Top-k Retrieval for Ontology Mediated Access to Relational Databases

Umberto Straccia

Istituto di Scienza e Tecnologie dell’Informazione (ISTI), Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy

Abstract

We address the problem of evaluating ranked top-k queries in the context of ontology mediated access over relational databases. An ontology layer is used to define the relevant abstract concepts and relations of the application domain, while facts with associated score are stored into a relational database. Queries are conjunctive queries with ranking aggregates and scoring functions. The results of a query may be ranked according to the score and the problem is to find efficiently the top-k ranked query answers.

Keywords: Description Logics, Top-k Retrieval, Ontologies

1. Introduction

Description Logics (DLs) \cite{4} provide popular features for the representation of structured knowledge. Nowadays, DLs have gained even more 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 languages to specify ontologies, such as OWL 2.\cite{1}

It becomes also apparent that in these contexts, data are typically very large and dominate the intentional level of the ontologies. Hence, while in the above mentioned contexts one could still accept reasoning, specifically query answering, that is exponential on the intentional part, it is mandatory that reasoning and query answering is polynomial in the data size, i.e. in data complexity \cite{35}.

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 \cite{3,7,8,12,13,17,18}. This is especially important in case of ontology mediated access to relational databases for data intensive applications, where data is stored into a database and a data complexity tractable DL is used to define the relevant abstract concepts and relations of the application domain. Specifically, the adoption of such a DL allowed the
development of an efficient query answering method that can roughly described as follows: given a query and an ontology, rewrite the query using the ontology as several queries that can then be submitted as SQL queries over the database. The effectiveness of this method is based on the fact that the rewriting of the query can be computed efficiently and SQL query answering may take advantage of the performance of state of the art database engines.

In this paper, we address a relatively novel issue for DLs with a huge data repository, namely the problem of evaluating ranked top-\(k\) queries. Usually, an answer to a query is a set of tuples that satisfy a query. Each tuple does or does not satisfy the predicates in the query. However, very often the information need of a user involves so-called scoring predicates \cite{14}. For instance, a user may need: “Find cheap hotels near to the conference location”. Here, cheap and near are scoring predicates. Unlike the classical case, tuples satisfy now these predicates to a score. In the former case the score may depend, e.g., on the price, while in the latter case it may depend e.g. on the distance between the hotel location and the conference location. Therefore, a major problem we have to face with in such cases is that now an answer is a set of tuples ranked according to their score. This poses a non-negligible challenge in case we have to deal with a huge amount of instances. Indeed, virtually every tuple may satisfy a query with a non-zero score and, thus, has to be ranked. Computing all these scores, ranking them and then selecting the top-\(k\) ones is not feasible in practice for large size databases \cite{14}.

So far, there are very view works addressing specifically the top-\(k\) retrieval problem for logic-based languages in general and, specifically, in the DL context \cite{15, 24, 26, 28, 32}. With respect to these works, the contribution of this paper can be summarised as follows.

- We consider a more general DL language than the one considered in top-\(k\) retrieval, such as described in \cite{15, 24, 26, 28}. Specifically,
  - at the ontology level, while the language is closely related to OWL QL (a so-called profile of the web ontology language OWL 2 \cite{2}), we further allow scoring functions to occur in the left hand side of an axiom to combine scores and assigning the result to the axiom’s right hand side;
  - at the query level, we consider Datalog like conjunctive queries with ranking aggregates and scoring functions with scoring predicates in the same way as \cite{14}. Our query language will subsume \cite{15, 16, 24, 26, 27, 28}.
- We provide a reasoning algorithm for the specified language. Specifically,
  - we provide a generalised reformulation algorithm in the spirit of \cite{8}, that given a query and an ontology, rewrites the query as several queries;
we then provide a novel algorithm for efficient submission of these queries to the database. To this end we extend the so-called Disjunctive Threshold Algorithm provided in [24, 25, 26] to cope with ranking aggregates and main memory problems encountered in preliminary experiments.

- We conduct an experiment to assess both the effectiveness of the approach, using our SoftFacts system [1, 33], and to illustrate some issues related to a naive approach to the top-k retrieval problem.

In the following, we proceed as follows. In the next section, we provide some basics on top-k retrieval for relational databases. Then, Section 3 describes our query and representation language, Section 4 describes the query answering algorithms, while Section 5 reports the experiments. Section 6 addresses related work and Section 7 concludes and provides an outlook to future work.

2. Top-k retrieval in relational databases

We recap here some salient notions related to the top-k retrieval problem for relational databases (see, e.g. [14]).

There are essentially three query models to specify the data objects to be scored, namely (i) top-k selection query; (ii) top-k join query; and (iii) top-k aggregate query, which we describe next. For illustrative purposes, we consider the following running example taken from [30].

Example 2.1. Consider a relational database excerpt (Fig. 1) about student’s Curricula Vitae, together with some basic information about the degrees they got.

2.1. Top-k selection query

In this model, the scores are assumed to be attached to base tuples. A top-k selection query is required to report the k tuples with the highest scores according to some user-defined scoring function.
Definition 2.1 (Top-k Selection Query). Consider a relation $R$, where each tuple in $R$ has $n$ attributes. Consider $m$ scoring predicates, $p_1, \ldots, p_m$ defined on these attributes. Let $f(t) = f(p_1(t), \ldots, p_m(t))$ be the overall score of tuple $t \in R$. A top-k selection query selects the $k$ tuples in $R$ with the largest $f$ values.

A SQL template for top-k selection query is the following:

```sql
SELECT *
FROM R
WHERE selection condition
ORDER BY f(p_1, \ldots, p_m)
LIMIT k
```

For instance, consider Example 3.1. Assume we are interested in finding the top-10 most young and tall students born in Berlin, where young is defined in terms of the student’s age as some scoring predicate $p_1$, while tall is defined in terms of the student’s height by means of some scoring predicate $p_2$. The global score has been defined by a user as the linear combination $0.75 \cdot p_1(t.\text{age}) + 0.25 \cdot p_2(t.\text{height})$, giving more weight to the student’s age. This query could be written as follows:

```sql
SELECT proID, lastName
FROM Profiles
WHERE cityOfBirth = 'Berlin'
ORDER BY 0.75 \cdot p_1(\text{age}) + 0.25 \cdot p_2(\text{height})
LIMIT 10
```

Note that concerning scoring predicates, typically we may use the so-called left shoulder, right shoulder, triangular and trapezoidal functions (see Fig. 2). For instance, may define $p_1(\text{age}) = ls(\text{age}; 20, 30)$ and $p_2(\text{height}) = rs(\text{height}; 175, 200)$.

2.2. Top-k Join Query

In this model, scores are assumed to be attached to join results rather than base tuples. A top-k join query joins a set of relations based on some arbitrary join condition, assigns scores to join results based on some scoring function, and reports the top-k join results.

Definition 2.2 (Top-k Join Query). Consider a set of relations $R_1, \ldots, R_n$. A top-k join query joins $R_1, \ldots, R_n$, and returns the $k$ join results with the
largest combined scores. The combined score of each join result is computed according to some function \( f(p_1, \ldots, p_m) \), where \( p_1, \ldots, p_m \) are scoring predicates defined over the join results.

A possible SQL template for a top-k join query is

```sql
SELECT *
FROM R_1, \ldots, R_n
WHERE join_condition(R_1, \ldots, R_n)
ORDER BY f(p_1, \ldots, p_m)
LIMIT k
```

For instance, consider Example 3.1. Suppose we are looking for the top-10 young and good students, where good is defined in terms of the marks by means of a scoring predicate \( p_3 \). We may formulate such a query as

```sql
SELECT P.profID, P.lastName, P.age, D.mark
FROM Profiles P, HasDegree D
WHERE P.profID = H.profID
ORDER BY 0.5 \cdot p_1(P.age) + 0.5 \cdot p_3(D.mark)
LIMIT 10
```

### 2.3. Top-k Aggregate Query

In this model, scores are computed for tuple groups, rather than individual tuples. A top-k aggregate query reports the \( k \) groups with the largest scores. Group scores are computed using a group aggregate function such as sum, min and avg.

**Definition 2.3 (Top-k Aggregate Query).** Consider a set of grouping attributes \( G = \{g_1, \ldots, g_r\} \), and an aggregate function \( f \) that is evaluated on each group. A top-k aggregate query returns the \( k \) groups, based on \( G \), with the highest \( f \) values.

A SQL formulation for a top-k aggregate query is

```sql
SELECT g_1, \ldots, g_r, f AS score
FROM R_1, \ldots, R_n
WHERE join_condition(R_1, \ldots, R_n)
GROUP BY g_1, \ldots, g_r
ORDER BY score
LIMIT k
```

An example top-k aggregate query is to score young and good students higher the more degrees they hold. Such a query may be formulated as

```sql
SELECT P.profID, P.lastName, P.age, SUM(0.5 \cdot p_1(P.age) + 0.5 \cdot p_3(D.mark)) AS score
FROM Profiles P, HasDegree D
WHERE P.profID = H.profID
GROUP BY P.profID, P.lastName, P.age
ORDER BY score
LIMIT 10
```
which concludes this section.

3. Our ontology representation and query language

For computational reasons, the particular logic we adopt is based on a DLR-Lite \cite{8} Description Logic (DL) \cite{4} variant. DLR-Lite is an extension of DL-Lite, which is the logic behind OWL QL (a profile of the web ontology language OWL 2 \cite{2}), and supports n-ary relations whereas DLs support usual unary relations (called concepts) and binary relations (called roles) only. Please note that we do not consider negation of atoms, as supported in DL-Lite, as these do not play any role at query answering time, the main task we are interested in. Negated atoms play a role only at knowledge base consistency checking time. In our setting, any knowledge base will be consistent. Therefore, to what concerns query answering, the drop of negation is harmless.

On one hand, our DL will be used in order to define the relevant abstract concepts and relations of the application domain. On the other hand, conjunctive queries will be used to describe the information needs of a user and rank the answers according to a scoring function and are the logical counterpart of top-k aggregate queries we have seen in the previous section.

Scoring space. To start with, answers to a query are scored according to some scoring function. We will call the score also degree. Hence we have to fix a scoring space. We will consider as scoring-space the interval of rational numbers \([0, \top]\), where \(\top \geq 1\). One might wonder why not \(\top = 1\), as usually assumed. This is due to the fact that we may use aggregation functions (e.g., sum of scores) that may generate typically a score above 1.0. In principle, any such score may be mapped to \([0, 1]\) via the function \(h(x) = x/\top\), but for convenience we prefer to consider \([0, \top]\). What is important is not the score of an answer per se, but rather the ranking of answers induced by the scores.

3.1. The ontology representation language

Knowledge base. A knowledge base \(\mathcal{K} = (F, O, M)\) consists of a facts component \(F\), an Ontology component \(O\) and an mapping component \(M\), which are defined below.

Facts Component. We allow to store so-called graded ground facts such as “the Audi TT is to some degree (e.g., 0.85) a sports car”. Indeed, \(F\) is a finite set of expressions of the form

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

\footnote{Most OWL QL constructs are supported in our language. We refer the reader for more details on this to \textit{Appendix A.}}
where \( R \) is an \( n \)-ary relation, every \( c_i \) is a constant, and \( s \) is a degree in \([0, \top] \).

For each relation \( R \), we represent the facts \( R(c_1, \ldots, c_n)[s] \) in \( \mathcal{F} \) by means of a relational \( n + 1 \)-ary table \( T_R \), containing the records \( \langle c_1, \ldots, c_n, s \rangle \). We assume that there cannot be two records \( \langle c_1, \ldots, c_n, s_1 \rangle \) and \( \langle c_1, \ldots, c_n, s_2 \rangle \) in \( T_R \) with \( s_1 \neq s_2 \) (if there are, then we remove the one with the lower degree). Each table is sorted in descending order with respect to the degrees. For ease, we may omit the degree component and in such case the value \( \top \) is assumed.

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

\[
 t(Rl_1, \ldots, Rl_m) \sqsubseteq Rr ,
\]

where

1. \( m \geq 1 \), all \( Rl_i \) and \( Rr \) have the same arity and where each \( Rl_i \) is a so-called left hand relation and \( Rr \) is a right hand relation;
2. \( t \) is a scoring function symbol that has a fixed interpretation \( \bar{t}: [0, \top]^m \to [0, \top] \). \( \bar{t} \) will combine the scores of the left hand relations into an overall score to be assigned to the right hand relation. We assume that \( \bar{t} \) is monotone, that is, for each \( v, v' \in [0, \top]^m \) such that \( v \leq v' \), \( \bar{t}(v) \leq \bar{t}(v') \) holds, where \((v_1, \ldots, v_m) \leq (v'_1, \ldots, v'_m) \) iff \( v_i \leq v'_i \) for all \( i \). We also assume that \( \bar{t} \) is both bounded, i.e., \( \bar{t}(\ldots, v_i, \ldots) \leq v_i \) and computable, i.e. for any input, the value of \( \bar{t} \) can be determined in finite time. As \( t \) will have fixed interpretation, for the sake of ease the presentation, we will use in the following \( \bar{t} \) in place of \( t \) instead.

