A Visual Aide for Understanding Endpoint Data

Fernando Florenzano[12], Denis Parra[1], Juan L. Reutter[12], and Freddie Venegas[1]

[1] Pontificia Universidad Católica de Chile
[2] Center for Semantic Web research, CL

Abstract. In order to pose queries on SPARQL endpoints, users need to understand the underlying structure of the data that is stored. Unfortunately, and despite the importance of endpoints in the Semantic Web infrastructure, in most (if not all) publicly available endpoints the only way of understanding this structure is by performing a considerable number of probe queries, perhaps inspired in a few examples that are also made available.
This paper looks into the problem of providing additional information for SPARQL-fluent users that need to query a RDF dataset they are not familiar with. We set up to understand what is the essential information that a user needs to query a SPARQL dataset, and then propose a visualisation that can effectively help users learn this information. This visualisation consists of a labelled graph whose nodes are the different types of entities in the RDF dataset, and where two types are related if entities of these types appear related in the RDF dataset. We illustrate our visualisation using the Linked Movie Database dataset.

1 Introduction

SPARQL Endpoints are one of the key elements in the current Semantic Web infrastructure. The idea of these endpoints is to allow users to extract information from a remote RDF dataset; the dataset is made available to be queried over the HTTP protocol, and the information must be obtained using SPARQL queries, according to the latest SPARQL specification [7].

From an algorithmic point of view, the problem of querying endpoints (or at least a single endpoint) has been very well studied, and nowadays there is a considerable number of working endpoints that permit querying databases from numerous different domains. See [15] for a list of working endpoints. Today, one can reasonably state that querying an endpoint is an easy task, assuming of course that the user is familiar in SPARQL and that she is familiar with the structure of the dataset in the endpoint.

Unfortunately, this is a pretty strong assumption: even if the endpoint is up and running, and even if the user is an expert in SPARQL, the task of producing a query that extracts the desired information may end up demanding more resources than those needed to actually compute the query. The problem is the unstructured nature of RDF: as there is no real notion of schema, knowing what and how the RDF data is stored is not an easy task. In contrast with relational databases, where users can directly consult the schema information for the names and attributes of tables, with RDF data one has almost no alternative but to understand the data by issuing several probe queries. This behaviour has been confirmed when analysing query logs of different endpoints [3, 14].

As an example of the challenges we face, consider and endpoint for *Linked Movie Database* (LinkedMDB [8]), a database storing information about movies: who starred them, who directed them, when and where were they shot, etc. Imagine now a SPARQL expert trying to obtain the information demanded by the query Q below:

> Q: Return the names of all directors that have also acted in one of their movies.

The landing page of our LinkedMDB endpoint of choice would probably not provide any information on the structure of the data residing in the endpoint. Furthermore, the only guidelines for constructing the appropriate SPARQL query would be a small number of general examples that may not even mention the entities we are looking for. In particular, for our query Q above we would need the following information.

- First, how are directors stored in the database? In RDF one typically identifies resources with types, so one would expect that a pattern of the form {`?x a linkedmdb:director`} would match precisely all directors in our database. However, to be able to produce such a query we need to obtain the precise IRI used in the dataset to indicate the director type, in this case http://data.linkedmdb.org/resource/movie/director.
- Next, how is the connection between actors, directors and movies stored? Assuming we already know how to identify actors, directors and movie resources with SPARQL, we still need information about the way these are linked together. In the case of LinkedMDB, directors are connected to entities of type `film` via the same predicate http://data.linkedmdb.org/resource/movie/director, and similarily for actors and films.
- How to understand when a director and an actor are the same person? There could be several possibilities. For example, there could be a linking triple {`A owl:sameAs D`} between an actor resource `A` and a director resource `D`. But in LinkedMDB this is not stored directly, so we need to query for actors `A` and directors `D` that have the same name. Thus, we also need to understand the way in which the names of actors and directors are stored in the database. Furthermore, there is no way of knowing that an `owl:sameAs` link will not work without an explicit SPARLQ query that looks for the existence of triples of this form.

