Ontological Query Answering under Many-Valued Group Preferences in Datalog+/–

Bettina Fazzinga[a], Thomas Lukasiewicz[b,c], Maria Vanina Martinez[d], Gerardo I. Simari[d], Oana Tifrea-Marciuska[b,c]

[a] *ICAR-CNR, National Research Council, Italy*
[b] *Department of Computer Science, University of Oxford, UK*
[c] *The Alan Turing Institute, London, UK*
[d] *Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS) and Institute for Computer Science and Engineering (ICIC), CONICET–UNS, Argentina*

Abstract

The Web has recently been changing more and more to what is called the Social Semantic Web. As a consequence, the ranking of search results no longer depends solely on the structure of the interconnections among Web pages. In this paper, we argue that such rankings can be based on user preferences from the Social Web and on ontological background knowledge from the Semantic Web. We propose an approach to top-k query answering under user preferences in Datalog+/– ontologies, where the queries are unions of conjunctive queries with safe negation, and the preferences are defined via numerical values. To this end, we also generalize the previous RankJoin algorithm to our framework. Furthermore, we explore the generalization to the preferences of a group of users. Finally, we provide experimental results on the performance and quality of our algorithms.

1. Introduction

During the recent years, the Web has been increasingly turning into the so-called Web of Data as a special case of the Semantic Web. Furthermore, users themselves play an increasingly central role in the creation and delivery of contents on the Web. The combination of these two technological waves is called the *Social Semantic Web* (or also *Web 3.0*), where the classical Web of interlinked documents is more and more turning into (i) semantic data and tags constrained by ontologies, and (ii) social data, such as connections, interactions, reviews, and tags. The Web is thus shifting away from data on linked Web pages towards interlinked data in social networks on the Web that rely on ontologies. This requires new technologies for search and query answering, where the ranking of search results is not solely based on the link structure between Web pages anymore, but on the information available in the Social Semantic Web—in particular, the underlying ontological knowledge present in user-created content, as well as preferences that the user implicitly or explicitly presents in such content [1, 2, 3]. Because of the explosion of social content, it is also important to model the preferences of large groups of users. Clearly, social media are a valuable source for mining preferences and opinions of groups of users for commercial or political purposes. In addition, also the users themselves profit: people post their preferences on social media and expect to get personalized information. For these reasons, in this work, we focus on exploiting users' preferences, in the form of scores, for top-k query answering over ontological knowledge bases, in order to generalize the current PageRank-based sorting of search results.

To address this problem, a model of preferences of individual users can be adopted and then the individual preferences can be aggregated into a group's preferences. However, this comes along with two additional challenges. The

Email addresses: bettina.fazzinga@icar.cnr.it (Bettina Fazzinga), thomas.lukasiewicz@cs.ox.ac.uk (Thomas Lukasiewicz), mvm@cs.uns.edu.ar (Maria Vanina Martinez), gis@cs.uns.edu.ar (Gerardo I. Simari), oana.tifrea@cs.ox.ac.uk (Oana Tifrea-Marciuska)

first challenge is to define a group preference semantics that solves (all but certain) *disagreements* among users—e.g., people (even friends) often have different tastes in restaurants; a system should return results in such a way that certain properties are satisfied (e.g., ensuring that each individual benefits from the result). The second challenge is to allow for efficient algorithms, i.e., to compute efficiently the answers to queries under aggregated group preferences [4].

In previous work, we studied the complexity of the single-user case, showing intractability for conjunctive queries (CQs) and motivating the search for tractable special cases [5]; then, in [6], we developed algorithms to answer k-rank queries for unions of atomic queries under group preferences and uncertainty in Datalog+/– [7] ontologies, where the preferences of every user are expressed as strict partial orders (irreflexive and transitive binary relations). The algorithms in [6] are not optimized for score-based preferences, as they do not leverage this simpler structure. As we show below, the special case of score-based preferences allows us to compute answers in polynomial time. In [8], the single-user case for CQs and scores is studied; here, the novelty relative to that work lies in adding negation, projection, and disjunction to queries, plus multiple users and an experimental evaluation. As underlying ontology languages, we chose the Datalog+/– family of ontology languages, because it is highly flexible, it generalizes the lightweight ontology languages in the *DL-Lite* family, and there are also implementations available [9, 10].

The main contributions of this work can be briefly summarized as follows:

(i) We propose an approach to top-k query answering under user preferences in Datalog+/– ontologies, where the queries are unions of conjunctive queries with safe negation, and the preferences are defined via numerical scores. The evaluation of queries involves joining and aggregating multiple inputs to provide users with the top-k results based on their preferences.

(ii) We develop an algorithm for top-k query answering in this framework, which is based on a generalization of the previous RankJoin algorithm [8] to ontology-based data access and to unions of conjunction queries (UCQs) with safe negation as queries.

(iii) We generalize the above approach to top-k query answering under the preferences of a group of users (rather than a single user only), which involves the aggregation of (potentially conflicting) user preferences. We study two different approaches to such aggregations and their properties.

(iv) We provide experimental results on the performance (in terms of running time) and the quality of our algorithms. More specifically, using standard measures from information retrieval, we explore which of the techniques for aggregating user preferences is the best and how much their results differ.

The rest of this paper is organized as follows. In Section 2, we provide some preliminaries on Datalog+/– and its generalization by (single-user) preferences. Section 3 introduces (single-user) k-rank answers to unions of conjunctive queries with safe negation, while Section 4 extends the RankJoin algorithm [8] to work with such queries in our framework. In Section 5, we then generalize this framework to group preferences. Section 6 reports on experimental results, and Section 7 discusses related work. In Section 8, we summarize the main results and give an outlook on future research.

2. Preliminaries

In this section, we briefly recall some necessary background from Datalog+/– [7], namely on relational databases, (Boolean) conjunctive queries ((B)CQs), tuple-generating dependencies (TGDs), negative constraints, universal models, and ontologies in Datalog+/–. We also briefly recall a generalization of Datalog+/– by preferences from [5].

Datalog+/–. We assume an infinite universe of *constants* Δ (the "normal" domain of a database), an infinite set of *(labeled) nulls* Δ_N, and an infinite set of variables $\mathcal{V}$. Different constants represent different values (*unique name assumption*), while different nulls may represent the same value. We assume a *relational schema* $\mathcal{R}$, which is a finite set of *predicate symbols* (or simply *predicates*). The *vocabulary* Φ consists of Δ, Δ_N, and $\mathcal{R}$. A *term* t is a constant, null, or variable. An *atom* has the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate, and $t_1, \ldots, t_n$ are terms. The *(extended) Herbrand base* for Φ, denoted HB_Φ, is the set of all atoms with predicates from $\mathcal{R}$ and arguments from $\Delta \cup \Delta_N$. Conjunctions of atoms are often identified with the sets of their atoms. An *instance* I is a (possibly infinite) set of atoms $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple over $\Delta \cup \Delta_N$. A *database* D is a finite instance with only constants.

A *homomorphism* is a substitution $h\colon \Delta \cup \Delta_N \cup \mathcal{V} \to \Delta \cup \Delta_N \cup \mathcal{V}$ that is the identity on Δ and that maps Δ_N to $\Delta \cup \Delta_N$; it is naturally extended to atoms, sets of atoms, and conjunctions of atoms. A *conjunctive query (CQ)* has the form $q(\mathbf{X}) = \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$, where $\varphi(\mathbf{X}, \mathbf{Y})$ is a conjunction of atoms with the variables $\mathbf{X}$, $\mathbf{Y}$, and possibly constants (but no nulls); its set of all *answers* over an instance I, denoted $q(I)$, is the set of all tuples $\mathbf{t}$ over Δ for which a homomorphism $\mu\colon \mathbf{X} \cup \mathbf{Y} \to \Delta \cup \Delta_N$ exists such that $\mu(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq I$ and $\mu(\mathbf{X}) = \mathbf{t}$. A *Boolean CQ (BCQ)* is a CQ $q()$, often written without quantifiers. A BCQ q is *true* over I, denoted $I \models q$, if $q(I) \neq \emptyset$.

A *tuple-generating dependency (TGD)* σ is a first-order formula $\forall \mathbf{X}\, \varphi(\mathbf{X}) \to \exists \mathbf{Y}\, p(\mathbf{X}, \mathbf{Y})$, where $\mathbf{X} \cup \mathbf{Y} \subseteq \mathcal{V}$, $\varphi(\mathbf{X})$ is a conjunction of atoms (without nulls), and $p(\mathbf{X}, \mathbf{Y})$ is an atom (without nulls); $\varphi(\mathbf{X})$ is the *body* of σ, while $p(\mathbf{X}, \mathbf{Y})$ is the *head* of σ. For clarity, we consider single-atom-head TGDs; however, our results can be extended to TGDs with a conjunction of atoms in the head. A TGD σ is *guarded*, if it contains an atom in its body that contains all universally quantified variables of σ. A TGD σ is *linear*, if it contains only a single atom in its body. An instance I satisfies σ, written $I \models \sigma$, if whenever a homomorphism h exists such that $h(\varphi(\mathbf{X})) \subseteq I$, then there exists $h' \supseteq h_{|\mathbf{X}}$, where $h_{|\mathbf{X}}$ is the restriction of h on $\mathbf{X}$, such that $h'(p(\mathbf{X}, \mathbf{Y})) \in I$. A *negative constraint (NC)* ν is a first-order formula $\forall \mathbf{X}\, \varphi(\mathbf{X}) \to \bot$, where $\mathbf{X} \subseteq \mathcal{V}$, $\varphi(\mathbf{X})$ is a conjunction of atoms (without nulls), called the *body* of ν, and $\bot$ denotes the truth constant *false*. An instance I satisfies ν, written $I \models \nu$, if there is no homomorphism h such that $h(\varphi(\mathbf{X})) \subseteq I$. We often omit the universal quantifiers in front of TGDs and NCs. As another component, Datalog+/– allows for special types of *equality-generating dependencies (EGDs)* [7], which are omitted here, as they can also be modeled via NCs. We define answers to CQs relative to databases, TGDs, and NCs as those answers that are true in *all* their models as follows. Given a set Σ of TGDs and NCs, an instance I *satisfies* (or is a *model* of) Σ, denoted $I \models \Sigma$, if I satisfies each TGD and NC of Σ. The *models* of a database D and Σ, denoted $mods(D, \Sigma)$, is the set of instances $\{I \mid I \supseteq D,\ I \models \Sigma\}$. The *answer* to a CQ q relative to D and Σ is the set of tuples $ans(q, D, \Sigma) = \bigcap_{I \in mods(D,\Sigma)} q(I)$. The answer to a BCQ q relative to D and Σ is *true*, denoted $D \cup \Sigma \models q$, if $ans(q, D, \Sigma) \neq \emptyset$.

BCQs q over D and sets of TGDs Σ can be evaluated on *universal models* $U_{D,\Sigma}$ of D and Σ (i.e., $D \cup \Sigma \models q$ iff $U_{D,\Sigma} \models q$), which can be homomorphically mapped onto every model in $mods(D, \Sigma)$. NCs are then easily added to Σ: violating any of them results into D and Σ being unsatisfiable, and so q being true. One universal model is the (possibly infinite) *chase* for D and Σ, denoted $chase(D, \Sigma)$ [7]. Query answering under general TGDs is undecidable [11, 12]. Here, we consider only sets of TGDs Σ where the evaluation of BCQs q is decidable and possible on a finite portion of $chase(D, \Sigma)$ of polynomial size in the data complexity, such as guarded and linear TGDs [7].

A *Datalog+/– ontology* $O = (D, \Sigma)$, where $\Sigma = \Sigma_T \cup \Sigma_{NC}$, consists of a database D, a finite set of TGDs Σ_T, and a finite set of negative constraints Σ_{NC}. The following example illustrates a simple Datalog+/– ontology.

Example 1. A Datalog+/– ontology $O = (D, \Sigma)$ for the restaurant domain is given below. Intuitively, D encodes that f_1, f_2, f_3, f_4, and f_5 are *pizza*, *pasta*, *salad*, *fish*, and *sushi*, respectively. Also, we have two places p_1 and p_2, where p_1 is a *pizzeria* and a *fine_dining* place that serves pizza and pasta, while p_2 is a *sushi bar* that serves fish and sushi. The TGDs in Σ encode that every place has a location and a type.

