Extending SPARQL for Recommendations

Victor Anthony Arrascue Ayala Martin Przyjaciel-Zablocki
Thomas Hornung Alexander Schätzle Georg Lausen

Department of Computer Science
University of Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
arrascue|zablocki|hornungt|schaetzle|lausen@informatik.uni-freiburg.de

ABSTRACT

For processing data on the Web, recommender systems and SPARQL are two popular paradigms, which however have rather different characteristics. SPARQL is a declarative language on RDF graphs which allows a user to precisely specify the desired information. In contrast, a recommender system suggests certain items to a user, based on similarity to other users or items. As the data to be processed by a recommender may be an RDF graph as well, the question arises whether both processing paradigms can benefit from each other. RecSPARQL fills this gap by extending the syntax and semantics of SPARQL to enable a generic and flexible way for collaborative filtering and content-based recommendations over arbitrary RDF graphs. Our experiments on the MovieLens data set demonstrate the applicability of our approach.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*

General Terms

Algorithms, Design, Experimentation

Keywords

Semantic Web Applications, Recommendation Systems, SPARQL Extension, Content-based, Collaborative Filtering

1. INTRODUCTION

One of the initial works on extracting semantic data from Wikipedia asked the question "What have Innsbruck and Leipzig in common?" [4]. With the success of Semantic Web initiatives like DBPedia [3] and other knowledge bases like YAGO [21], queries that express connections between different real-world objects can be approached by analyzing the paths in the underlying RDF graph. SPARQL 1.1 [8] was designed to find such *explicit* connections, e.g. by means of property paths. However, asking for the commonalities of both cities may mean more than simply asking for common objects to which we can find paths. We might also be interested to figure out whether both cities are similar where the notion of similarity is treated in a more fuzzy sense.

Asking for similarity between objects is one task of recommender systems and therefore the facet of similarity has been widely researched in the area of recommender systems [1] with a special focus on certain domains like music [10, 14], movies [12], or books [16] where a similarity function is adopted to predict new items.

Obviously, recommenders and SPARQL can be combined by using SPARQL for processing the recommender's input data, respectively output data. For instance, the approach followed in [17] and similar works is depicted in Fig. 1. In this scheme the task is separated into (A) pre-processing, (B) computation of recommendations, and (C) post-processing. Typically, the recommender component in (B) is outfitted with classic recommendation algorithms and it expects a set of users, their consumed items, and assigned ratings as input. As output it indicates users their recommended items, and prediction scores.

In the context of a collaborative filtering approach, (B) computes for each user a set of n similar users, i.e the neighborhood, from which recommendations are derived. Suppose, that one requires to customize this building process in order to impose certain characteristics on to the neighborhoods. For instance, one might want to consider neighborhoods where the age of each of their members differs by δ wrt. the user u for whom recommendations are produced. Another kind of neighborhoods could be one in which each of their members is geographically close to u.

These simple examples of customizing the neighborhood are a feature which is missing in existing recommenders for the Semantic Web. A system as the one depicted in Fig. 1 does not provide a shared recommendation model among (A), (B) and (C) in order to be able to consider all possible restrictions.

In this paper, we demonstrate that recommenders and SPARQL may complement each other more deeply, giving us the opportunity to provide integrated systems which will offer more adequate and interesting results to users than possible so far. To the best of our knowledge, RecSPARQL is the first extension of SPARQL that allows the customization of recommendations to offer the best of two worlds: a

SIGMOD '14, June 22-27, 2014, Snowbird, Utah, USA

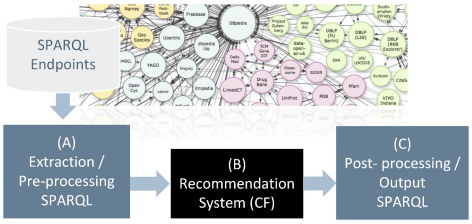

Figure 1: Abstract not integrated recommendation system

tight integration with the Semantic Web, building upon the flexibility inherited from SPARQL for highly parameterizable queries, combined with common recommendation techniques, i.e. content-based and collaborative filtering. Overall, our contributions can be summarized as follows:

- We propose RecSPARQL, an extension of SPARQL for customized recommendations to fill the gap between recommender systems and SPARQL.
- RecSPARQL is not restricted to a pre-defined and fixed ontology but can be applied to arbitrary RDF graphs from any domain.
- We also provide a standalone recommender repository as an extension to Sesame[1], RecSesame, which makes it possible to run recommendation queries by simply changing your project's dependencies or replacing the libraries.

The remainder of the paper is structured as follows. In Section 2 we briefly introduce the basics of recommender systems. Section 3 gives an overview of RecSesame. In Section 4 we present an introduction to RecSPARQL queries and the underlying recommendation model, whereas Section 5 explains some details of how recommendations are computed. Experiments in Section 6 demonstrate the applicability of our approach followed by the related work in Section 7 and a conclusion with an outlook on future work in Section 8.

