Preference-driven Control over Incompleteness of Knowledge Graph Query Answers

Till Affeldt
t.affeldt@tu-bs.de
TU Braunschweig
Institute for Information Systems
Braunschweig, Germany

Stephan Mennicke
mennicke@ifis.cs.tu-bs.de
TU Braunschweig
Institute for Information Systems
Braunschweig, Germany

Wolf-Tilo Balke
balke@ifis.cs.tu-bs.de
TU Braunschweig
Institute for Information Systems
Braunschweig, Germany

ABSTRACT

Entities in today's knowledge graphs do not only differ in their property values but also in the schematic structures they are represented by. Given their extraction-based foundation, it is quite common that in practical knowledge base instances totally unrelated graph structures describe entities of the same type. Hence, operators for handling such heterogeneity are mandatory when designing a robust query language for knowledge graphs. While SPARQL does offer optional patterns for this purpose, their query answers often suffer from an unintuitive matching behavior. In contrast, preference semantics seem to be a much more intuitive and still robust way of expressing how the optimal query result may look like. While preferences over data value domains are already applied for graph data, we argue for structural preferences to achieve fine-grained control of heterogeneity in the query answers. Therefore, we propose a new operator for SPARQL, enabling the expression of structural as well as some value preferences. Equipped with a Pareto-style semantics, we give examples of how to model preferences with the new operator. Our prototypical implementation allows for evaluating several encodings of the new construct at DBpedia's SPARQL endpoint.

CCS CONCEPTS

• **Information systems** → **Query operators**; Query reformulation; *Query intent*; **Retrieval models and ranking**; Resource Description Framework (RDF); *Query languages for non-relational engines*.

KEYWORDS

knowledge graphs, incompleteness, preference queries, SPARQL

ACM Reference Format:
Till Affeldt, Stephan Mennicke, and Wolf-Tilo Balke. 2020. Preference-driven Control over Incompleteness of Knowledge Graph Query Answers. In *12th ACM Conference on Web Science (WebSci '20), July 6–10, 2020, Southampton, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394231.3397911

WebSci '20, July 6–10, 2020, Southampton, United Kingdom

ACM ISBN 978-1-4503-7989-2/20/07…$15.00
https://doi.org/10.1145/3394231.3397911

1 INTRODUCTION

One of the major difficulties in working with today's knowledge graphs is that entities differ in the schematic structures they are represented by. During extraction, it is quite common that in practical knowledge base instances we have two entities (represented as nodes or resources) that share a type, e. g., scientists, but are characterized in totally different ways regarding their properties. Furthermore, extraction may even miss important aspects of an entity a user later expects in the knowledge graph. Upon querying by basic graph patterns (conjunctive queries), a user is most likely disappointed because too few or even no results are returned. Hence, operators for handling such heterogeneity are mandatory when designing a robust query language for knowledge graphs.

Although SPARQL offers optional patterns for handling incompleteness and heterogeneity issues in RDF graphs, it is only one particular way of doing so. Further deficiencies and peculiarities of optional patterns are comprehensively discussed by the research community [1, 2, 17]. Preference semantics seem to be a much more intuitive and still robust way of expressing how the optimal query result may look like [11]. While extensions of SPARQL regarding preferences over data value domains exist [8, 10, 16, 21, 22], a user cannot express her preferences regarding the completeness of query results. We want to call such preferences *structural preferences*, for which we show in Sect. 2 that the operators SPARQL already offers do not suffice for expressing these. Rather cumbersome constructions are necessary to express a query as simple as *I look for cars, preferably including their price and/or brand.* If the preferences are not satisfiable in a given knowledge graph, we are still looking for cars and expect that cars are returned if there are any. In contrast, the works on value preferences above would just allow for expressing certain preferences over the attribute values, e. g., the lowest price or a specific brand.

We contribute the new OPTIMAL operator for SPARQL, enabling the expression of structural as well as some value preferences. In Sect. 3, we motivate and define this operator, and give examples indicating the possibilities we have regarding preference modeling. Equipped with a Pareto-style semantics, we evaluate two different encodings on DBpedia's public SPARQL endpoint[1]. Sect. 5 provides a discussion on preference modeling in the literature. We conclude (Sect. 6) the paper by relating the new operator to the existing ones. Furthermore, we elaborate on some limitations the operator still has and what we plan to do to overcome these.

[1] Our prototypical implementation is available at Github: https://github.com/ifis-tu-bs/optisparql

2 SPARQL AND PREFERENCES

In this section, we give a brief overview of the *Resource Description Framework* (RDF) and its prime query language SPARQL. We then analyze SPARQL's built-in operators and features w. r. t. preference modeling.

SPARQL is the W3C-recommended language for querying RDF data [18, 23]. RDF builds on the notion of *RDF triples* [20], which are triples (s, p, o) consisting of a subject resource s, which is connected via property p to object resource or literal o. As the name suggests, RDF triples describe *resources* and their properties, which are universally identified by IRIs (International Resource Identifiers). Let us assume an infinite supply of IRIs $\mathcal{I}$. Following the RDF standard, subjects s may solely stem from this universe of IRIs, i. e., $s \in \mathcal{I}$. Properties p are also universally identified by an IRI. The reason is that we may want to append information about a certain property, such as their domain or range. Objects o are either resources or concrete data values, called *literals*, from a universe $\mathcal{L}$ of literals. Thus, an RDF triple follows the type $\mathcal{I} \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{L})$. Sets of RDF triples form *RDF graphs* G. In this paper, blank nodes and the entailment semantics induced by RDF or RDFS are irrelevant for the contents of the paper.

As for RDF, subject-predicate-object triples are first-class citizens of SPARQL, there called *triple patterns*. Besides resources and literals, all the components of a triple pattern may be drawn from a set of variables $\mathcal{V}$, that are bound to actual resources/literals during query evaluation. Thus, triple patterns t are elements of the set $(\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V})$. $vars(t)$ denotes the set of all variables occurring in triple pattern t. Sets of triple patterns once more form graph structures called *basic graph patterns* (BGPs).

