Evaluating SPARQL 1.1 Property Path Support *

Daniel Janke[1], Adrian Skubella[1], and Steffen Staab[1,2]

[1] Institute for Web Science and Technologies
Universität Koblenz-Landau, Germany
{skubella,danijank,staab}@uni-koblenz.de
http://west.uni-koblenz.de/
[2] Web and Internet Science Group
University of Southampton, UK
s.r.staab@soton.ac.uk
http://wais.ecs.soton.ac.uk/

Abstract. With the release of SPARQL 1.1 in 2013 property paths were introduced, which make it possible to describe queries that do not explicitly define the length of the path that is traversed within an RDF graph. Already existing RDF stores were adapted to support property paths. In order to give an insight on how well the current implementations of property paths in RDF stores work, we introduce a benchmark for evaluating the support of property paths. In order to support realistic RDF graphs as well as arbitrarily scalable synthetic RDF graphs as benchmark dataset, a query generator was developed that creates queries from query templates. Furthermore, we present the results of our benchmark for 4 RDF stores frequently used in academia and industry. These results indicate that many current implementations of property paths have several shortcomings.

1 Introduction

The SPARQL Protocol And RDF Query Language (SPARQL) is used to query data from RDF stores. In 2008 SPARQL 1.0 became an official World Wide Web Consortium recommendation [2], and has been used in academia and industry since then.

One limitation of SPARQL 1.0 was the missing possibility to formulate queries, which do not specify the length of traversed paths during their execution. Therefore, queries may become very complex, or it may even be impossible to retrieve complete results with one single query.

For example it is impossible to create a SPARQL 1.0 query that retrieves all friends of friends of a friend etc. from a social network. In order to make such complex queries more concise, or even possible, property paths were introduced with SPARQL 1.1 in 2013. The construct `<foaf:knows>*` could be used in a SPARQL 1.1 query to retrieve all direct or indirect friends of a friend.

After property paths had been introduced, already existing RDF stores have been adapted to support this new feature. To give insights into the current state of the integration of property paths and to detect possible shortcomings, the current implementations of property paths needs to be evaluated. RDF store developers could use such an evaluation of their own property path implementation to analyse the current state of their implementation and as help for the further development of their RDF store. In order to provide such an evaluation we offer two main contributions in this work:

1. In section 3 we introduce a benchmark, which makes it possible to evaluate the

* This paper is based on the findings in the bachelor thesis [1]

property path support of RDF stores based on the completeness and soundness of results and the execution time of queries. This benchmark is not restricted to a specific dataset by offering query templates and a query generator that instantiates these templates for arbitrary datasets.

2. In section 4 we present benchmark results for Apache Jena, Virtuoso, Allegrograph and RDF4J. These results indicate that the current property path implementation of some stores have several shortcomings. Queries containing property paths partly have an execution time of over one hour and for some queries RDF stores return incomplete or even incorrect results.

2 Preliminaries

In this section common definitions for RDF, SPARQL and property paths are given in order to define the terminology used in this work.

2.1 Resource Description Framework

The Resource Description Framework [3] is a general-purpose language for representing information in the web. It uses triples to represent the information as directed, labelled graphs. A graphical representation of a fictitious social network with the name social-nw is shown in figure 1. In this figure nodes depict people, for instance `snw:Ann`, which are connected by edges labelled with `foaf:knows` or `snw:isMarried`. For better legibility prefixes can be used to abbreviate IRIs. An example for such a prefix is given by `PREFIX snw: http://www.social-nw.com/`. This prefix defines that for instance `snw:Ann` actually means `http://www.social-nw.com/Ann`.

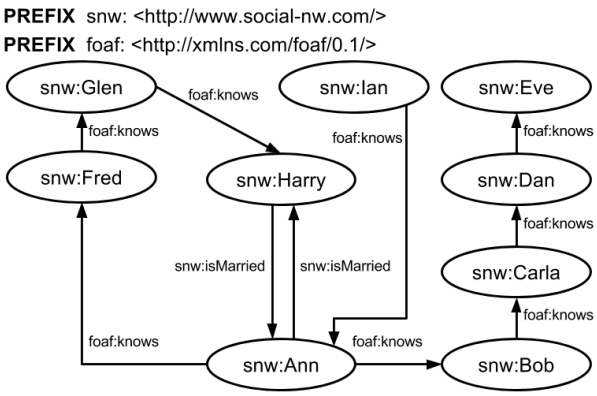

Fig. 1: An RDF graph depicting a fictitious social network

Definition 1. *RDF triple*
The triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ *is called RDF triple where I and L are two disjoint sets of all IRIs and literals, respectively and B is the set of all blank nodes. Furthermore, s is called the subject, p the predicate and o the object of the triple. [4]*

An example for such an RDF triple within the social network show in figure 1 is the IRI `snw:Ann` which denotes the subject, followed by `foaf:knows`, which is the predicate and finally `snw:Bob` which denotes the object of the triple.

Definition 2. *RDF graph*
An RDF graph G is a finite set of RDF triples. [5]

The graph shown in figure 1 is an RDF graph. In graphs paths exist between vertices.