We also write

\[
 Rl_1 \sqcap \ldots \sqcap Rl_m \sqsubseteq Rr
\]

in place of the axiom

\[
 \min(Rl_1, \ldots, Rl_m) \sqsubseteq Rr .
\]

Axioms of the form \( Rl_1 \sqcap \ldots \sqcap Rl_m \sqsubseteq Rr \) are those considered usually in crisp DL-Lite and DRL-Lite variants and, thus, are special cases (see Appendix A). Furthermore, we assume that relations occurring in \( \mathcal{F} \) do not occur in axioms (so, we do not allow that database relation names occur in \( \mathcal{O} \)).

**Example 3.1 (Example 2.1 cont.)**. An excerpt of the domain ontology is described in Fig. 3 and partially encodes an ontology used to describe Curriculum Vitæ. We assume that we have relations \( \text{hasDegree}(\text{profID}, \text{degID}, \text{marks}) \), \( \text{knowsLanguage}(\text{profID}, \text{lanID}) \) and an atomic concept (unary relation) \( \text{Degree}(\text{degID}) \). For instance, axiom \( 4 \) states that the languages known by profile \( \text{profID} \) should be languages. Here, \( \text{knowsLanguage} \) corresponds to the projection of relation \( \text{knowsLanguage} \) on the second column.
Figure 3: Excerpt of a CV ontology.

\[
\begin{align*}
\text{Legislators} & \sqsubseteq \text{Legal Support Workers} \\
\text{Auditors} & \sqsubseteq \text{Accountants and Auditors} \\
\text{Bakers} & \sqsubseteq \text{Food Processing Workers} \\
\text{knowsLanguage}[2] & \sqsubseteq \text{Language} \\
\text{hasDegree}[2] & \sqsubseteq \text{Degree}
\end{align*}
\]

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

\[
\begin{align*}
R_r & \rightarrow A \ | \ R[i_1, \ldots, i_k] \\
R_l & \rightarrow A \ | \ R[i_1, \ldots, i_k] | \ R[i_1, \ldots, i_k].(\text{Cond}_1, \ldots, \text{Cond}_h) \\
\text{Cond} & \rightarrow ([i] \leq v) \ | \ ([i] < v) \ | \ ([i] \geq v) \ | \ ([i] > v) \ | \ ([i] = v) \ | \ ([i] \neq v)
\end{align*}
\]

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 string value or a rational number. 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 \(k\). On the other hand, \(R[i_1, \ldots, i_k].(\text{Cond}_1, \ldots, \text{Cond}_h)\) further restricts the projection \(R[i_1, \ldots, i_k]\) according to the conditions specified in \(\text{Cond}_i\). For instance, \(([i] \leq v)\) 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, \(\text{Profile}[1,5].((5 \geq 28))\) corresponds to the set of tuples \(\langle \text{profID}, \text{birthDate} \rangle\) such that the fifth column of the relation \(\text{Profile}\), \(i.e.\) the person’s age, is equal or greater than 28.

**Mapping Component.** The mapping component is a set of “mapping 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. Formally, a mapping statement is of the form

\[
R \mapsto (c_1, \ldots, c_n)[c_{\text{score}}].sql,
\]

where \(sql\) is a SQL statement returning \(n\)-ary tuples \(\langle c_1, \ldots, c_n \rangle\) with score determined by the \(c_{\text{score}}\) column. The tuples have to be ranked in decreasing order of score and, as for the fact component, we assume that there cannot be two records \(\langle c, s_1 \rangle\) and \(\langle c, s_2 \rangle\) in the result set of \(sql\) with \(s_1 \neq s_2\) (if there are, then we remove the one with the lower score). The score \(c_{\text{score}}\) may be omitted and in that case the score \(\top\) is assumed for the tuples. We assume that \(R\) occurs in \(O\), while all of the relational tables occurring in the SQL statement occur in \(F\).
As illustrative purpose, assume that we have a relation Jobs in a database with signature Jobs(jobID, name). Then, an example of mapping statement is

\[
\text{jobName} \mapsto (\text{jobID}, \text{name}) (\text{SELECT jobID, name FROM Jobs}),
\]

by means of which we state that the relation jobName occurring in the ontology component, has arity two and has to be mapped into the relation Jobs occurring in the database. On the other hand, the mapping statement

\[
\text{BigCity} \mapsto (\text{id})[\text{score}] (\text{SELECT id, ls(size; 5 \cdot 10^5, 10^6) AS score FROM CityTable ORDER BY score })
\]

defines the abstract relation BigCity as the set of city IDs, where the score of being big is determined by left shoulder function \(ls(size; 5 \cdot 10^5, 10^6)\), where size is an attribute of the CityTable relation recording the number of inhabitants of a city. The city IDs are ranked in decreasing order of score.

Finally, we assume that there is at most one abstract statement for each abstract relational symbol \(R\).

### 3.2. The query language

Concerning queries, a query consists of a conjunctive query with a scoring function to rank the answers. A conjunctive query will be the logical counterpart of SQL top-k aggregate queries we have seen in Section 2.3.

**Top-k conjunctive query.** Let \(\oplus \in \{\text{SUM}, \text{AVG}, \text{MAX}, \text{MIN}\}\) be ranking aggregate function. Then a top-k conjunctive aggregate query, or simply, conjunctive query, is of the form

\[
q(x)[s] \leftarrow \exists y \ R_1(z_1)[s_1], \ldots, R_l(z_l)[s_l], \quad \text{GroupedBy}(w), \text{OrderBy}(s = \oplus[f(s_1, \ldots, s_l, p_1(z'_1), \ldots, p_h(z'_h))])
\]

where

1. \(q\) is an \(n\)-ary relation, every \(R_i\) is an \(n_i\)-ary relation. \(R_i(z_i)\) may also be a concrete unary predicate of the form \((z \leq v), (z < v), (z \geq v), (z > v), (z = v), (z \neq v)\), where \(z\) is a variable, \(v\) is a value of the appropriate type;
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. \(z_i, z'_j\) are tuples of constants or variables in \(x\) or \(y\);
5. \(w\) is a list of variables according to which we want to group the tuples.
   - We assume that \(w\) are variables in \(x\) or \(y\) such that each variable in \(x\) occurs in \(w\);
6. \(s, s_1, \ldots, s_l\) are distinct variables and different from those in \(x\) and \(y\);
7. \(p_j\) is an \(n_j\)-ary scoring predicate assigning to each \(n_j\)-ary tuple \(c_j\) a score \(p_j(c_j) \in [0, \top]\). We require that an \(n\)-ary scoring predicate \(p\) is safe, that is, there is not an \(m\)-ary scoring predicate \(p'\) such that \(m < n\) and \(p = p'\).

Informally, all parameters are needed in the definition of \(p\).
8. \( f \) is a scoring function \( f : ([0, \top])^{l+h} \to [0, \top] \), which combines the scores of the \( l \) relations \( R_i(c^i_j) \) and the \( n \) scoring predicates \( p_j(c^j_i) \) into an overall score. We assume that \( f \) is monotone, that is, for each \( v, v' \in ([0, \top])^{l+h} \) 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 scoring predicates \( p_i \) is bounded by a constant.

We call \( q(x)[s] \) its head, \( \exists y. R_1(z_1)[s_1], \ldots, R_l(z_l)[s_l] \) its body, OrderBy\( (s = @[f(s_1, \ldots, s_l, p_1(z'_1), \ldots, p_n(z'_n)]) \) the scoring atom and GroupBy\( (w) \) the grouping atom. We also allow the scores \( [s], [s_1], \ldots, [s_l] \) and the scoring and grouping atom to be omitted. In this case we assume the value \( \top \) 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 or 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 and the tuples are grouped according to \( w \).

Below, few example queries:

\[
q(x) \leftarrow \text{SportsCar}(x) \\
\quad // \text{find sports cars}
\]

\[
q(x) \leftarrow \text{SportsCar}(x), \text{hasSpeed}(x, y), (y \geq 240) \\
\quad // \text{find sports cars whose speed exceed 240}
\]

\[
q(x)[s] \leftarrow \text{SportsCar}(x)[s_1], \text{hasSpeed}(x, y), \text{isFast}(y)[s_2], \text{OrderBy}(s = s_1 \cdot s_2) \\
\quad // \text{find fast sports cars}
\]

\[
q(x)[s] \leftarrow \text{SportsCar}(x)[s_1], \text{hasPrice}(x, p), \text{OrderBy}(s = 0.7 \cdot s_1 + 0.3 \cdot \text{ls}(p; 10000, 14000)) \\
\quad // \text{find cheap sports cars}
\]

**Example 3.2 (Example 3.1 cont.).** Assume that we have also the abstract mappings

\[
\begin{align*}
\text{hasName} & \quad \leftrightarrow \quad (\text{profID}, \text{lastName}). (\text{SELECT profID, lastName FROM Profile}) , \\
\text{hasDegree} & \quad \leftrightarrow \quad (\text{profID}, \text{degID}). (\text{SELECT profID, degID FROM HasDegree}) \\
\text{hasMark} & \quad \leftrightarrow \quad (\text{profID}, \text{mark}). (\text{SELECT profID, mark FROM HasDegree}) \\
\text{hasDegreeName} & \quad \leftrightarrow \quad (\text{degID}, \text{name}). (\text{SELECT degID, name FROM Degree}) .
\end{align*}
\]

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

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

Assume now that we additionally would like to sum-up the scores of the degrees of each person. Then, such a query may be expressed as

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

Intuitively, for the above query, we ask to group all tuples according to \( \langle \text{id}, \text{name} \rangle \) 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 \(rs(mark;100,110)\) then the score \(s_g\) of the group \(g\) is \(\sum s_i\). A group \(g\) is ranked then according to its score \(s_g\) and the top-k ranked groups are returned.

Top-k disjunctive query. We conclude by defining a disjunctive query \(q\) as usual as a finite set of conjunctive queries in which all the rules have the same head. Intuitively, the answers to a disjunctive query are the union of the answers of the conjunctive queries.

3.3. Semantics

Interpretation. An interpretation \(\mathcal{I} = \langle \Delta, \cdot \rangle\) consists of a fixed infinite domain \(\Delta\) containing a domain \(\Delta_D\) of strings and rational numbers, and an interpretation function \(\cdot\) that maps

- every atom \(A\) to a partial function \(A^\mathcal{I}: \Delta \rightarrow [0, \top]\)
- maps an \(n\)-ary predicate \(R\) to a partial function \(R^\mathcal{I}: \Delta^n \rightarrow [0, \top]\)
- constants to elements of \(\Delta\) such that \(a^\mathcal{I} \neq b^\mathcal{I}\) if \(a \neq b\) (unique name assumption).

We also assume to have one object for each constant, denoting exactly that object. In other words, we have standard names, and we do not distinguish between the alphabet of constants and the objects in \(\Delta\).

Furthermore, we assume that the relations have a typed signature and the interpretations have to agree on the relation’s type. To the easy of presentation, we omit the formalisation of this aspect and leave it at the intuitive level.

Informally, for a relation \(R\) and tuple \(c\), an interpretation assigns a score to \(R^\mathcal{I}(c)\) (similarly for \(A^\mathcal{I}(c)\)).

Note that, since \(R^\mathcal{I}\) (resp. \(A^\mathcal{I}\)) may be a partial function, some tuples may not have a score. Alternatively, we may assume \(R^\mathcal{I}\) (resp. \(A^\mathcal{I}\)) to be a total function. We use the former formulation to distinguish the case where a tuple \(c\) may be retrieved, even though the score is 0, from the case where a tuple is not retrieved, since it does not satisfy the query. In particular, if a tuple does not belong to an extensional relation, then its score is assumed to be undefined, while if \(R^\mathcal{I}\) (resp. \(A^\mathcal{I}\)) is total, then the score of this tuple would be 0.

In the following, we use \(c\) to denote an \(n\)-tuple of constants, and \(c_{i_1,\ldots,i_k}\) to denote the \(i_1,\ldots,i_k\)-th components of \(c\). For instance, \((a,b,c,d)_{3,1,4}\) is \((c,a,d)\).

Please note that the order matters.

