Chapter 10
Visual Query Languages

Abstract The social web is becoming increasingly popular and important, because it creates the collective intelligence, which can produce more value than the sum of individuals. The social web uses the Semantic Web technology RDF to describe the social data in a machine-readable way. RDF query languages play certainly an important role in the social data analysis for extracting the collective intelligence. However, constructing such queries is not trivial because the social data are often quite large and assembled from a large number of different sources and because of the lack of structure information like ontologies. In order to solve these challenges, we develop a Visual Query System (VQS) for helping the analysts of social data and other semantic data to formulate such queries easily and exactly. In this VQS, we suggest a condensed data view, a browser-like query creation system for absolute beginners and a Visual Query Language (VQL) for beginners and experienced users. Using the browser-like query creation or the VQL, the analysts of social data and other semantic data can construct queries with no or little syntax knowledge; using the condensed view, they can determine easily what queries should be used. Furthermore, our system also supports a number of other functionalities, for example, precise suggestions to extend and refine existing queries. An online demonstration of our VQS is publicly available at http://www.ifis.uni-luebeck.de/index.php?id=luposdate-demo.

10.1 Motivation

The Social Web fosters collaboration and knowledge sharing and boots the collective intelligence of the society, organizations, and individual people. The significance of the Social Web is already beyond the field of computer science. The Semantic Web promises a machine-understandable web, and provides the capabilities for finding, sorting, and classifying web information, and for inferring new knowledge. The Semantic Web hence offers a promising and potential solution to mining and analyzing the social web.

Therefore, the emerging Social Semantic Web, the application of semantic web technologies to the social web, seems to be a certain evolution. In the Social

S. Groppe, *Data Management and Query Processing in Semantic Web Databases*,
DOI 10.1007/978-3-642-19357-6_10, © Springer-Verlag Berlin Heidelberg 2011

Semantic Web, the social data are described using the Semantic Web data format RDF, and the social data become hence machine-understandable.

Analysis of social data plays a critical role in, for example, extracting the collective intelligence, obtaining insights, making decisions, and solving problems. The Social Semantic Web gathers a huge amount of semantic data. For example, DBpedia (http://wiki.dbpedia.org/About) has extracted 479 million RDF triples from the social website Wikipedia (http://en.wikipedia.org/wiki/Main_Page). Analyzing such a large amount of data by browsing either the datasets or their visualizations is impractical. RDF query languages such as SPARQL (Prud'hommeaux and Seaborne 2008) are obviously an important tool for the analysis of large-scale social data.

Ontologies are used to describe the structure of the Semantic Web data. Due to the open world assumption of the Semantic Web, the data may contain structures, which are not described by the given ontologies. Ontologies are also often not given for considered data. This is especially true for the social data, which are contributed by a huge number of individual participants.

In order to formulate SPARQL queries for analyzing social data or other semantic data, the analysts have to hence look into the data as well as its ontology, besides having the knowledge of the SPARQL language. Looking into a large amount of social data is obviously inefficient and impractical. Therefore, in this work, we propose a condensed data view, which describes the structure of the initial large data, but uses a compact representation of the data.

We also develop a browser-like query creation system and a visual editor of SPARQL queries, which allows analysts to formulate SPARQL queries with little syntax knowledge of SPARQL. Using different frames for the browser-like creation and the visual editor within the same window, the user can use both approaches in parallel for query creation; that is, all effects during query editing are visible in the browser-like query creation as well as in the visual editor and the user can manipulate the query either in the browser-like query creation frame or in the visual editor. Furthermore, we support to extend and refine queries based on the concrete social data for constructing exacter queries for more precise and efficient data analysis during browser-like query creation as well as visual editing.

In order to demonstrate these approaches to facilitating the (social) data analysis, we develop a Visual Query System (VQS), which includes the following:

- Support of whole SPARQL 1.0.
- Browser-like query creation for absolute beginners.
- Visual editor of SPARQL queries, which hides SPARQL syntax from users and supports the creation of more complicated queries in comparison to browser-like query creation.
- Visual browsing and editing of RDF data.
- Condensed data view for avoiding browsing the initial large datasets.
- Extending and refining queries based on query results.
- Import and export of textual SPARQL queries.

