Property Paths over Linked Data: can it be done and how to start?

Jorge Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research

Abstract. One of the advantages that Linked Data offers over the classical database setting is the ability to connect and navigate through different datasets. At the moment the standard mechanism for exploring navigational properties of the Semantic Web data are SPARQL property paths. However, the semantics of property paths is only defined assuming one evaluates them over a single local database, and it is still not clear what is the correct way to implement them over the Web of Linked Data, nor if this is even feasible. In this paper we explore this topic in more depth and gauge the merits of different approaches of executing property paths over Linked Data. To this end we test how property paths perform if the Linked Data is assumed to be available locally, through endpoints, or if it is accessed directly through dereferencing IRIs.

1 Introduction

The *Web of Linked Data* comprises a wide variety of datasets that have been published under a set of best practices and standards that aim to improve the interconnection of these datasets and allow computers to search for information the way humans do with webpages (see e.g. [3]). The adoption of the linked data standard and the creation of this new web has brought up several challenges, one of the most important being how to query the Web of Linked Data.

In order to express queries, the recommendation is to use SPARQL, the default language for querying RDF datasets. Unfortunately, the official semantics of SPARQL assumes that we are dealing with a single dataset, and there is still no standard semantics for SPARQL queries over the Web of Linked Data. The main problem is that the open-world nature of the Web does not couple well with some fragments of SPARQL, as the answers to certain queries may be invalidated when dereferencing additional tuples. For this reason, most previous work has focused on simple, monotonic fragments of SPARQL (see e.g. [4, 10, 9]).

On the other hand, one of the more interesting features of Linked Data is the ability to traverse RDF datasets by navigating from one node to another using the properties connecting them inside an RDF triple, and to cross between different datasets utilizing these types of links. The standard mechanism for exploring navigational properties of RDF data are *Property Paths*, a new feature made available in SPARQL 1.1 [8]. Unfortunately, the semantic of property paths suffers from the same shortcomings as ordinary SPARQL queries when considered over Linked Data, and the topic of evaluating them in this context remains mostly unexplored. There has been some work about navigation within Linked Data (see e.g. [11, 6, 5]), but the first proposal for semantics was published less than a year ago [12]. Therefore, the main focus of this paper is to examine different options for evaluating property paths over Linked Data, and to provide good recommendations of when this is practically feasible, and when we might run into some issues.

To this end, we identified the following four initial approaches for evaluating property path queries over the LOD cloud:

1. **Through endpoints:** The first option we have is to find an appropriate SPARQL endpoint and execute our property path query there. To illustrate this by an example, suppose that we want to find a list of actors with a finite Bacon number[1]. If we decide to use the SPARQL endpoint of the YAGO dataset [17], the desired information (limited to e.g. 100 actors) can be retrieved using the following query:

   ```
   PREFIX yago: <http://yago-knowledge.org/resource/>

   SELECT * WHERE {
     ?x (yago:actedIn/^yago:actedIn)* yago:Kevin_Bacon
   } LIMIT 100
   ```

   the intuition is that actors are connected with movies via the `yago:actedIn` property, and thus a co-actor of Kevin Bacon is someone that witnesses the pattern `{?x yago:actedIn/^yago:actedIn yago:Kevin_Bacon}`. The star is then use to retrieve co-actors of a co-actor of Kevin Bacon, and the co-actors of them, and so on.
2. **Keeping a local copy of the data:** When endpoints fail, there is also an option of simply keeping an (updated) copy of the LOD cloud locally and ask the desired query over this dataset using a SPARQL engine. In the example above we might dispense with the "whole LOD cloud" and simply use YAGO, or another dataset we consider to contain sufficient information to answer our query.
3. **Live querying:** On the other hand, we can rely instead on the Linked Data infrastructure, and perform a search by dereferencing IRIs. In this paper we implement this approach using breadth-first search to evaluate property paths (when at least one end of the path is known, such as in the example above). This approximates the semantics for property paths on Linked Data given in [12], and might currently be the only feasible option available to evaluate property path queries which utilize more than one dataset.
4. **Hybrid approaches:** Finally, we explore how the approaches above can be combined to overcome some of their shortcomings when considered in isolation. In the paper we experiment with two possible combinations:

[1] Actors have Bacon number 1 if they have acted in a movie with Kevin Bacon, and Bacon number n if they have acted together with an actor with Bacon number $n-1$.

