \) indicates the number of answers to retrieve and is optional. If omitted, all answers are retrieved.

We call \( q(x)[s] \) its head, \( \exists y. R_1(z_1)[s_1], \ldots, R_i(z_i)[s_i] \) its body and \( \text{OrderBy}(s = f(s_1, \ldots, s_i, p_1(z_1^1), \ldots, p_k(z_k^1))) \) the scoring atom. We also allow the scores \( s_1, s_1, \ldots, s_i \) and the scoring atom to be omitted. In this case we assume the value 1 for \( s_i \) and \( s \) instead.

The informal meaning of such a query is: if \( z_i \) is an instance of \( R_i \) to degree at least equal to \( s_i \), then \( x \) is an instance of \( q \) to degree at least or equal to \( s \), where \( s \) has been determined by the scoring atom. Example queries are:

\[
q(x) \rightarrow \text{SportsCar}(x) \\
// find sports cars
\]
\[
q(x) \rightarrow \text{SportsCar}(x), \text{HasSpeed}(x, y), (y \geq 240) \\
// find sports cars whose speed exceed 240
\]
\[
q(x)[s] \rightarrow \text{SportsCar}(x)[s_1], \text{HasPrice}(x, p), \\
\text{OrderBy}(s = 0.7 \cdot s_1 + 0.3 \cdot l_5(p; 10000, 14000)) \\
// find cheap sports cars
\]

Note how in the last query we combine the degree of being sports car with the degree of being the car “cheap”, where this latter is defined by means of the left-shoulder function \( l_5(p; 10000, 14000) \).

Example 3 (Example 2 cont.): Assume that we have the two relational tables as in Fig. 6. Assume that we have also the abstract mappings

\[
\text{hasName} \rightarrow \text{Profile(lastName|string)} \\
\text{hasDegree} \rightarrow \text{HasDegree(profID|int, degID|int)} \\
\text{hasMark} \rightarrow \text{HasMark(profID|int, mark|int)} \\
\text{hasDegreeName} \rightarrow \text{Degree(degID|int, name|string)} \\
\]

A query searching for CV’s with a degree with mark between 100 (minimum) and 110 (maximum) can be expressed as

\[
q(id, \text{name, degree, mark})[s] \leftarrow \text{CV(id)}, \text{hasName(id, name)}, \text{hasDegree(id, y)}, \text{hasDegreeName(y, degree)}, \text{hasMark(id, mark)}, \\
\text{OrderBy}(s = rs(mark; 100, 110))
\]

Then, we may have the results

\[
\begin{array}{llll}
\text{id} & \text{name} & \text{degree} & \text{mark} \\
\hline
1 & 100 & 100 & 100 \\
2 & 100 & 100 & 100 \\
3 & 100 & 100 & 100 \\
\end{array}
\]
As next, we extend the query language by allowing so-called ranking aggregates to occur in a query [10]. Essentially, ranking aggregates apply usual SQL aggregate functions such as SUM, AVG, MAX, MIN to the scores of group of tuples, which are answers to a query. For instance, by referring to Example 3, a person (e.g., Gadducci) may held more than one degree and we would like to rank a CV with more degrees better than one with just one (e.g. we may would like to sum-up the scores of all degrees of Gadducci).

Example 4 (Example 3 cont.): Assume that we additionally would like to sum-up the scores of the degrees of each person and find the top-k ranked tuples. Then, such a query may be expressed as

\[
q(id, name)[s] \leftarrow \text{CV(id), hasName(id, name), hasMark(id, mark), GroupedBy(id, name), OrderBy(s = \text{SUM}[rs(mark; 100, 110)]), Limit(k)}
\]

Intuitively, for the above query, we ask to group all tuples according to (id, name) and then for each group to sum-up the scores. That is, if \( g = \{t_1, \ldots, t_n\} \) is a group of tuples with same id and name, where each tuple has score \( s_i \) computed as \( rs(\text{mark}; 100, 110) \) then the score \( s_g \) of the group \( g \) is \( \sum_i s_i \). A group \( g \) is ranked then according to its score \( s_g \) and the top-k ranked groups are returned.