10.2 Related Work

Query-by-form is used in many web applications for simple data access; users are provided a form, all fields of which are seen as parameters of a fixed query. Query-by-form is neither flexible nor expressive, as a form needs to be developed for each query.

Query-by-example (Zloof 1977) allows users to formulate their queries by selecting the names of the queried relations and fields and by entering keywords into a table.

As many databases are modeled conceptually using EER or ORM diagrams, a user can also query these databases starting from those diagrams by using *conceptual query languages*. Examples are EER-based (Czejdo et al. 1987; Parent and Spaccapietra 1989) and ORM-based (De Troyer et al. 1988; Hofstede et al. 1995) approaches. Ontology-aware VQSs (Russell and Smart 2008; Fadhil and Haarslev 2006; Vdovjak et al. 2003; Borsje and Embregts 2006; Catarci et al. 2004; OpenLink 2010) also belong to this class of visual query languages.

We classify the existing *VQSs* for the Semantic Web according to the support of suggestions for extending and refining queries. The VQSs, which provide suggestions for extending and refining queries, include vSPARQL/NITELIGHT (Russell and Smart 2008; Smart et al. 2008), GLOO (Fadhil and Haarslev 2006), EROS (Vdovjak et al. 2003), SPARQLViz (Borsje and Embregts 2006), SEWASIE (Catarci et al. 2004), and iSPARQL/OpenLink (OpenLink 2010). Other VQSs do not support suggestions, like RDF-GL/SPARQLinG (Hogenboom et al. 2010), MashQL (Jarrar and Dikaiakos 2008, 2009), and the one from DERI (Harth et al. 2006). All the contributions for extending and refining a query are based on ontologies. In comparison, our VQS LUPOSDATE-VEdit extends and refines a query based on concrete data.

The Semantic Web VQSs can be also classified according to the supported query languages. Some of the VQSs support SPARQL as query language (e.g., Borsje and Embregts 2006; Jarrar and Dikaiakos 2008, 2009; Hogenboom et al. 2010; Russell and Smart 2008; OpenLink 2010). The rest of the VQSs supports other query languages like triple pattern queries (Harth et al. 2006), RQL (Vdovjak et al. 2003), nRQL (Fadhil and Haarslev 2006), and conjunctive queries (Catarci et al. 2004). All the VQSs for SPARQL only support a smaller subset of SPARQL 1.0, whereas our LUPOSDATE VQS supports the whole SPARQL 1.0.

Other semantic web approaches use visual scripting languages, such as SPARQLMotion (SPARQLMotion 2008), Konduit (Möller et al. 2008a, b) and Deri Pipes (Tummarello et al. 2007). In these approaches, visual boxes and lines are used to connect different query and transformation steps, but the embedded queries must be inserted and edited in a textual form. This chapter contains contributions from (Groppe et al. 2011).

10.3 RDF Visual Editor

The core element of RDF data is the RDF triple *s p o*, where *s* is called the subject, *p* the predicate, and *o* the object of the RDF triple. A set of RDF triples builds a directed graph, called RDF graph. In an RDF graph, the subject and object are nodes and the predicate is a directed edge from the subject to the object. Therefore, the visual representation of RDF triples should conform to the RDF graph.

Table 10.1 contains textual and visual representations of two RDF triples. The visual representation is actually a screenshot of our RDF visual editor. In addition to browsing data, the RDF visual graphs can be used to update data easily. For example, the button + is used to add a triple and – to delete a triple easily.

10.4 SPARQL Visual Editor

The natural way to visualize the triple patterns of SPARQL should be analogous to the way for RDF triples. Table 10.2 contains the textual and visual representations of a SPARQL query. Again, the visual representation is a snapshot of our SPARQL visual editor. Using check and combo boxes in the visualization, users can immediately see the features of SPARQL without the need of learning the complex SPARQL syntax.

Our SPARQL visual editor allows the modification of queries in a similar way for RDF triples: using + to add a triple pattern or a new modifier; using – for removing. Furthermore, users' modification is parsed and checked immediately after each input, and error information is prompted to users. Consequently, our editor ensures users to construct syntactically correct queries.

