Comparative Preferences in SPARQL

Peter F. Patel-Schneider[1], Axel Polleres[2,3], and David Martin[1]

[1] NAIL Laboratory, Nuance Communications, Sunnyvale, CA, USA
[2] Vienna Univ. of Economics & Business / Complexity Science Hub Vienna, Austria
[3] Stanford University, CA, USA*

Abstract. Sometimes one does not want all the solutions to a query but instead only those that are most desirable according to user-specified preferences. If a user-specified preference relation is acyclic then its specification and meaning are straightforward. In many settings, however, it is valuable to support preference relations that are not acyclic and that might not even be transitive, in which case though their handling involves some open questions. We discuss a definition of desired solutions for arbitrary preference relations and show its desirable properties. We modify a previous extension to SPARQL for simple preferences to correctly handle any preference relation and provide translations of this extension back into SPARQL that can compute the desired solutions for all preference relations that are acyclic or transitive. We also propose an additional extension that returns solutions at multiple levels of desirability, which adds additional expressiveness over prior work. However, for the latter we conjecture that an effective translation to a single (non-recursive) SPARQL query is not possible.

1 Introduction

Preferences and the notion of the Semantic Web are tightly interwoven: the seminal vision article often cited as coining the term "Semantic Web" already mentions preferences in several places [1], for instance: "*Pete [...] set his own agent to [...] search with [...] preferences about location and time.*". The same article also already mentions standardization in terms of languages defining such preferences, such as the Composite Capability/Preference Profile (CC/PP) [2], which allows a user agent (typically a client application) to declare its preferences, e.g., in terms of device capabilities.[4]

Interestingly this early interest in expressing preferences and retrieving Semantic Web data compliant with these preferences has not found its way into later Semantic Web Standards, such as SPARQL. There is no built-in way to express and evaluate queries with preferences in SPARQL, but there have been several proposals [3–5] for adding preferences to SPARQL and defining a meaning for such preferential queries.

We argue that preferences in their full generality are not correctly handled in the proposals so far, and show how this can be addressed with a modified semantics. Additionally, we show that certain kinds of preferences can be expressed in a single SPARQL 1.1 query, although the most recent proposal that translates preferences into standard

* Axel Polleres' work was supported under the Distinguished Visiting Austrian Chair Professors program hosted by The Europe Center of Stanford University.

[4] CC/PP seems to be discontinued, according to https://www.w3.org/TR/CCPP-struct-vocab2/.

SPARQL queries fails to work on relatively simple examples due to problems in the SPARQL 1.1 query standard [6]. For general preferences we argue that a translation into a single SPARQL 1.1 query is not obvious, but provide a translation with both `CONSTRUCT` and `SELECT` queries.

Example 1 (Running Example). As a running example, let us assume data in a navigation scenario, where a user would be looking for gas stations with preferences among features such as $brand$, $distance$ to the user's location, and different $shops$ that sell various $antifreeze$ products, given as an RDF Graph G_1:

```
:p123 a :GasStation; :brand :Mobil;  :dist 1.1; :shop :TigerMart; :af :Prestone .
:p456 a :GasStation; :brand :Chevron; :dist 0.5; :shop :KwikieMart; :af :StarBrite .
:p789 a :GasStation; :brand :Shell; :dist 0.8; :shop :711; :af :Zerex .
:p012 a :GasStation; :brand :Citgo; :dist 6 .
```

User preferences could be of different forms, such as:

- **P1** "I prefer gas stations within 1 mile distance" (simple boolean)
- **P2** "I prefer the closest gas station" (quantitative)
- **P3** "I prefer Mobil over Chevron" (comparative)
- **P4** "I prefer solutions within 1 mile (**P1**) and among those I prefer Mobil over Chevron gas (**P3**), and Kwikie Mart over 7-11, and otherwise just the closest (**P2**)" (combinations)

So atomic preferences can be *simple boolean*, i.e. stating preferences for solutions fulfilling a certain boolean condition, *quantitative*, i.e., where each solution is given a score from a totally ordered set (generally the integers or rationals), or *comparative*, i.e., preferences expressed as a binary relation *between solutions*. Such atomic preferences can be combined in various ways.

A preference query takes the results of a non-preferential subquery (in this case, a subquery that returns gas stations), and selects most preferred ones. (The precise definition of "most preferred" is part of what we are examining in this paper.) The preferences we wish to handle call for a generalization of the skyline operator [7] in databases so we will talk about obtaining the skyline of a preference relation, i.e., the most preferred results based on a preference.[5] Subsidiary preference results can be defined, such as the nth skyline, i.e., the skyline after the first $n-1$ skylines have been removed, or even returning solutions along with their skyline number (which gives a rank for each solution).

We focus on comparative preferences, partly because (i) comparative preferences are more general than simple boolean or quantitative preferences, (ii) comparative preferences can capture combined preferences as part of their preference relation representation, (iii) user preferences are often comparative [8–10].

As for (i), we note that quantitative preferences generalize simple boolean preferences (by mapping true to 1 and false to 0), and comparative preferences generalize quantitative preferences, by preferring solutions with a better score to solutions with a worse score.

[5] In databases, skyline involves a multiway combination of totally ordered comparisons between the values in tuples; qualitative preferences here instead allow an arbitrary comparison relation.

2 Foundations and Motivation

Before listing the contributions of this work, we introduce some basic (and widely used) formal foundations and illustrate a few difficulties that follow from them. We adapt our formal definition of comparative preferences from that of Chomicki [11] as also used by other work in the area such as Troumpoukis et al. [5].

Definition 1. *Given a set of potential solutions P, a preference relation $\succ$ is any relation over $P \times P$. A solution s_1 is* dominated by *a solution s_2 if $s_2 \succ s_1$.*

Although a preference relation is defined over a universe of *potential* solutions, it is applied to finite sets of *candidate* solutions. Typically, potential solutions are all solutions that are representable under the schema of some information source. Candidate solutions are then solutions returned from the ordinary (non-preferential) part of a query. poster

Example 2. Referring back to Example 1, the four gas stations shown could be the *candidate* solutions returned by the ordinary SPARQL query, Q_1:

```
Q1: SELECT * {?X a :GasStation; :brand ?B; :dist ?D. FILTER(?D<=10)}
```

The *potential* solutions would be a much larger set (i.e., all gas stations that could be represented in an information source). The preference relation for **P3** would include all pairs (m, c) where m is a Mobil station and c a Chevron station. After applying this preference relation, for reasons discussed below, one would be left with the Mobil, Shell, and Citgo stations.

In this paper, we ground candidate solutions in results of SPARQL 1.1 queries [6] (that is, multisets of variable bindings). We also allow for solutions from SPARQL extended with external services.[6] The examples in this paper will not use such service extensions; for simplicity we will just use preferences over solutions of SPARQL queries over a static RDF graph, such as G_1 in Example 1.

