Towards Vague Query Answering in Logic Programming for Logic-based Information Retrieval

Umberto Straccia

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

Abstract. We address a novel issue for logic programming, namely the problem of evaluating ranked top-k queries. The problem occurs for instance, when we allow queries such as "find cheap hotels close to the conference location" in which vague predicates like cheap and close occur. Vague predicates have the effect that each tuple in the answer set has now a score in [0,1]. We show how to compute the top-k answers in case the set of facts is huge, without evaluating all the tuples.

Keywords: Logic Programming, Fuzzy, Top-k retrieval

1 Introduction

In this paper we address a novel issue for *Logic Programs* (LPs) with a huge set of facts, namely the problem of *evaluating ranked top-k queries*. In classical logic programming, an answer to a query is a set of tuples that satisfy a query. Each tuple may or may not satisfy the predicates in the query. However, very often the information need of a user involves so-called *fuzzy/vague predicates*. For instance, a user may have the following information need: "Find *cheap* hotels *near* to the conference location". Here, $cheap$ and $near$ are fuzzy predicates. Unlike the classical case, tuples satisfy now these predicates to a score (usually in $[0, 1]$). 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 new challenge in case we have to deal with a huge amount of facts. Indeed, virtually every tuple may satisfy a query with a non-zero score and, thus, has to be ranked. Of course, computing all these scores, ranking them and then selecting the top-k ones is not feasible in practice.

In this work, we address the top-k retrieval problem for Datalog. At the extensional level, each fact may have a score, while at the intentional level rules describe the domain of application. Queries are conjunctive queries in which vague predicates may occur.

2 Preliminaries

The formalism we consider is defined as follows. The *score space* (i.e. *truth space*) is $\mathcal{S} = [0, 1]$. We anticipate informally that an interpretation will assign a score (i.e. truth)

to a ground atom and that the answers to a query (i.e. ground instances of an atom) will be ranked (in decreasing order) according to their scores.

A *term* is either a *variable* or a *constant*.

Let $\mathcal{V}_E$ and $\mathcal{V}_I$ be disjoint sets of n-ary *extensional* and *intentional* predicate symbols, respectively. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and all t_j are terms. An atom is *ground* if no variable occurs in it. A *logic program* $\mathcal{P}$ is made out of an *extensional database* (EDB), $\mathcal{P}_E$, and an *intentional database* (IDB), $\mathcal{P}_I$. The extensional database is a set of *facts* of the form $r(\mathbf{c}) \leftarrow b$, where $r(\mathbf{c})$ is a ground atom, r is an extensional predicate and $b \in \mathcal{S}$ is a score value. The intuition here is that b is the assigned score to tuple $\mathbf{c}$ in relation r. For convenience, for each n-ary extensional predicate r, we represent the facts $r(c_1, \ldots, c_n) \leftarrow b$ in $\mathcal{P}$ by means of a relational $n+1$-ary table T_r, containing the records $\langle c_1, \ldots, c_n, b \rangle$. Thus, the table contains all the instances of r together with their scores. We assume that there cannot be two records $\langle c_1, \ldots, c_n, b_1 \rangle$ and $\langle c_1, \ldots, c_n, b_2 \rangle$ in T_r with $b_1 \neq b_2$ (in case their are, we remove the one with the lower score). Each table is sorted in descending order with respect to the scores. Usually, the score of a tuple in a relation has been computed (possibly off-line) by a specific system. For instance, we may have an underlying image retrieval system that for each identified object in an image has m scores, one for each of the m attributes (see, e.g. [8]).

The intentional database is a set of *rules* in which all variables in the head do also appear in the rule body. To facilitate the reading, we first give an example of rule with its intended meaning and then provide the formal definition. Assume we would like to represent the set of good conference hotels $q(x)$, which are hotels close to the conference location. Assume that we have a relational table of hotels, their location and their price, a table of conferences and their location, a distance table reporting the distance among two locations, and two tables for the extensional predicates, $cheap$ and $close$, whose instances are scored with respect to the following functions, where the former depends on the price, while the latter depends on the distance: $s_{cheap}(p) = \max(0, 1 - p/200)$, and $s_{close}(d) = \max(0, 1 - d/2000)$. Then following rule may be a candidate rule: for a given conference c

$$q(h) \leftarrow \min[hotel(h, hLoc, price), conference(c, cLoc), \\ distance(hLoc, cLoc, d)] \cdot cheap(price) \cdot close(d).$$

Essentially, in the above rule, the score of $q(h)$ is determined by taking the $\min$ of the first three atoms and then take the product of it with the last expression. This is similar as it happens in top-k retrieval in the context of relational databases [3, 4, 6]: the data is represented in relational tables and the SQL query language is extended to allow to express a scoring function, which may use the values occurring in the retrieved records, to compute the final score of the retrieved record.