We argue that users new to RDF datasets would benefit tremendously from a graphical interface that would explicitly give them the information they need to start producing meaningful SPARQL patterns. We want a quick and easy lightweight solution, that can be added onto endpoints without much computational overload. This issue has been recognised before, in e.g., [5, 10, 9, 11, 1, 4, 17], and several visualisation proposals have been made to aid users in performing SPARQL queries.

But how can one identify which of these visualisations has the necessary ingredients to be a good aide for querying endpoints? We try to answer this question by proposing 3 basic requirements that must be fulfilled by any system or visualisation if they aim to be a reasonable help for users posing SPARQL queries. Then, as a proof of concept, we present one visualisation that does fulfil all of our requirements, and explain the different choices made when creating this interface. Thorough the paper we assume familiarity with RDF, SPARQL, and the basics of SPARQL endpoint architecture.

2 Looking for the Precise Meta-Information

In RDF databases the term *schema* is used in two completely different contexts. First there is the notion of RDF schema [2], a number of classes and properties with predefined meaning that serve for structuring RDF data. For example, the keywords `rdf:type` and `rdf:subClassOf` are part of the RDF schema specification, and are used as properties within documents to specify, respectively, that a resource is of a certain type or that a type is a subclass of another type.

But in relational databases, a relational schema not only defines the structure of the data, but also gives us guidelines on how to query the data: the names of the relations, which attributes are associated to each relation, the domains of each of those attributes, etc. RDF schema, being stored in RDF itself, does not automatically provide this information, and thus this second notion of schema is not immediately present when dealing with RDF, and has to be obtained by other means.

Thus the natural question: what kind of meta-information is actually needed to write SPARQL queries? Think of the information that a SPARQL expert would need in order to produce a query that fulfils the task at hand, when confronted with a new RDF dataset she doesn't know. This information can be represented in several ways, so we do not discuss how this information should look like, but rather specify what should be expected from it. To this extent, we propose three requirements that any aide for querying SPARQL must have.

R1: The user should be able to identify all different types of resources in the database, and search for the types she is interested on.

This is the most basic requirement one could think of: if we need to query a certain type of entity (such as actors, directors, or films), we need to know how each of these entities is stored in the database, and how to recognise them by means of patterns. Usually one specifies the types of an entity `e` with triples of the form {`e rdf:type <type>`}, but we we refer to *type* in an abstract way; if RDF schema types are not present then there are other options to classify the resources in a database, such as discovering them as in [13].

The main challenge behind this requirement is that the amount of different types in a database may be so big that simply showing a list of types is useless (for example DBpedia, as reported in http://wiki.dbpedia.org, has more than a hundred thousand different types). Thus, a way of navigating this information has to be present or included in the interface, so that the user can look for the types she needs without being confronted to a list with thousands of entries.

> R2: When given a pair of types, the user should be able to identify how and when are two entities of these types connected to each other.

Even if we can identify entities of a given type, we still need to know how they link with each other. Thus, any meta-information that deems to be useful towards query formulation needs to be able to produce, for each pair of types A and B, all relations that span between an entity of type A and an entity of type B (and vice-versa). For some types we always know that all entities of the given types are connected to each other (like actors and films in LinkedMDB). But sometimes there might only be a few connections between entities of two given types. Thus, we also need a way of specifying how likely is that two entities are connected to each other.

> R3: For each type, the user should be able to identify which attributes are common to entities of this type, and how common is this attribute amongst all entities of the given type.

To put common terms between RDF and the relational models, we define the attributes of an entity `e` as all the properties `p` present in triples of the form {`e p l`}, where `l` is a literal. Attributes give us specific information of entities: if we want to know the name of an actor `a`, then it is probably stored in a triple of the form {`a actor_name <name>`}, and likewise for any other name or numerical attribute associated to `e`. Again, while some attributes may be common for all entities of a given type (such as the titles of films in LinkedMDB), in other case these attributes may appear only in a subset of the entities (such as the duration of films, as less than a fifth of the films in LinkedMDB has this information).

We also note that there is a *scalability* issue which is orthogonal to all of these requirements: small datasets are of course easier to understand, but bigger datasets (in terms of different types and relationships) are harder, and thus the visualisation must work regardless of the size of the dataset. For this reason, it is almost impossible to develop a static visualisation that satisfies these requirements. On the contrary, we believe that the best option is an interactive page when users can be clearly guided so that they see the details only in the information that they are looking for. We describe our approach at building such a system in the following section.