2. RECOMMENDER SYSTEMS

Recommender Systems (RS) are widely used to suggest items of interest, where the amount of available information exacerbates the process of taking a decision. In order to supply such suggestions, classical RS need some knowledge about *user preferences* or the *features of items*, which forms the underlying *recommendation model*. Whereas user preferences consist of some personal information (age, gender, spoken language, ...) and ratings (e.g. user u rated item i with 3/5 stars), features of items are based on representative attributes like price or description. The task of a recommender system is to utilize existing knowledge to predict which of the non-rated items might be the most valuable suggestions for a user.

There are two widely-used techniques for that: (1) *Content-Based Filtering* focuses on properties of items. The suggestions are based on the degree of similarity between already consumed items and unconsumed items. Typically, a similarity function takes two items with a set of (meaningful) features as input and returns a value representing the degree of similarity. (2) *Collaborative Filtering* is based on the relationship between users and items. The basic idea is that one can get the best recommendations from someone who has a similar taste. In this case, the similarity of two items is not derived from objective properties but from the ratings of users who tend to rate in a similar way.

[1]Framework for storing, querying, and reasoning RDF (`http://www.openrdf.org`)

3. RECSESAME OVERVIEW

RecSPARQL is an extension of SPARQL to generate recommendations based on the knowledge contained in RDF graphs. For the evaluation of RecSPARQL queries, we extended the Sesame SPARQL engine (RecSesame) with several recommender algorithms. A basic architecture is depicted in Fig. 2.

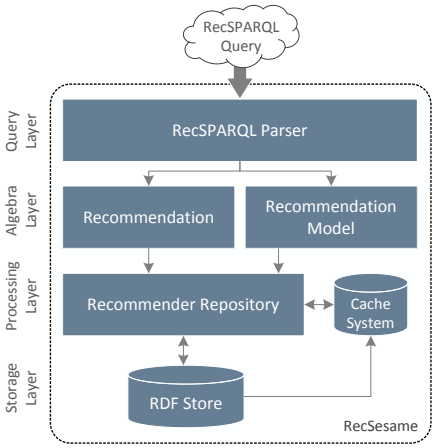

Figure 2: RecSesame Overview

Input Data. In contrast to other recommender systems, which require highly structured inputs, there are no preconditions, neither on the schema of the data nor on the ontology. This is achieved by an syntax-driven approach, which consists in defining the current *recommendation model* ad-hoc within the query. Hence, any kind of RDF graphs that is supported by Sesame can be utilized enabling flexible and dynamic recommendations.

Query Layer. RecSPARQL extends the syntax of SPARQL with recommender functionalities. We included a new query type, `RECOMMEND`, which allows us to project generated recommendations and a `BASED ON` clause to define the underlying recommendation model for computing similarities. From a technical perspective, we enriched the existing SPARQL parser of Sesame to support RecSPARQL as well. Section 4 gives a more detailed introduction to the syntax.

Algebra Layer. Indeed, we utilized existing SPARQL 1.1 algebra operators to express the semantic of RecSPARQL where possible, nonetheless it was inevitable to introduce also some new operators to define e.g. a similarity measure for items and users. A more detailed explanation on that can be found in Section 5.

Processing Layer. New algebra operators, which cannot be mapped to existing ones, require a wide range of changes to Sesames processing layer. We implemented our

own `SailRecommenderRepository` that computes similarities and generates recommendations efficiently based on the recommendation model defined within a RecSPARQL query.

Storage Layer. Our `SailRecommenderRepository` can use the same loaded data as the original one for SPARQL 1.1 queries, hence we support both RecSPARQL and SPARQL queries despite our changes to Sesame. To support our new algebra operators more efficiently, we included a caching layer for sets of queries where the recommendation model does not vary significantly. Section 6 shows some experiments to demonstrate the applicability of our approach.

4. RECSPARQL QUERIES

The syntax of RecSPARQL adheres to the principles of SPARQL 1.1, but allows us to produce recommendations over arbitrary RDF graphs. A basic query has the following structure:

```
RECOMMEND [Projected Variables]
USING [Recommendation Algorithm]
WHERE { [Basic Graph Pattern] }
BASED ON { [RecSPARQL Type Pattern]
           [RecSPARQL Model Building Pattern] }
```

Similar to SPARQL `SELECT`, we can use the `RECOMMEND` clause to specify the variables that should appear in the query result and the `WHERE` clause defines the Graph Pattern that matches a subgraph of the input RDF graph. The `USING` clause expects the name of an implemented recommendation algorithm, whereas the `BASED ON` specifies the underlying recommendation model. Currently supported strategies include content-based filtering (*CB*), collaborative filtering (*CF*) and hybrid recommendations (*HR*).