Definition 3. *Path*
A path $P = (< v_1, e_1, v_2 >, < v_2, e_2, v_3 >, ..., < v_n, e_n, v_{n+1} >)$ in an RDF graph G is a sequence of triples such that $(v_i, e_i, v_{i+1}) \in G$. Furthermore the length of the path is defined by the number of triples in the path. [5]

A path with the length 2 from `snw:Ann` to `snw:Carla` in the graph in figure 1 can be depicted as (<snw:Ann, foaf:knows, snw:Bob>, <snw:Bob, foaf:knows, snw:Carla>)

2.2 SPARQL

SPARQL can be used to query data from a stored RDF graph. In definition 4 basic graph patterns are defined, which are frequently used in queries.

Definition 4. *Basic graph pattern (BGP)*
A tuple $t \in (I \cup L \cup B \cup V) \times (I \cup V) \times (I \cup L \cup B \cup V)$ is a basic graph pattern, where I, L, B and V are disjoint sets of IRIs, literals, blank nodes and variables respectively. If P_1 and P_2 are basic graph patterns, then (P_1 . P_2) is a graph pattern.[5]

BGPs can be used with SELECT queries of SPARQL to retrieve data from RDF graphs.

Definition 5. *SELECT query*
If P is a graph pattern and V' $\subset$ V is a set of variables, then (`SELECT` V' `WHERE` {P}) is a SELECT query.[4]

The semantics of graph patterns are defined in terms of mappings in definition 6.

Definition 6. *Variable mappings*
Mappings are partial functions from variables V to an RDF term T, which is defined as $I \cup L$.[6]

The domain $dom(\mu)$ of a mapping μ is the set of variables on which μ is defined. Two mappings can be compatible as defined in definition 7.

Definition 7. *Compatible mappings*
Two mappings μ_1 and μ_2 are compatible (written as $\mu_1 \sim \mu_2$) if $\mu_1(x) = \mu_2(x)$ for all variables x that are in both $dom(\mu_1)$ and $dom(\mu_2)$.[6]

If $\mu_1 \sim \mu_2$, then $\mu_1 \cup \mu_2$ denotes the mapping obtained by extending μ_1 according to μ_2 on all variables in $dom(\mu_1)$ and $dom(\mu_2)$. The join of two sets of mappings is defined in definition 8.

Definition 8. *Join of mappings*
Given two sets of mappings M1 and M2, the join of M1 and M2 is defined as:
$M_1 \bowtie M_2 \implies \{\mu_1 \cup \mu_2 | \mu_1 \in M_1, \mu_2 \in M_2, \wedge \mu_1 \sim \mu_2\}$ *[6]*

For a triple pattern P and a mapping μ, $\mu(P)$ is written for the triple obtained from P by replacing each variable $x \in dom(\mu)$ by $\mu(x)$. In the following definition the evaluation $[[P]]_G$ of a graph pattern P over a graph G is defined. The set of all variables appearing in a pattern P is denoted by var(P).

Definition 9 *Evaluation of graph pattern*
if $P \in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$*, then* $[[P]]_G := \{\mu : var(P) \rightarrow T | \mu(P) \in G\}$*, if* P *is* $P_1.P_2$*, then* $[[P]]_G := [[P_1]]_G \bowtie [[P_2]]_G$ *[6]*

Finally the evaluation of `SELECT` queries is defined in definition 10.

Definition 10 *Semantics of SELECT query*
The evaluation $[[Q]]_G$ *of a query* Q *of the form* `SELECT X WHERE P` *is the set of all projections* $\mu|_X$ *of mappings* μ *from* $[[P]]_G$ *to* X*, where the projection of* $\mu|_X$ *is the mapping that coincides with* μ *on* X *and is undefined elsewhere.[6]*

Property Paths were introduced as part of SPARQL 1.1 in 2013. Property Paths make it possible to define queries that match with an arbitrary amount of edges.

Definition 11. *Property Paths*
sPo *is a property path where* $s \in V \cup I$*,* $o \in I \cup L \cup B \cup V$ *and* P *is a property path expression. Furthermore* sPo *is a basic graph pattern.*

There are several different property path expressions, which denote different paths. These different expressions and their syntax are presented in definition 12.

Definition 12. *Property Path expression*
1) $p \in I$ *is a property path expression.*
2) $\hat{}P$ *with property path expression* P*, is the inverse property path expression.*
3) P_1/P_2*, with property path expressions* P_1 *and* P_2*, is the sequence property path expression.*
4) $P_1|P_2$*, with property path expressions* P_1 *and* P_2*, is the alternative property path expression.*
5) P*, with property path expression* P*, is the transitive reflexive closure property path expression.*
6) P+*, with property path expression* P*, is the transitive closure property path expression.*

A query containing the property path expression * is shown in listing 1.

```
PREFIX snw:  http://www.social-nw.com/
PREFIX foaf: http://xmlns.com/foaf/0.1/
SELECT ?friend WHERE {
        <snw:Ann> <foaf:knows>* ?friend.}
```

Listing 1: Query containing the * operator of property paths

The evaluation of a property path is presented in the following definition:

Definition 13. *Evaluation of property paths*
For constants $s \in I, o \in I \cup L \cup B$ *and variables* $v, v_1, v_2 \in V$ *the evaluation of property paths is defined as:*

$$[[sPo]]_G := \begin{cases} \{\mu = \varnothing\} | if(s,o) \in [[P]]_G \\ \varnothing, otherwise \end{cases}$$

$$[[vPo]]_G := \{\mu | (\mu(v), o) \in [[P]]_G \wedge dom(\mu) = \{v\}\}$$
$$[[sPv]]_G := \{\mu | (s, \mu(v)) \in [[P]]_G \wedge dom(\mu) = \{v\}\}$$
$$[[v_1 P v_2]]_G := \{\mu | (\mu(v_1), \mu(v_2)) \in [[P]]_G \wedge dom(\mu) = \{v_1, v_2\}\}$$

The semantics of the evaluation of property path expressions are defined in definition 14.

Definition 14. *Evaluation of property path expressions*
The evaluation $[[P]]_G$ of a property path expression P over an RDF graph G is a set of pairs of RDF terms from $I \cup L \cup B \cup V$ defined as follows:
$[[p]]_G := \{(s,o) \mid (s,p,o) \in G\}$,
$[[\hat{}P]]_G := \{(s,o) \mid (o,s) \in [[P]]_G\}$,
$[[P_1/P_2]]_G := [[P_1]]_G \circ [[P_2]]_G$,
$[[P_1|P_2]]_G := [[P_1]]_G \cup [[P_2]]_G$,
$[[P+]]_G := \bigcup_{i \geq 1} [[P^i]]_G$,
$[[P*]]_G := [[P+]]_G \cup \{(s,s) \mid (s,p,o) \in G\}$,
where $\circ$ is the usual composition of binary relations, and p^i is the concatenation $p/.../p$ of i copies of p. [6]

The query in listing 1 contains the * operator and returns nodes that are connected to `snw:Ann` by a direct or indirect `foaf:knows` relation. Executing this query on the RDF graph illustrated in figure 1 would return `snw:Ann`, `snw:Bob`, `snw:Carla`, `snw:Dan`, `snw:Eve`, `snw:Fred`, `snw:Glen` and `snw:Harry`.

3 Property Path Benchmark Design

When benchmarking the performance of RDF stores, different datasets might lead to different results. As stated in [7] realistic datasets are less ordered and have a less consistent structure than generated datasets. Thus, benchmarks using synthetic datasets might lead to different results than benchmarks using realistic datasets. Nevertheless, dataset generators can produce arbitrary sized synthetic datasets to benchmark the scalability of RDF stores. To overcome the limitations of selecting a single dataset we have designed our novel property path benchmark[3] that can use every RDF dataset.

To make it possible to use an arbitrary dataset as benchmark dataset, we provide query templates instead of fixed queries. These query templates are designed to test property path expressions. With the help of a query generator the query templates can be instantiated for the user-selected dataset (see section 3.1).

When evaluating the implementations of property paths, we see the testing of soundness and completeness of the returned query results as well as the querying time as the most important measures (see section 3.3 for their definitions). In order to test whether the returned results are complete and sound, we need a reference result set for each query. The creation of these reference result sets is described in section 3.2.

Since query result sets might contain different identifier for the same blank node, the equality of two result sets containing blank nodes might become hard to check [8]. Therefore, we preprocess the dataset by replacing each blank node by a new unique IRI that did not exist in the dataset before. This procedure as well as the rest of the execution strategy of our benchmark is described in section 3.4.

3.1 Query Generator

With our benchmark we want to evaluate how well the individual property path primitives are supported. Therefore, the benchmark comprises 8 query templates (see appendix A). Half of them test a single property path primitive. The only difference between the + and * operator is that in case of a query `SELECT ?o WHERE {S P* ?o}` where `S` and `P` are IRIs, `S` is included in the result set, whereas it is not included in

[3] Available as open source under https://github.com/Institute-Web-Science-and-Technologies/propertyPathBenchmark

the case of `SELECT ?o WHERE {S P+ ?o}` . Thus we only test the * operator. The nesting of property path primitives may cause a higher degree on computational complexity. Therefore, the second half of the query templates test different nestings comprising templates with a high complexity containing nested * or + operators and templates with a low complexity containing, e.g., only alternative and sequence operators.

In order to generate queries that can be executed on the user-selected dataset each query template contains query variables and template variables. The former remain variables in the generated queries whereas the latter will be replaced by IRIs existing in the dataset. An example for a query template is shown in listing 2. In this listing the template variables S1 and P1 denote subject and predicate that are substituted with constant IRIs during the query generation process. The query variable `?o1` remains a variable in the actual generated query later on.

```
SELECT ?o1 WHERE{
        S1 P1* ?o1 . }
```

Listing 2: Query template for the * operator

Basically the start, the end and all edge labels of a property path can be query variables or template variables. As shown in [9], in most realistic SPARQL 1.0 queries, predicates consist of constant IRIs as well as at least one variable and one constant occurring at subject and object positions. We assume that property paths used in a realistic setting will have similar properties. Therefore, all of our query templates have template variables as predicates as well as one query variable and one template variable at the start or end of the property path.