So, let $\mathcal{F}$ be a set of total *score combination functions*, i.e. computable functions [1] $f\colon [0,1]^n \rightarrow [0,1]$ used to manipulate score values, e.g. $\min, \max, \cdot, +, \ldots$. Score combination functions will have a fixed interpretation, i.e. we will consider them as *built-in* functions. Then an intentional database is a set of rules of form $p(\mathbf{x}) \leftarrow f(A_1, \ldots, A_n)$, where (i) p is an intentional predicate; (ii) A_i is an atom $q(\mathbf{t})$ and $\mathbf{t}$ is a tuple of terms (q is either an intentional or an extensional predicate symbol); and (iii) f is a score

[1] With computable we mean that for any input, the value of f can be determined in finite time.

combination function, which is assumed to be monotone in its arguments. Note that the extensional predicates do not occur in the head of rules of the intentional database. Essentially, we do not allow that the fact predicates occurring in $\mathcal{P}_E$ can be redefined by $\mathcal{P}_I$. A *classical rule* is one in which f is $\min$.

From the semantics point of view, the *Herbrand universe* $H_\mathcal{P}$ of $\mathcal{P}$ is the set of constants appearing in $\mathcal{P}$. If there is no constant symbol in $\mathcal{P}$ then consider $H_\mathcal{P} = \{c\}$, where c is an arbitrary chosen constant. The *Herbrand base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of ground instantiations of atoms appearing in $\mathcal{P}$ (ground instantiations are obtained by replacing all variable symbols with constants of the Herbrand universe). Let $\mathcal{P}^*$ be the set of ground rule instantiations obtained from $\mathcal{P}$. Note that $\mathcal{P}^*$ is always *finite*. An *interpretation* I is a partial mapping from intentional and extensional atoms to $[0, 1]$ (we recall that for a constant c, $I(c) = c$). Note that, as I may be a partial function, some atoms may not have a score. Alternatively, we may assume I to be a total function. We use the former formulation to distinguish the case where a tuple $\mathbf{c}$ may be retrieved, though the score is 0, from the tuples which do not satisfy the query and, thus, would be not retrieved. In particular, if a tuple does not belong to an extensional relation then its score is assumed to be undefined, while if I is total, then the score of this tuple would be 0. We denote with $def(I)$ the set of ground atoms on which I is defined. We say that I is a *model* of $\mathcal{P}$, denoted $I \models \mathcal{P}$, iff for all facts $A \leftarrow b \in \mathcal{P}_E$, $I(A) \geq b$ and for all rules $A \leftarrow f(A_1, \ldots, A_n) \in \mathcal{P}_I$ such that all $I(A_i)$ are defined, $I(A) \geq f(I(A_1), \ldots, I(A_n))$ holds (note that the function $f \in \mathcal{F}$ has a fixed interpretation, which we identify with f itself). We say that an interpretation I is a *minimal model* of $\mathcal{P}$ iff $I \models \mathcal{P}$ and for any other model J of $\mathcal{P}$, $def(I) \subseteq def(J)$ and for all $A \in def(I)$ $I(A) \leq J(A)$ holds. [2] It is not difficult to see that there is an unique minimal model $M_\mathcal{P}$ of $\mathcal{P}$. The proof is based on the existence of a partial monotone immediate consequence operator $T_\mathcal{P}$, whose fixed-points are models of $\mathcal{P}$: for any ground atom $A \in B_\mathcal{P}$ $T_\mathcal{P}(I)(A) = \max\{I(\varphi) \mid A \leftarrow \varphi \in \mathcal{P}^*\}$, where $\max \emptyset$ is undefined ($\mathcal{P}^*$ is finite, so $\max$ can be used).