4.1 Recommendation Model

Recommendation algorithms need a highly structured model of the data called recommendation model to compute e.g. similarities between users and items. However, RDF graphs exhibit almost complementary characteristics due to their inherent diversity and unstructured data. And without a clear definition of items, users and their relations it is not feasible to apply any recommendation techniques at all. To overcome this gap, RecSPARQL enables to specify this missing definition within the `BASED ON` clause. At first place, one has to identify items and users in the current RDF graphs. This is done by so-called *RecSPARQL Type Pattern* (RTP) which are expressed through type relationships as in SPARQL. Currently, the following RecSPARQL-specific type classes are supported:

- `recsparql:User`: user who consumes items
- `recsparql:Item`: item that is consumed by users
- `recsparql:UserRating`: user-item rating
- `recsparql:ItemRating`: global item rating

We use the dataset in Fig. 3 as running example. For simplification, rated movies are interpreted as watched movies. Fig. 4 shows a RecSPARQL query, whose purpose is to recommend movies which are similar in terms on their *genres* using a content-based approach. The RTP (line 6+7) defines `?movie` as the recommended item and `?user` is a person for whom we want to recommend new items.

```
1  RECOMMEND ?user ?movie.REC ?RATING USING CB
2  WHERE {
3      ?user movies:rated ?persRating .
4      ?persRating movies:ratedMovie ?movie }
5  BASED ON {
6      ?user rdf:type recsparql:User .
7      ?movie rdf:type recsparql:Item .
8      ?movie movies:genre ?genre }
   Result:   Bob, Star Wars, 1
             Bob, Gravity, 0.5
```

Figure 4: RecSPARQL query that recommends movies

But its not enough to identify items and users only. If we want to benefit from the Semantic Web as a meaningful knowledge base for recommendations, we have to find ways to utilize other manifold information contained in a RDF graph, e.g. attributes of items or the different ways of how a user is connected to an item. Analogous to Basic Graph Pattern, we can define a *RecSPARQL Model Building Pattern* (RMBP) that matches a subgraph with the desired information that we want to consider in our recommendation model. A RMBP is in turn composed of multiple so-called property graphs, each representing one selected piece of information. This might be for example a path that describes the movies a user watched or a path which selects an attribute like the genre that classifies a movie. Depending on the chosen recommender algorithm, its important to specify appropriate features. Whereas for an collaborative filtering approach, we need paths between users and consumed items, a contend-based one needs paths from items to some attributes. As the current contend-based example (Fig. 4, line 8) defines genre to be the only attribute of interest for a movie, there is a path from the recommended item `?movie` to its attribute `?genre`. For a collaborative filtering approach as show in Fig. 5, we specify paths to express which items (`?movie`) was consumed by whom `?user` (cf. line 15+16).

```
1  RECOMMEND ?user ?movie.REC
2      ?RATING USING CF
3  WHERE {
4      ?user movie:rated ?persRating .
5      ?persRating movie:ratedMovie ?movie .
6      ?persRating movie:rating ?uRating .
7      ?user user:age ?age .
8      FILTER ( ?age >= ?age.REC - 5 ||
9          ?age <= ?age.REC + 5 )
10 }
11 BASED ON {
12     ?user rdf:type recsparql:User .
13     ?movie rdf:type recsparql:Item .
14     ?uRating rdf:type recsparql:UserRating .
15     ?user movie:rated ?persRating .
16     ?persRating movie:ratedMovie ?movie}
   Result:   Bob, Gravity, 6.3
             Bob, Star Wars, 1.4
```

Figure 5: RecSPARQL query with FILTER

4.2 Variables

RecSPARQL offers two different types of variables. The first type equals usual SPARQL variables that are used in the *WHERE* clause to match a subgraph that serves as input for the recommendation model. In Fig. 4, they correspond to `?user` and `?movie` which represent users with their watched

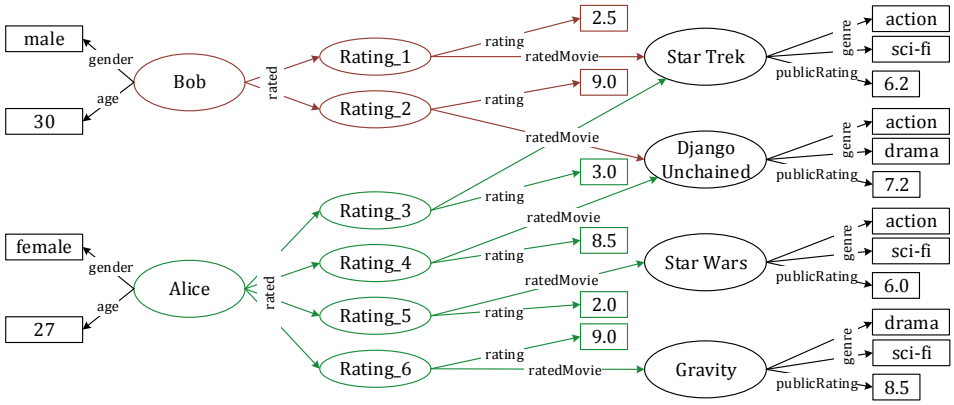

Figure 3: an RDF-graph that depicts a scenario where users watch movies and rate them.