RDF stores that are designed to process only SPARQL 1.0 queries regularly use SPO, OSP and POS indices as described in [10]. With the help of these indices and a cache of previous lookups, queries that traverse short paths during execution can be processed efficiently. Property paths containing * and + operators may easily match with paths longer than the paths usually matched with SPARQL 1.0 queries. Therefore, an efficient indexing strategy for property paths are required to achieve a high query performance. In order to stress the indices of the evaluated RDF stores, the query generator has to ensure that each generated query containing a * or a + operator matches with at least one path of length n in the dataset. This n can be chosen freely. Furthermore, the generated queries should return non-empty result sets to check whether the evaluated RDF stores are able to find all query results.

In order to ensure that the generated queries return non-empty results set, our query generator searches for a substitution of the template variables such that the resulting property path matches with at least one path in the dataset. In case of property paths containing * and + operators, this path has to have a minimal length of n. To find such a substitution our query generator first generates so called grounding queries only consisting of SPARQL 1.0 features. These grounding queries are sent to an RDF store, which has loaded the dataset, to find actual IRIs as substitutions for the template variables. An example for such a grounding query is shown in listing 3. It checks if there exists a path of at least 4 edges labelled with `?p1`. If this is the case `?s1` and thus the starting point of the query and `?p1`, the property is returned. The substitutions found by the query generator are used to replace the template variables with concrete IRIs found in the dataset resulting in the final queries that can be executed to actually perform the benchmark.

```
SELECT ?s1 ?p1 WHERE{
        ?s1 ?p1 ?o1.
        ?o1 ?p1 ?o2.
        ?o2 ?p1 ?o3.
        ?o3 ?p1 ?o4.
        }
```

Listing 3: Grounding query for query template in listing 2 ensuring a path length of at least n = 4

Assuming the dataset shown in figure 1 and the query template in listing 2 the query generator executes the grounding query shown in listing 3 in order to fill the template variables in the respective template. Thus, `?s1` would be bound to `snw:Ann` and `?p1` to `foaf:knows` because there only exists a path from `snw:Ann` to `snw:Eve` that has a length of 4 and has the same IRI `foaf:knows` on each edge. The resulting query, which then can be executed on the dataset, is shown in listing 4.

```
PREFIX snw:  http://www.social-nw.com/
PREFIX foaf: http://xmlns.com/foaf/0.1/
SELECT ?o1 WHERE{
        <snw:Ann> <foaf:knows>* ?o1.}
```

Listing 4: Benchmark query generated by query generator

3.2 Result Set Generator

Since we want to test for completeness and soundness of results, we need the correct and complete reference result sets for each generated query. These reference result sets are obtained with the help of a result set generator. This generator simulates queries that contain the inverse, sequence and alternative operator by executing one or more SPARQL1.0 queries that should return the same results as the property path queries. Furthermore, the generator can simulate the * and + property path expression by executing a breadth-first search. Thereby the data is retrieved with several SPARQL 1.0 queries[4]. We restrict these queries to SPARQL 1.0 since we assume that these features are implemented correctly for most RDF stores.

These queries are then send to all benchmarked RDF stores and the result sets are computed. For each query we compare the result sets of the different RDF stores. If the majority of the stores have the same result set we take this set as the correct result. If no majority can be found, we state that the result set is unknown. Thus, the completeness and soundness cannot be evaluated for this query.

Sometimes more than one query is needed to create the reference result set. In case of the prior created benchmark query in listing 4 the breadth-first search based result set generation works as follows: First a reference result set R is initialized with $R = \{(?o1, < snw : Ann >)\}$ because the * operator also returns the starting point of the query. Then the query shown in listing 5 is executed that returns all vertices connected to `snw:Ann` by an edge labelled with `foaf:knows` and the results are added to R. After that the query is extended to reach all vertices that are connected to `snw:Ann`

[4] The queries that are used to create the reference result set for each query template are listed in appendix A

by a path containing two occurrences of `foaf:knows`. The query is shown in listing 6 and the results retrieved from executing this query are added to R. This is done until a query does not return any new results. Then R is enclosed and is defined as the final reference result set for the query.

```
PREFIX snw:  http://www.social-nw.com/
PREFIX foaf: http://xmlns.com/foaf/0.1/
SELECT ?o1 WHERE{
        <snw:Ann> <foaf:knows> ?o1. }
```

Listing 5: Query that retrieves vertices connected to snw:Ann by one occurrence of foaf:knows

```
PREFIX snw:  http://www.social-nw.com/
PREFIX foaf: http://xmlns.com/foaf/0.1/
SELECT ?o1 WHERE{
        <snw:Ann> <foaf:knows> ?r1.
        ?r1       <foaf:knows> ?o1. }
```

Listing 6: Query that retrieves vertices connected to snw:Ann by two occurrences of foaf:knows

3.3 Metrics

The metrics of the new property path benchmark were chosen to focus on the completeness and soundness of results to ensure that the returned results are feasible and thus, the RDF store can be used with queries containing property paths.