3 Our Approach

Our approach is to compute, in advance, a graph that contains all the information mandated by requirements R1, R2 and R3, so that users can see this information when attempting to query the endpoint. As we have mentioned, our visualisation is not static, but instead allow the users to select the level of granularity they wish to see in this graph. We start in section 3.1 with a high-level description of our system, and then proceed, in section 3.2, to specify the more specific choices that were taken when designing our visualisation. As a running example we use LinkedMDB's dataset, but of course the ideas are independent of any particular dataset used[3]. Our visualisation of LinkedMDB is available at http://jreutter.sitios.ing.uc.cl/VisualRDF.html.

3.1 Overview of the System

Our main visualisation is a labelled, undirected graph, where the nodes are the types of the dataset, and where there is a p-labeled edge between nodes t_1 and t_2 if and only if there are entities u_1 and u_2 such that u_1 is of type t_1, u_2 is of type t_2 and the dataset contains the triple $\{u_1\ p\ u_2\}$. That is, two types are related to each other is there are entities of these types that are connected by a triple in the dataset.

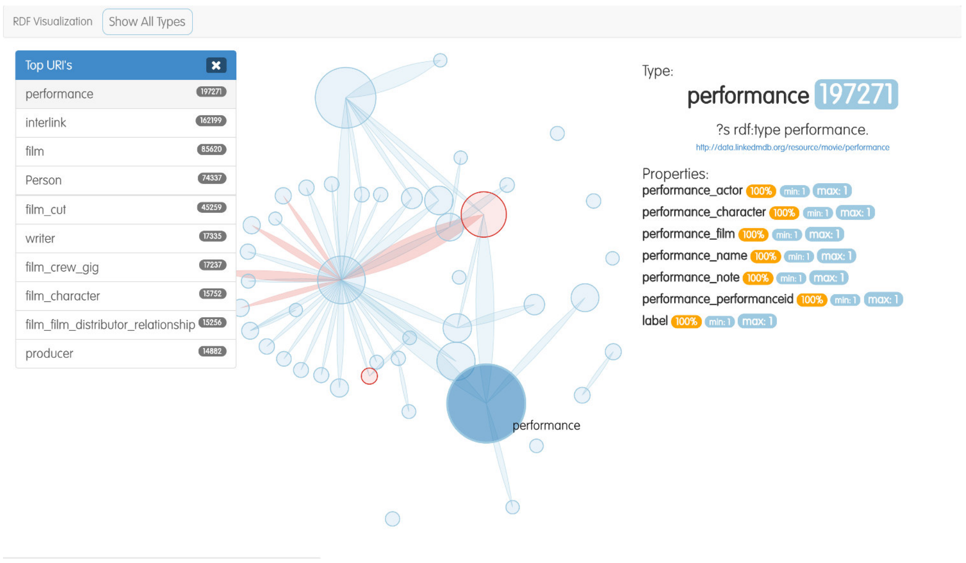
However, showing just this graph is probably not enough for users. In a similar research related to the exploration of social networks, Viegas et al. [16], found that solely relying on the traditional graph representation was a disadvantage compared to a user interface that complemented the graph metaphor with additional visual aids. For this reason we include two panels with additional information: a panel on the left side that allows the users to obtain additional information of the semantic graph as a whole and a panel on the right-side with information about the specific node or edge selected on the main pane. Figure 1 presents a screenshot of our visualisation applied to the LinkedMDB dataset. The main pane contains the aforementioned graph, the left-side panel display the top 10 types ranked according to the number of total entities of this type, and the right-side panel shows specific information about the node selected by the user (in this case *performance*).

In what follows we give a brief description of our visualisation, while discussing the satisfaction of requirements R1, R2 and R3 given in the previous section. Further details ara available in the following subsection.

