PreferenceSPARQL for Querying the Semantic Web

Klaus Emathinger and **Stefan Schödel** and **Markus Endres**[1]

Abstract. As global information production rises steadily, we must investigate new methods of information management. An especially pressing issue therein lies with the increasing discrepancy between production and filter capabilities. Users rightfully expect correct, reliable and relevant answers to their queries. PreferenceSQL can intuitively address these requirements. Meanwhile, the Semantic Web provides a comprehensive collection of standard technologies to realize a machine-readable Web of Data that can help users to retrieve relevant information. By lifting preferences from SQL to SPARQL, the query language of the Semantic Web, we get a new composite called 'PreferenceSPARQL'. With this composite, we seek to harness the relevancy benefits derived from preferences as well as the interoperability advantages of the Semantic Web. This work provides new implementation details for preferences in Apache Jena, specifically its SPARQL engine ARQ and contributes benchmarks that confirm the practicability of the approach.

1 Introduction

The primary audience of the World Wide Web is human users. The data is unstructured, mostly represented as free-form text, and the text organization principles are weak with different kinds of information co-existing. These factors make it unsuitable to machine consumption [21].

Even in the domain of Web services and service-oriented architectures, much human intervention is required to connect one service to another. Be it the interpretation of informal service descriptors, the harmonization of incompatible data schemata or the negotiation of communication protocols [22]. Today's computer systems typically lack the required common-sense reasoning as well as general or business-specific background knowledge [22].

Instead of just serving as a passive display of information, the vision of the Semantic Web is an intelligent system capable of assisting humans in the creation of meaning [23]. Information is modelled, manipulated and queried at the conceptual level [21].

One of the building blocks for the Semantic Web is the Resource Description Framework (RDF), a data model and language for describing web resources. The SPARQL Protocol and RDF Query Language (SPARQL) is the de facto standard for querying RDF data. SPARQL performs graph pattern matching, i.e., it provides capabilities for matching required or optional graph patterns along with their conjunctions and disjunctions. It realizes the filtering of information via hard constraints. Either a binding satisfies a filter expression and is added to the output or it does not, leading to the binding's exclusion from the output. From a machine's perspective, such behaviour is desired. From a human perspective, it is inadequate. A user expects a reasonable amount of results to be delivered. Hence, the query is manually adjusted until a reasonable result set is returned.

To facilitate an efficient and effective exchange of information, the filtering of such information must also evolve. If relevant data is sparse, an intelligent filter will relax its constraints to present the next best options. If relevant data is abundant, only the best results as inferred by some implicit ranking should be returned.

Preferences are a solution for intelligent filtering. Users state their preferences by adding soft constraints which more faithfully reflect the underlying intention of the user [11, 18]. Preferences can be interpreted as personalized wishes in the form of *'I like A more than B'* that are formalized by the strict partial order preference model [12]. Built upon this theoretical framework, PreferenceSQL[2] [14] extends the SQL standard introducing preferences to relational databases. Following a constructor-based approach, preference queries can be formulated intuitively and support a multi-criteria decision by default.

The objective of this paper is the seamless expression of user preferences in the Semantic Web. We developed and evaluated a SPARQL extension called 'PreferenceSPARQL', which supports strict partial order preferences natively, as shown in the example.

Example 1 *Assume we want to purchase an apartment and we already have a specific size in mind (e.g., 75 square metres). More importantly is the price, which is deemed affordable (let's say at most 250 000 Euros). Accordingly, in the listing below a simplified PreferenceSPARQL query is shown expressing our wishes:*

```
prefix : <http://example.com/real-estate/>

select ?sale_offer ?price, ?size
where
{
  ?sale offer a : sale_offer ;
                : price_eur ?price ;
                : size_sqm ?size ;
  prefer (
        ?size around 75 prior to
        ?price lower than 250000)
}
```

Given the extent of PreferenceSQL, this work limits itself to a subset of preference constructs and algorithms, namely: The major base and complex preference constructors presented in [14]. Input data may be partitioned (grouped) along multiple axes, while the preference selection is to be computed for each partition (group) separately. Finally, preference query execution is to be delegated to a varied range of independent preference algorithms: Block-Nested Loop (BNL), Sort and Limit Skyline algorithm (SaLSa), Linear Elimination Sort for Skyline (LESS) and Query Rewriting.

[1] University of Passau, Germany, email: {klaus.emathinger@uni-passau.de, stefan.schoedel@gmx.de, markus.endres@uni-passau.de

[2] PreferenceSQL: http://www.preferencesql.com

The rest of the paper is organized as follows: In Section 2 we discuss some related work. Section 3 describes foundations of preferences and introduces concepts of the Semantic Web, specifically RDF and SPARQL. We also elaborate on Apache Jena that is used for our implementation. In Section 4, we derive our new composite, PreferenceSPARQL, and discuss execution strategies. Section 5 shows the setup and results of our comprehensive experiments. The experiments present a quantitative comparison between four preference algorithms that we chose as execution strategies. In the final Section 6, we summarize the most important aspects of this paper, discuss limitations, and give an outlook for further research.

2 Related Work

An early prototype of preferences in the Semantic Web was done by Siberski et al. [20]. They implemented a preference query formalization of Chomicki [5] into ARQ, the Apache Jena SPARQL engine. They delegated the preference evaluation, which was restricted to HIGHEST (maximum), LOWEST (minimum), Pareto (equal important preferences), and Prioritization (more importance), to BNL. Further limitations of this approach are a lack of notion for regular *Substitutable-Values* semantics (cp. [14]), no partitioning values (without relying on an embedding into the boolean expression) and the absence of any evaluation. In our PreferenceSPARQL approach we solved all these drawbacks and in addition, we extended it with boolean preferences and base preferences as shown in Figure 1.