Formally, let \( @ \) be ranking aggregate function with \( @ \in \{\text{SUM, AVG, MAX, MIN}\} \) then a query with ranking aggregates is of the form

\[
q(x)[s] \leftarrow \exists y \ R_1(s_1), \ldots, R_l(s_l), \text{GroupedBy}(w), \text{OrderBy}(s = @ [ f(s_1), \ldots, s_l, p_1(z'_1), \ldots, p_m(z'_m) ] ), \text{Limit}(k)
\]

where additionally \( w \) is a list of variables according to which we want to group the tuples and \( @ \) is the aggregate function according to which to compute the score of the group. GroupBy(\( w \)) is called the grouping atom and we assume that \( w \) are variables in \( x \) or \( y \) such that each variable in \( x \) occurs in \( w \).

Finally, given a knowledge base \( K \), and a query \( q \), the top-k retrieval problem is the problem to retrieve \( k \) tuples \((c, s)\) that instantiate the query relation \( q \) with maximal scores (if \( k \) such tuples exist), and rank them in decreasing order relative to the score \( s \).

C. Query answering

It is well known in the database community that naive approaches to the top-k retrieval problem (e.g. retrieval all records, sort them and cut-out the top-k ones) fail for large size databases. This is true as well for top-k retrieval in logical languages, as shown [23]. So, specific algorithms have to be developed. Specifically, from a query answering point of view, SoftFacts extends the reasoning method [4] to the fuzzy case and is a generalisation of the one described in [4], [18], [20]. Informally, given a query \( q \)

1) by considering \( \mathcal{O} \), the user query \( q \) is reformulated into a set of conjunctive queries \( r(q, \mathcal{O}) \). Roughly, the basic idea is that the reformulation procedure closely resembles a top-down resolution procedure for logic programming, where each axiom is seen as a logic programming rule. For instance, given the query

\[
q(x)[s] \leftarrow A(x)[s'], \text{OrderBy}(s = f(s'))
\]

and suppose that \( \mathcal{O} \) contains the axioms

\[
B_1 \sqcap B_2 \sqsubseteq A \\
C_1 \sqcap C_2 \sqsubseteq A
\]

then we can reformulate the query into two queries

\[
q(x)[s] \leftarrow B_1(x)[s_1], B_2(x)[s_2], \text{OrderBy}(s = f(\min(s_1, s_2))) \\
q(x)[s] \leftarrow C_1(x)[s_1], C_2(x)[s_2], \text{OrderBy}(s = f(\min(s_1, s_2)))
\]

2) from the set of reformulated queries \( r(q, \mathcal{O}) \) we remove redundant queries;
3) the reformulated queries \( q' \in r(q, \mathcal{O}) \) are translated to ranked SQL queries and evaluated. The query evaluation of each ranked SQL query returns the top-k answer set for that query;
4) all the \( n = |r(q, \mathcal{O})| \) top-k answer sets are merged into the unique top-k answer set using the Disjunctive Threshold Algorithm (DTA, see e.g. [20], [23]).

We refer the reader to [23] for details on how the SoftFacts system addresses the top-k retrieval problem and the experiments we conducted with it. It can be shown that the top-k retrieval problem can be solved in LOGSPACE data complexity.

IV. SOFTFACTS PROTÉGÉ PLUG-IN

SoftFacts provides an graphical interface within the well-known Semantic Web ontology editor Protégé (see Fig. 2). First of all, we launch the Protégé editor and may load, edit, and change an OWL 2 QL ontology. Then, we move to the SoftFacts Protégé plug-in, which starts with the connection to the underlying database engine. In the database connection panel (see Fig. 7), we specify the parameters to connect to the database. There are two initialisation modalities, that are described below.

A. Automatic Database Management (ADM)

In the Automatic Database Management (ADM) modality, the system builds automatically three database tables to store the assertions (the facts) of an OWL ontology, so that the user has not to care about the structure of the database. For
Fig. 8. The database table of concept instances.

instance, Fig. 7 displays the case of the LUBM ontology. The SoftFacts Protégé plug-in starts then automatically to translate the LUBM ontology in SoftFacts syntax and creates and populates also three relational tables, with signature

\[
\begin{align*}
\text{class\_instances} & (\text{individual\_uri}\ [\text{string}], \text{class\_name}\ [\text{string}]) \\
\text{object\_properties\_instances} & (\text{individual\_uri\_1}\ [\text{string}], \text{property\_name}\ [\text{string}], \text{individual\_uri\_2}\ [\text{string}]) \\
\text{data\_properties\_instances} & (\text{property\_name}\ [\text{string}], \text{value\_string}\ [\text{string}], \text{value\_bool}\ [\text{string}], \text{value\_real}\ [\text{float}])
\end{align*}
\]

