Master's Thesis

ReSPARQL: a SPARQL Extension for Generic Recommendations on RDF-graphs

Victor Anthony Arrascue Ayala

February 2014

University of Freiburg

Department of Computer Science

Chair of Databases and Information Systems

Candidate

Victor Anthony Arrascue Ayala

Matr. number

3209050

Working period

26. 08. 2014 – 26. 02. 2014

Examiner

Prof. Dr. Georg Lausen

Supervisor

Martin Przyjaciel-Zablocki

Abstract

The abundance of data published using Semantic Web technologies ratifies their high degree of maturity reached. Moreover, the flexibility of the Resource Description Framework (RDF) enables it to model any knowledge within a specific domain. This has given rise to a potential use of RDF data as input for applications which were not originally designed to operate online on Web data sources.

Recommender systems are one example of such applications. These aim to predict the taste of a user towards a set of not consumed items and are typically well optimized for fixed domains. The benefit of having a recommender system which takes advantage of Web knowledge is that a user could be assisted in selecting information from the Web and, therefore, reducing the information overload. However, these systems cannot handle the diversity and unstructuredness of Semantic Web data. One of the reasons is that Semantic Web query languages, such as SPARQL, support retrieval of data exclusively based on facts; predictions or suggestions are entities that cannot be explicitly retrieved.

In this thesis ReSPARQL will be presented: an extension of the SPARQL syntax and semantics that fills this gap and enables a generic and flexible approach for recommendations over arbitrary RDF-graphs. It supports content-based and collaborative filtering recommendations and allows both paradigms to gain benefit from each other.

Kurzfassung

Die Fülle an Daten, die durch Semantic Web Technologien veröffentlicht wurden, bestätigt, dass diese Technologien sehr ausgereift sind. Die Flexibilität des Resource Description Frameworks (RDF) ermöglicht es darüber hinaus, jedes Wissen innerhalb eines spezifischen Anwendungsgebietes zu modellieren. Dies führt dazu, dass RDF-Daten als möglicher Input für Applikationen genutzt werden können, die ursprünglich nicht dafür konzipiert wurden, online Web Datenquellen zu verarbeiten.

Recommender Systeme sind ein Beispiel für diese Art von Applikationen. Sie haben zum Ziel, den Geschmack eines Verbrauchers in Bezug auf noch nicht konsumierte Objekte vorherzusagen, und sind typischerweise optimiert für feste Anwendungsgebiete. Der Vorteil eines Recommender Systems, das Kenntnisse aus dem Web nutzt, ist, zur Unterstützung des Verbrauchers Informationen aus dem Web auszuwählen, um die Informationsflut einzuschränken. Diese Systeme können jedoch nicht mit der Unterschiedlichkeit und Unstrukturiertheit von Semantic Web Daten umgehen. Einer der Gründe dafür ist, dass Semantic Web Abfragesprachen, so wie SPARQL, nur eine Datenerfassung unterstützen, die ausschließlich auf Fakten basiert. Vorhersagen und Vorschläge sind dagegen Datensätze, die nicht explizit abgerufen werden können.

In dieser Arbeit wird ReSPARQL vorgestellt, eine Erweiterung der SPARQL-Syntax und -Semantik, die diese Lücke füllt und einen allgemeinen und flexiblen Ansatz für Empfehlungen über beliebige RDF-Graphen ermöglicht. ReSPARQL unterstützt content-based und collaborative filtering Empfehlungen und ermöglicht es, dass beide Paradigmen voneinander profitieren.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SPARQL [15] and recommender systems [13] are two paradigms with different characteristics. On the one hand, SPARQL is one of the technology pillars of the Semantic Web vision [11]. It is both a protocol and declarative query language, which allows users and applications to interact with ontologies and RDF-data. Today, with more than 19 billion triples[1], the amount of linked data confirms the success of the Semantic Web initiative.

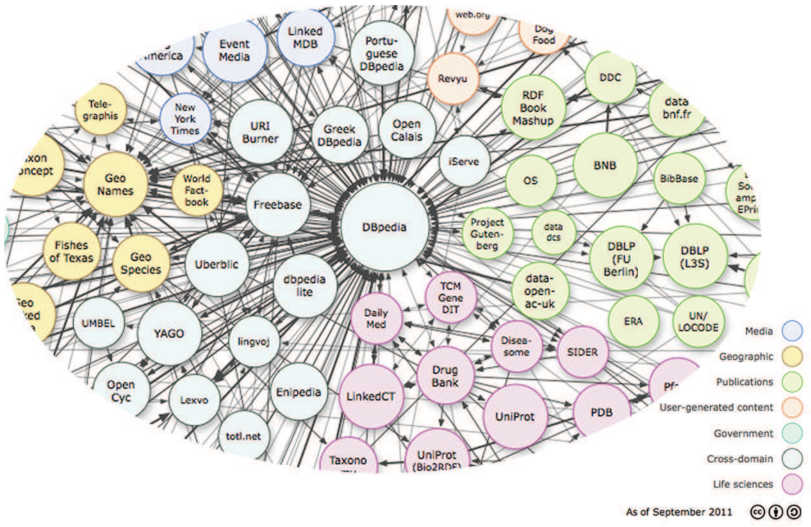


Figure 1.1: Fragment of the Linked Open Data visualization by Richard Cyganiak and Anja Jentzsch.

[1]http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics

SPARQL 1.1 and extensions like SPARQL 1.1 Federated Query[2] introduced mechanisms for executing queries distributed over different SPARQL endpoints. This enables us to combine information from different semantic knowledge bases, e.g. DBPedia [9] or YAGO [33].

Despite the many new useful features introduced in SPARQL 1.1 questions like "What kind of movie might a user be interested in?" cannot be answered, because SPARQL has been designed to find explicit patterns.

On the other hand, recommender systems provide suggestions that reflect users' interests based on their preferences, traits or interactions with other users [5]. The problem has been addressed since the mid-1990s and recommender systems made significant progress in specific domains like music [17, 22], movies [19], or books [24].

A recommender system on top of RDF-data would be beneficial for many reasons. First of all, RDF-data has a heterogeneous and ubiquitous structure, and it is already structured in an ontological way [16, 20]. This allows us to obtain a vast diversity of recommendations which goes beyond the classic user-item paradigm. Secondly, the lower layers of the Semantic Web like RDF and RDFS have reached maturity and the amount of interlinked data published in accessible databases [6, 20, 29] gives way to cross-domain recommendations. Finally, it is possible to improve the quality of the recommendations by eventually filling information gaps.

The question arises how both processing paradigms can be combined. One simple approach consists in combining both recommender systems and SPARQL, where the latter is used for preprocessing the recommender's input data. The recommender system is in this architecture a middle layer component responsible for computing similarities between objects, neighborhoods, etc. The computed information is in turn added to the original data source making it possible to use conventional query languages. One example of this is the approach presented by Policarpio et al. [25]. However, such integration is superficial and, as will be shown, it imposes limits to the recommender's flexibility.

The approach presented here consists in designing a query language to fill this gap. ReSPARQL[3] is an extension of SPARQL which produces sug-

[2] `http://www.w3.org/TR/sparql11-federated-query/`

[3] Recommener engine SPARQL Protocol and RDF Query Language

gestions directly from Semantic Web data sources in a flexible and generic way, being the first of its kind that offers a tight integration with the Semantic Web vision and the possibility of specifying highly parameterizable queries.

The advantages of having a query language like ReSPARQL are self-evident. To illustrate the utility, let's consider the following use-case: Suppose a developer maintains an RDF-store concerning a social network and is requested to implement a recommender system able to produce recommendations of different kinds:

Target of recommendations	**Recommended items**	**Criteria**
Users	Advertisement	Based on user preferences, origin, activities, etc.
Users	Friends (other users)	Based on similar characteristics.
Advertisement	Target audience (group of users)	Based on advertised product and user preferences.

Table 1.1: ReSPARQL's motivational example

The time and resources needed to implement such a system from scratch are significant. ReSPARQL queries, as will be shown, are expressive enough to cover the above mentioned cases and many more. Furthermore, a framework capable of evaluating ReSPARQL queries permits to obtain recommendations without having to design and implement new components.

1.1 Initial goal and contributions

The initial goal was to define a query language with a degree of expressiveness similar to REQUEST [6] while having the ability to operate on top of Semantic Web data. After acquiring a deep understanding of both Semantic Web and recommender systems, it became clear that it would be beneficial to design a query language that retrieves data like SPARQL, i.e. by means of graph pattern matching, but which could surpass its expressiveness.

In addition, the implementation of a framework able to execute ReSPARQL queries was also an aim. A recommender system was initially proposed based on the work made by Kämpgen et al. [18], i.e. a system on which queries are mapped to OLAP operations. Finally, the project went more and more in the direction of the Semantic Web because of the clear benefits in regards to the integration.

The main contributions of this master thesis can be summarized as follows. The first goal consisted in designing ReSPARQL as an extension of SPARQL in compliance with the principles of its specification. ReSPARQL is defined as a new query form (like *SELECT*, *CONSTRUCT*, *ASK* and *DESCRIBE*). An extension brings into the language all benefits of SPARQL, e.g. the fact that it is not restricted to a pre-defined and fixed ontology. ReSPARQL queries can be used to obtain both content-based and collaborative filtering recommendations. Hybrid recommendations are also partially supported. The second goal was to define formal semantics of ReSPARQL based on the SPARQL algebra. New algebra operators are defined to achieve this. Finally, the third goal consisted in implementing a standalone recommender repository packaged as an extension of Sesame's Sail API[4] which makes it possible to evaluate recommendation queries. This last goal required deep understanding of compiler design, architecture of an RDF store, etc. As a result, the recommender system can be integrated into a Java project just as Sesame, by simply pointing to the correct dependencies or by adding the libraries.

1.2 Thesis outline

The thesis is organized as follows: Chapter 2 presents the fundamentals of both Semantic Web and Recommender Systems, which are necessary to understand the approach of ReSPARQL. At the end of this chapter the reader will find the related works.

Chapter 3 is divided into two main parts: The first one presents the syntax of the query language through examples, whereas the latter focuses on the formalized semantics. For better comprehension, this part starts with an introduction of the SPARQL algebra and its operators.

[4]Sesame is an open source Java framework for storing, querying, and reasoning with RDF and RDF Schema. It can be used as a database for RDF and RDF Schema, or as a Java library for applications that need to work with RDF. `http://www.openrdf.org`

Chapter 4 depicts the architecture of the ReSPARQL recommender repository. The first section briefly explains the architecture of Sesame, followed by an overview of the repository's architecture and a more detailed description of the main components.

Finally, conclusions and future work are presented in chapter 5.

Chapter 2

Preliminaries

ReSPARQL is an extension of SPARQL and therefore, it also queries against RDF datasets, being these last two W3C recommendations. The first section provides an introduction to the Semantic Web and hence gives an overview of all related technologies with a special focus on RDF and SPARQL.

The second section presents the fundamentals of recommender systems: common techniques to compute similarities and predict suggestions. This covers two classic approaches, content-based filtering and collaborative filtering and provides basic notions of hybrid implementations.

The purpose of the third section, recommendations on top of RDF graphs, is to present an intuitive analysis about the patterns of which a recommender system could make use for computing recommendations.

The fourth section presents the related work. Different approaches have been proposed to obtain recommendations from RDF data. We will see how these differ from ReSPARQL.

2.1 The Semantic Web

The World Wide Web was conceived as a system of interlinked documents without a specific purpose. Additional layers were designed on top to permit us to request documents and present them to end-users. This flexibility led to a vast number of both client and server side web-development techniques and to a rapid increase of the system's size[1]. In this scenario search engines were an important tool to retrieve documents in the system with advanced keyword matching and rank algorithms.

However, during the gradual transition to the Web 2.0 the way users communicate, and the way information is diffused and used to provide services have also changed. Computers became over time, together with their traditional role of computing calculations and processing information, entry points of information [21]. In this new scenario users require answers to more complex questions. However, this is not possible under our current paradigm. To illustrate this, suppose one enters the following (paradox) query[2] in Google's search box:

```
Example of semantic queries Google cannot answer to
```

The top 5 pages for this query are illustrated in figure 2.1. None of the returned pages provides an answer to that question[3].

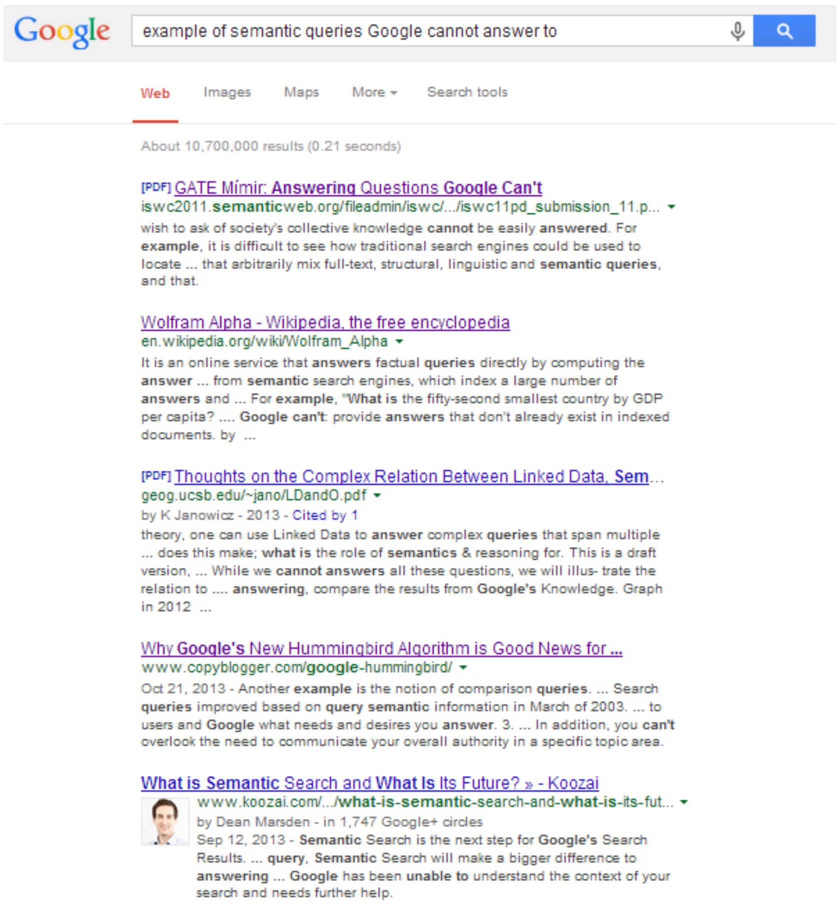
This overtakes the classic "no recall" problem of search-engines. The problem lies in the lack of understanding of the real semantic of the query. Even for queries with a reasonable set of returned pages is not the search engine providing users with answers, but users who have to browse the returned pages and to extract the information they need in order to answer to their own questions.

The biggest limitation to provide better answers is that the current model of interlinked documents doesn't contain additional meta-data that allows building a knowledge base that could be in turn used to interpret sentences and infer useful information from it.

[1]According to `www.worldwidewebsize.com` the Indexed Web contains at least 1.63 billion pages (queried on Wednesday, 08_{th} of January, 2014).

[2]These results were retrieved the 28th of December, 2013. In September 2013 Google launched a new search algorithm called Hummingbird, which partially supports semantic search. However, for most of the queries, the classic Google algorithm is still used.

[3]The top result is a scientific article about a GATE MIMIR, an information repository not directly related to Google. The paper has two instances of the word "Google". One is located in the title, "Answering questions Google can't", and the second one is in the abstract: "[...] and search engines such as Google [...]" The remaining four pages are related to semantic search, or to the new Google Hummingbird's algorithm, which can partially answering semantic queries, but none of the pages contains an answer to the original question.



example of semantic queries Google cannot answer to

About 10,700,000 results (0.21 seconds)

[PDF] GATE Mímir: **Answering** Questions **Google Can't**
iswc2011.**semantic**web.org/fileadmin/iswc/.../iswc11pd_submission_11.p...
wish to ask of society's collective knowledge **cannot** be easily **answered**. For **example**, it is difficult to see how traditional search engines could be used to locate ... that arbitrarily mix full-text, structural, linguistic and **semantic queries**, and that.

Wolfram Alpha - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Wolfram_Alpha
It is an online service that **answers** factual **queries** directly by computing the **answer** ... from **semantic** search engines, which index a large number of **answers** and ... For **example**, "**What is** the fifty-second smallest country by GDP per capita? .... **Google can't**: provide **answers** that don't already exist in indexed documents. by ...

[PDF] Thoughts on the Complex Relation Between Linked Data, **Sem**...
geog.ucsb.edu/~jano/LDandO.pdf
by K Janowicz - 2013 - Cited by 1
theory, one can use Linked Data to **answer** complex **queries** that span multiple ... does this make; **what is** the role of **semantics** & reasoning for. This is a draft version, ... While we **cannot answers** all these questions, we will illus- trate the relation to .... **answering**, compare the results from **Google's** Knowledge. Graph in 2012 ...

Why **Google's** New Hummingbird Algorithm is Good News for **...**
www.copyblogger.com/**google**-hummingbird/
Oct 21, 2013 - Another **example** is the notion of comparison **queries**. ... Search **queries** improved based on **query semantic** information in March of 2003. ... to users and **Google** what needs and desires you **answer**. 3. ... In addition, you **can't** overlook the need to communicate your overall authority in a specific topic area.

