Counting Beyond a Yottabyte, or how SPARQL 1.1 Property Paths will Prevent Adoption of the Standard

Marcelo Arenas
Department of Computer Science
PUC Chile
marenas@ing.puc.cl

Sebastián Conca
Department of Computer Science
PUC Chile
saconca@puc.cl

Jorge Pérez
Department of Computer Science
Universidad de Chile
jperez@dcc.uchile.cl

ABSTRACT

SPARQL –the standard query language for querying RDF– provides only limited navigational functionalities, although these features are of fundamental importance for graph data formats such as RDF. This has led the W3C to include the *property path* feature in the upcoming version of the standard, SPARQL 1.1.

We tested several implementations of SPARQL 1.1 handling property path queries, and we observed that their evaluation methods for this class of queries have a poor performance even in some very simple scenarios. To formally explain this fact, we conduct a theoretical study of the computational complexity of property paths evaluation. Our results imply that the poor performance of the tested implementations is not a problem of these particular systems, but of the specification itself. In fact, we show that any implementation that adheres to the SPARQL 1.1 specification (as of November 2011) is doomed to show the same behavior, the key issue being the need for counting solutions imposed by the current specification. We provide several intractability results, that together with our empirical results, provide strong evidence against the current semantics of SPARQL 1.1 property paths. Finally, we put our results in perspective, and propose a natural alternative semantics with tractable evaluation, that we think may lead to a wide adoption of the language by practitioners, developers and theoreticians.

Categories and Subject Descriptors

H.2.3 [**Languages**]: Query Languages

Keywords

SPARQL 1.1, property paths, bag semantics, counting complexity

1. INTRODUCTION

It has been noted that, although RDF is a graph data format, its standard query language, SPARQL, provides only limited navigational functionalities. This has led the W3C to include the *property-path* feature in the upcoming version of the standard, SPARQL 1.1. Property paths are essentially regular expressions that retrieve pairs of nodes of an RDF graph that are connected by paths conforming to those expressions. In this paper, we study the semantics of property paths and the complexity of evaluating them. We perform this study both from a theoretical and a practical point of view, and provide strong arguments against the current semantics of SPARQL 1.1 property paths.

We began our study by testing several SPARQL 1.1 implementations, and we were faced with an intriguing empirical observation: all these implementations of SPARQL 1.1 fail to give an answer

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2012, April 16–20, 2012, Lyon, France
ACM 978-1-4503-1229-5/12/04.

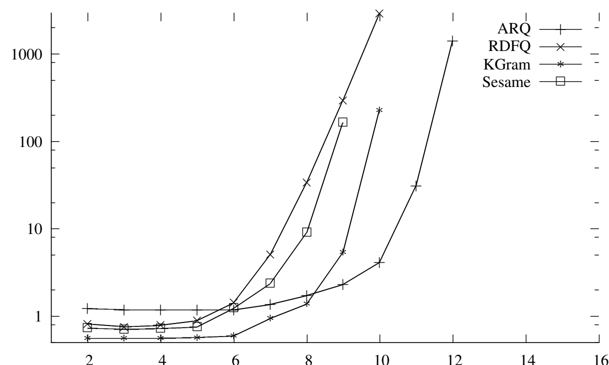

Figure 1: Time in seconds for processing Cliq-1 w.r.t. the clique size n (time axis in log-scale)

```
@prefix : <http://example.org/> .
:a0 :p :a1, :a2, :a3, :a4, :a5, :a6, :a7 .
:a1 :p :a0, :a2, :a3, :a4, :a5, :a6, :a7 .
:a2 :p :a0, :a1, :a3, :a4, :a5, :a6, :a7 .
:a3 :p :a0, :a1, :a2, :a4, :a5, :a6, :a7 .
:a4 :p :a0, :a1, :a2, :a3, :a5, :a6, :a7 .
:a5 :p :a0, :a1, :a2, :a3, :a4, :a6, :a7 .
:a6 :p :a0, :a1, :a2, :a3, :a4, :a5, :a7 .
:a7 :p :a0, :a1, :a2, :a3, :a4, :a5, :a6 .
```

Figure 2: RDF graph representing a clique with 8 nodes

in a reasonable time (one hour) even for small input graphs and very simple property path expressions. We conduct two sets of experiments, the *clique* experiments and the *foaf* experiments, testing four implementations: ARQ [22], RDF::Query [24], KGRAM-Corese [23], and Sesame [25]. For the first experiment, we consider RDF graphs representing cliques (complete graphs) of different sizes. For example, Figure 2 shows a clique with 8 nodes in N3 notation. In this scenario, we tested the performance of the implementations by using a very simple query:

Cliq-1: `SELECT * WHERE { :a0 (:p)* :a1 }`

that essentially asks for paths of arbitrary length between two fixed nodes. The experimental behavior for this query was quite surprising: no implementation was able to handle a clique with 13 nodes. That is, all implementations fail to give an answer after one hour for an input RDF graph with only 156 triples and 970 bytes of size on disk. In particular, Sesame fails for a clique with 10 nodes, KGRAM and RDF::Query for 12 nodes, and ARQ for 13 nodes. Our experiments show that for all implementations, the time needed to process Cliq-1 seems to grow doubly-exponentially w.r.t. the input data file (see the graph in Figure 1, which is in logarithmic scale). We also tested queries with *nested stars*, showing that nesting has an unexpected impact in query evaluation. In particular, we tested the query:

Input	ARQ	RDFQ	Kgram	Sesame
9.2KB	5.13	75.70	313.37	–
10.9KB	8.20	325.83	–	–
11.4KB	65.87	–	–	–
13.2KB	292.43	–	–	–
14.8KB	–	–	–	–
17.2KB	–	–	–	–
20.5KB	–	–	–	–
25.8KB	–	–	–	–

Table 1: Time in seconds for processing query Foaf-1

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
9.2KB	2.24	47.31	2.37	–	0.29	1.39
10.9KB	2.60	204.95	6.43	–	0.30	1.32
11.4KB	6.88	3222.47	80.73	–	0.30	1.34
13.2KB	24.42	–	394.61	–	0.31	1.38
14.8KB	–	–	–	–	0.33	1.38
17.2KB	–	–	–	–	0.35	1.42
20.5KB	–	–	–	–	0.44	1.50
25.8KB	–	–	–	–	0.45	1.52

Table 2: Time in seconds for processing Foaf-1D

Cliq-2: `SELECT * WHERE { :a0 ((:p)*)* :a1 },`

for which no implementation was able to handle even a clique with 8 nodes. That is, the implementations fail for the input graph shown in Figure 2 (which occupied only 378 bytes on disk).

To show that this behavior also appears with real data, we devised an experiment with data crawled from the Web. We constructed RDF graphs from foaf documents crawled by following `foaf:knows` links starting from Axel Polleres' foaf document. We considered several test cases of increasing size, from 9.2 KB (38 nodes and 119 triples) to 25.8 KB (76 nodes and 360 triples), and we tested the following simple query asking for the *network of friends* of Axel Polleres[1]:

Foaf-1: `SELECT * WHERE { axel:me (foaf:knows)* ?x }.`

As in the previous case, the results are striking. For query Foaf-1 all the implementations exceeded the timeout for an input RDF graph of 14.8 KB (with only 54 nodes and 201 triples). Table 1 shows the behavior of the different implementations for different input sizes. The "–" symbol in the table means *timeout* (one hour).

As our experiments show, for the tested implementations, property path evaluation is essentially infeasible in practice. But, what is the reason for this behavior? Is this only a problem of the particular implementations that we tested? Or is there a fundamental problem in the SPARQL 1.1 specification? Our theoretical results show that this last question is the key to understand this issue. In fact, we formally prove that, essentially, any implementation that follows the SPARQL 1.1 specification (as of November 2011) [10] will be doomed to show the same behavior.