movies. The second type of variables are RecSPARQL-specific and distinguished with the suffix `.REC`. They provide both insight and control into the recommending process. The key point of such a process is the computation of similarities, e.g. between two movies based on their genre. In this case, both movies and their similarity value can be easily accessed by means of the variable `?movie` that represents a watched movie, `?movie.REC` that represents a movie recommendation and the reserved variable `?SIMscore` that contains the computed similarity value of both. A further reserved variable `?RATING` provides the predicted rating for a recommendation. Moreover, both types of variables can be used the same way to reduce the amount of resulting mappings, e.g. with *(xsd:double(?SIMscore) >= 0.5)* which restricts the output to only those recommendations, which have a higher similarity value than 0.5.

4.3 Advanced Features.

As flexibility is the main goal of RecSPARQL, queries can be highly customized. For instance, the CF query in Fig. 5 aims to output suggestions from like-minded users that are five years younger or older (cf. line 8+9). Such a filter affects the so-called *neighborhood* [7] of each user, e.g. restricts who is compared to whom. Being able be to parametrize neighborhoods dynamically within a query enables not only more efficient computations but has also an high influence on the predicted rating accuracy as we will demonstrate in Section 6.

Furthermore, the query in Fig. 5 shows the difference between contend-based (CB) and collaborative filtering (CF) in RecSPARQL. CF is based on similarities between users, hence the recommender needs to know in which respect users should be considered to be similar. In our example, we define watched movies as a criteria for similarity. Thus, the RecSPARQL Model Building Pattern (RMBP) has to specify which movie has been watched by whom. The current example defines this relation as a path from a person `?user` to its consumed item `?movie`.

Comparing the results of the query in Fig. 5 to the query in Fig. 4, we can observe that "Gravity" got the highest predicted rating for "Bob" since there is only one similar user "Alice" who rated this movie with a high value. Such ratings are represented as another class of nodes, that can be specified as type within the RecSPARQL Type Pattern (RTP). The *recsparql:UserRating* represents explicit ratings of items given by a user; the *recsparql:ItemRating* represents a public available rating of items given by an external entity. The query in Fig. 6 is again contend-based, but to improve the accuracy of recommendations, user ratings are considered.

As RecSPARQL is based on SPARQL 1.1, it also supports grouping and aggregation as shown in Fig. 6. This CB query not only specifies what kind of recommendations to compute, but also how to pack the results. We project a unique group for each user and recommended movie. If the movie is recommended multiple times, we average the ratings and use HAVING to restrict the groups.

```
1  RECOMMEND ?user ?movie.REC
2      (AVG(?RATING) AS ?avgRat)
3      USING CB
4  WHERE {
5      ?user movie:rated ?persRating .
6      ?persRating movie:ratedMovie ?movie .
7      ?persRating movie:rating ?uRating .
8      ?movie movie:publicRating ?pubRating }
9  BASED ON {
10     ?user rdf:type recsparql:User .
11     ?movie rdf:type recsparql:Item .
12     ?uRating rdf:type recsparql:UserRating .
13     ?movie movis:genre ?genre}
14 GROUP BY
15     ?user ?movie.REC ?pubRating.REC
16 HAVING (?avgRat > ?pubRating.REC)
```

Result: Bob, Gravity, 6.75
Bob, Star Wars, 5.0

Figure 6: RecSPARQL query with GROUP BY

5. COMPUTING RECOMMENDATIONS

Graph pattern matching is the mechanism in SPARQL to retrieve data. In RecSPARQL this is used to match the input data required for constructing the recommendation model, which is in turn used by the recommendation algorithms. Thus, RecSPARQL is based on the semantics of SPARQL 1.1 where possible, and even the new clauses introduced in Section 4 are mapped to existing SPARQL operators at first glance. However, recommender specific computations that are applied to the retrieved data are no longer expressible with the SPARQL 1.1 algebra only, as they are based on complex similarity measurements.

In order to give more details about how the recommendation model is constructed from the subgraph that matched a RecSPARQL query and how it is used by the recommendation algorithms, we first need to define some formalisms based on the semantic of SPARQL 1.1. For a more detailed description of RDF and SPARQL we refer to [11] and [8, 15], respectively.