Models of a knowledge base. Concerning facts, an interpretation \(\mathcal{I}\) is a model of (or satisfies) a fact \(R(c)[s]\), denoted \(\mathcal{I} \models R(c)[s]\), iff \(R^\mathcal{I}(c) \geq s\) whenever \(R^\mathcal{I}(c)\) is defined. Furthermore, an interpretation \(\mathcal{I}\) is a model of (satisfies) a fact component \(F\) iff it satisfies each element in it.

Concerning comparison predicates, \(\mathcal{I}\) is a model of (or satisfies) \((i) \leq v)(c)\), denoted \(\mathcal{I} \models (i) \leq v)(c)\), iff \(c_i \leq v\), i.e., the projection of tuple \(c\) on the \(i\)-th
column is less or equal than the value \( v \) and similarly for the other comparison constructs, \( ([i] < v), ([i] \geq v), ([i] > v), ([i] = v) \) and \( ([i] \neq v) \). We assume in all of them that both the values \( v \) and \( c_i \) are of the appropriate type.

The interpretation function \( \mathcal{I} \) has to satisfy the following conditions: for all \( c \in \Delta^k \) and \( n \)-ary relation \( R \):

\[
R[i_1, \ldots, i_k]^{\mathcal{I}}(c) = \sup \{ R^\mathcal{I}(c') \mid c' \in \Delta^n, c'|_{i_1, \ldots, i_k} = c \}
\]

\[
(R[i_1, \ldots, i_k], (\text{Cond}_1, \ldots, \text{Cond}_l))^\mathcal{I}(c) = \sup \{ R^\mathcal{I}(c') \mid c' \in \Delta^n, c'|_{i_1, \ldots, i_k} = c, \mathcal{I} \models \text{Cond}_j(c') \}.
\]

Some explanations are in place. Consider \( R[i_1, \ldots, i_k] \). Informally, from a classical semantics point of view, \( R[i_1, \ldots, i_k] \) is the projection of the relation \( R \) over the columns \( i_1, \ldots, i_k \) and, thus, corresponds to the set of tuples

\[
\{ c \mid \exists c' \in R \text{ s.t. } c'|_{i_1, \ldots, i_k} = c \}.
\]

Note that for a fixed tuple \( c \) there may be several tuples \( c' \in R \) such that \( c'|_{i_1, \ldots, i_k} = c \). As now any such \( c' \) may have an attached score, we choose the maximal among these scores.

Now given an interpretation \( \mathcal{I} \), the notion of \( \mathcal{I} \) is a model of (satisfies) an axiom \( \tau \), denoted \( \mathcal{I} \models \tau \), is defined as follows:

\[
\mathcal{I} \models t(Rl_1, \ldots, Rl_m) \subseteq Rr \text{ iff for all } c \in \Delta^k, t(Rl_1^\mathcal{I}(c), \ldots, Rl_m^\mathcal{I}(c)) \subseteq Rr^\mathcal{I}(c),
\]

where we assume that the arity of \( Rr \) and all \( Rl_i \) is \( k \). An interpretation \( \mathcal{I} \) is a model of (satisfies) an ontology \( \mathcal{O} \) if it satisfies each element in it.

Concerning mapping statements, the notion of \( \mathcal{I} \) is a model of (satisfies) a mapping statement \( \sigma \), denoted \( \mathcal{I} \models \sigma \), is defined as follows: let SQL be a SQL statement and let use denote \( \text{sql}^\mathcal{I} = \{ (c, s) \mid (c, s) \text{ is an answer to the SQL statement } \text{sql} \} \), then

\[
\mathcal{I} \models R \rightarrow (t_1, \ldots, t_n)[c_{\text{score}}]. \text{sql} \text{ iff for all } c \in \Delta^n, R^\mathcal{I}(c) \geq s \text{ if } (c, s) \in \text{sql}^\mathcal{I}.
\]

An interpretation \( \mathcal{I} \) is a model of (satisfies) a mapping component \( \mathcal{M} \) iff it satisfies each element in it and \( \mathcal{I} \) is a model of (satisfies) a knowledge base if it satisfies each component.

**Models of a query.** Concerning queries, we may assume that they are of the form

\[
q(x)[s] \leftarrow \exists y \phi(x, y)[s],
\]

where \( \phi(x, y)[s] \) is

\[
R_1(z_1)[s_1], \ldots, R_l(z_l)[s_l],
\]

\[
\text{GroupBy}(w), \text{OrderBy}(s = @[f(s_1, \ldots, s_l, p_1(z'_1), \ldots, p_h(z'_h))]).
\]
Let \( R \) be an \( n \)-ary relation, \( c \) an \( n \)-ary tuple and \( s \) be a score. Then an interpretation \( I \) is a model of (satisfies) \( R(c)[s] \), denoted \( I \models R(c)[s] \), if and only if \( R(c)[s] \) is not satisfied.

Note that if \( R(c)[s] \) is not defined then \( R(c)[n] \) is not satisfied. We say that \( I \) is a model of (satisfies) a query of the form (6) if for all \( c \in \Delta^n \), \( q^I(c) \geq s \) whenever

\[
\begin{align*}
  s &= \sup\{ s' \mid g = \{ \langle c, c_1', \ldots, c_k' \rangle \}, \\
  g &= \text{a group of } k \text{ tuples with identical projection on the variables in } w, \\
  s' &= \sup\{ s^r \mid r \in [s^1, \ldots, s^k] \}, \\
  \text{where for } 1 \leq r \leq k, \text{ each score } s^r \text{ is computed as follows:} \\
  c'_r \in \Delta \times \cdots \times \Delta \text{ is a substitution of variables in } y, \\
  I \models R_i(c_r)[s^r], \text{ projection of } \langle c, c'_r \rangle \text{ on the variables } z_i, \\
  v_j = p_j(c'_j), \text{ where } c'_j \text{ is the projection of } \langle c, c'_r \rangle \text{ on the variables } z'_j, \\
  s^r &= f(s_1^1, \ldots, s_k^1, v_1^1, \ldots, v_k^1). 
\end{align*}
\]

In the above expression, for each group of tuples, \( g = \{ \langle c, c_1', \ldots, c_k' \rangle \} \), we compute the scores \( s^r \) each tuple belonging to that group and then apply the ranking aggregation function.

**Entailment.** We say \( K \) entails \( q(c) \) to degree \( s \), denoted \( K \models q(c)[s] \), if for each model \( I \) of \( K \), it is true that \( q^I(c) \geq s \) whenever \( q^I(c) \) is defined. We implicitly assume that the score \( s \) is as large as possible, i.e., there is no \( s' > s \) such that \( q^I(c) \geq s' \). For a disjunctive query \( q = \{ q_1, \ldots, q_m \} \), \( K \) entails \( q(c) \) to degree \( s \), denoted \( K \models q(c)[s] \), if \( K \models q_i(c)[s] \) for some \( q_i \in q \).

**Top-k answer set.** The answer set of \( q \) w.r.t. \( K \) is

\[
\text{ans}(K, q) = \{ \langle c, s \rangle \mid K \models q(c)[s] \}. 
\]

As now each answer to a query has a score, the basic inference problem we are interested in is the top-k retrieval problem, formulated as follows. Given a knowledge base \( K \), and a disjunctive query \( q \), retrieve \( k \) tuples \( \langle c, s \rangle \) that instantiate the query relation \( q \) with maximal scores (if \( k \) such tuples exist), and rank them in decreasing order relative to the score \( s \), denoted

\[
\text{ans}_k(K, q) = \text{Top}_k \text{ans}(K, q). 
\]

The following example, taken from [26], will be used to illustrate our top-k query answering procedure.

**Example 3.3.** Suppose the set of axioms is

\[
\mathcal{O} = \{ P_2[2] \subseteq A, \ A \subseteq P_1[1], \ B \subseteq P_2[1] \}. 
\]

Let us assume that we have the mapping statements

\[
\begin{align*}
  P_2 & \rightarrow (c, s).\text{(SELECT } c, s \text{ FROM TabP2)} \\
  B & \rightarrow (c).\text{(SELECT } c \text{ FROM TabB)} \\
  C & \rightarrow (c).\text{(SELECT } c \text{ FROM TabC)}
\end{align*}
\]
and that the set of facts $\mathcal{F}$ is

\[
\text{TabP}2 = \{(0,s),(3,t),(4,q),(6,q)\},
\]
\[
\text{TabB} = \{(1),(2),(5),(7)\},
\]
\[
\text{TabC} = \{(5),(3),(2),(4)\}.
\]

Assume our disjunctive query is $q = \{q', q''\}$ where

\[
q' := q(x)[s] \leftarrow \exists y \exists z. P_2(x,y), P_1(y,z), \text{OrderBy}(s = f(p(x)))
\]
\[
q'' := q(x)[s] \leftarrow C(x), \text{OrderBy}(s = f(r(x)))
\]

the scoring function $f$ is the identity $f(z) = z$ (if is monotone, of course), the scoring predicate $p$ is $p(x) = \max(0, 1 - x/10)$, and the scoring predicate $r$ is $r(x) = \max(0, 1 - (x/5)^2)$. Therefore, we can rewrite the query $q$ as

\[
q' := q(x)[s] \leftarrow \exists y \exists z. P_2(x,y), P_1(y,z), \text{OrderBy}(s = \max(0, 1 - x/10))
\]
\[
q'' := q(x)[s] \leftarrow C(x), \text{OrderBy}(s = \max(0, 1 - (x/5)^2))
\]

Now, it can be verified that

\[
\mathcal{K} \models q(3)[0.7] \quad , \quad \mathcal{K} \not\models q(2)[0.84]
\]

and for any $v \in [0, 1]$, $\mathcal{K} \not\models q(9)[v]$. In the former case, any model $\mathcal{I}$ of $\mathcal{K}$ satisfies $P_2(3,t)$. But, $\mathcal{I}$ satisfies $O$, so $\mathcal{I}$ satisfies $P_2[2] \subseteq P_1[1]$. As $\mathcal{I}$ satisfies $P_2(3,t)$, $\mathcal{I}$ satisfies $P_2[2](t)$ and, thus, $P_1[1](t)$. As $0.7 = \max(0, 1 - 3/10)$, it follows that $\langle 3, 0.7 \rangle$ evaluates the body of $q'$ true in $\mathcal{I}$. On the other hand, $\langle 3, 0.64 \rangle$ evaluates the body of $q''$ true in $\mathcal{I}$. Hence, under $\mathcal{I}$ the maximal score for 3 is 0.7, i.e., $q^2(3) \geq 0.7$. The other cases can be shown similarly.

In summary, it can be shown that the top-4 answer set of $q$ is $\text{ans}_4(\mathcal{K}, q) = \{(0,1.0),(1,0.9),(2,0.84),(3,0.7)\}$.

4. Query answering

In this section we will describe our top-k query answering procedure, which consists of four steps: (i) query reformulation; (ii) redundancy elimination; (iii) query evaluation; and (iv) result merging. Specifically, given a query $q(x)[s] \leftarrow \exists y \phi(x, y)[s]$, 1. by considering $O$, the user query $q$ is reformulated into a set of conjunctive queries $r(q, O)$. Informally, 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 $O$ contains the axioms

\[
t_1(B_1, B_2) \sqsubseteq A
\]
\[
t_2(C_1, C_2) \sqsubseteq A
\]
then we can reformulate the query into two queries
\[ q(x)[s] \leftarrow B_1(x)[s_1], B_2(x)[s_2], \text{OrderBy}(s = f(t_1(s_1, s_2))) \]
\[ q(x)[s] \leftarrow C_1(x)[s_1], C_2(x)[s_2], \text{OrderBy}(s = f(t_2(s_1, s_2))) ; \]

Note that the reason for, e.g., the expression \( f(t_1(s_1, s_2)) \) follows informally from the following argument. From \( B_i(x) \geq s_i, t_1(B_1(x), B_2(x)) \leq A(x) \) and the monotonicity of \( t \), we have \( A(x) \geq t(s_1, s_2) \). So, for \( s' = t(s_1, s_2) \), \( q(x) \geq s = f(s') = f(t_1(s_1, s_2)) \).

2. from the set of reformulated queries \( r(q, O) \) we remove redundant queries;
3. the reformulated queries \( q' \in r(q, O) \) are translated to top-k SQL queries and evaluated. The query evaluation of each top-k SQL query returns the top-k answer set for that query;
4. all the \( n = |r(q, O)| \) top-k answer sets have to be merged into the unique top-k answer set \( ans_k(K, q) \). To this purpose, we describe a method to merge all the answer sets without necessarily computing the union.

We next address in more detail the above steps.