What is Semantic Search and **What Is** Its Future? » - Koozai
www.koozai.com/.../**what-is-semantic**-search-and-**what-is**-its-fut...
by Dean Marsden - in 1,747 Google+ circles
Sep 12, 2013 - **Semantic** Search is the next step for **Google's** Search Results. ... **query**, **Semantic** Search will make a bigger difference to **answering** ... **Google** has been **unable to** understand the context of your search and needs further help.

Figure 2.1: Top five results of query "Example of semantic queries Google cannot answer to".

To improve this further, one approach consists in representing web content in a machine-processable form, creating scope for new algorithms. The Semantic Web is referred to as the initiative of gradually transforming the existing Web into a distributed and decentralized database of knowledge by adding semantic content to web documents. The idea is not to replace all existing documents, but to add semantic content that describes the resources within the documents. The movement is led by the World Wide Web Consortium (W3C) and Tim Berners-Lee, one of the Web pioneers, is one active contributor.

2.1.1 The Semantic Web Stack

Therefore, the Semantic Web required a new architecture in order to reengineer the WWW. The Semantic Web Stack is a layered architecture, but neither has it been the only one proposed nor it has stopped to evolve. The layers, which are built up one on top of another, illustrate which technologies have been standardized to make the Semantic Web possible.

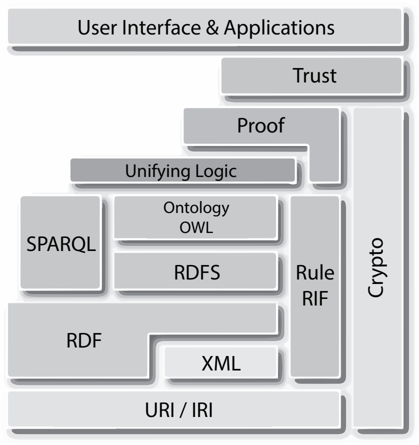


Figure 2.2: Illustration of the Semantic Web Stack[4]

In the remaining part of this section I will provide a brief description of the layers on which ReSPARQL is based.

At the bottom we find Uniform Resource Identifier (URI) and Internationalized Resource Identifier (IRI). If one wants to describe a resource, e.g. a movie, a mechanism to identify the resource is needed. An identifier also makes it possible to reference the resource across the web. For instance, if a remote knowledge database has already described all properties of a movie, there's no reason why this should be done again. URI provides a mechanism to uniquely identify resources on the web. This does not imply that the resource can be retrieved, but only identified. IRIs have the same role, but use a different character set, UNICODE, to allow users to create identifiers in all possible languages.

XML is a markup language. Although this layer is not directly used in ReSPARQL, it is by RDF. XML user-defined markups allow writing struc-

[4]Source: `http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/`

tured Web documents, easily understandable and usable by humans. XML allows us to integrate together content and tag-based information that describes the content. It is also a format for sending documents across the Web. However, XML is not expressive enough to "give" semantic to documents.

Resource Description Framework (RDF) is the Semantic Web's subjacent model that allows us to create databases of knowledge starting from small units, i.e. simple statements about Web resources. A resource is an object, an entity, anything that can be described. A statement is the basic knowledge block; it asserts a property of an object using three elements, subject-predicate-object, and for this reason is called triple. An RDF statement is similar to a statement in a natural language. For instance, this statement says something about a famous actor, our resource.

Marlon Brando	is an	actor.

A very intuitive graphic representation of a set of triples is a directed graph with labeled nodes and arcs directed from the resource (or subject) to the value (or object). RDF has an XML-based syntax (it is not the only syntax available) and therefore, it is positioned on top of the XML layer.

Ideally each web document will contain links between resources and its descriptors for easy retrieval[5]. By doing so a search engine would not only find the name of an actor in a document, but would also be able to access the description and hence to know that the resource is actually an actor and hence a person, etc., having a set of properties.

Sometimes to describe resources one has to abstract and find the relationship between classes of resources. For instance, a movie is a film production made by a producer with a certain budget. The resulting hierarchy is in this context called vocabulary or ontology. Since RDF doesn't make any assumptions about a particular domain, if one wants to define a specific vocabulary it is necessary to create a schema by means of RDF Schema (RDFS). This semantic information is also machine-accessible.

Actor	is a subclass of	movie staff.

[5]A proposed technology to achieve this is RDFa `www.w3.org/TR/rdfa-syntax/`.

This statement expresses that each instance of an actor is also a member of the movie staff. RDFS' derivation rules like the one above make it possible to derive new implicit statements, e.g. that Marlon Brando is indeed a member of a movie staff. RDFS provides a set of modelling primitives that permits to organize Web objects hierarchically, but it is not limited to this: application-specific knowledge is also possible.

SPARQL Query Language for RDF[6] is the recommended declarative query language for writing queries against RDF datasets. It also defines a protocol for accessing RDF Data. It is based on graph pattern matching. The basic block to build a graph pattern is a triple pattern, which is akin to an RDF triple. The difference resides in the fact that it's possible to replace one or more RDF terms, subject, predicate and/or object, with a variable. Moreover, it is possible to combine triple patterns to form more complex graph patterns and to apply filter conditions.

2.1.2 Resource Description Framework

According to the W3C (World Wide Web Consortium) Semantic Web Activity Statement, "*the Resource Description Framework (RDF) is a language designed to support the Semantic Web, in much the same way that HTML is the language that helped initiate the original Web. RDF is a framework for supporting resource description, or metadata (data about data), for the Web. RDF provides common structures that can be used for interoperable XML data exchange*" [27].

Indeed, RDF provides a mechanism to add machine-processable statements about resources. This goes around three key concepts, which will be here briefly explained: resources, properties and statements.

Resources, properties and statements

One might think of a resource as an object in the widest possible way: an entity that belongs to a certain domain and one wants to talk about. Every resource is then uniquely identified by an URI.

Properties describe relations between resources and are treated in RDF as a special kind of resources and therefore, are also identified by URIs.

[6]`http://www.w3.org/TR/rdf-sparql-query/`

The fact that URIs are used to identify both resources and properties is of strategic importance: it is possible to build a knowledge base by reusing resources somewhere defined and this allows us to get benefit also from their semantics.

Statements or RDF Triples are the basic building block to assert properties of resources, e.g. an attribute, a relationship, a characteristic, etc. It is a 3-tuple made of three units of information. This is referred as subject-predicate-object[7], because of its resemblance to its counterpart in linguistics. Indeed, in order to assert a property in natural language these three elements compose the required single unit. Subjects correspond to the above defined resources, and predicates correspond to properties. The object is the value of the subject's property type.

For instance, the following statement "The song Unfaithful is interpreted by Rihanna" can be expressed in RDF as follows:

```
http://www.resparql.org/lastfm#Unfaithful
http://www.resparql/resparql/lastfm#interpreted
http://www.resparql.org/lastfm#Rihanna
```

In RDF it is also possible to assert something, e.g. an opinion about a statement: "Beyonce thinks that the song Unfaithful is interpreted by Rihanna". This kind of properties in which one expresses belief or trust in another statement might be useful in some applications and is known as *reification*.

RDF Graph

The RDF graph is the default method for describing RDF data models. There are three kinds of nodes: URIref (which uses an URI identifier), blank nodes and literals. By convention, URIs are shown in ellipses and literals are enclosed in rectangles.

Nodes, for which not having an identifier is reasonable or for which this is unknown, are called blank or anonymous nodes and have local scope (i.e. they cannot be referenced outside of the graph they belong to).

A literal, on the other hand, consists of three parts: a character string, an optional language and data type like integer or date (the type is identified

[7]In some literature this is known as object-attribute-value.

by an URI). For instance, one could use a typed literal to describe Rihanna's age as being the integer number 25:

```
http://www.resparql.org/lastfm#Rihanna
http://www.resparql.org/lastfm#age
"25"^http://www.w3.org/2001/XMLSchema#integer
```

As the example shows the data typing scheme used is part of the XML Schema specification, which predefines a large range of data types including Booleans, integers, floating-point numbers, times, and dates.

The following figure illustrates a small RDF graph of a movie dataset designed within the scope of my thesis:

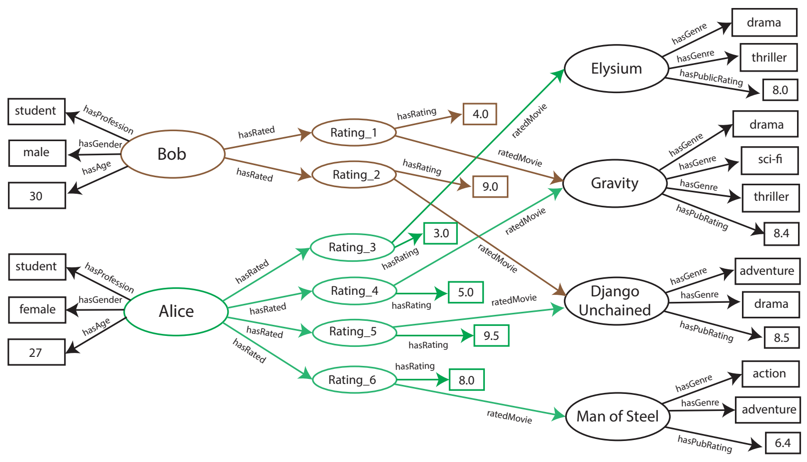


Figure 2.3: Movies dataset, reduced version.

Serialization

Along with RDF/XML, there are other formats in which RDF graphs can be serialized. In Notation3 or N3 syntax the subject, predicate and object are separated by spaces, and the triple is terminated with a period (.). To avoid writing the complete URI all the time, it is possible to declare a @prefix ⟨QName⟩ ⟨URI⟩, where QName is a qualified name, i.e. a name that follows a strict set of rules. For instance:

```
@prefix lastfm:<http://www.resparql.org/lastfm#> .
lastfm:Unfaithful lastfm:interpreted lastfm:Rihanna .
lastfm:Unfaithful lastfm:trackDuration "228000" .
```

N3 provides more abbreviation mechanisms such as replacing the subject with ";" to indicate that the subject is the same as in the previous triple, or "," to replace both subject and predicate.

N-triples is another serialization format, similar to N3, but without of most abbreviations shortcuts and therefore more verbose.

Turtle (Terse RDF Triple Language) is the kind of serialization used in ReSPARQL. Turtle's syntax is a subset of N3: it renounces to expressiveness power by removing the first-order-logic features of N3 in favor of simplicity. An RDF Dataset serialized in Turtle is also valid in N3 format.

2.1.3 RDF Schema

In order to add semantic content to our data sometimes it's necessary to specify a vocabulary. RDF does not define the semantics of any domain nor make assumptions about any particular application domain, so it is up to the user to define one.

In the remaining part of this section I will provide an overview of the constructs that can be used to describe a particular domain.

Class and Properties

The first step consists in defining the objects one wants to talk about. For example, in a context of movies one might want to talk about movies, actors, directors, etc., not about specific instances as in RDF, but about classes. A class in this sense is something similar to a class in a conventional programming language.

In RDFS it is also possible to restrict properties by specifying a range, i.e. the classes of resources and values to which the property applies. Top-level classes are rdfs:Class, rdf:Property, rdfs:Resource, rdfs:Literal, rdf:Statement. In RDF the predicate rdf:type is used to relate instances to the classes these belong to.

Hierarchies

A TV show and a movie are both film productions. But there might be properties that can be applied exclusively to the first and not to the second, e.g. seasons or number of episodes, or vice versa. As in classic programming languages it is possible to model a hierarchy of classes by means of inheritance. Hierarchy is not a concept exclusive for classes. In RDFS is also possible to define hierarchical relationships between properties. In order to define subclasses and subproperties these are some of the predicates that can be used: rdfs:subclassOf, rdfs:subPropertyOf.

In some cases the concept of inheritance is not precise enough to describe the nature of an object. Assume one wants to model a movie cast. This is clearly defined by the members of the cast, but an actor himself is not (a subclass of) a cast. To define a group of objects in RDFS one can use RDF containers or RDF collections. A not exhaustive list of predicates is: rdf:Alt, rdf:Seq, rdf:Bag, rdf:nil, rdfs:List, rdfs:Container.

2.1.4 SPARQL Protocol and RDF Query Language

SPARQL is the query language recommended by the W3C to query against an RDF dataset. The specification of SPARQL consists of three parts:

- SPARQL Query Language for RDF
- SPARQL Query Results XML Format
- SPARQL Protocol for RDF

The easiest example of a SPARQL query is probably the following:

```
(... PROLOG)
SELECT * WHERE { ?s ?p ?o }
```

This query retrieves a copy of the entire dataset with an SQL-like syntax. The query starts with a PROLOG. This allows us to define prefixes for namespaces, for instance[8]:

```
BASE <.../dataset.rdfs>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

The keyword *BASE* allows us to declare a base URI, which references a local vocabulary. Its syntax SPARQL is also similar to N3 and shares some of its abbreviation mechanisms. *PREFIX* is one of them.

SPARQL is based on graph pattern matching. In the previous example ?s ?p ?o corresponds to a triple where each of the RDF terms was replaced

[8]In this and the next examples, we skip the PROLOG, because our examples are not vocabulary-dependent.

with a variable. Therefore, any triple could fulfill the pattern. Suppose, for example, that instead of that triple pattern one writes:

```
SELECT * WHERE { ?s rdf:type rdfs:Class . }
```

This query retrieves all classes.

The *SELECT* clause returns a projection of a subset of the variables used in the *WHERE* clause. This can be followed by an optional clause *FROM*, which specifies the location, inclusive of protocol, of the graph source against which the query will be evaluated. If specified, this is the first clause evaluated. Otherwise, it is assumed that the query is evaluated in some specific local scope, e.g. within the bounds of a triple store. The clause *WHERE* allows us to define the set of graph patterns that one wants to retrieve. Additionally, one can specify filter conditions to filter out information from the result.

It is important to mention, that a query that uses projection doesn't return a valid RDF graph, which is one the key design features. Since it is possible to project an arbitrary number of variables, the result can be seen as a set of tuples (mappings from variables to values) and therefore materialized as a table where each column corresponds to a variable.
SELECT is not the only kind of query supported by SPARQL:

- *ASK*: returns true if there is at least one match in the graph, else false.
- *CONSTRUCT*: returns an RDF graph. It can be a subgraph (of the original RDF graph) or a newly constructed graph. This makes the language closed[9], which is a desirable property.
- *DESCRIBE*: returns an RDF Graph that describes those resources matched by the graph pattern of the query. It is up to the SPARQL's implementation to choose what triples are used to describe a resource. The specification states that, "*the DESCRIBE form returns a single result RDF graph containing RDF data about resources. [...] The description is determined by the query service.*"

Some of the limitations of SPARQL 1.0 are its inability of updating operations (however, this is supported as an extension), the absence of cursors

[9]The result of the query is of the same kind of the input.

to iterate over results, the lack of full-text search, etc. Moreover, some RDF models were not supported such as RDF collections. With regards to inference, SPARQL 1.0 provided only a guideline of how this should be accomplished:

"*The overall SPARQL design can be used for queries which assume a more elaborate form of entailment than simple entailment, by re-writing the matching conditions for basic graph patterns. Since it is an open research problem to state such conditions in a single general form which applies to all forms of entailment and optimally eliminates needless or inappropriate redundancy, this document only gives necessary conditions which any such solution should satisfy. These will need to be extended to full definitions for each particular case.*" [28].

On March 2013 SPARQL 1.1 [15] became a W3C recommendation. This adds many new features including the possibility of doing updates, of using sub-queries, property paths, negation, aggregation, federation, (partial) entailment and much more.

SPARQL also specifies a protocol to query a specific graph. An example of the how the protocol works is the following[10]:

```
http://.../qps?
query-lang=http://www.w3.org/TR/rdf-sparql-query/
&graph-id=http://resparql.org/lastfm.rdf
&query=PREFIX foaf: <http://xmlns.com/foaf/0.1/..
```

2.2 Recommender System

The utility of recommender systems nowadays goes beyond its role of a sales tool on e-commerce systems for which it was conceived. Today, every time a Web user has to decide which products to buy, which news to read, which places to sightsee, etc., he is overloaded with information, which makes the process of taking the decision frustrating.

Recommender systems could help to reduce this complexity by filtering out non-relevant information. Information based on users' preferences could be consequently highlighted for easy retrieval. Therefore, it would be beneficial to integrate recommender systems as an online tool into the Web and especially into the Semantic Web, but this has been achieved only partially.

[10]Source: `http://www.dajobe.org/talks/200603-sparql-stanford/`

The remaining part of this section introduces the method of operation of classic recommender systems. In order to produce suggestions a recommender system should have an approach to solve each of the following problems:

- how to extract user preferences;
- how to extract features of items;
- how to represent the above mentioned features and compute similarities;
- how to compute recommendations.

In the remaining of this section typical approaches will be described.

User preferences

Typically, a recommender system collects some information about user preferences and has to deduce some other. These can be classified as follows:

- user's personal information, e.g. local origin, spoken languages, age, gender, education, etc;
- explicit ratings: a function $r : (u, i) \to v \in \mathbb{R}$ where u is a user, i is an item and v is a numeric value that represents a degree of preference. It is called explicit, because the user is required to (optionally) provide the system with this information. The range and interpretation of this value depends on the system, e.g. negative values could be used to express dislike, we could have discrete or continue values, etc.;
- implicit ratings: deduced degree of preference gathered from observations of user behavior within the scope of the system, e.g. purchase history, navigation history, time a user spends searching for a product, etc. This kind of information might not be totally reliable, because e.g. two or more users might have shared the same account or a user might have forgotten to close an opened browser's tab.

The system could prioritize the kind of information obtained and use conveniently one kind in absence of other.

Item features

The features of a product are sometimes given, e.g. written by the manufacturer; sometimes the system has to deduce them. In some literature a set of features is called a profile.

For instance, a profile of a movie could consist of the following features:

1. cast: set of actors in the movie;
2. director;
3. year of production;
4. genre;

Sometimes an item has only one value for a given feature, like budget or date of premiere. Sometimes it has multiple values, e.g. a movie cast.

In order to determine the features of an item when these are not given, classic Information Retrieval techniques are used. For instance, if a specification which describes a product is available, representative terms (words) are selected and used as features. Here is a typical approach [5]:

- remove stop words from the document, e.g. "the", "a", "for", etc., which carry only small information about the product;
- compute weighting measure

  $$w_{i,j} = TF_{i,j} * IDF_i, \tag{2.1}$$

  where t_i is a term appearing in document d_j, $TF_{i,j} = \frac{f_{i,j}}{max_z fz, j}$ is a normalized term frequency ($f_{i,j}$ is the frequency of t_i in document d_j) and $IDF_i = log\frac{N}{n_i}$ is the inverse document frequency (N is the overall number of documents and n_i is the number of documents in which the term t_i appears);
- use the k terms with the highest score as the representative terms. One could fix k for all items or make it proportional to the document length.

Another classic technique consists in using classifier algorithms, e.g. Naive Bayes, to perform a soft clustering, i.e. each item is assigned to one or more classes. In this case each class is seen as a feature's value, e.g. "drama movie". Naive Bayes is a semi-supervised algorithm, i.e. it needs a training set in order to predict the class or classes to which an item belongs.

Naive Bayes' probabilistic model assumes that each term in the document is independent from all other terms in a document, but this is clearly unrealistic. For instance, the description of a smartphone could have words like "battery life" or "full HD" in which one can easily see that these terms are dependent in probabilistic terms.

Note that these techniques are not limited to find item features, but can also be applied to retrieve user traits or user preferences[11] as well.

Representation of features and similarities

Since one of the core problems within a recommender system consists in computing similarities between two items or users, the representation must lead to an efficient computation. Let $SIM_{feat=\{f_1,f_2,...,f_n\}}(obj_i, obj_j)$ be a function that computes the similarity of two objects, obj_i and obj_j, given a set of features $\{f_1, f_2, ..., f_n\}$. This function returns a score which is proportional to the degree of similarity of the two objects. For instance, the next function computes the similarity between two movies based on their genres:

$$SIM_{feat=\{genre\}}(movie1, movie2)$$

There exists some approaches in classic recommender systems which return a similarity score for one feature:

- Jaccard distance: each object has its own set of features values, e.g. object A = {red, white, blue}, object B = {red, green}. The similarity is calculated with the following formula:

$$d_{jaccard} = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{2.2}$$

- cosine distance: each object has its own vector of feature values. A and B are now two vectors, $A = \langle red, white, blue \rangle$ and $B = \langle red, green \rangle$. The formula

$$\cos ang(v_1, v_2) = \bullet(v_1, v_2) \ / \ \|v_1\|_2.\|v_2\|_2, \tag{2.3}$$

[11]For instance, using a description field from a blog or a social network.

where $\bullet$ is the dot product between two vectors and $\|v_i\|_2$ is the $l2-norm$, returns a value between 0 and 1.

- Pearson correlation: it is similar to cosine distance but measures the degree to which a linear relationship exists between two variables.

In ReSPARQL cosine distance is used to compute similarities. A concrete example: suppose one wants to compute the similarity between the movies "The Expendables" and "Escape Plan" based on their cast, $SIM_{feat=\{cast\}}(\text{"}TheExpendables\text{"}, \text{"}EscapePlan\text{"})$. Suppose we have the following vector of actors:

0	1	2	3	4	5	6	7	8	...
Sylvester Stallone	Arnold Schwarz.	Jason Statham	Jet Li	Faran Tahir	Amy Ryan	Sam Neil	Dolph Lundgren	Bruce Willis	... ...

We represent the two movies as follows[12]:
The Expendables ($v1$):

0	1	2	3	4	5	6	7	8	...
1	1	1	1	0	0	0	1	1	...

Escape Plan ($v2$):

0	1	2	3	4	5	6	7	8	...
1	1	0	0	1	1	1	0	0	...

Each cell contains a 1 at index i, if the actor at index i appears in the movie, 0 if otherwise. The length of each vector is equivalent to the overall number of actors, which is inconvenient. Therefore, in practice this representation is avoided for space reasons; instead, a compact representation, e.g. sparse matrix, is used where cells having 0s as values are omitted.

$$\cos ang(v_1, v_2) = \bullet(v_1, v_2)\ /\ \|v_1\|_2.\|v_2\|_2 = \frac{2}{\sqrt{6^2 + 5^2}}$$

In the example above, vectors were filled with binary values. But this is not the only possibility. Positive integers representing the frequency of values can be also used. For instance, let's consider a vector that represents the movies a user has seen. A frequency value, which represents the number of times a user has seen a movie, is used instead of binary values.

[12]Values from 9 to n are all 0

0	1	2	3	4	5	6	7	8	...
The Expendables	Escape plan	The Godfather	The Hobbit	... ...	... ...	... ...	... ...	... ...	... ...

User1 ($v1$):

0	1	2	3	4	5	6	7	8	...
1	0	4	1	0	0	0	0	1	...

User2 ($v2$):

0	1	2	3	4	5	6	7	8	...
0	0	2	0	0	0	1	0	0	...

User1 has seen "The Godfather" 4 times whereas User2 has seen it only 2 times. In this case the cosine distance returns a fair similarity score as well.

Our vector could also contain a third kind of values. Suppose that the system not only knows the movies that a user has watched, but it has also collected explicit user ratings. Calculating the similarity as above would be inaccurate: two users might have watched the same movies but assigned different ratings to them. Fortunately, cosine distance can be also applied to vectors weighted by the rating value. For example if User1 saw the movie 1 and gave it a rating of 5, then the vector has a 5 at index 1.

When vectors are weighted, values are typically rounded or normalized to produce more accurate results. For instance, normalization could be done by subtracting the average rating of a user from each rating he does. This transforms low ratings into negative numbers whereas high ratings remain positive numbers.

There are other possible representations, but these depend on the system and go beyond the scope of this work.

Computation of recommendations

Recommender systems are usually classified according to the technique used to predict ratings. Independently of the kind of information collected and available to the system, there are three classical approaches: content-based filtering, collaborative filtering and hybrid approaches. Moreover, each approach can be subsequently classified into heuristic-based and model-based. Since ReSPARQL is based on a heuristic approach, model-based approaches won't be covered here. They consist in building a probabilistic model from the underlying data using statistical learning and machine learning techniques. Predicted ratings are, under this paradigm, probabilities of users liking items.

2.2.1 Content-Based Filtering

Content-Based Filtering focuses on properties of items. Suggestions are searched among the domain of items. Each user gets suggestions based on how similar an item is to an item that user has already consumed and rated. The degree of preference depends on the degree of similarity between two items.

A heuristic-based approach could consist for instance in using the above described similarity function that takes two items as input and a set of features (a subset of the item's profile) and returns a value representing the degree of similarity (a value from 0 to 1). This method could not be accurate enough if we only consider a consumption relationship between user and item without taking the ratings into account. Otherwise, it could happen that an item is recommended to a user, because this item is similar to an item the user consumed but disliked. Whenever ratings are available they should be used to provide more accurate suggestions.

If the features were obtained from description texts, e.g. by means of TF-IDF, we can use cosine distance to predict the missing ratings. More formally, let $Prof(i) = \{w_{f_1}, w_{f_2}, ..., w_{f_n}\}$ be an item profile, obtained by means of TF-IDF (w is a weighting function and f_i is a term that describes a feature). Analogously, let $CBProf(u) = \{w_{f_1}, w_{f_2}, ..., w_{f_m}\}$ be the profile of user u that describes the degree of preference of features f_i. Then the predicted rating $r : (u, i) \rightarrow v \in \mathbb{R}$ can be computed using cosine distance as follows:

$$r(u,i) = \cos ang(\overrightarrow{w}_{Prof(i)}, \overrightarrow{w}_{CBProf(u)}) \tag{2.4}$$

Content-based filtering is feasible only when the features are already available in the system or if extraction of items' features is possible. In some domains it might be difficult to obtain and to extract features automatically, e.g. for multimedia data. Another limitation of content-based systems is that they cannot recommend items that are different from anything the user has consumed before (overspecialization). This also has an impact on diversity and range of recommendations. Finally, new users cannot get accurate recommendations due to the small number of ratings provided.

2.2.2 Collaborative filtering

Collaborative filtering focuses on the relationship between users and items. We will distinguish two kinds of CF approaches: user-based and item-based collaborative filtering. In general, CF algorithms operate on a set of users $U = \{u_1, u_2, ..., u_n\}$, a set of items $I = \{i_1, i_2, ..., i_m\}$, and partial rating functions $r : (u, i) \rightarrow v \in \mathbb{R}$, where $u \in U$ and $i \in I$. The range of ratings depends on the system.

A user-based collaborative filtering utilizes the ratings that users assign to items in order to find a peer of like-minded users for each user, called neighborhood. Within this neighborhood highly rated items can be recommended. Two users are considered to be similar if they by trend consume the same items and rate them in the same way.

The predicted rating $r(u, i)$ is computed taking into account other ratings in the neighborhood. Let N_u be the neighborhood of user u. The two most common formulas are:

$$r(u,i) = c \sum_{u' \in N_u} sim(u,u') * r(u',i) \tag{2.5}$$

$$r(u,i) = \mathbf{avg}(r(u)) + c \sum_{u' \in N_u} sim(u,u') * (r(u',i) - \mathbf{avg}(r(u'))) \tag{2.6}$$

Both formulas have a normalization constant c, typically:

$$c = \frac{1}{\sum_{u' \in N_u} sim(u,u')} \tag{2.7}$$

Moreover, $avg(r(u))$ is the average of ratings of user u. To calculate the similarity between users, $sim(u, u')$, the two most common approaches are cosine distance and Pearson correlation, but these are not the only option. The first formula is much simpler, but it has a limitation. Suppose that user u rates all movies with ratings 1, 2, 3, 4, ... and that a second user u' rates all with 2, 3, 4, 5, ... This could be simply due a different interpretation of the scale of values. The second formula gives better similarity scores in this case.

An item-based collaborative filtering is simply the symmetrical approach, i.e., we compute the neighborhood of items. The predicted ratings can be computed as follows [13]:

$$r(u,i) = c \sum_{i' \in N_i} sim(i',i) * r(u,i')$$

$$c = \frac{1}{\sum_{i' \in N_i} sim(i,i')}$$

CF recommenders suffer from the so called "cold problem". When a system is freshly started users have not consumed any item yet and no ratings are available. The problem persists when new users and items are added into the system. Another problem CF recommenders have is that a high number of users in the system is required in order to produce good quality recommendations.

2.2.3 Hybrid recommender systems

Hybrid recommender systems combine multiple recommendation techniques, e.g. both approaches, content-based and collaborative filtering, to produce recommendations. As mentioned in the previous sections, both approaches have weaknesses and provide better results under certain circumstances. Hybridization approaches can be implemented in many ways:

- independently calculating recommendations using the two approaches individually and then combine the predicted ratings, e.g. using a linear function or similar;
- implementing a collaborative approach that uses not only ratings but also content-based profiles of the user;
- using collaborative approach to soft-cluster users and then applying a content-based approach to each group;
- implementing a semantic that supports both, content-based and collaborative filtering.

2.3 Recommendations on top of RDF graphs

In this section we will show that RDF graphs are a feasible input for recommender systems. We will present a short analysis of the kind of patterns required for computing recommendations through a set of representative RDF schemas. ReSPARQL does not require any schema, but, as we will see, in order to produce recommendations the recommender system must be instructed about the role of nodes. See chapter 3 for more details.

User and items

In order to provide recommendations, two classes of nodes have to be identified in the RDF graph: users and items. If these are missing, it is not possible to compute recommendations. In practice, any class of nodes could potentially assume both roles. Even one class of nodes having both roles could be used, e.g. nodes representing persons could be used to recommend friends. Moreover, there must be a path of length > 0 connecting the user nodes to the item nodes. Let this path be the user-item path. This represents the user's consumption or a user's preference for an item. Without this path both content-based recommendations and collaborative filtering would not be feasible.

An example of a user-item path of length 1 and 2:

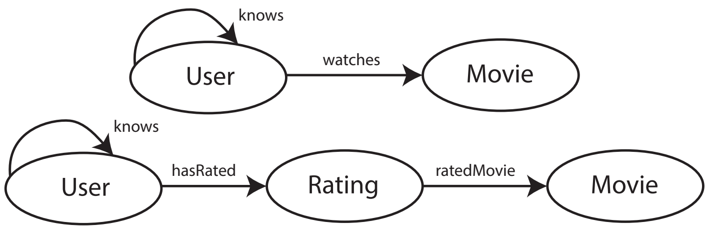


Figure 2.4: RDFS, user-item path of length 1 and 2.

It might also be possible to have multiple user-item paths, but not all of these necessarily represent a positive preference. Let's consider, for instance, the following graph:

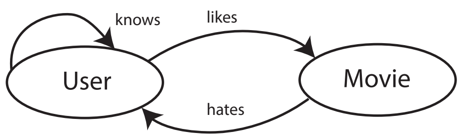


Figure 2.5: RDFS, two path between user and item.

Unless one wants to recommend movies, which users could particularly dislike with a content-based approach (which could also be the case), the path “likes” should be used instead of the path “hates”.

On the one hand, a content-based approach requires that item nodes have features. These have to be represented in the RDF graph as a path from the item nodes to the feature nodes, being the length of this path > 0. The same applies to users’ features.

On the other hand, a collaborative approach requires ratings given by users that describe the degree of preference towards an item. The rating will be typically represented in RDF Graphs as a literal node. Because this kind of rating is related to both user and items, in ReSPARQL this is called user-rating node.

The user-rating nodes have to be reachable from the user nodes and this path has to have at least one edge in common with the user-item path[13].

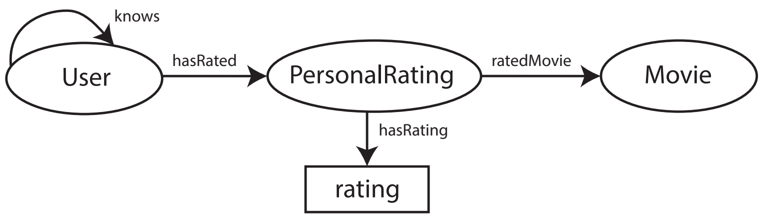


Figure 2.6: RDFS, users’ ratings.

The user-rating node is not the only kind of measure an item within an RDF graph could have. Sometimes an item could have been rated not by a single user, but by some entity representing a group of users. For instance, a movie in Rottentomatoes[14] has the so-called Tomatometer, which

[13]This implies that rating nodes are possible only when the user-item path is > 1.
[14]`www.rottentomatoes.com`

shows the percentage of approved critics who have given the movie a positive review. Aggregated user-ratings already available in the system belong to this category, too. These kinds of ratings are called in ReSPARQL item-ratings.

Item-ratings are represented by nodes reachable from the item nodes. Moreover, the user-item path and the path from the item nodes to the item-rating nodes must not have any edge in common.

Contextual information is information that usually describes the context in which a user consumed a given product. In the context of movies, a user could have seen a movie during the weekend, with friends, etc. This kind of information is important to provide richer and more precise recommendations. This is located "between" the user and item nodes in the RDF graph, i.e. the user-item path and the path from the user to the contextual nodes have to share some edges:

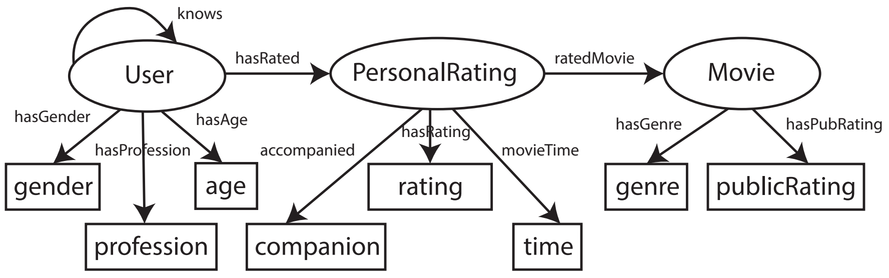


Figure 2.7: RDFS, movies dataset, time and companion are contextual nodes.

To conclude, a recommendation system on top of RDF requires:

1. user nodes representing users;
2. item nodes representing items;
3. the user-item path between these two entities;

Moreover, other nodes contained in the patterns mentioned above could be used to compute recommendations or/and improve its accuracy:

1. feature nodes of items and users;
2. user-rating nodes or item-rating nodes;

3. contextual nodes.

2.4 Related work

The idea of integrating semantic data models with recommender systems imposes non-trivial challenges. These can be divided into two categories: expressiveness and design of the system.

With regard to the first problem, it is highly desirable to have a language capable of querying any kind of recommendations against any possible RDF graph. However, recommendations are predictions and SPARQL has shown itself to be incapable of expressing "imprecise" queries. This problem has already been disclosed in areas such as Information Retrieval, where expressing advanced rankings over semantic metadata, e.g. rankings based on user preferences, is subject to the same problem [26]. Some proposed solutions consist in extending RDF query languages, e.g. RDQL [12] or SPARQL [31].

The second problem is related to the kind of architecture needed to compute recommendations. In order to process big amounts of Semantic Web data, typical approaches consist in bringing this data into other systems like relational database systems [34, 30], OLAP-based systems [18], or Map-Reduce systems [7]. However, implementing a recommender system in the mapped model imposes limits on its flexibility and results in hard-coded components difficult to reuse or adapt.

The idea of extending SPARQL for customizing recommendations is, to the best of my knowledge, a novel approach. The remaining part of this section will present the related work.

2.4.1 Recommendation engines for Linked Data

There is a wide spectrum of proposals of recommender systems tailored for Linked Data[15].

For instance, Policarpio et al. [25] designed a recommender engine with the ability of computing collaborative filtering, content-based and hybrid recommendations based on Linked Data. This approach uses RDF as the

[15]A project initially proposed by Tim Berners-Lee for large scale integration of data on the Web [1]. Currently, this is constituted by a vast set of interlinked RDF open datasets on the Web. This currently consists of over 19 billion of triples [2].

input format and is able to augment the original data by creating and adding further RDF triples, the recommendations computed, which can then be reused to refine new recommendations.

Policarpio's approach uses SPARQL as the query language to extract data from RDF triplestore endpoints, i.e. its use is limited to information extraction. The obtained result is then pre-processed and given as an input to the recommendation computation module. This system was developed assuming a distributed model (Hadoop / Map Reduce) and therefore has the ability to work with large volumes of data. Mapper and Reducer of the Hadoop's job are dynamically generated after data retrieval. Two are the approaches supported by the recommender system: item-based collaborative filtering and content-based. Hybrid recommendations rely on the fact that previous recommendations are stored in the data itself and therefore it is possible to combine the results obtained.

The computation module is fixed within the system. For instance, in case of collaborative filtering the module requires the following set of mappings (user, movie, rating). The SPARQL query must project the variables in exactly that order. For content-based recommendations the set of mappings must have the items, to which the similarity function is applied, as the first projected variable followed by the rest of features.

Although this approach achieves a good degree of integration, it still has the inconvenient problem of using the computation module as a black box. The module generates a job dynamically according to the kind of recommendation algorithm (CB or CF filtering), but no more tuning possibilities are offered. For instance, if one wants to change the way the neighborhood of a user or item is computed, the module has to be reimplemented.

Retrieval and integration of data from multiple sources is not a trivial problem. Therefore, some approaches focus on data acquisition [16], whereas recommendations are computed with classic models, like vector space model [20].

Other approaches use an ontology based model to enhance the semantics of the recommender system, e.g. to enhance contextual information [10], to represent the semantic distances between objects [22], or to simply represent the data acquired [14].

However, in almost all cases recommender systems are fixed within a specific domain, ranging from music [22], books [24], movies [20], news [14] or even suggestions of topics [32].

2.4.2 Special-purpose recommendation languages

Adomavicius et al. [4] proposed a novel special-purpose recommendation language called REQUEST[16], which accomplishes a multidimensional level of expressiveness. REQUEST queries against a multidimensional model based on the OLAP-paradigm to obtain recommendations and, therefore, it supports OLAP-like aggregation and filtering operations. Certainly, REQUEST does not to exploit the knowledge of semantic data sources, but it was inspirational for designing ReSPARQL.

Adomavicius was the first to propose treating recommender systems as inherent multidimensional entities. Indeed, the traditional two-dimensional user/item paradigm is insufficient for "Context-Rich" applications:

"[. . .] when recommending vacations to travelers, one would likely recommend a different vacation to a customer in the winter than in the summer, i.e., the time-of-travel context is clearly important when making recommendations. Similarly, when recommending groceries, a "smart" shopping cart (Wade 2003) needs to take into account not only information about products and customers, but also such information as shopping date/time, store, who accompanies the primary shopper, products already placed into the shopping cart, and its location in the store. Clearly, the two-dimensional paradigm of classical recommender systems is less suitable for these applications." [6]

The following example shows a multidimensional REQUEST query:

```
RECOMMEND Movie, Time TO User, Companion
USING MovieRecommender
RESTRICT User.Name="Tom" AND
Time.TimeOfWeek="weekend" AND
Companion.Type="Girlfriend"
BASED ON PersonalRating
SHOW TOP 3
```

The support of multidimensional queries is a desirable property for recommender systems. The keyword USING specifies the cube to be used as input.

[16]REQUEST is the acronym for REcommendation QUEry Statements.

It is assumed that the ratings cube is fully pre-computed before users start issuing recommendation queries.

This concludes the preliminaries.

Chapter 3

ReSPARQL's specification

In order to provide an overview about the syntax and semantics of ReSPARQL, a small dataset was designed based on the schema of fig. 3.1, here reported again for the sake of completeness. Additionally, the formalized semantics are presented in section 3.3.

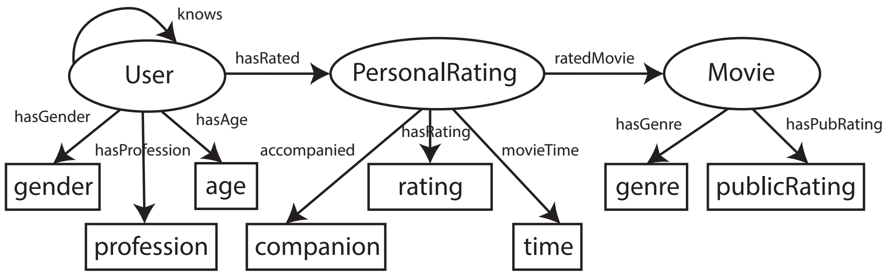


Figure 3.1: RDFS of movies dataset (same as fig. 2.7).

3.1 Example: movies dataset

Intuitively, an RDF that complies with the schema in fig. 3.1 depicts a scenario where users watch movies and rate them. This data, as seen in fig. 3.2, contains knowledge about three users and five movies.

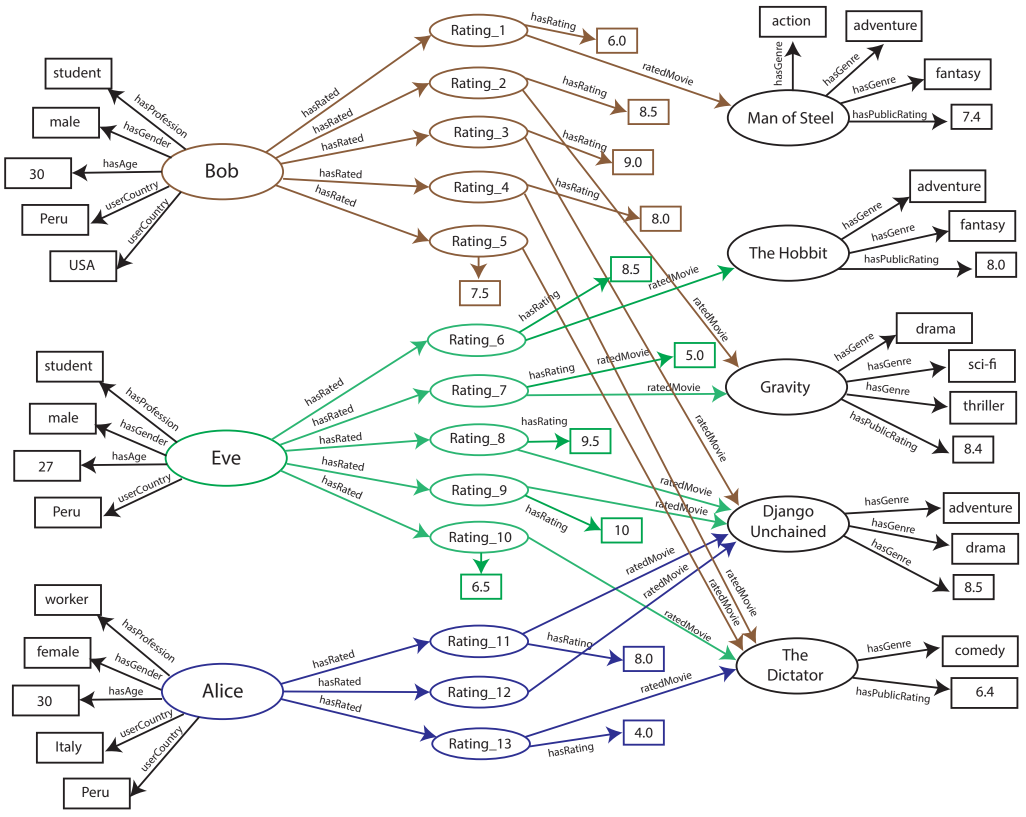


Figure 3.2: Movies dataset

3.2 Syntax and Semantics through examples

The syntax of ReSPARQL was designed in conformance to the principles stated in the specification of SPARQL 1.1. The designing goals were based on two pillars:

- keeping the language as close as possible to SPARQL, i.e. the query language should be able to retrieve data by matching graph patterns against RDF graphs;
- being intuitive enough to make the recommendation process transparent to the user;

In this section the syntax of ReSPARQL and the approach for computing the recommendations will be presented through a set of examples.

3.2.1 Basics

The basic example of a content-based ReSPARQL query is the following:

```
1  PREFIX resparql: <http://example.org/resparql#>
2  PREFIX movies: <http://examples.org/movies/>
3  RECOMMEND ?user ?movie.REC ?RATING
4  WHERE {
5      ?user rdf:type resparql:User .
6      ?movie rdf:type resparql:Item .
7      ?user movies:hasRated ?personalRating .
8      ?personalRating movies:ratedMovie ?movie .
9  }
10 BASED ON {
11     ?movie movies:hasGenre ?genre
12 }
```

Table 3.1: ReSPARQL content-based query that recommends movies similar in their genres to movies that the user has already watched.

The purpose of this query is to recommend movies similar in their genres to movies that the user has already watched (*RECOMMEND ?user ?movie.REC*).

Just as SPARQL, ReSPARQL starts with a prolog (lines 1 and 2) which allows a user to declare base URIs or namespaces prefixes. Another clause both languages have in common is *WHERE*. This allows specifying the pattern to be matched against the RDF graph. The resulting matched information is the input of the recommender system. In the example above we have four triple patterns (lines 5-8). However, in contrast to SPARQL the triple patterns in lines 5 and 6 are discarded in the process of building the graph pattern and therefore, these are not matched.

ReSPARQL does not rely on RDFS or meta-data describing the RDF graph and therefore, it is necessary to instruct the system about the roles of variables in the query. More concretely, we have to let the system know which variable represents users and which one represents items we aim to recommend. This can be expressed in the *ReSPARQL Type Pattern (RTP)*, a special graph pattern within the *WHERE* clause. In our example the triple patterns in lines 5 and 6 form the *RTP* and generally triple patterns to which a type within the “resparql” namespace[1] is assigned by means of the property *rdf:type*. In the example above *?movie* of type *resparql:Item* is the sought recommended item whereas *?user* of type *resparql:User* represents the users. The mechanism implemented by the *RTP* makes possible to avoid solutions applied in other recommender systems such as [25] where otherwise the order of variables is fixed, e.g. the first projected variable represents the users, the second the items, etc.

WHERE is followed by a ReSPARQL-specific clause, *BASED ON*. Despite the resemblance with *WHERE*, the purpose of *BASED ON* is to instruct the recommender about how to accomplish its task. The triple patterns herein defined, specify the set of attributes that the system has to take into account to compute the recommendations. Altogether this basic graph pattern is called *ReSPARQL System Pattern (RSP)*.

In the example above, line 11, the only triple pattern specified, represents the attribute “genre” of a movie. Since the roles of variables have been specified in the *RTP*, the system is now able to understand that *?genre* is a feature of a movie. To represent a single attribute one or more of these (concatenated) triple patterns could be needed. A set of triple patterns representing one feature is called *Feature Graph pattern (FGP)*. Let TP_B be a triple pattern from the *BASED ON* clause. The following property holds:

$$TP_B \subseteq FPG \subseteq RSP$$

Each *FGP* has to start with either a user variable, e.g. *?user*, or an item variable, e.g. *?item*. One query cannot contain *FGPs* that refer to both users and items. Depending on the kind of *FGPs* provided the recommender system performs either a user-based recommendation or an item-based recommendation. Moreover, the systems automatically decides to process the data using a content-based or collaborative filtering approach. Given the *FGP* specified in the example, an (item) content-based approach will be used to recommend movies.

In ReSPARQL, there are two different types of variables. The first type corresponds to variables used within the *WHERE* clause. Like in SPARQL these can be projected using the *RECOMMEND* clause, which has almost the same role as *SELECT*, i.e. projection. In our example *?user*, *?person-*

[1]The URL `http://example.org/resparql#` is used only for illustrative purposes.

alRating and *?movie* belong to this category. The second type of variables are ReSPARQL-specific. It consists of renamed variables and reserved score variables. On the one hand, there's a corresponding renamed variable for each variable within the *WHERE* clause, which has the same name as the original variable with the string .REC appended to the end. These renamed variables represent the recommended items and information directly linked to them in the RDF-graph. Now it should be clear what the difference between *?movie* and *?movie.REC* is: the first one represents movies that a user has actually seen, whereas *?movie.REC* represents potential recommended movies. On the other hand, ReSPARQL has two reserved score variables, *?SIMscore* and *?RATING*. *?SIMscore*, represents the computed similarity value of two objects, whereas *?RATING*, which gives insight to the predicted rating of a recommendation.

The computation of similarities is central for understanding the semantics of ReSPARQL, so it is worth to take a closer look. In section 2.2 a similarity function was defined. In our example, we are interested in finding similarities between pair of movies, for instance in computing the score:

$$SIM_{feat=\{genre\}}(Man\ of\ Steel, The\ Hobbit)$$

?SIMscore is the variable to which the similarity score is mapped. In the movies dataset there are five movies and each movie can have more than one genre. As the table 3.2 shows, each movie is represented as a vector of genres[2] and in order to compute similarities the cosine distance is used:

	action	**advent.**	**fantasy**	**drama**	**sci-fi**	**thriller**	**comedy**
Man of Steel	1	1	1	0	0	0	0
The Hobbit	0	1	1	0	0	0	0
Gravity	0	0	0	1	1	1	0
Django	0	1	0	1	0	0	0
The Dictator	0	0	0	0	0	0	1

Table 3.2: Movies represented as vectors of genres.

Cosine distance returns a value between 0 and 1. Now we know that:

[2]One task of the recommender system is to build this vectors.

$$SIM_{feat=\{genre\}}(Man\ of\ Steel, The\ Hobbit) = 0.816 = \mu(?SIMscore)^3$$

If more than one feature is provided, this process is repeated for each feature and then the final similarity score is averaged.

$$SIM_{feat=\{f_1,f_2,\ldots,f_n\}}(obj_i, obj_j) = \frac{\sum_{k=1}^{n} SIM_{feat=\{f_k\}}(obj_i, obj_j)}{n}$$

Note that all objects matched at least once for one of the features will be considered. Similarities are computed for all possible pairs and hence a cross product is required. Independently of the number of features considered, the *BASED ON* produces a solution mapping of the form:

?movie	**?movie.REC**	**?SIMscore**
Man of Steel	The Hobbit	0.816...
...	...	...

At this point the input data retrieved by evaluating the graph pattern in the *WHERE* clause is joined twice with the solution mapping above, once on *?movie* and once on *?movie.REC* (the above described renaming is applied). This process is better illustrated in the following figure:

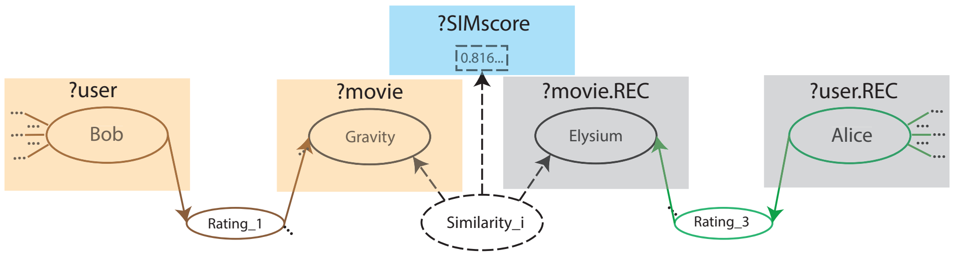


Figure 3.3: Joins between input data and two similar movies in query of table 3.1.

The dashed nodes are not part of the data, but they were added to depict the idea of the process of computing similarities. The figure shows also the renaming strategy of variables adopted.

The resulting set of mappings before the projection is illustrated in table 3.3.

[3] $\mu(v)$, where v is a variable, is the value mapped to v. This notation is used in SPARQL algebra, as will be shown in section 3.3

?user	**?movie**	**?movie.REC**	**?SIMscore**	**?user.REC**
Bob	Man of Steel	The Hobbit	0.816...	Eve
		Gravity	0	Bob
				Eve
		Django	0.408...	Bob
				Eve
				Eve
				Alice
				Alice
		The Dictator	0	Eve
				Alice
Bob	Gravity	Man of Steel	0	Bob
		The Hobbit	0	Eve
		Django	0.408...	Bob
				Eve
				Eve
				Alice
				Alice
		The Dictator	0	Bob
				Bob
				Eve
				Alice
Bob	Django	Man of Steel	0.408...	Bob
		The Hobbit	0.5	Eve
		Gravity	0.408...	Bob
				Eve
		The Dictator	0	Bob
				Bob
				Eve
				Alice
Bob	The Dictator	...	...	...
Bob	...	...	...	...
Eve	...	...	...	...

Table 3.3: Solution mappings for query in table 3.1 previous to the projection.

As table 3.3 shows, for only five movies the user “Bob” has gotten many results due to the cross product between items and further joins. However, this complexity is manageable. There's a way to reduce the size of the mappings and this can be achieved by applying filters. For instance, the following filter states that we are interested in movies whose similarity score is at least 0.5:

```
FILTER (xsd:double(?SIMscore) >= 0.5)
```

The resulting set of mappings after applying the filter is the following:

?user	**?movie**	**?movie.REC**	**?SIMscore**	**?user.REC**
Bob	Man Of Steel	The Hobbit	0.816...	Eve
Bob	Django	The Hobbit	0.5 ...	Eve

Table 3.4: Filtered solution mappings for query in table 3.1.

Further ways of reducing both the size of input data and the neighborhood of users/items can be found in section 3.2.7.

In regard to the computation of *?RATING*, the variable to which the predicted rating is mapped, is not shown in the previous tables, further details will be provided in section 3.2.5 about how to compute this value. *?RATING* is a linear function that takes into account *?SIMscore*, but also other measure parameters if these are specified in the system. In this example *?RATING* simply coincides with *?SIMscore*.

The fact that an item is recommended multiple times is an expected and reasonable behavior of recommender systems. In table 3.4 the movie "The Hobbit" is recommended twice because of its similarity to two other movies "Man Of Steel" and "Django". These results can then be packed in different ways. For instance, one could project *?user*, *?movie* and *?movie.REC* to show the reason for which a movie was recommended (to which movie *?movie.REC* is similar). Another approach could be to project a unique pair (user, recommended item) taking into account only the maximum predicted rating obtained. This can be achieved by means of GROUP BY and aggregations functions, as we will see in 3.2.6.

The semantics for (user) content-based recommendations are quite similar. The requirement is that, as explained above, all *FGPs* refer to users. Let's consider for instance the following query:

```
1   PREFIX resparql: <http://example.org/resparql#>
2   PREFIX movies: <http://examples.org/movies/>
3   RECOMMEND ?user ?movie.REC ?RATING
4   WHERE {
5       ?user rdf:type resparql:User .
6       ?movie rdf:type resparql:Item .
7       ?user movies:hasRated ?personalRating .
8       ?personalRating movies:ratedMovie ?movie .
9   }
10  BASED ON {
11      ?user movies:userCountry ?country
12  }
```

Table 3.5: ReSPARQL content-based query that recommends movies from other users similar in their origin.

The purpose of the query is to obtain recommended movies from other users similar in their origin. The origin is in this case represented by the *FGP* in line 11. We are considering here a content-based property of the user's profile and therefore, the focus of the similarity computation changes. As before, we can imagine each user is represented as a vector of countries:

	Peru	USA	Italy
Bob	1	1	0
Eve	1	0	0
Alice	0	0	1

Table 3.6: Users represented as vectors of countries.

The intermediate results are again used to join the input data.

?user	**?user.REC**	**?SIMscore**
Bob	Eve	0.707...
...	...	...

The following figure illustrates how information is joined for a pairs of compared users. By joining the input data with the intermediate similarity mappings, each user is linked to the movies of a similar user:

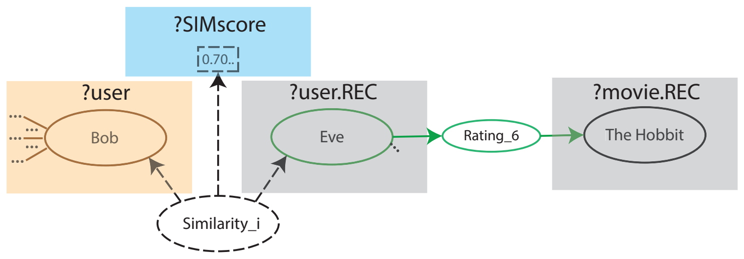


Figure 3.4: Joins between input data and two similar users in query of table 3.5.

As before, ReSPARQL computes all possible results unless filters are specified within the query. The set of mappings is defined over the same set of variables, however, the mappings are evidently different.

In this kind of queries an item a user has consumed, e.g. *?movie*, doesn't really play a role in a (user) content-based recommendation approach, but this information is kept until the end in case filters are applied to it. Also in this case the predicted rating, *?RATING*, coincides with *?SIMscore*.

The table 3.7 represents the solution mappings previous to the projection.

?movie	**?user**	**?user.REC**	**?SIMscore**	**?movie.REC**
Man of Steel	Bob	Eve	0.70...	The hobbit
				Gravity
				Django
				The Dictator
		Alice	0.5	Django
				Django
				The Dictator
...	...	...	...	...

Table 3.7: Solution mappings for query in table 3.5 previous to the projection.

3.2.2 Improvement of the predicted ratings

In order to improve the accuracy of recommendations it is possible to specify ratings of different kinds in ReSPARQL by means of *RTP*. As seen in section 2.3, two kinds of measure functions are considered in this work:

- user-ratings: explicit ratings that users give to items.
- item-ratings: public available ratings for items, given by an external entity.

The *MEASURES* clause allows us to specify the nodes that represent those ratings and to match the graph patterns containing that information, which is in turn given as input to the recommender system.

The following outline illustrates the use of this clause:

```
PREFIX resparql: <http://example.org/resparql#>
RECOMMEND ?user ?item.REC ...
WHERE { ... }
BASED ON { ... }
MEASURES {
    ?userRating rdf:type resparql:UserRating .
    ?itemRating rdf:type resparql:ItemRating .
     #exemplary graph patterns
    ?personalRating [...]:hasRating ?userRating .
    ?item [...]:hasPublicRating ?itemRating .
}
```

Table 3.8: Outline of a ReSPARQL query that uses *MEASURES*.

As in *WHERE*, the *RTP* in the *MEASURES* clause instructs the recommender about the roles of the variables *?userRating* and *?itemRating*: *resparql:UserRating* is the ReSPARQL class to assign a user-rating role, whereas the *resparql:ItemRating* corresponds to item-ratings. Neither *RTP* in *MEASURES* is matched against the RDF graph.

The remaining triple patterns in *MEASURES* are joined with those declared in the *WHERE* clause and for each variable therein declared there is also a correspondent renamed variable (.REC). In section 3.2.5 we will see how this metrics are used to compute the ratings.

3.2.3 Content-based and collaborative filtering

Thus far two examples of content-based queries, a user-based and an item-based query, were presented in tables 3.1 and 3.5. Whenever *FGPs* are provided, which are exclusively related to properties of users and items (profiles), the recommender system uses a content-based approach. For a collaborative approach it is instead required that one of the *FGPs* represents the user-item path (see section 2.3 for its definition). The following is an example of a collaborative filtering query based on the movies dataset:

```
1   PREFIX resparql: <http://example.org/resparql#>
2   PREFIX movies: <http://examples.org/movies/>
3   RECOMMEND ?user ?movie.REC ?RATING
4   WHERE {
5       ?user rdf:type resparql:User .
6       ?movie rdf:type resparql:Item .
7       ?user movies:hasRated ?personalRating .
8       ?personalRating movies:ratedMovie ?movie .
9   }
10  BASED ON {
11      ?user movies:hasRated ?personalRating .
12      ?personalRating movies:ratedMovie ?movie .
13  }
```

Table 3.9: ReSPARQL collaborative filtering query that recommends movies.

The purpose of the query is to recommend movies from like-minded users, who tend to watch the same movies. The *FGP* in lines 11 and 12 represents the user-item path. Certainly, the symmetrical case is allowed too: a path from item nodes to user nodes can be also provided and this results in an item-based collaborative filtering[4].

The query in table 3.9 shows also the difference between *WHERE* and *BASED ON*. Although both clauses contain the same triple pattern (lines 7-8, respectively lines 11-12), the first one is used to match the input data, whereas the second one, the *RSP*, specifies the criteria used to find similar

[4]Our dataset does not contain such a path and therefore item-based CF queries are not possible. This is one limitation of the language. In the section future work, 5.1, will be discuss a possible approach to solve this problem.

users. In this concrete example the system computes the similarity of users represented as vectors of the movies they watched. The vectors are filled with frequency values, which in this case represents the number of times a user watched a certain movie. If one wants to consider also the ratings that users assigned to movies, the following information has to be provided:

- an *FGP* containing the path from user nodes to user-rating nodes.
- a variable of type *resparql:UserRating*;

This information is respectively provided in lines 13-14 and line 17 of the following query:

```
1  PREFIX resparql: <http://example.org/resparql#>
2  PREFIX movies: <http://examples.org/movies/>
3  RECOMMEND ?user ?movie.REC ?RATING
4  WHERE {
5      ?user rdf:type resparql:User .
6      ?movie rdf:type resparql:Item .
7      ?user movies:hasRated ?personalRating .
8      ?personalRating movies:ratedMovie ?movie .
9  }
10 BASED ON {
       #FGP 1
11     ?user movies:hasRated ?personalRating .
12     ?personalRating movies:ratedMovie ?movie .
       #FGP 2
13     ?user movies:hasRated ?personalRating .
14     ?personalRating movies:hasRating ?userRating .
15 }
16 MEASURES {
17     ?userRating rdf:type resparql:UserRating .
18     ?personalRating movies:hasRating ?userRating
19 }
```

Table 3.10: ReSPARQL collaborative filtering query that recommends movies using ratings.

The purpose of the query is now to recommend movies from like-minded users, who tend to watch the same movies and to rate them in a similar way. With the new information the system can weight the vectors by multiplying the ratings to the frequency values. Cosine distance is also applied in this case. Note that both the content-based and the collaborative approaches use the same cosine measure. However, in the content-based case, it is used to measure the similarity between vectors of feature values, whereas in the collaborative case, it measures the similarity between vectors of the actual user-specified ratings.

3.2.4 Hybrid approach

In ReSPARQL it is possible to incorporate some content-based characteristics into the collaborative filtering approach by considering information from users' profiles. By doing so, ratings would then not be the only similarity criteria considered, and therefore the similarity score would be averaged over all considered features. Consequently, better results are obtained with respect to a pure collaborative approach in cases in which only a few pairs of users have a significant number of commonly rated items (sparsity problem [5]).

```
PREFIX resparql: <http://example.org/resparql#>
PREFIX movies: <http://examples.org/movies/>
RECOMMEND ?user ?movie.REC ?RATING
WHERE {
    ?user rdf:type resparql:User .
    ?movie rdf:type resparql:Item .
    ?user movies:hasRated ?personalRating .
    ?personalRating movies:ratedMovie ?movie .
}
BASED ON {
    ?user movies:hasRated ?personalRating .
    ?personalRating movies:ratedMovie ?movie .
    ?user movies:hasGender ?gender .
    ?user movies:hasProfession ?profession .
}
MEASURES {
    ?userRating rdf:type resparql:UserRating .
    ?personalRating movies:hasRating ?userRating
}
```

Table 3.11: ReSPARQL hybrid query that recommends movies using ratings and content-based profile.

The purpose of the query is to recommend movies from like-minded users, who tend to watch the same movies, but that are also similar in their gender and professions. Three *FGPs* are specified in this query: the first one represents the user-item path, the second and third belong to the user's profile.

3.2.5 Computation of predicted ratings

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 system is aware of the following information:

?user	*resparql:User*
?item	*resparql:Item*
?userRating	*resparql:UserRating*
?itemRating	*resparql:ItemRating*

Table 3.12: Example of *RTP* configuration.

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

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

Ratings for a CB query

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

$$r(u,i') = \frac{\frac{(r(u,i)+r(i))}{2} + sim_{(i,i')} * \frac{(r(u',i')+r(i'))}{2}}{2} \tag{3.1}$$

In case item-ratings are missing or not provided:

$$r(u,i') = \frac{r(u,i) + sim_{(i,i')} * r(u',i')}{2} \tag{3.2}$$

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

$$r(u,i') = sim_{(i,i')} \tag{3.3}$$

Ratings for a CF query

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

$$r(u,i') = sim_{(u,u')} * \frac{(r(u',i')+r(i'))}{2} \tag{3.4}$$

In case item-ratings are missing or not provided:

$$r(u,i') = sim_{(u,u')} * r(u',i') \tag{3.5}$$

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

$$r(u,i') = sim_{(u,u')} \tag{3.6}$$

Finally, we assign the following value to the reserved variable

$$?RATING \longleftarrow r(u,i') \tag{3.7}$$

3.2.6 Grouping capability

The language leaves a great degree of freedom for packing the results. This is achieved by means of the grouping capabilities and aggregation functions inherited from SPARQL.

The following two queries show different strategies to pack the results. In the outline in table 3.13 each user gets, for each item, the recommendation with the highest score. In the outline in table 3.14 results are averaged.

```
PREFIX resparql: <http://example.org/resparql#>
RECOMMEND ?user ?item.REC (MAX(xsd:double(?RATING)) AS ?rating)
WHERE {
    ?user rdf:type resparql:User .
    ?item rdf:type resparql:Item .
     #path from user node to item node
    ?user . . . ?item .
    #set one or more filters to tune results
     FILTER ( . . . )
}
BASED ON {
    #item features
    ?item [...]:.. ?feat1 .
    ... .
    ?item [...]:.. ?featn .
}
GROUP BY ?user ?item.REC
```

Table 3.13: Outline of CB filtering query where only the highest rating for each user and recommended item is shown.

```
PREFIX resparql: <http://example.org/resparql#>
RECOMMEND ?user ?item.REC (AVG(xsd:double(?RATING)) AS ?rating)
WHERE {
    ?user rdf:type resparql:User .
    ?item rdf:type resparql:Item .
    #path from user node to item node
    ?user . . . ?item .
    #set one or more filters to tune results
     FILTER ( . . . )
}
BASED ON {
    #path from user node to item node
    ?user [...]:.. ?item
}
GROUP BY ?user ?item.REC
```

Table 3.14: Outline of CF query where ratings are averaged for each pair of user and recommended item.

In addition to *GROUP BY* one can also use *HAVING* to filter out some groups:

```
RECOMMEND ?user ?movie.REC
    (AVG(?RATING) AS ?avgRat)
...
GROUP BY ?user ?movie.REC
HAVING (?avgRat > 8.0)
```

Table 3.15: Use of *HAVING* clause in ReSPARQL.

Arbitrary aggregation functions can be applied to the predicted ratings or in general to any score of the solution mappings[5]. The heuristic function changes depending on the kind of aggregation performed. For instance, let's assume that the rating is computed by means of the formula (3.2) from the previous section and that one groups by *?user* and *?item.REC*. Let's assume moreover that the average of *?RATING* is projected. The averaged rating corresponds then to:

$$r(u,i') = \frac{\sum_{u'} \frac{r(u,i) + sim_{(i,i')} * r(u',i')}{2}}{\mid u' \mid}, \tag{3.8}$$

where $\mid u' \mid$ is the number of users that rated the item i'.

3.2.7 Tailored neighborhood

ReSPARQL offers a great degree of flexibility that allows obtaining highly parameterizable recommendations.

One of the novel features of ReSPARQL is that it is possible to apply filters to both data within the RDF graph and recommended data. This helps to reduce the size of the neighborhood of users or items [13] and have consequently a big impact on the size of returned set of mappings. An example is given by the following fragment:

[5]A comprehensive list of aggregation functions supported in SPARQL 1.1 can be found in `http://www.w3.org/TR/sparql11-query/#rAggregate`

```
FILTER ( ?age >= ?age.REC - 5 ||
    ?age <= ?age.REC + 5 )
```

Table 3.16: Neighborhood reduction through filters.

Assuming that the variable *?age* represents the age of a user and that this fragment is part of a CF query, then we are reducing the neighborhood of each user by considering users five years younger or older.

Similarly, it is possible to filter the input data and recommended data. By excluding certain users or recommendations from the results, aggregated ratings are also affected:

```
FILTER (xsd:double(?itemRating.REC) > 7.5 &&
    xsd:integer(?age) >= 18 )
```

Table 3.17: Filter on input data and recommended data.

3.2.8 Beyond user-item recommendations

The projection of variables is not necessarily limited to projecting user and recommended item. As in SPARQL, all variables within the *WHERE* clause can be projected in the *RECOMMEND* clause. Additionally, it is possible to project all renamed variables. Therefore, it is possible to project other variables to obtain other kinds of recommendations. A concrete example based on the movies dataset is given in table 3.18.

In this example we join more information to our solution in order to project *?profession* and *?genre.REC*. Intuitively, we are interested in knowing what the recommended genre for a certain target group is.

In presence of contextual information, it is also possible to write more specific queries, than the one we have seen previously. For instance, based on the RDFS of the movies dataset we can set filters to produce recommendations for a specific context.

```
PREFIX resparql: <http://example.org/resparql#>
PREFIX movies: <http://examples.org/movies/>
RECOMMEND ?profession ?genre.REC (...AGG. FUNCTION on RATING)
WHERE {
    ?user rdf:type resparql:User .
    ?movie rdf:type resparql:Item .
    ?user movies:hasRated ?personalRating .
    ?personalRating movies:ratedMovie ?movie .
     ?user movies:hasProfession ?profession .
     ?movie movies:hasGenre ?genre .
}
BASED ON {
    ?user movies:hasAge ?age .
    ?user movies:userCountry ?nationality .
    ?user movies:hasProfession ?profession
}
GROUP BY ?profession ?genre.REC
```

Table 3.18: ReSPARQL query that recommends the genre of a movie to a target audience.

