An extension of SPARQL for expressing qualitative preferences

Antonis Troumpoukis[1][2], Stasinos Konstantopoulos[1], and Angelos Charalambidis[1]

[1] Institute and Informatics and Telecommunications, NCSR 'Demokritos'
Aghia Paraskevi 15310, Athens, Greece
{antru,konstant,acharal}@iit.demokritos.gr
[2] Department of Informatics and Telecommunications, University of Athens, Greece

Abstract. In this paper we present SPREFQL, an extension of the SPARQL language that allows appending a "PREFER" clause that expresses 'soft' preferences over the query results obtained by the main body of the query. The extension does not add expressivity and any SPREFQL query can be transformed to an equivalent standard SPARQL query. However, clearly separating preferences from the 'hard' patterns and filters in the "WHERE" clause gives queries where the intention of the client is more cleanly expressed, an advantage for both human readability and machine optimization. In the paper we formally define the syntax and the semantics of the extension and we also provide empirical evidence that optimizations specific to SPREFQL improve run-time efficiency by comparison to the usually applied optimizations on the equivalent standard SPARQL query.

Keywords: SPARQL query processing; expressing preferences; query execution optimization.

1 Introduction

Preferences can be used in situations where, while looking for the best solution with respect to a set of criteria, we find out that too strict criteria might not return any solutions, but relaxing them returns too many solutions to sift through. The integration of preferences allows to view some constraints as soft constraints that can be violated in the former case and return less-preferred results, but will be enforced in the latter case to only return more-preferred results.

Preferences have been explored in Artificial Intelligence [8], Database Systems [21], Programming Languages [7], and, more recently, enjoy a growing interest in the area of the Semantic Web [17]. In the Semantic Web context, preferences allow users to sift through data of varying trustworthiness, quality, and relevance from a specific end user's point of view [22]. As argued by Siberski et al. [20], the motivating example in the beginning of the seminal Semantic Web article [2] can be interpreted as a preference search.

Strictly speaking, preferences are not more expressive than standard SPARQL. Their most prominent feature, returning less-preferred binding sets in the absence of more-preferred ones, can be simulated with "NOT EXISTS" and, in general, with the syntax already offered by SPARQL. However, clearly separating preferences from the 'hard' patterns and filters in the "WHERE" clause gives us queries where the intention of the author is cleanly expressed and not obscured. This has advantages in both human readability and machine optimization.

In this paper, we first give a background on the treatment of preferences in databases (Section 2) and proceed to present our proposed SPREFQL syntax and semantics (Section 3). We then present our SPREFQL query processor implementations and our benchmarks on them (Section 4). These empirical results are used to support our claim above that optimizing directly at the SPREFQL syntax is more efficient than rewriting into standard SPARQL and passing the latter to an optimizing SPARQL query processor. We then present some related work on the Semantic Web and compare it with our approach (Section 5). We close the paper with conclusions and future research directions (Section 6).

2 Background

Preference representation formalisms are either *quantitative*, where preferences are represented by a preference value function [1, 13], or *qualitative*, where preferences are expressed by directly defining a binary preference relation between objects [5, 11]. In the example below:

Example 1. Show me Sci-fi movies, assuming I prefer longer movies.

there is a hard constraint for SciFi movies and a preference towards longer movies. Such a constraint can be represented both as a quantitative function of the movies' runtime and as a qualitative relation that compares movies' runtimes. With this example, however:

Example 2. Show me Sci-fi movies, assuming I prefer original movies to their sequels.

it becomes apparent that there are cases where not all objects are directly comparable, and therefore the total ordering implied by the preference value function cannot always be defined. In fact, Chomicki [5] argues that the qualitative approach is strictly more general than the quantitative approach, as not all preference relations can be expressed using a preference value function. In Chomicki's framework, *preference relations* are defined using first-order formulas:

Definition 1. *Given a relation schema* $R(A_1, \ldots, A_n)$ *such that* U_i, $1 \leqslant i \leqslant n$, *is the domain of the attribute* A_i, *a relation* $\succ$ *is a preference relation over* R *if it is a subset of* $(U_1 \times \cdots \times U_n) \times (U_1 \times \cdots \times U_n)$. *A result tuple* t_1 *is said to be* dominated *by* t_2, *if* $t_2 \succ t_1$.

This general preference relation is restricted into *intrinsic preference formulas* that do not rely on external information to compare two objects:

Definition 2. *Let t_1, t_2 denote tuples of a given database relation. A* preference formula *$P(t_1, t_2)$ is a first-order formula defining a preference relation $\succ_P$ in the standard sense, namely, $t_1 \succ_P t_2$ iff $P(t_1, t_2)$ holds. An* intrinsic preference formula *is a preference formula that uses only built-in predicates (i.e. equality, inequality, arithmetic comparison operations, and so on).*

Example 3. Consider the `movie(ID,Title,Genre,Duration)` relation shown in Table 1. Suppose that we have the following preference: 'I prefer one `movie` tuple over another iff their genre is the same and the first one runs longer'. The preference relation $\succ_P$ implied by the previous sentence can be defined using formula P:

$$(i, t, g, d) \succ_P (i', t', g', d') \equiv (g = g') \wedge (d > d').$$

Therefore, we prefer movie m_3 to m_2, movie m_2 to m_1, m_3 to m_1 and movie m_4 to m_5. Both conjuncts must be satisfied for the preference relation to hold, so there is no preference relation between movies from different genres regardless of their runtime.

A new relational algebra operator is introduced, called *winnow*. This operator takes two parameters, a database relation and a preference formula and selects from its argument relation the most preferred tuples according to the given preference relation.