Pivert et al. [18] published a good theoretical survey on SPARQL extensions with preferences that are classified into quantitative and qualitative ones, but they did neither present an implementation nor experiments. Gueroussova et al. [10] have developed a qualitative approach that adds a preferring graph pattern and rewrites queries to SPARQL. It supports multiple atomic constructors and conditional preferences in the form If E Then P_1 Else P_2. We use an extension of their approach as an alternative execution strategy in our evaluation. Patel-Schneider [17] proposed another extension of SPARQL that can handle simple comparative qualitative preferences.

Preference algorithms have been extensively studied in the context of relational databases and database systems at large. General-purpose algorithms such as BNL [3], Sort Filter Skyline (SFS) [5], SaLSa [1], LESS [8, 9] and Scalagon [6] deliver reasonable performance for most data sets. These algorithms share in common that they can process arbitrary data without prior preparation. More specialized (index-based) algorithms, however, discard this generality for speed, e.g., [15, 16, 19]. For more details, we refer to [4].

3 Background

3.1 Preferences

Preference queries are grounded in the observation that users easily describe their desires in sentences akin to *'I like Y more than X'*. It is only natural that the query engine directly derives the best matches from such preference statements [14]. Formally, we can write $X <_P Y$, where $<_P$ is a strict partial order (SPO). P denotes to a preference on a set of attributes A and is defined as $P := (A, <_P)$, where $<_P \subseteq \text{dom}(A) \times \text{dom}(A)$. Although it seems intuitive, modelling preferences is far from a trivial task as the continued study from a broad spectrum of the academic community shows.

We describe a subset of base and complex constructors developed by [12, 13] that are the foundation of our preference query model. A base constructor operates on a single attribute of either categorical or numerical domain. Figure 1 schematizes the hierarchy of the major base constructors. Each edge indicates a subsumption relationship.

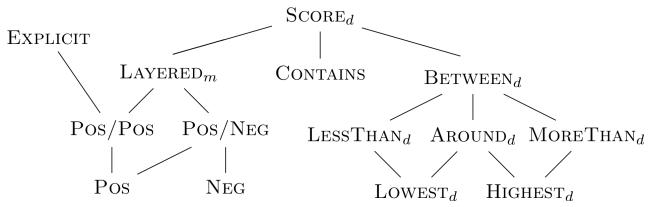

Figure 1. Taxonomy of base preference constructors.

SCORE$_d$ and its sub-constructors are weak order preferences (WOPs), i.e., a strict partial order and negative transitivity. For SCORE$_d$ a scoring function $f : \text{dom}(A) \to \mathbb{R}$ with a discretization factor $d \geq 0$ is used. A preference P is called a SCORE$_d(A, f)$ preference, iff $\forall x, y \in \text{dom}(A)$: $x <_P y \iff f_d(x) > f_d(y)$. The discretization factor d divides continuous values into classes of equal importance.

BETWEEN$_d$, a constructor for the numerical domain stating the desire for a value between a lower and an upper bound. Within the threshold $f(v) = 0$ applies, below $f(v) = low - v$ and above $f(v) = v - up$. Other numerical preferences are a specialization of BETWEEN$_d$. For example by setting low $=$ up, we arrive at AROUND$_d(A, z)$.

Categorical base preferences are sub-constructors of LAYERED$_m$. Let $m \geq 0$ and $L = [L_1, \ldots, L_{m+1}]$ be an ordered list of $m + 1$ disjoint sets of dom(A). Then a LAYERED$_m(A, L)$ preference is a SCORE preference with the subsequent utility function: $f(x) := i - 1 \iff x \in L_i$.

Complex constructors combine multiple preferences into a single preference. For example, a Pareto preference treats two preferences ($P := P_1 \otimes P_2$) equally, whilst a Prioritization ($P := P_1 \& P_2$) ranks them in sequential order. Furthermore, there is a numerical ranking preference RANK$_{F,d}$ that expresses weighted importance between preferences. For more details we refer to [12, 13, 14].

Preference selection performs intelligent filtering by taking the quality of the available data and the stated wishes of the user into account. The result of preference selection is a Best-Matches-Only-set (BMO-set). In PreferenceSQL the discussed constructors are part of the PREFERRING clause. Additionally, there is a GROUPING clause that allows partitioning with an attribute list. P is then evaluated on each partition separately. Lastly, BUT ONLY can be used for filtering in a similar way to WHERE only that it is used after P was evaluated.

3.2 Semantic Web and RDF

By consistently employing the Semantic Web standards like RDF and SPARQL, the web can be transformed into a platform for intelligent data and information exchange. Unstructured textual information like HTML will be replaced by context-aware structured information.

The Resource Description Framework (RDF) represents information via a set of statements which can be visualized as a labelled, directed graph. Each RDF statement consists of a triple in the form $\langle subject, predicate, object \rangle$, where we can distinguish three different types: resources, literals and blank nodes.

A resource is described by a Uniform Resource Identifier (URI) within a global namespace [24]. Literals encode necessary information like numbers, dates and strings. While blank nodes provide a way of introducing resources without explicitly naming them.

The statements can be serialized into a number of formats. Historically, RDF is associated with XML. Other formats such as Turtle or JSON-LD have been developed to be more human-readable and less complicated. We obtain an RDF graph by combining multiple triples as shown in the next example.

Example 2 *Figure 2 shows a small RDF graph describing the chemical element 'Titanium'. Each edge starts at the subject, is labelled with the predicate and points towards the object. A rectangle is used for literals and an oval for resources. It is visible that the object of one triple can be the subject of another triple. It proposes a graph-structural data model. We obtain an RDF data set by collecting several such graphs. The goal of RDF is to connect heterogeneously structured data from multiple sources [7].*

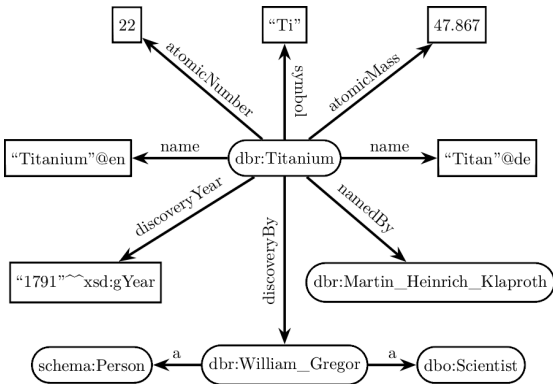

Figure 2. Sample RDF graph of the chemical element Titanium.