Hierarchical navigation of types. In most RDF graphs the type assignment is hierarchical. Entities can be of different types, but there is usually a hierarchy between them: there can be two types, A and B that are subclass of a higher type C. In turn, C itself can be a subclass of a type D, and so on. We take advantage of the forest-like shape of the RDF type hierarchy[4] and include in our visualisation the ability to *slice* the type graph in different levels of this hierarchy. Coming back to our example, Figure 2(a) describes the same visualization of LinkedMDB as in Figure 1, but where some nodes are hidden and the emphasis is on the Person node. In LinkedMDB there are several types which are a subclass

[3] We do note that computing the necessary files for our visualisation may take several minutes, or even hours, so it is not possible to offer on-demand visualisations.

[4] Note this is not necessarily the case with more powerful ontologies in the OWL profile, this is in fact an interesting direction for future work.



Top URI's	
performance	197271
interlink	162199
film	85620
Person	74337
film_cut	45259
writer	17335
film_crew_gig	17237
film_character	15752
film_film_distributor_relationship	15256
producer	14882

Type:

performance 197271

?s rdf:type performance.

http://data.linkedmdb.org/resource/movie/performance

Properties:

performance_actor	100%	min: 1	max: 1
performance_character	100%	min: 1	max: 1
performance_film	100%	min: 1	max: 1
performance_name	100%	min: 1	max: 1
performance_note	100%	min: 1	max: 1
performance_performanceid	100%	min: 1	max: 1
label	100%	min: 1	max: 1

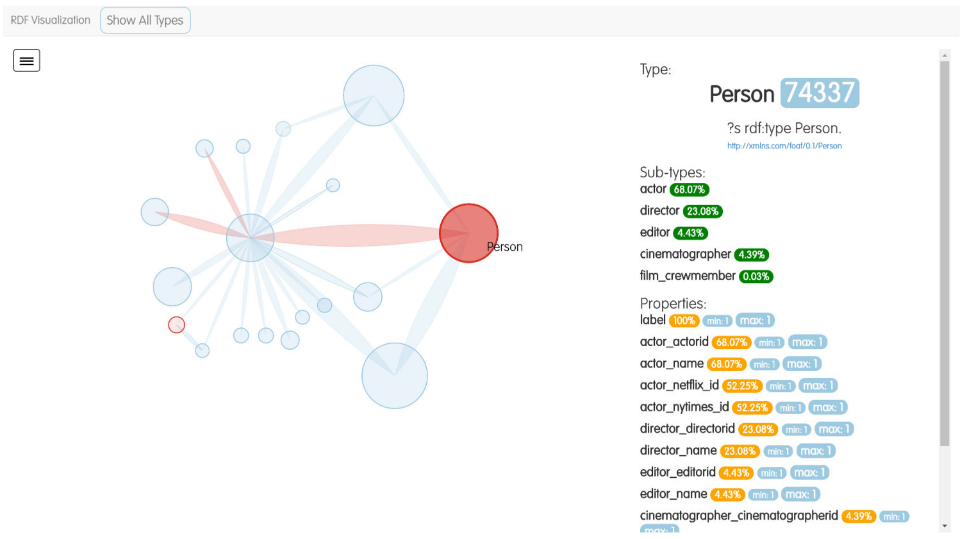
Fig. 1. The top-level visualisation of LinkedMDB's dataset

of Person, and users interested in one of them can drill down on this node, subdividing Person into each of its subclasses, as shown in Figure 2(b).

Together with the left panel, we believe this interactive visualisation fully satisfies requirement R1: all types are present in the graph, but we prevent an overload of information by showing only the top-level types first, allowing the user to produce more fined-grained views on demand.

Navigation and summarisation of relations. It is not strange to find different relations amongst entities of the same type in an RDF dataset. For example, there are several relations between persons and films in LinkedMDB. As we did with nodes, we aggregate all relationships into a single set, and only show the more fine-grained relationships whenever they are requested by the user, by hoovering over the relation. This again prevents information overload, and allows users to visualise only those relations that are interesting, thus satisfying requirement R2. Figure 3 shows an example of a relation that is in fact an aggregation of several different properties, shown in the right-side panel.