Example 3. For example, a service could be used to return gas stations within a 10-mile distance *from the user's current location*. Instead of the simple SPARQL query above, Q_1 (to be run over graphs like G_1), one could use a query over a graph that only contains the geo-locations of the gas stations and accesses a service to get the current user location, such as the following:[7]

```
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/geosparql/function>
SELECT * { ?X a :GasStation; :brand ?B; geo:hasGeometry ?point.
           SERVICE <userlocation> { SELECT * { [ geo:hasGeometry ?point1] } }
           FILTER( geof:distance(?point,?point1,uom:mile) <=10) }
```

[6] SPARQL 1.1 provides a basic mechanism for such external services (e.g., to look up or compute current prices or exchange rates), using the SERVICE keyword [12].

[7] This fictitious query uses the GeoSPARQL extension to SPARQL, cf. http://www.opengeospatial.org/standards/geosparql.

We do not require any other properties of preference relations. In particular, a preference relation here need not be irreflexive, asymmetric, acyclic, or transitive.[8] It may seem that these should be required aspects of a preference relation but we want to study what happens with arbitrary preference relations, such as those likely to be obtained directly from users.

Example 4. We allow a combination of comparative preferences on different aspects of gas stations. For example **P6a**: a preference for Mobil brand gas stations over Chevron; **P6b**: a preference for gas stations with KwickieMart stores over those with 7-11's; and **P6c**: a preference for gas stations selling Zerex antifreeze over those selling Prestone. (What is important here is that the preferences on the aspects are only *partial* orders, not total orders.)

Example 5. We also allow the obviously cyclic preference relation consisting only of **P3a**: a preference for Mobil brand gas stations over Chevron brand; **P3b**: a preference for Chevron over Citgo; and **P3c**: a preference for Citgo over Mobil.

The implementation of our preference relations in SPARQL will be arbitrary SPARQL expressions. If solutions contain objects from some sort of knowledge repository the relation can depend on anything accessible in the repository. (Troumpoukis et al. [5] call preferences that do not use external information intrinsic preferences.) In SPARQL this means that an expression defining a preference relation can access properties of a solution object from the underlying graph (such as the brand of fuel sold at a gas station, its current distance, or whether it has a roof over its pumps), and other information in the graph (such as whether it is currently raining, etc.), without having this information in the solution itself.

Our preference relations are not examinable in general, and the set of *potential* solutions will generally be very large, or even infinite.

Example 6. For instance, the preference **P5**: "I prefer between two solutions the one closer in distance", could apply to infinitely many *potential* solutions (not knowing the underlying graph G). Note also that there could be other gasoline brands in G that are not named explicitly in the preferences at hand.

It can thus be practically infeasible to compute the transitive closure of a preference relation, or indeed determine whether a preference relation is irreflexive, asymmetric, acyclic, or transitive.

Example 7. For instance, **P5** is transitive on all sets of candidate solutions. However, if we view **P3a-c** as a single preference relation, transitivity does not hold. As well, additional preferences such as **P3d** "I prefer Shell over Mobil" could make the preference relation lose completeness[9] (every element comparable with every other). See Figure 1 for a graphical illustration. Similarly, **P6a-c** produces a preference cycle on our example gas stations.

[8] To review the basic properties of binary preference relations see Chomicki [11].

[9] Chomicky calls this *connectivity* [11, Def.2.1].

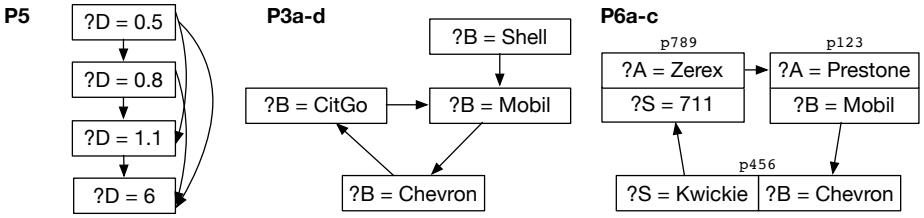

Fig. 1. Preference relations **P5** and **P3a-d** for the solutions of query Q_1 over graph G_1.

The basic operation in comparative preferences is the winnow operator [11]. The intuitive notion is that given an candidate set of solutions and a preference relation, the winnow operator returns those solutions that have no solution dominating them. Based on its similarity to skylines in databases, we call the result of the winnow operator the skyline of a preference relation. Formally this is (again adapted from Chomicki [11]):

Definition 2. *If S is a finite set of (candidate) solutions and $\succ$ a preference relation over some (potentially infinite) superset of S then the* skyline *of S with respect to $\succ$ is*

$$\omega_{\succ}(S) = \{s \in S | \neg \exists s' \in S . s' \succ s\}$$

We loosely refer to the definition of skyline as the "semantics" of preference relations. Note here that $\succ$ has access to extrinsic information about objects in solutions in S. In the SPARQL setting this means that $\succ$ has access to the graph being queried.

We can also define the second skyline as the skyline after the initial skyline has been removed from the solution set, and so on, thus defining "levels" of skyline solutions:

Definition 3. *If S is a finite set of (candidate) solutions and $\succ$ a preference relation over some superset of S then the nth* skyline *of S with $\succ$ is defined as*

$$\omega^n_{\succ}(S) = \{s \in S \setminus \bigcup_{i=1}^{n-1} \omega^i_{\succ}(S) \;\mid\; \neg \exists s' \in S \setminus \bigcup_{i=1}^{n-1} \omega^i_{\succ}(S) \;.\; s' \succ s\}$$

The rank *of a solution, s, in S with respect to $\succ$ is the number of the skyline that it is in, i.e.,* $\text{rank}^S_{\succ}(s) = n$ *for $s \in \omega^n_{\succ}(S)$. If a solution is not in any skyline, then its rank is undefined.*

Returning multiple skylines is valuable if all elements of the top skyline might be deemed unsuitable by later processing, or if a minimum number of solutions is required, which could exceed the size of the top skyline.

Example 8. Looking at Figure 1 let us assume for the moment a slight re-formulation of **P5** to be "I prefer between two solutions the one closer or equal in distance". This obviously is not very intuitive as it makes the preference relation reflexive, and thereby makes *each solution dominate itself* – resulting in an empty skyline. However, as we will see current proposals for encoding preferences in SPARQL allow such preferences.

Example 9. Cyclic preferences can appear in practice by collecting single user preferences, expressed on separate occasions, and with the definitions above can lead to unexpected results. We take here the example of **P3a-d** from Figure 1. Intuitively, among *all four* candidate solutions the solution with brand `Shell` is most preferred, but then under Definition 3 the second skyline is *empty*, since *each remaining solution is dominated by another candidate solution*. Our intuition is that in such a case all of these three solutions should be equally preferred, and in the second skyline. Note that **P6a-c** also produce a cycle in the 4 candidate solutions, even though their expression does not look obviously cyclic.