The SPARQL Protocol and RDF Query Language (SPARQL) is used for querying RDF data. It supports entailment regimes for RDF Schema, Web Ontology Language (OWL), and others. Ontological information can be used to consider implicit triples during query execution.

A SPARQL SELECT query usually starts with prefix and base statements. These are responsible for declaring URI abbreviations and defining a base for relative URIs, respectively. A SELECT clause follows after that and defines a list of variables that is projected.

The WHERE clause defines the query graph pattern. It can contain several basic graph patterns (triples) that can be combined with conjunctions and disjunctions. Another part of the WHERE clause is filters that restrict the solution sequence S to those solutions that satisfy a constraint R, written as S FILTER R.

Similar to SQL, SPARQL also supports grouping via GROUP BY, sorting via ORDER BY and LIMIT to restrict the result size. There are even more clauses, but only SELECT and WHERE are mandatory (the WHERE keyword may be elided).

3.3 Apache Jena

For our implementation, we have chosen Apache Jena and its SPARQL query engine ARQ. Apache Jena is a free and open-source Java framework for building Semantic Web and Linked Data applications. Due to its modular design and focus on extensibility ARQ simplifies integration greatly. Jena provides extensive library support by following published W3C recommendations. Beyond, it includes a fully-fledged SPARQL processor called 'ARQ', a native triple store called 'TDB' and a SPARQL end-point for exposing RDF datasets called 'Fuseki'. At its core, Jena stores information as RDF triples. It provides facilities for importing, exporting, storing, transforming, querying and publishing information mentioned above. Jena realizes these varied functionalities through a number of major subsystems, separated by clearly defined interfaces.

ARQ is a SPARQL 1.1 compliant engine. It supports remote federated queries, free text search via Lucene and a high degree of customization. Furthermore, Jena API is integrated into ARQ and can be utilized with different storage backends.

Figure 3 illustrates the basic query execution flow. A SPARQL query submitted by a user is first parsed by the SPARQL grammar. It generates an Abstract Syntax Tree (AST). Next, the AST is compiled into SPARQL algebra as described by the SPARQL specification. ARQ then optimizes the algebra via high-level algebraic transformations. This includes a rewriting of the algebra into new, equivalent algebra forms and introducing specialized algebra operators . The algebra is expressed as a SPARQL S-Expression (SSE), a custom syntax for stating SPARQL algebra in a concise format. Subsequently, a query plan is computed. This query plan is finally executed to get a solution sequence[3].

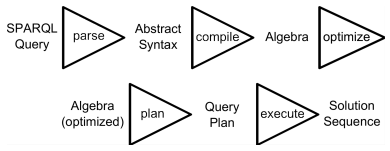

Figure 3. ARQ Query Execution Flow.

4 PreferenceSPARQL

4.1 Query Language

The goal of PreferenceSPARQL is to incorporate the advances made with PreferenceSQL into SPARQL. Given the extent of PreferenceSQL, this work has limited itself to a subset of preference constructs and algorithms. It includes the major base and complex preference constructors described above. Furthermore, a partitioning mechanism similar to GROUPING called 'PARTITION' has been realized.

SPARQL is a language for operating on graphs primarily. Thus, in order to be a first-class citizen of the SPARQL, the extension point should be a graph pattern as well. The SPARQL Filter is a prime candidate for imitation. An alternative approach would be the implementation of a solution modifier. This option, however, comes with the drawback of needlessly complicating queries with nesting (subqueries), if multiple independent preference statements are desired, e.g., for human readability. Given the preceding rationale, we have imitated the SPARQL Filter, i.e., the SPARQL grammar has been extended with a PREFER clause on the following rule:

```
GraphPatternNotTriples :==
  | GroupOrUnionGraphPattern
  | OptionalGraphPattern | MinusGraphPattern
  | GraphGraphPattern | ServiceGraphPattern
  | Filter | Bind | InlineData
  | Prefer
```

In order to define PREFER and PARTITION, let P be a graph pattern, $Pref$ be an inductively constructed preference and G be a set of variables. Then P PREFER $Pref$ PARTITION G is a graph pattern. We write P PREFER $Pref$ as shorthand for partitioning by an empty set G. PREFER and PARTITION realize PreferenceSQL's PREFERRING and GROUPING, respectively. A BUT ONLY can be emulated via a subsequent FILTER.

[3] Apache Jena documentation: https://jena.apache.org/documentation/

The set of attributes A is now a set of variables V_A. In SPARQL a solution mapping is a partial function $\mu : V \rightarrow T$. Consequently, a variable might be undefined, i.e., have no associated mapping. We augment the scoring function to consider this case explicitly by treating undefined variables in a solution mapping as the worst possible value, i.e., $f(x) := \infty$ if x is undefined. Given these prerequisites, the semantics of the prefer graph pattern can now be defined.

Definition 1 (Prefer and Partition) *Let Pref $= (V_A, <_P)$ be a preference, G be a set of variables and Ω be a solution sequence of mappings $\mu : V \rightarrow T$. Then the preference selection operator is defined as:*

$$\sigma[Pref](\Omega) := \\ [\mu \mid \mu \in \Omega \wedge \nexists \mu' \in \Omega : \mu[V_A] <_{Pref} \mu'[V_A]]$$

With PARTITION only the best solutions remain in each group

$$\sigma[Pref \text{ PARTITION } G](\Omega) := \\ [\mu \mid \mu \in \Omega \wedge \nexists \mu' \in \Omega : \mu[G] = \mu'[G] \wedge \mu[V_A] <_{Pref} \mu'[V_A]]$$

Overall, the syntax is closely related to PreferenceSQL, except for a key alteration w.r.t. categorical preferences. Due to a grammar conflict on the keyword IN, a break with prior convention is required. The new keyword is ONE OF.