- **Endpoints + local data:** Instead of having all of the data locally, we can also try to obtain only the data needed to answer a particular query. For instance, in the query above we do not really need the entire YAGO database, but only the part of it containing `yago:actedIn` links, so a smarter alternative is to retrieve only those links from YAGO, and nothing more, and then run the query locally. In other words, we can issue the query

```
PREFIX yago: <http://yago-knowledge.org/resource/>

CONSTRUCT WHERE {?x yago:actedIn ?y}
```

  to YAGO's public endpoint, which gives us all the information about actors and movies where they acted, and then load this database into a SPARQL engine to run the query.
- **Live querying + endpoints:** When running queries live over Linked Data one immediately observes some issues with dereferencing. This is most notable when working with inverses, since it is well known that publishers include only about a half of the triples where the requested IRI appears as the object [13], which can lead to incomplete answers. One workaround to this is using endpoints in order to retrieve the triples where the needed IRI appears as the object in order to compute the inverse links, so we also explore how this approach fares in practice.

In this paper we do a detailed comparison on the 4 approaches, and based on empirical evidence discuss their strengths and limitations when evaluating property path queries in the context of Linked Data.

Preliminaries: Let $\mathcal{I}$, $\mathcal{L}$, and $\mathcal{B}$ be countably infinite disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively. The set of *RDF terms* $\mathcal{T}$ is $\mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$. An *RDF triple* is a triple (s, p, o) from $\mathcal{T} \times \mathcal{I} \times \mathcal{T}$, where s is called *subject*, p *predicate*, and o *object*. An *(RDF) graph* is a finite set of RDF triples. We assume familiarity with SPARQL queries and specially with Property Paths, as defined in [8]. We also assume basic knowledge of the Linked Data infrastructure, including the notion of IRI dereferencing.

2 Experimental setup

Here we describe the queries used in evaluating the four approaches discussed above, and also describe how each approach is executed.

2.1 Queries

To test the functionalities of property paths over Linked Data we selected three queries. The number is low on purpose: mainly for brevity, but also because most property path queries fall into these categories, thus exhibiting similar behaviour in the evaluation scenarios we present (for many more examples please contact the authors). In our evaluation we focus exclusively on queries utilizing the star operator $*$, since star-free property paths can be rewritten into ordinary SPARQL queries.

Notice that in order to execute a property path query in the context of Linked Data we have to have the starting point (or the ending point) of the path (see e.g. [12, 10, 11, 14] for discussion on the subject). If this is not the case, we immediately exclude all the approaches that do not have the LOD cloud available locally. Because of this, we will have *three versions of each query* (apart the final one), with property paths starting in YAGO [17], DBpedia [2], and WikiData [16], respectively.

Our first query, **Q1**, is the one we mentioned in the introduction, that finds all the actors with a finite Bacon number. The intention of this query is to test how property paths work over a single well-known dataset, using a single property, but going in both directions. In the case of YAGO the query is as follows (for the DBpedia and WikiData versions see the online appendix [1]):

```
SELECT * WHERE
   { yago:Kevin_Bacon (^yago:actedIn/yago:actedIn)* ?x }
```

We draw the second query (**Q2**) from previous work in benchmarking property path implementations [7]. This query retrieves all types of geographic entities that have something to do with Berlin, or some other entity where Berlin is located in, and is intended to utilize several different properties. The YAGO version of the query is:

```
SELECT * WHERE
   { yago:Berlin yago:isLocatedIn*/yago:dealsWith/rdf:type ?y }
```