Definition 15. *Query soundness*
The percentage of right query results that are returned by each query. If R_q is the set of correct results for a query q and R_q^s is the set of returned results of an executed query q where s is the name of the respective RDF store, then the query soundness is defined as
$s(q) = \frac{|R_q \cap R_q^s|}{|R_q|}$

Definition 16. *Query completeness*
The percentage of all possible query results of the query. If R_q is the set of correct results for a query q and R_q^s is the set of returned results of an executed query q where s is the name of the respective RDF store then the query completeness
$c(q) = \frac{|R_q \cap R_q^s|}{|R_q^s|}$

On the other hand the execution time of queries is measured to compare the efficiency of query execution between the benchmarked stores.

Definition 17. *Average execution time per query*
The arithmetic mean $avexec(q)$ of the execution time $t(q)$ of each query q. The average execution time per query is:
$avexec(q) = \frac{\sum_{i=1}^{n} t(q_i)}{n}$ *where i is the ith execution of the query.*

3.4 Execution Strategy

Preparation phase: As first step of the benchmark execution, each blank node in the used dataset is replaced with an unique IRI. After that the dataset is loaded into the tested system. Then the query generator generates the queries from the query templates. After that the result queries are executed and thus, reference result sets are generated.
Evaluation phase: In order to measure the query execution time of the benchmark on a warm store and thus, to evaluate how the tested system performs in an authentic setting, the query mix is executed twice before the actual metrics are measured. The queries of the query mix are executed one after another and not in parallel. The query mix is executed 10 times in order to have ample results. Queries that are not finished after one hour are aborted and counted as failed queries. From the results the highest and lowest execution times are deleted to prevent the effect of outliers. Query 1 to query 8 are all executed once, before query 1 is executed again to prevent caching of results. Finally the resulting metrics are returned in a human readable log file.

4 Evaluation

With the help of the property path benchmark defined in the previous section we benchmarked RDF stores frequently used in academia and industry.

4.1 Experimental Setup

Benchmarked stores: The benchmarked stores are Virtuoso 7.2.2 community edition[5], RDF4J 2.0M1 formerly known as Sesame[6], Apache Jena 3.0.1[7] and Allegrograph 6.0.2 free edition[8].
Dataset: In [7] Duan et al. state that synthetically created datasets have a shortcoming due to their structuredness in comparison to real world datasets. Synthetically created datasets are often created by a data generator and do not reflect the characteristics of real world data. Duan argues that synthetically created datasets have a rather ordered and consistent structure, whereas real world datasets are often far less structured. Therefore we have decided to use the Billion Triple Challenge (BTC) 2014 dataset [11], which holds real world data, for our evaluation.

The BTC 2014 dataset was crawled in 14 iterations called hops. In the first hop all triples that contain a seed IRI as subject are crawled and added to the dataset. The objects of these triples were used as the seed IRIs for the second hop. In these 14 hops a dataset of about 4 billion quads was crawled [12].

Since the complete BTC 2014 dataset would consume a lot of time to be loaded without giving additional insight into the property path support of the RDF stores, we used only a subset of the complete dataset for our benchmark. This subset consists of the first three hops of the BTC dataset. This subset was preprocessed by iterating over all quads of the original dataset and deleting all syntactically incorrect quads. The remaining dataset consists of about 400 million quads.

The free edition of Allegrograph has a limit of 5 million triples [13] and therefore the BTC 2014 could not be used to benchmark it. In order to still get an insight into

[5] http://virtuoso.openlinksw.com/ retrieved at 3.7.2017
[6] http://RDF4J.org/ retrieved at 3.7.17
[7] https://jena.apache.org/ retrieved at 3.7.2017
[8] http://franz.com/agraph/allegrograph/ retrieved at 3.7.17

Allegrograph a Polish DBpedia dump[9] that consists of around 1.3 million triples was used to benchmark the store.
Evaluation Environment: The RDF stores were benchmarked on a virtual machine with 8 GB RAM, 500 GB disk space, 4 2.9 Mhz processor cores and Ubuntu 14.04 running on it. The Java version on the machine has been 1.8.0.77.

4.2 Evaluation of Results

While benchmarking Virtuoso, RDF4J and Allegrograph no errors or exceptions have occurred. During the benchmark process of Jena an `OutOfMemoryError` has been thrown whenever a query with the * operator was used. In order to identify the cause of the error, the amount of results the query should return has been limited to 100. The results that have been returned by a query of the form `SELECT ?o WHERE {A B* ?o.} LIMIT 100` where `A` and `B` are valid IRIs, consisted of 100 times `A`. Due to this fact it is presumable that the query containing the * operator returns `A` recursively until the main memory was full. To ensure that this behaviour is not caused by cycles in the dataset a query of the same form but with a predicate IRI that did not exist in the dataset was executed. This query still returned 100 times `A`. This indicates, that the * operator is not implemented correctly.

Due to the problems with the * operator the queries 4, 7 and 8 could not be processed. Additionally query 3, 5, and 6 returned no results after 1 hour and thus, were aborted. Query 1 returned an empty and thus, incomplete result set. Only for query 2 a valid result was returned. Due to the lack of comparable results, Jena has been omitted in the comparison of triple stores.