For convenience, PreferenceSPARQL supports the usage of SPARQL expressions in preferences. For instance, $?x+?y$ AROUND 0 is a valid preference which is equivalent to BIND ($?x+?y$ As $?z$) . $?z$ AROUND 0. Caution needs to be taken when using an expression as a goal value (e.g., low, up, d). The behaviour will be unpredictable, unless a constant value is used.

Table 1 lists all available base constructors and Table 2 shows the corresponding grammar.

Table 1. Base Constructors in PreferenceSPARQL.

Preference Constructor	SPARQL Equivalent
Between$_d(V_x, [low, up])$	?x BETWEEN low, up, d
Around$_d(V_x, z)$	?x AROUND z, d
MoreThan$_d(V_x, low)$	?x MORE THAN low, d
LessThan$_d(V_x, up)$	?x LESS THAN up, d
Highest(V_x)	?x HIGHEST
Lowest(V_x)	?x HIGHEST
Layered$_m(V_x, [S_{1\rightarrow k}, others, S_{k+1\rightarrow m}])$	?x LAYERED (S_1, ..., others, ..., S_m)
PosPos(V_x, S_1, S_2)	?x ONE OF S_1 ELSE S_2
PosNeg(V_x, S_1, S_2)	?x ONE OF S_1 NONE OF S_2
Pos(V_x, S)	?x ONE OF S
Neg(V_x, S)	?x NONE OF S
$S = \{s_1, \ldots, s_n\}$	$(s_1, \ldots, s_n)$

4.2 Query Execution

Recall the ARQ execution flow and its necessary steps (Figure 3). The query parser that creates an AST has been extended with the preference constructors described above. The parser will only recognize preferences if the syntax option is specified as 'PrefSPARQL'. ARQ validates the extended AST by generating valid SPARQL syntax and re-parsing the generated output. This requires the implementation of a corresponding formatter for preferences.

The AST and its elements are then compiled into an algebraic representation. The algebra generator uses Pareto to put multiple PREFER clauses inside a single basic graph pattern (BGP). This approach

Table 2. PreferenceSPARQL Grammar.

Prefer	:=	'PREFER' BracketedPrefer PartitionClause?
BracketedPrefer	:=	'(' ParetoPreference ')'
PartitionClause	:=	'PARTITION' '(' Var$^+$ ')'
ParetoPreference	:=	Prioritization ('AND' Prioritization)*
Prioritization	:=	PrefAtom ('PRIOR' 'TO' PrefAtom)*
PrefAtom	:=	BracketedPrefer | (Expression (DiscreteAtom | ContinuousAtom))
DiscreteAtom	:=	Layered | Pos | Neg
Layered	:=	"LAYERED" '(' (ListOfSets ',')? 'others' (',' ListOfSets)? ')'
Pos	:=	'ONE' 'OF' Set (('ELSE' Set) | ('NONE' 'OF' Set))?
Neg	:=	'NONE' 'OF' Set
ListOfSets	:=	Set (',' Set)*
Set	:=	'(' Expression (',' Expression)* ')'
ContinuousAtom	:=	Interval | Around | Highest | Lowest
Interval	:=	(('BETWEEN' Expression ',' Expression) | ('MORE' 'THAN' Expression) | ('LESS' 'THAN' Expression)) (',' Expression)?
Around	:=	'AROUND' Expression (',' Expression)?
Highest	:=	'HIGHEST'
Lowest	:=	'LOWEST'

is similar to the behaviour specified by the SPARQL standard for Filter. ARQ subsequently validates the algebra, i.e., it is transformed to SSE. A corresponding SSE parser and formatter for preferences have been realized.

The algebra is thereafter optimized, and a query plan is derived from the optimized structure. ARQ supports two query engines. A reference engine is intended for ascertaining correctness, while the main engine is intended for production deployments. Both engines have been extended to support preferences.

Execution of a P PREFER $Pref$ PARTITION G graph pattern is realized by first partitioning the solution sequence of P according to G. For each partition, the requested preference algorithm is called separately. The computed solution sequences $\Omega_1, \ldots, \Omega_n$ are subsequently concatenated ($\Sigma_{i=1}^{n} \Omega_i$) in order to arrive at the final solution sequence.

4.3 Preference Algorithms

In this section we present some preference algorithms which we modified and adapted for our PreferenceSPARQL experiments.

BNL was first presented by Börzsönyi et al. [3] and is an improvement of the nested-loop algorithm. Instead of comparing every tuple with each other, it reduces the number of comparisons by only considering tuples within a buffer (window) and immediately discarding dominated tuples. Since no assumption about the preference or the data is made, the algorithm is easy to implement. We use an in-memory variant (BNL-M) and an external variant (BNL-E) that is bounded by a certain window size. A major drawback of BNL in its current incarnation is the need to evaluate all expressions and their associated scoring functions repeatedly. When comparing $p <_{Pref} q$ followed by $p <_{Pref} w$, it would be desirable to re-use at least some intermediary scoring results of p. Partially caching f_d, in general, might lead to significant performance improvements. However, the implementation of such a caching strategy is not trivial due to the presence of Pareto and Prioritization, which obfuscate the underlying utility functions.

LESS is based on Sort Filter Skyline (SFS) that modifies BNL by first sorting the input according to a monotone function $\mathcal{F}$. SFS's approach greatly simplifies window management since an unread tuple p cannot dominate a tuple q that has already been placed within the window S (due to $\mathcal{F}(q) \geq \mathcal{F}(p)$) [1]. If $\mathcal{F}$ is suitably chosen, then preferences can be computed without comparing all tuples. LESS makes two refinements. Firstly, it adds an elimination-filter (EF) window at the beginning of the sorting phase, designed to reduce tuples early. Secondly, the final pass of the sort is combined with the first skyline-filter (SF) pass.

SaLSa is also based on SFS and was first introduced by Bartolini et al. [1]. It further exploits the benefits of topological sorting by using a stopping point. This point guarantees that unread tuples are dominated by already seen tuples and can, therefore, be discarded.