$D = \{food(f_1, pizza),\ food(f_2, pasta),\ food(f_3, salad),\ food(f_4, fish),\ food(f_5, sushi),\ serves(p_1, f_1),\ serves(p_1, f_2),$
$serves(p_2, f_4),\ serves(p_2, f_5),\ place(p_2, sushi\_bar),\ place(p_1, pizzeria),\ place(p_1, fine\_dining)\},$

$\Sigma = \{place(P, T) \to \exists L\, located\_in(P, L);\ place(P, T) \to type(T)\}.$ ∎

PrefDatalog+/–. We recall some basic concepts from PrefDatalog+/–. A *preference relation* is any binary relation $\succ \subseteq HB_\Phi \times HB_\Phi$. In the following, we assume that $a \succ b$ iff $a' \succ b'$, if a and b are isomorphic to a' and b', respectively. Here, s is *isomorphic* to s', if $\nu(s) = \nu(s')$, where ν substitutes every null by the same fresh null $\nu^\star$. Intuitively, any two atoms that cannot be distinguished depending on predicates and constants (and thus may represent the same objects, as some of the contained nulls may represent the same objects) should be treated as the same in "$\succ$". Here, we also adapt PrefDatalog+/– in [5] to the special case where the preference model is *score-based*, i.e., it is an assignment of a numeric score to each element in HB_Φ in such a way that $a_1 \succ a_2$ iff $score(a_1) > score(a_2)$ (also called *strict weak orderings*). In the sequel, we only refer to such score functions, and the corresponding preference relation is implicit. A *preference-based Datalog+/–* (*PrefDatalog+/–*) ontology (or knowledge base) $KB = (O, score)$ consists of a Datalog+/– ontology O and a score function $score$ from its extended Herbrand base HB_Φ to $[0, 1]$.

Note that score functions are generally defined via compact representations rather than enumerations (e.g., possibilistic networks from knowledge representation and reasoning [13] or rankings from machine learning [14]); they are either explicitly defined by the user, or implicitly mined from the user's behavior (e.g., search and click history).

	food		Score functions		
	ID	Name	$score_{u_1}$	$score_{u_2}$	$score_{u_3}$
t_1	f_1	pizza	0.8	0.3	0.7
t_2	f_2	pasta	0.7	0.6	0.65
t_3	f_3	salad	0.6	0.2	0.1
t_4	f_4	fish	0.2	0.9	0.7
t_5	f_5	sushi	0.2	0.2	0.5

	serves		Score functions		
	Place	Food	$score_{u_1}$	$score_{u_2}$	$score_{u_3}$
t_{14}	p_1	f_2	0.7	0.4	0.3
t_{15}	p_1	f_1	0.6	0.2	0.3
t_{16}	p_2	f_4	0.6	0.9	0.6
t_{17}	p_2	f_5	0.5	0.1	0.65

	type	Score functions		
	ID	$score_{u_1}$	$score_{u_2}$	$score_{u_3}$
t_6	sushi_bar	0.8	0.2	0.4
t_7	pizzeria	0.6	0.5	0.75
t_8	fine_dining	0.6	0.2	0.4

	place		Score functions		
	ID	Type	$score_{u_1}$	$score_{u_2}$	$score_{u_3}$
t_{11}	p_1	pizzeria	0.6	0.5	0.75
t_{12}	p_2	sushi_bar	0.6	0.2	0.4
t_{13}	p_1	fine_dining	0.1	0.1	0.2

	located_in		Score functions		
	Place	Location	$score_{u_1}$	$score_{u_2}$	$score_{u_3}$
t_9	p_1	l_1	0.7	0.7	0.7
t_{10}	p_2	l_2	0.8	0.6	0.1

Figure 1: Some scores of the score functions $score_{u_1}$, $score_{u_2}$, and $score_{u_3}$ of users u_1, u_2, and u_3, respectively, in the running example.

Example 2. Continuing Example 1, the scores of the atoms entailed by O over the predicates $food$, $serves$, $type$, $place$, and $located\_in$ under the score functions $score_{u_1}$, $score_{u_2}$, and $score_{u_3}$ are shown in Fig. 1. The scores of some non-entailed atoms are as follows: $score_{u_1}(serves(p_2, f_1)) = 0.5$, $score_{u_1}(serves(p_2, f_2)) = score_{u_1}(serves(p_2, f_3)) = 0.3$, $score_{u_1}(serves(p_1, f_3)) = score_{u_1}(serves(p_1, f_4)) = score_{u_1}(serves(p_1, f_5)) = 0.4$. ■

3. Answering UCQs with Negation

We now explore how to obtain k-rank answers to unions of CQs with (safe) negation (UNCQs).

As a first step, the query-relevant part of the Datalog+/– ontology is *materialized*, i.e., given an ontology $KB = ((D, \Sigma), score)$, we obtain one of the form $KB' = ((D', \emptyset), score)$ on which the query can be equivalently evaluated. For guarded Datalog+/– ontologies, this is possible in polynomial time (and thus the materialization has also a polynomial size) in the data complexity (where the schema $\mathcal{R}$, the set Σ of TGDs and NCs, and the query size are all fixed) by computing the guarded chase forest [7] up to a certain (query-dependant) depth. *This materialized database is equivalent to the original ontology in that it can be used to evaluate all CQs that are of bounded width, fixed, or atomic, and thus no loss of expressive power is suffered by taking this step*. This evaluation can clearly be done in polynomial time in the data complexity; however, although the materialized database has a polynomial size, it may be quite large in general, which motivates realizing the score-based evaluation of unions of CQs with safe negation via a combination with a corresponding generalization of the RankJoin algorithm [8], to make the evaluation more efficient. In the rest of this paper, whenever we refer to a Datalog+/– ontology O or a PrefDatalog+/– ontology $KB = (O, score)$, we assume that O is already materialized unless stated differently.

We focus on unions (i.e., disjunctions) of CQs that admit safely negated atoms, where safeness intuitively means that the domain of arguments of a negated atom is restricted to the domain of arguments of the positive atoms.

Definition 1 (NCQs/UNCQs). A *CQ with safe negation (NCQ)* has the form $q(\mathbf{X}) = \exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge \neg N_{m+1}(\mathbf{Z}_{m+1}) \wedge \cdots \wedge \neg N_{m+n}(\mathbf{Z}_{m+n})$, where (i) R_i and N_j are predicates from $\mathcal{R}$, (ii) $\mathbf{Z}_j$ are tuples of variables over $\mathbf{X} \cup \mathbf{Y}$ such that $\mathbf{X} \subseteq \mathbf{Z}_1 \cup \cdots \cup \mathbf{Z}_m$, and (iii) for every negated atom N_i, it holds that $\mathbf{Z}_i \subseteq \mathbf{Z}_1 \cup \cdots \cup \mathbf{Z}_m$.

The *positive part* of $q(\mathbf{X})$, denoted $Pos(q(\mathbf{X}))$, is defined as $\exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m)$. A *union of CQs with safe negation (UNCQ)* has the form $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, where each $q_i(\mathbf{X})$ is an NCQ. Its *positive part*, denoted $Pos(q(\mathbf{X}))$, is defined as $\bigvee_{i=1}^{l} Pos(q_i(\mathbf{X}))$.

Example 3. Consider the following NCQ $q_1(X_1, X_2) = \exists Y_1, Y_2\, (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ over the relations in Fig. 1, which asks for the pairs of places X_1 and foods X_2 such that X_2 is not served in X_1. The positive part is $\exists Y_1, Y_2\, (place(X_1, Y_1) \wedge food(X_2, Y_2))$. The query $q_1(X_1, X_2) = \exists Y_1\, place(X_1, Y_1) \vee \exists Y_2\, (\neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ is a UNCQ. ∎

Before defining the answers to NCQs and UNCQs, we introduce the notion of joined tuple and their scores for NCQs. Intuitively, any joined tuple is a sequence of tuples, one for each atom of the NCQ, whose natural join projected to the answer variables forms an answer to the NCQ. As for the scores of joined tuples, since we are dealing with a database containing score-based tuples, we need a suitable way to combine the scores of these tuples to obtain scores for query answers. A natural choice is that of using standard fuzzy operators, as widely adopted for tuples with scores in $[0, 1]$. Specifically, we use the min (resp., max) operator for computing the scores of tuples resulting from conjunction (resp., disjunction), and the not operator for negation (i.e., taking $1 - score(t_i)$, if t_i results from a negated atom of the query). This is quite common in the evaluation of top-k queries over score-based tuples. Indeed, considering an atom $p(t)$, if a user u assigns a score s to tuple t in $p(t)$, then this means that (i) u's appreciation for t is quantified as s, and (ii) u's dislike for t in $p(t)$ is quantified as $1 - s$. Thus, t is not fully liked or disliked by u, but it is both liked and disliked with a certain degree. Intuitively, among all correct query answers, when considering matching tuples for positive (resp., negated) query atoms, we maximize the user's appreciation (resp., dislike) for these matching tuples. The following definition formally states how the computation is done for NCQs.

Definition 2. Given a PrefDatalog+/– ontology $KB = (O, score)$ and an NCQ $q(\mathbf{X}) = \exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge \neg N_{m+1}(\mathbf{Z}_{m+1}) \wedge \cdots \wedge \neg N_{m+n}(\mathbf{Z}_{m+n})$, a *joined tuple* $\hat{t}$ of q over KB is a sequence of tuples $t_1, \ldots, t_m$ over $\Delta \cup \Delta_N$ where a homomorphism $\mu\colon \mathbf{Z}_1 \cup \cdots \cup \mathbf{Z}_m \rightarrow \Delta \cup \Delta_N$ exists such that, for every $i \in \{1, \ldots, m\}$, $\mu(\mathbf{Z}_i) = t_i$ for some $R_i(t_i) \in O$. We often identify $\hat{t}$ with the natural join of its tuples. Each such $\hat{t}$ is associated with the *score* $\hat{s} = min(min_{i=1}^{m} score(t_i),\ min_{j=1}^{n} score'(t_{m+j}))$, where $t_{m+j} = \mu(\mathbf{Z}_{m+j})$ and $score'(t_{m+j}) = 1 - score(t_{m+j})$, if $N_{m+j}(t_{m+j}) \notin O$, for all $j \in \{1, \ldots, n\}$.

Example 4. Consider $q_1(X_1, X_2) = \exists Y_1, Y_2\, (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ of Example 3 and consider the score function $score_{u_1}$. Then, $\hat{t} = p_2, sushi\_bar, f_1, pizza$ is a joined tuple of q_1; it comes from joining $t_{12} = (p_2, sushi\_bar)$ from the relation $place$ and $t_1 = (f_1, pizza)$ from $food$. As there is no tuple in $serves$ that matches the values for variables X_1 and X_2 set by $\hat{t}$, and we have from Example 2 that $score_{u_1}(serves(p_2, f_1)) = 0.5$, the score of $\hat{t}$ is $min(0.6, 0.8, 1 - 0.5) = 0.5$. Also, both $\hat{t}_1 = p_1, pizzeria, f_4, fish$ and $\hat{t}_2 = p_1, fine\_dining, f_4, fish$ are joined tuples of q_1, whose scores are $min(0.6, 0.2, 1 - 0.4) = 0.2$ and $min(0.1, 0.2, 1 - 0.4) = 0.1$, respectively, recalling that $score_{u_1}(serves(p_1, f_4)) = 0.4$ from Example 2. ∎

Given a PrefDatalog+/– ontology $KB = (O, score)$ and an NCQ $q(\mathbf{X}) = \exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge \neg N_{m+1}(\mathbf{Z}_{m+1}) \wedge \cdots \wedge \neg N_{m+n}(\mathbf{Z}_{m+n})$, we denote with $J(q, KB)$ the set of pairs $(\hat{t}, \hat{s})$ such that $\hat{t}$ is a joined tuple of q over KB, and $\hat{s}$ is its score, and by $J_t(q, KB)$ the set of pairs $(\hat{t}, \hat{s})$ of $J(q, KB)$ such that the projection onto $\mathbf{X}$ of $\hat{t}$ yields t. We now define the answers to NCQs.

Definition 3 (Answers to NCQs). Given a PrefDatalog+/– ontology KB and an NCQ $q(\mathbf{X})$, the set of *answers* to q over KB, denoted $ans(q, KB)$, is the set of all (t, p) such that (i) t is a tuple over Δ and the projection of some joined tuple $\hat{t}$ of q over KB onto $\mathbf{X}$, and (ii) $p = max_{(\hat{t}, \hat{s}) \in J_t(q(\mathbf{X}), KB)}\ \hat{s}$.

Example 5. Continuing the previous example, we have that the tuple's (p_1, f_4) score is computed as follows $max(min(0.6, 0.2, 1 - 0.4), min(0.1, 0.2, 1 - 0.4)) = 0.2$, i.e., we compute the maximum scores of the joined tuples $\hat{t}_1$ and $\hat{t}_2$. ∎

We are now ready to define answers to UNCQs. Intuitively, an answer to an UNCQ q is a pair (t, s) such that t is an answer to some NCQ q_i in q, and s is the maximum score of t among all q_i's.

Definition 4 (Answers to UNCQs). Given a PrefDatalog+/– ontology KB and a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, the set of *answers* to q over KB, denoted $ans(q, KB)$, is the set of all (t, g) such that there exists $(t, p) \in \bigcup_{i=1}^{l} ans(q_i, KB)$ with $g = max\{p \mid (t, p) \in \bigcup_{i=1}^{l} ans(q_i, KB)\}$.