Our final query (**Q3**) is intended to push the evaluation of property paths into the (true) realm of Linked Data by allowing them to utilize more that one dataset to obtain the answer. To this end, we modify the query (**Q1**) to find the actors with a finite Bacon number in either YAGO or DBpedia[2]. In order to jump between datasets, we take advantage of the `owl:sameAs` triples in them, assuming that two resources connected by this triples are actually the same entity (in this case an actor). This query is important because it can retrieve actors which are not stored YAGO, but are in DBPedia. The query is given below (assuming standard YAGO and DBpedia prefixes):

```
SELECT * WHERE
   { yago:Kevin_Bacon owl:sameAs*/
       ((^dbo:starring/dbo:starring) | owl:sameAs |
                                (yago:actedIn/^yago:actedIn))* ?x }
```

Note also that we take into the account the fact that the property names change (e.g. `actedIn` becomes `starring` in DBpedia), as well as the direction (`actedIn` links movies to their actors, but `starring` links in the opposite direction, from actors to movies).

[2] On can easily add WikiData to this mix, but since WikiData does not support Linked Data through dereferencing we exclude it from this comparison.

2.2 Evaluation Approaches

The four different evaluation approaches are set up as follows:

Using endpoints. For **Q1** and **Q2** we test a version of each query on the appropriate endpoint. In particular:

- For YAGO we use: `https://linkeddata1.calcul.u-psud.fr/sparql`;
- For DBpedia: `http://dbpedia.org/sparql`; and
- For WikiData: `https://query.wikidata.org/`.

In order to execute the query **Q3** we need to use the `SERVICE` functionality of SPARQL, but unfortunately mixing SERVICE with the star operator is not supported.

Local testing. For local testing we set up the Virtuoso 7.20.321 RDF datastore on two different servers:

- **Server 1:** This machine has 4GB of main memory and a Intel i5-4670 CPU processor running up to date Manjaro Linux
- **Server 2:** Is a machine with 192GB memory and 4 AMD Opteron(tm) 6366 H processors running Ubuntu 14 LTS.

To keep the presentation manageable, we ran local experiments only over the YAGO dataset (experiments over DBpedia and WikiData will be available in the journal version of the paper). To this end, we loaded a piece of the YAGO database of size 4.1GB containing the properties used in the tested queries.

Live querying: Our next option obtains the answers by dereferencing IRIs, thus computing them using only the linked data infrastructure (without endpoints).

To compute property paths, we view the Web of Linked Data as an edge labelled graphs, where IRIs are the nodes, and whenever a triple (u, p, v) appears in some dataset, we say that there is a p-labelled edge between u and v (for a full formalisation see e.g. [9]). A path in this graph of the form $u_1 p_1 u_2 p_2 u_3 p_3 \cdots p_{k-1} u_k$ implies that one can deduce the triple (u_1, p_1, u_2) by dereferencing u_1, (u_2, p_2, u_3) by dereferencing u_2, and so on. The idea is then to use the standard breadth first search (BFS) to look for paths such as the above where the corresponding triples satisfy the property path.

For example, in YAGO version of query **Q1** we start by dereferencing the IRI `yago:Kevin_Bacon`, obtaining a document which contains a number of triples of the form $(\texttt{yago:Kevin\_Bacon}, \texttt{yago:actedIn}, u)$ (and many other triples); each of these IRIs u is a movie in which Kevin Bacon acted. Since we do BFS, we first list all such movies, say $\{u_1, \ldots, u_n\}$. One by one, we dereference each u_i, obtaining documents which contain triples of the form $(a, \texttt{yago:actedIn}, u_i)$, telling us that a is a co-star of Kevin Bacon, and thus we can output it as part of our answer. The BFS algorithm continues, dereferencing now each new actor a that is found, obtaining new movies, new actors, and so on.

Note that, unless we are assured to obtain all relevant triples when dereferencing (as happens with e.g. DBpedia, but not YAGO), this procedure is not guaranteed to obtain all the answers for the query, and thus in a sense our BFS algorithm is simply an approximation of the query. This is especially the case when we consider the inverse links (such as the `^yago:actedIn` in **Q1** above), as it is well known that publishers include only about a half of the triples where the requested IRI appears as the object [13]. We will try to partially remedy this issue with one of our hybrid approaches.

We implement the BFS-based evaluation in Python using the RDFlib library to dereference IRIs. All the experiments using this approach were ran on **Server 1**, but since the resources needed here are very low, similar results can be obtained on a much weaker machine.