The last approach is **Query Rewriting**. It transforms PreferenceSPARQL into plain SPARQL queries during the high-level optimization phase of ARQ. Modifications during the execution phase are not necessary. Subsequently, a rewritten query can be sent to any regular SPARQL engine accessible on the Web.

5 Experiments

The primary goal of the experiments is to evaluate the preference algorithms for PreferenceSPARQL. However, the evaluation is constrained towards a quantitative approach, i.e., a qualitative evaluation does not take place. Furthermore, we use PreferenceSQL as a reference to better assess the practicability of our implementation.

5.1 Data Set

Preferences are especially useful in the domain of real estate purchases and sales. Enriching a carefully curated data set with factual information from outside sources offers multiple benefits to the customer experience. For instance, information regarding the nearest city or the neighbourhood could be included. Since e-commerce data is commonly not publicly available, we decided to generate a synthetic data set derived from real-world facts.

The data set is scalable and comes in two distinct representations. One uses RDF triple data that is intended for our PreferenceSPARQL client. The other one is for PreferenceSQL in the form of a relational data model. The relational data model consists of 9 coherent tables. One table, e.g., describes an agent who is responsible for a collection of sale offers. Correspondingly, the listing below shows some sample RDF triples. Other tables have information about the property, location, usage type of the land or internet availability. The general structure of the test suite is inspired by the well-known Berlin SPARQL Benchmark [2].

```
<uri/agent/12> a <uri/agent>;
  <uri/company_name> "Quast Immo GmbH";
  <uri/first_name> "Orlando";
  <uri/last_name> "Asmus" .

<uri/sale_offer/5> <uri/agent>
    <uri/agent/12> .

<uri/sale_offer/34> <uri/agent>
    <uri/agent/12> .

<uri/sale_offer/5> a <uri/sale_offer>;
  <uri/commission> 7.14E0;
  <uri/price_eur> 5988456 .

(...)
```

5.2 Test Queries

We came up with 20 queries that reflect the differences in preference usage as appropriate for each use case. Table 3 in Appendix A describes all queries and contains a rough estimation of the complexity. Due to limited space, we selected three queries out of the 20 to illustrate our results. The test queries reflect different performance profiles. All queries are parameterized (@parameter@), in order to prevent caching.

We use Query 5 (cp. Table 3) as an example that we explain in detail. An investor could use this query to look for properties that have a stable renter base, i.e., renters who pay on time. Equally important might be that the rental conditions compare favourably to market, i.e., the net rental return is higher than average. In order to retrieve all the necessary information, we need to join three tables in PreferenceSQL. The preferring clause consists of two base constructors (MORE THAN, LAYERED) and one complex constructor (AND) to combine them. Therefore, the dimension in Table 3 was set to 2.

```
select (..)
from sale_offer s, property p, contract c
where p.id = s.property_id and c.id = p.contract_id
preferring c.net_rental_return more than @net@
        and c.payment_behavior
        layered(('Punctual'), ('Unknown'), others));
```

Below we show the corresponding PreferenceSPARQL query. The query is more verbose, but the prefer clause is very similar to the clause in PreferenceSQL.

```
prefix: <http://example.com/real-estate/>
prefix xsd: (..)

select ?sale_offer ?price_eur (..)
where {
  ?sale_offer a :sale_offer ;
                :price_eur ?price_eur ;
                :commission ?commission ;
                :property ?property .
  ?property :contract ?contract .
  ?contract :net_rental_return ?net_ren(..) ;
            :payment_behavior ?payment_(..) .
  prefer( ?net_rental_return more than @net@
        and ?payment_behavior
        layered (('Punctual'), ('Unknown'), others))
}
```

5.3 Experimental Setting

All experiments have been conducted on a standard PC (Intel i7-4770K 3.9 GHz CPU, 16 GB RAM, Windows 7 x64). The JVM has been assigned 10 GB RAM (-d64 -Xms10G-Xmx10G). The backend database for PreferenceSQL is PostgreSQL 9.4, which has been deployed to the same computer with factory defaults. The PreferenceSPARQL implementation is built upon Apache Jena 3.7.0 with default settings. The spill factor is set to 100000 for externalized BNL.

We utilize TDB2 as our native triple store. Memory is bounded towards JVM-assigned RAM. Hence, performance estimates are optimistic. For a fair comparison to PreferenceSQL, TDB would have to be limited to the same amount of memory.

A test driver dispatches preference queries to the intended recipient and collects benchmark metrics for further processing. Each query stream repeatedly executes a random permutation of the test queries with uniformly distributed parameters. Once execution concludes, measurements are aggregated in a corresponding data structure. A certain number of warm-up runs are discarded. All our performance metrics have been derived from execution time. Specifically, we measure the following key indicators:

- *Query Execution Time (QET)* which denotes the time required to serve a query request (from dispatch till the reception of all solutions).
- *Aggregated Execution Time (AET)* which indicates the run time of all 20 individual queries by summation of their QET. Downtime spent by the test driver to prepare the next query is not measured.
- *Thread Execution Time (TET)* is the sum of all AETs of an algorithm for one query stream.

For statistical robustness, we use box-and-whisker diagrams. QETs and AETs are broken down according to lower quartile, median and upper quartile as indicated by the box. Outliers are depicted by individual points.

5.4 Results

Query Execution Time by Algorithm (QET) Figure 4 shows the query execution time of all algorithms. At the top is Query Rewriting (including query translation time), then LESS, SaLSa, the externalized version of BNL (BNL-E), the in-memory version of BNL (BNL-M) and finally PreferenceSQL.