Assuming, on the other hand, that `Citgo` was *not* within the candidate solutions, say by changing Q_1 to Q_2 as follows:

```
Q2: SELECT * {?X a :GasStation; :brand ?B; :dist ?D. FILTER(?D<=5)}
```

That would reduce the number of candidate solutions to just `Shell`, `Mobil`, and `Chevron`, which would be in the first, second and third skyline, respectively (this time in accord with our intuitions). So, we see that removing (and likewise adding) candidate solutions can change the rank and ordering of solutions.

To complicate things further, if we consider **P3a-d** or **P6a-c** in combination with **P1** the handling of the combined preference relation become even less clear, depending on the semantics of combinations and their respective consideration of candidate solutions.

Given examples such as these, we aim to shed more light on the semantics for preferences in queries, and provide definitions that handle these situations more intuitively and satisfy basic desiderata, such as that each level of skyline should contain at least one solution, even in cases where preferences are not necessarily coherent.

Our primary contributions in the present work are

- an analysis of what goes wrong with a widely used definition of the skyline operator when a preference relation fails to be acyclic and/or transitive,
- a definition that works for arbitrary preference relations,
- a slightly simpler definition that can be used with transitive preference relations, with performance benefits,
- translations to SPARQL 1.1 (of a SPARQL extension proposed in prior work) to implement these new definitions, and
- a further extension that allows a query to request *multiple levels* of skylines.

In the next section, we will review some earlier approaches to the formulation of preferences in SPARQL, in the light of whether they can express comparative semantics at all and how they would handle the cases above. We will then approach the analysis of preference relations and variations of the definition of the skyline operator more formally in Section 4 and Section 5, by discussing what we call simple (i.e., acyclic and/or transitive) preferences and non-simple preferences separately. Along the way we will refine the notion of skyline operators and provide two more variations thereof. Finally, in Sections 6 and 7 we will discuss which of these proposed variations can be implemented in SPARQL 1.1, and which require extensions in terms of bespoke evaluation algorithms. In this topic, we will identify problems in existing approaches to translating preference handling to SPARQL, and propose repairs, where possible.

3 Previous Work

The notion of preference has a central role in many disciplines, including economics, psychology and other social sciences, some areas of philosophy, decision theory and game theory (themselves interdisciplinary topics), and computer science. The formalization of comparative preferences as binary preference relations runs throughout much of this work, although the definitional details vary. Much work on preference relations assumes, mandates, or arranges for them to be acyclic and/or transitive (or constrained in other ways), for a variety of reasons. Nonetheless, there is a large literature showing that cyclic and/or intransitive preference relations do arise naturally in the real world of human judgments [8–10].

Turning to preferences used with query languages, Chomicki [11] analyzes intrinsic comparative preferences in a relational database setting. He considers preferences as multidimensional combinations of atomic preferences where each atomic preference is a built-in SQL function (such as numeric ordering). A preference function could be to consider distance and fuel brand separately, which ends up with the closest gas station for each brand, or to first consider distance and then some ordering of brands, which ends up with the closest gas station but if there is a tie for closest then chooses by brand. He analyzes the properties of his winnow operator with an eye to how it can be optimized in SQL queries.

Siberski et al. [3] transform a subset of these preferences into an early version of SPARQL, producing a SPARQL query extension for conjunctive (combine preferences over two different values) cascaded (consider preference over one value before preference over another) intrinsic preferences. They implement these preferences as an extension to ARQ[10], with a syntax inspired by an early version of the Preference SQL language [13].

Gueroussova et al. [4, 14] transform a version of Chomicki's preferences (which they call conditional preferences) to SPARQL, extending the work of Siberski et al. by adding more combination operators. They define an extension to SPARQL, which they call PrefSPARQL (after the later version of Preference SQL [15] which incorporates these features) and provide a mapping from PrefSPARQL to both SPARQL 1.0 queries [16] and SPARQL 1.1 queries.

Unfortunately their mapping to SPARQL 1.1 uses the SPARQL EXISTS operator. SPARQL EXISTS has many known problems [17], and their solution falls prey to one one them, namely that the SPARQL BOUND operator does not work correctly inside EXISTS. Their mapping to SPARQL 1.0 is not affected by this problem, as shown by the next example.

Both the approaches by Siberski et al. and Gueroussova et al. focus on combined preferences over boolean preferences, that is, while they allow the combination of different preferences, these preferences do not give the full flexibility of comparing arbitrary solutions pairwise, but only by declaring boolean "preferred conditions" C, that divide the solution space into preferred and non-preferred solutions according to the preference, according to the schema in Equation (1).

$$\texttt{SELECT } V \texttt{ WHERE } \{ \ P \ \textbf{PREFERRING} \ C \ \} \tag{1}$$

[10] https://jena.apache.org/documentation/query/

Example 10. In both the approaches above preferences like **P1** can be written by extending queries like Q_1. In PrefSPARQL this is:

```
Q3: SELECT * { ?X a :GasStation; :brand ?B; :dist ?D .
               PREFERRING (?D <= 1) }
```

Combined preferences can be expressed (so long as they do not involve comparisons between different attributes of solutions), such as this variant of **P4**:

> **P4'** "I prefer solutions within 1 mile and among those I prefer Mobil, and otherwise just the closest."

which in PrefSPARQL could be written as the following query:[11]

```
Q4: SELECT * { ?X a :GasStation; :brand ?B; :dist ?D .
               PREFERRING ((?D <= 1) AND ?B="Mobil") PRIOR TO LOWEST ?D) }
```

While referring for details to [14], we illustrate PrefSPARQL's translation back to "vanilla" SPARQL by the example of Q_3 which in SPARQL 1.1 yields $Q_3^{1.1}$:

```
Q3^1.1: 1  SELECT * { {?X a :GasStation; :brand ?B; :dist ?D .}
        2    FILTER NOT EXISTS { ?X' a :GasStation; :brand ?B'; :dist ?D' .
        3                        FILTER ((?D' <= 1) > (?D <= 1)) } }
```

That is, the translation relies in principle on creating a copy of the query pattern within a SPARQL 1.1 NOT EXISTS clause (line 2) and then encoding dominance of the solutions to this copy in an inner FILTER expression (line 3).

As shown in [14], this principle can be easily extended to combined preference relations (through encoding AND and PRIOR TO, and adding conditional IF-THEN-ELSE preferences from Preference SQL to more complex inner FILTER expressions). Specific quantitative comparative preference relations are also expressible with the keyword HIGHEST or LOWEST, but not general comparative preferences.