4.1. Query reformulation
The query reformulation step is a generalisation of [8, 24, 26] to our case and is as follows.

**Bound, unbound variable.** We say that a variable in a conjunctive query is bound if it corresponds to either a distinguished variable or a shared variable, i.e., a variable occurring at least twice in the query body, or a constant, while we say that a variable is unbound if it corresponds to a non-distinguished non-shared variable (as usual, we use the symbol "_" to represent non-distinguished non-shared variables). Note that an expression \( R[i_1, \ldots, i_k] \) can be seen as the relation \( R(x) \), where the variables in position \( i_1, \ldots, i_k \) are unbound. We write also \( R(\ldots, x_{i_1}, \ldots, x_{i_k}, \ldots, \ldots) \) to denote the relation \( R(x) \) in which all variables except those in position \( i_1, \ldots, i_k \) are unbound. Given a vector of variables \( x \), and a condition \( Cond \) occurring in the left hand side of an axiom then \( Cond(x) \) is defined as follows: \(([i] \leq v)(x) = (x_i \leq v), ([i] < v)(x) = (x_i < v), ([i] \geq v)(x) = (x_i \geq v), ([i] > v)(x) = (x_i > v), ([i] = v)(x) = (x_i = v), \) and \(([i] \neq v)(x) = (x_i \neq v)\).

**Applicable axiom.** An axiom \( \tau \) is applicable to an atom \( A(x)[s] \) in a query body, if \( \tau \) has \( A \) in its right hand side, while \( \tau \) is applicable to an atom \( R(\ldots, x_{i_1}, \ldots, x_{i_k}, \ldots, \ldots)[s] \) in a query body, if the right hand side of \( \tau \) is \( R[i_1, \ldots, i_k] \). We indicate with \( gr(g; \tau) \) the expression obtained from the atom or relation \( g \) by applying the inclusion axiom \( \tau \) and with \( \theta(g; \tau) \) the variable substitution obtained from the atom or relation \( g \) by applying the inclusion axiom \( \tau \) specifically, we have the following.

- If \( g \) is \( A(x)[s] \) and \( \tau \) is \( t(Rl_1, \ldots, Rl_m) \subseteq A \) then
  \[ gr(g; \tau) = \{ C_1(x)[s_1], \ldots, C_m(x)[s_m] \} \]
where for each \( r \in \{1, \ldots, m\} \), \( s_r \) is a new scoring variable,

\[
\theta(g; \tau) = \{s/t(s_1, \ldots, s_m)\}
\]

and

- if \( R_l = A_r \) then \( C_r(x)[s_r] \) is \( A_r(x)[s_r] \);
- if \( R_l = R[j] \) then \( C_r(x)[s_r] \) is \( R_r(\ldots, x, \ldots)[s_r] \), where \( x \) is in the \( j \)-th position;
- if \( R_l = R[j].(Cond_1, \ldots, Cond_h) \) then \( C_r(x)[s_r] \) is

\[
R_r(z)[s_r], Cond_1(z), \ldots, Cond_h(z),
\]

where \( l \) is the arity of \( R \), \( z = \langle z_1, \ldots, z_l \rangle \) and all \( z_h \) are new variables.

• If \( g \) is \( R(\ldots, x_1, \ldots, x_k)[s] \) and \( \tau \) is \( t(R_l_1, \ldots, R_l_m) \sqsubseteq R[i_1, \ldots, i_k] \) then \( gr(g; \tau) \) is

\[
\{C_1(x_1, \ldots, x_k)[s_1], \ldots, C_m(x_1, \ldots, x_k)[s_m]\}
\]

where for each \( r \in \{1, \ldots, m\}, s_r \) is a new scoring variable,

\[
\theta(g; \tau) = \{s/t(s_1, \ldots, s_m)\}
\]

and

- if \( R_l = A_r \) then \( k = 1 \), and \( C_r(x_1)[s_r] \) is \( A_r(x_1)[s_r] \);
- if \( R_l = R[j_1, \ldots, j_k] \) then \( C_r(x_1, \ldots, x_k)[s_r] \) is

\[
R_r(\ldots, x_1, \ldots, x_k)[s_r],
\]

where variables in position \( j_1, \ldots, j_k \) are \( x_1, \ldots, x_k \);
- if \( R_l = R[j_1, \ldots, j_k].(Cond_1, \ldots, Cond_h) \) then \( C_r(x_1, \ldots, x_k)[s_r] \) is

\[
R_r(z)[s_r], Cond_1(z), \ldots, Cond_h(z),
\]

where \( l \) is the arity of \( R \), \( z = \langle z_1, \ldots, x_1, \ldots, x_k, \ldots, z_l \rangle \), all \( z_h \) are new variables and the variables in position \( j_1, \ldots, j_k \) are \( x_1, \ldots, x_k \).

**Example 4.1.** Consider the query

\[
q_0 := q(x)[s] \leftarrow A(x)[s_1], B(x)[s_2], \text{OrderBy}(s = \min(s_1, s_2))
\]

and suppose that \( \Omega \) contains the axioms

\[
\tau_1 := 0.8 \cdot B_1 \sqsubseteq A \\
\tau_2 := 0.7 \cdot B_2 \sqsubseteq A
\]
Algorithm 1 \( \text{QueryRef}(q, \mathcal{O}) \)

**Input:** A disjunctive query \( q \), axioms \( \mathcal{O} \).

**Output:** Set of reformulated conjunctive queries \( r(q, \mathcal{O}) \).

1. \( r(q, \mathcal{O}) := q \)
2. repeat
3. \( S = r(q, \mathcal{O}) \)
4. for all \( q \in S \) do
5. for all \( g \in q \) do
6. if \( \tau \in \mathcal{O} \) is applicable to \( g \) then
7. \( r(q, \mathcal{O}) := r(q, \mathcal{O}) \cup \{g[g/\tau] \theta(g, \tau)\} \)
8. for all \( g_1, g_2 \in q \) do
9. if \( g_1 \) and \( g_2 \) unify then
10. \( r(q, \mathcal{O}) := r(q, \mathcal{O}) \cup \{\kappa(\text{reduce}(q, g_1, g_2))\} \)
11. \( r(q, \mathcal{O}) := \text{removeSubs}(r(q, \mathcal{O})) \)
12. until \( S = r(q, \mathcal{O}) \)
13. return \( r(q, \mathcal{O}) \)

Then for \( g = A(x)[s_1] \) we have

\[
\begin{align*}
\text{gr}(g; \tau_1) & := \{B_1(x)[s_3], \theta(g; \tau_1) := \{s_1/0.8 \cdot s_3\} \\
\text{gr}(g; \tau_2) & := \{B_2(x)[s_4], \theta(g; \tau_2) := \{s_1/0.7 \cdot s_4\}.
\end{align*}
\]

Now, using \( \text{gr}(g; \tau_i) \) and \( \theta(g; \tau_i) \) we may reformulate the original query by replacing in the query \( q \), the expression \( g \) with the elements in \( \text{gr}(g; \tau_i) \) and then applying the score variable substitution \( \theta(g; \tau_i) \) to the scoring atom. Therefore, we get two new queries

\[
\begin{align*}
q_1 & := q(x)[s] \leftarrow B_1[s_3], B(x)[s_2], \text{OrderBy}(s = \min(0.8 \cdot s_3, s_2)) \\\nq_2 & := q(x)[s] \leftarrow B_2[s_4], B(x)[s_2], \text{OrderBy}(s = \min(0.7 \cdot s_4, s_2)).
\end{align*}
\]

We are now ready to present the query reformulation algorithm (see Algorithm [4]). Given a disjunctive query \( q \) and a set of axioms \( \mathcal{O} \), the algorithm reformulates \( q \) in terms of a set of conjunctive queries \( r(q, \mathcal{O}) \), which then can be evaluated over the facts \( \mathcal{F} \) using the mappings in the mapping component \( \mathcal{M} \).

In the algorithm, the expression \( q[g/g'] \theta(g, \tau) \) denotes the query obtained from \( q \) by replacing the atom \( g \) with a new atom \( g' \). To the resulting query we apply the score variable substitution \( \theta(g; \tau_i) \) to the scoring atom. At step 8, for each pair of atoms \( g_1, g_2 \) that unify, the algorithm computes the query \( q' = \text{reduce}(q, g_1, g_2) \), by applying to \( q \) the most general unifier between \( g_1 \) and \( g_2 \). Due to the unification, variables that were bound in \( q \) may become unbound in \( q' \). Hence, inclusion axioms not applicable to atoms of \( q \), may become applicable to atoms of \( q' \) (in the next executions of step (5)). Function \( \kappa \) applied to \( q' \) replaces with each unbound variable in \( q' \). Finally, in step 11 we remove from the set of queries \( r(q, \mathcal{O}) \), those which are already subsumed in \( r(q, \mathcal{O}) \).

---

3We say that two atoms \( g_1 = r(x_1, \ldots, x_n) \) and \( g_2 = r(y_1, \ldots, y_n) \) unify, if for all \( i \), either \( x_i = y_i \) or \( x_i = \_ \) or \( y_i = \_ \). If \( g_1 \) and \( g_2 \) unify, then the unification of \( g_1 \) and \( g_2 \) is the atom \( r(z_1, \ldots, z_n) \), where \( z_i = x_i \) if \( x_i = y_i \) or \( y_i = \_ \) otherwise \( z_i = y_i \). [3]
Query subsumption. The notion of query subsumption is similar as for the classical database theory [34]. Given two queries \( q_i \) \((i = 1, 2)\) with same head \( q(x)[s] \) and \( q_1 \neq q_2 \), we say that \( q_1 \) is subsumed by \( q_2 \), denoted \( q_1 \sqsubseteq q_2 \), iff for any interpretation \( I \), for all tuples \( c \), \( q_1^I(c) \sqsubseteq q_2^I(c) \). Essentially, if \( q_1 \sqsubseteq q_2 \) and both \( q_1 \) and \( q_2 \) belong to \( r(q, O) \) then we can remove \( q_1 \) from \( r(q, O) \) as \( q_1 \) produces a lower ranked result than \( q_2 \) with respect to the same tuple \( c \).

A condition for query subsumption is the following. Assume that \( q_1 \) and \( q_2 \) do not share any variable. This can be accomplished by renaming all variables in e.g. \( q_1 \). Then it can be shown in a similar way as in [26] that

**Proposition 4.2.** If \( q_1 \) and \( q_2 \) share the same score combination function, then \( q_1 \sqsubseteq q_2 \) iff there is a variable substitution \( \theta \) such that for each relation \( R(z_2) \) occurring in the rule body of \( q_2 \) there is a relation \( R(z_1) \) occurring in the rule body of \( q_1 \) such that \( R(z_2) = R(z_1)\theta \).

More complicated are cases in which \( q_1 \) and \( q_2 \) do not share the same score combination function.

**Example 4.3 (Example 4.1 cont. ).** Suppose that the ontology component \( O \) has additionally the recursive axiom \( \tau_3 := 0.9 \cdot A \cdot B_3 \sqsubseteq A \). Therefore, through the query reformulation procedure we get a new query

\[
q_3 := q(x)[s] \leftarrow A(x)[s_5], B_3(x)[s_6], B(x)[s_2], \text{OrderBy}(s = \min(0.9 \cdot s_5 \cdot s_6, s_2))
\]

Now, let’s compare \( q_0 \) with \( q_3 \). It turns out that \( q_3 \sqsubseteq q_0 \) as for any score \( n \) of the ground atom \( A(c) \), \( \min(0.9 \cdot n \cdot s_6, s_2) \leq \min(n, s_2) \), for any value for \( s_2 \) and \( s_6 \) (as the scoring function in the axioms are bounded).

We can extend the query subsumption condition in Proposition 4.2 in the following way. Let \( q_1 \) and \( q_2 \) be two queries and let \( \sigma_1 \) and \( \sigma_2 \) be the scoring component of \( q_1 \) and \( q_2 \), respectively. Then as in [26], it can be shown that

**Proposition 4.4.** \( q_1 \sqsubseteq q_2 \) iff there is a variable substitution \( \theta \) such that for each relation \( R(z_2) \) occurring in the rule body of \( q_2 \) there is a relation \( R(z_1) \) occurring in the rule body of \( q_1 \) such that \( R(z_2) = R(z_1)\theta \), and \( \sigma_1 \theta \leq \sigma_2 \) for all variables occurring in \( \sigma_1 \theta \) and \( \sigma_2 \).

**Example 4.5 (Example 4.3 cont. ).** Let \( \theta = \{s_5/s_1\} \) then \( \sigma_0 = \min(s_1, s_2) \), while \( \sigma_3 \theta = \min(0.9 \cdot s_1 \cdot s_6, s_2) \). It is easily verified that \( \min(0.9 \cdot s_1 \cdot s_6, s_2) = \sigma_3 \theta \leq \sigma_0 = \min(s_1, s_2) \) as the score combination function is monotone in its arguments and \( 0.9 \cdot s_1 \cdot s_6 \leq s_1 \).