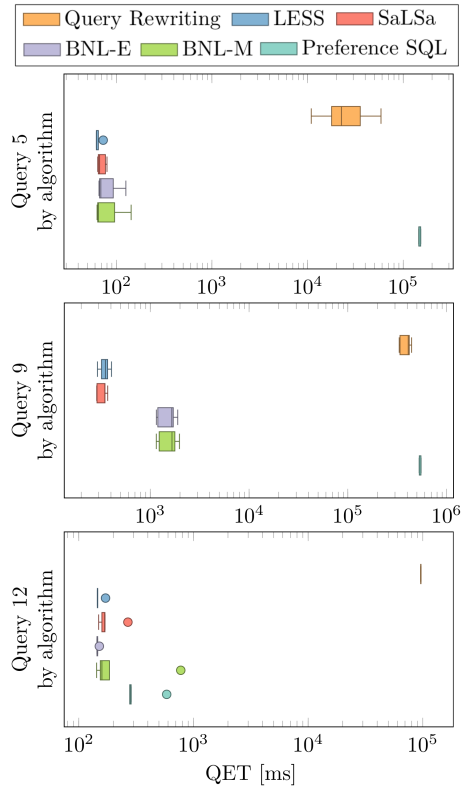

Figure 4. Algorithm comparison by single queries (100 agents, 5 iterations, single client)

The data set refers to 100 agents that correspond to 297 834 RDF triples. Query 5 was already discussed above and has low complexity. The other two have high complexity. Query 9 uses multiple Pareto preferences and Query 12 four Prioritizations and a nested quaternary Pareto preference (cp. Table 3).

It is apparent that the performance of Query Rewriting is subpar. ARQ is not optimized for complex filter expressions. Hence, Query Rewriting should only be employed if no other algorithm can be used, e.g., due to a proprietary SPARQL engine.

The performance of PreferenceSQL is mixed. On the one hand, it closely mirrors all native preference algorithms (Query 12). On the other hand, some queries are exceedingly slow (Query 5, Query 9). Multiple factors probably cause a performance bottleneck. For once, the PreferenceSQL settings are very conservative in comparison to the ARQ implementation, which tries to keep the whole solution sequence in memory if feasible. Furthermore, the middleware architecture inherently limits PreferenceSQL when a preference is applied to the whole data set without any further restrictions. Fetching the whole data set from the database incurs a non-negligible overhead which cannot be adequately compensated by better preference algorithms and cost-estimation.

Figure 5 focuses only on native preference algorithms. The data set was scaled up to 5 000 agents or 14 700 102 RDF triples. The performance of in-memory BNL is solid, mostly outperforming its externalized variant in simple queries (Query 5). In some circumstances, BNL is significantly slower than other preference algorithms, e.g. in Query 9, with multiple Pareto preferences. Choosing the correct window size is difficult, though, due to observed performance reductions with overly large window sizes.

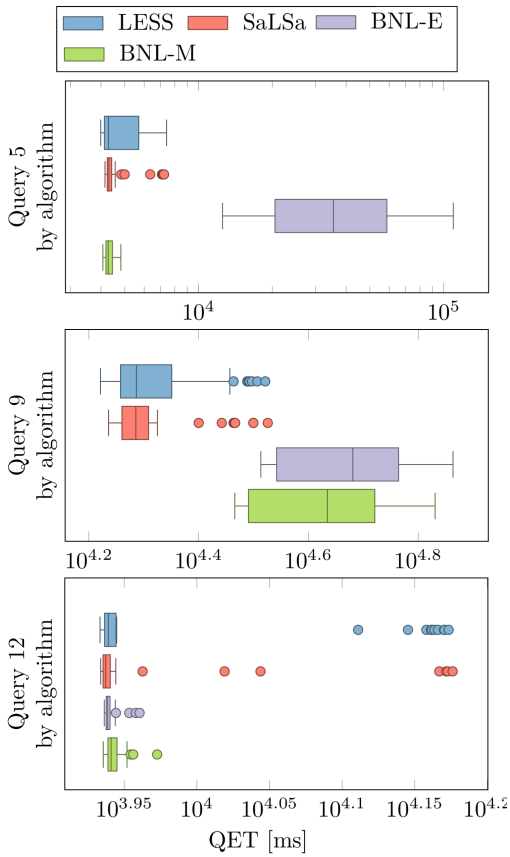

Figure 5. Algorithm comparison by single queries (5000 agents, 50 iterations, single client)

Across all queries, LESS leads in terms of execution time, but the expected performance gain over BNL is absent. The spread in execution time for most queries is very apparent. SaLSa appears to suffer from the different choices made w.r.t. evaluation of the tuples on larger data sets (Query 10, 14, 16). Replacing the evaluation code with the one used for LESS might increase performance in SaLSa's

favour. Notably, LESS and SaLSa produce quite a lot of outliers. We presume this is due to the memory limitation of 10 GB, repeatedly triggering garbage collection.

Aggregated Execution Time by Algorithm (AET) Figure 6 aggregates the QETs for all test queries. For 100 agents, the distribution of the AETs affirms our prior judgment that Query Rewriting is not suitable for larger workloads. The previously discussed non-performing queries skew the AET for PreferenceSQL. The AET for the well-performing query subset of PreferenceSQL is approximately equivalent to the native PreferenceSPARQL algorithms. For 5 000 agents, in-memory BNL outperforms its externalized cousin. SaLSa is roughly equivalent to in-memory BNL by median but gives rise to lots of outliers in both directions. Finally, LESS outperforms everyone else by the median, but just like SaLSa suffers from a large spread in execution times.

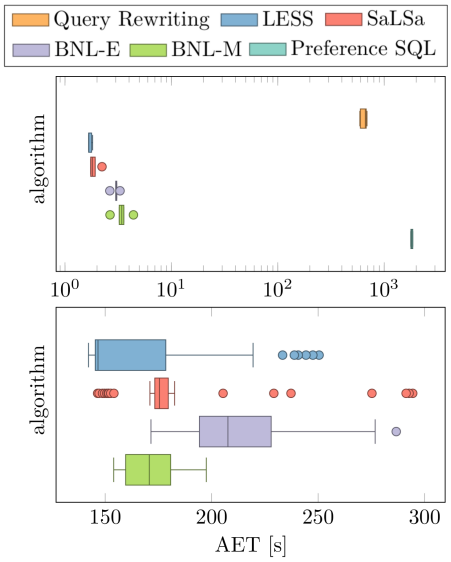

Figure 6. Algorithm comparison by aggregated execution time (Top: 100 agents, 5 iterations. Bottom: 5000 agents, 50 iterations.)