Example 11. The quantitative preference **P2** would be expressible in PrefSPARQL as query

```
Q4: SELECT ?X ?B ?D { ?X a :GasStation; :brand ?B; :dist ?D.
                      PREFERRING LOWEST ?D }
```

However, there exists as mentioned above a problem with this translation due to the semantics of NOT EXISTS, as illustrated with the following example:

Example 12. Let us assume **P7** saying "I prefer gas stations with a shop", which in PrefSPARQL could be expressed as

```
Q5: SELECT * { ?X a :GasStation. OPTIONAL { ?X :shop ?S. }
               PREFERRING ( BOUND(?S) ) }
```

in which case the translation from [14] no longer works, as shown in $Q_5^{1.1}$:

[11] Note that strictly speaking we would need to write here ((?D <= 1) AND ?B="Mobil") PRIOR TO (?D <= 1) PRIOR TO LOWEST ?D), but it is easy to see that by the last preference preferring just the closest the middle part is redundant.

```
Q_5^{1.1}: 1   SELECT * { { ?X a :GasStation. OPTIONAL { ?X :shop ?S. } }
       2     FILTER NOT EXISTS { ?X' a :GasStation. OPTIONAL { ?X' :shop ?S'. }
       3                         FILTER(BOUND(?S') > BOUND(?S)) } }
```

This fails because the substitution semantics of EXISTS here produces algebra expressions like BOUND (:TigerMart) that are undefined in SPARQL 1.1; we refer to details in [17]. We note though, that Gueroussova et al.'s [14] SPARQL 1.0-based way of encoding non-existence through a combination of `OPTIONAL` and `!BOUND()`, as illustrated in the following query $Q_5^{1.0}$, works as intended:

```
Q_5^{1.0}: 1   SELECT * { { ?X a :GasStation. OPTIONAL { ?X :shop ?S. } }
       2              OPTIONAL { ?X' a :GasStation. OPTIONAL { ?X' :shop ?S'. }
       3                         FILTER ( BOUND(?S') > BOUND(?S) ) }
       4              FILTER ( !BOUND(?X') ) } }
```

Troumpoukis et al. [5] expand on this work to allow arbitrary SPARQL expressions as the $\succ$ operator; that is, they define an extension to SPARQL 1.1 that can express full comparative preferences called SPREFQL, and also provide a mapping from SPREFQL into SPARQL 1.1 queries. They implemented SPREFQL and compared its performance to the performance of their mapping.

As opposed to PrefSPARQL, in SPREFQL one can express comparative preferences by explicitly referring to variables in the "copy" of the query pattern over which preferences are defined using a clause `PREFER-TO-IF` which creates two explicit copies V_1, V_2 of the variables V in the `SELECT` clause that can be referenced in a comparative condition C:

$$\texttt{SELECT } V \texttt{ WHERE } \{ \ P \ \} \ \textbf{PREFER } V_1 \ \textbf{TO } V_2 \ \textbf{IF } C \tag{2}$$

Example 13. The quantitative preference **P2** would be expressible in SPREFQL as a comparative preference in query Q_6:

```
Q_6: SELECT ?X ?B ?D { ?X a :GasStation; :brand ?B; :dist ?D. }
     PREFER ?X1 ?B1 ?D1 TO ?X2 ?B2 ?D2 IF (?D1 < ?D2)
```

More general comparative preferences such as **P3** can also be expressed in SPREFQL, as in query Q_7:

```
Q_7: SELECT ?X ?B ?D { ?X a :GasStation; :brand ?B; :dist ?D . }
     PREFER ?X1 ?B1 ?D1 TO ?X2 ?B2 ?D2
            IF (?B1 = :mobil && ?B2 = :chevron)
```

Combined preferences are also supported, through AND and PRIOR TO clauses. Unfortunately, however, with the syntactic expansion over PrefSPARQL, it is quite possible for the preference relation to be non-transitive or to have loops.

Example 14. For instance, imagine Q_6 modified by replacing $<$ with $\leq$ (which we will refer to as Q_6' below), so that each solution dominates itself. Another example involves expressing **P6a-c** as:

```
Q_8: SELECT ?X ?B ?S ?T {?X a :GasStation; :brand ?B; :shop ?S; :antifreeze ?A.}
     PREFER ?X1 ?B1 ?S1 ?A1 TO ?X2 ?B2 ?S2 ?A2
     IF ( (?B1 = :Mobil && ?B2 = :Chevron) ||
          (?S1 = :KwikieMart && ?S2 = :711) ||
          (?A1 = :Zerex && ?A2 = :Prestone) )
```

This is problematic, as Troumpoukis et al. only use the simple definition of winnowing above. If there is a preference loop in the candidate solutions, as for Q_8 on the example data, then none of the solutions in the loop will ever be returned, because each of them is dominated by another solution. The preference combination operators do not help: AND will just produce the same cyclic preference and PRIOR TO does not have a suitable meaning. This problem also occurs for reflexive loops, as for Q_6' where each solution dominates itself. We provide a simple but general solution for these problems below. As we will see, the absence of transitivity does not cause a problem by itself, but does complicate the problem of loops.

Also, note that Troumpoukis et al. provide a mapping to SPARQL 1.1 where the schema of Equation (2) is replaced by (again, we refer for details to [5]):

$$\texttt{SELECT } V \texttt{ WHERE } \{P \texttt{ FILTER NOT EXISTS } \{P_{(V/V_1)} \texttt{ FILTER } C_{(V_2/V)} \ \}\} \tag{3}$$

This is a straightforward extension of the PrefSPARQL translation, which however depends again on SPARQL EXISTS and also falls prey to the problem with BOUND, as in Example 12.

4 Simple Comparative Preferences

We first establish that acyclic preference relations are non-problematic: they are guaranteed to determine a nonempty skyline over any finite, non-empty set of solutions and ranks behave well. In addition, as we will discuss in detail in Section 6, skylines over acyclic preference relations are specifiable in SPARQL 1.1. As every transitive and irreflexive relation is acyclic, handling acyclic preference relations also handles this common requirement for preferences.

Theorem 1. *If the preference relation $\succ$ is acyclic over a finite, non-empty set of candidate solutions S, i.e., there is no candidate solution s for which there is a sequence of candidate solutions starting and ending with s such that each dominates the next, then $\omega_\succ(S)$ is non-empty.*

Proof. If $\succ$ is acyclic on finite *S then there are elements of S that are not dominated by any member of S, thus ensuring non-emptiness.* □

Because each skyline is non-empty for a finite, non-empty set of solutions if the preference relation is acyclic, each solution has a uniquely defined rank.

Corollary 1. *If the preference relation $\succ$ is acyclic over a finite, non-empty set of candidate solutions S then $\texttt{rank}_\succ^S(s)$ is defined for each $s \in S$.*

Proof. (By induction)

If $S = \{\}$ the conclusion follows trivially.