Let I, B and L be respectively the set of all IRIs, the set of all literals and the set of all blank nodes. Then a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple*. An *RDF graph* is a set of *RDF triples*. Let an *RDF term* be $T \in (I \cup B \cup L)$. Then μ is a solution mapping, a partial function $\mu : V \to T$. The domain of μ, dom(μ), is the subset of V where μ is defined. Two solution mappings μ_1 and μ_2 are compatible if, for every variable $v \in dom(\mu_1) \cap dom(\mu_2)$, then $\mu_1(v) = \mu_2(v)$. Note that two mappings with disjoint domains are always compatible. Next, Ω is a multiset of solution mappings, obtained by evaluating a graph pattern as described in [15]. We report the definition of the join between Ω_1 and Ω_2 as:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible mappings}\};$$

Let Ω_W be the multiset of solution mappings obtained by evaluating the graph pattern P_W specified in the *WHERE* clause. Moreover, let Ω'_W be the multiset of solution mappings obtained by evaluating a renamed graph pattern P'_W, where the suffix "*.REC*" is appended to each variable.

A RecSPARQL-specific evaluator allows us to generate Ω_B, the multiset of solution mappings resulting from the evaluation of the *RMBP* in the *BASED ON* clause. The solution mappings within Ω_B are defined over three variables, either the variable user and a renamed variable user (with suffix "'*.REC*") or the variable item and a renamed variable item (with suffix "'*.REC*") and a third variable which represents the similarity score *?SIMscore*.

With the multiset of mapping solutions obtained from the different clauses of the RecSPARQL query, it is possible to build the final result. Let Ω_{REC} be the resulting multiset:

$$\Omega_{REC} = \Omega_W \bowtie \Omega_B \bowtie \Omega'_W$$

Computing predicted ratings. Ω_{REC} does not contain the predicted rating $?RATING$. Given a $\mu \in \Omega_{REC}$, this mapping is computed and added as follows. A new multiset of solution mappings is built to incorporate the predicted rating.

$$\Omega_{\text{RecSPARQL}} = \{\mu \mid \mu' \in \Omega_{REC},\ dom(\mu) = \{dom(\mu') \cup ?RATING\},\ \forall\ v \in dom(\mu'),\ \mu(v) = \mu(v') \text{ and } \mu(?RATING) \text{ is calculated as described}\}$$

A predicted rating is a linear function of the similarity score and the explicit ratings or metrics specified through the *RTP*. Let's assume that the following variables have been specified by means of the TP_{RTP}:

?user : *resparql:User*

?item : *resparql:Item*

?userRating : *resparql:UserRating*

?itemRating : *resparql:ItemRating*

To simplify the illustration formulas, variables that represent the values of certain mappings will hereinafter be used:

$$\begin{aligned} u &= \mu(?user) \\ u' &= \mu(?user.REC) \\ i &= \mu(?item) \\ i' &= \mu(?item.REC) \\ r(u,i) &= \mu(?userRating) \\ r(u',i') &= \mu(?userRating.REC) \\ r(i) &= \mu(?itemRating) \\ r(i') &= \mu(?itemRating.REC) \\ sim &= \mu(?SIMscore), \text{ can be } sim_{(u,u')} \text{ or } sim_{(i,i')} \\ &\quad \text{depending on the kind of approach triggered} \\ &\quad \text{(user-based or item-based).} \end{aligned}$$

Ratings for a CB query. In case all variables are available on each solution mapping, then we aim to calculate $r(u, i')$:

$$r(u,i') = \frac{\frac{(r(u,i) + r(i))}{2} + sim_{(i,i')} * \frac{(r(u',i') + r(i'))}{2}}{2}$$

In case item-ratings are missing or not provided:

$$r(u,i') = \frac{r(u,i) + sim_{(i,i')} * r(u',i')}{2}$$

In case neither user-ratings nor item-ratings are provided:

$$r(u,i') = sim_{(i,i')}$$

Ratings for a CF query. In case all variables are available on each solution mapping, then we aim to calculate $r(u, i')$:

$$r(u,i') = sim_{(u,u')} * \frac{(r(u',i') + r(i'))}{2}$$

In case item-ratings are missing or not provided:

$$r(u,i') = sim_{(u,u')} * r(u',i')$$

In case neither user-ratings nor item-ratings are provided:

$$r(u,i') = sim_{(u,u')}$$

Finally, we assign the following value to the reserved variable:

$$?RATING \longleftarrow r(u,i')$$

6. EXPERIMENTS

We demonstrate the applicability of RecSPARQL by a case study that emphasizes the advantages of a customizable neighborhood. This feature strongly relies on a tight integration of the recommendation algorithms with SPARQL and wouldn't be feasible if we treated both paradigms as independent boxes. We will see that being able to adjust the neighborhood in accordance with the underlying RDF graph enables us more adequate and interesting results than possible so far. The dataset used in the experiments was collected through a real life application called MovieLens[2] and mapped into the RDF format.

[2]GroupLens Research Project, University of Minnesota http://grouplens.org/datasets/movielens/. The queries were designed for the smallest dataset, as it contains both user and item features.