Example 6. Consider the following UNCQ $q_1(X_1) = \exists Y_1\, place(X_1, Y_1) \vee \exists X_2\, serves(X_1, X_2)$ over the relations in Fig. 1 with the score function $score_{u_1}$. Then, $(p_1, 0.7) = (p_1, max(0.7, 0.6))$ is an answer to q_1. ∎

We next define top-k answers to UNCQs, which are intuitively at most k answers whose scores are always above or equal to the score of all non-top-k answers.

Definition 5 (Top-k Answers). Given a PrefDatalog+/– ontology KB and a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, a *top-$k$ answer* to q over KB, denoted $ans_k(q, KB)$, is a sequence of pairs (t, s) of atoms and scores such that:

(i) $ans_k(q, KB) \subseteq ans(q, KB)$,

(ii) $|ans_k(q, KB)| = min(k, |ans(q, KB)|)$, and

(iii) $s \geq s'$ for all $(t, s) \in ans_k(q, KB)$ and $(t', s') \in ans(q, KB) - ans_k(q, KB)$.

Example 7. Consider again the NCQ $q_1(X_1, X_2) = \exists Y_1, Y_2\, (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ over the relations in Fig. 1 with the score function $score_{u_1}$, and recall the scores of the non-entailed atoms of Example 2. Then, all the top-1 answers to q_1 are $\{(p_2, f_2)\}, \{(p_2, f_3)\}, \{(p_1, f_3)\}$, as all have the score of 0.6. ∎

In the rest of the paper, we refer to "a top-k answer", since (as shown above) ties in scores can lead to different sequences satisfying the conditions in the definition of top-k answer. We use the notation $pos(a, s)$ to refer to the position of an element a in a sequence s. For simplicity, we also slightly abuse notation by sometimes referring to answers as sequences of atoms (without the scores), and we sometimes treat sequences as sets.

4. Computing Top-k Answers

Our approach extends the RankJoin operator in [8] to work with UNCQs. The original RankJoin operator takes as input an integer k, a positive conjunctive query, a monotonic ranking function f, and m relations (where each tuple in each relation is accompanied by a score), and it yields the top-k joined tuples over the m relations, in descending order of their combined scores (computed using f). Specifically, it assumes that each of the m input relations is sorted by tuple-score in descending order, and tuples from the m relations are scanned and joined until k joined tuples are found such that the lowest among their scores is greater than or equal to a certain threshold.

Our ranking function f computes the scores $\hat{s}$, p, and g of tuples by applying the min and max operators to the scores of the atoms of HB_Φ, as defined in Definitions 2, 3, and 4, respectively. This generalizes conjunctions and disjunctions in classical logic; the framework can be easily adapted to other score computations.

We describe in detail our algorithm, since it is important to prove the correctness of the computation of the top-k answers, without the need to compute the scores of all the joined tuples.

***RankJoin-UNCQ* Algorithm.** The algorithm is shown in Fig. 2. It takes as input a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$ over $KB = (O, score)$, and an integer k, and it produces as output a set *top-k* of pairs (t, g), representing a top-k answer to $q(\mathbf{X})$ over KB, according to Definition 5.

Variables and initialization. *RankJoin-UNCQ* uses two local variables $querySet$ and *top-k*, where $querySet$ stores the NCQs $q_i(\mathbf{X})$ composing the input query $Q(\mathbf{X})$, and *top-k* stores the set of pairs (t, g) that represent the top-k answers to $Q(\mathbf{X})$ computed so far. Furthermore, we assume three global variables $relList$, $hashTab$, and Its, which are mappings from elements in $querySet$ into lists of relation names, sets of hash tables, and sets of iterators, respectively. $InitStructures$, called in line 3 of *RankJoin-UNCQ*, initializes these structures as follows: for each NCQ q_i in $querySet$, $relList$ contains the list with the (names of the) m relations $R_1, \ldots, R_m$ that appear in $Pos(q_i)$ (these are the positive relations over which q_i is evaluated), $hashTab$ contains a hash table for each R_j in $relList$, which is used to store the tuples of R_j that have already been scanned, and Its contains a set of iterators, one for each table R_j in $relList(q_i)$; iterator It_j in $Its(q_i)$ points to the next tuple to be scanned in R_j in the evaluation of q_i, so it is

initialized to the first tuple in R_j. Note that if q_i contains the same relation R multiple times (self-joins), $relList$ accordingly contains R multiple times, and $Its(q_i)$ contains an iterator for each occurrence of R. We assume, as done in [8], that relations are sorted by score from maximum to minimum; in this way, $getNext$ always retrieves the tuple with the maximum score from the ones not parsed yet.

Main loop. At each iteration, *RankJoin-UNCQ* randomly chooses an NCQ q from $querySet$ via $chooseQuery$ (line 5), and retrieves a set TS of pairs (t, p) by calling $getTuples$ (Fig. 4, line 6), which deals with the progressive evaluation of the NCQ q. If TS is empty (line 7), i.e., no more tuples can be obtained for the evaluation of q, *RankJoin-UNCQ* removes q from $querySet$ (line 8). Otherwise, $UpdateTopK$ is called for each pair (t, p) to update the current top-k answers, if needed. As each relation is sorted by score, then at each iteration, $getTuples$ returns the answers to q with the highest possible score. Thus, if *top-k* contains already k answers, and $getThreshold(q)$ is lower than the minimum score of the answers in *top-k*, then none of the rest of the answers to q can contribute to the top-k answers, and therefore q is removed from $querySet$ (line 14). The value $getThreshold(q)$ is the maximum tuple score obtainable in subsequent steps of the evaluation of q, computed as $max\{T_1, \ldots, T_m\}$, where $T_i = min\{score(r_1^1), \ldots, score(r_{i-1}^1), score(r_i^{last}), score(r_{i+1}^1), \ldots, score(r_m^1)\}$, where r_i^1 is the first tuple of R_i, and r_i^{last} is the last scanned tuple of R_i. Let T be the maximum tuple score obtainable in subsequent steps of the evaluation of the *whole* query $Q(\mathbf{X})$, computed by taking the maximum of the values provided by $getThreshold(q)$ for every NCQ q in $querySet$. *RankJoin-UNCQ* halts when (a) the lowest score of *top-k* is greater than T, and k answers have been found, or (b) $querySet$ becomes empty, i.e., no query is left to be evaluated. In every case, *RankJoin-UNCQ* returns the set *top-k* (line 15).

Function *getTuples*. The pseudocode for this subroutine is shown in Fig. 4; it deals with the progressive evaluation of an NCQ q, i.e., it progressively computes $ans(q, KB)$, as formalized in Definition 3. At each invocation, it yields a set TS of pairs (t, p) such that (i) t is an instance of the free variables in $\mathbf{X}$ obtained from $R_1, \ldots, R_m$ following the RankJoin strategy, and (ii) p is the computed score for t. As the first step, $getTuples$ retrieves $H_1, \ldots, H_m$ and $It_1, \ldots, It_m$ with the corresponding values for q from the global variables. The while-loop in line 5 finds the next relation to scan: if $relList(q)$ is not empty, then a relation $currRel$ is randomly chosen from it; if this relation has already been fully scanned (line 8), then it is removed from $relList(q)$, and the process is repeated until a relation that still has tuples to scan is found. If, however, all the relations have been entirely scanned, then all the possible join combinations have been tried, and no more tuples can be provided for the evaluation of $q(\mathbf{X})$, thus $getTuples$ returns the empty set (line 10). Otherwise, $currRel$ holds the next relation R_i to scan; function $getNext(It_i)$ fetches the pair $(t, score(t))$, where t is the next tuple in R_i that has not been scanned yet, and $score(t)$ is the corresponding score for t in R_i; as tuples in R_i are sorted in descending order relative to their scores, t has a score higher or equal than the scores of the tuples that were not chosen yet by the iterator in R_i. The pair $(t, score(t))$ provided by $getNext(It_i)$ is added to H_i (line 13)—recall that H_i is the hash table associated with R_i, and $join$ is invoked afterwards (line 14).

Function *join*. This function builds a set J of joined tuples resulting from joining tuple t with the tuples already in $H_1, \ldots, H_{i-1}, H_{i+1}, \ldots H_m$, according to q; $join$ effectively computes the join (Definition 2) in q for t with all the tuples from the other relations that have already been scanned.

Function *computeScore*. This function (Fig. 5, right), invoked on a set of joined tuples $\hat{t}$ and an NCQ $q(\mathbf{X})$, computes $\hat{s}$ as in Definition 2 for each $\hat{t}$ (line 15).

Procedure *UpdateTS*. For each of the obtained joined tuples $\hat{t}$, procedure $UpdateTS$ is called (line 17 of $getTuples$). $UpdateTS$ in Fig. 5 (left) projects $\hat{t}$ on the attributes provided by $Att(\mathbf{X})$, which are the attributes of the relations of $q(\mathbf{X})$ over which the variables in $\mathbf{X}$ are mapped to, yielding t. Next, $UpdateTS$ checks whether t occurs in TS with a score p lower than $\hat{s}$: in this case, the score p of t in TS is updated with $\hat{s}$. In the case that t does not appear in TS, the pair $(t, \hat{s})$ is added to TS. In line 18, function $getTuples$ returns set TS as output.

Procedure *UpdateTopK*. This procedure first checks if t is already in *top-k*, and, if so, then it updates the score of t in *top-k*, if necessary—it keeps the maximum score for t according to Definition 5. Otherwise, if *top-k* does not have k pairs yet, then (t, p) is added to *top-k*, else if $|$*top-k*$| = k$, and p is greater than the minimum score of the tuples in *top-k*, then the tuple in *top-k* with the minimum score is replaced with (t, p).

The following theorem states that our algorithm is correct.

Theorem 1. *Given a PrefDatalog+/– ontology KB and a UNCQ $q(\boldsymbol{X}) = \bigvee_{i=1}^{l} q_i(\boldsymbol{X})$, RankJoin-UNCQ correctly computes a top-k answer to $q(\boldsymbol{X})$.*

Algorithm *RankJoin-UNCQ*

Input: (i) PrefDatalog+/– ontology $KB = (O, score)$, (ii) UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, and (iii) integer $k > 0$.

Output: a top-k answer to $q(\mathbf{X})$.

Global Variables: $relList$, $hashTab$, Its.

1. $querySet := \{q_1(\mathbf{X}), \ldots, q_l(\mathbf{X})\}$;
2. *top-k* $:= \emptyset$;
3. $InitStructures(querySet)$; /* Fig.3 */
4. while *CanContinue*($querySet$, *top-k*, k) do
5. $q := chooseQuery(querySet)$;
6. $TS := getTuples(q)$; /* Fig. 4 */
7. if ($TS = \bot$) /* all the iterators in $Its(q)$ reached the end */
8. remove q from $querySet$
9. else
10. for each (t, p) in TS do
11. $UpdateTopK$(*top-k*, (t, p));
12. $T := getThreshold(q)$;
13. if (|*top-k*| $= k$ and $T \leq minScore$(*top-k*)) then
14. remove q from $querySet$ if present;
15. return *top-k*.

Figure 2: The main algorithm *RankJoin-UNCQ*.

Procedure $InitStructures$

Input: $querySet = \{q_1(\mathbf{X}), \ldots, q_l(\mathbf{X})\}$.

1. $relList :=$ empty mapping from queries q_i in $querySet$ to lists of relations;
2. $hashTab :=$ empty mapping from queries q_i in $querySet$ to sets of hash tables;
3. $Its :=$ empty mapping from queries q_i in $querySet$ to sets of iterators;
4. for each q_i in $querySet$ do
5. for each relation R_j appearing in $Pos(q_i(\mathbf{X}))$ do
6. add R_j to $relList(q_i)$;
7. add empty hash table H_j to $hashTab(q_i)$;
8. add It_j to $Its(q_i)$, and set It_j to the first tuple in R_j.

Figure 3: Initialization procedure.