Preference relations can be composed in order to form more complex ones. Since preference relations are defined through preference formulas, in order to combine two such relations one must combine their corresponding formulas. Given two preference relations $\succ_P, \succ_Q$, the most common composition operations are the following:

- *Boolean:* (e.g. intersection) $t_1 \succ_{P \wedge Q} t_2 \equiv (t_1 \succ_P t_2) \wedge (t_1 \succ_Q t_2)$,
- *Pareto:* $t_1 \succ_{P \otimes Q} t_2 \equiv ((t_1 \succ_P t_2) \wedge (t_2 \nsucc_Q t_1)) \vee ((t_1 \succ_Q t_2) \wedge (t_2 \nsucc_P t_1))$,
- *Prioritized:* $t_1 \succ_{P \rhd Q} t_2 \equiv (t_1 \succ_P t_2) \vee ((t_1 \sim_P t_2) \wedge (t_1 \succ_Q t_2))$,

where $t_1 \nsucc_P t_2 \equiv \neg(t_1 \succ_P t_2)$ and $t_1 \sim_P t_2 \equiv (t_1 \nsucc_P t_2) \wedge (t_2 \nsucc_P t_1)$.

In order to select the 'best' tuples from a given relation r based on a preference formula P, the *winnow* operator is introduced:

Table 1. A sample movies relation.

ID	Title	Genre	Duration	Sequel
m_1	Star Wars Ep.IV: A New Hope	Sci-fi	121	m_2
m_2	Star Wars Ep.V: The Empire Strikes Back	Sci-fi	124	m_3
m_3	Star Wars Ep.VI: Return of the Jedi	Sci-fi	130	
m_4	Die Hard	Action	131	m_5
m_5	Die Hard with a Vengeance	Action	128	

Definition 3. *Let r be a relation and let P be a preference formula defining a preference relation $\succ_P$. The* winnow *operator is defined as*

$$w_P(r) = \{t \in r : \neg\exists t' \in r \text{ such that } t' \succ_P t\}.$$

Example 4. Given the relation `movie` in Table 1 and the preference formula C of Example 3, the result of the $w_P(\texttt{movie})$ operation is the movies with IDs m_3 and m_4. m_1 and m_2 are not included in the result because they are less preferred than m_3 and m_5 because it is less preferred than m_4. Since there is no preference relation between m_3 and m_4, they are both included in the result.

Although winnow can be expressed using standard relational algebra operators [5], there also exist algorithms that directly compute the result of the winnow operator $w_P(R)$. The most prominent such algorithms are the *Nested Loops (NL)* algorithm and the *Blocked Nested Loops (BNL)* algorithm. In NL, each tuple of R is compared with all tuples in R, therefore the complexity of NL is quadratic in the size of R. In BNL, a fixed amount of main memory (a *window*) is used, in order to keep a set of incomparable tuples, which at the end of the algorithm will become the dominating tuples of R. Even though the asymptotic time complexity of BNL is also quadratic, in practice BNL performs better than NL. Especially in the case that the result set of winnow fits into the window, the algorithm operates in one or two iterations (i.e. linear time to the size of R) [3]. Regarding the correctness of the result of each algorithm, NL produces the correct result for every preference relation (even in unintuitive cases such as preference relations in which a tuple is preferred to itself). On the other hand, BNL produces the correct result only if the preference relation $\succ$ is a *strict partial order* [5], that is to say iff the relation is (1) *irreflexive* $\neg(x \succ x)$ (2) *transitive* $(x \succ y) \wedge (x \succ z) \Rightarrow (x \succ z)$ and (3) *asymmetric* $(x \succ y) \Rightarrow \neg(y \succ x)$.

Example 5. Let us assume the relation `movie` in Table 1 and the following preference formula C':

> 'I prefer one `movie` tuple over another iff their genre is the same and the first one has the second as sequel.'

In this case, BNL is *not* guaranteed to produce the correct result because m_1 'sequel' m_2 and m_2 'sequel' m_3, but m_1 'sequel' m_3 is not asserted, making the 'sequel' property (and thus the whole preference relation) not transitive. The result of the BNL algorithm depends on the order in which pairs are tested: if m_2 is compared to m_1 before being compared to m_3, the first comparison will remove m_2 from the window making m_1 and m_3 incomparable and the result is $\{m_1, m_3, m_4\}$; if m_2 is compared to m_3 before being compared to m_1, then both m_3 and m_2 will be removed and the result is $\{m_1, m_4\}$.

3 The SPREFQL Language

In this section we introduce SPREFQL, which is an extension of SPARQL that supports the expression of qualitative preferences. User preferences are expressed as a new solution modifier which eliminates the solutions that are dominated by (i.e., are less preferred than) another solution. This modifier is similar to a preference formula in Chomicki's framework discussed above. In this section we present the syntax and the semantics of SPREFQL, discuss its expressive power, and we will give some examples of SPREFQL queries.

3.1 Syntax

We assume as a basis the EBNF grammar that defines SPARQL syntax [10, Section 19.8] and we extend it by changing the definition of the ⟨*SolutionModifier*⟩ non-terminal (Rule 18). The new definition adds a ⟨*PreferClause*⟩ non-terminal between the ⟨*HavingClause*⟩ and the ⟨*OrderClause*⟩ non-terminals. The rationale for this positioning is that:

- The prefer clause should be after the group-by/having clauses, as it would make sense to use in the former the aggregates computed by the latter.
- The prefer clause should be before the limit/offset clauses, as it would be counter-intuitive to miss preferred solutions because they have been limited out, so the limit should apply to the preferred solutions.
- The prefer clause could equivalently be either before or after the order-by clause, but there is no reason to sort solutions that are going to be discarded afterwards. Naturally an optimizer could also re-order these computations, but there is no reason why the default execution plan should not put these in the more efficient order already. A further advantage of placing the prefer clause before the order-by clause is that this avoids requiring from compliant SPREFQL implementations that they maintain the order of the result set.