A *query* is an intentional predicate symbol q. The *answer set* of q w.r.t. $\mathcal{P}$ is defined as the set $ans(q, \mathcal{P})$ of tuples $\langle \mathbf{c}, s\rangle \in H_\mathcal{P} \times \ldots \times H_\mathcal{P} \times [0, 1]$ such that $M_\mathcal{P}(q(\mathbf{c})) = s$ (the score of $\mathbf{c}$ is s in the minimal model).

Example 1. Given the logic program $\mathcal{P} = \{(q(x) \leftarrow 0.5 \cdot (p(x) + r(x))), (p(a) \leftarrow 0.9), (p(b) \leftarrow 0.2), (r(b) \leftarrow 0.4)\}$ then $M_\mathcal{P}(p(a)) = 0.9$, $M_\mathcal{P}(p(b)) = 0.2$, $M_\mathcal{P}(r(b)) = 0.4$, $M_\mathcal{P}(q(b)) = 0.3$ and $ans(q, \mathcal{P}) = \{\langle b, 0.3\rangle\}$, while $ans(p, \mathcal{P}) = \{\langle a, 0.9\rangle, \langle b, 0.2\rangle\}$. If $M_\mathcal{P}$ has to be a total interpretation then additionally $M_\mathcal{P}(r(a)) = 0$, $M_\mathcal{P}(q(a)) = 0.45$ and $\langle a, 0.45\rangle \in ans(q, \mathcal{P})$.

Example 2. Given the logic program $\mathcal{P} = \{(q(x) \leftarrow (q(x) + 1)/2), (p(a) \leftarrow 0.4)\}$ then $M_\mathcal{P}(p(a)) = 0.4$, $M_\mathcal{P}(q(a)) = 1$ and $ans(q, \mathcal{P}) = \{\langle a, 1\rangle\}$. Note that $\mathcal{P}$ exhibits a well-known behaviour, requiring ω steps of $T_\mathcal{P}$ iterations to obtain the minimal model [5, 13].

The basic reasoning service that mainly concerns us is:

[2] The least interpretation is unique and is $\mathtt{I}_\perp$, where $def(\mathtt{I}_\perp) = \emptyset$, i.e. $\mathtt{I}_\perp$ is undefined everywhere.

Top-k retrieval: Given $\mathcal{P}$, retrieve the top-k ranked tuples of the answer set of q w.r.t. the score, denoted $ans_k(q, \mathcal{P}) = \mathtt{Top}_k(ans(q, \mathcal{P}))$.

We note that retrieving the top-k answers of an extensional predicate symbol r is trivial as we have just to retrieve the first k tuples in the relational table T_r associated to r. Hence, we restrict top-k retrieval to intentional predicates only.

3 Top-k information retrieval

We next provide an incremental top-down top-k query answering algorithm. Note that, as Example 2 shows, computing an answer (and, thus, the top-k answers) may not be possible in finite time in general. A usual way to overcome this situation is to rely on *bounded* score combination functions f, i.e. for all i, $f(x_1, \ldots, x_n) \leq x_i$. In this case it can be shown that the least-fixed point is reached after a finite number of T_p iterations [9].

To start with, we use the usual relation "directly depends on" among predicate symbols, i.e. given $\mathcal{P}$, we say that predicate symbol p *directly depends on* predicate symbol q if there is a rule in $\mathcal{P}$ such that p occurs in the head of it and q occurs in the body of it. The relation *depends on* is the transitive closure of "directly depends on". The *dependency graph* of $\mathcal{P}$ is a directed graph where nodes are predicate symbols and the set of edges is the "directly depends on" relation. The program is *recursive* if there is a cycle in the dependency graph (i.e. there is p depending on p). We also say that $\mathcal{P}$ is *deterministic* if for each intentional predicate symbol p there is at most one rule in $\mathcal{P}$ having p in its head.

A practical useful case is when the logic program contains only classical rules, except for the rules having the query in the head. This depicts the scenario when a top-k query involving vague predicates is issued on top of a classical logic program (deductive database), as for the "find cheap hotels" example. We call such programs *classical top-k* programs. The top-k retrieval problem for non-recursive classical top-k programs has been addressed in [11], were it has been shown that for a non-recursive classical top-k program $\mathcal{P}$ and query predicate q, $ans_k(q, \mathcal{P})$ can be determined in LogSpace w.r.t. the size of $\mathcal{P}_E$.