Completeness and Soundness

As presented in table 1 the result sets of Virtuoso and RDF4J on the BTC dataset are different for almost all queries. For instance query 1 and 6 deliver completely different result sets, because for query 6 RDF4J returns 0 results and for query 1 Virtuoso returns 0 results. Due to the fact that 0 results were returned, nothing can be said about the soundness of results as it is depicted by - in the table. Only for query 5 Virtuoso and RDF4J returned the same result set.

Query	**Virtuoso**			**RDF4J**			**Reference**
	$R_q^{Virtuoso}$	$s(q)$	$c(q)$	R_q^{RDF4J}	$s(q)$	$c(q)$	R_q
1 = ^P1	0	-	0	4111	1	1	4111
2 = P1/P2	2	0.5	0.5	2	1	1	2
3 = P1| P2	4	1	0.66	6	1	1	6
4 = P1*	6	1	1	1	1	0.166	6
5 = (P1| P2)/P3	3	1	1	3	1	1	3
6 = P1+/P2	6	1	1	0	-	0	6
7 = P1*| P2*| P3*	8	1	0.57	14	1	1	14
8 = P1+/P2*	5	1	0.71	7	1	1	7

Table 1: The soundness and completeness for Virtuoso and RDF4J

[9] http://wiki.dbpedia.org/Downloads2015-04 retrieved at 29.06.17

The soundness and completeness of the results of RDF4J is 100% for queries 1, 2, 3, 5, 7 and 8. For query 6 RDF4J returned an empty result set and for query 4 it only returns 1 out of 6 correct results. Thus, the soundness for the results is 100% for query 4 but the completeness is only 16.6%. On the other hand Virtuoso found the correct and complete result sets for queries 4, 5 and 6. For queries 7 and 8 Virtuoso returns correct but incomplete results. In case of query 1 Virtuoso returns none of the 4111 correct results. For query 2 Virtuoso misses 1 result and even returns a wrong result. Thus, it can be concluded that in our evaluation Virtuoso has problems with returning all results and in case of query 2 even correct results at all, whereas RDF4J seems to have problems only with the execution of * and + operators.

Due to this outcome we benchmarked only Allegrograph and RDF4J with the Polish DBpedia dump. In table 2 the amount of results in the reference dataset and the soundness and completeness of each store are displayed.

Query	**Allegrograph**			**RDF4J**			**Reference**
	$R_q^{Allegro}$	$s(q)$	$c(q)$	R_q^{RDF4J}	$s(q)$	$c(q)$	R_q
1 = ^P1	2	1	1	2	1	1	2
2 = P1/P2	1	1	1	1	1	1	1
3 = P1| P2	2	1	1	2	1	1	2
4 = P1*	5	1	1	5	1	1	5
5 = (P1| P2)/P3	2	1	1	2	1	1	2
6 = P1+/P2	1	1	1	1	1	1	1
7 = P1*| P2*| P3*	6	1	1	6	1	1	6
8 = P1+/P2*	4	1	1	4	1	1	4

Table 2: The soundness and completeness for Allegrograph and RDF4J

The table shows that the result sets of Allegrograph and RDF4J are the same for each store and equal to the reference dataset. Thus, it can be concluded that the soundness and completeness of results is 100% for both stores.

Since RDF4J returned the correct result sets for the + and also for the * operators on the smaller Polish DBpedia dump but not on the larger BTC 2014 subset, RDF4J might ignore some intermediate results if the number of the intermediate results becomes to huge.

Execution Times

The execution times of Virtuoso and RDF4J can be compared only partly. Due to the fact that Virtuoso fails to return complete and sound results for most queries and RDF4J also fails to do so for some queries, the mere comparison of execution times of queries is not very meaningful. Only for query 5, where the result sets are complete and sound for Virtuoso and RDF4J, it can be said that RDF4J executed the query with 79ms faster than Virtuoso with 1122ms. Nothing can be said about the execution times for any of the other queries.

As shown in figure 2 the execution time for Allegrograph is higher for each query but query 5. Due to the fact that both stores return sound and complete result sets, it can be concluded that RDF4J evaluates queries with property path expressions more efficiently than Allegrograph.