Fig. 1. The SPREFQL grammar.
The full EBNF grammar for SPREFQL is the result of starting with the grammar for SPARQL 1.1 [10, Section 19.8], replacing Rule 18 with the first rule below, and appending the rest of the rules below.

```
⟨SolutionModifier⟩  ::= [ ⟨GroupClause⟩ ] [ ⟨HavingClause⟩ ] [ ⟨PreferClause⟩ ]
                        [ ⟨OrderClause⟩ ] [ ⟨LimitOffsetClauses⟩ ]

⟨PreferClause⟩      ::= 'PREFER' ⟨VarList⟩ 'TO' ⟨VarList⟩ 'IF' ⟨ParetoPref⟩

⟨VarList⟩           ::= ⟨Var⟩
                     |  '(' ⟨Var⟩+ ')'

⟨ParetoPref⟩        ::= ⟨PrioritizedPref⟩ [ 'AND' ⟨ParetoPref⟩ ]

⟨PrioritizedPref⟩   ::= ⟨BasicPref⟩ [ 'PRIOR' 'TO' ⟨PrioritizedPref⟩ ]

⟨BasicPref⟩         ::= '(' ⟨ParetoPref⟩ ')'
                     |  ⟨SimplePref⟩

⟨SimplePref⟩        ::= ⟨Constraint⟩
```

Figure 1 gives the EBNF rules that define $\langle$*PreferClause*$\rangle$ and also re-define $\langle$*SolutionModifier*$\rangle$. All non-terminals that are not defined in this table are defined by standard SPARQL syntax: $\langle$*GroupClause*$\rangle$ (Rule 19), $\langle$*HavingClause*$\rangle$ (Rule 21), $\langle$*OrderClause*$\rangle$ (Rule 23), $\langle$*LimitOffsetClauses*$\rangle$ (Rule 25). $\langle$*Constraint*$\rangle$ (Rule 69), and $\langle$*Var*$\rangle$ (Rule 108). Note, in particular, how basic preferences are a conjunction of the standard SPARQL $\langle$*Constraint*$\rangle$ used in the definitions of `"HAVING"` and `"FILTER"` clauses. This means that preferences are expressed using the familiar syntax of SPARQL constraints.

In the remainder, we shall call *query base* $B(\mathcal{Q})$ the standard SPARQL query that is derived from a SPREFQL query $\mathcal{Q}$ by removing the `"PREFER"` clause. We shall also call *full result set* the result set of $B(\mathcal{Q})$ and *preferred result set* the result set of $\mathcal{Q}$. We continue with a simple example in SPREFQL.

Example 6. Suppose that we want to query an RDF database with movies and we have the following preference:

> 'I prefer one movie to another iff their genre are the same and the first one runs longer.'

The size of the preferred result set is equal to the number of the available genres in the dataset (since two films with different genre are incomparable). For each genre, the selected film must be the one with the longest runtime. The corresponding SPREFQL query is listed in Listing 1.

To express preference of one binding set over another, we first use the `"PREFER"` clause to assign variable names to the bindings in the two binding sets, so that the two binding sets can be distinguished from each other. We then use the `"IF"` clause to express the conditions that make the first binding set dominate the second one. In the query in Listing 1, for example, there are three bindings in each result, `(?title ?genre ?runtime)`. In order to compare two binding sets, the `"PREFER"` clause assigns the bindings in the first result to the variables `(?title1 ?genre1 ?runtime1)` and the bindings in the second result to the variables `(?title2 ?genre2 ?runtime2)`. These new variable names are then used in the `"IF"` clause to specify when the first result dominates the second result. Notice that any name can be used for the variables in the `"PREFER"` clause, and what maps them to the variables in the `"SELECT"` clause is the order of appearance. For example, in this query, variables `?title1`, `?title2` correspond to variable `?title`, the variables `?genre1`, `?genre2` correspond to variable `?genre` and so on. Note also that the names in the `"PREFER"` clause need to be distinct from each other, but they do *not* need to be distinct from the names in the `"SELECT"` clause. In this manner, the style shown in Listing 2 is also possible, if the query author prefers it.

Given the above, we define well-formed SPREFQL queries as follows:

Definition 4. *Let* $\mathcal{Q}$ = `SELECT` L `WHERE` P_1 `PREFER` L_1 `TO` L_2 `IF` P_2 *be a SPREFQL query produced by the grammar of Figure 1. Then,* $\mathcal{Q}$ *is well-formed iff* $|L| = |L_1| = |L_2|$ *and all variables of* L_1, L_2 *are distinct.*

Listing 1. 'I prefer one movie over another iff their genre is the same and the duration of the first is longer'.

```
SELECT ?title ?genre ?runtime WHERE {
 ?s a :film. ?s :title ?title. ?s :genre ?genre. ?s :runtime ?runtime.
}
PREFER (?title1 ?genre1 ?runtime1) TO (?title2 ?genre2 ?runtime2)
IF (?genre1 = ?genre2 && ?runtime1 > ?runtime2)
```

Listing 2. 'I prefer one movie over another iff their genre is the same and the duration of the first is longer'.

```
SELECT ?title ?genre ?runtime WHERE {
 ?s a :film. ?s :title ?title. ?s :genre ?genre. ?s :runtime ?runtime.
}
PREFER (?t ?genre ?runtime) TO (?otherT ?otherGenre ?otherRuntime)
IF (?genre = ?otherGenre && ?runtime > ?otherRuntime)
```