The procedure is based on a query reformulation step, in which a rule involving the query predicate in the head is reformulated by replacing an atom A in the body by means of the rule body ϕ, for $A \leftarrow \phi \in \mathcal{P}$, and finally applying a top-k algorithm for relational databases to the obtained query transformations.

For the more general case, this simple strategy is no longer possible. Of course, we always have the possibility to compute all answers (whenever termination is guaranteed), to rank them and to select the top-k ones only. However, this requires to compute the score of all answers. We would like to avoid this in cases in which the extensional database is large and potentially too many tuples would satisfy the query. A distinguishing feature of our query answering procedure is that we do not determine all answers by discovering *all proofs* as e.g. in [5, 13], but rather apply a variant of so-called *memoing* techniques developed for classical logic programming –see, e.g. [15] for an overview. Essentially, the basic idea of our procedure is to collect, during the computation, all answers incrementally together in a similar way as it is done for classical Datalog. Hence,

for instance, we do not rely on any notion of *atom unification*, but rather iteratively access relational tables using relational algebra.

The presentation of our algorithm proceeds as follows. We present a top-k answering procedure for deterministic logic programs (at most one rule per predicate symbol p in the head). Due to lack of space we are not able to include also the more general case of non-deterministic LPs as well (more than one rule per predicate symbol p in the head), which will be included in an extended version of the paper. For this latter case, we just show the problem introduced and outline the solution for it. For the rest of this paper we will assume that the score combination functions are bounded, to avoid such cases as shown in Example 2.

Given $r : q(\mathbf{x}) \leftarrow \phi \in \mathcal{P}$, with $\mathtt{s}(q, r)$ we denote the set of *sons* of q w.r.t. r, i.e. the set of intentional predicate symbols occurring in ϕ. With $\mathtt{p}(q)$ we denote the set of *parents* of q, i.e. the set $\mathtt{p}(q) = \{p_i \colon q \in \mathtt{s}(p_i, r)\}$ (the set of predicate symbols directly depending on q).

Top-k query answering for deterministic LPs. The procedure $TopAnswers$ is detailed in Table 1. The procedure uses some auxiliary functions and data structures: (i) the variable `rankedList` contains, for each intentional predicate p, the current top-ranked tuples together with their score. For each p, the tuples $\langle \mathbf{c}, s \rangle$ in $\mathtt{rankedList}(p)$ are ranked in decreasing order with respect to the score s. We do not allow $\langle \mathbf{c}, s \rangle$ and $\langle \mathbf{c}, s' \rangle$ to occur in $\mathtt{rankedList}(p)$ with $s \neq s'$ (if so, we remove the tuple with the lower score); (ii) the variable `dg` collects the predicate symbols the query predicate q depends on; (iii) the array variable `exp` traces the rule bodies that have been "expanded" (the predicate symbols occurring in the rule body are put into the active list); (iv) the variable `in` keeps track of the predicate symbols that have been put into the active list so far due to an expansion (to avoid, to put the same predicate symbol multiple times in the active list due to rule body expansion). There are other variables, which however do play a role in the procedure $getNextTuple$ only (see Table 1), and are defined in the $TopAnswers$ procedure as they act as global variables. We will discuss them in detail once we address the $getNextTuple$ procedure later on.

Overall, the procedure works as follows. Assume, we are interested in determining the top-k answers of $q(\mathbf{x})$. We start with putting the predicate symbol q in the *active* list of predicate symbols `A`. At each iteration step we select a new predicate p from the queue `A` and get a new tuple ($getNextTuple(p, r)$) satisfying the rule body r whose head contains p with respect to the answers gathered so far. If the evaluation leads to a new answer for p ($\Delta_r \neq \emptyset$), we update the current answer set $\mathtt{rankedList}(p)$ and add all predicates p_j directly depending on p to the queue `A`. At some point the active list will become empty and we have actually found correct answers of $q(\mathbf{x})$. A threshold will be used to determine when we can stop retrieving tuples. Indeed, the threshold determines when any newly retrieved tuple for q scores lower than the current top-k and, thus, cannot modify the top-k ranking (step 9). So, step 1 loops until we do not have k answers above the threshold or, two successive loops do not modify the current set of answers (step 9). Step 2 initializes the active list of predicates. Step 3. loops until no predicate has to be processed anymore. In step 4, we select a predicate symbol to be processed. In step 5, we retrieve the next answer for p. If a new answer has been retrieved (step 6, $\Delta_r \neq \emptyset$) then we update the current answer set $\mathtt{rankedList}(p)$ and

Table 1. The top-k query answering procedure.