10.5 Browser-Like Query Creation

The browser-like query creation (see Fig. 10.1) starts with a query for all data or with a query transformed from a visual or textual query. The result of the current

Table 10.1 Example RDF Data `Records.rdf`

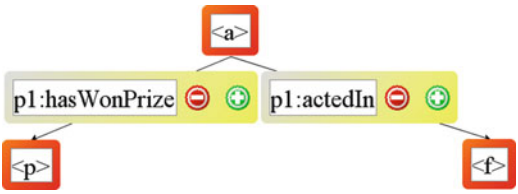
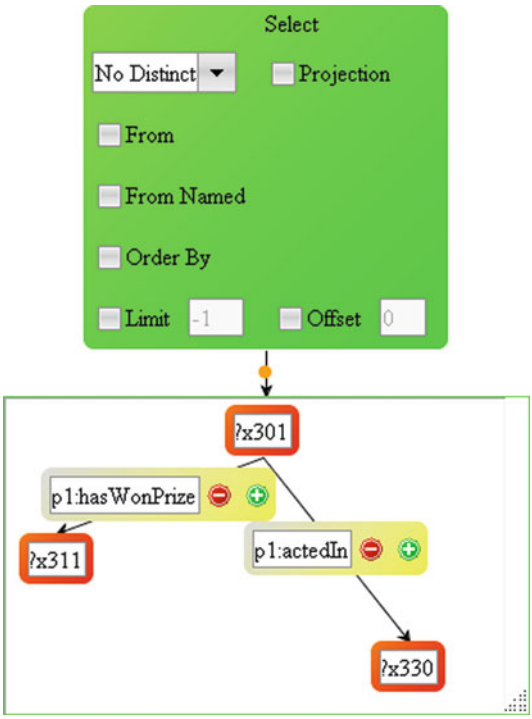
Textual representation	Visual representation
<a> p1:hasWonPrize <p>. <a> p1:actedIn <f>.	

Table 10.2 Example SPARQL Query `DLCRecords.sparql`

Textual representation	Visual representation

```
PREFIX p1: <http://www.p/>

SELECT *
WHERE {
    ?x301 p1:hasWonPrize ?x311.
    ?x301 p1:actedIn ?x330.
}
```



query is presented in the form of a result table. The result table contains buttons to further modify the query. For example, the "Rename" button allows renaming the column name of the result table, that is, renaming the corresponding variable name in the query. The button "Sort" supports to sort the result according to a certain column, the button "Exclude" to hide the column and the button "Refine" to refine the query based on the values of this column. We will explain the refinement of queries in later sections. The "Filter" button supports to filter the result table according to the values of a column and excludes the other rows in the result table. The browser-like query creation also allows to go back to previously created queries (button "<") and then to return to the later created queries (button ">").

The advantage of the browser-like query creation is that it is very easy to use, features are always visible, and no knowledge of the SPARQL syntax is necessary. Furthermore, users often like to work on the concrete data rather than on an abstract query seeing immediately the effect of the query modification regarding a correct query result as hint for a correct query. The disadvantage is that not all features of SPARQL can be provided by an easy graphical user interface like it is the case