We begin our theoretical study by formalizing the semantics of property paths. SPARQL 1.1 defines a *bag* (or *multiset*) semantics for these expressions. That is, when evaluating property-path expressions one can obtain several duplicates for the same solution, essentially one duplicate for every different path in the graph satisfying the expression. For example, every solution of query Cliq-1 is an empty tuple that can have several duplicates in the output. ARQ for instance, represents this empty tuple as `|    |`, and for query Cliq-1 it returns several copies of `|    |`. Since RDF graphs containing cycles may lead to an infinite number of paths, the official specification defines the semantics by means of a particular counting procedure, which handles cycles in a way that ensures that the final count is finite. We formalize this procedure, and some other alternative semantics, and prove theoretical bounds on the computational complexity of the evaluation problem, showing that the bag semantics for property paths is the main reason for the infeasibility of the evaluation of property paths in SPARQL 1.1.

Our theoretical study allowed us to prove some extremely large lower bounds for property-path evaluation: for query Cliq-2 and the RDF graph in Figure 2, we show that every implementation that strictly adheres to the SPARQL 1.1 specification should provide as output a file of size more than 79 Yottabytes! It should be noticed that some studies estimate that the cumulative capacity of all the digital stores in the world in 2011 is less than 1 Yottabyte [8] [2].

We study the computational complexity of the setting for property paths proposed by the W3C, as well as for several alternative settings. Given the bag semantics of property paths, we measure the complexity in terms of *counting complexity classes*. The most studied and used intractable counting class is #P [19], which is, intuitively, the counting class associated to the NP problems: while the prototypical NP-complete problem is checking if a propositional formula is satisfiable (SAT), the prototypical #P-complete problem is counting how many truth assignments satisfy a propositional formula (COUNTSAT). We also make a distinction between *data complexity* and *combined complexity*. Data complexity is the complexity of evaluating a query on a database instance assuming that the query is fixed, that is, the complexity is measured only in terms of the size of the database. Combined complexity considers both the query and the database instance as input of the problem [20].

We prove several complexity results. In particular, one of our main results states that property-path evaluation according to the W3C semantics is #P-complete in data complexity. Moreover, we prove that the combined complexity of this problem is not even inside #P. It has been argued that a possibility to deal with the problem of counting paths in the presence of cycles is to consider only simple paths (a simple path is a path with no repeated nodes). We prove that for this alternative semantics the problem is still #P-complete for data complexity, and remains in #P for combined complexity. Thus, although the evaluation problems for these two semantics are intractable, the one based on simple paths has lower combined complexity.

All our results indicate that evaluating property-path queries according to the official SPARQL 1.1 semantics is essentially infeasible. But not all are bad news. A possible solution to this problem is to not use a semantics that considers duplicates, but instead a more traditional existential semantics for path queries, as it has been done for years in graph databases [13, 4, 3], in XML [12, 9], and even on RDF [1, 16] previous to SPARQL 1.1. It is well-known that for this semantics the evaluation problem is tractable, and even linear in data complexity.

As our final theoretical result, we prove that the existential semantics for property paths is equivalent to the SPARQL 1.1 semantics when duplicates are eliminated. Notice that the language has a special feature for this: `SELECT DISTINCT`. Thus, since the existential semantics can be efficiently evaluated, one would expect implementations to take advantage of the `SELECT DISTINCT` feature. Unfortunately, as our final experiments show, no significant improvement in performance can be observed in the tested implementations when the `SELECT DISTINCT` feature is used. For example, consider the following query:

Foaf-1D:
```
SELECT DISTINCT *
WHERE { axel:me (foaf:knows)* ?x }.
```

[1] `axel:` prefix is `<http://www.polleres.net/foaf.rdf#>`.

[2] 1 Yottabyte (YB) = 1 trillion Terabytes.

Although the tested implementations spent less time processing some inputs, none of them was able to process an input file of 14.8 KB (Table 2). As a comparison, we also tested two implementations of existential paths in SPARQL 1.0: Psparql [1] and Gleen [27], which return the same answers as the other tested implementations for the queries using the `SELECT DISTINCT` feature. The numbers speak for themselves (Table 2).

Organization of the paper: Section 2 presents our experiments. Sections 3 and 4 present the formalization of SPARQL 1.1 and property paths. Section 5 presents our main complexity results. In Section 6, we study alternative semantics for counting paths. Section 7 introduces the existential semantics and provide some experimental and theoretical results. Finally, we outline in Section 8 a proposal for a semantics with tractable query evaluation.

2. EXPERIMENTS

In this section, we describe our experimental setting and present more details about the data and results described in the introduction. We assume some familiarity with RDF and the most simple SPARQL features, in particular, with the `SELECT` and `FILTER` keywords [10], and we only treat property paths at an intuitive level (we formalize the language in Sections 3 and 4).

As described in the current SPARQL 1.1 specification, *"a property path is a possible route through a graph between two graph nodes [...] (and) query evaluation determines all matches of a path expression [...]"* [10]. More specifically, property-path expressions are regular expressions over properties (edge labels) in the graph. For example, if `:p` is a property, then `(:p)*` is a property-path expression that matches pairs of nodes that are connected by a sequence of zero or more `:p` properties in the graph. The *star operator* (`*`) and its derivatives (like the *one or more* construct) are the only operators that add expressiveness to the language. The official semantics of the other property-path constructors are defined in terms of SPARQL 1.0 operators [10] (and thus, can be simulated in the previous version of the SPARQL standard). Hence, our tests focus on the star operator, and, in particular, on the most simple expressions that can be generated by using this construct.

2.1 Experimental setting

All our experiments are repeatable; the tools that we implemented for them as well as the data and queries that we used are available at http://www.dcc.uchile.cl/~jperez/papers/www2012/. In our tests, we consider the following SPARQL 1.1 implementations:

ARQ – version 2.8.8, 21 April 2011 [22]: ARQ is a java implementation of SPARQL for Jena [5]. When testing ARQ, we use the command-line tool `sparql` provided in the standard distribution.

RDF::Query – version 2.907, 1 June 2011 [24]: RDF::Query is a perl module implementing SPARQL 1.1, and we test it with the executable tool `query.pl` provided with the standard distribution.

KGRAM – version 3.0, September 2011 [23]: KGRAM [6] provides a set of java libraries that implements SPARQL. To test this engine, we implemented a command-line tool `kgsparql.java`.

Sesame – version 2.5.1, 23 September 2011 [25]: Sesame provides a set of java libraries to execute SPARQL 1.1 queries. To test Sesame, we implemented a command-line tool `sesame.java`.

We run all our tests in a dedicated machine with the following configuration: Debian 6.0.2 Operating System, Kernel 2.6.32, CPU Intel Xeon X3220 Quadcore with 2.40GHz, and 4GB PC2-5300 RAM. Whenever we run a java program, we set the java virtual machine to be able to use all the available RAM (4 GB). All tests were run considering main memory storage. This should not be considered as a problem since the maximum size of the input RDF graphs that we used was only 25.8 KB. We considered a timeout of 60 minutes. For each test, the number reported is the average of the results obtained by executing the test (at least) 4 times. No experiment showed a significant standard deviation.

n	ARQ	RDFQ	Kgram	Sesame	Solutions
5	1.18	0.90	0.57	0.76	16
6	1.19	1.44	0.60	1.24	65
7	1.37	5.09	0.95	2.36	326
8	1.73	34.01	1.38	9.09	1,957
9	2.31	295.88	5.38	165.28	13,700
10	4.15	2899.41	228.68	–	109,601
11	31.21	–	–	–	986,410
12	1422.30	–	–	–	9,864,101
13	–	–	–	–	–

Table 3: Time in seconds and number of solutions for query Cliq-1 (CliqF-1 for RDF::Query)

n	ARQ	RDFQ	Sol.
2	1.40	0.76	1
3	1.19	0.84	6
4	1.65	19.38	305
5	97.06	–	418,576
6	–	–	–

n	ARQ	RDFQ	Sol.
2	1.20	0.77	1
3	1.42	6.85	42
4	–	–	–

Table 4: Time in seconds and number of solutions for queries Cliq-2 (left) and Cliq-3 (right)

2.2 The clique experiment

In our first experiment, we considered cliques (complete graphs) of different sizes, from a clique with 2 nodes (containing 2 triples) to a clique with 13 nodes (156 triples). Query Cliq-1 described in the introduction was the first query to be tested. Since this query has no variables, the solution is an *empty tuple*, which, for example, in ARQ is represented by the string `| |`, and in Sesame by the string `[]` (when the query solution is printed to the standard output). RDF::Query does not support queries without variables, thus for this implementation we tested the following query:

CliqF-1:
```
SELECT * WHERE
{ :a0 (:p)* ?x  FILTER (?x = :a1) }.
```

Table 3 shows the result obtained for this experiment in terms of the time (in seconds) and the number of solutions produced as output, when the input is a clique with n nodes. The symbol "–" in the table means *timeout* of one hour. Notice that this table contains a tabular representation of the numbers shown in Figure 1.