Procedure $TopAnswers(\mathcal{P}, q, k)$
Input: Logic program $\mathcal{P}$, query predicate q, $k \geq 1$;
Output: Mapping `rankedList` such that `rankedList`(q) contains top-k answers of q
Init: $\delta = 1$, **for all** rules $r : p(\mathbf{x}) \leftarrow \phi$ in $\mathcal{P}$ **do**
  if p intentional **then** `rankedList`$(p) = \emptyset$, `Q`$(p, r) := \emptyset$;
  if p extensional **then** `rankedList`$(p) = T_p$ **endfor**
1. **loop**
2. `A` $:= \{q\}$, `dg` $:= \{q\}$, `in` $:= \emptyset$, `rL`$' :=$ `rankedList`, **for all** rules $r : p(\mathbf{x}) \leftarrow \phi$ **do** `exp`$(p, r) =$ `false`;
3. **while** (`A` $\neq \emptyset$) **do**
4. **select** $p \in$ `A` where $r : p(\mathbf{x}) \leftarrow \phi$, `A` $:=$ `A` $\setminus \{p\}$, `dg` $:=$ `dg` $\cup$ `s`(p, r);
5. $\Delta_r := getNextTuple(p, r)$
6. **if** $\Delta_r \neq \emptyset$ **then** `rankedList`$(p) :=$ `rankedList`$(p) \cup \Delta_r$, `A` $:=$ `A` $\cup$ (`p`$(p) \cap$ `dg`);
7. **if not** `exp`(p, r) **then** `exp`$(p, r) =$ `true`, `A` $:=$ `A` $\cup$ (`s`$(p, r) \setminus$ `in`), `in` $:=$ `in` $\cup$ `s`(p, r);
  endwhile
8. Update threshold δ;
9. **until** (`rankedList`(q) does contain k top-ranked tuples with score above δ) **or** (`rL`$' =$ `rankedList`);
10. **return** top-k ranked tuples in `rankedList`(q);

Procedure $getNextTuple(p, r)$
Input: Intentional predicate symbol p and rule $r : p(\mathbf{x}) \leftarrow f(A_1, \ldots, A_n) \in \mathcal{P}$;
Output: Next tuple satisfying the body of the r together with the score
Init: Let p_i be the predicate symbol occurring in A_i;
1. **if** `Q`$(p, r) \neq \emptyset$ **then**
  $\langle \mathbf{t}, s \rangle := getTop($`Q`$(p, r))$, remove $\langle \mathbf{t}, s \rangle$ from `Q`(p, r), **return** $\{\langle \mathbf{t}, s \rangle\}$ **fi**
loop
2. Generate the set T of all new join tuples $\mathbf{t}$, using all tuples seen so far in all `rankedList`(p_i) using symmetric hash join
3. **for all** $\mathbf{t} \in T$ **do**
4. $s :=$ compute the score of $p(\mathbf{t})$ using f;
5. **if** neither $\langle \mathbf{t}, s' \rangle \in$ `rankedList`(p) nor $\langle \mathbf{t}, s' \rangle \in$ `Q`(p, r) with $s \leq s'$ **then** insert $\langle \mathbf{t}, s \rangle$ into `Q`(p) **endfor**
until `Q`$(p, r) \neq \emptyset$ or no new valid join tuple can be generated
6. **if** `Q`$(p, r) \neq \emptyset$ **then** $\langle \mathbf{t}, s \rangle := getTop($`Q`$(p, r))$, remove $\langle \mathbf{t}, s \rangle$ from `Q`(p), **return** $\{\langle \mathbf{t}, s \rangle\}$ **else return** $\emptyset$ **fi**

add all predicates p_j, that directly depend on p, to the queue `A`. In step 7, we put once all intentional predicate symbols appearing in the rule body of p in the active list for further processing.

We next describe the $getNextTuple$ procedure (see Table 1). It's main purpose is, given a predicate symbol p and a rule $r : p(\mathbf{x}) \leftarrow \phi$, to get back the next tuple (and its score) satisfying the conditions of the rule r. The procedure is a generalization of the analogous $getNext$ procedure described in [4] and uses the so-called *symmetric Hash Rank Join* (HRJN) algorithm. This is not surprising as the list of atoms in a rule body may be seen as multiple joins together with a scoring function.