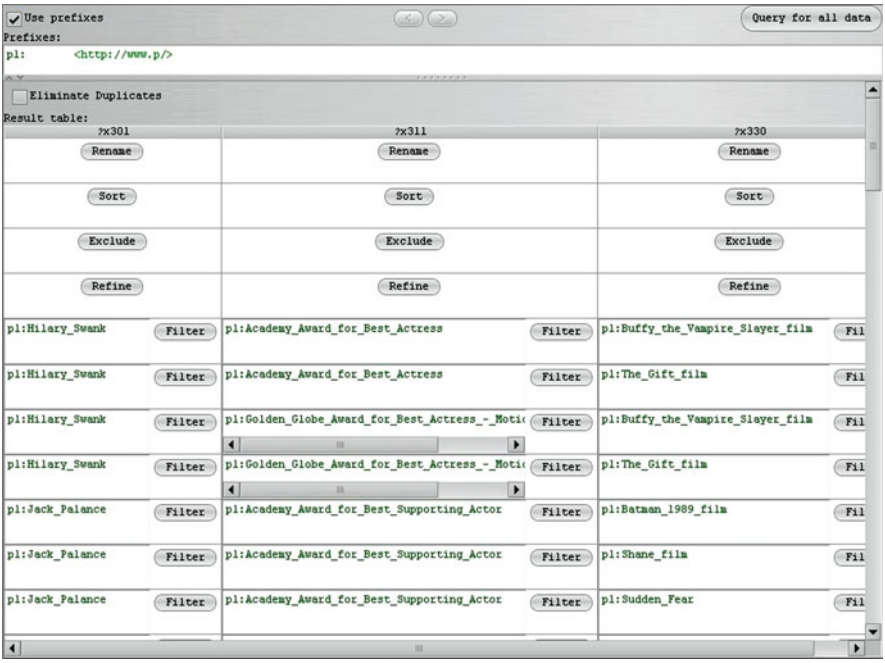


Fig. 10.1 The browser-like query creation

during visual query editing. For example, the union of two (sub) queries cannot be created with the browser-like query creation approach. However, such a query can be taken over from the visual query editor and the query can be further modified with the browser-like query creation approach.

10.6 Generating Condensed Data View

A condensed data view provides an efficient way to see the data structure and to get an overview of the original large datasets. Using the condensed view, one can easily determine the exact queries for the concrete purposes.

A condensed view is generated by integrating the nodes of the RDF graph with certain common features into one node. We use a stepwise approach to condensing data. The main steps are as follows:

Step 1. integrating all *leafs* with the same subject into one node (see Fig. 10.2a). A leaf is an object with only one incoming and without outgoing edges. We condense first all leafs in order to condense the information of a subject into one node, which is otherwise only displayed in a space-consuming way, and to preserve the information of long paths in the data.

Step 2. integrating all objects with the same subject and predicate into one node (see Fig. 10.2b). The structure of the data related to objects with the same predicate

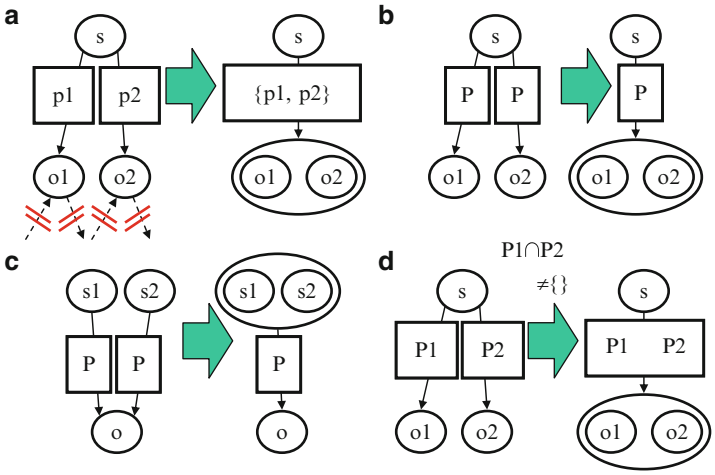


Fig. 10.2 Condensing steps

are often the same or similar, such that this condensing strategy often yields good results.

Step 3. integrating all subjects with the same predicates and objects into one node (see Fig. 10.2c). Similar remarks as for Step 2 also apply to the quality of condensing for this step.

Step 4. integrating the objects with the same subject and at least one predicate in common into one node (see Fig. 10.2d). This step condenses further allowing a sparse representation of the data.

Figure 10.2 describes the condensing steps and Fig. 10.3 presents the condensed view of the Yago data (Suchanek et al. 2007). All the four steps can be processed in reasonable time even for large datasets.

10.7 Refining Queries

Our VQS efficiently supports refinement of queries by a *suggestion functionality* in our SPARQL visual editor. The refined queries can retrieve more exact results for efficient data analysis.

In order to refine a query, the user selects the related node from the SPARQL graph in the SPARQL visual editor, for example, the node for the variable *?x330* in Table 10.2, or presses the button "Refine" in the browser-like query creation frame (see Fig. 10.1). Afterward, we create two suggestion-computing queries (see Sect. 10.8) and evaluate them on the RDF data. According to the evaluation results of these queries, our system can suggest the triple patterns related with the node for *?x330*. Among these suggested triple patterns, those with *?x330* as object are called *preceding triple patterns*, and those with *?x330* as subject are called *succeeding*