Summarisation of attributes. Recall that an attribute of a an entity u is a property that relates u with a certain string or value. For example, in Linked MBD, entities of type performance have attributes such as performance_actor, performance_character, etc., according to Figure 1. As we have specified in requirement R3, attributes are vital to users when asking questions meant to retrieve literal values, such as *What is the name of...?* or *In which year did...?*. However, due to the unstructured nature of RDF data, it is presumable that there will be several attributes that could potentially be associated to each type, and possibly not all entities of the same type have the same attributes. More-



Type:

Person 74337

?s rdf:type Person.

http://xmlns.com/foaf/0.1/Person

Sub-types:

actor 68.07%

director 23.08%

editor 4.43%

cinematographer 4.39%

film_crewmember 0.03%

Properties:

label 100% min: 1 max: 1

actor_actorid 68.07% min: 1 max: 1

actor_name 68.07% min: 1 max: 1

actor_netflix_id 52.25% min: 1 max: 1

actor_nytimes_id 52.25% min: 1 max: 1

director_directorid 23.08% min: 1 max: 1

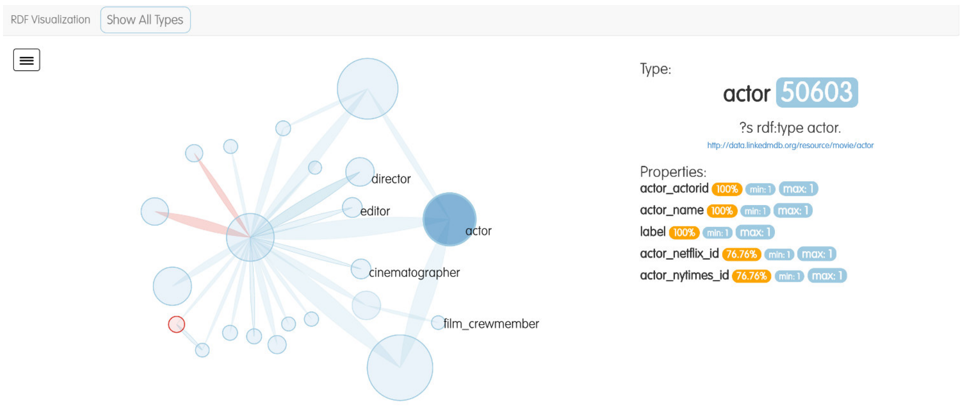
director_name 23.08% min: 1 max: 1

editor_editorid 4.43% min: 1 max: 1

editor_name 4.43% min: 1 max: 1

cinematographer_cinematographerid 4.39% min: 1 max: 1

(a) Graph with Person node



Type:

actor 50603

?s rdf:type actor.

http://data.linkedmdb.org/resource/movie/actor

Properties:

actor_actorid 100% min: 1 max: 1

actor_name 100% min: 1 max: 1

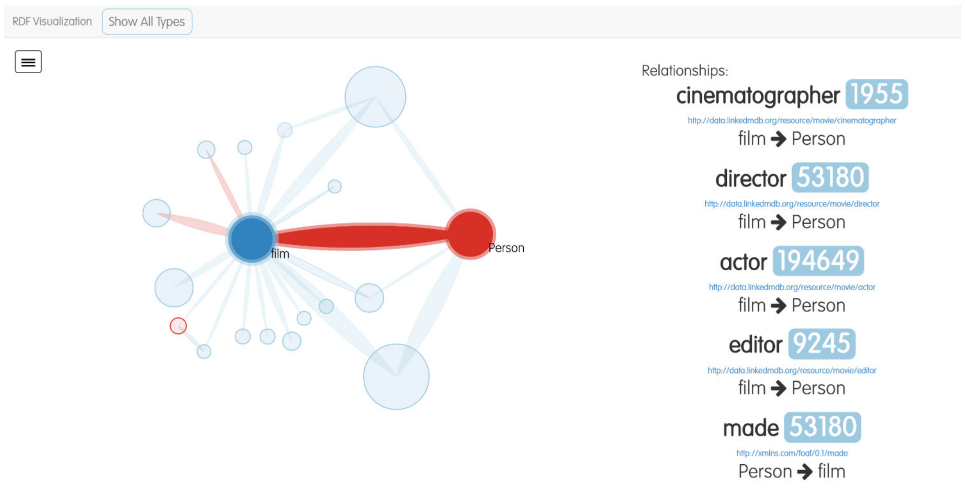
label 100% min: 1 max: 1

actor_netflix_id 76.76% min: 1 max: 1

actor_nytimes_id 76.76% min: 1 max: 1

(b) Result of dividing the Person node

Fig. 2. Visualisation of LinkedMDB when the Person node is subdivided into actor, director, editor, cinematographer and film_crewmember.