Let us first describe the intuition behind the procedure. For the sake of illustrative purposes assume that the rule r associated to p is $p(x) \leftarrow p_1(x, y) \cdot p_2(y, z) \in \mathcal{P}$. The idea is as follows:

1. we incrementally generate new valid join combinations $\langle x, y, z \rangle$ from the tuples in `rankedList`(p_1) and `rankedList`(p_2) using some join strategy. In particular, as [4], we alternatively access first `rankedList`(p_1) and then `rankedList`(p_2). We select the next unseen tuple in `rankedList`(p_1) and then build all join combinations with the tuples seen so far in `rankedList`(p_2). Then we select the next unseen tuple in `rankedList`(p_2) and then build all join combinations with the tu-

ples seen so far in $\texttt{rankedList}(p_1)$ and so on until we find some valid join tuples ($\texttt{Q}(p, r) \neq \emptyset$).
2. the join tuples and their scores will be put in the queue $\texttt{Q}(p, r)$ and the top-ranked one is returned.

Specifically, in step 1, whenever we already have some tuples in the queue $\texttt{Q}(p, r)$ of p (obtained by a previous call) then we just return the top-ranked one. Ties are split randomly. In step 2 we generate all candidate joins, involving all seen tuples of the predicates occurring in the rule body of p. For each of them we compute its score (step 4). We put the results on the queue $\texttt{Q}(p, r)$ (step 5) and return the top-ranked one. As $\texttt{Q}(p, r)$ may still contain answers for p, the next time we ask for a next tuple with respect to p and r, we access $\texttt{Q}(p, r)$ directly (step 1).

Finally, threshold δ is determined as follows. It is computed as in [4]. Suppose that for the query predicate q we have a rule $r : q(x) \leftarrow f(p_1, \ldots, p_n) \in \mathcal{P}$. Let $\mathbf{t}_i^{\perp}$ be the last tuple seen in $\texttt{rankedList}(p_i)$, while let $\mathbf{t}_i^{\top}$ be the top ranked one in $\texttt{rankedList}(p_i)$. With $\mathbf{t}_i.score$ we indicate the tuple's score. Then δ is the maximum of the following n values:

$$\begin{aligned}
\delta_1 &= f(\mathbf{t}_1^{\perp}.score, \mathbf{t}_2^{\top}.score, \ldots, \mathbf{t}_n^{\top}.score) \\
\delta_2 &= f(\mathbf{t}_1^{\top}.score, \mathbf{t}_2^{\perp}.score, \ldots, \mathbf{t}_n^{\top}.score) \\
&\vdots \\
\delta_n &= f(\mathbf{t}_1^{\top}.score, \mathbf{t}_2^{\top}.score, \ldots, \mathbf{t}_n^{\perp}.score) \,.
\end{aligned}$$

For instance, for $q(x) \leftarrow p_1(x, y) \cdot p_2(y, z) \in \mathcal{P}$ we have

$$\begin{aligned}
\delta_1 &= \mathbf{t}_1^{\perp}.score \cdot \mathbf{t}_2^{\top}.score \\
\delta_2 &= \mathbf{t}_1^{\top}.score \cdot \mathbf{t}_2^{\perp}.score \\
\delta &= \max(\delta_1, \delta_2) \,.
\end{aligned}$$

It is not difficult to see that whenever we consider a new join combination, its score will be below to δ. Indeed, if we consider a new join tuple using the next unseen tuple from $\texttt{rankedList}(p_1)$ and a seen tuple in $\texttt{rankedList}(p_2)$, its score will be below δ_1, while if we consider a new join tuple using the next unseen tuple from $\texttt{rankedList}(p_2)$ and a seen tuple in $\texttt{rankedList}(p_1)$, its score will be below δ_2. Therefore, overall the score will be below δ. It is thus not surprising that whenever we have top-k answers for q with score above δ we can stop the retrieval process (see step 9 of $TopAnswers$). This property can be generalized to n-ary joins (see [4], Theorem 4.2.1). For the sake of illustrative purposes, let us consider the following abstract examples.