Assume the conclusion for all $S^i \subset S$. As $\omega_\succ(S)$ is non-empty, $S' = S \setminus \omega_\succ(S) \subset S$, so $\texttt{rank}_\succ^{S'}(s)$ is defined for $s \in S'$. Since $\texttt{rank}_\succ^S(s) = 1 + \texttt{rank}_\succ^{S'}s$ for $s \in S$, $\texttt{rank}_\succ^S(s)$ is defined for $s \in S'$. Since $\texttt{rank}_\succ^S(s) = 1$ for $s \in \omega_\succ(S)$, $\texttt{rank}_\succ^S(s)$ is defined for $s \in S$. □

Dominance between solutions is reflected in their relative ranks.

Theorem 2. *If the preference relation $\succ$ is acyclic over a finite, non-empty set of candidate solutions S then $s_1 \succ s_2$ implies $\texttt{rank}^S_\succ(s_1) < \texttt{rank}^S_\succ(s_2)$.*

Proof. Assume otherwise, i.e., there are $s_1, s_2 \in S$ with $s_1 \succ s_2$ and $\texttt{rank}^S_\succ(s_1) \geq \texttt{rank}^S_\succ(s_2)$.

Let $r = \texttt{rank}^S_\succ(s_2)$. Then $s_1 \in S \setminus \bigcup_{i=1}^{r-1} \omega^i_\succ(S)$ because otherwise $\texttt{rank}^S_\succ(s_1) < r$. But $s_1 \succ s_2$, so $\exists s' \in S \setminus \bigcup_{i=1}^{r-1} \omega^i_\succ(S)$ such that $s' \succ s$ contradicting $r = \texttt{rank}^S_\succ(s_2)$. □

As Troumpoukis et al. [5] use $\omega_\succ$ as their winnow operator, their solution performs correctly on acyclic preference relations. However, if the preference relation has loops, then none of the solutions in the loop will be in any skyline (and thus none of them will have a rank), even if there is no other solution dominating any solution in the loop. Even a simple reflexive loop, such as accidentially writing $\leq$ instead of $<$ as in Example 14, query Q'_6 above, causes problems for Definition 2. A preference written like this seems unintuitive, but it is not forbidden in SPREFQL. As arbitrarily complex SPARQL expressions can occur in the IF clause it might not be obvious or even possible beforehand, i.e., without querying the data, to determine whether a preference is irreflexive.

5 Non-Simple Comparative Preferences

To address such cases, we begin by addressing the empty-skyline problem mentioned above, which can be done by modifying the definition of skyline.

What makes intuitive sense in the presence of preference loops *in the candidate solutions* is to consider all the solutions in the loop as if they were equally preferred, i.e., they don't count as dominating each other or themselves. This regains the desirable property that a finite, non-empty set of candidate solutions has a non-empty skyline.

Formally, we modify the definition of skyline (Definition 2) as follows:

Definition 4. *If S is a finite set of solutions and $\succ$ a preference relation over some superset of S then the* skyline *of S with $\succ$ is*

$$\omega^l_\succ(s) = \{s \in S | \neg \exists s' \in S.(s' \succ^* s \wedge \neg(s \succ^* s'))\}$$

where $\succ^*$ is the transitive closure of $\succ$ over candidate *solutions, i.e., there is a sequence of candidate solutions, each dominating the next.*

In English, this says that a solution is in the skyline if there is no solution that transitively dominates it and that it does not transitively dominate.

This definition regains the desirable properties from above, slightly modified.

Theorem 3. *$\omega^l_\succ(S)$ is non-empty for S any finite, non-empty set of candidate solutions.*

Proof. The second part of the condition, $\neg(s \succ^* s')$, in the definition of $\omega'_\succ(S)$ prevents any solutions in a loop from knocking themselves (and each other) out of the loop. Because S is finite and non-empty it has to have maximal elements if loops are collapsed into a single solution. □

Corollary 2. *$\texttt{rank}^S_\succ(s)$ is defined for each $s \in S$ for S any finite, non-empty set of candidate solutions.*

Proof. Same as for Corollary 1. □

Loops cause the rank of solutions in the loop to be the same, so dominance only produces a rank at least as large.

Theorem 4. *s_1, s_2 in S $s_1 \succ s_2$ implies $\texttt{rank}^S_\succ(s_1) \leq \texttt{rank}^S_\succ(s_2)$.*

Proof. Assume otherwise, i.e., $s_1, s_2 \in S$ with $s_1 \succ s_2$ and $\texttt{rank}^S_\succ(s_1) > \texttt{rank}^S_\succ(s_2)$.
Let $r = \texttt{rank}^S_\succ(s_2)$ and $S^r = S \setminus \bigcup_{i=1}^{r-1} \omega^i_\succ(S)$.
Then $\neg\exists s' \in S^r.(s' \succ^* s_2 \wedge \neg(s_2 \succ^* s'))$.
Also $s_1 \in S^r$, because otherwise $r > \texttt{rank}^S_\succ(s_1)$.
Since $\texttt{rank}^S_\succ(s_1) > r$, therefore $\exists s' \in S^r.(s' \succ^* s_1 \wedge \neg(s_1 \succ^* s'))$.
Let s'' be one of these solutions.
Suppose $s_2 \succ^* s''$. Then $s_1 \succ s_2 \succ^* s''$ and so $s_1 \succ^* s''$ So $\neg(s_2 \succ^* s'')$.
But $s'' \succ^* s_1 \succ s_2$ and so $s'' \succ^* s_2$.
This contradicts $\neg\exists s' \in S^r.(s' \succ^* s_2 \wedge \neg(s_2 \succ^* s'))$. □

Note that transitive closure is needed for both the ancestor and the descendant of s. Consider a solution s that dominates only a single element s_1 of a minimal dominance cycle[12] $s_1, \ldots, s_n, s_1$, with $n > 2$. Now s should knock each s_i out of a skyline but no s_j should. To get to *only* s requires looking at the transitive dominators of s_i, not just its direct dominators.

Example 15. Getting back to the preference relation from the right-hand-side of Figure 1, the solution s_{Shell} dominates s_{Mobil} directly, but $s_{Chevron}$ and s_{Citgo} are dominated only indirectly. Thus, if in Definition 4 $s' \succ^* s$ were replaced by $s' \succ s$, then $s_{Chevron}$ and s_{Citgo} – counter to our intuition – would end up in the first skyline. On the other hand, if we replaced $\neg(s \succ^* s')$ with $\neg(s \succ s')$, it is easy to see that the second skyline would be empty, again going counter to our intuition.

As a special case (specifically considered here because of its relationship to SPARQL), if the preference relation is known to be transitive then there is no need to compute its transitive closure; that is, the following simpler definition suffices to deal such known transitive (including reflexive) preferences, even if they are cyclic. Here, in English, a solution is in the skyline if there is no solution that *directly* dominates it and that it does not *directly* dominate.

Definition 5. *If S is a finite set of (candidate) solutions and $\succ$ a transitive preference relation over some superset of S then the* skyline *of S with $\succ$ is*

$$\omega^t_\succ(S) = \{s \in S | \neg\exists s' \in S.(s' \succ s \wedge \neg(s \succ s'))\}$$

[12] A domination cycle is minimal if the only domination relationships between elements of the cycle are those from one element of the cycle to the next.