We use Proposition 4.4 to avoid the recursive application of the query reformulation steps (see, e.g., Example 4.3). In such cases the check of the condition \( \sigma_1 \theta \leq \sigma_2 \) is easy due to the monotonicity of the score combination function, the monotonicity and boundness of the scoring function in axioms. This concludes the query reformulation step.

The following is a complete example.
Example 4.6. Consider Example 3.3. At step 1, \( r(q, \mathcal{O}) \) is initialized with \( \{q', q''\} \). It is easily verified that both conditions in step 6 and step 9 fail for \( q'' \). So we proceed with \( q' \). Let \( \sigma = s = \max(0, 1 - x/10) \). Then at the first execution of step 7, the algorithm inserts query \( q_1 \),

\[
q_1 := q(x)[s] \leftarrow P_2(x, y), A(y), \text{OrderBy}(\sigma)
\]

into \( r(q, \mathcal{O}) \) using the axiom \( A \sqsubseteq P_1[1] \).

At the second execution of step 7, the algorithm inserts query \( q_2 \),

\[
q_2 := q(x)[s] \leftarrow P_2(x, y), P_2(\sigma, y), \text{OrderBy}(\sigma)
\]

using the axiom \( P_2[2] \sqsubseteq A \). Since the two atoms of the second query unify, \( \text{reduce}(q, q_1, q_2) \) returns

\[
q(x)[s] \leftarrow P_2(x, y), \text{OrderBy}(\sigma)
\]

and since now \( y \) is unbound (\( y \) does not occur in \( \sigma \)), after application of \( \kappa \), step 10 inserts the query \( q_3 \),

\[
q_3 := q(x)[s] \leftarrow P_2(x, \_), \text{OrderBy}(\sigma)
\]

At the third execution of step 7, the algorithm inserts query \( q_4 \),

\[
q_4 := q(x)[s] \leftarrow B(x), \text{OrderBy}(\sigma)
\]

using the axiom \( B \sqsubseteq P_2[1] \) and stops. Note that we need not to evaluate all queries \( q_i \). Indeed, it can be verified that for each \( q_i \) and all constants \( c \), the scores of \( q_3 \) and \( q_4 \) are not lower than all the other queries \( q_i \) and \( q' \). That is, we can restrict the evaluation of the set of reformulated queries to \( r(q, \mathcal{O}) = \{q'', q_3, q_4\} \) only. As a consequence, the top-4 answers to the original query are \( [(0, 1.0), (1, 0.9), (2, 0.84), (3, 0.7)] \), which are the top-4 ranked tuples of the union of the answer sets of \( q'' \), \( q_3 \) and \( q_4 \).

4.2. Computing top-k answers

The main property of the query reformulation algorithm is as follows:

\[
\text{ans}_k(K, q) = \text{Top}_k\{\langle c, v \rangle \mid q_i \in r(q, \mathcal{O}), \mathcal{F} \models q_i(c, v)\}
\]

The above property dictates that the set of reformulated queries \( q_i \in r(q, \mathcal{O}) \) can be used to find the top-k answers, by evaluating them over the set of instances \( \mathcal{F} \) only, i.e., over the database, without referring to the ontology \( \mathcal{O} \) anymore. In the following, we show how to find the top-k answers of the union of the answer sets of conjunctive queries \( q_i \in r(q, \mathcal{O}) \).

A naive solution would be: we compute for all \( q_i \in r(q, \mathcal{O}) \) the whole answer set \( \text{ans}(q_i, \mathcal{F}) = \{\langle c, v \rangle \mid \mathcal{F} \models q_i(c, v)\} \), then we compute the union, \( \bigcup_{q_i \in r(q, \mathcal{O})} \text{ans}(q_i, \mathcal{O}) \), of these answer sets, order it in descending order of the scores and then we take the top-k tuples. We note that each conjunctive query
\( q_i \in r(q, O) \) can easily be transformed into a top-k SQL query expressed over \( DB(F) \), i.e., the database encoding \( F \). The transformation is conceptually simple.

A major drawback of this solution is the fact that there might be too many tuples with non-zero score and hence for any query \( q_i \in r(q, O) \), all these scores should be computed and the tuples should be retrieved. This may not be feasible in practice. Indeed, in our experiments (see Fig. 5, queries 1, 2, 7), in some cases this approach didn’t work at all due to main memory problems and motivated the algorithm described in the next sections, called *Disjunctive Threshold Algorithm*, initially proposed in [26], that does address the case for top-k join queries only. Therefore, in the following we first recap from [26] the case of top-k join queries (Section 4.2.1) and then describe our novel algorithm to handle queries with ranking aggregates too (Section 4.2.2).

### 4.2.1. The DTA for top-k join queries

An immediate method to compute \( ans_k(K, q) \) for top-k join queries is to compute for all \( q_i \in r(q, O) \), the top-k answers \( ans_k(F, q_i) \). If both \( k \) and the number, \( n_q = \vert r(q, O) \vert \), of reformulated queries is reasonable, then we may compute the union,

\[
U(q, K) = \bigcup_{q_i \in r(q, O)} ans_k(F, q_i)
\]

of these top-k answer sets, order it in descending order w.r.t. score and then we take the top-k tuples.

As an alternative, we can avoid to compute the whole union \( U(q, K) \), so further improving the answering procedure, by relying on a disjunctive variant [26] of the so-called Threshold Algorithm (TA) [11], called *Disjunctive TA* (DTA).

We recall that the TA has been developed to compute the top-k answers of a conjunctive query with monotone score combination function. In the following we show that we can use the same principles of the TA to compute the top-k answers of the union of conjunctive queries without ranking aggregates:

1. First, we compute for all \( q_i \in r(q, O) \), the top-k answers \( ans_k(F, q_i) \), using top-k rank-based relational database engine. Now, let us assume that the tuples in the top-k answer set \( ans_k(F, q_i) \) are sorted in decreasing order with respect to the score.
2. Then we process each top-k answer set \( ans_k(F, q_i) \) (\( q_i \in r(q, O) \)) according to some criteria (e.g., in parallel, or alternating fashion, or by selecting the next tuple from the answer set with highest threshold \( \theta_i \) defined below), and top-down (i.e. the higher scored tuples in \( ans_k(F, q_i) \) are processed before the lower scored tuples in \( ans_k(F, q_i) \)).
   
   (a) For each processed tuple \( c \), if its score is one of the \( k \) highest we have already computed, then remember tuple \( c \) and its score \( s_c \) (ties are broken arbitrarily, so that only \( k \) tuples and their scores need to be remembered at any time).
(b) For each answer set \( \text{ans}_k(\mathcal{F}, q_i) \), let \( \theta_i \) be the score of the last tuple processed in this set. Define the threshold value \( \theta \) to be
\[
\theta = \max(\theta_1, ..., \theta_{n_q}) .
\]
(c) As soon as at least \( k \) tuples have been processed whose score is at least equal to \( \theta \), then halt (indeed, any successive retrieved tuple will have score \( \leq \theta \)).
(d) Let \( Y \) be the set containing the \( k \) tuples that have been processed with the highest scores. The output is then the set \( \{ (c, s_c) \mid c \in Y \} \). This set is \( \text{ans}_k(\mathcal{K}, q) \).

It is not difficult to see that the DTA determines the top-\( k \) answers. Indeed, if at least \( k \) tuples have been processed whose score is at least equal to \( \theta \) then any new not yet processed tuple \( c \) will have score bounded by \( \theta \) and, thus, it cannot make it into the top-\( k \). Hence, we can stop and the top-\( k \) tuples are among those already processed. The following example illustrates the DTA.

**Example 4.7.** Consider Example 4.6. Suppose we are interested in retrieving the top-3 answers of the disjunctive query \( q = \{ q', q'' \} \). We have seen that it suffices to find the top-3 answers of the union of the answers to \( q_3, q_4 \) and to \( q'' \). Let us show how the DTA works. First, we submit \( q_3, q_4 \) and \( q'' \) to a rank-based relational database engine, to compute the top-3 answers. It can be verified that
\[
\begin{align*}
\text{ans}_3(\mathcal{F}, q_3) &= [(0,1.0), (3,0.7), (4,0.6)] \\
\text{ans}_3(\mathcal{F}, q_4) &= [(1,0.9), (2,0.8), (5,0.5)] \\
\text{ans}_3(\mathcal{F}, q'') &= [(2,0.84), (3,0.64), (4,0.36)].
\end{align*}
\]
The lists are in descending order w.r.t. the score from left to right. Now we process alternatively \( \text{ans}_3(\mathcal{F}, q_3) \), then \( \text{ans}_3(\mathcal{F}, q_4) \) and then \( \text{ans}_3(\mathcal{F}, q'') \) in decreasing order of the score. The table below summaries the execution of our DTA algorithm. The ranked list column contains the list of tuples processed.

<table>
<thead>
<tr>
<th>Step</th>
<th>Tuple</th>
<th>( \theta_{q_3} )</th>
<th>( \theta_{q_4} )</th>
<th>( \theta_{q''} )</th>
<th>( \theta )</th>
<th>ranked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,1.0)</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(0,1.0)</td>
</tr>
<tr>
<td>2</td>
<td>(1,0.9)</td>
<td>1.0</td>
<td>0.9</td>
<td>-</td>
<td>-</td>
<td>(0,1.0), (1,0.9)</td>
</tr>
<tr>
<td>3</td>
<td>(2,0.84)</td>
<td>2.0</td>
<td>0.8</td>
<td>0.84</td>
<td>1.0</td>
<td>(0,1.0), (1,0.9), (2,0.84)</td>
</tr>
<tr>
<td>4</td>
<td>(3,0.7)</td>
<td>0.7</td>
<td>0.9</td>
<td>0.84</td>
<td>0.9</td>
<td>(0,1.0), (1,0.9), (2,0.84),(3,0.7)</td>
</tr>
<tr>
<td>5</td>
<td>(2,0.8)</td>
<td>0.7</td>
<td>0.8</td>
<td>0.84</td>
<td>0.84</td>
<td>(0,1.0), (1,0.9), (2,0.84),(3,0.7)</td>
</tr>
</tbody>
</table>

At step 5 we stop as the ranked list already contains three tuples above the threshold \( \theta = 0.84 \). So, the final output is \( \text{ans}_k(\mathcal{F}, q_3) = [(0,1.0), (1,0.9), (2,0.84)] \). Note that not all tuples have been processed.

As computing the top-\( k \) answers of each query \( q_i \in r(q, \mathcal{O}) \) requires (sub) linear time w.r.t. the database size, it is easily verified that the disjunctive TA algorithm is linear in data complexity.

**Proposition 4.8** ([28]). Given \( \mathcal{K} = (\mathcal{F}, \mathcal{O}, \mathcal{M}) \) and a disjunctive query \( q \) without ranking aggregates, then the DTA determines \( \text{ans}_k(\mathcal{K}, q) \) in (sub) linear time w.r.t. the size of \( \mathcal{F} \). \( \square \)
Furthermore, the above method has the non-negligible advantage to be based on existing technology for answering top-\(k\) queries over relational databases, improves significantly the naive solution to the top-\(k\) retrieval problem, and is not difficult to implement.

### 4.2.2. The DTA with ranking aggregates

Unfortunately, in case where a ranking aggregate occurs in a query, the DTA has to be modified as the stopping condition does not work anymore, as illustrated in the following example.

**Example 4.9.** Suppose that \(r(q, O) = \{q_1, q_2\}\), where

\[
q_1 := q(x)[s] \leftarrow R(x, y)[s_1], \text{GroupBy}(x, y), \text{OrderBy}(s = \text{Min}[s_1])
\]

\[
q_2 := q(x)[s] \leftarrow P(x, y)[s_1], \text{GroupBy}(x, y), \text{OrderBy}(s = \text{Min}[s_1])
\]

and that

\[
\text{ans}_3(F, q_1) = \{(a, 1.0), (b, 0.7), (d, 0.4)\}
\]

\[
\text{ans}_3(F, q_2) = \{(d, 0.9), (e, 0.6), (f, 0.5)\}
\]

By applying the DTA, we would get

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Step} & \text{Tuple} & d_{q_1} & d_{q_2} & \theta & \text{ranked list} \\
\hline
1 & (a, 1.0) & 1.0 & - & - & (a, 1.0) \\
2 & (d, 0.9) & 1.0 & 0.9 & 1.0 & (a, 1.0), (d, 0.9) \\
3 & (b, 0.7) & 0.7 & 0.9 & 0.9 & (a, 1.0), (d, 0.9), (b, 0.7) \\
4 & (e, 0.6) & 0.7 & 0.6 & 0.7 & (a, 1.0), (d, 0.9), (b, 0.7), (e, 0.6) \\
\hline
\end{array}
\]

We stop at Step 4, as we have three answers above the threshold \(\theta = 0.8\). However, this is not correct as \((d, 0.9)\) is not the score for \(d\). In fact, as there is still \((d, 0.4) \in \text{ans}_3(F, q_1)\) and, by applying the ranking aggregate function, \(\min(0.9, 0.4) = 0.4\), \((d, 0.4)\) is the score for \(d\) and, thus, \((d, 0.4)\) is not in the top-3.