We recall that in OWL2 QL, one may assert that an individual \(a\) is instance of a concept \(A\), a pair of individuals \((a, b)\) is an instance of an object property \(T\), and that a pair \((a, v)\), where \(a\) is an individual and \(v\) is a value of some type, is an instance of a data type property \(D\). So, for instance, a fact \(\text{Person}(tom)\) is stored into the class\_instances tables as tuple \(\langle\text{tom}, \text{Person}\rangle\), a fact \(\text{hasFriend}(tom, susan)\) is stored into the object\_properties\_instances table as the triple \(\langle\text{tom}, \text{hasFriend}, \text{susan}\rangle\), while a fact \(\text{hasAge}(tom, 32)\) is stored into the data\_properties\_instances table as the tuple \(\langle\text{tom}, \text{hasAge}, \text{−−−−}, \text{−−−−}, \text{32}\rangle\). As illustrative example, Fig. 8 shows an excerpt of the class\_instances table.

In the SoftFacts Protégé plug-in, in the ADM modality, we may distinguish (see Fig. 2):

(i) In the upper panel, there are three tabs

- **Axioms**: here you see the SoftFacts axioms automatically generated from the ontology (expressions of the OWL ontology that cannot be mapped into SoftFacts are dropped). This panel is not editable to avoid inconstancies with the ontology.
- **Mapping**: this panel shows the abstraction statements created to connect the abstract concepts of the ontology to the database. This panel is not editable to avoid inconstancies with the database.
- **Macros**: this panel shows some useful macros to be used at query time to perform top-k queries over the ontology. The panel is editable.

(ii) In the middle part of the main panel, we may define SoftFact queries to be submitted to the SoftFacts system, as shown in Fig. 2.

(iii) In the lower part of the main panel, we may perform the following operations:

- We may save the ontology (or parts of it) in SoftFacts syntax as file;
- We may synchronise modifications to the ontology (left panel) by means of the Protégé editor with the SoftFacts system (UpdateAll button).

1) **Modifying the ontology**: In the following, we show how to add a modification to the ontology and making it effective to the SoftFacts reasoner.

Suppose we would like to add the axiom \(\text{Monograph} \sqsubseteq \text{Book}\) and to assert an individual Monograph0 as instance of the class Monograph (see Fig. 9). To make these changes effective to SoftFacts, we use the UpdateAll button and after that we may notice the adding of the SoftFacts axiom in the axioms panel (Fig. 10).

B. Manual Database Management (MDM)

In the Manual Database Management (MDM) case, thought for an expert user, the user needs to set-up manually the interaction with the database in a consistent manner. Specifically, once an ontology has been loaded, the system does only translate the ontology axioms into SoftFacts specific axioms. It is then up to the user to define the mapping to the underlying database tables. The individuals in the ontology are not stored into the database. After the initialisation, the plugin is similar as for the ADM case, except that we have an additional editable concretisation axioms panel (and editable mapping panel).

Here the user defines so-called concretisation axioms used to populate some specific concepts of the ontology. These axioms have on the right side abstract concepts, while have on the left side concrete concepts, i.e. concepts whose instances are stored into a database.
1) Concretisation axioms: For instance, assume that we have initialised the LUBM ontology according to MDM and let us assume that we consider the LUBM database that has been created for the ADM case, which has three tables. At first, we need to add the abstraction statements that related database tables to the concepts of the ontology:

\[
\text{isClassInstance} \rightarrow \text{classInstances}[\text{individual_uri}[\text{string}], \text{class_name}[\text{string}])
\]

\[
\text{isDataPropertyInstance} \rightarrow \text{dataPropertiesInstances}[\text{property_name}[\text{string}], \text{value_string}[\text{string}], \text{value_real}[\text{float}])
\]

\[
\text{isObjectPropertyInstance} \rightarrow \text{objectPropertiesInstances}[\text{individual_uri_1}[\text{string}], \text{property_name}[\text{string}], \text{individual_uri_2}[\text{string}])
\]