Proof. All the relations in $Pos(q)$ are assumed to be sorted by score. Let *position*(*tuple*, R) be the position of *tuple* in the relation R. The proof is by contradiction. Suppose the k-th tuple of the *top-k* list yielded by *RankJoin-UNCQ* is t_k, with score g_k. Assume there is a joined tuple $\hat{t}$ of some $q_a(\mathbf{X})$, with $a \in \{1, \ldots, l\}$, such that its score $\hat{s}$ has $\hat{s} > g_k$, and $\hat{t}$ has not been processed by *getTuples*($q_a(\mathbf{X})$), thus (the projection of) $\hat{t}$ does not belong to *top-k*. Let $q_a(\mathbf{X})$ be of the form $\exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge \neg N_{m+1}(\mathbf{Z}_{m+1}) \wedge \cdots \wedge \neg N_{m+n}(\mathbf{Z}_{m+n})$, and $\hat{t}_1, \ldots, \hat{t}_m$ be the tuples of $R_1, \ldots, R_m$, respectively, whose join yields $\hat{t}$, and *score*($\hat{t}_1$) $\ldots$, *score*($\hat{t}_m$) the scores of $\hat{t}_1, \ldots, \hat{t}_m$ in $R_1, \ldots, R_m$, respectively. As *RankJoin-UNCQ* halted, $g_k \geq T$. This means that $g_k \geq T_a$, where T_a=*getThreshold*(q_a) = *max*$\{T_1, \ldots, T_m\}$. Since, by hypothesis, $\hat{s} > g_k$, it follows that $\hat{s} > T_a$, and thus $\hat{s}$ must be greater than every threshold T_j, with $i \in \{1, \ldots, m\}$. Let $(r_i^1, score(r_i^1))$ and $(r_i^{last}, score(r_i^{last}))$ be the first and the last scanned tuple of R_i along with their scores, respectively, with $1 \leq i \leq m$, and consider *any* $T_i \in \{T_1, \ldots, T_m\}$. By definition, T_i is equal to *min*$\{score(r_1^1), \ldots, score(r_i^{last}), \ldots, score(r_m^1)\}$, and the score $\hat{s}$ is computed as $\min\{$ *score*($\hat{t}_1$), $\ldots$, *score*($\hat{t}_m$), *score*($\hat{t}_{m+1}$), $\ldots$, *score*($\hat{t}_{m+n}$)$\}$. Since *score*($\hat{t}_i$) $\leq$ *score*(r_i^1) for every $i \in \{1, \ldots, m\}$, and *score*($\hat{t}_j$) ≤ 1 for every $j \in m + \{1, \ldots, n\}$, it is easy to see that, in order for $\hat{s} > T_i$ to hold, it must hold that *score*($\hat{t}_i$) $>$ *score*(r_i^{last}). This in turn implies that *position*($\hat{t}_i, R_i$) $>$ *position*(r_i^{last}, R_i), as R_i is scanned in descending score order. By applying

Function $getTuples$

Input: NCQ $q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$.

Output: Set TS of pairs (t, p), where t is an instance of the free variables in $\mathbf{X}$, and p is its score.

1. Let $\{H_1, \ldots, H_m\}$ be $hashTab(q)$; /* hash tables over relations $R_1, \ldots, R_m$ in $relList(q)$ */
2. Let $\{It_1, \ldots, It_m\}$ be $Its(q)$; /* iterators over relations $R_1, \ldots, R_m$ in $relList(q)$ */
3. $TS := \emptyset$; $currRel := \bot$;
4. $done := false$;
5. while ($relList(q) \neq \emptyset$ and $!done$) do
6. $\quad currRel := chooseRel(relList(q))$;
7. $\quad$ Let i be such that $currRel = R_i$;
8. $\quad$ if $!hasNext(It_i)$ then remove R_i from $relList(q)$ /*R_i has been completely scanned */
9. $\quad$ else $done := true$;
10. if ($relList(q) = \emptyset$) then return $\bot$;
11. Let i be such that $currRel = R_i$;
12. $(t, score(t)) := getNext(It_i)$;
13. Put $(t, score(t))$ in H_i;
14. $J := join(\Phi, H_1, \ldots, H_{i-1}, t, H_{i+1}, \ldots, H_m)$;
15. $JP := computeScore(J, q)$; /* Fig. 5 (left) */
16. for each $(\hat{t}, \hat{s}) \in JP$ do
17. $\quad UpdateTS((\hat{t}, \hat{s}), TS)$; /* Fig. 5 (right) */
18. return TS.

Figure 4: Function evaluating a CQ.

this reasoning to *every* $T_i \in \{T_1, \ldots, T_m\}$ (as $\hat{s}$ must be greater than *every* threshold T_i), we have that *score*$(\hat{t}_i) >$ *score*(r_i^{last}) for *every* $i \in \{1, \ldots, m\}$, implying that *position*$(\hat{t}_i, R_i) >$ *position*(r_i^{last}) for *every* $i \in \{1, \ldots, m\}$. This means that, since every relation is scanned in descending score order, every $\hat{t}_i$ must have been returned by *getNext*(It_i) (see line 12 of Figure 4) for every $i \in \{1, \ldots, m\}$, then the joined tuple $\hat{t}$ must have been produced by *getTuples*(q_a), contradicting the initial assumption. □

The following example shows how *RankJoin-UNCQ* works.

Example 8. Consider again the NCQ $q_1(X_1, X_2) = \exists Y_1, Y_2\ (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ over the relations in Fig. 1, asking for the pairs of places X_1 and foods X_2 such that X_2 is not served in X_1; consider the score function $score_{u_1}$, and assume that we are interested in a top-1 answer.

Since $querySet$ contains only q_1, $chooseQuery$ always returns q_1. $InitStructures$ initializes $relList$ (mapping q_1 to $[place, food]$), $hashTab$ (mapping q_1 to the set of hash tables $\{H_{place}, H_{food}\}$; both are empty), and Its (mapping q_1 to the set of iterators $\{It_{place}, It_{food}\}$, where It_{place} and It_{food} point to the tuples t_{11} and t_1, respectively).

Fig. 6 shows each iteration of *RankJoin-UNCQ*, while Fig. 7 shows each corresponding call of $getTuples$ within the iteration of *RankJoin-UNCQ*. The values in the rows of both tables show values of their variables after finishing each iteration and each function call, respectively.

In the first iteration, $getTuples$ is called and returns the empty set, as (assuming $place$ is chosen) only t_{11} is scanned, and no joins are produced yet. In the second call to $getTuples$, $food$ is chosen, and t_1 is scanned. The invocation of $join$ in this case produces the joined tuple $\hat{t} = t_{11}, t_1 = (p_1, pizzeria, f_1, pizza)$. As $t_{15} = (p_1, f_1)$ is in $serves$, which matches the values for the variables X_1 and X_2 set by $\hat{t}$, $\hat{t}$ is discarded and the empty set is returned.

In the third call to $getTuples$, $place$ is chosen, and t_{12} is scanned. Then, $join$ produces $\hat{t} = t_{12}, t_1 = (p_2, sushi\_bar, f_1, pizza)$. As no tuple in $serves$ matches the values for the variables X_1 and X_2 set by $\hat{t}$, $\hat{t}$ is not discarded, and its score $\hat{s}$ is computed as $min\{0.6, 0.8, 0.5\} = 0.5$, as $serves(p_2, f_1)$ is a non-entailed atom, whose score is 0.5. Then, $UpdateTS$ projects $\hat{t}$ onto $Att(X)$ yielding $t = (p_2, f_1)$, which is added to TS with score 0.5 and returned. Then, *RankJoin-UNCQ* adds $((p_2, f_1), 0.5)$ to *top-k*, and $getThreshold(q_1)$ yields $T = 0.6$. Since $lowerScore(\textit{top-k}) = 0.5$ is lower than 0.6, q_1 is not removed from $querySet$, and the algorithm does not stop.

Procedure *UpdateTS*

Input: A pair $(\hat{t}, \hat{s})$, a set TS of pairs (t, p).

1. $t := project(\hat{t}, Att(\mathbf{X}))$;
2. if $((t, p) \in TS$ and $\hat{s} \geq p$ then
3. update p with $\hat{s}$;
4. if $((t, p) \notin TS$ then
5. put $(t, \hat{s})$ in TS.

Function *computeScore*

Input: A set J of tuples $\hat{t} = t_1, \ldots, t_m \mid (t_j, score(t_j)) \in H_j$, a UNCQ $q(\mathbf{X}) = \exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge \neg N_{m+1}(\mathbf{Z}_{m+1}) \wedge \cdots \wedge \neg N_{m+n}(\mathbf{Z}_{m+n})$.

Output: A set JP of pairs $(\hat{t}, \hat{s})$.

1. $JP := \emptyset$;
2. for each $\hat{t} \in J$ do
3. *discard* := false;
4. $\hat{s} := min(score(t_j), \ldots, score(t_m))$;
5. for each N_j do
6. Let $q_j(\mathbf{X}) = \exists \mathbf{Y} R_1(\mathbf{Z}_1) \wedge \cdots \wedge R_m(\mathbf{Z}_m) \wedge N_j(\mathbf{Z}_j)$;
7. If there exists tuple t_j in N_j such that the projection of $t_1, \ldots, t_m, t_j$ onto $\mathbf{X}$ is an answer to q_j then
8. *discard* = true; break
9. else $\hat{s} := min\{\hat{s}, 1 - score(a_j)\}$, where a_j is the
10. atom such that $N_j(a_j) \notin O$;
11. if *discard* = false add $(\hat{t}, \hat{s})$ to JP;
12. *return* JP.

Figure 5: Updating TS with a new scored tuple (left), and computing the scores of joined tuples (right).

	querySet	*getTuples*	*top-1*	$T = getThreshold(q_1)$
1	$\{q_1\}$	$\emptyset$	$\emptyset$	$max\{min\{0.6, 0.8\}, min\{0.6, 0.8\}\} = 0.6$
2	$\{q_1\}$	$\emptyset$	$\emptyset$	$max\{min\{0.6, 0.8\}, min\{0.6, 0.8\}\} = 0.6$
3	$\{q_1\}$	$\{((p_2, f_1), 0.5)\}$	$\{((p_2, f_1), 0.5)\}$	$max\{min\{0.6, 0.8\}, min\{0.6, 0.8\}\} = 0.6$
4	$\{q_1\}$	$\{((p_2, f_2), 0.6)\}$	$\{((p_2, f_2), 0.6)\}$	$max\{min\{0.6, 0.8\}, min\{0.6, 0.7\}\} = 0.6$

Figure 6: Trace of *RankJoin-UNCQ* with query $q_1(X_1, X_2) = \exists Y_1, Y_2\, (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$ for user u_1.

	R_i	It_{place}, It_{food}, H_{place}, H_{food}	$\hat{t}$, with $(\hat{t}, \hat{s}) \in J$	$\hat{s}$, with $(\hat{t}, \hat{s}) \in J$	TS
1	*place*	$t_{12}, t_1, \{(t_{11}, 0.6)\}, \emptyset$	$\emptyset$	0	$\emptyset$
2	*food*	$t_{12}, t_2, \{(t_{11}, 0.6)\}, \{(t_1, 0.8)\}$	$\{\hat{t} = t_{11}, t_1\}$	0	$\emptyset$
3	*place*	$t_{13}, t_2, \{(t_{11}, 0.6), (t_{12}, 0.6)\}, \{(t_1, 0.8)\}$	$\{\hat{t} = t_{12}, t_1\}$	$\{min\{0.6, 0.8, 0.5\} = 0.5\}$	$\{((p_2, f_1), 0.5)\}$
4	*food*	$t_{13}, t_3, \{(t_{11}, 0.6), (t_{12}, 0.6)\}$, $\{(t_1, 0.8), (t_2, 0.7)\}$,	$\{(\hat{t}_1 = t_{11}, t_2)$, $(\hat{t}_2 = t_{12}, t_2)\}$	$\{min\{0.6, 0.7, 0.7\} = 0.6\}$	$\{((p_2, f_2), 0.6)\}$

Figure 7: Trace of $getTuples$ on each call for q_1 for user u_1.

In the fourth call to $getTuples$, *food* is chosen, t_2 is scanned, and *join* produces $\hat{t}_1 = t_{11}, t_2$ and $\hat{t}_2 = t_{12}, t_2$. As $t_{14} = (p_1, f_2)$ is in *serves*, which matches the values for the variables X_1 and X_2 set by $\hat{t}_1$, $\hat{t}_1$ is discarded. Tuple $\hat{t}_2$, instead, is mantained, and its score is computed as $min\{0.6, 0.7, 0.7\} = 0.6$, as $serves(p_2, f_2)$ is a non-entailed atom, whose score is 0.3. Function $UpdateTS$ projects $\hat{t}_2$ onto $Att(X)$ yielding $t = (p_2, f_2)$, which is added to TS with score 0.6 and returned. After replacing $((p_2, f_1), 0.5)$ into *top-k* with $((p_2, f_2), 0.6)$ to *top-k*, *RankJoin-UNCQ* calls $getThreshold(q_1)$ that returns 0.6. Since $lowerScore$(*top-k*) is equal to 0.6, q_1 is removed from $querySet$, and the algorithm stops, producing the top answer $\{((p_2, f_2), 0.6)\}$. ∎

5. Answering Queries for Groups

Group decision making [15] comes up in many scenarios in which important choices are made; for instance, consider a committee put together by a funding agency to decide how to select which research projects to fund. Here, each committee member submits their opinion on how the proposals should be ranked, and the committee wishes to reach a decision that *adequately* represents each individual member's views. Some of the issues that can arise in this scenario are: Is it possible for members to strategically misrepresent their opinions in order to push their agenda? Can they make a fair decision in the presence of conflicting preferences?

One way of tackling these issues is via the top-k query answering mechanisms discussed in the previous sections—here, we explore different ways in which the questions raised above can be addressed. The central issue is thus the combination of individual preferences to produce a ranking of elements based on an aggregated view of the group. There are two main computational challenges in accomplishing this: (i) the fact that disagreement inevitably comes up and must be resolved in some principled manner, and (ii) tractability is even harder to accomplish compared to the single-user case (as it involves the additional aggregation of potentially contradicting preferences of a potentially large collection of users). To this end, in this section we extend the framework presented so far in the following ways:

- Allow the inclusion of a set of users and their respective score functions.
- Define two specific strategies to compute the aggregated view of the group; the first computes individual top-k answers for each user and then combines them into a single one representing the group, and the other performs the combination step first and then answers the top-k over the result.
- Explore different semantic properties that describe different aspects of fairness, as well as how results change in response to changes in individual preferences.