Relationships:

cinematographer 1955
http://data.linkedmdb.org/resource/movie/cinematographer
film ➔ Person

director 53180
http://data.linkedmdb.org/resource/movie/director
film ➔ Person

actor 194649
http://data.linkedmdb.org/resource/movie/actor
film ➔ Person

editor 9245
http://data.linkedmdb.org/resource/movie/editor
film ➔ Person

made 53180
http://xmlns.com/foaf/0.1/made
Person ➔ film

Fig. 3. Highlighting the relations between films and persons in LinkedMDB

over, it is impossible to show each of the types attributes in the same graph at the same time. For this reason we construct our visualisation so that, when users hover over or select a particular node in the graph, the right-side panel displays all the information about the attributes of entities of this type. Additionally, we also show the percentage of entities of the chosen type that posses this attribute.

3.2 Visualization

Finally, we shed light on the specific choices and details of our visualisation.

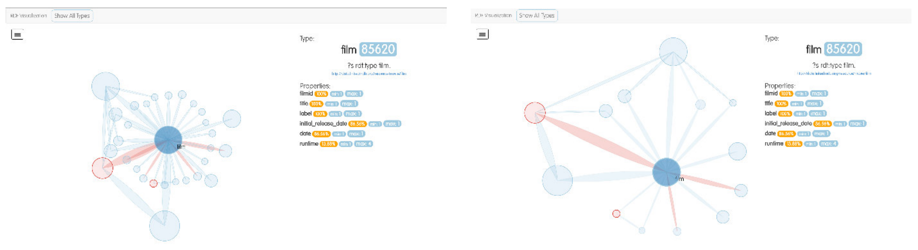
Architecture: The input to the visualisation is a JSON file that needs to be pre-computed from the dataset wanted to be visualised. This JSON contains the information of all types present in the dataset, as well as the relationships between them. The visualisation is separated into a back end, in charge of retrieving the appropriate data from the JSON file, and a front end built with D3 that displays only the nodes and relations pertinent to the current state of the interaction. This separation allows the server to handle the bigger document with all the pre-processed types, and whereas the computational power of producing the visualisation is shifted to the front end, with the double advantage of reducing the amount of computational resources demanded from the server, and hasting the response time of simple operations in the visualisation.

Panes: The visualisation is divided in three panels. The central and main panel is our connection graph. The left-side panel is available on demand, and it shows a box with the 10 most important entities in the current graph. The left-side panel is connected to the graph, so that a selection of one of the entities in the box it is reflected as if it was selected on the graph. For example, in Figure 1, the user is selecting the node *performance* in the left side box, which appears highlighted in the graph. The right-side panel is where the details of the nodes are shown. There are two different visualisations. When a node is selected, the box shows the URI and name of the corresponding type, and all properties that connect at least one entity of this type with a literal (names are assigned automatically from URIs). In turn, for each property we show how many entities of the selected type have this connection, the minimum number of such connections in an entity of this type, and the maximum number of such connections. For example, looking at the first line after Properties in the right-side box in Figure 1, we see that the 100% of entities of type performance have a connection of the form {`e performance_actor <string>`}. Finally, if the type has any subclasses, then the right-side panel also shows all of these subtypes. This can be seen in Figure 2(a), when the Person node is selected. Next, when a relation is selected, the right-side box shows all the different relations that connect entities of this type, as well as the total number of such connections, their URI, and their direction. For example, Figure 3 highlights all connection between entities of type `film` and type `Person`. The right-side box in this case shows, for example, that there are 1955 triples of the form {`e1 cinematographer e2`} where `e1` is of type `film` and `e2` is of type `Person`.

Color: The graph uses essentially two colours: we use red for nodes and relationships that can be navigated (or split) in several subtypes or that contain different relations, and blue for the ordinary nodes and relations. Coming back to LinkedMDB, we see that the node `Person` is shown in red, because one can divide it (into `actor`, `cinematographer`, etc). On the other hand, nodes such as `film` are instead shown in blue, because `film` has no subclasses. The system highlights nodes and/or relationships by applying a more vivid version of the same colour. Furthermore, the name of the node or relationship is always shown when nodes are highlighted.