Let G be an RDF graph. During the evaluation of a triple pattern t w. r. t. G, partial functions μ are returned, mapping the variables in t to resources and/or literals in G. Furthermore, μ assigns resource/literal components of t to themselves, i. e., $\mu : (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V}) \hookrightarrow (\mathcal{I} \cup \mathcal{L})$ with $\mu(x) = x$ for all $x \in \mathcal{I} \cup \mathcal{L}$. The set of all variables $v \in \mathcal{V}$, for which μ is defined is denoted by $dom(\mu)$. μ is a *match for triple pattern* $t = (s, p, o)$ *in* G iff $vars(t) = dom(\mu)$ and $(\mu(s), \mu(p), \mu(o)) \in$ G ($\mu(t) \in$ G for short). Consequently, μ is a *match for BGP* $\mathbb{G}$ *in* G iff $dom(\mu) = vars(\mathbb{G})$[2] and $\mu(t) \in$ G for all triple patterns $t \in \mathbb{G}$. We denote the set of all matches for $\mathbb{G}$ in G by $[\![\mathbb{G}]\!]_G$.

Instead of a classical introduction to syntax and semantics of further language constructs of SPARQL [17], we subsequently discuss certain features of the language and discuss them w. r. t. preferences they express. Given any two SPARQL queries Q_1 and Q_2, for which we want to express that we prefer an answer regarding Q_1 and Q_2 but we would still be happy with an answer to just Q_1. We elaborate on some of SPARQL's options for combining Q_1 and Q_2 w. r. t. the degrees of preference they express. There are three basic operators that allow for the combination of Q_1 and Q_2: conjunction (AND), union (UNION), and optional patterns (OPTIONAL). Furthermore, built-in filter conditions are included to filter query results for some desired properties.

- Q_1: Just asking for all the matches for Q_1 is a safe way of answering our preference query. However, Q_2 is neglected and we will never get an answer to Q_1 and Q_2, even if there are some in the RDF graph.
- Q_1 AND Q_2: This query forms the other extreme because here answers to Q_1 and Q_2 are enforced. If this set of matches is empty, we will not get the opportunity to observe possible matches for Q_1.
  On a technical level, the query is compositionally evaluated. Every match μ must be constructed from *compatible* matches $\mu_1 \in [\![Q_1]\!]_G$ and $\mu_2 \in [\![Q_2]\!]_G$. μ_1 and μ_2 are *compatible* iff they agree on all shared variables, i. e., $\forall v \in dom(\mu_1) \cap dom(\mu_2) : \mu_1(v) = \mu_2(v)$.
- Q_1 UNION Q_2: Every match μ for Q_1 or Q_2 is returned but they are not combined. Therefore, it is not possible to obtain matches obeying the preference order, i. e., that Q_1 must be matched. The same holds for the query Q_1 UNION (Q_1 AND Q_2). Although all matches we desire are included in the result set, we have superfluous matches to Q_1 that could be extended by matches of Q_2.
- Q_1 OPTIONAL Q_2: Such optional patterns are the SPARQL means of handling incomplete information in RDF graphs. Upon evaluation of Q_1 OPTIONAL Q_2, all the matches for Q_1 AND Q_2 are returned, i. e., all preferred matches. Additionally, query evaluation returns all matches for Q_1 that may not be extended by matches for Q_2. Optional patterns enforce our preference only locally, i. e., per match. That means, a match $\mu_1 \in [\![Q_1]\!]_G$ is not a match to Q_1 OPTIONAL Q_2 if there is a compatible match $\mu_2 \in [\![Q_2]\!]_G$. However, the existence of μ_2 and the fact that μ_1 is compatible with μ_2 does not influence the other matches for Q_1. Hence, the set of matches $[\![Q_1$ OPTIONAL $Q_2]\!]_G$ likely contains matches for Q_1 and for Q_1 AND Q_2. Further note that optional patterns constitute the single source of PSPACE-completeness of SPARQL's evaluation problem [17, 19].

Other formulations of queries, e. g., using built-in filter conditions, are possible but generally tend to be quite complex and unreadable (cf. the encodings we use for implementing our preference operator in Sect. 4). Furthermore, such constructions are rather unnatural. It is unlikely that a user comes up with a query like this to fulfill her information need based on structural preferences. Throughout the next section, we aim for an operator in the style of AND, UNION, and OPTIONAL, being simple binary operators between SPARQL queries.

3 MODELING PREFERENCES WITH OPTIMAL

We used a rather simple form of preference between two SPARQL queries so far. This section is devoted to further requirements for a (structural) preference operator and their resolution in so-called *optimal patterns*. We discuss abstract syntax, semantics, and some examples illustrating the use of the new OPTIMAL operator.

3.1 Requirements

Before we start defining new operators for SPARQL, we elaborate on the requirements we aim to satisfy by our proposal.

Types of Preferences. We distinguish two types of preferences, *value preferences* and *structural preferences*. Value preferences apply to attributes of data objects. Examples include *I prefer red cars* or

[2] $vars(\mathbb{G}) := \bigcup_{t \in \mathbb{G}} vars(t)$

I prefer cars with a maximum velocity of 180km/h. Structural preferences, on the other hand, apply to the matches found within an RDF graph. The relevance of a match is assessed by the information it includes. We may, for instance, ask for *cars with color information* or *I prefer cars with details on the manufacturer over those with color/pricing information.* In this paper, we primarily focus on the structural preferences.

Dependencies between Preferences. If more than one preference is involved, they may have different relations to each other. Two or more preferences are independent if satisfying one of them does not affect the assessment of the others. For example, *I prefer red cars manufactured in Germany.* The preference for the color *red* may be assessed independently of the country of origin *Germany.* An expressed preference may also depend on another property of the data object, e. g., *For sports cars, I prefer the color red.* The color preference is only present for cars of a certain type. Other cars may have arbitrary color. Another type of dependency follows the structural category. These dependencies appear in statements like *I prefer cars with information about the manufacturer and its head office.* Here, the inclusion of the *head office* depends on a successful match of the manufacturer.