```
PREFIX resparql: <http://example.org/resparql#>
PREFIX movies: <http://examples.org/movies/>
RECOMMEND DISTINCT ?user ?movie.REC
     ?time.REC ?companion.REC ?RATING
WHERE {
    ?user rdf:type resparql:User .
    ?movie rdf:type resparql:Item .
    ?user movies:hasRated ?personalRating .
    ?personalRating movies:ratedMovie ?movie .
     ?personalRating movies:movieTime ?time .
     ?personalRating movies:accompanied ?companion
      FILTER ( (?user = “Bob”) && (?time.REC = “weekend”)
         && (?companion.REC = “partner” ) )
}
BASED ON {
    ?movie hasGenre ?genre
}
MEASURES { ... }
```

Table 3.19: ReSPARQL query that recommends for a specific user within a specific context.

This last query returns recommendations, movies that other users have seen in a specific context, for a specific user "Bob".

ReSPARQL is based on SPARQL 1.1. This is a not exhaustive list of the clauses supported by ReSPARQL: *AGGREGATION FUNCTIONS*, *GROUP BY*, *HAVING*, *DISTINCT*, *LIMIT*, *OFFSET*, *OPTIONAL*, *FILTER*, *FILTER EXISTS*, *IN*. A fragment of ReSPARQL grammar can be found in Appendix A.

3.3 Semantics

This section expounds the formal semantics of ReSPARQL. The approach here presented is based on [23], one of the first formalizations presented which built the basis for the SPARQL 1.0 semantics. This described the evaluation of a (simplified) fragment of SPARQL and hence it was not complete, but provided a tool which was strong enough to analyze SPARQL evaluation's complexity[6].

An exhaustive description of the current version of SPARQL's semantics, SPARQL's 1.1, can be found in the specification of the language [15] (section 18: Definition of SPARQL).

3.3.1 SPARQL algebra and evaluation semantics

Some preliminaries notions of SPARQL's algebraic syntax are here reported. All these definitions can be found in [15, 23]:

- RDF-T (RDF-Terms) is a set $I \cup$ RDF-L $\cup$ RDF-B, where I is the set of all IRIs, RDF-L is the set of all literals and RDF-B is the set of all blank nodes in RDF graphs[7].
- An RDF-Triple is a member of the set $\in (I \cup$ RDF-B$) \times I \times$ $(I \cup$ RDF-B $\cup$ RDF-L$)$.
- An RDF-Graph is a set of RDF triples.

[6]Evaluation of graph pattern in SPARQL is PSPACE-complete, whereas evaluation of queries is coNP-complete.

[7]The sets I, RDF-B, and RDF-L are infinite and pairwise disjoint.

- An RDF-Dataset is a set $\{G, (< i_1 >, G_1), (< i_2 >, G_2), \ldots, (< i_n >, G_n)\}$ where G and each G_i are graphs, and each $< i_i >$ is an IRI. Each $< i_i >$ is distinct. G is called the default graph. $(< i_i >, G_i)$ are called named graphs.

- V is the set of query variables. It is infinite and disjoint from RDF-T.

- A TP (triple pattern) is a member of the set (RDF-T $\cup V) \times (I \cup V) \times$ (RDF-T $\cup V$).

The syntax of SPARQL uses concatenation (.) and the operators $OPTIONAL$, $UNION$, $FILTER$ are used to construct graph patterns expression [23]. P is a graph pattern[8] [9] recursively defined:

- P is a TP

- If P_1 and P_2 are graph patterns, then expressions $(P_1\ AND\ P_2)$, $(P_1\ OPT\ P_2)$, and $(P_1\ UNION\ P_2)$ are graph patterns too.

- If P is a graph pattern and F is a filter expression then the expression $(P\ FILTER\ F)$ is also a graph pattern.

The following terminology allows us to introduce the semantics of graph patterns expressions:

- μ is a solution mapping, a partial function $\mu : V \rightarrow$ RDF-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.

- Given a mapping μ: $V \rightarrow$ RDF-T and a set of variables $W \subseteq V$, the restriction of μ to W, denoted by $\mu_{|W}$ is a mapping such that $dom(\mu_{|W}) = dom(\mu) \cap W$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in dom(\mu) \cap W$.

[8] In SPARQL's documentation the following terminology is used instead: BGP (basic graph pattern) corresponds to concatenation of triple patterns, GGP (group graph pattern) are patterns grouped with "{}", and AGP (alternative group pattern) corresponds to $UNION$ of triple patterns)

[9] Algebraic properties of graph patterns are defined in [23]

- Ω is a multiset of solution mappings. We define the join of, the union of, the difference and left outer join between Ω_1 and Ω_2 as:
  - $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible mappings$\}$;
  - $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$;
  - $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid$ for all $\mu' \in \Omega_2$, μ and μ' are not compatible$\}$;
  - $\Omega_1 \bowtie_{lo} \Omega_2 = (\Omega_1 \bowtie \Omega_2\ ) \cup (\Omega_1 \setminus \Omega_2)$ (left outer join).
- Since Ω is a multiset every mapping is annotated with a positive integer that represents cardinality of that mapping, $\text{card}_\Omega(\mu)$. If $\mu \notin \Omega$, then $\text{card}_\Omega(\mu) = 0$. The above defined operations do not discard duplicates[10].

The evaluation of a graph pattern P is a function $\|P\|_G : P \to \Omega$, where G is an RDF-Graph. $\|P\|_G$ is recursively defined as follows:

- If P is a TP, then $\|P\|_G = \{\mu : V \to \text{RDF-T} \mid dom(\mu) = var(TP)$ and $\mu(TP) \in G\}$, where $var(TP)$ denotes the set of variables occurring in the components of TP and $\mu(TP)$ is the set of triples obtained by replacing the variables in TP according to μ;
- If P is (P_1 AND P_2), then $\|P\|_G = \|P_1\|_G \bowtie \|P_2\|_G$;
- If P is (P_1 OPT P_2), then $\|P\|_G = \|P_1\|_G \bowtie_{lo} \|P_2\|_G$;
- If P is (P_1 UNION P_2), then $\|P\|_G = \|P_1\|_G \cup \|P_2\|_G$;
- $\|(P\ FILTER\ R)\|_G = \{\mu \in \|P\|_G \mid \mu$ satisfies $R\}$.

In the next subsection, we will see how ReSPARQL queries can be evaluated. This requires new definitions. For the sake of simplicity, we will assume that the RDF-Graphs do not have blank nodes or literal subjects.

3.3.2 ReSPARQL algebra and evaluation semantics

A ReSPARQL query contains a special subset of triple patterns, whose purpose it to provide the system with information about the role of variables. These will be for convenience denoted as TP_{RTP}.

[10]For the sake of brevity, we omit here how the cardinalities are updated when applying an operator. This is explained in [8].

- $TP_{RTP} \in V \times$ (rdf:type)[11] $\times$ (resparql:[...])[12]

As we know, these triple patterns are not matched against the RDF-Graph. Therefore, we need to extend the definitions of the previous operators as follows:

- If P is (P_1 AND | OPT | UNION P_2) and P_1 and P_2 are both triple patterns TP_{RTP}, then $\|P\|_G = \mu_\phi$, i.e. the mapping with empty domain[13];
- If P is (P_1 AND | OPT | UNION P_2) and P_1 is a TP_{RTP}, then $\|P\|_G = \|P_2\|_G$;
- If P is (P_2 AND | OPT | UNION P_2) and P_2 is a TP_{RTP}, then $\|P\|_G = \|P_1\|_G$;

The information provided by the triple patterns TP_{RTP} allows the recommender to set four variables:

- $\mathrm{v}_u \in V$ is the variable that represents users;
- $\mathrm{v}_i \in V$ is the variable that represents items;
- $\mathrm{v}_{ru} \in V$ is the variable that represents user-ratings;
- $\mathrm{v}_{ri} \in V$ is the variable that represents item-ratings;

The evaluation of the *BASED ON* clause is the key of the computation of recommendations. Let TP_B denote the triple patterns appearing within this clause.

- A TP_B is a member of the set $V \times I \times V$.
- We define a way of accessing the two variables of which a TP_B is composed: $TP_B(s)$ is the variable on the left of the IRI and $TP_B(o)$ is the variable on the right of the IRI.

[11]rdf ist the prefix of `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

[12]resparql is the prefix of ReSPARQL's namespace.
The URL `http://example.org/resparql#` is used only for illustrative purposes.

[13]μ_ϕ is compatible with any other mapping

Let's define now a special kind of graph pattern P called feature F. This is defined recursively as follows[14]:

- P is a TP_B. P is a feature F if $P(s) = v_u$ or $P(s) = v_i$;
- $P = (P_1$ AND $P_2)$ is a feature F if P_1 is a feature, $P_1(o) = P_2(s)$, and P_1 and P_2 appear in that order within the *BASED ON* clause. We say that P_1 was extended by P_2. Moreover, $P(s) = P_1(s)$ and $P(o) = P_2(o)$.

We define the following function $\nu : (v \rightarrow t, \Omega, v') \rightarrow \Gamma$, a multiset of RDF-T terms $(t, card(t))$, which we call feature function, because it collects all feature values of a given object. For instance, suppose one particular movie is represented by the mapping $?m \rightarrow$ "$TheHobbit$". We want to obtain the feature genre represented by the variable $?g$. We can then apply the function $\nu_1(?m \rightarrow$ "$TheHobbit$", Ω, $?g)$ and obtain the multiset $\{($"$adventure$"$, 1), ($"$fantasy$"$, 1)\}$. The second value of each element is the cardinality of that element within the multiset.

Formally, ν_1 consists of all RDF-T terms $\mu(v')$ for which $v \rightarrow t \in \mu$ and $\mu \in \Omega$:

- $\nu_1(v \rightarrow t, \Omega, v') = \{\mu(v') \mid \mu \in \Omega, v \textit{ and } v' \in V,\ t \in \text{RDF-T and } \mu(v) = t\}$

The following function has a similar purpose:

- $\nu_2(v \rightarrow t, \Omega, v', r) = \{(\mu(v'), \mu(r)) \mid \mu \in \Omega, v, v' \textit{ and } r \in V,\ t \in \text{RDF-T and } \mu(v) = t\}$

As before the variable v' represents a feature, whereas r represents a given rating. The goal is to build a multiset of RDF-T terms $((t, r), card(t))$. Note that it is now the combination of both the RDF-T term and the rating which defines the members of Γ.

Another function needed for the evaluation is the following:

- $\zeta : (\Gamma_1, \Gamma_2) \rightarrow [0, 1]$

[14] Triple patterns TP_B can be only concatenated. Therefore, other operators such as $OPTIONAL$, $UNION$, $FILTER$, etc., will not be considered.

Γ_1 and Γ_2 are two multisets of the same kind, i.e. either $(t, card(t))$ or $((t, r), card(t))$. The goal of ζ is to compute similarities between two multisets and to return a value that represents the similarity degree (1 is the max.). This can be achieved as follows:

- convert Γ_1 and Γ_2 into two sets by ignoring the cardinalities of elements and then compute the Jaccard distance or;
- convert Γ_1 and Γ_2 into two vectors v_1 and v_2 and then compute the cosine distance. The process of converting from multisets to vectors is here described[15]:
  - If Γ_1 and Γ_2 are both of kind $(t, card(t))$, then v_1 and v_2 have each length $= \mid \Gamma_1 \cup \Gamma_2 \mid$. Assign an index to each of the elements $\in \Gamma_1 \cup \Gamma_2$. Finally, for each element of Γ_1, set a 1 in v_1's cell at the assigned index and repeat the process for Γ_2 and v_2. Optionally, one can weight vectors by the cardinality of each element.
  - If Γ_1 and Γ_2 are both of kind $((t, r), card(t))$, then build two new multisets Γ'_1 and Γ'_2. For each member of Γ_1 having the same term t_k, collect all ratings r_i and put into Γ'_1 the member $(t_k, \frac{\sum_{i=1}^{n} r_i}{n})$. Repeat the process for Γ_2. Then use Γ'_1 and Γ'_2 to build two vectors using the procedure described in the previous point and weight them by the averaged rating.

Finally, we need to define a renaming operator. Given a multiset of mapping solutions, Ω:

- $\rho_{(?oldVarName:=?newVarName)}(\Omega)$: for each $\mu \in \Omega$, if $?oldVarName \in dom(\mu)$ then that variable is renamed $?newVarName$.

Since we will typically rename all variables, we also define the following shortcut:

- $\rho_{.REC}(\Omega)$: for each $\mu \in \Omega$ rename each $v \in dom(\mu)$ by appending the string ".REC" to the end of v.

[15]The similarity engine of ReSPARQL is based on the cosine distance approach.

Evaluation of Graph Patterns in the *WHERE* and *MEASURES* clauses

The process of translating a ReSPARQL graph pattern into a ReSPARQL algebra expression is akin to SPARQL: within the *WHERE* and *MEASURES* clause, the revised recursive definition for triple patterns TP_{RTP}, based on union, join and left join is used to create a graph pattern.

After obtaining the graph patterns individually, we evaluate $\|P_W \; AND \; P_M\|_G$.

- Let be Ω_{WM}, the multiset of mappings obtained from the evaluation of $\|P_W \; AND \; P_M\|_G$.

Note that *MEASURES* is only an optional clause in ReSPARQL. If this is not provided the evaluation of Ω_{WM} is then the result of evaluating $\|\mathrm{P}_W\|_G$.

Evaluation of Graph Patterns on the *BASED ON* clause

The evaluation of the *BASED ON* clause requires the notions formalized at the beginning of this subsection.

In order to evaluate the graph patterns in the *BASED ON* clause, we need to partition the set of triple patterns TP_B into a set of features $\{F_1, F_2, ..., F_{n'}\}$ having the following properties:

- $\forall i, F_i$ cannot be extended;
- each $TP_B \in$ to exactly one F_i;
- either $F_i = v_u, \forall i \; or \; F_i = v_i, \forall i$.

If these properties do not hold for the partition, then the whole evaluation process is aborted.

$F_i(s)$ and $F_i(o)$ represent respectively the variable of objects to which the similarities function is applied and the variable of features on which the similarity is based.

We distinguish two kind of evaluations: user-based recommendation or item-based recommendation. The first one is triggered when $F_i(s) = v_u, \forall i$. The second, when $F_i(s) = v_i, \forall i$. Let the variable be:

$$v_{sim} = \begin{cases} v_u : & \text{if it is a user-based recommendation;} \\ v_i : & \text{if it is a item-based recommendation.} \end{cases}$$

- $\widehat{v}_{sim} = \{\mathrm{v}_u, \mathrm{v}_i\} \setminus \mathrm{v}_{sim}$.

We search in the partition the following features:

- F_{ui}: a feature whose $F_{ui}(s) = v_u(v_i)$ and whose $F_{ui}(o) = v_i(v_u)$;
- F_{ru}: a feature whose $F_{ru}(s) = v_u(v_{ru})$ and whose $F_{ru}(o) = v_{ru}(v_u)$.

If both features are found, we remove them from the partition and add $P_{rat} = (F_{ui}$ AND $F_{ru})$. The initially found partition is otherwise used. Suppose the resulting partition is $\varphi = \{[P_{rat}], F_1, F_2, ..., F_n\}$.

We evaluate each of the features independently: $\{[\|P_{rat}\|_D], \|F_1\|_D, \|F_2\|_D, ..., \|F_n\|_D\}$ and obtain the following multiset of mappings: $\{[\Omega_{P_{rat}}], \Omega_{F_1}, \Omega_{F_2}, ..., \Omega_{F_n}\}$.

Now, we get the set of all RDF-T terms having at least one of the features:

- $\sigma = \{\bigcup_{i=1}^{n} (\mu(F_i(s)) \mid \mu \in \Omega_{Fi}) \bigcup (\mu(v_{sim}) \mid \mu \in \Omega_{P_{rat}})\}$

Since σ is a set, it has no duplicates of RDF-T terms. We will also use the cross product operator, which is defined for sets.

- $\sigma \times \sigma = \{(\sigma_i, \sigma_j) \mid \sigma_i, \sigma_j \in \sigma\}$.

For each pair of objects (σ_i, σ_j) and for each $\Omega \in \varphi$, we compute the similarity score:

$$score_{\sigma_i,\sigma_j,F} = \begin{cases} \zeta(\nu_1(F(s) \to \sigma_i, \Omega_F, F(o)) \\ \quad \nu_1(F(s) \to \sigma_j, \Omega_F, F(o))), & \text{if } \Omega_F = \Omega_{F_k}; \\ \zeta(\nu_2(v_{sim} \to \sigma_i, \Omega_F, \widehat{v}_{sim}, v_{rui}), \\ \quad \nu_2(v_{sim} \to \sigma_j, \Omega_F, \widehat{v}_{sim}, v_{rui})) & \text{if } \Omega_F = \Omega_{P_{rat}} \end{cases}$$

For each pair of terms, we calculate the average of all scores obtained by considering different features:

$$score_{\sigma_i,\sigma_j} = \frac{(\sum\limits_{k=1}^{n} score_{\sigma_i,\sigma_j,F_k}) + score_{\sigma_i,\sigma_j,P_{rat}}}{\mid \varphi \mid};$$

Finally, we construct a multiset of mappings as follows:

- Ω_B, the multiset of mappings obtained by evaluating the graph patterns within the BASED clause.

- $\Omega_B = \{v_{sim} \rightarrow \sigma_i, v_{sim}.REC \rightarrow \sigma_j, ?SIMscore \rightarrow score_{\sigma_i,\sigma_j} \mid (\sigma_i, \sigma_j) \in \sigma\}$

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

- $\Omega_{REC} = \Omega_{MW} \bowtie \Omega_B \bowtie \rho_{.REC}(\Omega_{MW})$

Ω_{REC} does not contain the predicted rating $?RATING$. Given a $\mu \in \Omega_{REC}$, this mapping is computed as described in 3.2.5. A new multiset of solution mappings is built to incorporate the predicted rating.

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

This concludes the semantics of ReSPARQL.

Chapter 4

Implementation

In order to evaluate ReSPARQL recommendation queries a standalone recommender repository as an extension to Sesame[1] was designed and implemented. This was achieved by extending Sesame's Sail and Repository API. The recommender repository will be hereinafter referred to as ReSPARQL recommender.

The purpose of this chapter is to present the key features of the system with a special focus on the following components:

- ReSPARQL's pre-parser and parser;
- recommender repository;
- extended algebra operators;
- evaluation strategy for ReSPARQL queries;
- cache-system to optimize recommendations.

4.1 Sesame's architecture

The system documentation of Sesame is not exhaustive. In order to implement a dedicated recommended repository the first task consisted in under-

[1]Sesame is an open source Java framework for storing, querying, and reasoning with RDF and RDF Schema. It can be used as a database for RDF and RDF Schema, or as a Java library for applications that need to work with RDF `http://www.openrdf.org`

standing the role of components by analyzing the data flow and by studying the javadoc API [3].

Sesame is probably the most used framework in the Semantic Web community. It is not only an RDF store, but it also provides tools to query and manipulate data locally and remotely. The framework is fully extensible and configurable in terms of storage mechanisms, inferencers, RDF file formats, query result formats and query languages. This flexibility led to a highly integrated recommender repository which can be easily integrated into a Sesame-based project by simply replacing the dependencies.

Sesame has a layered architecture as shown in the following figure:

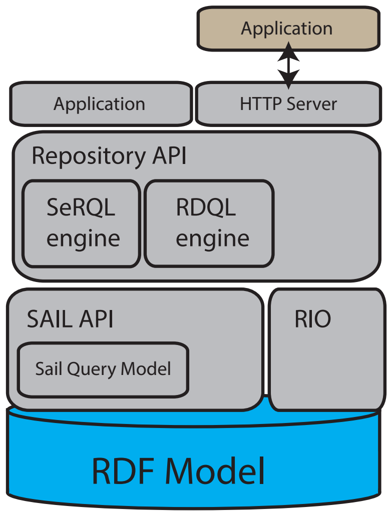


Figure 4.1: Sesame's architecture.

At the bottom we find the RDF-model. All components of Sesame are partially dependent on this layer. The RDF-model specifies interfaces and implementations of all RDF building blocks: IRIs, anonymous nodes, literals, statements, etc.

Data can be loaded into the system by means of the RDF Input/Output (RIO) component. The RIO input component consists of a set of parsers that makes it possible to load serialized RDF data. Almost any serialization format is supported, from RDF/XML to N3 and Turtle. The output component of RIO allows us to serialize statements stored in the RDF-model.

Actual RDF data can be stored in different ways, e.g. in main memory, as a relational database, by using dedicated disk-based data structures, or by implementing a hybrid approach. Independently of the storage, a uniform way of accessing and manipulating data that abstracts from implementation details is realized by means of the so-called Sail API (Storage And Inference Layer). This low level communication interface permits the performing of operations such as adding, removing and querying RDF triples.

An operation is accomplished in Sesame by two different components: a Sail object and a connection (*SailConnection*) set to communicate with the object. In Sesame there are several implementations of these pairs of components to support different features. For instance, some connections and objects handle transactions and concurrent access, whereas other do not support these functionalities to allow for a more efficient evaluation. Some Sails can be stacked on top of other Sails by implementing the *StackableSail* interface. In that case all calls directed to the bottom Sail will pass through the Sails that are on top of it. Another key design feature of Sesame is that all data extracted from a Sail object is returned in the form of (forward-only) iterators. This scalable approach allows one to fetch a set of stored statements by only keeping one statement at a time in main memory.

Out of the box, Sesame supports SPARQL and SeRQL (Sesame Rdf Query Language) querying. The engines transforms a query into a Sail query object. The Sail Query Model provides a uniform model for representing queries. A language-specific parser transforms a query into a tree-based representation, which can then be optimized and evaluated.