We also tested the impact of using *nested stars*. In particular, we consider query Cliq-2 described in the introduction and query

Cliq-3:
```
SELECT * WHERE { :a0 (((:p)*)*)* :a1 }
```

For these expressions containing nested stars, Sesame produces a run-time error (we have reported this bug in the Sesame's mailing list), and KGRAM does not produce the expected output according to the official SPARQL 1.1 specification [10]. Thus, for these cases it is only meaningful to test ARQ and RDF::Query (we use `FILTER` for RDF::Query, as we did for the case of query CliqF-1). The results are shown in Table 4.

As described in the introduction, our results show the infeasibility of evaluating property paths including the star operator in the the four tested implementations. We emphasize only here the unexpected impact of nesting stars: for query Cliq-3 both implementations that we tested fail for an RDF graph representing a clique

File	#nodes	#triples	size (N3 format)
A	38	119	9.2KB
B	43	143	10.9KB
C	47	150	11.4KB
D	52	176	13.2KB
E	54	201	14.8KB
F	57	237	17.2KB
G	68	281	20.5KB
H	76	360	25.8KB

Table 5: Description of the files (name, number of nodes, number of RDF triples, and size in disk) used in the foaf experiment.

File	ARQ	RDFQ	Kgram	Sesame	Solutions	Size (ARQ)
A	5.13	75.70	313.37	–	29,817	2MB
B	8.20	325.83	–	–	122,631	8.4MB
C	65.87	–	–	–	1,739,331	120MB
D	292.43	–	–	–	8,511,943	587MB
E	–	–	–	–	–	–

Table 6: Time in seconds, number of solutions, and output size for query Foaf-1

with only 4 nodes, which contains only 12 triples and has a size of 126 bytes in N3 notation. Although in this example the nesting of the star operator does not seem to be natural, it is well known that nesting is indeed necessary to represent some regular languages [7]. It is also notable how the number of solutions increase w.r.t. the input size. For instance, for query Cliq-1, ARQ returns more than 9 million solutions for a clique with 12 nodes (ARQ's output in this case has more than 9 million lines containing the string `| |`).

2.3 The foaf experiment

For our second experiment, we use real data crawled from the Web. We decided to consider the `foaf:knows` property, as it has been used as a paradigmatic property for examples regarding path queries (notice that it is in all the examples used to describe property paths in the official SPARQL 1.1 specification [10]).

To construct our datasets we use the SemWeb Client Library [28], which provides a command-line tool `semwebquery` that can be used to query the Web of Linked Data. The tool receives as input a SPARQL query Q, an integer value k and a URI u. When executed, it first retrieves the data from u, evaluates Q over this data, and follows the URIs mentioned in it to obtain more data. This process is repeated k times (see [11] for a description of this query approach). We use a `CONSTRUCT` query to retrieve URIs linked by `foaf:knows` properties with Axel Polleres' foaf document as the starting URI. We set the parameter k as 3, which already gave us a file of 1.5MB containing more than 33,000 triples. To obtain a file of reasonable size, we first filtered the data by removing all triples that mention URIs from large Social Networks sites (in particular, we remove URIs from MyOpera.com and SemanticTweet.com), and then we extracted the *strongly connected component* to which Axel Polleres' URI belongs, obtaining a file of 25.8 KB. From this file, we constructed several test cases by deleting subsets of nodes and then recomputing the strongly connected component. With this process we constructed 8 different test cases from 9.2 KB to 25.8 KB. The description of these files is shown in Table 5. Just as an example of the construction process, file D is constructed from file E by deleting the node corresponding to Richard Cyganiack's URI, and then computing the strongly connected component to which Axel's URI belong. All these files can be downloaded from http://www.dcc.uchile.cl/~jperez/papers/www2012/.

We tested query Foaf-1 described in the introduction, which asks for the network of friends of Axel Polleres. Since the graphs in our test cases are strongly connected, this query retrieves all the nodes in the graph (possibly with duplicates). The time to process the query, the number of solutions produced, and the size of the output produced by ARQ are shown in Table 6 (file E is the last file shown in the table, as all implementations exceed the timeout limit for the larger files). As for the case of the clique experiment, one of the most notable phenomenon is the large increase in the output size.

In the following sections, we provide theoretical results that explain the behavior showed by our tests. We begin by formalizing the SPARQL language and the official semantics of property paths.

3. FORMALIZING SPARQL 1.1

In the following sections, we formalize the semantics of property paths proposed by the W3C [10], and then study the complexity of evaluating property paths under such semantics. To this end, we present in this section an algebraic formalization of the core operators in SPARQL 1.1, which follows the approach given in [14, 15]. Start by assuming there are pairwise disjoint infinite sets $\mathbf{I}$ (IRIs), $\mathbf{B}$ (blank nodes) and $\mathbf{L}$ (literals). A tuple $(s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is called an RDF triple, where s is the subject, p is the predicate and o is the object. A finite set of RDF triples is called an RDF graph. Moreover, assume the existence of an infinite set $\mathbf{V}$ of variables disjoint from the above sets, and assume that every element in $\mathbf{V}$ starts with the symbol ?.

A SPARQL 1.1 graph pattern expression is defined recursively as follows: (1) A tuple from $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is a graph pattern (a triple pattern); (2) if P_1 and P_2 are graph patterns, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, $(P_1 \text{ UNION } P_2)$ and $(P_1 \text{ MINUS } P_2)$ are graph patterns; and (3) if P is a graph pattern and R is a SPARQL 1.1 built-in condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern. In turn, a SPARQL 1.1 built-in condition is constructed using elements of the set $(\mathbf{I} \cup \mathbf{V})$, equality, logical connectives and some built-in predicates [17]. In this paper, we restrict to the fragment where a built-in condition is a Boolean combination of terms constructed by using $=$ and predicate bound, that is, (1) if $?X, ?Y \in \mathbf{V}$ and $c \in \mathbf{I}$, then $\text{bound}(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions; and (2) if R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions. Finally, if P is a graph pattern and W is a set of variables, then $(\text{SELECT } W\ P)$, $(\text{SELECT DISTINCT } W\ P)$, $(\text{SELECT } *\ P)$ and $(\text{SELECT DISTINCT } *\ P)$, are queries in SPARQL 1.1.

To define the semantics of SPARQL 1.1 queries, we borrow some terminology from [17, 15]. A mapping μ is a partial function $\mu : \mathbf{V} \to (\mathbf{I} \cup \mathbf{L})$. Abusing notation, given $a \in (\mathbf{I} \cup \mathbf{L})$ and a mapping μ, we assume that $\mu(a) = a$, and for a triple pattern $t = (s, p, o)$, we assume that $\mu(t) = (\mu(s), \mu(p), \mu(o))$. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of $\mathbf{V}$ where μ is defined. Two mappings μ_1 and μ_2 are compatible, denoted by $\mu_1 \sim \mu_2$, when for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. The mapping with empty domain is denoted by $\mu_\emptyset$ (notice that this mapping is compatible with any other mapping). Finally, given a mapping μ and a set W of variables, the restriction of μ to W, denoted by $\mu_{|_W}$, is a mapping such that $\text{dom}(\mu_{|_W}) = \text{dom}(\mu) \cap W$ and $\mu_{|_W}(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$. Notice that if $W = \emptyset$, then $\mu_{|_W} = \mu_\emptyset$.

The semantics of a SPARQL 1.1 query is defined as a *bag* (or *multiset*) of mappings [10], which is a set of mappings in which every element μ is annotated with a positive integer that represents the cardinality of μ in the bag. Formally, we represent a bag of mappings as a pair $(\Omega, \text{card}_\Omega)$, where Ω is a set of mappings and card_Ω is a function such that $\text{card}_\Omega(\mu)$ is the cardinality of μ in Ω (we assume that $\text{card}_\Omega(\mu) > 0$ for every $\mu \in \Omega$, and $\text{card}_\Omega(\mu') = 0$ for every $\mu' \notin \Omega$). With this notion, we have the necessary ingre-

dients to define the semantics of SPARQL 1.1 queries. As in [14, 15], this semantics is defined as a function $[\![\cdot]\!]_G$ that takes a query and returns a bag of mappings. More precisely, the evaluation of a graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G$, is defined recursively as follows (for the sake of readability, the semantics of filter expressions is presented separately).