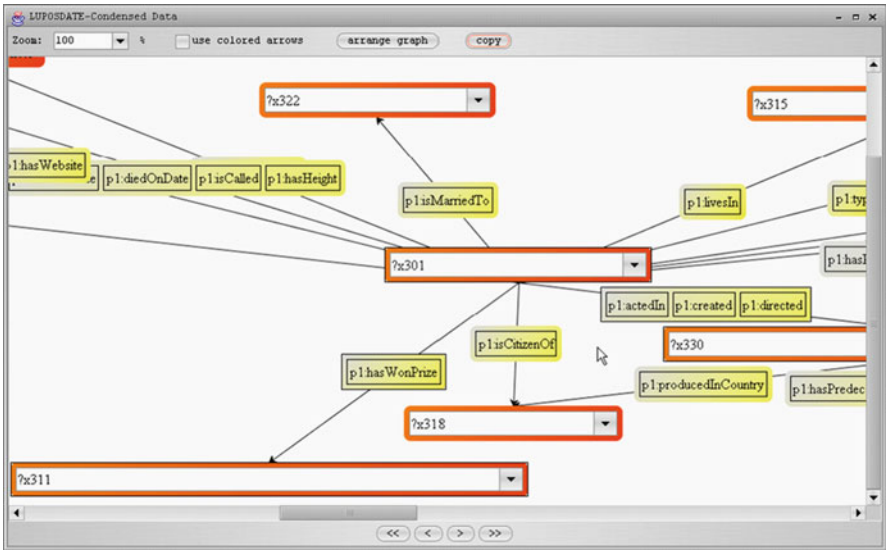


Fig. 10.3 Condensed data view of Yago data

triple patterns. We will explain how to retrieve the suggested triple patterns in detail in Sect. 10.9.

Our system will automatically insert the suggested triple patterns selected by users into the query. Figure 10.4 presents the suggestion dialog for the node *?x330*.

10.8 Query Formulation Demo

Assume that there is a kind of users, who is neither familiar with the structure of the data nor with the SPARQL query language for whatever reasons. We want to demonstrate by a simple example how such kind of users can easily create the queries by using our VQS.

In order to formulate queries for, for example, Yago data, a user first browses the condensed view of the Yago data. In order to obtain the condensed view, the user selects the Yago data and then clicks "*Condense data*" from the main window. A condensed view window is popped, which contains the condensed data views from different condensing steps. Figure 10.3 presents a condensed data view of the Yago data (Suchanek et al. 2007).

The user is interested in the actors, which have won a prize, and in the films, in which the actors acted. Thus, the user selects the corresponding nodes in the condensed view and copies them by clicking *Copy* in the condensed view window. Afterwards, the user opens the query editor from the main window, chooses a

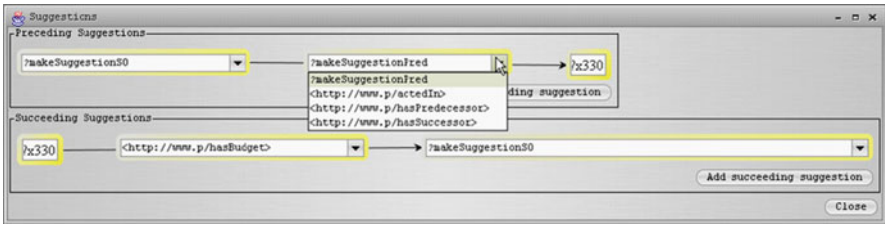


Fig. 10.4 Suggestion dialog for the variable *?x330*

SELECT query, and pastes the nodes into the query editor by clicking *Edit – Paste*. The generated visual query is described in Table 10.2 and its query result is displayed in the browser-like query creation frame (see Fig. 10.1).

After investigating the query result, the user wants to have more information about the films, and thus the user should refine the query. In order to refine queries easily, the user uses our suggestion functionality in either the browser-like query creation approach (button "Refine" in the column for variable *?x330*) or the SPARQL visual editor. In the SPARQL visual editor, the user needs to call *Edit – Make suggestions*. After that, the user clicks the node for the variable *?x330* (which selects films), and a suggestion window for query refinement is popped (see Fig. 10.4).

From the suggestion dialog, the user chooses a succeeding triple pattern, *?x330 p1:hasBudget ?b*. The triple pattern is automatically integrated into the edited query in the visual editor as well as in the browser-like query creation frame after clicking the corresponding button *Add succeeding suggestion*.