Given a set of n users $\mathcal{U} = \{u_1, \dots u_n\}$ and $a \in HB_\Phi$, we denote with $score_{u_i}(a)$ the score assigned by user u_i to an atom a, and assume that this mapping is defined for all pairs of ground atoms entailed by the ontology and users in the group. Here, we naturally assume that one global Datalog+/– ontology is common to all users.

Example 9. In the relations in Fig. 1, the last three columns of each table specify the score function for three different users. For instance, for the relation $food$, user u_1 assigns a score of 0.8 to $food(f_1, pizza)$, while users u_2 and u_3 assign it 0.3 and 0.7, respectively. Now, consider the query $q_1(X_1, X_2) = \exists\, Y_1, Y_2\ (place(X_1, Y_1) \wedge \neg serves(X_1, X_2) \wedge food(X_2, Y_2))$—the top-1 answer for u_1 was computed in Example 8; but we would now like to know the top-1 answer considering also the scores from users u_2 and u_3. ∎

As mentioned above, the main challenge in query answering relative to preferences of a group of users is that the user preference models may be in disagreement with each other. The study of preference aggregation strategies to address this problem in ontological query answering was first proposed in [6], where an operator is defined for the aggregation of n individual strict partial orders (SPOs), each representing the preferences of an individual. The following definition is an adaptation of aggregation operators where the preference models are based on score functions.

Definition 6. Given a set R of tuples and n score functions $score_{u_1}, \dots, score_{u_n}$ over the tuples in R, a *score aggregation operator* $\biguplus(R, score_{u_1}, \dots, score_{u_n})$ yields a score function $score^*$ over the tuples in R.

These aggregation operators are quite general, asking only to satisfy the property that the resulting preference relation is also a strict weak order; depending on the application, other properties may be desirable, such as the additional ones studied in [6] for SPOs. For the particular setting of score-based preference functions, the problem of aggregating such functions is similar to that of rank fusion that we discussed as related work. In Section 6, we select some specific ranking aggregations from the literature and implement them in our framework.

In the rest of this section, we analyze two approaches to compute an answer to a top-k query for a set of users: *aggregation-last* and *aggregation-first*. Both strategies adapt techniques developed in [6] to the special case of score functions, and the general case of UNCQs—previous work focused on disjunctive atomic queries and preference relations expressed as SPOs.

Algorithm AggLast

Input: (i) Datalog+/– ontology O;
(ii) set of users $\mathcal{U} = \{u_1, \ldots, u_n\}$, where each u_i has associated score function $score_{u_i}$;
(iii) UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$; (iv) integer $k > 0$; and (v) score aggregation operator $\biguplus$.

Output: A top-k answer to $q(\mathbf{X})$.

1. for every user $u_i \in \mathcal{U}$ do
2. *k-ans*$_i$:= *RankJoin-UNCQ*$((O, score_{u_i}), q(\mathbf{X}), k)$;
3. Res := apply $\biguplus$ to *k-ans*$_1, \ldots,$ *k-ans*$_n$;
4. return k highest-scoring elements in Res.

Figure 8: Agg-Last top-k query answering.

5.1. Aggregation-last Query Answering

The first strategy that we discuss computes a top-k answer for each individual and then applies an aggregation operator to the results in order to obtain a single top-k answer. We begin by formalizing the notion of top-k aggregation-last answers. Given a Datalog+/– ontology O, a set of n users $\mathcal{U} = \{u_1, \ldots, u_n\}$ with score functions $score_{u_1}, \ldots, score_{u_n}$, a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, and an aggregation operator $\biguplus$, in the following, we denote with *k-ans*$_i$ the set $ans_k(q, (O, score_{u_i}))$, i.e., the top-$k$ answers for user u_i.

Note that *k-ans*$_i$ consists of at most k pairs of the form (t, scr_t), where $t \in D$, and scr_t is as in Definition 5. Intuitively, user u_i is inducing a score-based preference relation to the tuples in D, where the score for a tuple not appearing in *k-ans*$_i$ is zero; therefore, in the following definition, we use *k-ans*$_i$ to represent the set of top-k answers for user u_i as well as to represent the score-based preference relation that it induces.

Definition 7 (Agg-last Top-k Answers). Given a Datalog+/– ontology O, a set of n users $\mathcal{U} = \{u_1, \ldots, u_n\}$ with corresponding score functions $score_{u_1}, \ldots, score_{u_n}$, a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, and an aggregation operator $\biguplus$, let $A = \{(t, scr_{t,u_i}) \mid (t, scr_{t,u_i}) \in \bigcup_{u_i \in \mathcal{U}} k - ans_i\}$ and $TU = \{t \mid \exists (t, scr) \in A\}$. An *Agg-last* top-$k$ answer for $\mathcal{U}$ is a set *ansG-last*$_k(q, O, \mathcal{U})$ such that:

(i) *ansG-last*$_k(q, O, \mathcal{U}) \subseteq A$,

(ii) for each $(t, score^*(t)) \in$ *ansG-last*$_k(q, O, \mathcal{U})$, we have $score^*(t) \geq score^*(t')$ for every tuple t' appearing in A but not in *ansG-last*$_k(q, O, \mathcal{U})$, where $score^* = \biguplus(TU,$ *k-ans*$_1, \ldots,$ *k-ans*$_n)$, and

(iii) $|$*ansG-last*$_k(q, O, \mathcal{U})| = min(k, |TU|)$.

We now show how to compute Agg-Last top-k answers for a set of users by leveraging Algorithm *RankJoin-UNCQ* described in Section 3. The main strategy works as follows: (i) for each user u_i, compute the top-k answers to query $q(\mathbf{X})$ using Algorithm *RankJoin-UNCQ*; and (ii) aggregate the answers of all users into one top-k answer using a score aggregation operator $\biguplus$. Clearly, in step (i), *RankJoin-UNCQ* could be replaced by any algorithm that computes top-k answers for preference-based Datalog+/– ontologies. However, all the results shown in this paper assume the use of Algorithm *RankJoin-UNCQ*. Algorithm *AggLast* (cf. Fig. 8) performs this process.

For the second step, in this work, we consider the following aggregation operators based on *ranking aggregation methods* [16]: $\biguplus_x$ with $x \in \{max, min, borda, sum, avg\}$. For every tuple t in R, operator $\biguplus_{max}(R, score_1, \ldots, score_n)$ assigns to t the maximum score among all $score_i(t)$. Operators $\biguplus_{min}$, $\biguplus_{avg}$, and $\biguplus_{sum}$ apply minimum, average, and sum, respectively. Clearly, $\biguplus_{max}$ (resp., $\biguplus_{min}$) is equivalent to the max-based (resp., min-based) linear combinator mentioned in [16]. For $\biguplus_{borda}$ (Borda count in [16]), each user ranks their top-k answers. For each user, the top-ranked answer is given k points, the second-ranked one $k-1$ points, and so on. Note that we first need to normalize the scores in each individual rank as shown in [16].

Example 10. Consider again query $q_1(X_1, X_2)$ from Example 9. First, the top-1 answer for each user is computed using *RankJoin-UNCQ*. We have then that the top-1 answer for u_1 is *k-ans*$_1 = \{((p_1, f_1), 0.6)\}$ (as computed in

Example 8); possible top-1 answers for u_2 and u_3 are *k-ans*$_2 = \{((p_1, f_4), 0.5)\}$ and *k-ans*$_3 = \{((p_1, f_4), 0.7)\}$, respectively. The universe of elements in this case (i.e., the set TU in Definition 7) is $\{(p_1, f_1), (p_1, f_4)\}$. The operator $\biguplus_{max}$ yields $((p_1, f_1), 0.6)$ and $((p_1, f_4), 0.7)$; thus, the top-1 answer for the group is $((p_1, f_4), 0.7)$ using $\biguplus_{max}$. ■

An alternative to the aggregation operator is to consider voting mechanisms from social choice (as proposed in [6]). For example, $\biguplus_{plurality}$ can be used, which works as follows. First, compute the top-k answers for each user (a user's top-k choices are taken as their top-preferred items). Then, each items' frequency for all the users are summed up, and the k items with the highest number of votes win. The final scores are 1 to the chosen tuples and 0 to the rest.

Example 11. If we use the operator $\biguplus_{plurality}$ in Example 10, then (p_1, f_4) has two votes, and (p_1, f_1) has one vote; therefore, the same answer as before is obtained through plurality voting. ■

The correctness of Algorithms 8 is proved by the following proposition.

Proposition 1. *Given a Datalog+/– ontology $KB = (D, \emptyset)$, a set of users $\mathcal{U}$, a UNCQ $q(\boldsymbol{X}) = \bigvee_{i=1}^{l} q_i(\boldsymbol{X})$, and an aggregation operator $\biguplus$, Algorithm* AggLast *correctly computes an Agg-Last top-k answer for $\mathcal{U}$.*

5.2. Aggregation-first Query Answering

Our second proposed approach applies the aggregation operator to the input tables, effectively merging the users' preferences into a single score assignment. This can be seen as the construction of a single virtual user that aggregates the preferences of all the individuals from the group; the top-k answers are then computed over this new score assignment using the algorithm described in Section 3. We formalize this in the following definition.

Definition 8 (Agg-First Top-k answers). Given a relational schema $\mathcal{R}$ and a Datalog+/– ontology O on $\mathcal{R}$, a set of n users $\mathcal{U} = \{u_1, \ldots, u_n\}$ with score functions $score_{u_1}, \ldots, score_{u_n}$, a UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$, and an aggregation operator $\biguplus$, an *Agg-First* top-k answer for $\mathcal{U}$ is the set *ansG-first*$_k(q, O, \mathcal{U}) = ans_k(q, KB')$ where $KB' = (KB, score^*)$ is a preference-based Datalog+/– ontology and $score^* = \biguplus(R, score_{u_1}, \ldots, score_{u_n})$ for every $R \in \mathcal{R}$.

Algorithm AggFirst (Fig. 9) implements Agg-First top-k answering using *RankJoin-UNCQ* (Fig. 2) to compute the top-k answers to a UNCQ based on the aggregated preference relation. Analogously to AggLast (Fig. 8), the call to *RankJoin-UNCQ* in step 5 can be replaced by any algorithm that computes the top-k answers for a preference-based Datalog+/– ontology.

Example 12. Consider again the CQ $q_1(X_1, X_2)$ from Example 9. The first step in *AggFirst* is to aggregate the scores of the different users in every relation appearing in the query. In this case, we have relations $food$, $place$, and $serves$. The following table summarizes the result of aggregating the individual scores determined by score functions $score_{u_1}$, $score_{u_2}$, and $score_{u_3}$ for each tuple in these relations. We assume the aggregation operator $\biguplus_{max}$ as in Example 10.

food	
tuple	$score^*$
t_4	0.9
t_1	0.8
t_2	0.7
t_3	0.6
t_5	0.5

serves	
tuple	$score^*$
t_{16}	0.9
t_{14}	0.7
t_{17}	0.65
t_{15}	0.6

place	
tuple	$score^*$
t_{11}	0.8
t_{12}	0.5
t_{13}	0.2

After the aggregation step, Algorithm *RankJoin-UNCQ* is called for q_1 as before, but the function $score^*$ is used instead. Note that Algorithm *RankJoin-UNCQ* assumes that the tables appearing in q_1 are sorted by $score^*$ in descending order. The top-1 answer in this case is $\{((p_1, f_4), 0.8)\}$. ■

Proposition 2. *Given a Datalog+/– ontology $KB = (D, \emptyset)$, a set of users $\mathcal{U}$, a UNCQ $q(\boldsymbol{X}) = \bigvee_{i=1}^{l} q_i(\boldsymbol{X})$, and an aggregation operator $\biguplus$, Algorithm* AggFirst *correctly computes an Agg-First top-k answer for $\mathcal{U}$.*

Algorithm AggFirst

Input: (i) Datalog+/– ontology O;
(ii) set of users $\mathcal{U} = \{u_1, \ldots, u_n\}$, where each u_i has associated score function $score_{u_i}$;
(iii) UNCQ $q(\mathbf{X}) = \bigvee_{i=1}^{l} q_i(\mathbf{X})$; (iv) integer $k > 0$; and (v) score aggregation operator $\biguplus$.

Output: A top-k answer to $q(\mathbf{X})$.

1. Let $R_1, \ldots, R_m$ be all relations that appear in $q(\mathbf{X})$;
2. For each R_i do
3. $\quad Res_i := \biguplus \left(R_i, score_{u_1}, \ldots, score_{u_n}\right)$;
4. $\quad score^*(t) := Res_i(t)$ for every $t \in R_i$;
5. return *RankJoin-UNCQ*$((KB, score^*), q(\mathbf{X}), k)$.