Ranking and Priority. Besides the just discussed dependencies between preferences, we may want to express preferences with different levels of importance. Queries incorporating statements like *The color is more relevant than the brand* suggest a prioritization of one preference over another. In this case, both preferences should be satisfied but, if this is impossible, the more important preference should. If preferences are equally important, any combination of preferences may be desired for the answer, as long as there is no other dominant answer.

Expression Complexity. As mentioned at the end of the last section, it is possible to get more fine-grained preferences into standard SPARQL queries. However, such queries come at the price of being unreadable because they do not follow the simple binary shape of a conjunction (Q_1 AND Q_2) or an optional pattern (Q_1 OPTIONAL Q_2). Single graph patterns would have to be included multiple times within a single implementation of a preference query. The complexity in modeling preference queries would hinder the application of such a construct in practice. Instead, we pursue a simple and intuitive modeling of preferences by a single operator, i. e., queries of the form Q_1 OPTIMAL Q_2.

We subsequently introduce abstract syntax, intuitive meaning, and formal semantics of preference queries by the OPTIMAL operator, (partially) obeying the just sketched requirements.

3.2 Syntax

Let Q_1 and Q_2 be SPARQL queries, i. e., they do not contain the new OPTIMAL operator. Like OPTIONAL, OPTIMAL is a left-associative operator combining Q_1 and Q_2 to Q_1 OPTIMAL Q_2 in its simplest form. The answer to Q_1 OPTIMAL Q_2 must fulfill Q_1 but also Q_2 if possible. This way, we may formulate necessary conditions for a data object (e. g., we look for cars) as well as preferences for the returned structures (e. g., the color of the returned cars should be known). Candidate matches for Q_1 or Q_2 are less relevant than those fulfilling Q_1 and Q_2. Only if there are no matches for Q_1 and Q_2 (i. e., Q_1 AND Q_2), we return the matches for Q_1 only. In contrast to the related optional pattern Q_1 OPTIONAL Q_2, Q_1 OPTIMAL Q_2 either returns matches to Q_1 AND Q_2 or to Q_1. Thus, non-emptiness of the result set of Q_1 AND Q_2 influences the way the other matches for the query are found. The just mentioned query about cars and their color is exemplified by the following query:

$$?car\ is\_a\ \texttt{Car}\ \text{OPTIMAL}\ (?car\ color\ ?color) \tag{1}$$

Loosely following the SPARQL ASCII syntax within our example queries, ?*car* and ?*color* are variables (leading ?), `Car` is an RDF resource identifying a type (typewriter font), and *is_a* and *color* are properties (italics shape). For literals, we will also use the typewriter font but we put them in single ticks, e. g., `'42.0'` for the literal real value 42.0. We neglect the unfolded and longish syntax of IRIs and literals for the sake of space and readability.

We allow for a clean syntax for expressing prioritized preferences (e. g., a car's color is more important than its brand).

$$\begin{array}{lll} ?car\ is\_a\ \texttt{Car} & \text{OPTIMAL} & (?car\ color\ ?color) \\ & \text{OPTIMAL} & (?car\ brand\ ?brand) \end{array} \tag{2}$$

In (2), the left-associativity of the OPTIMAL operator applies. First, we aim at fulfilling the preference for cars with color information. Only then the cars' brands are evaluated if possible. The first occurrence of the OPTIMAL operator defines the structure with higher priority than later occurrences of the same operator.

In Sect. 3.1, we also argued for preferences of equal importance. To satisfy this requirement by simultaneous fulfillment of the others (especially expression complexity), we have to break with the standard syntactic structure of SPARQL. We have to be able to assign several preferences (Q_1,Q_2,...,Q_n) to a single OPTIMAL operator. Hence, we allow for queries of the shape

$$Q\ \text{OPTIMAL}\ (Q_1, Q_2, \ldots, Q_n).$$

This allows us to put the preferences for color and brand on equal footing, as in

$$?car\ is\_a\ \texttt{Car} \quad \text{OPTIMAL} \quad \left(\begin{array}{l} ?car\ color\ ?color, \\ ?car\ brand\ ?brand \end{array} \right) \tag{3}$$

In query (3), the optimal matches are those cars having information on both, color and brand. If there is no such car in the database, we get all cars with color information as well as all cars with brand information. Only if no such cars exist, we simply get all resources of type `Car`. Especially the example of (3) makes apparent the need for a Pareto-style semantics of the operator. We will discuss some more involved examples of preference modeling in Sect. 3.4, just after we have formally fixed the semantics of the OPTIMAL operator.

3.3 Semantics

To obtain only relevant answers to optimal patterns, we follow a Pareto semantics delivering only the best matches for our queries. Such a semantics entails a *skyline* of matches. Every candidate match must be checked for *Pareto dominance.* Finally, only maximal (i. e., non-dominated) matches are returned.

We assume the standard semantics for arbitrary SPARQL queries $Q_0, Q_1, Q_2, \ldots$, as sketched in Sect. 2 and given in detail, e. g., by Pérez et al. [17]. This allows us to solely focus on the evaluation semantics of the optimal pattern $Q = Q_1$ OPTIMAL Q_2. We call the left-hand side of an optimal pattern its *necessary part*, i. e., the

necessary part of Q is Q_1. The right-hand side of optimal patterns contains one or more subqueries, forming the *preferred part* of them. Since preferred parts of optimal patterns may contain more than one subquery, we call the number of respective subqueries *preference degree*.