- If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(t)$ and $\mu(t) \in G\}$, where $\mathrm{var}(t)$ is the set of variables mentioned in t. Moreover, for every $\mu \in [\![P]\!]_G$: $\mathrm{card}_{[\![P]\!]_G}(\mu) = 1$.
- If P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = \{\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$ and $\mu_1 \sim \mu_2\}$. Moreover, for every $\mu \in [\![P]\!]_G$ we have that $\mathrm{card}_{[\![P]\!]_G}(\mu)$ is given by the expression:

$$\sum_{\mu_1 \in [\![P_1]\!]_G} \left[\sum_{\substack{\mu_2 \in [\![P_2]\!]_G \\ \mu = \mu_1 \cup \mu_2}} \left(\mathrm{card}_{[\![P_1]\!]_G}(\mu_1) \cdot \mathrm{card}_{[\![P_2]\!]_G}(\mu_2) \right) \right].$$

- If P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![(P_1 \text{ AND } P_2)]\!]_G \cup \{\mu \in [\![P_1]\!]_G \mid \forall \mu' \in [\![P_2]\!]_G : \mu \not\sim \mu'\}$. Moreover, for every $\mu \in [\![P]\!]_G$, if $\mu \in [\![(P_1 \text{ AND } P_2)]\!]_G$, then $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![(P_1 \text{ AND } P_2)]\!]_G}(\mu)$, and if $\mu \notin [\![(P_1 \text{ AND } P_2)]\!]_G$, then $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu)$.
- If P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G = \{\mu \mid \mu \in [\![P_1]\!]_G$ or $\mu \in [\![P_2]\!]_G\}$. Moreover, for every $\mu \in [\![P]\!]_G$:

$$\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu) + \mathrm{card}_{[\![P_2]\!]_G}(\mu).$$

- If P is $(P_1 \text{ MINUS } P_2)$, then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G \mid \forall \mu' \in [\![P_2]\!]_G : \mu \not\sim \mu'$ or $\mathrm{dom}(\mu) \cap \mathrm{dom}(\mu') = \emptyset\}$. Moreover, for every $\mu \in [\![P]\!]_G$, it holds that $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu)$.

The evaluation of a SPARQL 1.1 query Q over an RDF graph G, denoted by $[\![Q]\!]_G$, is defined as follows. If Q is a SPARQL 1.1 query (SELECT W P), then $[\![Q]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$ and for every $\mu \in [\![Q]\!]_G$:

$$\mathrm{card}_{[\![Q]\!]_G}(\mu) = \sum_{\mu' \in [\![P]\!]_G \,:\, \mu = \mu'_{|W}} \mathrm{card}_{[\![P]\!]_G}(\mu').$$

If Q is a SPARQL 1.1 query (SELECT $*$ P), then $[\![Q]\!]_G = [\![P]\!]_G$ and $\mathrm{card}_{[\![Q]\!]_G}(\mu) = \mathrm{card}_{[\![P]\!]_G}(\mu)$ for every $\mu \in [\![Q]\!]_G$. If Q is a SPARQL 1.1 query (SELECT DISTINCT W P), then $[\![Q]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$ and for every $\mu \in [\![Q]\!]_G$, we have that $\mathrm{card}_{[\![Q]\!]_G}(\mu) = 1$. Finally, if Q is a SPARQL 1.1 query (SELECT DISTINCT $*$ P), then $[\![Q]\!]_G = [\![P]\!]_G$ and for every $\mu \in [\![Q]\!]_G$, we have that $\mathrm{card}_{[\![Q]\!]_G}(\mu) = 1$.

To conclude the definition of the semantics of SPARQL 1.1, we need to define the semantics of filter expressions. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if (omitting the usual rules for Boolean connectives): (1) R is $\mathrm{bound}(?X)$ and $?X \in \mathrm{dom}(\mu)$; (2) R is $?X = c$, $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) = c$; (3) R is $?X = ?Y$, $?X \in \mathrm{dom}(\mu)$, $?Y \in \mathrm{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$. Then given an RDF graph G and a graph pattern expression $P = (P_1 \text{ FILTER } R)$, we have that $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G \mid \mu \models R\}$, and for every $\mu \in [\![P]\!]_G$, we have that $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu)$.

4. PROPERTY PATHS

In this section, we use the framework presented in the previous section to formalize the semantics of property paths in SPARQL 1.1. According to [10], a property paths is recursively defined as follows: (1) if $a \in \mathbf{I}$, then a is a property path, and (2) if p_1 and p_2 are property paths, then $p_1|p_2$, p_1/p_2 and p_1^* are property paths. Thus, from a syntactical point of view, property paths are regular expressions over the vocabulary $\mathbf{I}$, being $|$ disjunction, $/$ concatenation and $(\ )^*$ the Kleene star. It should be noticed that the definition of property paths in [10] includes some additional features that are common in regular expressions, such as $p?$ (zero or one occurrences of p) and p^+ (one or more occurrences of p). In this paper, we focus on the core operators $|$, $/$ and $(\ )^*$, as they suffice to prove the infeasibility of the evaluation of property paths in SPARQL 1.1.

A property-path triple is a tuple t of the form (u, p, v), where $u, v \in (\mathbf{I} \cup \mathbf{V})$ and p is a property path. SPARQL 1.1 includes as atomic formulas triple patterns and property-path triples. Thus, to complete the definition of the semantics of SPARQL 1.1, we need to specify how property-path triples are evaluated over RDF graphs, that is, we need to extend the definition of the function $[\![\cdot]\!]_G$ to include property-path triples.

To define the semantics of property-path triples we follow closely the standard specification [10]. Assume that $u, v \in (\mathbf{I} \cup \mathbf{V})$, $W = (\{u, v\} \cap \mathbf{V})$ and p is a property path. Notice that if $u, v \in \mathbf{I}$, then $W = \emptyset$. Then the evaluation of property-path triple $t = (u, p, v)$ over an RDF graph G, denoted by $[\![t]\!]_G$, is defined recursively as follows. If $p = a$, where $a \in \mathbf{I}$, then (u, p, v) is a triple pattern and $[\![t]\!]_G$ is defined as in Section 3. Otherwise, we have that either $p = p_1|p_2$ or $p = p_1/p_2$ or $p = p_1^*$, where p_1, p_2 are property paths, and $[\![t]\!]_G$ is defined as follows. First, if $p = p_1|p_2$, then $[\![t]\!]_G$ is defined in [10] as the result of evaluating the pattern $((u, p_1, v) \text{ UNION } (u, p_2, v))$ over G. Thus, we have that:

$$[\![t]\!]_G = \{\mu \mid \mu \in [\![(u, p_1, v)]\!]_G \text{ or } \mu \in [\![(u, p_2, v)]\!]_G\},$$

and for every $\mu \in [\![t]\!]_G$, we have that:

$$\mathrm{card}_{[\![t]\!]_G}(\mu) = \mathrm{card}_{[\![(u,p_1,v)]\!]_G}(\mu) + \mathrm{card}_{[\![(u,p_2,v)]\!]_G}(\mu).$$

Second, if $p = p_1/p_2$, then assuming that $?X$ is a variable such that $?X \notin W$, we have that $[\![t]\!]_G$ is defined in [10] as the result of first evaluating the pattern $((u, p_1, ?X) \text{ AND } (?X, p_2, v))$ over G, and then projecting over the variables of property-path triple t (and, thus, projecting out the variable $?X$). Thus, we have that:

$$[\![t]\!]_G = \{(\mu_1 \cup \mu_2)_{|W} \mid \mu_1 \in [\![(u, p_1, ?X)]\!]_G, \mu_2 \in [\![(?X, p_2, v)]\!]_G \text{ and } \mu_1 \sim \mu_2\},$$

and for every $\mu \in [\![t]\!]_G$, we have that:

$$\mathrm{card}_{[\![t]\!]_G}(\mu) = \sum_{\mu_1 \in [\![(u,p_1,?X)]\!]_G} \Bigg[\sum_{\substack{\mu_2 \in [\![(?X,p_2,v)]\!]_G \\ \mu = (\mu_1 \cup \mu_2)_{|W}}} \left(\mathrm{card}_{[\![(u,p_1,?X)]\!]_G}(\mu_1) \cdot \mathrm{card}_{[\![(?X,p_2,v)]\!]_G}(\mu_2) \right) \Bigg].$$