The repository API is a higher level API that in turn uses the Sail API. This allows the user to perform operations similar to those one can perform in a Sail object, but it adds a further layer that allows for abstracting architectural details. A local repository is designed to operate on a local context, i.e. on the same Java virtual machine. In contrast, a remote repository API has been designed to operate on a client-server architecture. There are several implementations of this API, e.g. *SailRepository* and *HTTPRepository*. The former translates calls to a Sail implementation of choice, the latter offers transparent client-server communication to a Sesame server over HTTP.

The top-most layer in the diagram is the HTTP Server. It consists of a number of Java Servlets that implement a protocol for accessing Sesame repositories over HTTP. The details of this protocol can be found in Sesame's system documentation. The RESTful HTTP interface supports the SPARQL Protocol for RDF.

4.2 ReSPARQL recommender's architecture

In this section the architecture of the system will be described. The implementation of the recommender is an extension of Sesame 2.7, which is a stable release. This recommender repository is a local repository, which stores data in main memory (but it is capable at the same time of synchronizing data with a file). The following figure illustrates a simplified workflow of the system:

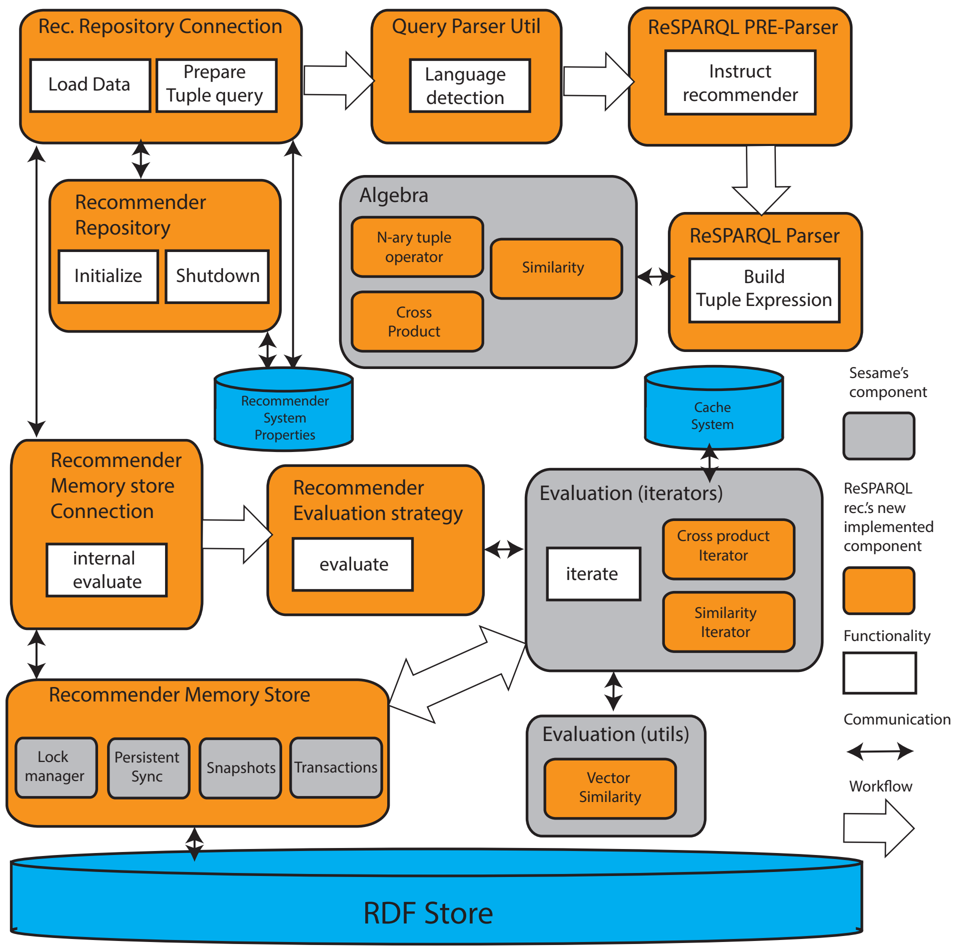


Figure 4.2: Simplified workflow of ReSPARQL recommender respository.

In order to implement the ReSPARQL recommender, new components in almost each layer had to be created as figure 4.2 illustrates. However, many of the components are extensions of already existing implementations and therefore many functions didn't have to be reimplemented.

In the remaining part of this chapter these new components will be described according to the workflow triggered by a query execution.

4.2.1 Recommender Sail and Repository

The Recommender Sail is an extension of *MemoryStore*. This implementation of a Sail stores its data in main memory, but can also use a file for persistent storage. *MemoryStore* supports single, isolated transactions. This means that changes to the data are not visible until a transaction is committed and that concurrent transactions are not possible. If a transaction is active, calls to *startTransaction()* waits until the active transaction is committed or rolled back [3].

RecommenderMemoryStore extends the class *MemoryStore* and inherits most of its features. Since a store is a Sail object, a Sail connection is required in order to perform operations on it. *RecommenderMemoryStore* returns a *RecommenderMemoryStoreConnection*. In this component, two functionalities were redefined.

First of all, the evaluation of queries forwards the flow to recommender-specific components. Secondly, the *RecommenderMemoryStoreConnection* is responsible for starting the pre-processing of RDF data to enhance the computation of similarities. This occurs when a request for pre-processing arrives from components on top of it, e.g. when the data is initially loaded, or when the evaluation of a query requires data from the cache, and this has not been initialized yet. In section 4.2.4, the pre-processing phase is described in more detail.

Just as in Sesame, a *SailRecommenderRepository* and a *SailRecommender-RepositoryConnection* were implemented on top of the above described Sail. One important task redefined for the repository connection, was the parsing of queries. It forwards the control to recommender-specific components.

Each of the components of these two layers have also access to the properties of the system and the cache system, which can then be passed by reference to other components, e.g. the parsing or evaluation module.

4.2.2 ReSPARQL pre-parser and parser

The parser was generated by means of JavaCC[2]. The grammar of SPARQL's syntax in Sesame is specified in a file called *sparql.jj*, which is distributed together with the sources of the project. The extended ReSPARQL clauses were added to it, such as *RECOMMEND*, which was added as a new query type (such as *SELECT* or *CONSTRUCT*). The file can then be given as input to JavaCC, which generates a parser from it. However, the parser could not be directly integrated into Sesame; it was necessary to further modify some of the generated classes to achieve this.

The goal of the parser is to take a ReSPARQL query as input and to generate a tuple expression tree (a tree representation of the query) from it. The tree's nodes represent either SPARQL algebra operators or ReSPARQL-specific operators.

While the parser was designed to validate the grammar of the query and build the tree, a pre-parser was designed to extract information from the query, whose purpose is to instruct the recommender system about how to perform recommendations.

To illustrate the role of each component, let's consider the query in table 4.1:

...	
1	**RECOMMEND** ?user ?movie.REC ?RATING
2	**WHERE** {
3	**?user rdf:type resparql:User .**
4	**?movie rdf:type resparql:Item .**
5	?user movies:hasRated ?persRating .
6	?persRating movies:ratedMovie ?movie }
7	**BASED ON** {
8	**?movie movies:hasGenre ?genre**
9	}

Table 4.1: Example to illustrate the roles of pre-parser and parser in ReSPARQL.

Some of the tasks of the pre-parser consist in:

[2]JavaCC is an open source parser generator and lexical analyzer generator for the Java programming language.

- verifying that the query contains a prefix of resparql's namespace;
- using the *RTP* to extract the roles of variables;
- detecting graph patterns within the *RSP* which correspond to features;
- verifying that all *FGPs* are related to either users or items but not both;
- automatically detecting the approach to be followed, i.e. content-based or collaborative filtering, based on the *FGPs* provided.

If all information is successfully extracted from the query, it is forwarded to ReSPARQL's parser, which builds a syntax-based tree in two phases. First of all, it first generates a syntax-based tree that represents the clauses used in the query, as the figure 4.3 shows:

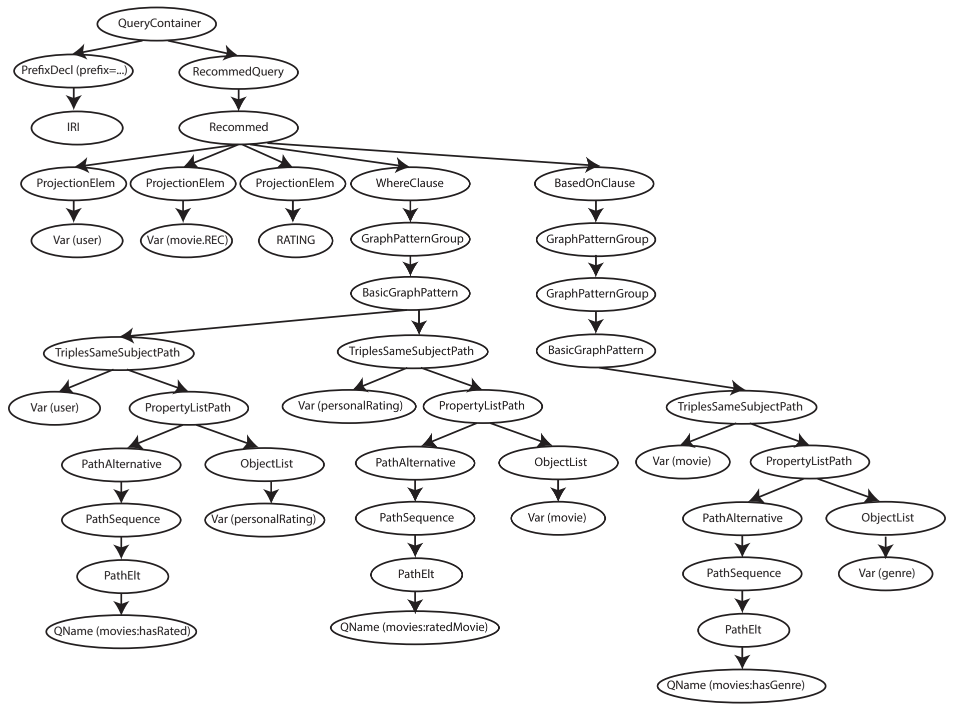


Figure 4.3: ReSPARQL query container.

Finally, this tree is processed again and the algebra tree is built from it, as the figure 4.4 shows:

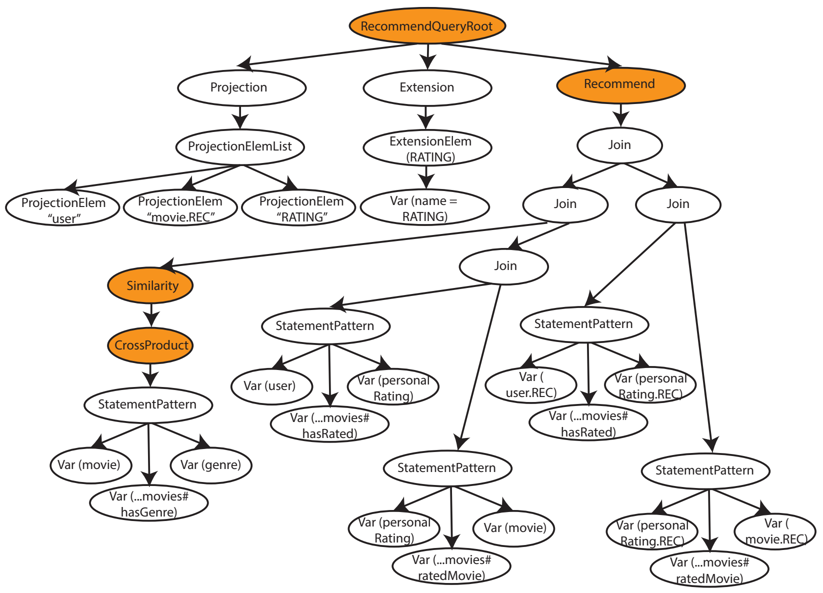


Figure 4.4: ReSPARQL tuple expression (algebra tree).

All nodes in orange in fig. 4.4 are the new operators introduced in ReSPARQL in order to compute recommendations.

4.2.3 ReSPARQL algebra tree

The purpose of an algebra tree in Sesame is to provide a hierarchical abstract model of operators that can be applied to data in order to produce a certain result. An algebra tree can be optimized and evaluated.

The node at the top of tree in fig. 4.4 is the node which distinguishes the kind of operation that one wants to accomplish. The node *Recommend-QueryRoot* carries with it information that the recommender system needs to compute recommendations, e.g. the role of nodes, or the kind of approach to use (CF or CB).

The node *Recommend* is the node responsible for computing the predicted ratings when the complete solution mapping is available, i.e. when explicit data has been joined with the similarities solution mappings.

The *CrossProduct* operator shown in figure 4.4 takes a set of features as input (it is an N-ary operator, although in this example it has only one feature as input) and produces pairs of objects, either users or items, having at least one of the listed features together with a set of feature vectors, which can then be compared by the *Similarity* operator. In Sesame, algebra trees are processed bottom-up, i.e. each of the nodes in the tree applies the operator and then passes the result to the parent node.

The *Similarity* operator computes the similarity of the objects, two at a time, and applies the cosine distance function to each pair of feature vectors.

4.2.4 Pre-processing and cache system

A cache system was implemented and integrated into Sesame's architecture. The main goal of the cache system is to maintain a multimap, whose purpose is to enable a faster retrieval of both, objects having a certain feature, and, of feature values for a given object. Two implementation of multimaps were used in the project: the first one is based on Guava Libraries[3], whereas the second one is based on MapDB[4]. In particular, the implementation based on the second library guarantees that the multimap can be backed in a file if the amount of memory is not enough.

In the multimap keys are created by concatenating a subject and predicate's URI, representing respectively an object and a feature, and whose values are all the values that the object has for that feature. In this way, it is possible to retrieve the feature vector for a given object-feature in a reasonable time. Therefore, an iterator doesn't need to retrieve the information from the RDF-store.

The second and, not less important, task of the cache is to store information about the queries itself, e.g. the variables representing the features on each *FGP*, or the prefixes used in the query.

When the system is started and data is loaded for the first time into it, the system pre-processes all paths of length 1. For each triple subject-predicate-object found, a key "subject.predicate" is created and the "object" is stored in the multimap as the value. Further features are added into the cache if required.

[3]The Guava project contains several of Google's core libraries that we rely on in our Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. `https://code.google.com/p/guava-libraries/`

[4]MapDB provides concurrent Maps, Sets and Queues backed by disk storage or off-heap memory. `http://www.mapdb.org/`

4.2.5 Evaluation

In Sesame a tuple expression tree is evaluated by implementing a strategy. For instance, a strategy for evaluating SPARQL queries is represented by the class *EvaluationStrategyImpl*. Consequently, a strategy to evaluate ReSPARQL queries was designed which can also receive benefit from the cache system and therefore minimize data retrieval.

The tree is processed recursively and the evaluation is triggered from the root node. In a strategy class, one iterator is responsible for evaluating each of the operators (nodes of the tree). Therefore, data is first retrieved by iterators which evaluate operators at the leaves of the tree and the solution mappings are given as input to other iterators. This kind of iterators accesses data directly from the Sail object, *RecommenderMemoryStore*, using a connection. An iterator has two main methods: *hasNext()* and *next()*. The first one verifies that solution mappings coming from other iterators are still available. The second one retrieves the next available solution mapping.

The evaluation through iterators is a key feature in Sesame: there's at most one "tuple" at a time in main memory. When the final solution mapping reaches the iterator at the top of the tree, it is returned to the caller or serialized by means of the RIO interface.

Two iterators were implemented in ReSPARQL: *CrossProductIterator* and *SimilarityIterator*. These work tightly with the cache system, i.e. they extract data only when this is necessary. For instance, *CrossProductIterator* first determines if a feature could have been stored in the cache. If this is the case, it doesn't need to recursively call other iterators to retrieve the data from them. As we will see in the next section, having a feature in cache means that, first of all, the iterator can get a list of all objects having that feature. Secondly, the values that an object could have for a specific feature (they might have multiple values, e.g. cast for a movie) are also stored in the cache. A solution mapping is created which consists of two objects and, for each of them, the location of each feature vector stored in the cache, i.e. a key to retrieve the vector.

If a feature has not been previously processed by the system, then it might not be in cache. In that case, the normal flow is triggered. A recursive call retrieves a solution mapping from another iterator. Eventually, data is retrieved from the Sail object. A feature vector is built for an object and the information is added into the cache. Finally, the iterator builds a solution mapping as described above, as if the feature vectors would have been already in the cache.

The *SimilarityIterator* simply retrieves the feature vectors for each object using the provided keys and computes the cosine distance for each pair of feature vectors. The final score is the average of the individual cosine distances obtained.

The computation of the cosine distance is made by the class *VectorSimilarity* which is a utility class within the evaluation module. *SimilarityIterator* returns a solution mapping which contains both the pair of objects considered and the final similarity score.

Chapter 5

Conclusion

The purpose of this work was to determine whether the two processing paradigms of SPARQL and recommender systems could be combined in one solution. In order to answer that question, a query language, ReSPARQL, was designed as an extension of SPARQL.

A tight integration was evidenced in this novel approach and thereby both paradigms receive benefits from each other. In regard to expressiveness, ReSPARQL has shown itself capable of taking advantage of SPARQL's features, e.g. the possibility of working on arbitrary RDF-graphs and retrieving and filtering data by means of graph pattern matching. Meanwhile ReSPARQL also overcomes SPARQL's limitations, e.g. the limitation of producing implicit patterns. Both content-based and collaborative-filtering are possible in ReSPARQL. By adding content-based features of users to a collaborative approach, hybrid recommendations are also partially supported. ReSPARQL queries are highly customizable: they allow the specifying of tailored neighborhoods, the enabling of querying recommendations which go beyond the classic user/item paradigm, and they allow for the grouping of results in a flexible way.

The ReSPARQL recommender system, implemented during the course of this work, demonstrates that an implementation to evaluate ReSPARQL queries is feasible. With the approach followed in this work a query model can be built from a query, which in turn can be optimized and evaluated. This was successfully combined with a cache system, which helped to accelerate the computation of similarities.

5.1 Future work

ReSPARQL is a novel approach for computing recommendations against RDF-graphs which covers many aspects: from the specification of the language, which determines its expressiveness, to the development of new techniques to obtain recommendations more efficiently.

In the remaining part of this section, an overview of potential areas for future research will be provided by categorizing them into two parts: expressiveness of ReSPARQL and implementation of the ReSPARQL recommender system.

Expressiveness of ReSPARQL

- **Tighter integration with SPARQL 1.1:** the integration of both paradigms, SPARQL and recommender systems, could be further enhanced by supporting further SPARQL 1.1 features, e.g. sub-queries, property paths, inference mechanisms, etc. This would allow us to express more sophisticated connections between users, items and their features. One limitation seen in ReSPARQL is that, depending on the direction of the arcs, it might be possible to use only one approach, either a user-based CF or an item-based CF. With property paths this could be solved by matching an inverse path.

- **Multiple semantic knowledge bases:** as mentioned in the introduction, a recommender system in the Semantic Web vision should aim to produce cross-domain recommendations by working on different semantic knowledge bases, e.g. on distributed RDF data in the Linked Data model. This would bring the benefit of filling information gaps, e.g., of recommended items. A model to be followed could be based on the specification of SPARQL 1.1 Federated Query.

- **Beyond accuracy:** allowing users to express the desired degree of diversity, novelty or serendipity would increase the expressiveness of the language [35].

- **User-defined ratings:** in ReSPARQL the predicted rating is computed internally, but it could be worthwhile to allow users to define their own heuristic functions. This could be achieved in the future if expressions are allowed directly in the projection clause in SPARQL[1].

- **Advanced similarity function:** when comparing features with single values, e.g. the age of a user, the similarity computed with cosine distance returns either a “1” if both users have the same age, otherwise “0”. The similarity heuristic does not reflect the fact that two users, whose age differ by 5 years, are more similar than two users, whose age differ by 10 years. The similarity function is not able to understand the semantics of the features, but this would be desirable in a semantic recommender.

- **Normalized ratings:** In classic approaches, predicted ratings are normalized (see section 2.2.2). It would be desirable to implement this in ReSPARQL.

- **Hybrid approach:** some hybrid approaches were presented in section 2.2.3. In ReSPARQL hybridization is achieved by implementing a collaborative approach that uses not only ratings but also content-based profiles of users. This could certainly be extended to support more kinds of hybrid approaches, e.g. by combining predicted ratings obtained from content-based an collaborative approaches. In order to achieve this, ReSPARQL should permit *FGPs* which refer to both users or items.

- **Partition by:** for recommender applications it might be useful to have an analytic clause, like in SQL, that divides the result set into partitions, without necessarily aggregate them. For instance, *PARTITION BY ?user SHOW TOP 5 ?RATING* would show for each user the top 5 ratings obtained. This cannot be currently achieved by SPARQL.

Implementation of ReSPARQL recommender system

- **Automatic detection of users and items from RDF-graph:** recommenders are typically focused on certain types of relationships

[1]It is still debated whether this feature will be added or not:
`http://www.w3.org/TR/sparql-features/#Project_expressions`

between objects and it is still actively researched how to choose the kind of objects that should be recommended at runtime in an ad hoc fashion.

- **Improvement of the cache system:** the cache system was designed to be able to rapidly retrieve all objects having a certain feature and to retrieve vectors of features, which certainly saves times accessing and retrieving data from Sesame's memory store. This could, however, be extended to directly store the similarity scores to further avoid the computation of the cosine distance on queries with the same *RSP*.

- **Improvement of the pre-processing phase:** computation of similarities between different kinds of objects is typically done in classic recommender systems in a pre-processing phase. However, in ReSPARQL this is a very hard task, because the system cannot know in advance (until at least one query is evaluated) which nodes represent users/items. The roles of nodes could even change from query to query. Considering all possible scenarios for pre-computation of similarities is unfeasible.

- **Optimization of evaluation:** the model built from a ReSPARQL query can be optimized, e.g. by relocating the filters in the correct nodes of the expression tree.

- **Schema awareness:** a key feature of ReSPARQL consists in providing recommendations without relying on a RDFS. However, if this kind of information is available, the system should be able to use it to provide more accurate recommendations.

- **Use of ontologies in ReSPARQL:** as for other recommended systems implemented for RDF data sources, it could be useful to specify a ReSPARQL ontology to enhance contextual information or to represent the semantic distances between objects.

- **Augmentation of data sources:** it could be useful to augment the original data with RDF triples which represent the recommendations, e.g. to analyze the trend of recommendations over time.

Appendix A

Appendix

A.1 RESPARQL JJTree fragment (grammar)

The following is a fragment of RESPARQL JJTree input file for JavaCC. The purpose of this appendix is to show how ReSPARQL specific tokens where intertwined with SPARQL tokens. The complete SPARQL JJTree is available as a part of Sesame's documentation [3].