Hybrid Approach 1 (Endpoints + local data): To avoid downloading a large amount of data which will not be used and then computing our queries over an unnecessarily big database, we use endpoints to obtain only the data needed in our queries. More precisely, if a property path uses the set {`p1`, . . . , `pn`} of properties, we issue the following query (including of course the relevant prefixes):

```
CONSTRUCT {?x1 p1 ?y1 . .... . ?xn pn ?yn}
WHERE {
    {?x1 p1 ?y1} UNION ... UNION {?xn pn ?yn}
}
```

The endpoint will give us a turtle file with the constructed triples. These are then loaded into Virtuoso on **Server 1** and **Server 2** and queries **Q1**, **Q2** and **Q3** are executed using a database containing only this data and nothing more. As in the case of local testing, we only consider the YAGO version of the queries.

Hybrid Approach 2 (Live querying + endpoints): Finally, to reduce the incomplete information resulting when the dereferenced IRI does not contain all inverses in the BFS-based implementation, we can use endpoints. More precisely, while running the BFS algorithm above, if we are processing the label $\hat{}p$ (that is, we want to follow p in the reverse direction), each time we dereference an IRI u we do not consider only the triples obtained by dereferencing, but also the answers to the SPARQL query `SELECT ?s WHERE {?s p u}` posed over YAGO and/or DBPedia endpoint (depending on the query). For instance in the YAGO version of **Q1** we add the answer to `SELECT ?s WHERE {?s yago:actedIn u}` to the triples obtained by dereferencing u, each time we want to traverse this property in reverse (from u). In our extended BFS implementation this is hard-coded into the queries so that they consult the appropriate endpoint each time we need to process an inverse edge and the tests are run on **Server 2**.

We would like to note that the community is already building an infrastructure that would eliminate this mismatch: the *Linked Data Fragments* initiative [15], which aims to study different ways of publishing linked data on the web. Specifically, one of the proposals of this initiative is to build an infrastructure that can support the answer of any SPARQL triple pattern [14].

3 Analysis of the results

In this section we report on our findings when experiments from Section 2 were executed according the strategies described in the introduction. We report (when possible) the amount of memory used, total execution time, and the number of triples received while executing the data. Next, we discuss the merits and shortcomings of each approach in more detail. We would like to stress once again that this analysis applies only to property path queries (using the star operator $*$), and not to general SPARQL queries.

3.1 Using endpoints

The results of our endpoints run are given in Table 1. As we see, our experiments show that property path queries are often not supported on current implementations of SPARQL endpoints. For both DBpedia and YAGO, the execution of **Q1** does not complete due to an exceeded memory limit (1GB), and on WikiData due to an exceeded time limit. Further experimentation shows that the implementation of the star operator is the likely culprit[3]; namely, when one rewrites this query into a union of patters which obtain actors with a Bacon number 1, 2, 3, all the way up to 7, the resulting pattern[4] executes with no substantial problems. Next, the query **Q3** is not supported, since querying two datasets inside a single property path is not possible, even using the `SERVICE` operator. On the other hand, the query **Q2** runs very well on Yago and Wikidata, and the execution is almost instantaneous, which suggests that when there are not that many intermediate results, current SPARQL implementations can handle property paths without problem.

Endpoint	Query	finishes
Yago	**Q1**	no
Yago	**Q2**	yes (<1 sec)
DBpedia	**Q1**	no
DBpedia	**Q2**	no
Wikidata	**Q1**	no
Wikidata	**Q2**	yes (<1 sec)

Table 1. Summary of queries when posed over three popular endpoints. We do not show **Q3** because the `SERVICE` operator is not supported in any of these endpoints.

Overall, it seems that running property paths on endpoints is costly in terms of the memory used (although we could not measure the exact amount of memory and processor time), and full use of Linked Data is not really supported. On

[3] An alternative interpretation related to the semantics of property paths is given in the online appendix [1].

[4] Although this is not the complete query it is a good approximation.

the other hand, when queries do run, they execute almost instantaneously, so using endpoints seems to be a good alternative for property path queries which are based in a single dataset and have relatively few intermediate results. In summary, we can describe the pros and cons of this approach as follows:

Pros: Easy to use, virtually free for the user (in terms of required resources), when it works it works well.