Furthermore, the user also wants to see the querying result sorted according the budgets. By only looking at the visual editor, the user can easily see that the feature *Order By* is the one wanted. In the browser-like query creation frame it is even more obvious to click on the "Sort" button in the column for the variable *?b*. Figure 10.5 presents the final visual query.

10.9 Computation of Suggested Triple Patterns for Query Refinement

Our VQS supports the suggesting functionality for the refinement and extension of queries. Our suggesting functionality computes a group of triple patterns related to the node marked by the user. These triple patterns are recommended to the user for query extension and refinement. In order to determine this group of triple patterns, we first create two SPARQL queries, which we name *suggestion-computing queries*.

For example, a user wants the suggested triple patterns related to the node *?x330* in Table 10.2. In order to compute these triple patterns, we first construct the following two suggestion-computing queries Q1 and Q2:

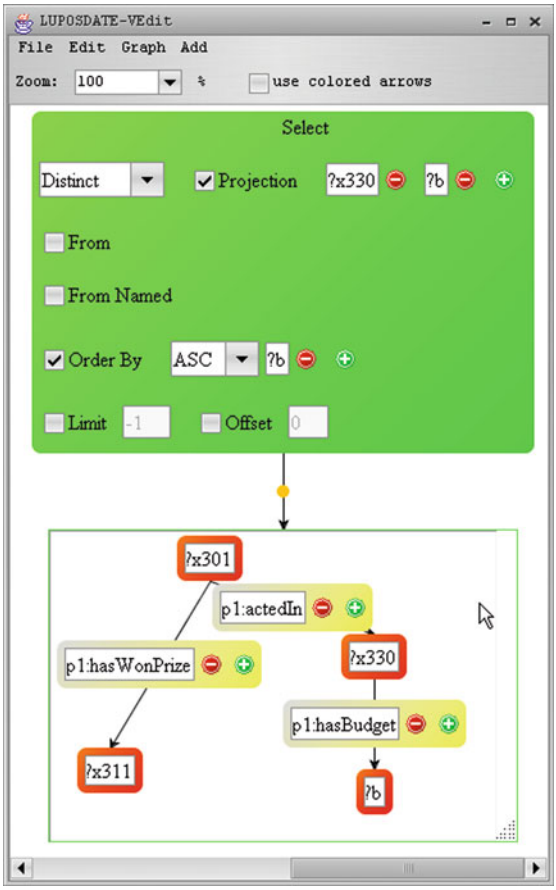


Fig. 10.5 Final visual query

Q1:	Q2:
PREFIX p1: <http://www.p/> *SELECT DISTINCT ?x330 ?p ?so* *WHERE {* *?x301 p1:hasWonPrize ?x311.* *?x301 p1:actedIn ?x330.* *?x330 ?p ?so.* *}*	*PREFIX p1: <http://www.p/>* *SELECT DISTINCT ?x330 ?p ?so* *WHERE {* *?x301 p1:hasWonPrize ?x311.* *?x301 p1:actedIn ?x330.* ***?so ?p ?x330***. *}*

Q1 and Q2 are then evaluated on the given RDF data. From the result of Q1, we can generate the succeeding triple patterns related to the node *?x330*. For example, *?x330=<TheFilmII>*, *?p=<http://www.p/hasSuccessor>*, and *?so=<TheFilmIII>* is one result of Q1; on the basis of this result, we can recommend the following succeeding triple patterns:

- *?x330 <http://www.p/hasSuccessor> <TheFilmIII>*,
- *?x330 ?p <TheFilmIII>*, and
- *?x330 <http://www.p/hasSuccessor> ?so*.

Likewise, from the result of Q2, we can obtain the related preceding triple patterns. Figure 10.4 presents the dialog displaying the suggested triple patterns.

10.10 Summary and Conclusions

In this chapter, we present our solution to formulate SPARQL queries for analyzing the semantic data from, for example, the Social Semantic Web. Such data are often large-scale and are collected from a large number of different sources. By supporting a condensed data view and visual query editing as well as browser-like query creation, our VQS allows the users, who are neither familiar with the SPARQL language nor the data structure, to construct queries easily. We define the visual query language, develop the rules of the transformations between textual and visual representations, and support automatic transformations between the textual and visual representations. Furthermore, our system also provides a precise approach to query refinement based on the query result, thus generating more exact queries.