With the just fixed set of notions, we formalize the evaluation semantics of general optimal patterns

$$Q = Q_0 \text{ OPTIMAL } (Q_1, Q_2, \ldots, Q_n)$$

of preference degree n w. r. t. some RDF graph $\mathbf{G}$. Recall that matches μ to Q necessarily cover Q_0, i. e., there is a match $\mu_0 \in [\![Q_0]\!]_\mathbf{G}$ with $\mu_0 \subseteq \mu$. Furthermore, any subset of the preferred subqueries $Q_1, Q_2, \ldots, Q_n$ may also be covered. A partial function $\mu : (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V}) \hookrightarrow (\mathcal{I} \cup \mathcal{L})$ is called a *candidate match for the optimal pattern* Q iff there are $\mu_i \in [\![Q_i]\!]_\mathbf{G} \cup \{\emptyset\}$ $(i = 0, 1, \ldots, n)$, such that

(1) $\mu_0 \neq \emptyset$ (as Q_0 is necessary to be matched),
(2) for all $i, j \in \{0, 1, \ldots, n\}$, μ_i and μ_j are compatible, and
(3) $\mu = \mu_0 \cup \mu_1 \cup \ldots \cup \mu_n$.

We stress here that any submatch $\mu_1, \ldots, \mu_n$ may be empty. A preferred subquery Q_i $(i = 1, \ldots, n)$ is covered by μ iff there is a match $\mu_i \in [\![Q_i]\!]_\mathbf{G}$ with $\mu_i \neq \emptyset$, contributing to μ, i. e., $\mu_i \subseteq \mu$. The set of all subqueries in Q that are covered by μ is denoted by $cover_Q(\mu)$. In the worst case, no preference is covered, i. e., $\mu = \mu_0$ and $cover_Q(\mu) = \{Q_0\}$.

Among the candidate matches, we are only interested in the best matches. Therefore, a dominance relation between candidate matches is crucial. With respect to the given optimal pattern Q, a candidate match μ *is dominated by candidate match* μ', denoted by $\mu \prec_Q \mu'$, iff $cover_Q(\mu) \subsetneq cover_Q(\mu')$. This means, μ' covers all the preferred subqueries μ does but includes at least one more subquery.

Finally, we call μ a *match for the optimal pattern* Q iff μ is a candidate match for Q that is not dominated by any other candidate match. In other words, matches must be maximal w. r. t. the dominance relation $\prec_Q$.

3.4 Further Examples

In this part of the section, we give two more examples to illustrate the possibilities of preference modeling with SPARQL and the newly introduced optimal patterns. First, we would like to know about the places visited by 'JohnDoe', preferably in 'Germany' but alternatively in 'Europe'. Since Germany is a country and Europe a continent, each type of information require a different query pattern. Once more, we use prioritized preference modeling to implement this query:

```
?person name 'JohnDoe' AND ?person visited ?place
OPTIMAL(?place country ?region FILTER ?region = 'Germany')
OPTIMAL(?place continent ?region FILTER ?region = 'Europe')
FILTER (bound(?region))
```
(4)

The construction of (4) uses built-in filter conditions. Especially the last one (*bound*(?*region*)) guarantees that no result is returned if 'JohnDoe' never went to Germany nor Europe. Furthermore, both preferred subqueries are in conflict with each other. If a place in Germany has been found, variable ?*region* is bound to the literal 'Germany' and cannot be changed to 'Europe' as required by the second preference. This example shows how well prioritized preferences may be implemented using optimal patterns.

During the last example we already used built-in filter conditions of standard SPARQL, which make the implementation of some value preferences available. Suppose, we would like to find a new car, preferably below a price of 20,000 Euro. If this is impossible, we prefer cars with a price between 20,000 and 25,000 Euro over cars with any other price or those cars with no pricing information. Especially the last part of the query requires an optional pattern:

```
?car is_a Car    OPTIMAL    (?car price ?price FILTER
                            (?price <= 20000))
                 OPTIMAL    (?car price ?price FILTER
                            (?price <= 25000))
                 OPTIONAL   (?car price ?price)
```
(5)

Note how the second preference for cars between 20,000 and 25,000 Euro is modeled by incorporating the first preference. If we do not find a car cheaper than 20,000 Euro with the first preference, we will not find one when looking for cars cheaper than 25,000 Euro. Query (5) also shows how natural OPTIMAL patterns combine with standard SPARQL syntax.

The example queries (1)–(5) provide glimpses on the possibilities with this new operator. Next, we observe some characteristics of these queries during different query evaluation processes.

3.5 Query Rewriting

The new operator is likely to profit from specialized query evaluation techniques. These require more time for further research and implementation. For demonstration and testing purposes, we propose query rewriting in order to encode OPTIMAL patterns with existing query patterns. That way, we can evaluate OPTIMAL patterns on standard query processors. The goal is to make the new operator available and usable on any system supporting SPARQL 1.0 without the need of adjusting existing query evaluation.

We have found two different methods of encoding optimal patterns using SPARQL's standard operators. Both methods work by first constructing a superset of desired results and then filtering out all candidates that are dominated by other matches (cf. Sect. 3.3). The superset can be constructed either by combining all possible preference combinations using UNION operators or by selecting all locally dominant mappings using optional patterns. In order to check which preferences are satisfied in each result, we bind a fresh variable in the respective subquery. Thus, the variable remains unbound for results that do not match the preference. We have previously defined a Pareto semantics by pair-wise checking for domination between all answer candidates. Instead of directly comparing matches, we instead group all matches by their fulfilled preferences (i. e., by which $i = 1, ..., n$ a non-empty mapping is chosen for the candidate). In the next step, we determine which groups of preference structures yield at least one result. Therefore, we use EXISTS filter conditions detecting matches for every possible preference and store the result in a fresh variable. Finally, we filter out any results that are within a dominated group.