```
TOKEN [IGNORE_CASE] :
{
<BASE: "base">
| <PREFIX: "prefix">
| <RECOMMEND: "recommend">
| <SELECT: "select">
| <CONSTRUCT: "construct">
| <DESCRIBE: "describe">
| <ASK: "ask">
[...]
| <WHERE: "where">
| <BASED_ON: "based on">
| <MEASURES: "measures">
[...]
}

[...]
void Query() #void :{}
{ RecommendQuery() }
```

```
void RecommendQuery() :{}
{ Recommend()
( DatasetClause() )*
WhereClause()
BasedOnClause()
( MeasuresClause() )?
SolutionModifier()
[BindingsClause()] }

[...]
void Recommend() :{}
{ <RECOMMEND>
[
<DISTINCT> {jjtThis.setDistinct(true);} |
<REDUCED> {jjtThis.setReduced(true);} ]
(
<STAR> { jjtThis.setWildcard(true); }  |
( ProjectionElem() )+
) }

[...]
void WhereClause() :{}
{ [<WHERE>] GroupGraphPattern() }

void BasedOnClause() :{}
{ [<BASED_ON>] GroupGraphPattern() }

void MeasuresClause() :{}
{ [<MEASURES>] GroupGraphPattern() }

void SolutionModifier() #void :{}
{ [GroupClause()] [HavingClause()] [OrderClause()] [LimitOffsetClauses()] }

[...]
void GroupClause() :{}
{ <GROUP> <BY> ( GroupCondition() )+ }

void OrderClause() :{}
{ <ORDER> <BY> ( OrderCondition() )+ }

void GroupCondition() :{}
{
FunctionCall()
| BuiltInCall()
| <LPAREN> Expression() [ <AS> Var() ] <RPAREN>
| Var()
}

void HavingClause() :{}
{ <HAVING> Constraint() }

void OrderCondition() :{}
{
[ <ASC> | <DESC> {jjtThis.setAscending(false);}] BrackettedExpression()
| FunctionCall()
| BuiltInCall()
| Var()
}

void LimitOffsetClauses() #void :{}
{
Limit() [ Offset() ] |
Offset() [ Limit() ]
}

void Limit() :
{ Token t; }
{
<LIMIT> t = <INTEGER>
{ jjtThis.setValue(Long.parseLong(t.image)); }
}

void Offset() :
{ Token t; }
{
<OFFSET> t = <INTEGER>
{ jjtThis.setValue(Long.parseLong(t.image)); }
}
[...]
```

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Bibliography

[1] Linked Data. http://www.w3.org/standards/semanticweb/data. [Online; accessed 28-December-2013].

[2] SWEO Community Project: Linking Open Data on the Semantic Web - Statistics on Data sets - . http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics. [Online; accessed 28-December-2013].

[3] OpenRDF Sesame Core 2.7.9 API. . http://openrdf.callimachus.net/sesame/2.7/apidocs/index.html, 2013.

[4] Gediminas Adomavicius and Alexander Tuzhilin. Multidimensional recommender systems: A data warehousing approach. In *WELCOM*, pages 180–192, 2001.

[5] Gediminas Adomavicius and Alexander 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.

[6] Gediminas Adomavicius, Alexander Tuzhilin, and Rong Zheng. Request: A query language for customizing recommendations. *Information Systems Research*, 22(1):99–117, 2011.

[7] Kemafor Anyanwu. A vision for sparql multi-query optimization on mapreduce. In *ICDE Workshops*, pages 25–26, 2013.

[8] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. On the semantics of sparql. In *Semantic Web Information Management*, pages 281–307. 2009.

[9] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, pages 722–735, 2007.

[10] Punam Bedi, Harmeet Kaur, and Sudeep Marwaha. Trust based recommender system for semantic web. In *IJCAI*, pages 2677–2682, 2007.

[11] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: Scientific American. *Scientific American*, May 2001.

[12] Abraham Bernstein and Christoph Kiefer. Imprecise rdql: towards generic retrieval in ontologies using similarity joins. In *SAC*, pages 1684–1689, 2006.

[13] Robin Burke. *Recommender Systems: An Introduction*, by dietmar jannach, markus zanker, alexander felfernig, and gerhard friedrichcambridge university press, 2011, 336 pages. isbn: 978-0-521-49336-9. *Int. J. Hum. Comput. Interaction*, 28(1):72–73, 2012.

[14] Iván Cantador, Pablo Castells, and Alejandro Bellogín. An enhanced semantic layer for hybrid recommender systems: Application to news recommendation. *Int. J. Semantic Web Inf. Syst.*, 7(1):44–78, 2011.

[15] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013. http://www.w3.org/TR/sparql11-query/, 2013.

[16] Benjamin Heitmann and Conor Hayes. Using linked data to build open, collaborative recommender systems. In *AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*, 2010.

[17] Thomas Hornung, Cai-Nicolas Ziegler, Simon Franz, Martin Przyjaciel-Zablocki, Alexander Schätzle, and Georg Lausen. Evaluating hybrid music recommender systems. In *Web Intelligence*, pages 57–64, 2013.

[18] Benedikt Kämpgen, Sean O'Riain, and Andreas Harth. Interacting with statistical linked data via olap operations. In V. Lopez E. Motta P. Buitelaar R. Cyganiak (eds.) C. Unger, P. Cimiano, editor, *Proceedings of Interacting with Linked Data (ILD 2012), workshop co-located with the 9th Extended Semantic Web Conference*, pages 36–49. CEUR-WS.org (http://ceur-ws.org/Vol-913), Mai 2012.

[19] Bradley N. Miller, Istvan Albert, Shyong K. Lam, Joseph A. Konstan, and John Riedl. Movielens unplugged: experiences with an occasionally connected recommender system. In *IUI*, pages 263–266, 2003.

[20] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. Linked open data to support content-based recommender systems. In *I-SEMANTICS*, pages 1–8, 2012.

[21] Simon Parsons. *A Semantic Web Primer*, second edition by antoniou grigoris and harmelen frank van, mit press, 288 pp. *Knowledge Eng. Review*, 24(4):415, 2009.

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

[23] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference*, pages 30–43, 2006.

[24] Ladislav Peska and Peter Vojtás. Enhancing recommender system with linked open data. In *FQAS*, pages 483–494, 2013.

[25] Sean Policarpio, Soren Brunk, and Giovanni 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 the European Conference on Artificial Intelligence (ECAI)*, August 2012.

[26] Luis Polo, Iván Mínguez, Diego Berrueta, Carlos Ruiz, and José Manuél Gómez-Pérez. User preferences in the web of data. *Semantic Web*, 5(1):67–75, 2014.

[27] Shelley Powers. *Practical RDF - solving problems with the resource description framework*. O'Reilly, 2003.

[28] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/, 2008.

[29] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. Rdfpath: Path query processing on large rdf graphs with mapreduce. In *ESWC Workshops*, pages 50–64, 2011.

[30] Mariano Rodriguez-Muro, Martín Rezk, Josef Hardi, Mindaugas Slusnys, Timea Bagosi, and Diego Calvanese. Evaluating sparql-to-sql translation in ontop. In *ORE*, pages 94–100, 2013.

[31] Wolf Siberski, Jeff Z. Pan, and Uwe Thaden. Querying the semantic web with preferences. In *International Semantic Web Conference*, pages 612–624, 2006.

[32] Milan Stankovic, Werner Breitfuss, and Philippe Laublet. Linked-data based suggestion of relevant topics. In *I-SEMANTICS*, pages 49–55, 2011.

[33] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[34] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Evaluation of sparql property paths via recursive sql. In *AMW*, 2013.

[35] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.