Listing 3. 'Given two action movies, I prefer the longest one and more recent one with equal importance'.

```
SELECT ?title ?genre ?runtime WHERE {
 ?s a :film. ?s :genre :action.
 ?s :title ?title. ?s :runtime ?runtime. ?s :year ?year.
}
PREFER (?title1 ?runtime1 ?year1) TO (?title2 ?runtime2 ?year2)
IF (?runtime1 > ?runtime2) AND (?year1 > ?year2)
```

Listing 4. 'Given two action movies, I prefer the one that runs between 115 and 125 minutes. If they are the same to me according to this criterion, I prefer the ones that they are after 2005'.

```
SELECT ?title ?genre ?runtime WHERE {
 ?s a :film. ?s :genre :action.
 ?s :title ?title. ?s :runtime ?runtime. ?s :year ?year.
}
PREFER (?title1 ?run1 ?year1) TO (?title2 ?run2 ?year2)
IF ( ?run1 >= 115 && ?run1 <= 125 && (?run2 < 115 || ?run2 > 125) )
PRIOR TO (?year1 >= 2005 && ?year2 < 2005)
```

Listing 5. 'I want to watch a movie with "Mad Max" in the title, and I prefer original movies to their sequels'.

```
SELECT ?film ?title WHERE {
 ?film a :film . ?film :title ?title. FILTER regex(?title, "Mad␣Max").
}
PREFER (?film1 ?title1) TO (?film2 ?title2)
IF EXISTS { ?film1 :sequel ?film2 }
```

Listing 6. Rewrite of the "PREFER" clause in Listing 3 without using the "AND" combinator.

```
PREFER (?title1 ?runtime1 ?year1) TO (?title2 ?runtime2 ?year2)
IF ( ((?runtime1 > ?runtime2) && !(?year2 > ?year1))
  || ((?year1 > ?year2) && !(?runtime2 > ?runtime1)) )
```

Listing 7. Rewrite of the "PREFER" clause in Listing 4 without using the "PRIOR TO" combinator.

```
PREFER (?title1 ?run1 ?year1) TO (?title2 ?run2 ?year2)
IF ( (?run1 >= 115 && ?run1 <= 125 &&  (?run2 < 115 || ?run2 > 125))
     ||
     ( !(?run1 >= 115 && ?run1 <= 125 && (?run2 < 115 || ?run2 > 125)) &&
       !(?run2 >= 115 && ?run2 <= 125 && (?run1 < 115 || ?run1 > 125)) &&
        (?year1 >= 2005 && ?year2 < 2005)
     ) )
```

In Section 2 we presented some ways so that two preference relations can be combined into one more complex one. As in the framework of Chomicki, we can also use boolean operators to combine the individual boolean expressions (boolean composition). Besides logical operators, we offer the following two preference combinators for combining preference relations:

- Pareto composition: the "AND" combinator composes a relation from two preference relations that are of equal importance (cf. Listing 3). We follow previous work [12, 20] in using "AND" for the Pareto combinator, noting that it should not be confused with the logical conjunction operator.
- Prioritized composition: the "PRIOR TO" combinator composes a preference relation where the less-important right-hand side argument is only applied if the more-important left-hand side argument does not impose any preference between two object (cf. Listing 4).

These combinations can be expressed within a simple constraint with the elaborate use of boolean operators. But this 'syntactic sugar' makes useful expressions a lot more readable. Compare, for example, the queries in Listings 3 and 4 with their equivalent queries without using the "AND" and "PRIOR TO" combinators, in Listings 6 and 7 respectively.

Since a basic simple preference is a *Constraint*, anything that can appear as a parameter in a SPARQL "FILTER" clause can be used as a simple basic user preference, and has the same meaning as in SPARQL "FILTER" clauses. This could be also an "EXISTS" expression, as it is shown in Listing 5. These type of preference relations are known as *extrinsic* preferences [5], and are not supported by Chomicki's framework. A preference relation is extrinsic if the decision of whether an element is preferred over another depends not only on the values of the elements themselves, but also on external factors (such as the the `:sequel` predicate in our example).

3.2 Semantics

In this section we will define the semantics of SPREFQL. Our semantics extend the standard semantics of SPARQL [10]. We assume basic familiarity of the semantics of SPARQL, but we will present some basic terminology when needed.

We denote by $\mathbf{T}$ the set of all *RDF terms* and by $\mathbf{V}$ the set of all *variables*. A *mapping* μ is a partial function $\mu : \mathbf{V} \rightarrow \mathbf{T}$. The *domain* of a mapping μ, denoted as $\mathsf{dom}(\mu)$ is the subset of $\mathbf{V}$ where μ is defined. It is straightforward to see that mappings express variable bindings and that given a mapping μ it is always possible to construct a `"VALUES"` clause that expresses the same bindings as μ does.

Example 7. Let $\mu = \{(\mathtt{g}, \texttt{"Sci-fi"}), (\mathtt{r}, 121)\}$ Then μ expresses the same binding of variable `"?g"` as the clause `"VALUES ( ?g ?r ) { "Sci-fi" 121 }"`.

Following Pérez et al. [16] we denote by $[\![\cdot]\!]_D$ the *evaluation* of a SPARQL query over a dataset D. If a query Q is a SELECT query, then $[\![Q]\!]_D$ is a set of mappings, which are the solutions that satisfy Q over D. If Q is an ASK query, then $[\![Q]\!]_D$ is equal to true if there exists any solution for Q in D, otherwise it is equal to false.