Similar approaches towards encoding preferences with optional patterns and filter conditions have already been discussed for other SPARQL extensions like PrefSparql [8] or SPREFQL [15]. To the

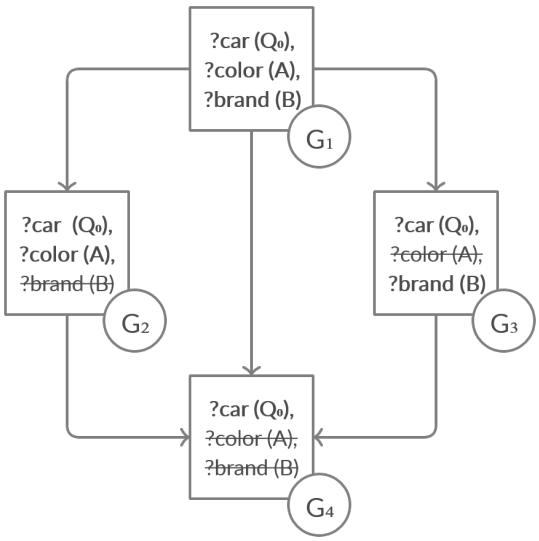

Figure 1: Candidate groups and dominance relations

best of our knowledge, an encoding using UNION operators has not been suggested so far.

For this new encoding process we use query (3) as an example. Given is a query Q with a necessary condition Q_0 (i. e., `?car` needs to be of type `Car`) and two equally important preferences A (i. e., the car should have information on its color) and B (i. e., the car should have information on its brand). All mappings out of the expected result set $[\![Q]\!]_G$ are expected to fulfill Q_0. Furthermore, if at least one mapping exists that fulfills both, A and B, then all mappings are expected to fulfill both preferences. If this is not the case but at least one mapping fulfills either A or B then all mappings need to fulfill A or B. Thus, we define four groups G_1 to G_4. G_1 fulfills both preferences, G_2 only fulfills A, G_3 only fulfills B and G_4 fulfills neither. Each answer candidate is sorted into exactly one group. We construct a basic graph pattern for each of the four groups and assign an identifier to the mapping (e. g., via `BIND(?group, 1)` for G_1). Then we join the results using a UNION operation. The resulting query will result in a superset of our desired resolution set. Dominated results still need to be filtered out.

Fig. 1 shows all four groups with their respective fulfilled (in bold) and unfulfilled (stroked-through) preferences. The arrows between the groups denote dominance relations between all mappings of each group towards all mappings of the respective other group. The outgoing end of the arrow means a group dominates the other, whereas the incoming end means a group is being dominated.

In the next step, we use each of the individual basic graph patterns and execute them as part of an `EXISTS`-clause. We store each result in a new variable. These variables tell us which preference combinations result in at least one mapping.

Last, we use the grouped dominance relations (as shown in Fig. 1) in order to remove all mappings of dominated groups. The resulting query will be answered as previously defined. This technique can also be used for preferences of different importance. For query (2), all of the steps stay the same except for the constructed dominance relation. In order to accommodate for the more important `?color` preference, an additional group relation needs to be added, indicating dominance of G_2 over G_3.

4 EXPERIMENTAL EVALUATION

In this section we evaluate the usability of resulting query results on a small set of typical query patterns. We use encodings as described in Sect. 3.5 in order to simulate our new operator using existing methods. On Github[3], we provide automated transformation processes that take pre-parsed optimal queries and turn them into queries using SPARQL's standard operators. In order to verify whether our method yields usable results, we simulate this rewriting process and test the resulting queries on DBpedia's vast set of data.

Our two main criteria are result set size (i. e., the number of distinct query answers) and query performance (approximated by server response times). In order for the new operator to be useful, result set sizes should be much smaller than those of OPTIONAL in at least some cases. Meanwhile, performance should stay within acceptable boundaries for a user to take advantage of it.

4.1 Setup and Configuration

It is our goal to roughly outline query performance and result set sizes in a real-world querying scenario. We compare them to the results of SPARQL's existing method for handling incompleteness (i. e., optional patterns). To do so, we have chosen some queries which are well-formed and use the OPTIONAL operator. In these queries we replace the OPTIONAL operators by OPTIMAL. We provide a tool generating the rewritten queries via both methods (UNION and OPTIONAL). In order to avoid programming a full query parser, our tool instead generates queries from a simple JSON object, that we construct manually.

Using the automatically generated SPARQL 1.0 encodings, we can test the queries on a Virtuoso instance. In order to better simulate a real-world scenario and to make our procedures and results replicable, we choose to use DBPedia's public SPARQL endpoint[4] [3, 14]. On the other hand, our performance results are less precise by doing so. The comparatively small HTTP overhead does not change the general notion of our results, though. For better comparability we use `COUNT(*)` and `DISTINCT` operators to flatten result sizes and to avoid unrepresentative size mismatches due to recurring responses.

All our selected configuration options for DBPedia's endpoint are listed in Tab. 1. Some of these deviate from the default values in order to better fit our test scenario. First, we disable the execution timeout by setting it to zero. This is important to execute more complex queries. Also, this setting prevents the endpoint from returning partial results when the timeout is reached, which would otherwise lead to unrepresentative result set sizes. Second, we disable strict checking of void variables. Some of our rewritten queries would otherwise be rejected. Last, we set the results format to JSON for ease of processing.

[3] https://github.com/ifis-tu-bs/optisparql
[4] A web interface can be used at http://dbpedia.org/sparql

Table 1: DBPedia Endpoint Settings

Endpoint Setting	Selected value
Default Dataset Name	http://dbpedia.org
Results Format	JSON
Execution timeout	0
Strict checking of void variables	Disabled
Log debug info at the end of the screen	Disabled
Generate SPARQL compilation report	Disabled

Table 2: Response Sizes of Query Answers

#	Preferences	Original	OPTIMAL	Stripped Results
Stepwise evaluation				
1	1	40	40	0
2a	3	1,138,470	1,138,448	22
3a	3	7,673	6,162	1,511
4	1	2,955	355	2,600
5	1	5,476	355	5,121
6	1	65,001,490	64,941,633	59,857
7a	3	1,344,546	285	1,344,261
8a	3	21,433	1,227	20,206
Simultaneous evaluation				
2b	3	1,138,470	1,138,448	22
3b	3	7,673	6,162	1,511
7b	3	1,344,546	3,468	1,341,078
8b	3	21,433	1,227	20,206

Finally, we send our unmodified original queries to the SPARQL endpoint, as well as our generated OPTIMAL representations to the endpoint. We record the returned result set size and the browser waiting time. In order to calculate waiting time, we subtract all HTTP related latencies including the time to wait for a connection to be started, to resolve the DNS, to create an HTTP connection, to send the request, and to read the response from memory.