This definition maintains the desirable properties from Theorems 2, 3, and 4, for transitive preference relations. They all come from the simple observation that the transitive closure of a transitive relation is itself, so Definitions 4 and 5 coincide for transitive preference relations. Note that transitivity is not actually required in general for Definition 5 to suffice, but only transitivity into loops, i.e., a direct dominator of any solution in a loop is a direct dominator of every solution in the loop. We call the preference relation $\succ$ *clique-cyclic* in this case.

Theorem 5. *Let S be any finite set of (candidate) solutions. Let $\succ$ be clique-cyclic, i.e., for any solutions s_1 and s_2, if $s_1 \succ^* s_2$ and $s_2 \succ^* s_1$ then for any solution s if $s \succ s_1$ then $s \succ s_2$. Then $\omega^t_\succ(S) = \omega^l_\succ(S)$.*

Proof. Suppose $s' \in S$ such that $s' \succ s \wedge \neg(s \succ s')$. Because $\succ$ is clique-cyclic, if s' is in a loop with s then $s \succ s'$, so s' is not in a loop with s and so $\neg(s \succ^* s')$. Trivially, $s' \succ^* s$ so $s' \succ^* \wedge\neg(s \succ^* s')$.

Suppose there is no $s' \in S$ such that $s' \succ s \wedge \neg(s \succ s')$.

Consider s^2 such that $s^2 \succ^* s \wedge \neg(s \succ^* s^2)$.

It is not possible for s^2 to be s because $s \succ^* s$.

If s^2 is in a loop with s then $s^2 \succ s$. Trivially $\neg(s \succ s^2)$ so s^2 cannot be in a loop with s.

Consider a downward path from s^2 to s. Take the last node in the path that is neither s nor not in a loop with s and call it s^3. This node must exist because s^2 cannot be s and cannot be in a loop with s. The next node in the path is either s or is in a loop with s and call it s^4. If it is s then $s^3 \succ s$ but $\neg(s \succ s^3)$, which contradicts the assumption. If it is in a loop with s then $s^3 \succ s^4 \succ^* s$ and $s^3 \succ s$ because $\succ$ is clique-cyclic but $\neg(s \succ s^3)$, which again contradicts the assumption.

So then there is no $s' \in S$ such that $s' \succ^* s \wedge \neg(s \succ^* s')$.

So $\omega^t_\succ(S) = \omega^l_\succ(S)$. □

6 Comparative Preferences in SPARQL

As discussed in Section 3 above, languages like PrefSPARQL and SPREFQL have suggested translations to native SPARQL, thus showing a – not necessarily very efficient – implementation path in terms of off-the-shelf engines, and proving that the respective languages do not add expressivity on top of SPARQL. However, since both these translations only implement the simple skyline operator from Definition 2, we now turn to the question whether $\omega^t_\succ(S)$, $\omega^l_\succ(S)$ can likewise be expressed in SPARQL 1.1 or not.

As we want to allow for general comparative preferences, we adopt the SPREFQL syntax from Troumpoukis et al. [5] as opposed to the syntax of earlier work such as PrefSPARQL [4].

We recall the mapping from SPREFQL to SPARQL 1.1 by the schema given in Equation (3). While Troumpoukis et al. already suggest that evaluating this translation is potentially more expensive than directly implementing PREFER as it creates the cross product of solutions, for now we are concerned with mainly the expressibility of the different variations of skyline operators we introduced in SPARQL.

We recall there are two problems in the original translation: first, the use of NOT EXISTS in the translation into SPARQL, and second, that the de facto reliance on the skyline operator of Definition 2 only works for the simple case of acyclic preference relations.

The first problem can be overcome using the translation that uses `OPTIONAL` and `!BOUND()` instead of `NOT EXISTS` from Gueroussova et al. [14], cf. Example 12. This idea can be generalized to SPREFQL with the following mapping for Equation (2).

Mapping 1 (Simple Mapping to SPARQL)

```
SELECT V WHERE { P
      OPTIONAL { P_(V/V1) FILTER ( C_(V2/V) ) BIND 1 TO ?exists }
      FILTER (!BOUND(?exists)) }
```

Theorem 6. *Mapping1 correctly implements $\omega_{\succ}$.*

Proof. At the first filter there are unmodified variable bindings for one solution and modified $_1$ bindings for a second. The first filter then evaluates the preference relation on the modified and unmodified bindings. That is, it determines whether the solution from inside the OPTIONAL dominates the one from outside the OPTIONAL. If this is so then `?exists` *is bound. Then the second filter eliminates any solutions for which this is the case. So the only solutions that are retained are those with no dominator, as required for $\omega_{\succ}$.* □

Handling the second problem however requires changing the semantics of `PREFER`. When we view the intuitive meaning of Equation (2) decoupled from the mapping to SPARQL it conceptually reduces to first constructing the solution set S for

```
SELECT L WHERE { P }
```

and then eliminating non-dominated solutions of this query according to the chosen operator $\succ$. So, intuitively we need to repair the semantics to use our winnow operator ω^l, which as we showed above produces desirable results for any preference relation, instead of the original ω. Of course this is quite a significant change. On the plus side, it doesn't have problems with loops. On the negative side, it may require computing (a part of) the transitive closure of $\succ$. We will discuss next whether and how this computation can also be realized within SPARQL itself, or by means of bespoke algorithms.

7 Implementing SPARQL Preferences

The repaired version of comparative preferences can be efficiently implemented. Instead of just checking whether a solution has a direct dominator we have to check its transitive dominators and see whether they are in a loop. This sounds expensive, going around the loops over and over again, but can actually be done relatively efficiently.

The basic idea behind the algorithm is to check $\succ$ between each pair of candidate solutions. The algorithm keeps track of a representative for each solution which represents all the solutions that are in loops involving the solution. When a new loop is

found, the solutions in the loop are given the same representative. This operation has to be done efficiently over the entire exploration, fortunately the union-find algorithm [18] does precisely this in time $O(n\log^*(n))$, where n is the number of candidate solutions. After the representatives are found, all that is needed is to check whether a solution has a direct dominator with a different representative.