Figure 9: Agg-First top-k query answering.

Note that, in certain cases, it may not be necessary to produce the aggregated scores for the entire tables used in the query; instead, we compute the required tuples on demand to avoid unnecessary computations. The specific strategy to accomplish this depends on the aggregation operator used, and may not be possible in general. One example of an operator for which relatively minor changes to our algorithm suffice to implement this optimization is $\biguplus_{max}$.

Worst case running times. The worst-case time complexity of our algorithms is as follows. Let u be the number of users, q be the number of disjuncts (NCQs) in the UNCQ input, r be the maximum number of positive relations in one NCQ, k be the number of answers returned, and n_{max} be the maximum number of tuples in any table.

In the worst case, RankJoin-UNCQ runs in time $O(n_{max} * r * q)$, since for each disjunct in the input query and each positive relation (NCQ) in that disjunct, it must scan all tuples. *AggFirst* must first aggregate all scores and then answer the query; in the worst case, this is $O(n_{max} * r * u + n_{max} * r * q)$. *AggLast* must first answer the query for each user, and then aggregate the individual top-k answers; in the worst case, this is $O(u * n_{max} * r * q + u * k)$.

5.3. Semantic Properties of Aggregation-last Top-k Querying

An interesting line of research is the study of the different properties that are satisfied (or violated) by the different approaches that address top-k query answering. Several works in the literature study semantic properties for top-k answers over certain and uncertain data; among them, we refer the reader to [17, 18]. As we intend to model the preference relation that represents the preferences of a group of individuals as a whole, we have focused instead on a set of properties that are based on the ones usually studied in social choice theory [19]; such properties were studied in [6] for strict partial orders (that, for their generality, also offer less guarantees). Nevertheless, it is important to note that most of the properties that can be found in [17, 18] are subsumed and adapted for the group preference scenario.

An in-depth study of such properties is out of the scope of this paper; however, we present a brief discussion to show the kind of results that we have obtained so far and are pursuing further. The Monotonicity and Stability properties are adapted from our previous work on answering top-k queries over groups in a more general setting [6].

The following properties refer to the effects of improving and lowering the scores of tuples. We denote as $Ans_k(q, KB)$ the set containing all the top-k answers $ans_k(q, KB)$.

Monotonicity 1: Given $k > 0$, let $r \in Ans_k(q, KB)$ for $\mathcal{U} = \{u_1, \ldots, u_n\}$, $t \in r$, and $u_i \in \mathcal{U}$ be such that $score_{u_i}(t) = v$. If $u_i' \notin \mathcal{U}$ is identical to u_i except that $score_{u_i}(t) = v'$ with $v' > v$, then there exists $r' \in Ans_k(q, KB)$ for $\mathcal{U}' = \{u_1, \ldots, u_{i-1}, u_i', u_{i+1}, \ldots, u_n\}$ such that $t \in r'$.

Intuitively, this states that if a tuple t is in a top-k answer, it is still in a top-k answer if the score function of some user changes so that t's (and only its) score is increased.

Monotonicity 2: Given $k > 0$, let t be a tuple such that there is no $r \in Ans_k(q, KB)$ for $\mathcal{U} = \{u_1, \ldots, u_n\}$ with $t \in r$, and let $u_i \in \mathcal{U}$ be such that $score_{u_i}(t) = v$. If $u_i' \notin \mathcal{U}$ is identical to u_i except that $score_{u_i}(t) = v'$ with $v' < v$, then there does not exist $r' \in Ans_n(q, KB)$ for $\mathcal{U}' = \{u_1, \ldots, u_{i-1}, u_i', u_{i+1}, \ldots, u_n\}$ such that $t \in r'$.

Conversely, Monotonicity 2 states that if a tuple t does not appear in any answer, it cannot appear in any answer if the score function of a user changes so that t's (and only its) score is decreased.

Faithfulness: Given $k > 0$, let a and b be two tuples such that for all $u \in \mathcal{U} = \{u_1, \ldots, u_n\}$ it holds that $score_u(a) > score_u(b)$; then, there is no top-k answer $r \in Ans_k(q, KB)$ for $\mathcal{U}$ such that $pos(b, r) > pos(a, r)$.

This property, adapted from [20], states that if every user assigns a greater value to tuple a than tuple b, then a must appear before b in every possible top-k answer. The following proposition shows for which aggregation operators algorithm Algorithm *AggLast* satisfies the above properties.

Proposition 3. *Algorithm* AggLast *satisfies:*

(1) Monotonicity 1 and 2, for operators $\biguplus_x$ *with* $x \in \{plurality, max, sum, avg\}$*; and*

(2) Faithfulness for operators $\biguplus_x$ *with* $x \in \{max, sum, avg\}$.

Proof (sketch). Monotonicity 1: Consider *aggLast* and $\biguplus_{plurality}$, and let *k-ans*$_i$ be top-k answers to q for each u_i; since only the score function for u_i changes, then we can assume that the same other $n-1$ answers are obtained when computing the top-k answer for each user in $\mathcal{U}'$. Clearly, all atoms appearing in *k-ans*$_i$ maintain the same number of votes, including t; therefore, t remains in the top-k answer for $\mathcal{U}$.

Consider *AggLast* and $\biguplus_{max}$, and let $scr_{\mathcal{U}}(t)$ be the score for t in answer r; this score is computed as the maximum among all the scores assigned to t in all individual top-k answers. Then, as only the score of t changes in *k-ans*$'_i$, we have that $scr_{\mathcal{U}'}(t)$, the score for t in answer r', is greater than $scr_{\mathcal{U}}(t)$; as the scores for the rest of the answers remain the same, t must remain within the top-k answer. An analogous argument holds for $\biguplus_{avg}$ and $\biguplus_{sum}$.

Monotonicity 2: Consider again the argument for Monotonicity 1 and $\biguplus_{plurality}$; clearly, as t received no votes prior to the change in score, it will not receive any votes after, and thus cannot appear in any top-k answer. For $\biguplus_{max}$, $\biguplus_{avg}$, and $\biguplus_{sum}$, a similar argument can be made—if t did not influence the value of max, avg, or sum before the change to make the element appear in a top-k answer, it cannot do so after its score is lowered even further.

Faithfulness: First, we clarify that this property is not satisfied by $\biguplus_{plurality}$, because of the way votes are assigned: it does not matter that all users score element a higher than b—as long as both elements are in individual users' top-k answers, they both receive one vote. Thus, ties in number of votes leads to the existence of top-k answers that exchange the positions of these elements.

For max, avg, and sum, a similar argument to the one used for Monotonicity can be applied; these functions enjoy the uniformity property, so if $score_{u_i}(a) > score_{u_i}(b)$ for all $u_i \in \mathcal{U}$, then the aggregate function applied to each side preserve the relation. □

Comparison with previous work. It is interesting to contemplate the relationship between the semantic properties that hold in the more general models of [6] (see Fig. 6) and the score function-based ones in this paper. Of course, the positive results obtained there carry over to this setting; unfortunately, some of the negative results also carry over. One interesting property is *Stability*, which is included in [6] (see Fig. 6) as two separate properties, one in the case in which a new element is added and the other in which an existing element is removed; below, we reformulate the former for the present model; we use the notation KB_{add} to refer to the knowledge base that results from adding a new element c, and assume that all score functions are extended accordingly:

Stability-Add: For each top-k answer $r \in Ans_k(q, KB)$ for $\mathcal{U} = \{u_1, \ldots, u_n\}$, there is a top-$k$ answer $r' \in Ans_k(q, KB_{add})$ such that either $r = r'$ or (i) $r' - r = \{c\}$ and (ii) let $r = (a_1, \ldots, a_k)$ and $r' = (b_1, \ldots, b_k)$; for each pair b_i, b_j such that $1 \leq i < j \leq k$ and $b_i = a_{i'}$ and $b_j = a_{j'}$, it holds that $i' < j'$.

Essentially, the property ensures that if a new element is added, the result is either unaltered or the new element appears, and the relative order of all the rest stays the same. This property is not satisfied in the general SPO-based model, but one could hope that the added structure of scores would help (at least for some operators). Unfortunately, it is simple to construct counterexamples for the operators studied here. For $\biguplus_{plurality}$, this is accomplished by adding the element c in such a way that other elements are pushed out of the individual top-k answers, causing the number of votes to change so that the final order is altered. The same kind of counterexample is possible for the cases of avg and sum. For $\biguplus_{max}$, the same strategy leads to a counterexample for the property; for the property to be violated, there must exist two elements, a and b, such that they are swapped in the top-k answer after the addition of the new element. That is, $scr(a) > scr(b)$ and $scr'(b) > scr'(a)$, where $scr(.)$ and $scr'(.)$ denote the scores associated with the element in the answer prior to the addition and after, respectively. Suppose now that a loses its position (to c) in an individual top-k answer; if that was the score that determined $scr(a)$, then clearly $scr'(a)$ can be lower than $scr'(b)$.

Other negative results explored in previous work still hold here, with the same counterexamples found for the more general model. These include the other version of the Stability property for Plurality, called S2 in [6] (where an element is removed), as well as those for Fairness (Unanimous Winner, both Stability properties, and Non-Dictatorship when $k = 1$). We refer the interested reader to [6] (see Appendix B.5) for the counterexamples.

The properties we discussed are most naturally formulated for the aggregation-last approach; for aggregation-first, the relationship between scores given to basic tuples by users and the final answers to queries is not as clear—we are currently investigating what properties can be used to compare these approaches in a principled manner.

6. Experimental Evaluation

In this section, we report the results of evaluating our algorithms on real-world data, analyzing both the running time and the quality of their results.

6.1. Implementation and Hardware

We implemented the algorithms by extending the Datalog+/– query answering engine [9, 10], which involved adding query answering with score-based preferences for a group of users. We implemented four algorithms for AggLast and two for AggFirst—the code and the dataset will be made available as open source. For AggLast, these algorithms are LastBorda, LastPlurality, LastHitMax, and LastHitMin, which use $\biguplus_x$ with x among $borda$, $plurality$, max, and min, respectively. We use score normalization [16] for LastHitMax and LastHitMin, and Borda normalization [16] for Borda. To compute the final scores, we used linear combination: the sum of normalized Borda scores for Borda, and the sum of hits (the number of users that include an element in their individual result) multiplied by max (resp., min) of score normalizations for LastHitMax (resp., LastHitMin). For AggFirst, we implemented FirstAvg and FirstMax, which use $\biguplus_x$ with x among avg and max, respectively.

The entire implementation was done in Java; all runs were done on an Intel Core i7 processor at 2.2 GHz and 8GB of RAM, under the MacOS X 10.6.8 OS and a Sun JVM Standard Edition with maximum heap size of 512 MB. To minimize experimental variation, all results are averages of three independent runs.

6.2. Experimental Setup

Inputs to our system consist of tuples $(q, O, \mathcal{U}, k)$, where q is a query, O is the Datalog+/– ontology, $\mathcal{U}$ is a group preference model, and k is the number of query results.

Data. All runs were carried out using an ontology built on data from the Yelp Dataset Challenge [21], which contains $11,537$ businesses in the Phoenix (USA) metropolitan area, $8,282$ check-ins, $43,873$ users, and $229,907$ reviews—each business has one or more associated categories.

The Datalog+/– ontology was constructed as follows. We created seven different relations: $place$, $placeType$, $cuisine$, $food$, $isPlaceType$, $servesCuisine$, and $servesFood$; e.g., given the "sushi-bar", "Asian", and "sushi" categories for a business x, called $name$ in city y, we have $place(x, y, name)$, $placeType(sushi\text{-}bar)$, $cuisine(asian)$, $food(sushi)$, $isPlaceType(x, sushi\text{-}bar)$, $servesCuisine(x, asian)$, and $servesFood(x, sushi)$.

The set of dependencies is $\Sigma = \{isPlaceType(X, Y) \rightarrow placeType(Y)$; $servesCuisine(X, Y) \rightarrow cuisine(Y)$; $servesFood(X, Y) \rightarrow food(Y)$; $isPlaceType(X, Y) \rightarrow \exists Z, T\ place(X, Z, T)$; $servesCuisine(X, Y) \rightarrow \exists Z, T\ place(X, Z, T)$; $servesFood(X, Y) \rightarrow \exists Z, T\ place(X, Z, T)\}$.

User Preferences. We used the preference dataset[1] gathered in previous work [6], which consists of preferences for 49 users over *cuisine*, *type of food*, and *type of place* (breakfast, lunch, and dinner). Users entered their preferences as strict partial orders (represented as graphs) via a GUI. For each such graph G, we computed the *layer* corresponding to each vertex (the undominated nodes comprise layer 1, and so on). If the number of layers of G is n, and vertex v belongs to layer ℓ, then the score of v is computed as a random number in $[1 - (\ell/n), 1 - (\ell - 1/n)]$ (e.g., if the users specified that they prefer bagels over sushi, then the score of $food(bagel)$ is higher than that of $food(sushi)$).