We will now continue with the semantics of the preference solution modifier. Firstly though, we have to include some preliminary definitions:

Definition 5. *Let $L = (l_1, \ldots, l_n), B = (b_1, \ldots, b_n)$ be two variable lists and μ be a mapping s.t. $\mathsf{dom}(\mu) = \mathcal{B}$, where $\mathcal{B}$ is the set of all variables of B. Then, we denote by $\mathsf{Rename}_{B \rightarrow L}(\mu)$ a mapping that is created from μ by renaming variable b_i to l_i, for all $i = 1, \ldots, n$.*

Definition 6. *Let L, L', B be three variable lists, s.t. $|L| = |L'| = |B|$ and all variables that appear in L, L' are distinct. Also, let μ, μ' be two mappings s.t. $\mathsf{dom}(\mu) = \mathsf{dom}(\mu') = \mathcal{B}$, where $\mathcal{B}$ is the set of all variables of B. Then, we denote by $\mathsf{ConstructMapping}_{B \rightarrow L, B \rightarrow L'}(\mu, \mu')$ a mapping such that*

$$\mathsf{ConstructMapping}_{B \rightarrow L, B \rightarrow L'}(\mu, \mu') = \mathsf{Rename}_{B \rightarrow L}(\mu) \cup \mathsf{Rename}_{B \rightarrow L'}(\mu').$$

Definition 7. *Let C be a* SPARQL Constraint *and μ be a mapping. Then, we denote by $\mathsf{ConstructQuery}(C, \mu)$ a query of the form* `"ASK { FILTER` C S `}"` *where s is the* SPARQL ValuesClause *that corresponds to the mapping μ. Note:* SPARQL Constraint *and* SPARQL ValuesClause *as defined in the SPARQL specification [10].*

Example 8. Let $\mu = \{(\mathtt{g}, \texttt{"Sci-fi"}), (\mathtt{r}, 121)\}$, $\mu' = \{(\mathtt{g}, \texttt{"Sci-fi"}), (\mathtt{r}, 124)\}$, $B = (\mathtt{g}, \mathtt{r})$, $L = (\mathtt{g1}, \mathtt{r1})$, $L' = (\mathtt{g2}, \mathtt{r2})$ and $C =$ `"( g1 = g2 && r1 > r2 )"`. Then,

$$\mathsf{ConstructMapping}_{B \rightarrow L, B \rightarrow L'}(\mu, \mu') = \mu^* = \left\{ \begin{array}{l} (\mathtt{g1}, \texttt{"Sci-fi"}), (\mathtt{r1}, 121), \\ (\mathtt{g2}, \texttt{"Sci-fi"}), (\mathtt{r2}, 124) \end{array} \right\}$$

$\mathsf{ConstructQuery}(C, \mu^*) =$

```
"ASK { FILTER ( ?g1 = ?g2 && ?r1 > ?r2 )
       VALUES ( ?g1 ?r1 ?g2 ?r2 )
              { ( "Sci-fi" 121 "Sci-fi" 124 ) } }"
```

As stated earlier, our preference solution modifier expresses a preference relation between the results of the query base, therefore the meaning of the `"PREFER"` clause is actually a binary predicate p such that $p(\mu, \mu')$ holds if μ is preferred over μ'. Hence, below, the *evaluation* $[\![\cdot]\!]_D$ of a `"PREFER"` clause takes two mappings as input. Recall that except from a simple *Constraint*, a preference relation can be expressed using the *Pareto* and *Prioritized* preference compositors.[3]

Definition 8. *Let D be a dataset. Also, let C be a constraint and L, L', B be three variable lists, s.t. $|L| = |L'| = |B|$ and all variables that appear in L, L' are distinct. Also, let μ, μ' be two mappings s.t. $\mathsf{dom}(\mu) = \mathsf{dom}(\mu') = \mathcal{B}$, where $\mathcal{B}$ is the set of all variables of B. Then,*

$$[\![\texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } C]\!]_{D,B} = \big\{ \big(\mu, \mu'\big) : [\![\mathsf{ConstructQuery}(C, \mu^*)]\!]_D = \mathsf{true}, \\ \mu^* = \mathsf{ConstructMapping}_{B \to L, B \to L'}(\mu, \mu') \big\}$$

Composite clauses using the `"PRIOR TO"` and `"AND"` combinators are defined as follows:

1. $[\![\texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } P \texttt{ PRIOR TO } Q]\!]_{D,B} = [\![\mathcal{P}]\!]_{D,B} \rhd [\![\mathcal{Q}]\!]_{D,B}$,
2. $[\![\texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } P \texttt{ AND } Q]\!]_{D,B} = [\![P]\!]_{D,B} \otimes [\![Q]\!]_{D,B}$,

where $\mathcal{P} = \texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } P$, $\mathcal{Q} = \texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } Q$, C is a constraint expression and P, Q non-terminal symbols.

Notice that in Example 8, $[\![\texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } C]\!]_D(\mu, \mu') = \mathsf{true}$ for every dataset D, or in other words the evaluation of the corresponding preference predicate is independent from the dataset D. This is the case for all constraint expressions that use only built-ins. The reason why we use the construction of this ASK query, is in the case of preferences that are defined with the use of an EXISTS expression (see for example Listing 5). In that example, in order to check whether a mapping is preferred from another, one has to check the dataset D for the existence of the corresponding `:sequel` triple.

Having defined the meaning of preference relations, we can proceed to define how the preference solution modifier uses a preference relation to reduce the full result set of the query base into the preferred result set. For this, we refer to the *winnow* operator $w_P([\![Q]\!]_D)$ which outputs the preferred result set when given the preference relation P and the full result set $[\![Q]\!]_D$ (cf. Definition 3).