The next example shows an even more serious problem.

**Example 4.10.** Suppose that \(r(q, O) = \{q_1, q_2\}\), where

\[
q_1 := q(x)[s] \leftarrow R(x, y)[s_1], \text{GroupBy}(x, y), \text{OrderBy}(s = \text{SUM}[s_1])
\]

\[
q_2 := q(x)[s] \leftarrow P(x, y)[s_1], \text{GroupBy}(x, y), \text{OrderBy}(s = \text{SUM}[s_1])
\]

and that we are looking for the top-1 answer. Assume that the answers are

\[
\text{ans}(F, q_1) = \{(a, 1.0), (b, 0.4), (e, 0.3)\}
\]

\[
\text{ans}(F, q_2) = \{(b, 0.9), (e, 0.2), (a, 0.1)\}
\]

Hence, the score for tuple \(a\) should be \(1.0 + 0.1 = 1.1\), the score for \(b\) should be \(0.9 + 0.4 = 1.3\), and the score for \(e\) should be \(0.3 + 0.2 = 0.5\). Therefore, the top-1 answer is \(b\) with score 1.3. Now, if we would rely on submitting just a top-1 query to the underlying database engine, as we do for the current DTA, we get \(\text{ans}_1(F, q_1) = \{(a, 1.0)\}\) and \(\text{ans}_1(F, q_2) = \{(b, 0.9)\}\). Then the score for tuple \(a\) would be 1.0, while the score for \(b\) would be 0.9 and, thus, not only we get incorrect scores, but also more importantly we get a wrong ranking result (tuple \(a\) would be top-1).
As shown in Examples 4.9 and 4.10, in presence of ranking aggregates, not yet processed tuples in top-k answers, may affect the final score. This is the case for aggregation functions \( \@ \in \{ \text{MIN, AVG, SUM} \} \), but not for \( \@ = \text{MAX} \). In fact, we have the following.

**Case \( \@ = \text{MAX} \).** In this case, no modification to the DTA is required as the DTA implicitly retrieves the maximal score for each tuple, i.e.

**Proposition 4.11.** Given \( K = (\mathcal{F}, \mathcal{O}, \mathcal{M}) \) and a disjunctive query \( q \) with ranking aggregate \( \text{MAX} \) only, then the DTA computes \( \text{ans}_k(K,q) \) in (sub) linear time w.r.t. the size of \( \mathcal{F} \).

\( \square \)

It remains to address the cases \( \@ \in \{ \text{MIN, AVG, SUM} \} \).

In the following, for all \( q_j \in r(q, \mathcal{O}) \), consider the answer set \( L_j = \text{ans}(\mathcal{F},q_j) \) \((1 \leq j \leq n = |r(q, \mathcal{O})|)\). Let \( c_i \) be a tuple and assume that it occurs in the top \( r \) lists \( L_{i_1}, \ldots, L_{i_r} \) with score \( s_{i_1}, \ldots, s_{i_r} \). We will say that the definitive score of \( c_i \) is \( \langle s_{i_1}, \ldots, s_{i_r} \rangle \). Note that, \( \langle c, s \rangle \in \text{ans}(K,q) \) is an answer for \( q \) iff \( s \) is the definitive score of \( c \).

In case we are interested in the top-k answers to a query \( q \), Example 4.10 points out that if we consider \( L_j^k = \text{ans}_k(\mathcal{F},q_j) \) in place of \( L_j = \text{ans}(\mathcal{F},q_j) \), we may not necessarily be able to compute from the top-k ranked lists \( L_j^k \) the definitive score of a tuple \( c \) occurring in some \( L_j^k \). Of course, if \( c \) occurs in all top-k ranked lists \( L_1^k, \ldots, L_n^k \) with score \( s_1, \ldots, s_n \) then \( s = \text{definitive score of } c \).

In the following, let \( gw \) a non-negative integer denoting a value called *group window*. For positive integer \( i \geq 1 \), with \( \text{ans}_{i,gw}(\mathcal{F},q) \) we will denote the ranked list of \( gw \) answers of \( q \), ranked between position \( (i - 1) \cdot gw + 1 \) and \( i \cdot gw \), i.e.

\[
\text{ans}_{i,gw}(\mathcal{F},q) = \text{ans}_{(i-1),gw}(\mathcal{F},q) \setminus \text{ans}_{i,gw}(\mathcal{F},q),
\]

where \( \text{ans}_0(\mathcal{F},q) = \emptyset \). For instance, for \( gw = 10 \), \( \text{ans}_{1,10}(\mathcal{F},q) \) are the top-10 answers to \( q \), while \( \text{ans}_{2,10}(\mathcal{F},q) \) are the answers to \( q \) ranked between position \( [11, \ldots, 20] \). We will use \( \text{ans}_{i,gw}(\mathcal{F},q) \) to incrementally retrieve a successive ordered group of \( gw \) answers to \( q \).

**Case \( \@ \in \{ \text{MIN, AVG, SUM} \} \).**

1. Initialization: let \( n = |r(q, \mathcal{O})| \), for all \( q \in r(q, \mathcal{O}) \), let \( i_q = 1 \), \( L_q = \emptyset \) and let \( \theta = \top \). Compute for all \( q \in r(q, \mathcal{O}) \), the top-\( gw \) answers and let \( L_q = \text{ans}_{gw}(\mathcal{F},q) \), using top-\( gw \) rank-based relational database engine. We will assume that the tuples in the top-\( gw \) answer set \( \text{ans}_{gw}(\mathcal{F},q_i) \) are sorted in decreasing order with respect to the score.

2. Then we process each answer set \( L_q \) according to some criteria (e.g., in parallel, or alternating fashion, or by selecting the next tuple from the

---

\(^4\)We point out that \( \text{ans}_{i,gw}(\mathcal{F},q) \) can be computed, in e.g. RankSQL, by means of a statement of the form **SELECT ... FROM ... GROUPBY ... ORDERBY ... DESC LIMIT gw OFFSET (i - 1) \cdot gw.**
answer set with highest threshold $\theta_q$ defined below), and top-down \(i.e\.,\) the higher scored tuples in $L_q$ are processed before the lower scored tuples in $L_q$.

(a) For each answer set $L_q$, let $\theta_{L_q}$ be the score of the last tuple processed in this set.

(b) If for a list $L_q$ we have processed the last tuple in it and we didn’t stop, then update $i_q := i_q + 1$ and append $\text{ans}_{i_q,gw}(F,q)$ to $L_q$ (\(i.e\.), retrieve the next $gw$ answers of $q$). If $\text{ans}_{i_q,gw}(F,q) = \emptyset$ do not process further $L_q$ and set $\theta_q = 0$. Such a list $L_q$ is called completed.

(c) For each tuple $c$ processed so far, let $\bar{L}_c$ be the set of (non-completed) ranked lists in which $c$ has not yet been processed so far:

(d) For each processed tuple $c$, let $l_c = [s_{i_1}, \ldots, s_{i_r}]$ be the actual list of $r$ scores computed so far for $c$ and let $s_c = \bar{l}_c$ be the actual score of $c$.

(e) If $c$ has been processed in all lists $L_{q_1}, \ldots, L_{q_n}$, or all lists in which $c$ haven’t be processed so far are completed, then $s_c$ is the definitive score of $c$;

(f) For each tuple $c$ processed so far, which has not yet a definitive score, we define the upper score of $c$, $\bar{s}_c$, as follows ($\bar{s}_c$ is the highest possible score $c$ may eventually achieve):

\begin{itemize}
  \item if $\bar{a} = \text{Min}$ then $\bar{s}_c$ is $s_c$
  \item if $\bar{a} = \text{AVG}$ then $\bar{s}_c$ is defined as follows. If there is no $L \in \bar{L}_c$ with $\theta_L > s_c$ then $\bar{s}_c := s_c$. Otherwise, $\bar{s}_c := (r \cdot s_c + \max_{L \in \bar{L}_c} \theta_L)/(r + 1)$. In summary,
    
    $\bar{s}_c := \max(s_c, (r \cdot s_c + \max_{L \in \bar{L}_c} \theta_L)/(r + 1))$.

  \item if $\bar{a} = \text{SUM}$ then $\bar{s}_c$ is
    
    $\bar{s}_c := s_c + \sum_{L \in \bar{L}_c} \theta_L$;
\end{itemize}

(g) As soon as at least $k$ tuples have been processed whose actual score is definitive and is at least equal to the definitive score or upper score of all other processed tuples, then halt.

(h) Let $Y$ be the set containing the $k$ tuples that have been processed with the highest definitive score. The output is then the set $\{\langle c, s_c \rangle \mid c \in Y\}$. This set is $\text{ans}_k(K,q)$.

Let us show that the stopping criteria works correctly and that we get the top-$k$ ranked tuples. Consider the top-k ranked tuples $\langle c_1, s_1 \rangle, \ldots, \langle c_k, s_k \rangle$, returned by the DTA above. Hence, $k$ tuples have been processed whose actual score is definitive and is at least equal to the definitive score or upper score of all other processed tuples. Now, consider a processed tuple $\langle c, s_c \rangle$, which didn’t make it into the top-k. If the score $s_c$ is definitive, it will not change any longer and, thus, the score of $c$ is below $s_k$. If $s_c$ is not definitive, then $c$ may possibly occur
in some of the ranked lists \( L \in \mathcal{L}_c \). Suppose that \( c \) will not eventually occur in some of the ranked lists \( L \in \mathcal{L}_c \). Then, the score \( s_c \) of \( c \) will be definitive and below \( s_k \). Otherwise, if \( c \) will eventually occur with score \( s_L \) in a ranked list \( L \in \mathcal{L}_c \), then we have to consider all three cases \( @ \in \{ \text{MIN}, \text{AVG}, \text{SUM} \} \). By the stopping criteria \( s_k \geq s_c \) holds.

**Case** \( @ = \text{MIN} \). As \( \min(s_c, s_L) \leq s_c \), the definitive score of \( c \) will be below \( s_k \).

**Case** \( @ = \text{AVG} \). We known that \( s_L \leq \max_{L \in \mathcal{L}_c} \theta_L \). Therefore, the definitive score of \( c \) cannot be greater than \( \max_{L \in \mathcal{L}_c} (r \cdot s_c + s_L)/(r + 1) \leq (r \cdot s_c + \max_{L \in \mathcal{L}_c} \theta_L)/(r + 1) = \bar{s}_c \leq s_k \).

**Case** \( @ = \text{SUM} \). We known that \( s_L \leq \max_{L \in \mathcal{L}_c} \theta_L \). Therefore, the definitive score of \( c \) is \( s_c + \sum_{L \in \mathcal{L}_c} s_L \leq s_c + \sum_{L \in \mathcal{L}_c} \theta_L = \bar{s}_c \leq s_k \), which concludes.

In the following, we call the DTA dealing with ranking aggregates DTA@. Hence, from the discussion above we have

**Proposition 4.12.** Given \( K = (F, O, M) \) and a disjunctive query \( q \) with ranking aggregates, then DTA@ computes \( ans_k(K, q) \) in (sub) linear time w.r.t. the size of \( F \).

**Example 4.13.** Consider Example 4.10. The execution of the DTA@ is shown below (\( gw = 1 \)). For each tuple we hold the actual score and the upper score. The actual score is in bold if it is definitive and, in this, case we omit the upper score.

<table>
<thead>
<tr>
<th>Step</th>
<th>Tuple</th>
<th>( \theta_{q_1} )</th>
<th>( \theta_{q_2} )</th>
<th>ranked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a, 1.0)</td>
<td>1.0</td>
<td></td>
<td>(a, 1.0, 2.0)</td>
</tr>
<tr>
<td>2</td>
<td>(6, 0.9)</td>
<td>1.0</td>
<td>0.9</td>
<td>(a, 1.0, 1.9), (6, 0.9, 1.9)</td>
</tr>
<tr>
<td>3</td>
<td>(6, 0.4)</td>
<td>0.4</td>
<td>0.9</td>
<td>(6, 1.3), (a, 1.0, 1.9)</td>
</tr>
<tr>
<td>4</td>
<td>(c, 0.2)</td>
<td>0.4</td>
<td>0.2</td>
<td>(6, 1.3), (a, 1.0, 1.2), (c, 0.2, 0.6)</td>
</tr>
</tbody>
</table>

We stop at Step 4, as we have tuple b with definitive score 1.3 that is at least equal to the upper score of all other processed tuples. Note that none of the tuples a, c may make it into the top-1.