[1] https://github.com/personalised-semantic-search/dataset_qualitative

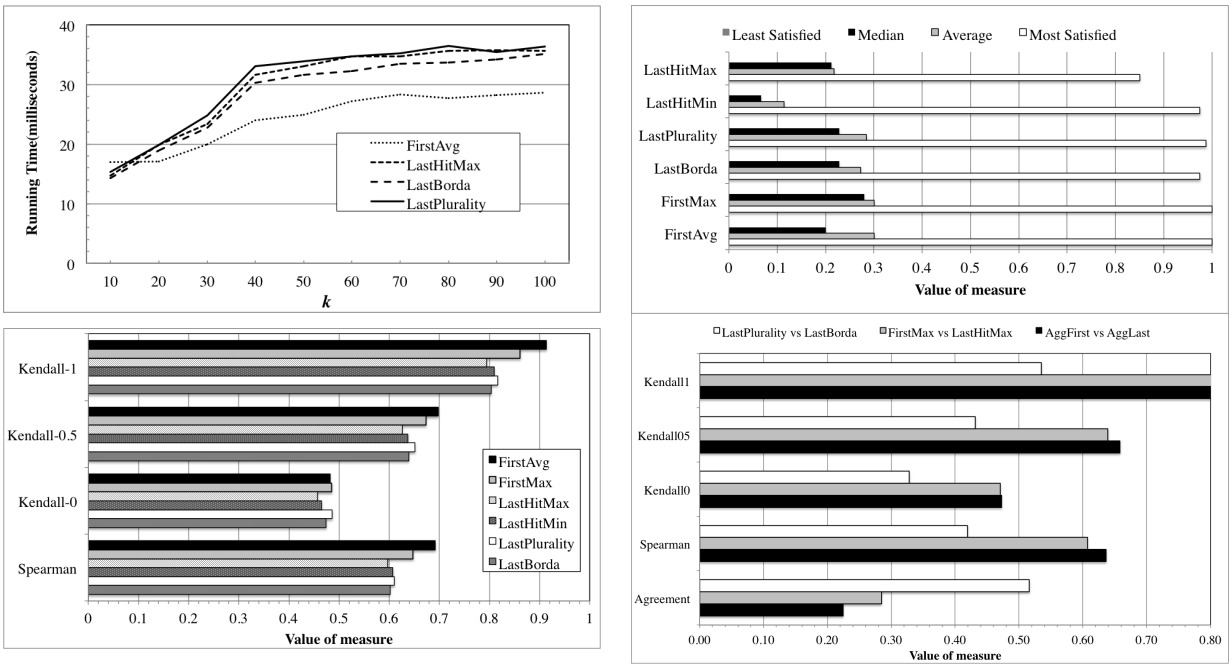

Figure 10: *Top left:* Running time performance for the real-world Yelp Dataset Challenge (see above); *Top right:* Quality evaluation (Agreement); *Bottom left:* Quality evaluation (other metrics); *Bottom right:* Comparison of methods.

For each business, the Yelp dataset provides a numerical rating from 0 to 5. We used this information to compute the scores of tuples of the relation $place$ as a random number in the interval $[(r-\sigma)/(5+\sigma), (r+\sigma)/(5+\sigma)]$, where r is the rating, and σ is the associated standard deviation. To compute the scores for the ground atoms $isPlaceType(p,t)$, $servesCuisine(p,c)$, and $servesFood(p,f)$, we computed the maximum between the score of $place(p,y,name)$ and the scores of $placeType(t)$, $cuisine(c)$, and $food(f)$, respectively.

Group Definition. We used the same groups from [6]: 19 groups of 3 to 7 users. Given that there are three types of meals, this produced an overall number of $19 \cdot 3 = 57$ group choice scenarios.

Queries. We evaluated the following UNCQs:

$q(X) = \exists F, C, T\,(q_1(X,F) \vee q_2(X,C) \vee q_3(X,T))$,
$q'(X) = \exists F, C, T\,(q_4(X,F) \vee q_5(X,C) \vee q_3(X,T))$, and
$q''(X) = \exists F, C, T\,(q_1(X,F) \vee q_2(X,C) \vee q_6(X,T))$, where
$q_1(P,F) = food(F) \wedge \neg servesFood(P,F) \wedge place(P)$,
$q_2(P,C) = servesCuisine(P,C) \wedge cuisine(C) \wedge place(P)$,
$q_3(P,T) = placeType(T) \wedge isPlaceType(P,T) \wedge place(P)$,
$q_4(P,F) = food(F) \wedge servesFood(P,F) \wedge place(P)$,
$q_5(P,C) = \neg servesCuisine(P,C) \wedge cuisine(C) \wedge place(P)$,
$q_6(P,T) = placeType(T) \wedge \neg isPlaceType(P,T) \wedge place(P)$.

These queries represent situations where groups wish to decide where to go for a meal. For instance, taking into account the score-based semantics, q requests places where preferred cuisines are served, or the place type is preferred, or the place does not serve a preferred food. All runs have $10 \leq k \leq 100$, varied in steps of 10.

6.3. Results

Performance Evaluation. As performance metric, we use the time required to answer queries, varying k. Fig. 10 (top left) reports the results: our algorithms are very efficient; even the slowest ones return a top-k answer in less than 40 ms for high values of k, confirming the computational feasibility of our approach. Furthermore, FirstAvg is faster

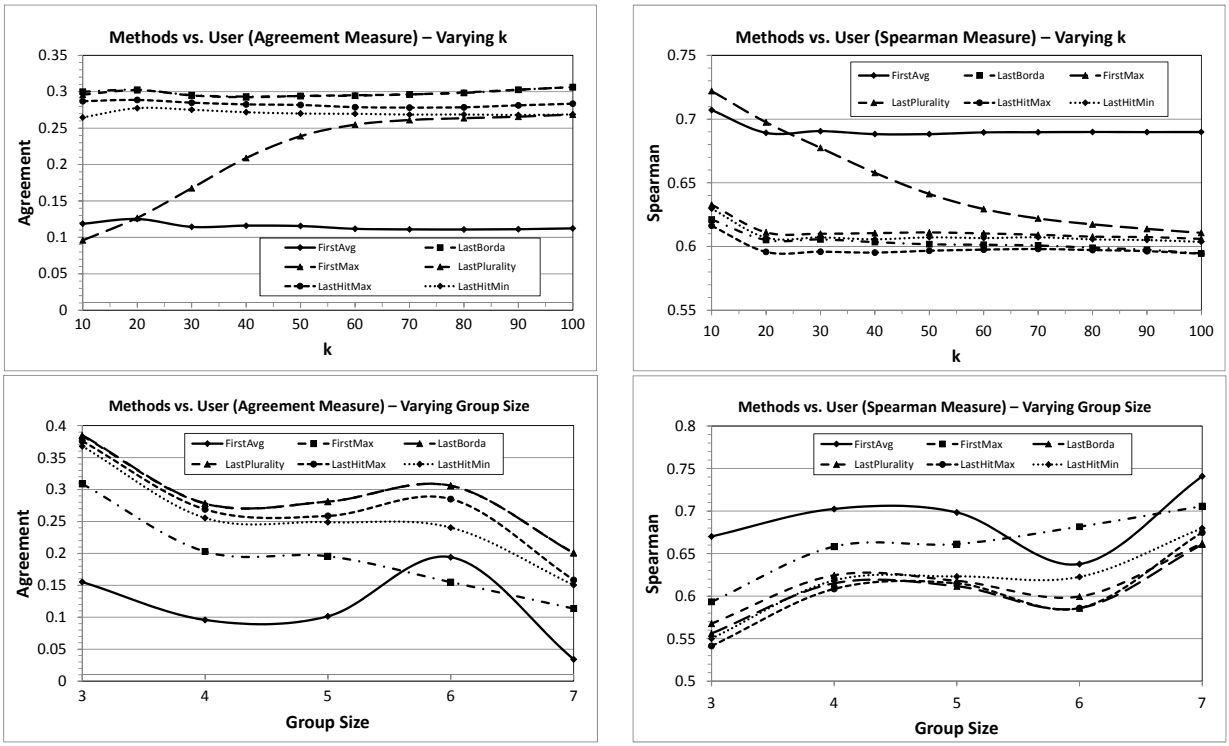

Figure 11: *Top left:* Quality evaluation Agreement varying k; *Top right:* Quality evaluation Spearman varying k; *Bottom left:* Quality evaluation Agreement varying group size; *Bottom right:* Quality evaluation Spearman varying group size. Note that the ranges of values in the Y-axes are not equal—this was done to improve the readability of the graphs.

than the other methods for this dataset, especially at higher values of k. Note that the materialization is not considered here (it was done off-line); the performance of the materialization in different benchmarks has already been shown to be practical in terms of the time, size, and memory consumption [22].

Quality Evaluation. We now discuss the results of the quality evaluation.

Evaluation Metrics. We use quality measurements that are often applied in evaluating information retrieval and group recommender systems [23, 24], namely, *tuple agreement*, *Kendall tau distance*, and *Spearman's footrule*.

Given two top-k lists, *tuple agreement* is defined as the number of tuples that appear in both lists (ordering is not considered). To allow comparisons across different values of k, we normalize the tuple agreement and call this value *Agreement—higher is better*, a value of 1 indicating lists with the same elements. The second measure is a variation of the *Kendall tau distance* for partial orders that computes the distance between two partial rankings based on the number of pairwise disagreements between them [23]—a disagreement receives a penalty of 1. In case the two top-k lists are permutations of each other, this is the number of exchanges required to convert one into the other. for cases where a pair (i, j) appears in one list but not in the other, it takes a parameter p indicating how pessimistic the metric should be ($p = 1$ is most pessimistic). The measure is normalized by the square of the length of the union of the two lists, and this value is called $Kendall_p$*—lower is better for Kendall*. Finally, the third measure is a variation of *Spearman's footrule* for partial orders that computes the distance between two partial rankings using the difference of positions of elements of one list compared with the other [23]. Whereas *Kendall* counts number of swaps, this metric counts how far each element must be moved to reach the place occupied in the other list. This measure is normalized as before, and the result is called *Spearman—lower is better for Spearman*.

Question 1: Which aggregation method yields the best results? To answer this question, for each group, we computed a top-k query answer r_g, along with a top-k query answer r_i for each user in the group. We then computed the value of each measure over r_g and r_i, and finally report the result of aggregating all such results. The choice of aggregation function is dependent on what is considered to be *best*: the value of the least satisfied user's score (min

for Agreement, and max for the others), the most satisfied user's score (max for Agreement, and min for the others), the median satisfaction, or the mean satisfaction. We computed these four functions for the Agreement measure.

We carried out three different comparisons: overall, varying the value of k, and varying group size—for the first, we report six different measures, while for the other two, we focus on *Agreement* and *Spearman*.

Overall Comparison. Fig. 10 (top right) shows that on average AggFirst (FirstAvg and FirstMax) is better than AggLast (LastBorda, LastPlurality, LastHitMax, and LastHitMin). Since there are cases where some users did not have any of their items in the top-k of the group for any of the methods, the least satisfied users have $Agreement = 0$ for all methods (this is why least satisfied is not plotted). In case of LastBorda and LastPlurality, some users have all their items in the top-k in the group, while for other methods, this does not hold. For reasons of space, we only show the results for the average. Fig. 10 (bottom left) shows that LastHitMax is the best at keeping the ordering of the items in users' top-k and the items in the group's top-k. From this point of view, FirstAvg performs the worst.

Varying Value of k. Fig. 11 (top) shows the results when varying k between 10 and 100—as before. We have results for the *Agreement* measure on the left and for *Spearman* on the right. We see that increasing the value of k affords an increase of performance for Plurality voting over both measures. For the rest of methods, we have not noticed any observation regarding the variation of k for either Agreement and Spearman.

Varying Group Size. Fig. 11 (bottom) shows the results when varying group size—we eliminated group sizes for which we did not have enough data. The results show that differently from the results in [6], where the larger the group, the more difficult to satisfy the user when k is fixed (since it is more likely that conflicting preferences exist), we cannot draw any such conclusions regarding our methods here.

Question 2: How different are the results produced by each aggregation method? Fig. 10 (bottom right) plots how different approaches compare when taken pairwise; "AggFirst" and "AggLast" are aggregations of methods computed as means over pairs (a_1, a_2), where a_1 is an aggregation first method, and a_2 is an aggregation last. We can see that AggFirst and AggLast are more similar when considering the right ordering than they are with respect to the Agreement measure. LastPlurality and LastBorda seem to include more elements in common and similar ordering in comparison with the AggFirst–AggLast pair and FirstMax–LastHitMax pair. Since the choice of the CQ and of the relation in the $chooseRel$ subroutine is done randomly in our implementation, this explains the fact that we do not observe high Agreement between methods and users: the top-k converges, but the list of elements that are higher or equal to the minimum score of top-k can be larger than k.