Definition 9. *Let Q be a SELECT query. Then, we denote by $\mathsf{ProjVarList}(Q)$ the projection list in the same order that it appears in the SELECT clause.*

Definition 10. *Let D be a dataset, Q be a SELECT query and L, L' be two variable lists such that $|\mathsf{ProjVarList}(Q)| = |L| = |L'|$ and all variables that appear in L, L' are distinct. Then,*

$$[\![Q \texttt{ PREFER } L \texttt{ TO } L' \texttt{ IF } C]\!]_D = w_{[\![\texttt{PREFER } L \texttt{ TO } L' \texttt{ IF } C]\!]_{D,B}}([\![Q]\!]_D),$$

where $B = \mathsf{ProjVarList}(Q)$ and C be a non terminal symbol.

[3] We use a slightly different notation in the following definitions from the definitions in Section 2. Instead of writing $[\![\mu \succ_C \mu']\!]_{D,B}$ we write $[\![C]\!]_{D,B}(\mu, \mu') = \mathsf{true}$. In addition, the operators $\rhd$ and $\otimes$ correspond to the Prioritized and Pareto compositions.

3.3 Expressive power of SPREFQL

Winnow can be expressed using standard relational algebra operators [5]. Therefore, a SPREFQL query, which is essentially a SPARQL 1.1 query extended with a winnow operation, can be also expressed using standard SPARQL 1.1, using a `"NOT EXISTS"` query rewriting. Given a SPREFQL query of the form

$$\texttt{SELECT}\ L\ \texttt{WHERE}\ \{\ P\ \}\ \texttt{PREFER}\ L_1\ \texttt{TO}\ L_2\ \texttt{IF}\ C$$

the preferred result set consists of the result mappings of the query base that are the most preferred ones, or equivalently all mappings in the full result set such that there does not exist any mapping that is more preferred. This fact can be expressed using a standard SPARQL query of the following form

$$\texttt{SELECT}\ L\ \texttt{WHERE}\ \{\ P\ \texttt{FILTER}\ \texttt{NOT}\ \texttt{EXISTS}\ \{\ P_{\{L/L_1\}}\ \texttt{FILTER}\ C_{\{L_2/L\}}\ \}\ \}$$

where $P_{\{L/L_1\}}$ is created by P by replacing all variable names of P that appear in L with its corresponding variable in L_1, and $C_{\{L_2/L\}}$ is created by C by replacing all variable names of C that appear in L_2 with its L. The remaining variables on the new constructions are replaced with fresh variables. If C is a Pareto or a Prioritized composition, we first apply the rewritings into their corresponding simple preferences (ref. Section 2, Listing 6 and Listing 7). For example, the corresponding rewriting of Listing 1 is illustrated in Listing 8.

Comparing the two queries, we observe that the SPREFQL query is smaller (it contains half the number of triple patterns), and it separates the definition of preferences from the hard constraints. This separation alleviates the need for the query author to include in the query body the actual operation that performs the selection of the best solutions, and to express the desired definition of preferences is more clearly. Apart from the advantages in human readability, there exist advantages in machine optimization as well. It would be difficult for a general purpose SPARQL optimizer to find out that in the query in Listing 8 actually implements an operation that resembles a self-join and the result can be computed even in a single pass (as in BNL algorithm).

Listing 8. Rewriting of Listing 1 into standard SPARQL.

```
SELECT ?title ?genre ?runtime
WHERE {
  ?s a :film. ?s :title ?title. ?s :genre ?genre.
  ?s :runtime ?runtime.
  FILTER NOT EXISTS {
    ?s_tmp a :film. ?s_tmp :title ?title1. ?s_tmp :genre ?genre1.
    ?s_tmp :runtime ?runtime1.
    FILTER (?genre1 = ?genre && ?runtime1 > ?runtime)  }
}
```

4 Experiments

4.1 Implementation and experimental setup

This section experimentally validates the idea that optimizations specific to SPREFQL (such as efficient implementations of the winnow operator) can improve the overall query performance in comparison to the equivalent standard SPARQL query and its standard optimizations. As a proof of concept, we provide an open source prototype implementation of SPREFQL.[4] Our implementation is developed in Java within the RDF4J framework,[5] and it includes two implementations of the winnow operator (i.e using NL and BNL algorithms) and a query rewriter which transforms a SPREFQL query into the equivalent SPARQL query, using the `"NOT EXISTS"` transformation. Our evaluator has the ability to operate over a simple memory store using the standard RDF4J evaluation mechanism, or over a remote SPARQL endpoint, in which the query base is executed.

In this experiment we are performing SPREFQL queries on the LinkedMDB database. [6] Our query set contains 7 queries. The queries are: **Q1:** Listing 1, **Q2:** Listing 3, **Q3:** Listing 4, **Q4:** Listing 3 without genre restriction, **Q5:** Listing 4 without genre restriction, **Q6:** Listing 5 and **Q7:** Listing 5 without the FILTER, but for all movies that feature the character 'James Bond', instead.[7] Firstly, we issue the query bases for each SPREFQL query directly on the SPARQL endpoint, and then we evaluate all SPREFQL queries, using i) the NL algorithm, ii) the query rewriting method and iii) the BNL algorithm. The window size for the BNL algorithm was set large enough to contain all results, since we know that BNL behaves better if the preferred result set fits entirely in the window. The experiment was performed on a Linux machine (Ubuntu 14.04 LTS) with a 4-core Intel(R) Xeon(R) CPU E31220 at 3.10GHz and 30 GB RAM. The LinkedMDB dataset was loaded into a locally deployed Virtuoso SPARQL endpoint. [8]

4.2 Results

Table 2 gives the experimental results. We observe that NL has the worst query execution times, and its performance is quadratic in the execution time of the query base. On the first 6 queries, BNL performs better than rewriting. Since BNL was configured so that to perform at its best, the query execution time of BNL is in most cases almost equal to that of the query base. The difference between the execution times of BNL and the query base in Q4 and Q5, can be explained due to the fact that the full result set is larger and BNL has to make more comparisons to calculate the preferred result. The rewrite method in those cases performs much worse than BNL (but much better than NL). In Q7

[4] cf. https://bitbucket.org/dataengineering/sprefql

[5] cf. http://rdf4j.org

[6] cf. http://www.linkedmdb.org

[7] These listings are edited in the paper for conciseness. The exact queries used in the experiment can be found at our code repository, cf. Footnote 4.