5. Experiments

We conducted an experiment to evaluate some aspects of our query answering procedure. To this end we considered our top-k retrieval system SoftFacts [1, 32], that incorporates the proposed algorithm (however, in the SoftFacts system axioms are currently limited to the form \( Rl_1 \cap \ldots \cap Rl_m \subseteq Rr \)). To this end, we considered an ontology for describing Curriculum Vitae. It consists of 5115 axioms, 22 mapping statements and 2550 relations. The ontology has been used for “competence search” and is described in more details in [30, 31].

We considered size \( d = 500000 \) for the fact component consisting in automatically generated CVs and stored them into a database having 17 relational tables. We build several queries (see Appendix B), with/without scoring atom and with/without ranking aggregates. The queries have been submitted to the system for different values for \( k \) in case of top-k retrieval \( (k \in \{1, 10\}) \), and we measured for each query (time is measured in milliseconds)
1. the number of queries generated after the reformulation process (|r(q, O)|);
2. the number of reformulated queries after redundancy elimination (q_DB);
3. the time of the reformulation process (t_ref);
4. the time of the query redundancy elimination process (t_red);
5. the query answering time of the database, t_DBall, t_DB1, t_DB10, i.e. time to retrieve all, top-1 and top-10 answers from the database.

Note that the measures 1-4 do neither depend on the number d of CVs nor on the number k for top-k retrieval.

We run the experiments using our top-k retrieval system, with indexes on the keys of the relational database tables. The tests have been performed on a MacPro machine with Mac OS X 10.6.2, 2 x 3 GHz Dual-Core processor and 9 GB or RAM. The database is Postgres\footnote{http://www.postgres.org}, which has been modified by RankSQL.

Fig. 4 reports the measures 1-4, which are independent of the relational database. We may note that the size of the set of reformulated queries |r(q, O)| may be non negligible, that the redundancy elimination may remove almost one third of them and the time of the reduction phase is negligible, except for query 9, which generates a large set of queries.

In Fig. 5 we report the retrieval times statistics. We also fixed the group window to gw = 100.

Let us consider few comments about the results:

- overall, the response time is reasonable (almost fraction of second, except for query 9) taking into account the non negligible size of the ontology and the number of CVs;

- if the answer set is large, e.g., query 1, then there is a significant drop in response time, for the top-k case;

| Query | |r(q, O)| | q_DB | t_ref | t_red |
|-------|-----------------|----------|----------|----------|
| 1     | 1599            | 1066     | 1.723    | 0.010    |
| 2     | 69              | 46       | 0.016    | 0.001    |
| 3     | 18              | 12       | 0.006    | 0.001    |
| 4     | 1242            | 552      | 2.072    | 0.015    |
| 5     | 75              | 50       | 0.027    | 0.001    |
| 6     | 18              | 12       | 0.005    | 0.001    |
| 7     | 69              | 46       | 0.013    | 0.001    |
| 8     | 1242            | 552      | 1.242    | 0.013    |
| 9     | 5175            | 2300     | 17.850   | 0.058    |
| 10    | 108             | 48       | 0.229    | 0.003    |
| 11    | 18              | 12       | 0.004    | 0.001    |
| 12    | 828             | 552      | 0.794    | 0.005    |
| Average| 961.5          | 468.4    | 2.318    | 0.01     |
| Median | 91.5            | 49       | 0.128    | 0.002    |

Figure 4: Query reformulation statistics.
### 5. Retrieval Statistics

| Query | All | top-1 | top-10 | \(|\text{ans}(K,q)|\) | \(t_{\text{DBall}}\) | \(t_{\text{DB1}}\) | \(t_{\text{DB10}}\) |
|-------|-----|-------|-------|----------------|----------------|----------------|----------------|
| 1     | -   | 1,852 | 1,907 | > 297,195      | -              | 0,142          | 0,174          |
| 2     | -   | 0,083 | 0,180 | > 260,470      | -              | 0,050          | 0,106          |
| 3     | 18,005 | 0,136 | 0,198 | 12,222         | 17,947         | 0,038          | 0,128          |
| 4     | 35,879 | 2,149 | 2,377 | 2259           | 33,930         | 0,078          | 0,232          |
| 5     | 23,086 | 0,060 | 0,064 | 13,323         | 23,032         | 0,022          | 0,024          |
| 6     | 554,209 | 0,531 | 0,570 | 56293          | 554,185        | 0,513          | 0,537          |
| 7     | -   | 1,558 | 1,707 | > 260,470      | -              | 1,534          | 1,551          |
| 8     | 51,050 | 2,283 | 4,279 | 10,432         | 49,230         | 0,412          | 2,201          |
| 9     | 568,452 | 167,625 | 168,235 | 43462          | 550,792         | 150,038       | 150,370       |
| 10    | 260,607 | 3,072 | 3,082 | 37471          | 260,446         | 2,911          | 2,928          |
| 11    | 70,093 | 2,998 | 3,243 | 22247          | 70,072          | 2,971          | 3,227          |
| 12    | 56,535 | 47,189 | 47,617 | 8556           | 55,446          | 46,115         | 46,494         |
| AVG   | 215,898 | 17,935 | 18,260 | 1,3323         | 212,795         | 15,572         | 15,825         |

| Median | 51,050 | 1,705 | 1,807 | -              | 49,230          | 0,277          | 0,385          |

Figure 5: Retrieval statistics \(d = 500000\).

- for each query, the response time is increasing while we increase the number of retrieved records;
- the answering time of the database is increasing with the number of top-k results to be retrieved.

The important aspect we wanted to highlight with this experiment is that for some queries (1, 2, 7) we were not able to compute all answers as the answer set becomes too big to be held in main memory (within the current prototypical implementation), while the relative top-k problems have been solved easily. We also point to query 9. In that case, even if the database responds within 0.07 seconds/query in average, the large size of queries (2300) prevents a reasonable query answering time so far. In this case we have both a large answer set as well as a large set of reformulated queries. Also, if \(|r(q, \mathcal{O})|\) is small (\(\leq 100\)), then the query answering time is almost the same as the database retrieval time. All the test data can be accessed at [http://www.straccia.info/software/SoftFacts/Experiments/URLTest.zip](http://www.straccia.info/software/SoftFacts/Experiments/URLTest.zip).

### 6. Related Work

While there are many works addressing the top-k problem for vague queries in databases (see e.g., [14] for a survey), little is known for the corresponding problem in knowledge representation and reasoning. For instance, [26] considers non-recursive fuzzy logic programs in which the score combination function is a function of the score of the atoms in the body only (no scoring predicates are allowed). The work [25] considers non-recursive fuzzy logic programs as well, though the score combination function may consider scoring predicates. However, a score combination function is allowed in the query rule only. We point out that in the case of non-recursive rules and/or axioms, we may rely on
a query rewriting mechanism, which, given an initial query, rewrites it, using rules and/or axioms of the KB, into a set of new queries until no new query rule can be derived (this phase may require exponential time relative to the size of the KB). The obtained queries may then be submitted directly to a top-k retrieval database engine. The answers to each query are then merged using the disjunctive threshold algorithm given in [25]. The works [24, 26] address the top-k retrieval problem for the description logic DL-Lite only, though recursion is allowed among the axioms. Again, the score combination function may consider scoring predicates. However, a score combination function is allowed in the query only. The work [32] shows an application of top-k retrieval to the case of multimedia information retrieval by relying on a fuzzy variant of DLR-Lite and is subsumed by the work here (as no scoring function is allowed in the axioms, no scoring function is allowed in the query language), while [30, 31] report its use in the context of “competence search” (see also [19, 20, 21]). Finally, [27] addresses the top-k retrieval for general (recursive) fuzzy LPs, though no scoring predicates are allowed, which however are allowed in [15]. Closest to our work are [15, 27]. In fact, our work extends [15, 27] by allowing scoring predicates and scoring aggregate functions, which have the effect that the threshold mechanism designed in [15, 27] does not work anymore and no DTA has been provided. Furthermore, in this paper, we made an effort to plug-in current top-k database technology, while [27] does not and provides an ad-hoc solution.

We also refer the reader to [9] for a crisp DL-Lite implementation and application, and to [16] for a fuzzy DL-Lite language in the spirit of [26] (and less expressive language than ours) but not addressing specifically the top-k retrieval problem and ranking aggregates. [16] also reports some experiments on the LUBM benchmark [6] which relies on a small ontology, around 80 axioms (around 40 atomic concepts, 20 binary relations). Therefore, $r(q, O)$ is of almost of negligible size and, thus, so will be the query answer time (database engine retrieval time).

Finally, we point out the difference to so-called fuzzy relational databases (see, e.g. [5, 27]), in which values of a column may be fuzzy (e.g., the age of a person is “young”), whereas this is not the case here and in which the top-k retrieval problem is not addressed specifically.

7. Summary and Outlook

The top-k retrieval problem is an important problem in logic-based languages for the Semantic Web. We have addressed this issue by describing an ontology mediated top-k information retrieval framework over relational databases. Indeed, 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 a mapping component. The results of a query may be ranked according to some scoring

\[ \text{http://swat.cse.lehigh.edu/projects/lubm/} \]
function and aggregation functions. We have illustrate the representation and the query language, and the query answering algorithm. We have also provided an experiment.

Concerning future work, we recall that the query reformulation step is a generalisation of [8, 24, 26] to our case. Recent works, such as [17] propose an alternative query rewriting method than [8] for the crisp case. We plan to investigate how and whether this method can be adapted to our case as well and how it impacts the size of the rewritings. Concerning the experimental part, we are planning extensive tests with the SoftFacts system also with other datasets, which will be the subject of a separate work (which also includes several practical optimization methods, which are beyond the scope of this paper).

References


Papers, number 4925 in Lecture Notes in Computer Science, pages 26–47. Springer Verlag, 2008.


Appendix A. Relation to OWL QL

The syntax of OWL QL expressions is essentially as follows. In the following, $A$ is an atom (unary predicate), $R$ is a role (binary predicate) and $R^-$ is the inverse of role $R$. Symbols may have a subscript.

**Assertions.** Facts are expressions of the form $A(c)$ and $R(c,d)$, specifically,

$$\text{ClassAssertion}(A,c), \text{ObjectPropertyAssertion}(R,c,d)$$

**Class Expressions.** In OWL 2 QL, there are two types of class expressions. The subClass production defines the class expressions that can occur as sub class expressions in SubClassOf axioms, and the superClass production defines the classes that can occur as super class expressions in SubClassOf axioms. Their syntax is:

$$\text{SubClass} \rightarrow A \mid \exists R$$

$$\text{SuperClass} \rightarrow A \mid \text{SuperClass}_1 \cap \text{SuperClass}_2 \mid \neg \text{SubClass} \mid \exists R.A$$

Here, $\exists R.A$ is a qualified existential quantification on roles and in FOL, it corresponds to the formula $\exists y. R(x,y) \land A(y)$, while $\exists R$ is an unqualified existential quantification on roles that corresponds to the formula $\exists y. R(x,y)$.

**Class Axioms.** The class axioms of OWL 2 QL are of the following form

$$\text{SubClassOf(SubClass,SuperClass)}$$
$$\text{EquivalentClasses(SubClass}_1,\text{SubClass}_2)$$
$$\text{DisjointClasses(SubClass}_1,\text{SubClass}_2)$$
Property Axioms. The property axioms of OWL 2 QL are of the following form

- SubObjectPropertyOf($R_1, R_2$)
- EquivalentObjectProperties($R_1, R_2$)
- DisjointObjectProperties($R_1, R_2$)
- InverseObjectProperties($R_1, R_2$)
- ObjectPropertyDomain($R, \text{SuperClass}$)
- ObjectPropertyRange($R, \text{SuperClass}$)
- ReflexiveObjectProperty($R$)
- SymmetricObjectProperty($R$)
- AsymmetricObjectProperty($R$)

On the translation of OWL 2 QL to DL-Lite$_R$. We recap here the translation of OWL 2 QL to DL-Lite$_R$ [10]. We recall that in DL-Lite$_R$ class and property (role) expressions have the following syntax ($A$ is an atomic concept, $P$ is a role and $P^-$ is its inverse):

$$
\begin{align*}
B &\rightarrow A \mid \exists R \\
C &\rightarrow B \mid \neg B \\
R &\rightarrow P \mid P^- \\
E &\rightarrow R \mid R^-
\end{align*}
$$

Here, $B$ is a so-called basic concept. $\exists R$ is an unqualified existential quantification on roles (in FOL, it corresponds to the formula $\exists y. R(x, y)$) and in our setting is simulated as the projection of $R$ on the first column, i.e. $R[1]$. $C$ denotes a (general) concept, which can be a basic concept or its negation, and $E$ denotes a (general) role, which can be a basic role or its negation.