4.2 Result Set Sizes

Tab. 2 shows the result set sizes we have retrieved by executing both the original optional patterns and our optimal encodings. It also shows the difference in result size between both query types. We divided the table in two parts. The first one shows optimal patterns with prioritized preferences according to the order given by the original query. The second section shows the same queries modeled as a skyline of all preferences. Queries with only one preference are the same in both cases. Thus, we only show those queries in the second section that use more than one preference.

Every query using an encoding of our optimal patterns has the same or fewer results than the respective original query. We observe, the number of stripped results varies vastly between 0 and 99.98% of the total result set. Note that this value can never reach 100% because OPTIMAL only strips results that are dominated by at least one other result. Running the two queries in question (numbered 1 and 7) again, this time using a conjunction instead of an optional pattern, leads to the same result set size of 40 for query 1 but an empty result set for query 7. This additional test proves all candidate matches for query 1 to be complete but all candidate matches for query 7 to be incomplete. Querying over a complete dataset has the effect that no answers at all are stripped from the result set. This is expected because every complete answer satisfies all preferences. Thus, all answers are equally relevant and returned as a valid result. On the other hand, querying over an incomplete dataset leads to a much higher number of stripped candidate matches. We also observe that the result set for query 7 is much larger when using a skyline over all preferences (denoted 7b), as opposed to applying them individually (denoted 7a). This happens because results that are rated badly in more important preferences are filtered out despite being well ranked in less important preferences. When applying a complete skyline instead, these results still appear in the result set as long as they are not dominated w. r. t. the total set of preferences. Because of that, a complete skyline always returns at least the same number of results as any ranked preference query over the same query. We expect this difference in results to increase with heterogeneity of incomplete datasets. On the other hand, all other queries have the same number of results for both individual and concurrent preference evaluation. This suggests that partially complete datasets or datasets with homogeneous incompleteness are also very common. In these cases, optimal patterns return the same result sets for both methods of evaluation.

Table 3: Response Times [in milliseconds]

#	Preferences	Original	OPTIMAL (Union)	OPTIMAL (Optional)
Stepwise evaluation				
1	1	30	186	134
2a	3	1,980	3,470	3,820
3a	3	2,640	3,500	3,040
4	1	29	101	84
5	1	85	48	56
6	1	30,170	34,710	81,000
7a	3	15,390	1,530	8,040
8a	3	441	994	945
Simultaneous evaluation				
2b	3	1,980	9,180	8,591
3b	3	2,640	3,450	2,942
7b	3	15,390	1,310	1,223
8b	3	441	589	1,652

4.3 Performance Results

Tab. 3 illustrates the same queries as before. In this table we show the number of inquired preferences and the response times in milliseconds by both the original optional patterns as well as the two encodings using either UNION or OPTIONAL. We have divided this table into two parts as well; again based on the modeled evaluation method.

The first thing to point out is the lack of any queries with more than three preferences. Our encodings for optimal patterns include

a list of all possible preference combinations in order to test for dominance between results. Consequently, query sizes increase exponentially with the number of preferences. The DBPedia SPARQL endpoint uses GET parameters to enquire user queries. For these reasons, every tested optimal pattern with at least four preferences either fails with HTTP error code *414 URI Too Long* or by running into an HTTP timeout because of the quite complex query evaluation. Queries with more preferences need dedicated testing using a Virtuoso instance on a controlled machine. Even for just four preferences we expect these queries to take at least several seconds or minutes under usual conditions based on size and structure of the dataset.

In most cases both types of optimal patterns take significantly more time to process than the original patterns. This is expected because of the additional filtering steps. It is, however, surprising that in 3 out of 12 tests (in particular queries 6, 7a and 7b) evaluating optimal patterns is actually faster. In all three cases the majority of results are stripped from the result set. We assume that this reduction of results leads to a faster computation of a distinct result set during the projection step. After all, applying a set of filter conditions can be done in linear time whereas pair-wise comparison for distinctiveness can only be done in quadratic time. More tests are needed in order to verify this conjecture.

Both encodings deliver similar performance results. The encoding using optional patterns is slightly faster in most instances. This may be because of optimized query evaluation for optional patterns that filter out some of the dominated answers more efficiently. Due to the nature of our testing methods, this may also be caused by random interference from other influence factors. The general notion, however, consists of relatively equal response times via both encodings. Some queries pose an exception to this general observation. In particular, queries 6, 7a and 8b lead to better response times of the UNION-based encoding. Exchanging the evaluation method for queries 7 and 8 leads to similar response times between the two encodings, indicating an impact of the preference order on response times of query evaluation. Thus, we consider the UNION-based encoding to be more scalable due to its lower fluctuation in performance. This is no surprise, given the evaluation complexity of union queries to be NP-complete while the same problem already is PSPACE-complete for queries with optional patterns [19].

5 RELATED WORK

Preferences are well-studied in the realm of relational databases [5–7, 12]. Some developments have even been made in the Semantic Web [8, 16, 21, 22]. The just mentioned works focus on a preference model based on data values, i. e., value preferences (cf. Sect. 3.1). In contrast, we let the user define her preferences regarding the information included in query answers.

Modeling value preferences with SPARQL has been the goal of multiple extensions to the language. Pivert et al. [16] offer a good overview on different techniques that are already in use.

The survey authors argue that **Top-k Queries** can be expressed even without any extensions but using standard operations. A simple example is `ORDER BY DESC(?score) LIMIT 10`, in which query results will be sorted by a specified ranking condition and then truncated to only include the best k answer candidates. A preference for the 100 cheapest cars can be modeled similarly: `ORDER BY ASC(?price) LIMIT 100`. Pivert et al. note the inefficiency of completely answering a query and filtering afterwards. Thus, they point out the importance of optimizing the information retrieval process for this common kind of query, being an observation we share with them.