First, we will start with the RecSPARQL query in Fig. 7 that has no restriction on the neighborhood. It computes the degree of preference of each pair of users and items. Since the query is not parameterized, each user has a neighborhood whose size equals the size of the domain of users. Items will be recommended multiple times and recommendations that are based on a low similarity score are also included in the result. This adheres to RecSPARQL's philosophy: the customization of recommendation queries is left to the discretion of the query writer.

```
RECOMMEND ?user ?user.REC ?movie.REC
    ?SIMscore ?RATING USING CF
WHERE {
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie .
    ?personalRating ml:hasRating ?uRating }
BASED ON {
    ?user rdf:type recsparql:User .
    ?movie rdf:type recsparql:Item .
    ?uRating rdf:type recsparql:UserRating .
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie }
```

Figure 7: Query without restricted neighborhood

As next, we consider a set of three collaborative filtering queries. These queries gradually narrow the neighborhood of each user to produce more tailored recommendations. Hence, we do not compare each pair of users anymore, but include some restrictions. The first query $Q1$ in Fig. 8 filters the neighborhood for those users who have a similarity score above 0.5. For a collaborative approach this score is based on the movies watched by both users.

```
RECOMMEND ?user ?user.REC ?movie.REC
    ?SIMscore ?RATING USING CF
WHERE {
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie .
    ?personalRating ml:hasRating ?uRating .
    FILTER(xsd:double(?SIMscore) > 0.5 ) }
BASED ON {
    ?user rdf:type recsparql:User .
    ?movie rdf:type recsparql:Item .
    ?uRating rdf:type recsparql:UserRating .
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie }
```

Figure 8: RecSPARQL query $Q1$

Query $Q2$ in Fig. 9 furtherly narrows the user's neighborhood by only considering users 5 years younger or older. Finally, query $Q3$ in Fig. 10 adds a geographical constraint by applying a filter to the zipcode. In addition to this, some content-based user properties such as *age*, *gender* and *occupation* have been added to the BASED ON clause. In this way the similarity of users is now only partially based on the movies watched. Moreover, the query has been parametrized with the placeholder `%K%`, representing the age difference and `%L%`, representing the required proximity.

Fig. 11 compares the actual impact of the neighborhood restriction of Q1, Q2, and Q3(a) on their respective neighborhood size. We can observe that for the most restrictive query Q3(a) most of the users exhibit a rather small neighborhood. This can even end up having users without

```
RECOMMEND ?user ?age ?user.REC ?movie.REC
    ?age.REC ?SIMscore ?RATING USING CF
WHERE {
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie .
    ?personalRating ml:hasRating ?uRating .
    ?user ml:hasAge ?age .
    FILTER (
      abs(xsd:integer(?age) - xsd:integer(?age.REC)) <= 5) .
    FILTER(xsd:double(?SIMscore) > 0.5 ) }
BASED ON {
    ?user rdf:type recsparql:User .
    ?movie rdf:type recsparql:Item .
    ?uRating rdf:type recsparql:UserRating .
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie }
```

Figure 9: RecSPARQL $Q2$

```
RECOMMEND ?user ?age ?zip ?user.REC ?movie.REC
    ?age.REC ?zip.REC ?SIMscore ?RATING USING CF
WHERE {
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie .
    ?personalRating ml:hasRating ?uRating .
    ?user ml:hasAge ?age .
    ?user ml:hasZipCode ?zip .
    FILTER (
      abs(xsd:integer(?age) - xsd:integer(?age.REC)) <= %K%) .
    FILTER (
      abs(xsd:integer(?zip) - xsd:integer(?zip.REC)) <= %L%) .
    FILTER(xsd:double(?SIMscore) > 0.75 ) }
BASED ON {
    ?user rdf:type recsparql:User .
    ?movie rdf:type recsparql:Item .
    ?uRating rdf:type recsparql:UserRating .
    ?user ml:rates ?personalRating .
    ?personalRating ml:ratedMovie ?movie
    ?user ml:hasAge ?age .
    ?user ml:hasGender ?gender .
    ?user ml:hasOccupation ?occupation }
```

Parameters: (a) **K** = 5, **L** = 1000
(b) **K** = 10, **L** = 2000
(c) **K** = 20, **L** = 4000
(d) **K** = 40, **L** = 6000