Example 3. Assume that we have the following query rule $q(x) \leftarrow \min(r_1(x, y), r_2(y, z))$, where q is the query predicate and r_1, r_2 are extensional predicates with tables (with millions of tuples)

$recId$	r_1	r_2
1	a b 1.0	m h 0.95
2	c d 0.9	m j 0.85
3	e f 0.8	f k 0.75
4	l m 0.7	m n 0.65
5	o p 0.6	p q 0.55
⋮	⋮ ⋮ ⋮	⋮ ⋮ ⋮

The table below reports a top-2 retrieval computation. The left table reports data related to each loop in the $TopAnswers$ procedure, while the other shows at each iteration the execution $getNextTuple$ ($r_j(i)$ means that we access the i-th tuple in relation r_j). The first call of $getNextTuple(q)$ requires several alternative accesses to r_i before a tuple can be found ($\langle e, k, 0.75\rangle$). In the second call we get immediately two candidate tuples. In the third call, as $\mathtt{Q}(q, r) \neq \emptyset$ we get immediately the next candidate ($\langle l, j, 0.7\rangle$). Finally, in the fourth call, we retrieve $\langle l, n, 0.65\rangle$. As now $\mathtt{rankedList}(q)$ contains 2 answers above the threshold of 0.7, we can stop and return $\{\langle e, k, 0.75\rangle, \langle l, h, 0.7\rangle\}$. Note that no new retrieved answer may have a score above 0.7. Indeed, the next one would be $\langle o, q, 0.55\rangle$ and, thus, *not all tuples are processed* (which would be unfeasible in practice).

$TopAnswers$					
$Iter$	A	p	Δ_r	$\mathtt{rankedList}(p)$	δ
1.	q	q	$\langle e, k, 0.75\rangle$	$\langle e, k, 0.75\rangle$	0.8
2.	q	q	$\langle l, h, 0.7\rangle$	$\langle e, k, 0.75\rangle, \langle l, h, 0.7\rangle$	0.75
3.	q	q	$\langle l, j, 0.7\rangle$	$\langle e, k, 0.75\rangle, \langle l, h, 0.7\rangle, \langle l, j, 0.7\rangle$	0.75
4.	q	q	$\langle l, n, 0.65\rangle$	$\langle e, k, 0.75\rangle, \langle l, h, 0.7\rangle,$ $\langle l, j, 0.7\rangle, \langle l, n, 0.65\rangle$	0.7

$getNextTuple$			
$Iter$	p_i	$\langle t_i, s_i\rangle$	$\mathtt{Q}(p, r)$
1.	r_1	$r_1(1)$	–
	r_2	$r_2(1)$	–
	r_1	$r_1(2)$	–
	r_2	$r_2(2)$	–
	r_1	$r_1(3)$	–
	r_2	$r_2(3)$	$\langle e, k, 0.75\rangle$
2.	r_1	$r_1(4)$	$\langle l, h, 0.7\rangle, \langle l, j, 0.7\rangle$
3.	–	–	$\langle l, j, 0.7\rangle$
4.	r_2	$r_2(4)$	$\langle l, n, 0.65\rangle$

From computational point of view, by a similar analysis as in [9], it can be shown that $TopAnswer$ is exponential with respect to $|\mathcal{P}|$ (combined complexity), but polynomial in $|\mathcal{P}_E|$ (data complexity), and we have:

Proposition 1. *Given a deterministic logic program $\mathcal{P}$ in which all scoring functions are bounded, then $TopAnswers(\mathcal{P}, q, k)$ terminates with $TopAnswers(\mathcal{P}, q, k) = ans_k(q, \mathcal{P})$.*

Top-k query answering for general LPs. We first illustrate the problem that is introduced in the case a predicate symbol p is in the head of multiple rules and then sketch how we solve it. Our top-k retrieval algorithm is based on the fact that whenever we find a new instance $\langle \mathbf{c}, s\rangle$ for a predicate p occurring in $\mathcal{P}$, any successive retrieved instance $\langle \mathbf{c}', s'\rangle$ for p is scored lower than $\langle \mathbf{c}, s\rangle$, i.e. $s' \leq s$ (the fact that score combination functions are bounded is crucial here), which allows us to apply the stopping criteria based on a threshold. Unfortunately, if p is in the head of more than one rule this is no longer true. Indeed, clearly two rules $p(\mathbf{x}) \leftarrow \phi_1$ and $p(\mathbf{x}) \leftarrow \phi_2$ are equivalent to the rule $p(\mathbf{x}) \leftarrow \max(\phi_1, \phi_2)$, and $\max$ is not a bounded score combination function. So, it is not difficult to find an example where given a retrieved instance $\langle \mathbf{c}, s\rangle$ for p, a successive retrieved instance $\langle \mathbf{c}', s'\rangle$ for p may have a score higher than $\langle \mathbf{c}, s\rangle$, i.e. $s' > s$.