The extension **f-SPARQL** supplies fuzzy logic operators to built-in filter conditions of SPARQL. These operators can be used to rate and select mappings based on relative data values. Keywords and operators to express the desired value range are used. A user saying *I prefer cars that are comparatively cheap* can express this preference in f-SPARQL by using a keyword inside a filter condition like `FILTER (?price = low)`. Likewise, a user preferring *a car that costs around $30,000* can express this wish using a fuzzy operator like `FILTER (?price AROUND 30000)`. Based on the chosen keyword, f-SPARQL will then define a lower and upper threshold for values to fit the description. The value of the selected thresholds depends on scale and variability of stored data in the database graph. Values will then be ranked according to a trapezoid function over these thresholds. For example, when looking for cheap cars a value will be rated with a zero if it exceeds the upper threshold and a one if it falls below the lower threshold. Any value in between will be ranked between zero and one linearly according to its position within the two thresholds. By default all values ranked greater than zero will be returned. It is also possible to specify a parameter k to retrieve only the k highest ranked values. When specifying multiple preferences, the criterions can be ranked by supplying an additional ranking parameter. `FILTER (?price = low) with 0.3` will reduce the impact of the price preference to 30%.

A different extension with a similar goal is called **PrefSPARQL**. This extension takes a slightly different approach in that it does not assign numerical ratings to query results. Instead, PrefSPARQL generates a Pareto skyline of results based on the individual preferences. Based on previous work of Siberski et al. [21], the extension adds a new PREFERRING clause to the language. A user preferring *a car that costs less than $30,000 if available* can express this statement using the new clause that acts similarly to a FILTER method: `PREFERRING ?price < 30000`. Just like f-SPARQL, it also supports fuzzy operators like AROUND or HIGHEST. It includes a PRIOR TO operation that allows ranking between multiple preferences. When used, a prioritized preference will be evaluated first. A less important preference will then apply its filter to the result set of the prioritized preferences. PrefSPARQL queries can be rewritten using OPTIONAL and `FILTER NOT EXISTS` operations to be answered by any SPARQL 1.0-compatible DBMS.

In addition to the aforementioned techniques listed in the survey, we mention **SPREFQL** as an important extension. Troumpoukis et al. [22] use the groundwork provided by Chomicki [7] to bring his relational algebra to the Semantic Web. Instead of rating or filtering results based on their respective data values, this framework makes use of pairwise comparison to generate a skyline. For this reason SPREFQL adds a new PREFER clause to SPARQL that enables a user to specify when to prefer a result set over another one. A query selecting price and color from cars could be extended so that lower prices and a blue color are preferred. The resulting clause would look like this: `PREFER ?price1 ?color1 TO ?price2`

`?color2 IF (?price1 < ?price2) AND (?color1 = "blue" && ?color2 != "blue")`. The part after PREFER describes the first result's values. These are compared to the second result which, in turn, is defined after the TO operation. The specified variable assignment must match the one specified in the SELECT operation. Everything after the IF operation defines preferences. `?price1 < ?price2` means that a result with a lower price is to be preferred over one with a higher price. Multiple preferences can be combined using the new AND operation to create a Pareto skyline. It is also possible to specify a ranking between preferences using the OVER operation. Just like PrefSPARQL queries, queries using SPREFQL can be rewritten to be answered by a SPARQL 1.0 compatible DBMS.

Keles et al. [10] have recently proposed a method to generate Pareto skylines on the client. The so-called **SkyTPF** extends SPARQL by a new SKYLINE OF clause that enables users to prefer either as small values as possible using MIN or values as big as possible using MAX. Stating a preference towards cheap cars can be accomplished by supplementing a SPARQL query like this: `SKYLINE OF ?price MIN`. It is possible to specify multiple preferences by separating them with a comma; SkyTPF will then return a Pareto skyline. This paper is worth mentioning despite its small feature set because it computes preference results in a new and interesting way. Instead of relying on query rewriting, SkyTPF provides an API on the client to send multiple queries to a backend that supports Bindings-Restricted Triple Pattern Fragments as described by Hartig et al. [9]. The query results are then processed on the client using Divide & Conquer to generate a skyline. Using this method Keles et al. achieve a more efficient way to process large skyline queries than previous research.

6 CONCLUSION

We have presented a new operator for SPARQL that allows for a fine-grained handling of the inherent incompleteness of RDF knowledge bases. With the example of SPARQL's standard operators, we have developed requirements for a structural preference operator. The new OPTIMAL operator positions itself just between the rather restrictive query conjunction and the loose optional patterns of SPARQL. In fact, one can prove that for any two queries Q_1 and Q_2, the set of matches of the optimal pattern Q_1 OPTIMAL Q_2 is just between those evaluated for Q_1 AND Q_2 and Q_1 OPTIONAL Q_2:

$$[\![Q_1 \text{ AND } Q_2]\!]_G \subseteq [\![Q_1 \text{ OPTIMAL } Q_2]\!]_G \subseteq [\![Q_1 \text{ OPTIONAL } Q_2]\!]_G.$$

We thereby forged a new construct that is worth studying deeper. For instance, we did not yet analyze the operator's expressive power and/or complexity. Our experiments have shown that the encodings we used for evaluating optimal patterns on a standard SPARQL query engine have deficiencies regarding evaluation runtime. One reason is the increased size and complexity of the encoded queries. Better encodings or the incorporation of specialized evaluation procedures [4, 10, 13] may enhance the overall evaluation time.