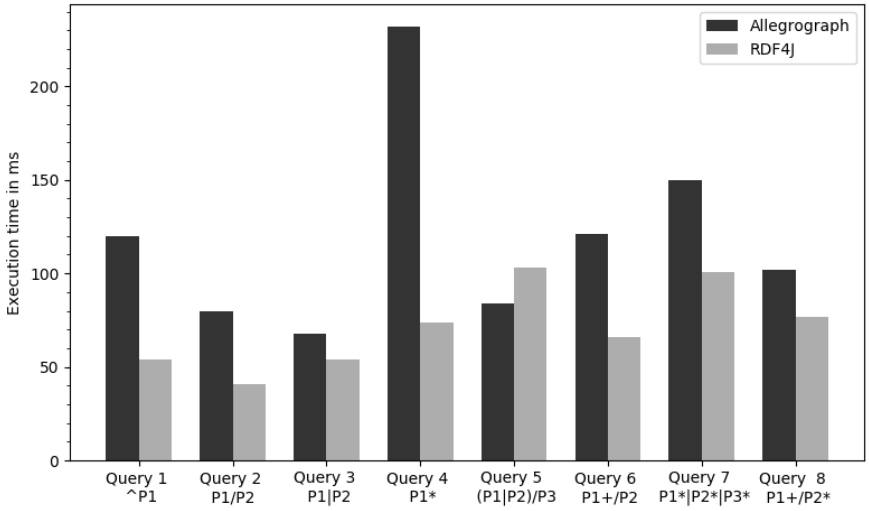

Fig. 2: The average execution time for each query for Allegrograph and RDF4J

4.3 Discussion

In our evaluation we evaluated the RDF stores Apache Jena, Virtuoso, RDF4J and Allegrograph, that are used frequently in academia and industry. Before performing the benchmark we expected that the RDF stores would process property path queries correctly. When performing our benchmark this expectation was not fulfilled. Jena could not return results for any query in under 1 hour besides query 2. Furthermore, the * operator could not be evaluated at all and the inverse operator returned empty result sets.

Virtuoso returned mostly incomplete results and in one case even a wrong result. RDF4J could handle most property path expressions but had problems with some queries containing the * and + operator when executed on the larger BTC 2014 subset. On the smaller Polish DBpedia dump RDF4J and Allegrograph could find the complete and correct result sets for all queries. All in all it can be said that our evaluation indicates that some RDF stores have shortcomings in the context of property paths.

5 Related Work

When examining the already existing benchmarks for RDF stores, we found no benchmark that focused on evaluating the support of property paths. Nevertheless, we checked, whether the used queries could serve as a basis for property path queries.

The Berlin SPARQL Benchmark [14] uses an e-commerce use case in which the dataset describes different vendors, their products and users' reviews. The queries emulate the search and navigation pattern of consumers. Since the execution of queries traverse at most 2 edges, this benchmark is not usable to test property paths.

In [15] the Lehigh University Benchmark is presented, which was designed to evaluate the performance of RDF stores over a dataset generated based on an ontology. The queries were chosen in respect to different characteristics like their length, complexity or selectivity. The longest path that is traversed by a query has a length of 3 and thus, the benchmark queries are not suitable for testing property paths.

The DBpedia SPARQL Benchmark [9] uses the DBpedia knowledge base as dataset,

which can be scaled to an arbitrary size. The queries were created by mining the official DBpedia SPARQL endpoint for a three month period and then graph clustering algorithms were used to identify frequently posted queries. These query patterns are used to generate queries for the scaled dataset. Due to the fact that the generated queries match with at most 2 edges they cannot serve as basis for queries evaluating the support of property paths.

The Semantic Publishing Benchmark [16] is a benchmark developed by the Linked Data Benchmark Council, which was inspired by the media and publishing industry. It was designed along a scenario in which news, articles and media assets are updated regularly. The benchmark offers a data generator that produces scalable synthetic data. The queries were chosen in order to evaluate how well an RDF store handles several technical challenges like the parallel execution of unions, optional and nested optional clauses or how well regular expressions are evaluated. Since the longest path that is traversed when the queries are executed has a length of 2, the queries are not usable to test property paths.

To the best of our knowledge, no benchmark that evaluates the property path support of RDF stores in particular exists. Due to paths with a length of at most 3 traversed edges during query execution, none of the benchmarks provides queries that can be used as a basis for realizing queries testing the * or + operator support.

6 Conclusion

Property Paths that make it possible to define queries that match with an arbitrary amount of edges were introduced with SPARQL 1.1 in 2013. Already existing RDF stores were extended to make it possible to process queries containing such property paths. In order to evaluate how well these RDF stores handle property paths we introduced a novel benchmark. With our benchmark the RDF stores can be tested with several datasets. For each dataset queries are generated that test the implementation of the property path expressions. Beside the query execution time the benchmark also examines the completeness and soundness of the returned query results.

When benchmarking several RDF stores frequently used in academia and industry, especially the check of sound and complete query results lead to surprising results. Several tested RDF stores could not return complete result sets and in one case even incorrect results were returned. Furthermore, in our evaluation another RDF store needed over one hour to execute queries containing property paths. This long execution time indicates that queries with property paths are executed inefficiently.

In summary our evaluation indicates that the current implementations of property paths in frequently used RDF stores is not ready for practical usage. Thus, our novel property paths benchmark has shown to be a valuable tool to systematically evaluate the property path support of existing and future RDF stores.

References

1. A. Skubella, “Benchmarks for sparql property paths,” bachelor thesis, Institute for Web Science and Technologies, Universität Koblenz-Landau available under https://west.uni-koblenz.de/sites/default/files/studying/theses-files/bachelorarbeit-adrian-skubella-benchmarks-for-sparql-property-paths.pdf, 2016.
2. https://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation/ retrieved at 5.07.17.