Finally, if $p = p_1^*$, then $[\![t]\!]_G$ is defined in [10] in terms of the procedures COUNT and ALP shown in Figure 3. More precisely,

$$[\![t]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = W \text{ and } \textsc{Count}(\mu(u), p_1, \mu(v), G) > 0\}.$$

Moreover, for every $\mu \in [\![t]\!]_G$, it holds that

$$\mathrm{card}_{[\![t]\!]_G}(\mu) = \textsc{Count}(\mu(u), p_1, \mu(v), G).$$

Procedure ALP in Figure 3 is taken from [10]. It is important to notice that lines 5 and 6 in ALP formalize, in our terminology, the use of a procedure call `eval` in the definition of ALP in [10]. According to [10], procedure ALP has to be used as follows to compute $\mathrm{card}_{[\![t]\!]_G}(\mu)$, where $t = (u, p_1^*, v)$. Assuming that *Result* is the empty list and *Visited* is the empty set, first one has to invoke

Function COUNT(a, *path*, b, G)
Input: $a, b \in \mathbf{I}$, *path* is a property path and G is an RDF graph.
1: *Result* := empty list
2: *Visited* := empty set
3: ALP(a, *path*, *Result*, *Visited*, G)
4: n := number of occurrences of b in *Result*
5: **return** n

Procedure ALP(a, *path*, *Result*, *Visited*, G)
Input: $a \in \mathbf{I}$, *path* is a property path, *Result* is a list of elements from $\mathbf{I}$, *Visited* is a set of elements from $\mathbf{I}$ and G is an RDF graph.
1: **if** $a \in$ *Visited* **then**
2: **return**
3: **end if**
4: add a to *Visited*, and add a to *Result*
5: $\Omega := [\![(a, \mathit{path}, ?X)]\!]_G$
6: let *Next* be the list of elements $b = \mu(?X)$ for $\mu \in \Omega$, such that the number of occurrences of b in *Next* is $\mathrm{card}_\Omega(\mu)$
7: **for each** $c \in$ *Next* **do**
8: ALP(c, *path*, *Result*, *Visited*, G)
9: **end for**
10: remove a from *Visited*

Figure 3: Procedures used in the evaluation of property-path triples of the form (u, path^*, v).

ALP($\mu(u)$, p, *Result*, *Visited*, G), then one has to check whether $\mu(v)$ appears in the resulting list *Result*, and if this is the case then $\mathrm{card}_{[\![t]\!]_G}(\mu)$ is set as the number of occurrences of $\mu(v)$ in the list *Result*. For the sake of readability, we have encapsulated in the auxiliary procedure COUNT these steps to compute $\mathrm{card}_{[\![t]\!]_G}(\mu)$ from procedure ALP, and we have defined $[\![t]\!]_G$ by using COUNT, thus formalizing the semantics proposed by the W3C in [10].

The idea behind algorithm ALP is to incrementally construct paths that conform to a property path of the form p_1^*, that is, to construct sequences of nodes $a_1, a_2, \ldots, a_n$ from an RDF graph G such that each node a_{i+1} is reachable from a_i in G by following the path p_1, but with the important feature (implemented through the use of the set *Visited*) that each node a_i is distinct from all the previous nodes a_j selected in the sequence (thus avoiding cycles in the sequence $a_1, a_2, \ldots, a_n$).

5. INTRACTABILITY OF SPARQL 1.1 IN THE PRESENCE OF PROPERTY PATHS

In this section, we study the complexity of evaluating property paths according to the semantics proposed by the W3C. Specifically, we study the complexity of computing $\mathrm{card}_{[\![t]\!]_G}(\cdot)$, as this computation embodies the main task needed to evaluate a property-path triple. For the sake of readability, we focus here on computing such functions for property-path triples of the form (a, p, b) where $a, b \in \mathbf{I}$. Notice that this is not a restriction, as for every property path triple t and every mapping μ whose domain is equal to the set of variables mentioned in t, it holds that $\mathrm{card}_{[\![t]\!]_G}(\mu) = \mathrm{card}_{[\![\mu(t)]\!]_G}(\mu_\emptyset)$ (recall that $\mu_\emptyset$ is the mapping with empty domain). Thus, we study the *counting* problem COUNTW3C, whose input is an RDF graph G, elements $a, b \in \mathbf{I}$ and a property path p, and whose output is the value $\mathrm{card}_{[\![(a,p,b)]\!]_G}(\mu_\emptyset)$.

It is important to notice that property paths are part of the input of the previous problem and, thus, we are formalizing the *combined complexity* of the evaluation problem [20]. As it has been observed in many scenarios, and, in particular, in the context of evaluating SPARQL [15], when computing a function like $\mathrm{card}_{[\![(a,p,b)]\!]_G}(\cdot)$, it is natural to assume that the size of p is considerably smaller than the size of G. This assumption is very common when studying the complexity of a query language. In fact, it is named *data complexity* in the database literature [20], and it is defined in our context as the complexity of computing $\mathrm{card}_{[\![(a,p,b)]\!]_G}(\cdot)$ for a fixed property-path p. More precisely, assume given a fixed property path p. Then COUNTW3C(p) is defined as the problem of computing, given an RDF graph G and elements $a, b \in \mathbf{I}$, the value $\mathrm{card}_{[\![(a,p,b)]\!]_G}(\mu_\emptyset)$.

5.1 Property path evaluation

To pinpoint the complexity of COUNTW3C and COUNTW3C(p), where p is a property path, we need to consider the complexity class #P mentioned in the introduction (we refer the reader to [19] for its formal definition). A function f is said to be in #P if there exists a non-deterministic Turing Machine M that works in polynomial time such that for every string w, the value of f on w is equal to the number of accepting runs of M with input w. As mentioned in the introduction, a prototypical #P-complete problem is the problem of computing, given a propositional formula φ, the number of truth assignments satisfying φ. Clearly #P is a class of intractable computation problems [19].

Our first result shows that property path evaluation is intractable.

Theorem 5.1 COUNTW3C(p) *is in* #P *for every property path* p. *Besides,* COUNTW3C(c^*) *is* #P*-complete, where* $c \in \mathbf{I}$.

Theorem 5.1 shows that the problem of evaluating property paths under the semantics proposed by the W3C is intractable in data complexity. In fact, it shows that one will not be able to find efficient algorithms to evaluate even simple property paths such as c^*, where c is an arbitrary element of $\mathbf{I}$.

We now move to the study of the combined complexity of the problem COUNTW3C. In what follows, we formalize the clique experiment presented in Section 2.2, and then provide lower bounds in this scenario for the number of occurrences of a mapping in the result of the procedure (ALP) used by the W3C to define the semantics of property paths [10]. Interestingly, these lower bounds show that the poor behavior detected in the experiments is not a problem with the tested implementations, but instead a characteristic of the semantics of property paths proposed in [10]. These lower bounds provide strong evidence that evaluating property paths under the semantics proposed by the W3C is completely infeasible, as they show that COUNTW3C is not even in #P.