Algorithm 1 *(Compute Ranks according to a preference relation)*

Inputs:

S *a set of SPARQL solution sets, with bindings from* V*, a set of SPARQL variables*
e *a boolean SPARQL expression over pairs of solution sets, i.e., there are two mappings* $V \to V_1$*,* $V \to V_2$ *and* e *is an expression with variables from* $V_1 \cup V_2$*,* $e(s_1, s_2)$ *is evaluated with bindings* $s_1(V/V_1) \cup s_2(V/V2)$

Effect: for each $s \in S$ rank(rep(s)) *is the skyline* s *is in*

Subroutines (union-find algorithm):

rep(s) *returns the representative of* s*; runs in nearly constant time*
unify(s,s') *takes representatives of two subsets of* S *and gives all the elements of both sets the same representative; runs in nearly constant time*

```
explore(s,n):
  Mark s as active, level n
  Set looping to +infinity
  For each s' ∈ S
    If e(s',s)
      If rep(s') is marked as done /* found a non-loop dominator */
        If rank(rep(s')) ≥ rank(rep(s))
          set rank(rep(s)) to rank(rep(s'))+1
      If rep(s') is marked as active, level m /* found a new loop */
        set rank of rep(s) and rep(s') to their minimum rank
        Unify rep(s) and rep(s')
        If m < looping, set looping to m
      Else
        eloop = explore(s')
        if eloop ≤ n /* still in a loop from s' */
          set rank of rep(s) and rep(s') to their minimum rank
          Unify rep(s) and rep(s')
        else /* found a non-loop dominator */
          if rank(rep(s')) ≥ rank(rep(s))
            set rank(rep(s)) to rank(rep(s'))+1
        if eloop < looping set looping to eloop
  if looping ≥ n /* not in loop with parent */
    mark rep(s) as done
```

return if looping $< n$ *then looping else +infinity*

Initially there are no marks
Set the representative of each $s \in S$ *to* s
Set the rank of each $s \in S$ *to* 1
For each s *in* S
  If s has no mark explore(s,1)

A simple analysis of the algorithm shows that it unifies all solutions that are in a loop. Then it is easy to see that solutions with rank 1 have no transitive dominators that are not unified with the solution, i.e, not in a loop with it. So the algorithm correctly computes $\omega^l_\succ$. An inductive argument then shows that the algorithm correctly computes the rank of all solutions.

The algorithm checks $\succ$ between each pair of solutions so its running time is dominated by the n^2 computations of $\succ$; the union-find algorithm only adds $O(n\log^*(n))$. The most significant change in actual running time between the computation of $\omega_\succ$ and $\omega^l_\succ$ will be due to not being able to quit checking for dominators of a solution when the first one is found.

If the preference relation is known to belong to either of the special cases discussed in the context of Definition 5, i.e., if either the prererence relation is acyclic, transitive or clique-cyclic, then it is possible to translate preferences back into SPARQL itself. For an *acyclic* preference relation the translation is the one in Mapping 1. For transitive preference relations (or, likewise, if $\succ$ is clique-cyclic) a slightly more complex translation is needed.

Mapping 2 (Mapping to SPARQL for transitive Preferences)

```
SELECT V WHERE { P
      OPTIONAL { P_(V/V1) FILTER ( C_(V2/V) && ! C_(V1/V,V2/V1) )
                 BIND 1 TO ?exists }
      FILTER (!BOUND(?exists)) }
```

Theorem 7. *Mapping 2 correctly implements the winnow operator* $\omega^t_\succ$.

Proof. The only difference between the $\omega_\succ$ *and* $\omega^t_\succ$ *is in their internal condition, which is the only change between the two mappings. As the first filter correctly implements the internal condition from* $\omega^t_\succ$ *is is thus correct.* □

The general case is much tougher to translate back to a single SPARQL query as it has to compute (part of) the transitive closure of $\succ$ over the solutions. If, however, we allow for multiple queries we can first construct a graph that reifies the solutions and asserts $\succ$ between them and then determine the skyline with a subsequent query against the union of that constructed graph and the original graph.

Mapping 3 (Mapping to Multiple SPARQL Queries)
Note: This mapping does not take into consideration solutions with unbound variables. Unbound variables can be handled by using OPTIONAL *constructs.*
Step 1*: generate a graph with a "fresh" name* `:SG` *in the current dataset containing nodes representing each solution of the original query by a CONSTRUCT query*

```
CONSTRUCT { _:b a :solution ; :_1 V^1 ; ...; :_n V^n } WHERE { P }
```

where V^i is the ith variable in V.
***Step 2**: Use a CONSTRUCT query to generate a graph with* `:dominates` *edges for solution domination and give it a "fresh" name* :DG *in the current dataset:*

```
CONSTRUCT { ?s1 :dominates ?s2 }
WHERE { ?GRAPH :SG { ?s1 a :solution ; :_1 V_1^1 ; ...; :_n V_1^n .
                     ?s2 a :solution ; :_1 V_2^1 ; ...; :_n V_2^n . }
        FILTER ( C ) }
```

***Step 3**: Finally, return the solutions that are not transitively dominated by a solution that they do not transitively dominate*

```
SELECT V FROM NAMED :SG  FROM NAMED :DG
WHERE { GRAPH :SG { ?s a :solution ; :_1 V^1 ; ...; :_n V^n . }
        GRAPH :DG { OPTIONAL { ?s1 :dominates+ ?s .
                               OPTIONAL { ?s :dominates+ ?s1. BIND(1 AS ?dd)}
                               FILTER ( !bound(?dd) )
                               BIND(1 AS ?d)}
        FILTER ( !bound(?d) ) }
```

As this is not a complete definition of the required mapping we do not provide a proof of its correctness.

We do not see any effective way of computing this transitive closure inside a *single* SPARQL query, so we believe that our repaired version of comparative preferences requires an extension to SPARQL. There have been respective proposals to add recursion to SPARQL [19][13], or likewise nested CONSTRUCT clauses[14] would allow us to encode the the steps above into one query; however our results [20] suggest that there is no effective mapping (that is, a mapping without a significant blowup) of such nested CONSTRUCT queries into a single SPARQL query.

8 Multiple Skylines

There are situations where more solutions than the top skyline are desired, and it is inconvenient to submit multiple different queries to get those additional solutions, manually eliminating prior solutions. Thus, a SPARQL extension that returns multiple levels of skylines, possibly including an indicator of each solution's rank, seems natural.

Siberski et al. [3] propose that a use of SPARQL's existing keyword pattern "`LIMIT` k", in combination with their proposed keyword `PREFERRING`, can inform the query evaluator to retrieve enough levels of skyline to produce solutions numbering at least k. However, this approach has 3 weaknesses: it gives the `LIMIT` modifier a counterintuitive special meaning when it is used in that combination; it precludes the use of `LIMIT` with

[13] This recursive operator could be used to realize $\succ$ and its transitive closure so our version of comparative preferences could be mapped into a single recursive SPARQL query.

[14] I,e., the possibility to use CONSTRUCT queries within FROM clauses, cf. https://www.w3.org/2009/sparql/track/issues/7

its usual meaning; and it does not support the specification of an explicit number of complete skylines to be returned.