3. M. L. Richard Cyganiak, David Wood, "Rdf 1.1 concepts and abstract syntax," tech. rep., W3C Recommendation, 2014.
4. M. Arenas and J. Perez, "Federation and navigation in sparql 1.1," in *Reasoning Web. Semantic Technologies for Advanced Query Answering* (T. Eiter and T. Krennwallner, eds.), vol. 7487 of *Lecture Notes in Computer Science*, pp. 78–111, Springer Berlin Heidelberg, 2012.
5. B. DuCharme, *Learning SPARQL, Chapter 2 pp 19-44, Chapter 3 pp 45-100*. O'Reilly Media, Inc., 2011.
6. E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč, *The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, ch. SPARQL with Property Paths, pp. 3–18. Cham: Springer International Publishing, 2015.
7. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, "Apples and oranges: A comparison of rdf benchmarks and real rdf datasets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 145–156, ACM, 2011.
8. L. Chen, H. Zhang, Y. Chen, and W. Guo, "Blank nodes in rdf," *Journal of Software*, vol. 7, pp. 1993–1999, September 2012.
9. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "Dbpedia sparql benchmark: Performance assessment with real queries on real data," in *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, (Berlin, Heidelberg), pp. 454–469, Springer-Verlag, 2011.
10. D. Wood, P. Gearon, and T. Adams, "Kowari: A platform for semantic web storage and analysis," in *In XTech 2005 Conference*, pp. 05–0402, 2005.
11. T. Käfer and A. Harth, "Billion Triples Challenge data set." Downloaded from http://km.aifb.kit.edu/projects/btc-2014/, 2014.
12. T. Käfer, J. Umbrich, A. Hogan, and A. Polleres, "Towards a dynamic linked data observatory," in *In LDOW at WWW*, 2012.
13. http://franz.com/agraph/allegrograph/ag_commercial_edition.lhtml retrieved at 16.06.17.
14. C. Bizer and A. Schultz, "The berlin sparql benchmark.," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 1–24, 2009.
15. Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semant.*, vol. 3, pp. 158–182, Oct. 2005.
16. V. Kotsev, A. Kiryakov, I. Fundulaki, and V. Alexiev, "Ldbc semantic publishing benchmark (spb) - v2.0 first public draft release," tech. rep., LDBC, 2014.

A Query Templates

In the following the query templates and the respective SPARQL 1.0 queries for the generation of the reference result set are presented. For query templates 1, 2, 3, 5 and 6 the reference result sets are obtained by simply executing the corresponding SPARQL 1.0 queries. Query 4 and 7 have an initial result set $R = \{(?o1, S1)\}$. These initial result sets are extended with results returned from SPARQL 1.0 queries that translate P1* into 1, 2, 3... matches of P1. This process stops if no new results can be added to the reference result set.

Query template 8 contains the sequence of P1+ and P2*. In this case we start with an empty result set and then systematically execute queries that first match P1 one or several times followed by zero, one or more matches of P2. We indicate this by showing intermediate reference result set creating queries, in which only P2* is translated. Each of these queries still contains P1+ that is then translated similar to the reference result set creating queries for query template 6.

Query Templates	Reference Result Set Creating Queries
SELECT ?o1 WHERE{ *?o1 ^P1 S1.*}	SELECT ?o1 WHERE{ *S1 P1 ?o1.*}
Query 1	Result Set Query 1
SELECT ?o1 WHERE{ *S1 P1/P2 ?o1.*}	SELECT ?o1 WHERE{ *S1 P1 ?r1.* *?r1 P2 ?o1.*}
Query 2	Result Set Query 2
SELECT ?o1 WHERE{ *S1 P1|P2 ?o1.*}	SELECT ?o1 WHERE{ *S1 P1 ?o1.*} SELECT ?o1 WHERE{ *S1 P2 ?o1.*}
Query 3	Result Set Queries 3
SELECT ?o1 WHERE{ *S1 P1* ?o1.*}	SELECT ?o1 WHERE{ *S1 P1 ?o1.*} SELECT ?o1 WHERE{ *S1 P1 ?r1.* *?r1 P1 ?o1.*} ... Result Set Queries 4
Query 4	
SELECT ?o1 WHERE{ *S1 (P1|P2)/P3 ?o1.*}	SELECT ?o1 WHERE{ *S1 P1 ?r1.* *?r1 P3 ?o1.*} SELECT ?o1 WHERE{ *S1 P2 ?r1.* *?r1 P3 ?o1.*}
Query 5	Result Set Queries 5

```
SELECT ?o1 WHERE{
    S1 P1+/P2 ?o1 . }
```

Query 6

```
SELECT ?o1 WHERE{
    S1 P1 ?r1 .
    ?r1 P2 ?o1 . }
```

```
SELECT ?o1 WHERE{
    S1 P1 ?r1 .
    ?r1 P1 ?r2 .
    ?r2 P2 ?o1 . }
```

...

Result Set Queries 6

```
SELECT ?o1 WHERE{
    S1 P1*|P2*|P3* ?o1 . }
```

Query 7

$\forall P \in \{P1, P2, P3\}$

```
SELECT ?o1 WHERE{
    S1 P ?o1 . }
```

```
SELECT ?o1 WHERE{
    S1 P ?r1 .
    ?r1 P ?o1 . }
```

...

Result Set Queries 7

```
SELECT ?o1 WHERE{
    S1 P1+/P2* ?o1 . }
```

Query 8

```
SELECT ?o1 WHERE{
    S1 P1+ ?o1 . }
```

```
SELECT ?o1 WHERE{
    S1 P1+/P2 ?o1 . }
```

```
SELECT ?(o1) WHERE{
    S1 P1+/P2/P2 ?o1 . }
```

...

Result Set Queries 8