Thread Execution Time by Algorithm (TET) The evaluated data set sizes range from 100 to 5 000 liaisons that are 97 834 to 14 700 102 triples, respectively. **Figure 7** shows the overall run time of the native PreferenceSPARQL algorithms across these data set sizes. From 100 to 1000 agents, the performance of LESS is excellent. SaLSa is closely behind LESS. Externalized BNL is the slowest method, but not far behind its in-memory counterpart. The gap between in-memory LESS and BNL rapidly closes as the size of the data set further increases. With 500 agents LESS only spends 58.63 % of BNL-M's run time. With 5 000 agents LESS already requires 97.19 %. Given the memory usage pattern observed during the evaluation, ARQ appears to run into the 10 GB memory limit. With a machine that has more memory, LESS should lead significantly in performance. An externalization for LESS and SaLSa should alleviate this problem in general.

6 Summary and Conclusion

After introducing the Semantic Web in general and SPARQL in particular, we have presented the concept of preferences as personalized wishes. We have adopted the formalization by Kießling et al., modelling such preferences as strict partial orders via an inductive

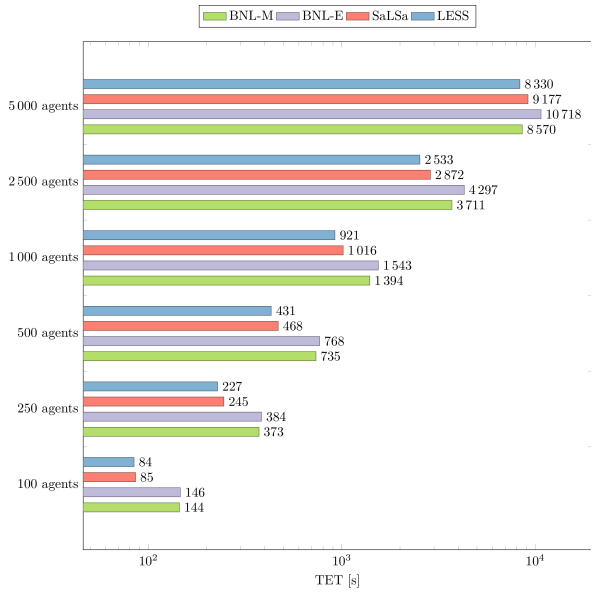

Figure 7. Algorithm comparison by overall run time across various data set sizes (50 iterations, single client)

constructor-based technique. Inspired by PreferenceSQL, we have defined a replica for SPARQL called PreferenceSPARQL. Given the extent of PreferenceSQL, our current realization has been constrained towards the major base and complex constructors. In terms of execution strategies, we presently support BNL, SaLSa, LESS and Query Rewriting for preference selection. Finally, we contribute a benchmark suite built with preferences in mind. Situated in the domain of real estate purchases and sales, the Real Estate Benchmark allows for the evaluation of SQL and SPARQL preference queries upon a dataset derived from real-world facts.

Our evaluation shows that overall, the in-memory LESS implementation leads in terms of execution time, but the expected performance gain over BNL is absent. SaLSa and LESS produce many outliers on larger data sets that might be caused by memory limitations. Query Rewriting has the worst performance. ARQ is not optimized to handle complex filter expressions. Lastly, we compared PreferenceSPARQL and an externalized implementation of PreferenceSQL. The results show that PreferenceSPARQL is superior to PreferenceSQL, but the comparison is skewed. PreferenceSQL should be re-evaluated with in-memory settings to get fair results. An evaluation of the qualitative aspect is considered future work.

A problem is that most of the Semantic Web currently consists of descriptive repositories of facts. Governments and scientific institutions have begun to release their data sets to the public and crowd-sourced efforts like DBpedia have extracted structured information from the Wikipedia project. In the future, job-searching might be a promising domain for PreferenceSPARQL. Since Google launched its job search engine, more and more companies and job platforms provide structured data for their job offers.

Regarding PreferenceSPARQL much remains to be done. Firstly, the remaining base and complex constructors defined by PreferenceSQL should be implemented. Secondly, more sophisticated evaluation strategies could be developed, either by realizing new algorithms from the literature (e.g., index-based) or by optimizing existing binding evaluation. Furthermore, the consideration of ontological knowledge or the formulation of preferences on the actual structure of the RDF graph might speed up preference evaluation significantly.

REFERENCES

[1] I. Bartolini, P. Ciaccia, and M. Patella, 'SaLSa: Computing the Skyline Without Scanning the Whole Sky', in *Proceedings of CIKM '06*, pp. 405–414. ACM, (2006).

[2] C. Bizer and A. Schultz. Berlin SPARQL Benchmark (BSBM) Specification - V3.1. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/, 2011.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker, 'The Skyline Operator', in *Proceedings of ICDE '01*, pp. 421–430. IEEE, (2001).

[4] J. Chomicki, P. Ciaccia, and N. Meneghetti, 'Skyline Queries, Front and Back', *ACM SIGMOD Record*, **42**(3), 6–18, (2013).

[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, 'Skyline with Presorting', in *Proceedings of ICDE '03*, pp. 717–816. IEEE, (2003).

[6] M. Endres, P. Roocks, and W. Kießling, 'Scalagon: An Efficient Skyline Algorithm for all Seasons', in *Proceedings of DASFAA '15*, eds., M. Renz et al., pp. 292–308. Springer International, (2015).

[7] B. Glimm, 'Using SPARQL with RDFS and OWL Entailment', in *Reasoning Web. Semantic Technologies for the Web of Data*, eds., A. Polleres et al., 137–201, Springer Berlin Heidelberg, (2011).

[8] P. Godfrey, R. Shipley, and J. Gryz, 'Maximal Vector Computation in Large Data Sets', in *Proceedings of VLDB '05*, pp. 229–240. VLDB Endowment, (2005).