Inclusion axioms are of the form

$$
B \sqsubseteq C \quad \text{and} \quad R \sqsubseteq E
$$

Precisely we refer to assertions of the form $B_1 \sqsubseteq B_2$ (resp. $R_1 \sqsubseteq R_2$) as positive inclusion assertions (PIs), and to assertions of the form $B_1 \sqsubseteq \neg B_2$ (resp. $R_1 \sqsubseteq \neg R_2$) as negative inclusion assertions (NIs). Therefore general concepts (resp., roles) may only occur on the right-hand side of inclusion assertions.

Note that there are some differences among DL-Lite$_R$ and OWL 2 QL:

1. existential quantification to a class $\exists R.C$;
2. symmetric property axioms and asymmetric property axioms;
3. reflexive property axioms and irreflexive property axioms.

Notice that, although 1. and 2. are not natively supported by DL-Lite$_R$, they are actually expressible in DL-Lite$_R$ by suitably processing the intensional level of the ontology. Conversely, reflexivity and irreflexivity axioms are brand new features and require new inference mechanisms.

Native handling of qualified existential quantification. It is well known that an axiom $\tau$ of the form

$$
B \sqsubseteq \exists R.A
$$
can be replaced with the transformation
\[ \tau \mapsto \{ B \sqsubseteq \exists R_{aux}, R_{aux} \sqsubseteq R, \exists R_{aux}^{-} \sqsubseteq A \} , \]
where \( R_{aux} \) is a new role. We point out that [10] provides also a query reformulation method that does not require the translation.

**Native handling of symmetric and asymmetric object property axioms.**
1. each SymmetricObjectProperty\((R)\) is managed by adding \( R \sqsubseteq R^{-} \) to the DL component;
2. each AsymmetricObjectProperty\((R)\) is managed by adding \( R \sqsubseteq \neg R^{-} \) to the DL component;

**Handling of Reflexive and irreflexive object property axioms.** It has been observed in [10] that since (i) the asymmetry axiom on a property implies the irreflexivity axiom on the same property; and (ii) the asymmetry axiom influences only the consistency check on the ontology; the irreflexivity axiom influences only the consistency check, too. [10] shows how to deal with irreflexivity axioms in the consistency checking phase.

On the other hand, a reflexive role declarations impact the query reformulation phase, cannot be represented in DL-Lite\(_R\) and they have to dealt with specifically in the reformulation algorithm, though the solution is easy. Essentially, a reflexive role \( R \) means that \( R(x,x) \) is always true, we have to appropriately drop any occurrence of \( R(x,x) \) occurring in a query during the query reformulation procedure. We refer the reader to [10] for the details.

**Mapping OWL 2 QL statements.** We now show how to map OWL 2 QL statements into DL\(_{Lite}\). Assertions (class assertions and object property assertions) are mapped directly as DL-Lite\(_R\) facts. Concerning roles, \( R^{-}^{-} \) is mapped into \( R \) and, thus, OWL 2 QL object properties are DL-Lite\(_R\) roles.

Concerning class axioms, we have:

**SubClassOf:** an axiom SubClassOf\((SubClass, SuperClass)\) is transformed according the following rules:

\[
\begin{align*}
\text{SubClassOf}(SubClass, SuperClass_1 \sqcap SuperClass_2) & \mapsto \text{SubClassOf}(SubClass, SuperClass_1), \text{SubClassOf}(SubClass, SuperClass_2) \\
\text{SubClassOf}(SubClass_1, SubClass_2) & \mapsto \text{SubClass}_1 \sqsubseteq \text{SubClass}_2 \\
\text{SubClassOf}(SubClass_1, \neg \text{SubClass}_2) & \mapsto \text{SubClass}_1 \sqsubseteq \neg \text{SubClass}_2 \\
\text{SubClassOf}(SubClass, \exists R.A) & \mapsto \text{SubClass} \sqsubseteq \exists R.A
\end{align*}
\]
**EquivalentClasses:** an axiom \( \text{EquivalentClasses}(\text{SubClass}_1, \text{SubClass}_2) \) is transformed as

\[
\text{EquivalentClasses}(\text{SubClass}_1, \text{SubClass}_2) \mapsto \text{SubClassOf}(\text{SubClass}_1, \text{SubClass}_2), \text{SubClassOf}(\text{SubClass}_2, \text{SubClass}_1)
\]

**DisjointClasses:** an axiom \( \text{DisjointClasses}(\text{SubClass}_1, \text{SubClass}_2) \) is transformed as

\[
\text{DisjointClasses}(\text{SubClass}_1, \text{SubClass}_2) \mapsto \text{SubClassOf}(\text{SubClass}_1, \neg \text{SubClass}_2)
\]

Concerning property axioms, we have:

**SubObjectPropertyOf:** an axiom \( \text{SubObjectPropertyOf}(R_1, R_2) \) is transformed as

\[
\text{SubObjectPropertyOf}(R_1, R_2) \mapsto R_1 \subseteq R_2
\]

**EquivalentObjectProperties:** an axiom \( \text{EquivalentObjectProperties}(R_1, R_2) \) is transformed as

\[
\text{EquivalentObjectProperties}(R_1, R_2) \mapsto R_1 \subseteq R_2, R_2 \subseteq R_1
\]

**DisjointObjectProperties:** an axiom \( \text{DisjointObjectProperties}(R_1, R_2) \) is transformed as

\[
\text{DisjointObjectProperties}(R_1, R_2) \mapsto R_1 \subseteq \neg R_2
\]

**InverseObjectProperties:** an axiom \( \text{InverseObjectProperties}(R_1, R_2) \) is transformed as

\[
\text{InverseObjectProperties}(R_1, R_2) \mapsto R_2 \subseteq R_1
\]

**ObjectPropertyDomain:** an axiom \( \text{ObjectPropertyDomain}(R, \text{SuperClass}) \) is transformed as

\[
\text{ObjectPropertyDomain}(R, \text{SuperClass}) \mapsto \text{SubClassOf}(\exists R, \text{SuperClass})
\]

**ObjectPropertyRange:** an axiom \( \text{ObjectPropertyRange}(R, \text{SuperClass}) \) is transformed as

\[
\text{ObjectPropertyRange}(R, \text{SuperClass}) \mapsto \text{SubClassOf}(\exists R^-, \text{SuperClass})
\]

**ReflexiveObjectProperty and IrreflexiveObjectProperty:** cannot be mapped into DL-Lite \( R \) and needs special treatment, as pointed out before;

**SymmetricObjectProperty:** an axiom \( \text{SymmetricObjectProperty}(R) \) is transformed as

\[
\text{SymmetricObjectProperty}(R) \mapsto R \subseteq R^-
\]

**AsymmetricObjectProperty:** an axiom \( \text{AsymmetricObjectProperty}(R) \) is transformed as

\[
\text{AsymmetricObjectProperty}(R) \mapsto R \subseteq \neg R^-
\]
On the translation of OWL 2 QL to our language. We first translate OWL 2 QL into DL-Lite$_R$ axioms. Then, we drop all axioms involving atom negation or role negation and then use the transformation

$$\exists R \mapsto R[1] \quad \exists R^- \mapsto R[2] \quad R^- \mapsto R[2,1].$$

We point out that:

- we do not support atom negation or role negation and, thus, constructs such as disjointness of classes and roles and asymmetry of roles. However, please note that these play a role in KB consistency checking only, but do not play any role in query answering. Therefore, to what concerns query answering, the drop of such axioms is harmless;

- as for DL-Lite$_R$, we do not support role reflexivity. However, we can easily include this by using exactly the same method as described in [10].

Appendix B. Queries of the experiment

The queries used in the experiment are the followings (for illustrative purposes, for some queries we provide also the encoding in our language):

1. Retrieve CV’s with knowledge in Engineering Technology

   $q(id, lastName, hasKnowledge, Years) \leftarrow profileLastName(id, lastName),$
   $\text{hasKnowledge}(id, classID, Years, Type, Level),$
   $\text{knowledgeName(classID, hasKnowledge),}
   \text{Engineering and Technology(classID)}$

2. Retrieve CV’s referred to candidates with degree in Engineering

3. Retrieve CV’s referred to candidates with knowledge in Artificial Intelligence and degree final mark no less than 100/110

4. Retrieve CV’s referred to candidates with knowledge in Artificial Intelligence, degree in Engineering with final mark no less than 100/110

5. Retrieve CV’s referred to candidates experienced in Information Systems (no less than 15 years), with degree final mark no less than 100

6. Retrieve top-k CV’s referred to candidates with knowledge in Artificial Intelligence and degree final mark scored according to $rs(mark; 100, 110)$

   $q(id, lastName, degreeName, mark, hasKnowledge, years) \leftarrow$
   $\text{profileLastName(id, lastName),}
   \text{hasDegree(id, degreeId, mark),}$
   $\text{degreeName(degreeId, degreeName),}
   \text{hasKnowledge(id, classID, years, type, level),}$
   $\text{knowledgeName(classID, hasKnowledge),}
   \text{Artificial Intelligence(classID),}$
   $\text{OrderBy(s = rs(mark; 100, 110))}$

7. Retrieve CV’s referred to candidates with degree in Engineering and final mark scored according to $rs(mark; 100, 110)$

36
8. Retrieve top-k CV’s referred to candidates with knowledge in Artificial Intelligence, degree in Engineering with final mark scored according to \( rs(\text{mark}; 100, 110) \)

9. Retrieve CV’s referred to candidates with knowledge in Information Systems, degree in Engineering and with degree final mark and years of experience both scored according to \( rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot 0.6 \).

10. Retrieve CV’s referred to candidates with good knowledge in Artificial Intelligence, and with degree final mark, years and level of experience scored according to \( rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot \text{pref}(\text{level}; \text{Good}/0.6, \text{Excellent}/1.0) \cdot 0.6 \);

\[ q(\text{id}, \text{lastName}, \text{degreeName}, \text{mark}, \text{hasKnowledge}, \text{years}, \text{kType}) = \]
\[ \text{profileLastName}(\text{id}, \text{lastName}), \]
\[ \text{hasDegree}(\text{id}, \text{degreeId}, \text{mark}), \]
\[ \text{degreeName}(\text{degreeId}, \text{degreeName}), \]
\[ \text{hasKnowledge}(\text{id}, \text{classID}, \text{years}, \text{type}, \text{level}), \]
\[ \text{knowledgeLevelName}(\text{level}, \text{kType}), \]
\[ \text{Good}(\text{level}), \]
\[ \text{knowledgeName}(\text{classID}, \text{hasKnowledge}), \]
\[ \text{ArtificialIntelligence}(\text{classID}), \]
\[ \text{OrderBy}(s = rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot \text{pref}(\text{level}; \text{Good}/0.6, \text{Excellent}/1.0) \cdot 0.6) \]

11. Retrieve CV’s referred to candidates with knowledge in Artificial Intelligence, grouped by \( \text{MAX}[rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot 0.6] \)

\[ q(\text{id}, \text{lastName}) = \text{profileLastName}(\text{id}, \text{lastName}), \]
\[ \text{hasDegree}(\text{id}, \text{degreeId}, \text{mark}), \]
\[ \text{hasKnowledge}(\text{id}, \text{classID}, \text{years}, \text{type}, \text{level}), \]
\[ \text{ArtificialIntelligence}(\text{classID}), \]
\[ \text{GroupBy}(\text{id}, \text{lastname}), \]
\[ \text{OrderBy}(s = \text{MAX}[rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot 0.6]) \]

12. Retrieve CV’s referred to candidates with knowledge in Artificial Intelligence, a degree in Engineering and grouped by \( \text{AVG}(rs(\text{mark}; 100, 110) \cdot 0.4 + rs(\text{years}; 15, 25) \cdot 0.6) \)

Queries 1-5 are crisp. As each answer has score \( \top \), we would like to verify whether there is a retrieval time difference between retrieving all records, or just the \( k \) answers. The other queries are top-k queries. In query 9, we show an example of score combination, with a preference on the number of years of experience over the degree’s mark, but scores are summed up. In query 10, we use the preference scoring function \( \text{pref}(\text{level}; \text{Good}/0.6, \text{Excellent}/1.0) \) that returns 0.6 if the level is good, while returns 1.0 if the level is excellent. In this way we want to privilege those with an excellent knowledge level over those with a good level of knowledge. Queries 11 and 12 use ranking aggregates, which were not supported at the time of the experiments in \([30, 31]\).