Finally, we ran two-tailed two-sample Student's t-tests comparing all pairs of experiments reported in this section. All tests yielded p-values well below 0.001, which shows statistical significance, except for three cases: (i) "LastPlurality vs. User"/ "Borda vs. User" for Agreement ($p = 0.84$) and Spearman ($p = 0.01$), (ii) "Borda vs. User" / "LastHitMin vs. User" for Spearman ($p = 0.18$) and Kendall-1 ($p = 0.28$), Kendall-0.5 ($p = 0.56$), and (iii) "LastHitMin vs. User" / "LastHitMax vs. User" for Agreement ($p = 0.05$), Spearman, Kendall-0.5 and Kendall-1 ($p = 0.01$ in all cases)—note that these higher p-values are expected, given the similarity of the results in those cases.

7. Related Work

Ranking queries have many applications in information systems: scores assigned to data can have many meanings, such as desirability to users, similarity to objects of interest, or quantifications of uncertainty associated with pieces of information—in all these cases, obtaining the top-k elements that satisfy a certain condition is a basic operation, and carrying it out efficiently is thus of central interest. In databases, this was recognized over a decade ago with the incorporation of tuple ranking into the RDBMS at the same basic level as Boolean filtering is implemented [8, 25]. This line of work builds on prior research on the basic problem of computing ranked answers to queries in relational databases [26, 27, 28]; for a survey of these topics see [29] and the more recent works of [30, 31].

As our main interest in the problem arises from Semantic Web applications, we discuss approaches in the literature to adding preferences to ontology languages, of which there have only been a few. To our knowledge, the first such proposal was [32], where an extension to SPARQL is developed, so that users can add their preferences to queries via a `PREFERRING` solution sequence modifier that supports both skyline semantics as well as soft constraints—in the latter, preference is given to answers that satisfy the constraints, but they can also be relaxed, if necessary. There have also been other approaches for preference-based querying in RDF graphs, such as [33, 34, 35]. Perhaps closest to our work is that of [36], which explores top-k query answering over relational databases that are accessed via *DLR-Lite*

ontologies (i.e., ontology-mediated access). The main differences with our approach lie in the query language (they consider only CQs, without negation) and the ontology language (guarded Datalog+/– subsumes the entire *DL-Lite* family of description logics). Moreover, their approach is based on query rewriting rather than database materialization. Finally, as mentioned in the introduction, PrefDatalog+/– [5] is also closely related; however, the general strict partial orders assumed in that work for representing preferences cause CQ answering to be Σ_2^p-complete—in this work, restricting preferences to arise from scores allows us to develop optimizations that result in tractable algorithms.

Another line of work that is relevant to ours is that of modeling *group decisions*; specifically, the development of both theoretical and practical approaches to solve the problem of choosing a set of elements in such a way that the preferences of a group of individuals are addressed as closely as possible. There are many fields that address this topic, such as mathematics, economics, and sociology [37, 38]. In social choice theory [39], the goal is to combine preferences to produce a new preference relation; methods range from those using score-based relations (e.g., approval voting) to others using more general ones (e.g., ranked pairs)—one example is the work of [40], which studies possibility/impossibility results generalizing properties such as Arrow's theorem to the case in which incomparable elements exist. Another highly relevant area is that of recommendation systems for groups of users [41, 4], though these tools are less general in the sense that queries are not explicitly issued.

The problem of aggregating preferences for a group of users is similar to that of *rank aggregation* (or *rank fusion*) [42], for which there is a quite extensive literature given that several problems in Web applications can be reduced to this problem. In particular, much work has been done on meta-search, i.e., the combination of result lists returned by multiple search engines in response to a given query; in general, these individual results are sorted lists of elements (URLs) accompanied by a relevance score. In [16], several methods for rank aggregation are studied and experimentally compared for the meta-search problem, while [43] studies an alternative approach to rank aggregation based on decision rules identifying positive and negative reasons for judging whether an element should get a better rank than another (based on two basic principles: majority and respect of minorities). In a related approach from recommender systems, [44] introduces a "blend" operator that combines several recommendations consisting of scores assigned to tuples—this is accomplished via a method that is part of the input. Independently of the method used, generally, the first step is to normalize the scores assigned to the items over all the rankings, for which there are also several proposals in the literature, such as [16, 45]. Recent approaches to rank aggregation include [46, 47].

8. Conclusion

We have proposed an approach to top-k query answering under user preferences in Datalog+/– ontologies, where the queries are unions of conjunctive queries with safe negation, and the preferences are defined via numerical scores. To this end, we have generalized the previous RankJoin operator to our framework. Furthermore, we have explored the generalization of the above approach to the preferences of a group of users. Finally, we have provided experimental results on the performance and quality of our algorithms.

One topic of ongoing and future work is to further experimentally evaluate the similarity of preference aggregations to human judgment in order to select the best suited ones for search and query answering in the Social Semantic Web. Another interesting topic for future research is to generalize this approach to ontologies with uncertainty.

Acknowledgments

This work was supported by The Alan Turing Institute under the UK EPSRC grant EP/N510129/1, by an EPSRC Doctoral Prize under the EPSRC grant EP/N509711/1, by the EPSRC grants EP/J008346/1, EP/L012138/1, and EP/M025268/1, by the ERC (FP7/2007–2013) grant 246858 "DIADEM", by a Google European Doctoral Fellowship, by a Yahoo! Research Fellowship, by the US Department of the Navy, Office of Naval Research, grant N00014-15-1-2742, by the EU H2020 Research and Innovation Programme under the Marie Sklodowska-Curie grant agreement No. 690974 for the project "MIREL", and by funds provided by Universidad Nacional del Sur, Agencia Nacional de Promoción Científica y Tecnológica, and CONICET, Argentina.

References

[1] F. Rossi, K. B. Venable, T. Walsh, A Short Introduction to Preferences: Between Artificial Intelligence and Social Choice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2011.

[2] N. Li, W. Cushing, S. Kambhampati, S. Yoon, Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences, ACM Trans. Intell. Syst. Technol. 5 (2) (2014) 29:1–29:32.
[3] K. Cheng, I. Zuckerman, D. S. Nau, J. Golbeck, Predicting agents' behavior by measuring their social preferences, in: Proc. ECAI, 985–986, 2014.
[4] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, C. Yu, Group recommendation: Semantics and efficiency, Proc. VLDB Endow. 2 (1) (2009) 754–765.
[5] T. Lukasiewicz, M. V. Martinez, G. I. Simari, Preference-based query answering in Datalog+/– ontologies, in: Proc. IJCAI, 1017–1023, 2013.
[6] T. Lukasiewicz, M. V. Martinez, G. I. Simari, O. Tifrea-Marciuska, Ontology-based query answering with group preferences, ACM T. Internet Techn. 14 (4) (2014) 25:1–25:24.
[7] A. Calì, G. Gottlob, T. Lukasiewicz, A general Datalog-based framework for tractable query answering over ontologies, J. Web Sem. 14 (2012) 57–83.
[8] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, Supporting top-k join queries in relational databases, The VLDB Journal 13 (3) (2004) 207–221.
[9] G. Gottlob, G. Orsi, A. Pieris, Ontological queries: Rewriting and optimization, in: Proc. ICDE, 2–13, 2011.
[10] G. Gottlob, G. Orsi, A. Pieris, Query rewriting and optimization for ontological databases, ACM Trans. Database Syst. 39 (3) (2014) 25.
[11] C. Beeri, M. Y. Vardi, The implication problem for data dependencies, in: Proc. ICALP, vol. 115 of *LNCS*, 73–85, 1981.
[12] A. Calì, G. Gottlob, M. Kifer, Taming the infinite chase: Query answering under expressive relational constraints, in: Proc. KR, 70–80, 2008.
[13] N. B. Amor, D. Dubois, H. Gouider, H. Prade, Possibilistic preference networks, Information Sciences (2017) In press.
[14] T. Joachims, Optimizing search engines using clickthrough data, in: Proceedings KDD, 133–142, 2002.
[15] L. Xia, Improving group decision-making by artificial intelligence, in: Proc. IJCAI, 5156–5160, 2017.
[16] M. E. Renda, U. Straccia, Web metasearch: Rank vs. score based rank aggregation methods, in: Proc. SAC, 841–846, 2003.
[17] X. Zhang, J. Chomicki, On the semantics and evaluation of top-k queries in probabilistic databases, in: Proc. ICDE Workshops, 556–563, 2008.
[18] G. Cormode, F. Li, K. Yi, Semantics of ranking queries for probabilistic data and expected ranks, in: Proc. ICDE, 305–316, 2009.
[19] W. Gaertner, A Primer in Social Choice Theory, Oxford University Press, 2009.
[20] X. Zhang, J. Chomicki, Semantics and evaluation of top-k queries in probabilistic databases, Distrib. Parallel Dat. 26 (2009) 67–126.
[21] Yelp, Yelp Dataset Challenge, URL `http://www.yelp.co.uk/dataset_challenge/`, 2012.
[22] B. Fazzinga, G. Gianforme, G. Gottlob, T. Lukasiewicz, Semantic Web search based on ontological conjunctive queries, J. Web Sem. 9 (4) (2011) 453–473.
[23] R. Fagin, R. Kumar, D. Sivakumar, Comparing top k lists, SIAM J. Discrete Math. 17 (1) (2003) 134–160.
[24] E. Ntoutsi, K. Stefanidis, K. Nørvåg, H.-P. Kriegel, Fast group recommendations by applying user clustering, in: Proc. ER, vol. 7532 of *LNCS*, 126–140, 2012.
[25] C. Li, K. C.-C. Chang, I. F. Ilyas, S. Song, RankSQL: Query algebra and optimization for relational top-k queries, in: Proc. SIGMOD, 131–142, 2005.
[26] R. Fagin, Combining fuzzy information from multiple systems, J. Comput. Syst. Sci. 58 (1) (1999) 83–99.
[27] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, J. S. Vitter, Supporting incremental join queries on ranked inputs, in: Proc. VLDB, vol. 1, 281–290, 2001.
[28] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66 (4) (2003) 614–656.
[29] I. F. Ilyas, G. Beskales, M. A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (4) (2008) 11.
[30] K. Stefanidis, G. Koutrika, E. Pitoura, A survey on representation, composition and application of preferences in database systems, ACM Trans. Database Syst. 36 (3) (2011) 19:1–19:45.
[31] A. Arvanitis, G. Koutrika, Towards preference-aware relational databases, in: Proc. ICDE, 426–437, 2012.
[32] W. Siberski, J. Z. Pan, U. Thaden, Querying the Semantic Web with preferences, in: Proc. ISWC, vol. 4273 of *LNCS*, 612–624, 2006.
[33] G. Chamiel, M. Pagnucco, Exploiting ontological information for reasoning with preferences, in: Multidisciplinary Workshop on Advances in Preference Handling, 2008.
[34] L. Chen, S. Gao, K. Anyanwu, Efficiently evaluating skyline queries on RDF databases, The Semanic Web: Research and Applications (2011) 123–138.
[35] S. Magliacane, A. Bozzon, E. Della Valle, Efficient execution of top-k SPARQL queries, in: Proc. ISWC, vol. 7649 of *LNCS*, 344–360, 2012.
[36] U. Straccia, Top-k retrieval for ontology mediated access to relational databases, Inf. Sci. 198 (2012) 1–23.
[37] P. K. Pattanaik, Voting and Collective Choice: Some Aspects of the Theory of Group Decision-Making, Cambridge University Press, 1971.
[38] A. D. Taylor, Social Choice and the Mathematics of Manipulation, Cambridge University Press, 2005.
[39] W. Gaertner, A Primer in Social Choice Theory: Revised edition, Oxford University Press, USA, 2009.
[40] M. S. Pini, F. Rossi, K. B. Venable, T. Walsh, Aggregating partially ordered preferences, J. Logic Comput. 19 (3) (2009) 475–502.
[41] G. Linden, B. Smith, J. York, Industry report: Amazon.com recommendations: Item-to-item collaborative filtering, IEEE Internet Comput. 7 (1) (2003) 76–80.
[42] C. Dwork, R. Kumar, M. Naor, D. Sivakumar, Rank aggregation methods for the web, in: Proc. WWW, 613–622, 2001.
[43] M. Farah, D. Vanderpooten, An outranking approach for rank aggregation in information retrieval, in: Proc. SIGIR, 591–598, 2007.
[44] G. Koutrika, B. Bercovitz, H. Garcia-Molina, FlexRecs: Expressing and combining flexible recommendations, in: Proc. SIGMOD, 745–758, 2009.
[45] M. Fernández, D. Vallet, P. Castells, Probabilistic score normalization for rank aggregation, in: Proc. ECIR, 553–556, 2006.
[46] I. Caragiannis, X. Chatzigeorgiou, G. A. Krimpas, A. A. Voudouris, Optimizing positional scoring rules for rank aggregation, in: Proc. AAAI, 430–436, 2017.
[47] I. Caragiannis, E. Micha, Learning a ground truth ranking using noisy approval votes, in: Proc. IJCAI, 149–155, 2017.