Cons: Unreliable, often not fully supported, based on a single dataset.

3.2 Keeping the data locally

Even before executing the queries we can see that this approach to evaluating property paths is extremely costly in terms of the resources required, since at the very minimum one needs a dedicated machine. Next comes the cost of storage and data transfer. The most complete option would be to keep an updated copy of the entire LOD cloud and run the experiments on this dataset. Of course, if Linked Data is to become a success, this solution would simply not be feasible for a single user because of the sheer volume of the data available, and might be supported only by a big centralised service provider. Next, even if we settle for a single dataset this can provide costly since e.g. YAGO weighs around 100GB, so downloading it on a regular basis is time consuming. Even the small part of YAGO we used in our testing (parts of core, taxonomy and geonames) takes around 4.1GB, which is not a trivial amount of data to transfer over the Web. This is also one of the reasons we decided to run the tests only on the YAGO dataset, and leave the other two sources for future work.

The behaviour is consistent to what we saw in the endpoints: the systems cannot cope with neither **Q1** nor **Q3**, while **Q2** runs only on the high level server (in our commodity server (**Server 1**) we could not even load the database). The memory cost when computing **Q2** was approximately 6GB.

Overall, we can conclude that having Virtuoso locally with all the data necessary to answer the queries basically simulates what is happening in endpoints, but it does allow us a higher degree of control, and the ability to make use of additional data. On the other hand, the total cost of maintaining this solution is quite high even if we only want to execute simple queries such as the ones we propose, as the total amount of transmitted data is rather big, and the computational power needed to make this approach run substantial. Despite of this, once SPARQL engines are capable to run transitive closure efficiently (or use more memory for intermediate results), the solution might have some merits for a big service trying to centralise Linked Data. We summarise the pros and cons of this approach as follows:

Pros: Full control over query execution, reliable, when it works it works well.

Cons: Requires a lot of resources, costly to keep it updated, performance seems to suffer the same issues as endpoints do.

3.3 Live querying using BFS

Here we run the BFS-based algorithm for evaluating property paths by fetching the required documents by dereferencing IRIs. However, as we have discussed, the main shortcoming is the fact that the answers we obtain are rather incomplete (even if we wait for all possible answers), mainly due to the fact that dereferencing an IRI does not give us all the triples where it occurs as an object (this fact is well documented over Linked Data, see e.g. [13]). In particular, the YAGO linked data architecture does not deliver any inverse triples for our queries, so all that could be retrieved was the starting point of the query. Worse, Wikidata does not even support IRI dereferencing, so we could not access their data in this way.

On the other hand, DBpedia returns much more complete data (see Table 2); and the system memory usage, the amount of transmitted data and total query execution time are very low, indicating that this solution can be quite efficient to answer our queries, and one of the advantages it has over a SPARQL engine is that once a single answer is found it can be returned immediately.

Starting Dataset	query	system memory	time (sec.)	transmitted triples
DBPedia	**Q1**	77MB	36	74015
DBPedia	**Q2**	57MB	15	22591
DBPedia	**Q3**	80MB	182	82436

Table 2. Results of running a BFS search over DBPedia until 1000 answers are computed. Results are not given for YAGO because only one answer is selected due to the lack of inverses, nor for Wikidata because linked data is not supported.

Overall, we can see that this is a lightweight solution requiring very few triples to be transmitted, and not a lot of system memory, but the query answer times can be rather long, since they largely depend on internet traffic and server response times. We summarise the pros and cons of this approach as follows:
Pros: Efficient, cheap, up-to-date, supports multiple datasets, incremental results.
Cons: Incomplete answers (dependent on the triples published as Linked Data), performance depends on Internet traffic.

3.4 Hybrid approach 1: Endpoints + local data

Using a `CONSTRUCT` query over an endpoint to retrieve only the data needed to answer the query seems to completely avoid the huge download cost factor as opposed to having all the data locally. One would also expect that computing over such a small database would result in more efficient evaluation, but here is where things get more complex.