The new operator still has some limitations and important semantic differences compared to most operators of standard SPARQL. For a potentially incomplete RDF graph, SPARQL usually returns only those results that are guaranteed to be correct answers, i. e., *certain answers*. Additional RDF triples that are added later almost always yield additional or more complete matches to a query. However, added triples will not cause other mappings to be removed from a result set. We defined the OPTIMAL operator to work very differently. Results to OPTIMAL queries deliver the best matches possible for the data at hand. No match is guaranteed to stay a match universally. Thus, added triples may remove formerly dominating matches from result sets. The just described situation is similar to non-monotonicity of some optional patterns [2].

Optimal patterns provide an easy yet precise method for modeling preferences. We have shown how structural and even some value preferences can be modeled. Preferences can have different prioritization or can be used collectively to form a skyline. We have not yet discussed dependencies between multiple preferences, though. Dependencies between distinct preferences are necessary for advanced preference modeling. They allow a user to attach conditions to preferences. They can also ensure structural integrity of results that answer contiguous preferences like *I prefer cars with information about the manufacturer and its head office* — the inclusion of the head office depends on the inclusion of the manufacturer. In the current state, optimal patterns allow for easier modeling of preferences in general. Modeling dependent preferences still requires a significant amount of modeling using complex filter conditions. In order to enable a more usable method of this feature, we are currently considering an additional set of operators for modeling dependencies between multiple preferences. These operators should allow for complex $n : m$ relations between preferences and enable both, positive as well as negative dependencies. This is subject for future work.

REFERENCES

[1] Marcelo Arenas, Claudio Gutierrez, Daniel P. Miranker, Jorge Pérez, and Juan F. Sequeda. 2013. Querying Semantic Data on the Web? *SIGMOD Rec.* 41, 4 (2013), 6–17. https://doi.org/10.1145/2430456.2430458

[2] Marcelo Arenas and Jorge Pérez. 2011. Querying Semantic Web Data with SPARQL. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '11)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/1989284.1989312

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*. Springer Berlin Heidelberg, Berlin, Heidelberg, 722–735.

[4] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. 2004. Efficient Distributed Skylining for Web Information Systems. In *Advances in Database Technology - EDBT 2004 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 256–273. https://doi.org/10.1007/978-3-540-24741-8_16

[5] Patrick Bosc and Olivier Pivert. 1995. SQLf: a relational database language for fuzzy querying. *IEEE Transactions on Fuzzy Systems* 3, 1 (Februar 1995), 1–17. https://doi.org/10.1109/91.366566

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst.* 27, 2 (June 2002), 153–187. https://doi.org/10.1145/568518.568519

[7] Jan Chomicki. 2003. Preference Formulas in Relational Queries. *ACM Trans. Database Syst.* 28, 4 (December 2003), 427–466. https://doi.org/10.1145/958942.958946

[8] Marina Gueroussova, Axel Polleres, and Sheila A. McIlraith. 2013. SPARQL with Qualitative and Quantitative Preferences (Extended Report). *University of Toronto CSRG Report* 619 (October 2013).

[9] Olaf Hartig and Carlos Buil-Aranda. 2016. Bindings-Restricted Triple Pattern Fragments. In *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna (Eds.). Springer International Publishing, Cham, 762–779. https://doi.org/10.1007/978-3-319-48472-3_48

[10] Ilkcan Keles and Katja Hose. 2019. Skyline Queries over Knowledge Graphs. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer International Publishing, Cham, 293–310. https://doi.org/10.1007/978-3-030-30793-6_17

[11] Werner Kießling. 2002. Foundations of preferences in database systems. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB*

'02). VLDB Endowment, Hong Kong, China, 311–322.

[12] Werner Kießling and Gerhard Köstler. 2002. Chapter 91 - Preference SQL — Design, Implementation, Experiences. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias (Eds.). Morgan Kaufmann, San Francisco, 990 – 1001. https://doi.org/10.1016/B978-155860869-6/50098-6

[13] Michael Morse, Jignesh M. Patel, and H. V. Jagadish. 2007. Efficient skyline computation over low-cardinality domains. In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*. VLDB Endowment, Vienna, Austria, 267–278.

[14] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In *The Semantic Web - ISWC 2011*. Springer Berlin Heidelberg, Berlin, Heidelberg, 454–469. https://doi.org/10.1007/978-3-642-25073-6_29

[15] Peter F. Patel-Schneider, Axel Polleres, and David Martin. 2018. Comparative Preferences in SPARQL. In *Knowledge Engineering and Knowledge Management*, Catherine Faron Zucker, Chiara Ghidini, Amedeo Napoli, and Yannick Toussaint (Eds.). Springer International Publishing, Cham, 289–305. https://doi.org/10.1007/978-3-030-03667-6_19

[16] Olivier Pivert, Olfa Slama, and Virginie Thion. 2016. SPARQL Extensions with Preferences: A Survey. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 1015–1020. https://doi.org/10.1145/2851613.2851690

[17] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (August 2009). https://doi.org/10.1145/1567274.1567278

[18] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[19] Michael Schmidt, Michael Meier, and Georg Lausen. 2008. Foundations of SPARQL Query Optimization. *arXiv:0812.3788 [cs]* (2008). http://arxiv.org/abs/0812.3788

[20] Guus Schreiber and Yves Raimond. 2014. *RDF 1.1 Primer*. Technical Report. W3C.

[21] Wolf Siberski, Jeff Z. Pan, and Uwe Thaden. 2006. Querying the Semantic Web with Preferences. In *The Semantic Web - ISWC 2006*, Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora M. Aroyo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 612–624. https://doi.org/10.1007/11926078_44

[22] Antonis Troumpoukis, Stasinos Konstantopoulos, and Angelos Charalambidis. 2017. An Extension of SPARQL for Expressing Qualitative Preferences. In *The Semantic Web – ISWC 2017*, Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin (Eds.). Springer International Publishing, Cham, 711–727. https://doi.org/10.1007/978-3-319-68288-4_42

[23] W3C SPARQL Working Group. 2013. SPARQL 1.1 Overview. W3C Recommendation. http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. All contributers listed at https://www.w3.org/TR/sparql11-overview/#Acknowledgements.