[9] P. Godfrey, R. Shipley, and J. Gryz, 'Algorithms and Analyses for Maximal Vector Computation', *The VLDB Journal*, **16**, 5–28, (2007).

[10] M. Gueroussova, A. Polleres, and S.A. McIlraith, 'SPARQL with Qualitative and Quantitative Preferences', in *Proceedings of OrdRing '13*, pp. 2–8, Aachen, Germany, (2013). CEUR-WS.org.

[11] A. Hadjali, S. Kaci, and H. Prade, 'Database Preference Queries - A Possibilistic Logic Approach with Symbolic Priorities', in *Foundations of Information and Knowledge Systems*, eds., S. Hartmann and G. Kern-Isberner, volume 63, pp. 291–310. Springer Berlin Heidelberg, (2008).

[12] W. Kießling, 'Foundations of Preferences in Database Systems', in *Procedings of VLDB '02*, pp. 311–322. VLDB Endowment, (2002).

[13] W. Kießling, 'Preference Queries with SV-Semantics', in *COMAD '05: Advances in Data Management 2005, Proceedings of the 11th International Conference on Management of Data*, eds., Jayant R. Haritsa and T. M. Vijayaraman, pp. 15–26, Goa, India, (2005). Computer Society of India.

[14] W. Kießling, M. Endres, and F. Wenzel, 'The Preference SQL System - An Overview', *Bulletin of the Technical Commitee on Data Engineering, IEEE Computer Society*, **34**(2), 11–18, (2011).

[15] D. Kossmann, F. Ramsak, and S. Rost, 'Shooting Stars in the Sky: An Online Algorithm for Skyline Queries', in *Proceedings of VLDB '02*, pp. 275–286. VLDB Endowment, (2002).

[16] D. Papadias, Y. Tao, G. Fu, and B. Seeger, 'An Optimal and Progressive Algorithm for Skyline Queries', in *Proceedings of SIGMOD '03*, pp. 467–478. ACM, (2003).

[17] P. F. Patel-Schneider, A. Polleres, and D. Martin, 'Comparative Preferences in SPARQL', in *21st International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, volume 11313 of *Lecture Notes in Computer Science*, pp. 289–305. Springer, (2018).

[18] O. Pivert, O. Slama, and V. Thion, 'SPARQL Extensions with Preferences: A Survey', in *Proceedings of SAC '16*, pp. 1015–1020. ACM, (2016).

[19] J. Selke and W.T. Balke, 'SkyMap: A Trie-Based Index Structure for High-Performance Skyline Query Processing', in *Database and Expert Systems Applications*, eds., A. Hameurlain et al., pp. 350–365. Springer Berlin Heidelberg, (2011).

[20] W. Siberski, J.Z. Pan, and U. Thaden, 'Querying the Semantic Web with Preferences', in *The Semantic Web - ISWC 2006*, eds., I. Cruz et al., pp. 612–624. Springer Berlin Heidelberg, (2006).

[21] H. Stuckenschmidt, F. Harmelen, W. Siberski, and S. Staab, 'Peer-to-Peer and Semantic Web', in *Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information*, eds., S. Staab and H. Stuckenschmidt, 1–17, Springer Berlin Heidelberg, (2006).

[22] R. Studer, A. Abecker, and S. Grimm, 'Introduction', in *Semantic Web Services: Concepts, Technologies, and Applications*, 1–11, Springer Berlin Heidelberg, (2007).

[23] M. Workman, 'Introduction', in *Semantic Web: Implications for Technologies and Business Practices*, Springer International, (2016).

[24] L. Yu, 'The Building Block for the Semantic Web: RDF', in *A Developer's Guide to the Semantic Web*, 23–95, Springer Berlin Heidelberg, (2014).

A Test Queries

Table 3. Queries with dimensions and short descriptions. The dimensionality roughly indicates the complexity of the preference, e.g., '2/4' denotes a binary Pareto prioritized over a quaternary Pareto.

Query	Dimensions	Preference
01	2	Lots with the lowest price and enough space for the construction of a single-family house.
02	3	Properties with highest guide value, most residential units and most recently modernized.
03	2/4	Properties with certain target size and a certain number of residential units. Less importantly, the price, energy consumption and two other categorical characteristics.
04	1	Lots around a specific area.
05	2	Properties with an above-average net return and punctual rental payments.
06	2/2	Properties with a certain size and amount of residential units. Less importantly, price and the year of the last modernization.
07	3	Agricultural land with three specific numerical characteristics.
08	4	Properties with specific categorical characteristics regarding the construction.
09	14	Properties with eight specific categorical and six numerical characteristics regarding general facts.
10	4/2	Properties with four specific characteristics regarding the neighbourhood. Less importantly, are two specific numerical characteristics.
11	5/1	Agents with four specific sale characteristics before the lowest possible commission.
12	1/1/4/1	Municipalities with a range of different and equally important numerical characteristics.
13	2	Municipalities with a low photovoltaics adoption rate and high yearly returns per square meter.
14	1/7/1	Properties built before 1970. Less importantly, seven specific characteristics regarding the interior and, least importantly, the number of amenities.
15	1/3	Warehouses that exceed a certain capacity before three specific numerical preferences.
16	6	Properties with six characteristics that are similar to another property's characteristics.
17	2 partition 2	Lots with the highest internet upload and download rate for each municipality and internet type.
18	3/2	Properties with three certain electrical requirements. Less importantly, the construction year and the condition of the building.
19	1/1/1/1/1	Properties with five specific characteristics that have all different importance.
20	1 partition 1	Sale offers with a certain price compared to their market value grouped by the municipality.

Query 12

```
select m.*
from query12_rent_markets m
preferring m.sale_count more than 15 prior to m.net_
    return_avg more than 0.03 prior to
(m.sale_count highest and m.lot_gv_avg highest and m
    .property_gv_avg highest and m.unit_avg highest)
     prior to
m.net_return_avg highest;
```