Figure 10: RecSPARQL query $Q3$

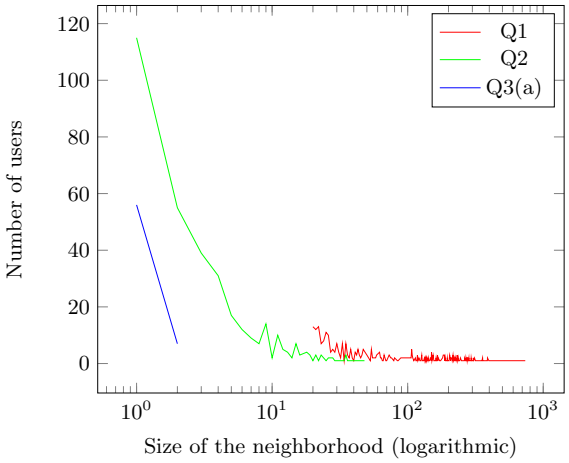

Figure 11: Number of users per neighborhood size

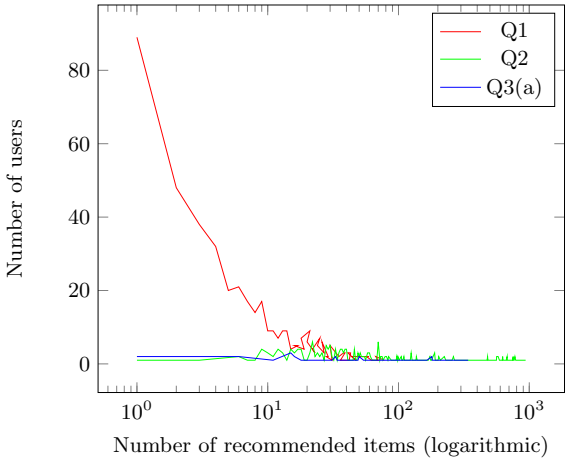

Figure 12: Number of users per number of recommended items

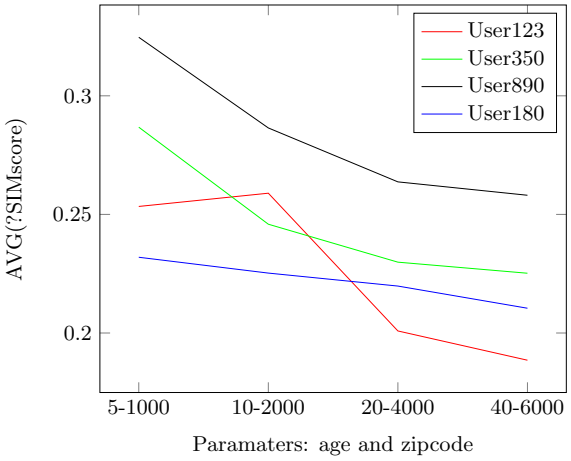

Figure 13: Average of similarity score for different parameters k and l and 4 users

	(K=10, L=2000)	(K=20, L=4000)	(K=40, L=6000)
∅rating	1.6199	1.8899	2.0268
∅similarity	0.2541	0.2285	0.2205

Table 1: Averaged movie ratings and similarity scores

a neighborhood in the worst case. It is not surprising to see that the less restrictive the queries are, the larger the neighborhood size is. Certainly, this influences also the number of recommended items per query as shown in Fig. 12. The more the neighborhood is restricted, the less recommendations are generated.

To illustrate the impact of narrowing the neighborhood more deeply and to understand its advantage, we proceed having a closer look on some user recommendations. Therefore, we chose four users who watched movies and assigned ratings to them. We evaluated Q3 (a-d) for those users and analyzed at first their averaged top movie ratings. For query Q3(a) only one of the four evaluated users got recommendations and the results are not comparable to the rest of the queries. For the remaining queries the results shown in Tab. 1 show that the more we restrict our neighborhood, the lower this average is. However, this is an expected behaviour since our specified neighborhood criteria does not consider the item rating itself, hence we are also removing potentially good rated movie recommendations. But this does not mean that the quality of the recommended movies is therefore reduced. Recommended movies with a lower rating might fit the needs of a user if they come from a more restricted neighborhood, e.g. the same country or the same age. This assumption is underpinned by the averaged similarity of a user and his neighbours as illustrated in Tab. 1. Actually, the more restrictive the neighborhood is, the more similar the people are. This means, although we are getting recommendations with a lower averaged rating, the recommended movies might fit better since they are derived from users with a higher similarity score.

7. RELATED WORK

To the best of our knowledge, RecSPARQL is the first extension of SPARQL that allows to customize recommendations. But there has been lot of work done dealing with semantic data for recommender systems. Policarpio et al. [17] presented a solution where SPARQL is used to extract data from RDF triplestores, which in turn is used as input for the recommender engine. But their approach is still not reaping the full benefits of a close integration of a semantic data representation with state-of-the-art recommendation algorithms. Other research efforts use public Semantic Web sources to enrich the contextual knowledge about items and users [5, 9, 13], ranging from music [14] and news recommendations [6] to the suggestion of relevant topics [20] or books [16]. However, such systems are developed to recommend a certain type of item and are often restricted to work with a fixed built-in ontology. In contrast, RecSPARQL comes with the flexibility for recommendations on arbitrary RDF graphs, as the recommender engine is customized within a RecSPARQL query. This includes not only the recommender technique but also the attributes that should be taken into account for computing similarities. Yet, there is not much work on such kind of flexible recommender systems that use a language to specify desired recommendations. Adomavicius et al. [2] introduced REQUEST, a recommendation language with support for so-called multidimensional queries. It extends the traditional two dimensional view between items and users with any other dimensions, e.g. time, while offering OLAP-type aggregation and filtering operations. However, REQUEST does not make use of Semantic Web data to improve the quality of recommendations. In [18], the authors presented a smooth integration of a recommender system into a DBMS called RecDB, where most computations are done in the DMBS layer instead the application layer. In contrast to our approach, the recommendation model is build once in advance and not created dynamically during query execution time. Although this approach has advantages in terms of query execution time, its much less customizable, since even small changes on the recommendation model, e.g. the considered attributes, require