[8] Community edition Version 7.1, cf. http://virtuoso.openlinksw.com

though, where an extrinsic preference is expressed, we have a different situation. The comparisons that BNL has to make are not that many (they are at most $23 \cdot 22$), but here BNL has to consider the database each time in order to decide whether one solution is preferred over another. So, BNL issues a heavy load of ASK queries to the endpoint, and therefore rewriting outperforms BNL in Q7. This also explains why BNL has a comparable execution time for Q7 and Q1, although Q1 fetches and considers orders of magnitude more results than Q7. As Q6 also expresses an extrinsic preference, we would expect query rewriting to outperform BNL, but the base result set is very small and the cost to prepare the rewrite is not recuperated. Overall, in our experiments BNL performed better in intrinsic preferences while rewriting performed better in extrinsic preferences.

In the last two queries, we observe that the number of the results that BNL returns is greater than the expected result. This happens because here the preference relation (which is the same for Q6 and Q7) is not a transitive relation (the `:sequel` is not a transitive predicate). This is a known issue of BNL, since BNL returns the correct number of results only on preference relations that impose a *strict partial order* (cf. Section 2). Therefore, in terms of the correctness of the result, rewriting is better than BNL for non strict partial order intrinsic preferences (in extrinsic preferences, rewriting is preferred anyway due to time performance). Checking whether an intrinsic preference expression corresponds to a strict partial order relation is not computationally challenging, as it depends only the size of the expression itself [5, Section 3.1]. In extrinsic expressions, transitivity needs to be confirmed extensionally by issuing `"ASK"` queries.

Regarding the memory footprint of the BNL algorithm, since BNL only maintains the current set of undominated results it is expected to require considerably less space than the base result set. In most cases, the maximum number of results maintained in memory will be close to the final number of results. In our experiments, only Q2 and Q4 required a slight amount of extra space, which can happen when many results that do not dominate each other are received before a result that dominates them.

Table 2. Number of returned results and query execution time (in milliseconds) for NL, query rewriting, and BNL. For BNL, the number of binding sets that need to be maintained in memory is also given, and the total number of bindings in these sets.

	Query Base		NL		rewrite		BNL			
	exec. time	num. res.	exec. time	num. res.	exec. time	num. res.	exec. time	num. bindsets	num. bindings	num. res.
Q1	**556**	6,955	**1,613,515**	36	**4,750**	36	**812**	36	108	36
Q2	**52**	390	**9,124**	5	**188**	5	**65**	6	18	5
Q3	**52**	390	**10,530**	8	**254**	8	**91**	8	24	8
Q4	**872**	9,612	**3,272,789**	8	**197,044**	8	**1,238**	9	27	8
Q5	**872**	9,612	**3,452,048**	108	**193,338**	108	**2,370**	108	324	108
Q6	**135**	4	**794**	1	**296**	1	**170**	2	4	2
Q7	**85**	23	**1,276**	2	**93**	2	**820**	8	16	8

5 Related Work

In the Semantic Web literature there have been proposed SPARQL extensions that feature the expression of preferences [17], typically transferring ideas and results from relational database frameworks much like the work presented here.

When it comes to quantitative preferences, prominent examples include the extensions proposed by Cheng et al. [4] and Magliacane et al. [15]. Closer to our work, influential databases research on *qualitative* preferences includes the work of Kießling [11, 12]. This was used by Siberski et al. [20] to propose a SPARQL extension using a "PREFERRING" solution modifier. Contrary to our approach, these preferences are expressed using unary preference constructors. These constructors are of two types: *boolean preferences* where the preferred elements fulfill a specific boolean condition while the non-preferred do not; and *scoring preferences*, denoted with a "HIGHEST" or "LOWEST" keyword, where the preferred elements have a higher (or lower) value from the non preferred ones. Simple preferences expressed with these constructors can be further combined using Pareto and prioritized composition operators. Gueroussova et al. [9] further extended this language with an "IF-THEN-ELSE" clause which allows expressing *conditional preferences* that apply only if a condition holds. Conditional preferences allow several other 'syntactic sugar' preference constructors to be defined, such as "AROUND" and "BETWEEN".

By comparison, the work presented here is (to the best of our knowledge) the first one to transfer to the Semantic Web the more general framework by Chomicki [5], allowing the expression of extrinsic preferences. Each of the basic preference constructors (boolean, scoring and conditional preferences) as well as the compositions in the approaches by Siberski et al. [20] and Gueroussova et al. [9] can be transformed in SPREFQL. For example, a query of the form

```
SELECT ?s ?o WHERE {?s :p ?o} PREFERRING HIGHEST(?o)
```

can be transformed into SPREFQL:

```
SELECT ?s ?o WHERE {?s :p ?o} PREFER (?s1 ?o1) TO (?s2 ?o2) IF (?o1>?o2)
```

Since in SPREFQL the preference relation is expressed using a binary formula, the reverse translation is not always possible (for example in Listings 1 and 5).

6 Conclusions and Future Work

In this paper we propose SPREFQL, an extension of SPARQL that allows the query author to specify a preference that modifies the query solutions. Although a SPREFQL query can be transformed into standard SPARQL, standard SPARQL query processing misses opportunities to optimize execution by avoiding the exhaustive comparison of all solution pairs. Our experiments demonstrate that when the BNL algorithm is applicable, even for relatively small result sets of under 10k tuples its execution can be two orders of magnitude faster than that of state-of-art SPARQL query processors.