User interaction: Apart from displaying the left-side panel and highlighting nodes or relationships, by right-clicking the objects a user can interact with the visualisation in the following ways: First, one can hide a particular node and all the edges that connect this particular node. But also one can hide everything that is not related to a given node (keeping only those nodes connected to the selected node), or show all nodes that are related to a given node (in case they where previously hidden). Finally, there is the subdivide operation: Any red node can be subdivided into their immediate subclasses. When this operation is performed the red node dissapears, and for each subtype of the red node we create the corresponding node with their relations. In our LinkedMDB example, Figures 2(a) and 2(b) show the outcome of dividing the `Person` node, creating nodes `actor`, `director`, `editor`, `cinematographer` and `film_crewmember`.

Size and positioning of objects: In both nodes and edges, the total area is proportional to the number of entities (respectively, triples) of a given type



Type:
film 85620
?s rdf:type film.
Properties:
filmid 100% min: 1 max: 1
title 100% min: 1 max: 1
label 100% min: 1 max: 1
initial_release_date 86.56% min: 1 max: 1
date 86.56% min: 1 max: 1
runtime 13.88% min: 1 max: 4

Type:
film 85620
?s rdf:type film.
Properties:
filmid 100% min: 1 max: 1
title 100% min: 1 max: 1
label 100% min: 1 max: 1
initial_release_date 86.56% min: 1 max: 1
date 86.56% min: 1 max: 1
runtime 13.88% min: 1 max: 4

(a) (b)

Fig. 4. Result of hiding and dragging objects in our graph

(relationship). The positioning is also relative to the area, so that the bigger the nodes the farther away they appear. This is done to avoid visual stress by clogging two big nodes together. We take advantage of a number of D3 libraries to produce a graph that can be dragged and repositioned at will. Together with the ability to hide nodes, this facilitates the exploration and analysis of the graph. Figures 4(a) and Figure 4(b) provide an example of this power: The second graph is built from the first by simply hiding a few unimportant nodes and repositioning the important ones.

3.3 Related Work

The system closest to our ideas is LODSight [5], which works in a very similar way than our visualisation. However, LODSight does not have any hierarchical functionality, and thus the visualisation is rather static. We view our system as an alternative for LODSight that provides more opportunities of interaction. Furthermore, [10] proposes an algorithm and a system that is similar in spirit to ours, but with more emphasis on understanding the used ontologies: the system can describe different namespaces, and is in general oriented at a database with different namespaces. However, there is again no hierarchical navigation on classes or relationships, and cannot deal with summarisation when types are not present. Finally, [11] also provides an interesting alternative to visualise endpoints, but the proposal is based more on structured tables than a graph. It remains to see which approach is the best.

More on the profiling side one can highlight ExpLOD [9], that summarises RDF based on the most frequently occurring patterns. This creates a much more specific and smaller view, but on contrast it hides information inside the shape of the patterns. This is again a problem for users that are not familiar with the dataset, as they would need to spend considerable time understanding how the summarisation works. Loupe [12] and ProLOD++ [1] are systems that create a systematic profile of an RDF dataset. As a system they are much more robust, in the sense that it can create a much bigger set of statistics and relationships. However, bigger also means that there is more information to process, and it seems these types of profiling tools aims more at maintainers and regular users of endpoints instead of casual users, and thus it might too much for a user that is just looking to produce a few SPARQL queries: in this case we argue that a more lightweight option is better. Finally, LodLive [4] and RDF Pro (http://www.linkeddatatools.com/rdf-pro-semantic-web) are good tools for visualizing particular pieces of an RDF dataset, but they are not focused on providing an aerial view of the entire dataset, and there are also FedViz [6] and ViziQuer [17], interfaces with a much more ambitious goal of helping users which are not even fluent in SPARQL. Instead, we look for the simplest way of presenting not the full information to the user, but only what is enough to produce the desired SPARQL query. This includes information about, for example, how many entities of a certain type have a particular type of label, which is something that ViziQuer does not show.