Example 4. Consider the seven rules $q(x) \leftarrow t_1(x)$, $q(x) \leftarrow p_1(x)$, $p_1(x) \leftarrow t_2(x)$, $t_1(a) \leftarrow 0.4$, $t_1(b) \leftarrow 0.3$, $t_2(c) \leftarrow 0.5$, $t_2(d) \leftarrow 0.2$. A naive extension of our procedure, may retrieve first $\langle a, 0.4\rangle$ for q, second $\langle c, 0.5\rangle$, third $\langle b, 0.3\rangle$ and, eventually $\langle d, 0.2\rangle$.

As we can see in the above example, it may not be guaranteed that any successive retrieved tuple for q is scored lower than the previous one. However, there is still a

simple strategy to overcome to this problem. In fact, note that any successively retrieved tuple for rule $r_1 : q(x) \leftarrow t_1(x)$ is scored lower than the one retrieved before for r_1. Similarly, any successively retrieved tuple for rule $r_2 : q(x) \leftarrow p_1(x)$ is scored lower than the one retrieved before for r_2. Therefore, one strategy may be to gather at least one answer for each of the rules r_1 and r_2 and only then merge the retrieved answers for r_1 and r_2 to build the answers for q. This will guarantee that successively retrieved answers for r_1 and r_2 are scored lower than the already retrieved ones for q. Of course, the threshold δ for q is now $\delta = \max(\delta_{r_1}, \delta_{r_2}\}$, where δ_{r_1} and δ_{r_2} are computed as previously for rule r_1 and r_2, respectively. The detailed procedure will be described in more detail in an extended work.

4 Conclusions

The problem of top-k retrieval will be an important problem, e.g. in logic-based (multimedia) information retrieval. We have addressed this issue in the context of logic programs. We are unaware of any other work addressing this problem for many-valued (recursive) logic programs (for non-recursive logic programs, see [11, 14]), computing the answers iteratively accessing relational tables using relational algebra (and, thus, is not resolution-based).

Major topics for future research include: (i) can we refine our strategy in case the score combination functions are not bounded? (ii) How can we deal with aggregates (maybe relying on [7])? (iii) Can we apply similar ideas to other popular logical formalism, such as *Description Logics* (DLs) [1] (the only work we know about are [12, 10]) and their combination with LPs?

References

1. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
2. Didier Dubois and Henri Prade. Possibility theory, probability theory and multiple-valued logics: A clarification. *Annals of Mathematics and Artificial Intelligence*, 32(1-4):35–66, 2001.
3. Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Rec.*, 31(2):109–118, 2002.
4. Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In VLDB-03, pages 754–765, 2003.
5. Michael Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
6. Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. RankSQL: query algebra and optimization for relational top-k queries. In SIGMOD-05, pages 131–142, New York, NY, USA, 2005. ACM Press.
7. Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In SIGMOD-06, USA, 2006. ACM Press.
8. Carlo Meghini, Fabrizio Sebastiani, and Umberto Straccia. A model of multimedia information retrieval. *Journal of the ACM*, 48(5):909–970, 2001.
9. Umberto Straccia. Query answering in normal logic programs under uncertainty. In ECSQARU-05, LNCS 3571, pages 687–700, Barcelona, Spain, 2005. Springer Verlag.

10. Umberto Straccia. Answering vague queries in fuzzy DL-Lite. In *Proceedings of the 11th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, (IPMU-06)*, 2006.
11. Umberto Straccia. Towards top-k query answering in deductive databases. In *Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics (SMC-06)*. IEEE, 2006.
12. Umberto Straccia. Towards top-k query answering in description logics: the case of DL-Lite. In JELIA-06, LNCS 4160, 2006. Springer Verlag.
13. Peter Vojtáš. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124:361–370, 2001.
14. Peter Vojtáš. Fuzzy logic aggregation for semantic web search for the best (top-k) answer. In Elie Sanchez, editor, *Fuzzy Logic and the Semantic Web*, Capturing Intelligence, chapter 17, pages 341–359. Elsevier, 2006.
15. David S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.