Our first future work direction will be to evaluate the mean gain that can be achieved on realistic workflows. We plan to achieve this by identifying potential test cases where the SPREFQL extensions can be used, so that we can estimate how often the BNL optimization is applicable. This will also help us further develop the language, identifying additional 'syntactic sugar' constructs that can hint at optimizations targeting intransitive relations that fall outside the scope of BNL. Further extensions could allow the client to refer to preferences and preference-related metadata within the knowledge base itself [14, 18, 19].

A more ambitious future extension is to allow the client application to not only request the most preferred results, but to also be able to request all results ordered in different 'layers' of preference. This is a more general solution than any quantitative preference ranking system, as it handles the full generality of partially ordered preferences. We plan to base this on graph-theoretic work in sequencing and scheduling, such as the Coffman-Graham algorithm [6] which is widely used to visualize graphs as layers panning out of a central vertex. By representing arbitrary (including partial-order) preference relations as a directed graph, we can use similar layering approaches to order results in such a way that no dominated tuple is returned before any of the tuples that dominate it.

Acknowledgements

The work described here has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644564. For more details, please visit https://www.big-data-europe.eu

Bibliography

[1] Agrawal, R., Wimmers, E.L.: A framework for expressing and combining preferences. In: Proc. 2000 ACM SIGMOD Intl Conference on Management of Data, Dallas, Texas, USA, 16–18 May 2000. pp. 297–306 (2000)

[2] Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American 284(5), 28–37 (May 2001)

[3] Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering (ICDE 2001), 2–6 April 2001, Heidelberg, Germany. pp. 421–430 (2001)

[4] Cheng, J., Ma, Z.M., Yan, L.: f-SPARQL: A flexible extension of SPARQL. In: Proceedings of the 21st International Conference on Database and Expert Systems Applications (DEXA 2010), Bilbao, Spain, 30 Aug – 3 Sep 2010, Part I. pp. 487–494 (2010)

[5] Chomicki, J.: Preference formulas in relational queries. ACM Trans. Database Syst. 28(4), 427–466 (2003)

[6] Coffman, E.G.J., Graham, R.L.: Optimal scheduling for two-processor systems. Acta Informatica 1 (1972), doi.org/10.1007/bf00288685

[7] Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence 20(2), 308–334 (2004).

[8] Domshlak, C., Hüllermeier, E., Kaci, S., Prade, H.: Preferences in AI: an overview. Artificial Intelligence 175(7–8), 1037–1052 (2011)

[9] Gueroussova, M., Polleres, A., McIlraith, S.A.: SPARQL with qualitative and quantitative preferences. Proceedings of the 2nd International Workshop on Ordering and Reasoning (OrdRing 2013), at ISWC 2013, Sydney, Australia, 22 October 2013. CEUR Workshop Proceedings, vol. 1059 (2013).

[10] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Recommendation, W3C (Mar 2013), https://www.w3.org/TR/sparql11-query

[11] Kießling, W.: Foundations of preferences in database systems. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, 20–23 August 2002. pp. 311–322 (2002)

[12] Kießling, W., Köstler, G.: Preference SQL - design, implementation, experiences. In: Proc. 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, 20–23 August 2002. pp. 990–1001 (2002)

[13] Koutrika, G., Ioannidis, Y.E.: Personalization of queries in database systems. In: Proc. 20th International Conference on Data Engineering (ICDE 2004), Boston, MA, USA, 30 March – 2 April 2004. pp. 597–608 (2004)

[14] Lukasiewicz, T., Martinez, M.V., Simari, G.I.: Preference-based query answering in Datalog+/- ontologies. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), Beijing, China, 3–9 August 2013. pp. 1017–1023 (2013).

[15] Magliacane, S., Bozzon, A., Valle, E.D.: Efficient execution of top-k SPARQL queries. In: Proceedings of ISWC 2012, Boston, MA, USA, 11–15 November 2012. Part I. LNCS, vol. 7649 (2012).

[16] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34(3), 16:1–16:45 (2009), http://doi.acm.org/10.1145/1567274.1567278

[17] Pivert, O., Slama, O., Thion, V.: SPARQL extensions with preferences: a survey. In: Ossowski, S. (ed.) Proc. 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016. pp. 1015–1020. ACM (2016).

[18] Polo, L., Mínguez, I., Berrueta, D., Ruiz, C., Gómez-Pérez, J.M.: User preferences in the web of data. Semantic Web 5(1), 67–75 (2014), http://dx.doi.org/10.3233/SW-2012-0080

[19] Rosati, J., Noia, T.D., Lukasiewicz, T., Leone, R.D., Maurino, A.: Preference queries with ceteris paribus semantics for linked data. In: Proc. On the Move to Meaningful Internet Systems: OTM 2015 Confederated International Conferences, Rhodes, Greece, 26-30 Oct 2015. LNCS, vol. 9415, pp. 423–442. Springer (2015), http://dx.doi.org/10.1007/978-3-319-26148-5_28

[20] Siberski, W., Pan, J.Z., Thaden, U.: Querying the semantic web with preferences. Proceedings of ISWC 2006, Athens, GA, USA, 5–9 November 2006. LNCS, vol. 4273, pp. 612–624. Springer (2006).

[21] Stefanidis, K., Koutrika, G., Pitoura, E.: A survey on representation, composition and application of preferences in database systems. ACM Transactions on Database Systems 36(3), 19:1–19:45 (2011).

[22] Valle, E.D., Schlobach, S., Krötzsch, M., Bozzon, A., Ceri, S., Horrocks, I.: Order matters! Harnessing a world of orderings for reasoning over massive data. Semantic Web 4(2), 219–231 (2013).