First, **Q1** returns the same error (1GB limit of temp memory exceeded) as when ran over a much bigger piece of YAGO (on both servers), which is quite surprising taking into the account the fact that the size of the loaded dataset is less than 1MB. On the other hand, **Q2** runs the same as before, taking less than a second on both the laptop and the high level server. As before, since **Q1** did not run, we also do not execute **Q3**.

Overall, we see that not much is gained compared to having the data locally, although less resources are needed, and the amount of transmitted data is substantial increased. The main problem of this approach, however, is that query answers are again incomplete: the `CONSTRUCT` query form in endpoints has a maximum of 10.000 triples, and anything above that is simply not returned, without any warnings, so it is not easy to realise when a statement delivers all triples or when is it just a fraction of them. In summary, we have:

Pros: Reliable, more control over execution than BFS-based approaches, amount of transmitted data is reasonable.

Cons: A lot of computational power is still needed, same issues as having the data locally, reliant on endpoints and Internet traffic, potentially incomplete answers.

3.5 Hybrid approach 2: Endpoints + BFS

As discussed in Subsection 3.3, when computing inverses in property paths, most answers we get are incomplete, since dereferencing an IRI does not give us all the triples where it appears as the object. One workaround to solve this problem is described in Section 2, where we obtain reverse links is done by consulting the appropriate endpoint.

The results of this approach are presented in Table 3. In particular, since we can now use YAGO's inverses (retrieved with the endpoint), we obtain much more answers that with the vanilla BFS algorithm. We do, however, keep the 1000 answers limit in order to avoid overloading YAGO's infrastructure. Note that it does not make sense to run these algorithms over DBpedia, because it already gives all the inverses when dereferencing. Furthermore, once again we leave Wikidata out since it does not support live querying in this way.

Starting Dataset	query	system memory	time (sec.)	transmitted triples
Yago	**Q1**	62MB	101	52791
Yago	**Q2**	-	-	-
Yago	**Q3**	62MB	101	52791

Table 3. Results of running a BFS search over YAGO, asking endpoints for inverses. Results are the same for **Q1** and **Q3** because YAGO does not include sameAs links when dereferencing triples, and **Q2** is simply not supported in the live part of YAGO. For DBPedia results see Table 2 (they are the same because DBPedia gives all inverses when dereferencing IRIS). Results are not given for Wikidata because linked data is not supported.

In summary, we can obtain many more answers at a slightly higher cost, thus allowing us to keep the advantages of the BFS-based approach, but also retrieve more answers than in that case. Pros and cons are classified as follows:
Pros: Efficient, cheap, up-to-date, supports multiple datasets, more complete answers than pure BFS.
Cons: Still incomplete answers, performance depends on Internet traffic, extra overhead for contacting endpoints, usage of endpoints has to be hard-coded.

4 Conclusions and Future Work

In this paper we tested out several possible options of evaluating property path queries over Linked Data. Overall, we can see that there are no clear winners when it comes to selecting a single approach. On the one hand, the endpoint infrastructure does not support mixing data from different datasets inside a single property path, and the current SPARQL implementations still seem to be having some issues when executing the star operator, possibly because the way they interpret the semantics of property paths. On the other hand, ad-hoc approaches such as live querying seem to be quite cost effective, but they suffer from incomplete answers. Finally, hybrid approaches seem to run better, but also partially suffer from the same issues as their base counterparts.

Although no one-solution-fits-all seems to be available, it seems that once algorithms for evaluating the star operator are improved, running a dedicated server with a SPARQL engine might be a good solution for a high end user wishing to provide centralised support for property paths, especially when they are meant to be executed over a single dataset. Similarly, for a user requiring a more lightweight solution and is not concerned in obtaining only partial answers, using live querying might be a good option.

Therefore, we plan to purse future work on two fronts. First, we wish to look into improving the current algorithms for evaluating the star operator in existing systems, which would allow us to have an out-of-the-box solution for a broad range of uses of property paths. And second, since live querying already with a simple algorithm such as BFS seems to work quite well, and support having more than a single dataset, we plan to look into more sophisticated algorithms for performing a Linked Data search based on a property path query. One promising direction here is to use the famous A* algorithm utilized in AI search and we already have some preliminary results on this (see our online appendix [1]).