To address these issues, we propose the addition of the keyword `SKYLINE`, to be used in conjunction with `LIMIT`, in either of the following patterns:

- `LIMIT SKYLINE` n `[TO` m`]` `[AS` v_{rank}`]` ... return complete skyline(s) with rank n (or with ranks n to m, with $1 \leq n \leq m$, resepcetively.
- `LIMIT SKYLINE ALL AS` v_{rank} ... return all skylines

The `AS` v_{rank} is optional if explicit limits are given. If present it adds a new binding to the solution bindings of the query whose variable is v_{rank} and whose value is the rank of the solution.

The absence of any `LIMIT SKYLINE` clause is equivalent to `LIMIT SKYLINE 1`, i.e., we start counting ranks from 1. It would be permitted to use `LIMIT`, in one of its traditional patterns, in the same query as these new proposed patterns. The use of n `TO` m limits can be used similar to the regular `LIMIT ... OFFSET` solution modifier combination to return skylines within a certain range. Other modifiers, such as `ORDER BY SKYLINE` that orders solutions by increasing rank skyline number could be defined analogously.

Our implementation of ω_l can be simply extended to compute which skyline a solution is in. Each solution is initially given a tentative skyline of 1. When a non-looping dominator of a solution is found the solution's representative is assigned to the tentative skyline that is the maximum of its previous tentative skyline and one plus the tentative skyline of the dominator's representative. As non-looping dominators are only found when their dominators have been completely processed their tentative skyline is their final one. In this way each solution representative is assigned to their skyline number so the algorithm can produce the first (top) skyline, solutions in the skyline(s) with ranks between n and m along with their skyline number, or all solutions with their skyline number. Note that, as opposed to the syntax we presented so far, which is 1-to-1 identical with Troumpoukis et al. [5, Figure 1], the LIMIT SKYLINE solution modifier needs new grammar productions added to SPREFQL, which are as follows:

$$
\begin{array}{lll}
\langle SolutionModifier \rangle & ::= & [\,\langle GroupClause \rangle\,][\,\langle HavingClause \rangle\,] \\
 & & [\,\langle PreferClause \rangle\,]\,[\,\langle OrderClause \rangle\,] \\
 & & [\,\langle LimitOffsetClauses \rangle\,]\,[\,\langle LimitSkylineClauses \rangle\,] \\
\langle LimitSkylineClauses \rangle & ::= & \texttt{'LIMIT SKYLINE'} \\
 & & (\,\langle INTEGER \rangle\,[\,\texttt{'TO'}\ \langle INTEGER \rangle\,]\,[\,\texttt{'AS'}\ \langle Var \rangle\,] \\
 & & \quad |\ \texttt{'ALL' 'AS'}\ \langle Var \rangle\,)
\end{array}
$$

9 Conclusions

We have considered the semantics of queries that rely on user preferences, extensions of SPARQL that would allow for handling such queries, and their possible implementation by translation into SPARQL. We identified several categories of preference relations with different characteristics that are significant in terms of specifying preference query semantics and specifying translations to SPARQL. By "preference relation semantics", we (loosely) mean the simplest definition of skyline that provides the desired

results from a preference query. We summarize our conclusions, in order of increasing generality of the allowed preference relations.

- The semantics of *acyclic* preference relations are as (implicitly) indicated by Troumpoukis et al. [5]. However, we identified a problem with their translation to SPARQL, and showed how it can be repaired.
- Transitive, irreflexive preference relations, which occur in many applied settings, as a special case of acyclic preference relations, are subject to the same observations.
- For transitive preference relations in general, a modified semantics is needed, as well as a slightly more complex translation into SPARQL. This semantics allows for more efficient processing than the most general semantics mentioned below.
- We defined a category of *clique-cyclic* preference relations (a superset of transitive preference relations, cf. Theorem 5), which can be dealt with the same semantics as transitive preference relations.
- Finally, for arbitrary preference relations, we gave a semantics that is slightly more complex than that for transitive preference relations, and showed that preference queries can be implemented by translation to multiple (sequentially executed) SPARQL queries.

In addition, we showed that our proposed semantics and implementation for each of these categories satisfies basic desiderata for the results of queries with preferences, and we discussed an algorithm that would be more efficient than the implementation by multiple, nested SPARQL queries. Finally, we proposed an additional SPARQL extension that provides a means of retrieving multiple skylines from a single query.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5) (2001) 28–37
2. Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M.H., Tran, L.: Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0 (January 2004) W3C Recommendation.
3. Siberski, W., Pan, J.Z., Thaden, U.: Querying the semantic web with preferences. In: The 5th International Semantic Web Conference (ISWC 2006). (2006) 612–624
4. Gueroussova, M., Polleres, A., McIlraith, S.A.: SPARQL with qualitative and quantitative preferences. In: 2nd International Workshop on Ordering and Reasoning (OrdRing 2013), at ISWC 2013. (2013) CEUR Workshop Proceedings, Volume 1059.
5. Troumpoukis, A., Konstantopoulos, S., Charalambidis, A.: An extension of SPARQL for expressing qualitative preferences. In: The 16th International Semantic Web Conference (ISWC 2017). (2017) 711–727
6. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (March 2013) Available at http://www.w3.org/TR/sparql11-query/.
7. Borzsonyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings 17th International Conference on Data Engineering. (2001) 421–430
8. Tversky, A.: Intransitivity of preferences. Psychological review **76**(1) (1969) 31–48
9. Bar-Hillel, M., Margalit, A.: How vicious are cycles of intransitive choice? Theory and Decision **24**(2) (Mar 1988) 119–145
10. Nurmi, H.: Making sense of intransitivity, incompleteness and discontinuity of preferences. In: Group Decision and Negotiation. A Process-Oriented View, Springer (2014) 184–192

11. Chomicki, J.: Preference formulas in relational queries. ACM Transactions on Database Systems **28**(4) (2003) 427–466
12. Buil-Aranda, C., Arenas, M., Corcho, O., Polleres, A.: Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. Journal of Web Semantics **18**(1) (2013) 1–17
13. Kießling, W., Köstler, G.: Preference SQL - design, implementation, experiences. In: 28th International Conference on Very Large Data Bases. (2002) 990–1001
14. Gueroussova, M., Polleres, A., McIlraith, S.A.: SPARQL with qualitative and quantitative preferences (extended report). University of Toronto CSRG Report 619 (2013)
15. Kießling, W., Endres, M., Wenzel, F.: The preference SQL system - an overview. IEEE Data Engineering Bulletin **34**(2) (2011) 11–18
16. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation, https://www.w3.org/TR/rdf-sparql-query/ (2008)
17. Patel-Schneider, P.F., Martin, D.: EXISTStential aspects of SPARQL. In: The 15th International Semantic Web Conference (ISWC 2016). (October 2016)
18. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM **22**(2) (1975) 215–225
19. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. In: The 14th International Semantic Web Conference (ISWC 2015). (October 2015)
20. Polleres, A., Reutter, J., Kostylev, E.V.: Nested constructs vs. sub-selects in SPARQL. In: Alberto Mendelzon International Workshop on Foundations of Data Management (AMW2016). (June 2016)