Fix an element $c \in \mathbf{I}$ and an infinite sequence $\{a_i\}_{i \geq 1}$ of pairwise distinct elements from $\mathbf{I}$, which are all different from c. Then for every $n \geq 2$, let $\mathrm{clique}(n)$ be an RDF graph forming a clique with nodes $a_1, \ldots, a_n$ and edge label c, that is, $\mathrm{clique}(n) = \{(a_i, c, a_j) \mid i, j \in \{1, \ldots, n\}$ and $i \neq j\}$. Moreover, for every property path p, define COUNTCLIQUE(p, n) as $\mathrm{card}_{[\![(a_1,p,a_n)]\!]_{\mathrm{clique}(n)}}(\mu_\emptyset)$. Then we have that:

Lemma 5.2 *For every property path p and* $n \geq 2$*:*

$$\textsc{CountClique}(p^*, n) \quad = \quad \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \textsc{CountClique}(p, n)^k}{(n-k-1)!}$$

Let $p_0 = c$ and $p_{s+1} = p_s^*$, for every $s \geq 0$. For example, $p_1 = c^*$ and $p_3 = ((c^*)^*)^*$. From Lemma 5.2, we obtain that:

$$\textsc{CountClique}(p_{s+1}, n) \ = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \textsc{CountClique}(p_s, n)^k}{(n-k-1)!}, \quad (1)$$

for every $s \geq 0$. This formula can be used to obtain the number of occurrences of the mapping with empty domain in the answer to the property-path triple (a_1, p_s, a_n) over the RDF graph $\mathrm{clique}(n)$.

s	n	CountClique(p_s, n)	s	n	CountClique(p_s, n)
1	3	2	1	5	16
2	3	6	2	5	418576
3	3	42	3	5	$> 10^{23}$
4	3	1806	4	5	$> 10^{93}$
1	4	5	1	6	65
2	4	305	2	6	28278702465
3	4	56931605	3	6	$> 10^{53}$
4	4	$> 10^{23}$	4	6	$> 10^{269}$

Table 7: Number of occurrences of the mapping with empty domain in the answer to property-path triple (a_1, p_s, a_n) over the RDF graph $\text{clique}(n)$, according to the semantics for property paths proposed by the W3C in [10].

For instance, the formula states that if a system implements the semantics proposed by the W3C in [10], then with input $\text{clique}(8)$ and $(a_1, (c^*)^*, a_8)$, the empty mapping would have to appear more than $79 \cdot 10^{24}$ times in the output. Thus, even if a single byte is used to store the empty mapping[3], then the output would be of more than 79 Yottabytes in size! Table 7 shows more lower bounds obtained with formula (1). Notice that these numbers coincide with the results obtained in our experiments (Tables 3 and 4). Also notice that, for example, for $n = 6$ and $s = 2$ the lower bound is of more than 28 billions, and for $n = 4$ and $s = 3$ is of more than 56 millions, which explains why the tested implementations exceeded the timeout for queries Cliq-2 and Cliq-3 (Table 4). Most notably, Table 7 allows us to provide a *cosmological lower bound* for evaluating property paths: if one proton is used to store the mapping with empty domain, with input $\text{clique}(6)$ (which contains only 30 triples) and $(a_1, (((c^*)^*)^*)^*, a_6)$, every system implementing the semantics proposed by the W3C [10] would have to return a file that would not fit in the observable universe!

From Lemma 5.2, we obtain the following double-exponential lower bound for CountClique(p_s, n).

Lemma 5.3 *For every $n \geq 2$ and $s \geq 1$:*

$$\text{CountClique}(p_s, n) \quad \geq \quad (n-2)!^{(n-1)^{s-1}}$$

From this bound, we obtain that CountW3C is not in #P. Besides, from the proof of Theorem 5.1, we obtain that CountW3C is in the complexity class #EXP, which is defined as #P but considering non-deterministic Turing Machines that work in exponential time.

Theorem 5.4 CountW3C *is in* #EXP *and not in* #P.

It is open whether CountW3C is #EXP-complete.

5.2 The complexity of the entire language

We consider now the data complexity of the evaluation problem for the entire language. More precisely, we use the results proved in the previous section to show the major impact of using property paths on the complexity of evaluating SPARQL 1.1 queries. The evaluation problem is formalized as follows. Given a fixed SPARQL 1.1 query Q, define EvalW3C(Q) as the problem of computing, given an RDF graph G and a mapping μ, the value $\text{card}_{[\![Q]\!]_G}(\mu)$.

It is easy to see that the data complexity of SPARQL 1.1 without property paths is polynomial. However, from Theorem 5.1, we obtain the following corollary that shows that the data complexity is considerably higher if property paths are included, for the case of the semantics proposed by the W3C [10]. In this corollary, we show that EvalW3C(Q) is in the complexity class FP$^{\#P}$, which is the class of functions that can be computed in polynomial time if one has access to an efficient subroutine for a #P-complete problem (or, more formally, one has an oracle for a #P-complete problem).

Corollary 5.5 EvalW3C(Q) *is in* FP$^{\#P}$*, for every SPARQL 1.1 query Q. Moreover, there exists a SPARQL 1.1 query Q_0 such that* EvalW3C(Q_0) *is* #P*-hard.*

6. INTRACTABILITY FOR ALTERNATIVE SEMANTICS THAT COUNT PATHS

The usual graph theoretical notion of path has been extensively and successfully used when defining the semantics of queries including regular expressions [13, 4, 1, 16, 3]. Nevertheless, given that the W3C SPARQL 1.1 Working Group is interested in counting paths, the classical notion of path in a graph cannot be naively used to define a semantics for property-path queries, given that cycles in an RDF graph may lead to an infinite number of different paths. In this section, we consider two alternatives to deal with this problem. We consider a semantics for property paths based on classical paths that is only defined for *acyclic RDF graphs*, and we consider a general semantics that is based on *simple paths* (which are paths in a graph with no repeated nodes). In both cases, we show that query evaluation based on counting is intractable. Next we formalize these two alternative semantics and present our complexity results.

A path π in an RDF graph G is a sequence $a_1, c_1, a_2, c_2, \ldots, a_n, c_n, a_{n+1}$ such that $n \geq 0$ and $(a_i, c_i, a_{i+1}) \in G$ for every $i \in \{1, \ldots, n\}$. Path π is said to be from a to b in G if $a_1 = a$ and $a_{n+1} = b$, it is said to be nonempty if $n \geq 1$, and it is said to be a *simple path*, or just s-path, if $a_i \neq a_j$ for every distinct pair i, j of elements from $\{1, \ldots, n+1\}$. Finally, given a property path p, path π is said to *conform* to p if $c_1 c_2 \cdots c_n$ is a string in the regular language defined by p.

6.1 Classical paths over acyclic RDF graphs

We first define the semantics of a property-path triple considering classical paths, that we denote by $[\![\cdot]\!]_G^{\text{path}}$. Notice that we have to take into consideration the fact that the number of paths in an RDF graph may be infinite, and thus we define this semantics only for acyclic graphs. More precisely, an RDF graph G is said to be cyclic if there exists an element a mentioned in G and a nonempty path π in G from a to a, and otherwise it is said to be acyclic. Then assuming that G is acyclic, the evaluation of a property-path triple t over G in terms of classical paths, denoted by $[\![t]\!]_G^{\text{path}}$, is defined as follows. Let $t = (u, p, v)$ and $W = (\{u, v\} \cap \mathbf{V})$, then

$$[\![t]\!]_G^{\text{path}} \;=\; \{\mu \mid \text{dom}(\mu) = W \text{ and there exists a path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\},$$

and for every $\mu \in [\![t]\!]_G^{\text{path}}$, the value $\text{card}_{[\![t]\!]_G^{\text{path}}}(\mu)$ is defined as the number of paths from $\mu(u)$ to $\mu(v)$ in G that conform to p.

Similarly as we defined the problem CountW3C in Section 5, we define the problem CountPath as the problem of computing $\text{card}_{[\![(a,p,b)]\!]_G^{\text{path}}}(\mu_\emptyset)$ given as input an acyclic RDF graph G, elements $a, b \in \mathbf{I}$, and property path p. We also define, given a fixed property path p, the problem CountPath(p) as the the problem of computing, given an acyclic RDF graph G and elements $a, b \in \mathbf{I}$, the value $\text{card}_{[\![(a,p,b)]\!]_G^{\text{path}}}(\mu_\emptyset)$.

To pinpoint the exact complexity of the problems CountPath and CountPath(p), we need to consider two counting complexity classes: #L and spanL. We introduce these classes here, and we refer the reader to [2] for their formal definitions. #L is the counting class associated with the problems that can be solved in logarithmic space in a non-deterministic Turing Machine (NTM).

[3]Recall that the empty mapping $\mu_\emptyset$ is represented as the four-bytes string `| |` in ARQ, and as the two-bytes string `[]` in Sesame.