References

1. Online appendix. `http://dvrgoc.ing.puc.cl/Planning/COLD/` (2016)
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. Springer (2007)
3. Berners-Lee, T., Bizer, C., Heath, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)

4. Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., Sheets, D.: Tabulator: Exploring and analyzing linked data on the semantic web. In: SWUI Workshop (2006)
5. Fionda, V., Gutierrez, C., Pirrò, G.: The swget portal: Navigating and acting on the web of linked data. J. Web Sem. 26, 29–35 (2014)
6. Fionda, V., Pirrò, G., Gutierrez, C.: NautiLOD: A Formal Language for the Web of Data Graph. TWEB 9(1), 5:1–5:43 (2015)
7. Gubichev, A., Bedathur, S.J., Seufert, S.: Sparqling kleene: fast property paths in RDF-3X. In: GRADES (2013)
8. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C (2013)
9. Hartig, O.: Sparql for a web of linked data: Semantics and computability. In: The Semantic Web: Research and Applications, pp. 8–23. Springer (2012)
10. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the web of linked data. Springer (2009)
11. Hartig, O., Pérez, J.: Ldql: A query language for the web of linked data. In: The Semantic Web-ISWC 2015, pp. 73–91. Springer (2015)
12. Hartig, O., Pirrò, G.: A context-based semantics for SPARQL property paths over the web. In: ESWC 2015. pp. 71–87 (2015)
13. Hogan, A., Umbrich, J., Harth, A., Cyganiak, R., Polleres, A., Decker, S.: An empirical survey of linked data conformance. J. Web Sem. 14, 14–44 (2012)
14. Verborgh, R., Hartig, O., Meester, B.D., Haesendonck, G., Vocht, L.D., Sande, M.V., Cyganiak, R., Colpaert, P., Mannens, E., de Walle, R.V.: Querying datasets on the web with high availability. In: ISWC 2014. pp. 180–196 (2014)
15. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through linked data fragments. In: LDOW (2014)
16. Wikimedia: Wikidata: The Free Knowledge Base. `http://www.wikidata.org` (October 2015)
17. YAGO: A High-Quality Knowledge Base. `http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`

Appendix

A All versions of our queries

Q1: We have one version for YAGO, one for DBpedia and one for WikiData:

– YAGO:

```
SELECT * WHERE
   { yago:Kevin_Bacon (^yago:actedIn/yago:actedIn)* ?x }
```

– DBpedia:

```
SELECT * WHERE
   { dbr:Kevin_Bacon (^dbo:starring/dbo:starring)* ?x }
```

– WikiData:

```
SELECT * WHERE
   { wd:Q3454165 (^wdt:P161/wdt:P161)* ?movie }
```

Q2: We have one version for YAGO, one for DBpedia and one for WikiData:

– YAGO:

```
SELECT * WHERE
   { yago:Berlin yago:isLocatedIn*/yago:dealsWith/rdf:type ?y }
```

– DBpedia:

```
SELECT * WHERE
   { dbr:Berlin (^dbo:locatedInArea/dbo:locatedInArea)*/dct:subject ?z }
```

– WikiData:

```
SELECT * WHERE
   { wd:Q64 wdt:P131*/wdt:P463 ?z }
```

Q3: As explained in the paper we have only one version of this query.

B Kleene star on virtuoso

We have found a simple small database that suggests that Virtuoso does not implement the official semantics of property paths using the Kleene star operator.

Consider the following database,

$$\{(X, a, Y) \mid X, Y \in \{\text{“A”}, \ldots, \text{“Z”}\} \wedge X < Y\}$$

where Characters follow the usual lexicographic order.

SPARQL standard requires that the following query,

```
select * where {
  A a* ?x
}
```

computes only $\{\text{“B”}, \ldots, \text{“Z”}\}$, as duplicates using Kleene star should be avoided.

However, the query will raise a memory limit error on an instance with 2GB available to Virtuoso.

To understand what happens, we ran,

```
select count(*) where {
  C a* ?x
}
```

Which returned $8\,388\,608 = 2^{23} = 2^{\text{“Z”} - \text{“C”}}$.