a complete new loading phase. Further, RecDB is based on a highly structured relational data model as input and therefore incapable of capturing the diversity and flexibility of an RDF graph. Noteworthy related work has been also done in the area of personalized information retrieval techniques. [19, 22] make use of ontologies to represent profiles, which in turn are used to re-rank search results. This process can be also seen as some kind of automatized recommendation, since the list of suggested items is somehow aligned with contextual knowledge about users.

8. CONCLUSION

With RecSPARQL, we can demonstrate the benefits of a tight integration of recommender systems with SPARQL. Their smooth interplay enables freely customizable recommendations on arbitrary RDF graphs while taking advantage of rich knowledge sources of the semantic web. Moreover, if we consider the continuously increasing datasets in the semantic web, techniques for retrieving the most relevant information become indispensable. In future work, we will further enhance the integration of both paradigms to support more recommendations techniques and to increase the expressiveness of RecSPARQL, e.g. by means of subqueries as filters or property paths to express for more sophisticated connections between users, items and their features.

9. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Trans. Knowl. Data Eng.*, 17(6):734–749, 2005.

[2] G. Adomavicius, A. Tuzhilin, and R. Zheng. REQUEST: A Query Language for Customizing Recommendations. *Information Systems Research*, 22(1):99–117, 2011.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC*, pages 722–735, 2007.

[4] S. Auer and J. Lehmann. What Have Innsbruck and Leipzig in Common? Extracting Semantics from Wiki Content. In *ESWC*, pages 503–517, 2007.

[5] P. Bedi, H. Kaur, and S. Marwaha. Trust Based Recommender System for Semantic Web. In *IJCAI*, pages 2677–2682, 2007.

[6] I. Cantador, P. Castells, and A. Bellogín. An Enhanced Semantic Layer for Hybrid Recommender Systems: Application to News Recommendation. *Int. J. Semantic Web Inf. Syst.*, 7(1):44–78, 2011.

[7] A. F. Dietmar Jannach, Markus Zanker and G. Friedrich. Recommender Systems: An Introduction,press, 2011, 336 pages. isbn: 978-0-521-49336-9. *Int. J. Hum. Comput. Interaction*, 28(1):72–73, 2012.

[8] S. Harris and A. Seaborne. SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013, 2013.

[9] B. Heitmann and C. Hayes. Using Linked Data to Build Open, Collaborative Recommender Systems. In *AAAI Spring Symposium: Linked Data Meets A.I.*, 2010.

[10] T. Hornung, C.-N. Ziegler, S. Franz, M. Przyjaciel-Zablocki, A. Schätzle, and G. Lausen. Evaluating Hybrid Music Recommender Systems. In *WI*, pages 57–64, 2013.

[11] F. Manola, E. Miller, and B. McBride. RDF Primer. W3C Recom., 2004.

[12] B. N. Miller, I. Albert, S. K. Lam, J. A. Konstan, and J. Riedl. MovieLens unplugged: experiences with an occasionally connected recommender system. In *IUI*, pages 263–266, 2003.

[13] T. D. Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker. Linked open data to support content-based recommender systems. In *I-SEMANTICS*, pages 1–8, 2012.

[14] A. Passant. dbrec - music recommendations using dbpedia. In *International Semantic Web Conference (2)*, pages 209–224, 2010.

[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[16] L. Peska and P. Vojtás. Enhancing recommender system with linked open data. In *FQAS*, pages 483–494, 2013.

[17] S. Policarpio, S. Brunk, and G. Tummarello. Implementation of a SPARQL Integrated Recommendation Engine for Linked Data with Hybrid Capabilities. In *Artificial Intelligence meets the Web of Data (AImWD) Workshop, at ECAI*, Aug. 2012.

[18] M. Sarwat, J. Avery, and M. F. Mokbel. A recdb in action: Recommendation made easy in relational databases. *PVLDB*, 6(12):1242–1245, 2013.

[19] A. Sieg, B. Mobasher, and R. D. Burke. Ontological user profiles for representing context in web search. In *Web Intelligence/IAT Workshops*, pages 91–94, 2007.

[20] M. Stankovic, W. Breitfuss, and P. Laublet. Linked-data based suggestion of relevant topics. In *I-SEMANTICS*, pages 49–55, 2011.

[21] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *WWW*, pages 697–706, 2007.

[22] L. Zhuhadar and O. Nasraoui. Semantic Information Retrieval for Personalized E-Learning. In *ICTAI (1)*, pages 364–368, 2008.