In fact, a function f is said to be in this class if there exists an NTM M that works in logarithmic space such that for every string w, the value of f on w is equal to the number of accepting runs of M with input w. A prototypical #L-complete problem is the problem of computing, given a deterministic finite automaton A and a string w, the number of strings that are accepted by A and whose length is smaller than the length of w [2]. SPANL is defined in a similar way to #L, but considering logarithmic-space NTMs with output. More precisely, a function f is said to be in this class if there exists such TM M such that for every string w, the value of f on w is equal to the number of different outputs of M with input w. A prototypical SPANL-complete problem is the problem of computing, given a non-deterministic finite automaton A and a string w, the number of strings that are accepted by A and whose length is smaller than the length of w [2]. Although classes #L and SPANL look alike, they are quite different in terms of complexity: #L is known to be included in FP, the class of functions that can be computed in polynomial time, while it is known that SPANL is a class of intractable computation problems, if SPANL $\subseteq$ FP, then P = NP.

Our first result shows that even for the simple case considered in this section, the problem of evaluating property paths is intractable.

Theorem 6.1 COUNTPATH *is* SPANL*-complete.*

Interestingly, our second complexity result shows that at least in terms of data complexity, the problem of evaluating property paths is tractable if their semantics is based on the usual notion of path.

Theorem 6.2 COUNTPATH(p) *is in* #L *for every property path* p*. Moreover, there exists a property path* p_0 *such that* COUNTPATH(p_0) *is* #L*-complete.*

Although COUNTPATH(p) is tractable, it only considers acyclic RDF graphs, and thus leaves numerous practical cases uncovered.

6.2 Simple paths

We continue our investigation by considering the alternative semantics for property paths that is defined in terms of simple paths. Notice that even for cyclic RDF graphs, the number of simple paths is finite, and thus, this semantics is properly defined for every RDF graph. Formally, assume that G is an RDF graph, $t = (u, p, v)$ is a property-path triple and $W = (\{u, v\} \cap \mathbf{V})$. The evaluation of t over G in terms of s-paths, denoted by $[\![t]\!]_G^{\text{s-path}}$, is defined as:

$$[\![t]\!]_G^{\text{s-path}} = \{\mu \mid \text{dom}(\mu) = W \text{ and there exists an s-path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\},$$

and for every $\mu \in [\![t]\!]_G^{\text{s-path}}$, the value $\text{card}_{[\![t]\!]_G^{\text{s-path}}}(\mu)$ is defined as the number of s-paths from $\mu(u)$ to $\mu(v)$ in G that conform to p. For the case of s-paths, we define the problem COUNTSIMPLEPATH as follows. The input of this problem is an RDF graph G, elements $a, b \in \mathbf{I}$ and a property path p, and its output is the value $\text{card}_{[\![(a,p,b)]\!]_G^{\text{s-path}}}(\mu_\emptyset)$. As for the previous problems, we define COUNTSIMPLEPATH(p) as COUNTSIMPLEPATH for a fixed property path p. The following result shows that these problems are also intractable.

Theorem 6.3 COUNTSIMPLEPATH *is in* #P*. Moreover, if* $c \in \mathbf{I}$*, then* COUNTSIMPLEPATH(c^*) *is* #P*-complete.*

Notice that the data complexity of evaluating property paths according to the s-path semantics is the same as evaluating them according to the W3C semantics. The difference is in the combined complexity that is radically higher for the W3C semantics: for the case of the semantics based on s-paths the combined complexity is in #P, while for the W3C semantics it is not in #P.

7. AN EXISTENTIAL SEMANTICS TO THE RESCUE

We have shown in the previous section that evaluating property-path triples according to the semantics proposed in [10] is essentially infeasible, being the core of this problem the necessity of counting different paths. We have also shown that the version in which one counts simple-paths is infeasible too. A possible solution to this problem is to not use a semantics that requires counting paths, but instead a more traditional existential semantics for property-path triples. That is, one just checks if two nodes are connected (or not) by a path that conforms to a property-path expression. This existential semantics has been used for years in graph databases [13, 4, 3], in XML [12, 9], and even on RDF [1, 16] previous to SPARQL 1.1. In this section, we introduce this semantics and study the complexity of evaluating property paths, and also SPARQL 1.1 queries, under it. We also compare this proposal with the current official semantics for property paths, and present some experimental results that validate our proposal.

The most natural way to define an existential semantics for property paths is as follows. Assume that $u, v \in (\mathbf{I} \cup \mathbf{V})$, $W = (\{u, v\} \cap \mathbf{V})$, $t = (u, p, v)$ is a property-path triple, and G is an RDF graph. Then define $[\![t]\!]_G^{\exists(\text{path})}$ as:

$$[\![t]\!]_G^{\exists(\text{path})} = \{\mu \mid \text{dom}(\mu) = W \text{ and there exists a path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\}.$$

Moreover, define the cardinality of every mapping μ in $[\![t]\!]_G^{\exists(\text{path})}$ just as 1. Notice that with the semantics $[\![t]\!]_G^{\exists(\text{path})}$, we are essentially discarding all the duplicates from $[\![t]\!]_G^{\text{path}}$. This allows us to consider general graphs (not necessarily acyclic graph as in Section 5). To study the complexity of evaluating property paths under this semantics, we define the decision problem EXISTSPATH, whose input is an RDF graph G, elements $a, b \in \mathbf{I}$ and a property-path triple $t = (a, p, b)$, and whose output is the answer to the question: is $\text{card}_{[\![t]\!]_G^{\exists(\text{path})}}(\mu_\emptyset) = 1$? That is, the problem EXISTSPATH is equivalent to checking whether $\mu_\emptyset \in [\![t]\!]_G^{\exists(\text{path})}$.

Notice that with EXISTSPATH, we are measuring the combined complexity of evaluating paths under the existential semantics. The following result shows that EXISTSPATH is tractable. This is a corollary of some well-known results on graph databases (e.g. see Section 3.1 in [16]). In the result, we use $|G|$ to denote the size of an RDF graph G and $|p|$ to denote the size of a property-path p.

Proposition 7.1 EXISTSPATH *can be solved in time* $O(|G| \cdot |p|)$*.*

7.1 Discarding duplicates from the standard and simple-paths semantics

A natural question at this point is whether there exists a relationship between the existential semantics defined in the previous section and the semantics that can be obtained by discarding duplicates from $[\![t]\!]_G$ and $[\![t]\!]_G^{\text{s-path}}$ for a property-path triple t. We formalize and study these two semantics in this section.

Assume that G is an RDF graph and t is a property-path triple. Then we define $[\![t]\!]_G^{\exists}$ as having exactly the same mappings as in $[\![t]\!]_G$, but with the cardinality of every mapping in $[\![t]\!]_G^{\exists}$ defined just as 1. Similarly, we define $[\![t]\!]_G^{\exists(\text{s-path})}$ as having exactly the same mappings as in $[\![t]\!]_G^{\text{s-path}}$, but with the cardinality of every mapping in $[\![t]\!]_G^{\exists(\text{s-path})}$ defined as 1. In this section, we study the decision problem EXISTSW3C, whose input is an RDF graph G, elements $a, b \in \mathbf{I}$ and a property-path triple $t = (a, p, b)$, and whose output is the answer to the question: is $\text{card}_{[\![(a,p,b)]\!]_G^{\exists}}(\mu_\emptyset) = 1$? We also

study the complexity of the decision problem ExistsSimplePath, which is defined as ExistsW3C but considering the semantics $[\![\cdot]\!]_G^{\exists(\text{s-path})}$ instead of $[\![\cdot]\!]_G^{\exists}$.

Our first result shows that, somehow surprisingly, the semantics $[\![\cdot]\!]_G^{\exists}$ coincides with $[\![\cdot]\!]_G^{\exists(\text{path})}$. Thus, even though the official semantics of property paths is given in terms of a particular procedure [10], when one does not count paths, it coincides with the classical existential semantics based on the usual notion of path.

Theorem 7.2 *For every RDF graph G, mapping μ and property-path triple t:* $\mu \in [\![t]\!]_G^{\exists}$ *if and only if* $\mu \in [\![t]\!]_G^{\exists(\text{path})}$.