4 Future Work

There is still much to do in terms of understanding the best way to facilitate SPARQL query answering, and it would be interesting to continue demonstrating the appropriateness of our requirements. There are two main directions that one could take. First, it is important to analyse previous work on visual information about RDF datasets from the lenses of our requirements: what do they satisfy, what don't they satisfy, and how does this reflects at the time of producing SPARQL queries. In this respect, it would also be useful to understand whether a graph visualisation such as ours is the best one can do in order to fulfil these requirements, or whether one should look for other paradigms. The other direction is to demonstrate, via an empirical experiment, the usefulness of our particular visualisation. A possible way of doing this experiment would be to divide a group of SPARQL experts not familiar with a particular dataset, and assign the same set of tasks to both groups, but where only one group is allowed to use our visualisation while the other is not. The feedback from this experiment could then be used to improve our visualisation. We believe this is an interesting line of work and, to our best knowledge, one that is almost unexplored.

Acknowledgements. Work funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004. The author Denis Parra was funded by Chilean research agency Conicyt, Fondecyt grant number 11150783.

References

1. Z. Abedjan, T. Gruetze, A. Jentzsch, and F. Naumann. Profiling and mining rdf data with prolod++. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1198–1201. IEEE, 2014.
2. D. Brickley and R. V. Guha. {RDF vocabulary description language 1.0: RDF schema}. 2004.

3. C. Buil-Aranda, M. Ugarte, M. Arenas, and M. Dumontier. A preliminary investigation into sparql query complexity and federation in bio2rdf. In *Alberto Mendelzon International Workshop on Foundations of Data Management*, page 196, 2015.
4. D. V. Camarda, S. Mazzini, and A. Antonuccio. Lodlive, exploring the web of data. In *Proceedings of the 8th International Conference on Semantic Systems*, pages 197–200. ACM, 2012.
5. M. Dudáš, V. Svátek, and J. Mynarz. Dataset summary visualization with lodsight. In *European Semantic Web Conference*, pages 36–40. Springer, 2015.
6. S. S. e Zainab, A. Hasnain, M. Saleem, Q. Mehmood, D. Zehra, and S. Decker. Fedviz: A visual interface for sparql queries formulation and execution.
7. S. Harris, A. Seaborne, and E. Prudhommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.
8. O. Hassanzadeh and M. P. Consens. Linked movie data base. In *LDOW*, 2009.
9. S. Khatchadourian and M. P. Consens. Explod: summary-based exploration of interlinking and rdf usage in the linked open data cloud. In *Extended Semantic Web Conference*, pages 272–287. Springer, 2010.
10. S. Kinsella, U. Bojars, A. Harth, J. G. Breslin, and S. Decker. An interactive map of semantic web ontology usage. In *2008 12th International Conference Information Visualisation*, pages 179–184. IEEE, 2008.
11. F. Maali. Sparqture: a more welcoming entry to sparql endpoints. In *Proceedings of the 3rd International Conference on Intelligent Exploration of Semantic Data-Volume 1279*, pages 78–82. CEUR-WS. org, 2014.
12. N. Mihindukulasooriya, M. Poveda-Villalón, R. Garcıa-Castro, and A. Gómez-Pérez. Loupe-an online tool for inspecting datasets in the linked data cloud. In *14th International Semantic Web Conference (ISWC), Posters & Demonstrations Track*, 2015.
13. M.-D. Pham, L. Passing, O. Erling, and P. Boncz. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, pages 864–874. ACM, 2015.
14. M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo. Lsq: The linked sparql queries dataset. In *International Semantic Web Conference*, pages 261–269. Springer, 2015.
15. P.-Y. Vandenbussche, C. B. Aranda, A. Hogan, and J. Umbrich. Monitoring the status of sparql endpoints. In *Proceedings of the 2013th International Conference on Posters & Demonstrations Track-Volume 1035*, pages 81–84. CEUR-WS. org, 2013.
16. F. B. Viégas and J. Donath. Social network visualization: Can we go beyond the graph. In *Workshop on social networks, CSCW*, volume 4, pages 6–10, 2004.
17. M. Zviedris and G. Barzdins. Viziquer: a tool to explore and query sparql endpoints. In *Extended Semantic Web Conference*, pages 441–445. Springer, 2011.