For instance, the first one says that isClassInstance is a binary relation that maps to the classInstances table of the LUBM database and whose columns are classInstances and class_name, respectively. These abstraction statements have to be added to the Mapping panel. Now, we may use concretisation axioms to populate some specific concepts starting from the database tables. Examples of concretisation axioms are the following:

\[
\text{isClassInstance}[1], (([2] = \text{AdministrativeStaff}) \subseteq \text{AdministrativeStaff})
\]

\[
\text{isClassInstance}[1], ([2] = \text{Article}) \subseteq \text{Article}
\]

\[
\text{isObjectPropertyInstance}[1, 3], ([2] = \text{advisor}) \subseteq \text{advisor}[1, 2]
\]

\[
\text{isObjectPropertyInstance}[1, 3], ([2] = \text{affiliateOf}) \subseteq \text{affiliateOf}[1, 2]
\]

\[
\text{isDataPropertyInstance}[1, 3], ([2] = \text{age}) \subseteq \text{age}[1, 2]
\]

For the sake of explanation, the first one says that instances of the concept AdministrativeStaff are the projection on the first column of the relation isClassInstance where the second column’s value is “AdministrativeStaff”. Once all axioms and abstraction statements have been created, we need to push the UpdateAll button so that the SoftFacts systems becomes aware of all the changes.

2) Modifying the database: Now, we show how we need to update the SoftFacts systems to handle the situation in which we modify the database structure. So, suppose that now we would like to associate to the publication its publication year and let us assume that we modify the database with

\[
\text{CREATE TABLE publication_year (publication_uri character varying NOT NULL, year integer NOT NULL)}
\]

and populate the table via SQL, e.g.

\[
\text{INSERT INTO publication_year VALUES ('http://www.Department0.University0.edu/FullProfessor3/Publication13', 2000)};
\]

Now we need to connect the new table to some new abstract relation using a SoftFacts abstraction statements. Hence, we define the abstract statement

\[
\text{publicationYear} \rightarrow \text{publication_year}(\text{publication_uri}[^\text{STRING}], \text{year}[^\text{INT}])
\]

that creates a new abstract relation publicationYear that is related to the relational table publication_year. Additionally, we define an axiom that states that the domain of this relation are publications:

\[
\text{publicationYear}[1] \subseteq \text{Publication}
\]

Finally, we push the UpdateAll button to update the SoftFacts system.

3) Top-k retrieval query: We are going now to use this additional data to submit a top-k query. Suppose we would like to have a ranked query, which is as follows: Find the top-10 ranked authors whose publications are scored according to the right-shoulder function (see Fig. 5) \(r(x; 2002, 2008)\) and that the final author’s score is the sum of his publication scores. In our abstract syntax, this query may be written as

\[
q(x) \leftarrow \text{publicationAuthor}(x, y), \text{publicationYear}(y, t), \text{GroupedBy}(x), \text{OrderBy}(s = \text{SUM}[rs(t; 2002, 2008)]), \text{Limit}(10)
\]

In concrete SoftFacts syntax, the query is

\[
\text{RETREIVE (?author) WHERE( (publicationAuthor ?pub ?author) (publicationYear ?pub ?year) } \text{ORDERBY( SUM(rs(2002 2008 ?year)) ) LIMIT(10) )}
\]

Submitting the query by mans of the Submit button we get the results as e.g. shown in Fig. 11.

V. CONCLUSION

The top-k retrieval problem is an important problem in logic-based languages for the Semantic Web. We have addressed this issue in the SoftFacts system, an ontology mediated top-k information retrieval system over relational databases. In SoftFacts, an ontology layer is used to define (in terms of a tractable DLR-Lite like description logic) the relevant abstract concepts and relations of the application domain, facts are stored into a relational database, accessed via an abstraction component. The results of a query may be ranked according to scoring functions. We have illustrate the architecture, the representation and the query language, sketched the reasoning algorithms of the SoftFacts system and its use within the Protégé editor. We refer the reader to [23] for technical details on the algorithms and experiments conducted with the SoftFacts system, which show good results from a scalability point of view (we tested with an ontology of more than 5000 axioms to represent CVs and 500000 CVs).
REFERENCES