As a corollary of Propositions 7.1 and Theorem 7.2, we obtain that:

Theorem 7.3 ExistsW3C *can be solved in time $O(|G| \cdot |p|)$.*

The situation is radically different for the case of simple paths. From some well-known results on graph databases [13], one can prove that ExistsSimplePath is an intractable problem, even for a fixed property-path. More precisely, for a fixed property-path p, the decision problem ExistsSimplePath(p) has as input an RDF graph G and elements $a, b \in \mathbf{I}$, and the question is whether $\text{card}_{[\![(a,p,b)]\!]_G^{\exists(\text{s-path})}}(\mu_\emptyset) = 1$.

Proposition 7.4 ExistsSimplePath *is in* NP. *Moreover,* ExistsSimplePath($(c/c)^*$) *is* NP*-complete, where $c \in \mathbf{I}$.*

7.2 Existential semantics and SPARQL 1.1

We have shown that when bags are considered for the semantics of property paths, the evaluation becomes intractable, even in data complexity. However, the previous version of SPARQL, that did not include path queries, considered a bag semantics for the mapping operators (AND, OPT, UNION, FILTER and SELECT), which has proved to be very useful in practice. Thus, a natural question is whether one can construct a language with functionalities to express interesting queries about paths in RDF graphs, with bag semantics for the mappings operators, and that, at the same time, can be efficiently evaluated. In this section, we give a positive answer to this question. We show that if one combines existential semantics for property paths and bag semantics for the SPARQL 1.1 operators, one obtains the best of both worlds and still has tractable data complexity.

We start by formalizing this alternative way of evaluating SPARQL 1.1 queries that considers existential semantics for property-path triples. Given a SPARQL 1.1 query Q, define $[\![Q]\!]_G^{\exists}$ exactly as $[\![Q]\!]_G$ is defined in Sections 3 and 4, but evaluating property-paths triples according to the semantics $[\![\cdot]\!]_G^{\exists}$ defined in Section 7.1 (that is, $[\![t]\!]_G$ is replaced by $[\![t]\!]_G^{\exists}$ if t is a property-path triple), and likewise for $[\![Q]\!]_G^{\exists(\text{s-path})}$ and $[\![Q]\!]_G^{\exists(\text{path})}$. Notice that for the three semantics $[\![Q]\!]_G^{\exists}$, $[\![Q]\!]_G^{\exists(\text{path})}$ and $[\![Q]\!]_G^{\exists(\text{s-path})}$, we are not discarding all duplicates but only the duplicates that are generated when evaluating property paths. Thus, these semantics are still bag semantics, and therefore we consider the following computation problems. We define first the computation problem EvalExistsW3C(Q), whose input is an RDF graph G and a mapping μ, and whose output is the value $\text{card}_{[\![Q]\!]_G^{\exists}}(\mu)$. Moreover, we also consider the computation problems EvalExistsSimplePath(Q) and EvalExistsPath(Q), that have the same input as EvalExistsW3C(Q) and are defined as the problems of computing $\text{card}_{[\![Q]\!]_G^{\exists(\text{s-path})}}(\mu)$ and $\text{card}_{[\![Q]\!]_G^{\exists(\text{path})}}(\mu)$, respectively. Notice that in these three problems, we are considering the data complexity of SPARQL 1.1 under the respective semantics.

Notably, the next result shows that the just defined semantics $[\![\cdot]\!]_G^{\exists}$ and $[\![\cdot]\!]_G^{\exists(\text{path})}$ are tractable, in terms of data complexity. This result is a consequence of Theorem 7.3 and Proposition 7.1. In the formulation of this result we use the class FP, which is defined as the class of all functions that can be computed in polynomial time (and thus, it is a class of tractable functions).

Theorem 7.5 EvalExistsW3C(Q) *and* EvalExistsPath(Q) *are in* FP *for every SPARQL 1.1 query Q.*

We conclude this section by showing that for the case of the semantics $[\![\cdot]\!]_G^{\exists(\text{s-path})}$, the data complexity is unfortunately still high. To study this problem we need the complexity classes $\text{FP}^{\|\text{NP}}$ and $\text{FP}^{\text{NP}[O(\log n)]}$, which are defined in terms of oracles as for the case of the complexity class $\text{FP}^{\#\text{P}}$ used in Corollary 5.5. More precisely, the class $\text{FP}^{\|\text{NP}}$ contains all the functions that can be computed in polynomial time by a procedure that is equipped with an efficient subroutine (oracle) for an NP-complete problem, with the restriction that all the calls to the subroutine should be made *in parallel*, that is, no call to the subroutine can depend on the result of a previous call to this subroutine [21]. The class $\text{FP}^{\text{NP}[O(\log n)]}$ is defined in the same way, but with the restriction that the subroutine for an NP-complete problem can be called only a logarithmic number of times. Both classes $\text{FP}^{\text{NP}[O(\log n)]}$ and $\text{FP}^{\|\text{NP}}$ are considered to be intractable. Moreover, it is known that $\text{FP}^{\text{NP}[O(\log n)]} \subseteq \text{FP}^{\|\text{NP}}$, but it is open whether this containment is strict [18].

Theorem 7.6 EvalExistsSimplePath(Q) *is in* $\text{FP}^{\|\text{NP}}$ *for every SPARQL 1.1 query Q. Moreover, there exists a query Q_0 such that* EvalExistsSimplePath(Q_0) *is* $\text{FP}^{\text{NP}[O(\log n)]}$*-hard.*

Theorem 7.6 shows that simple paths are not a good option even if duplicates are not considered.

7.3 Experiments for the existential semantics

In the previous section, we showed that SPARQL 1.1 is tractable in terms of data complexity if one considers the existential semantics $[\![\cdot]\!]_G^{\exists}$ and $[\![\cdot]\!]_G^{\exists(\text{path})}$ for property paths. The goal of this section is to show the impact of using these semantics in practice, by conducting a final experiment with two implementations that extends SPARQL 1.0 with existential path semantics: Psparql (version 3.3) [26], and Gleen (version 0.6.1) [27]. These two implementations evaluate SPARQL queries according to $[\![\cdot]\!]_G^{\exists(\text{path})}$, although they use a slightly different syntax for path queries (see http://www.dcc.uchile.cl/~jperez/papers/www2012/ for the definitions of these queries).

In our experiments, we use the following result that allows us to compare SPARQL 1.1 implementations mentioned in Section 2.1 with Psparql and Gleen. It is important to notice that this result is of independent interest, as it shows that the implementations of SPARQL 1.1 that follow the official specification [10] can be highly optimized when using the `SELECT DISTINCT` feature.

Theorem 7.7 *Let P be a SPARQL 1.1 graph pattern, G an RDF graph and W a set of variables. Then we have that:*

$$[\![(\text{SELECT DISTINCT } W\ P)]\!]_G = [\![(\text{SELECT DISTINCT } W\ P)]\!]_G^{\exists(\text{path})}$$

$$[\![(\text{SELECT DISTINCT } *\ P)]\!]_G = [\![(\text{SELECT DISTINCT } *\ P)]\!]_G^{\exists(\text{path})}$$

In view of this theorem, we consider all the queries in Section 2, but this time using the `SELECT DISTINCT` feature:

n	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
8	1.68	32.61	1.39	9.08	0.18	1.24
9	2.00	213.99	5.34	166.82	0.20	1.23
10	3.65	2123.90	227.66	–	0.20	1.25
11	29.71	–	–	–	0.23	1.25
12	1394.06	–	–	–	0.24	1.24
13	–	–	–	–	0.27	1.24

Cliq-1D

n	ARQ	RDFQ	Psparql	Gleen
2	1.40	0.76	0.14	1.23
3	1.19	0.84	0.14	1.23
4	1.65	19.38	0.14	1.23
5	97.06	–	0.15	1.22
6	–	–	0.16	1.23
7	–	–	0.16	1.23

Cliq-2D

n	ARQ	RDFQ	Psparql	Gleen
2	1.18	0.77	0.14	1.24
3	1.41	6.78	0.14	1.23
4	–	–	0.15	1.24
5	–	–	0.15	1.24
6	–	–	0.16	1.24
7	–	–	0.16	1.24

Cliq-3D

Table 8: Time in seconds for queries Cliq-1D, Cliq-2D, and Cliq-3D (symbol "–" means one-hour timeout).

```
Cliq-1D:  SELECT DISTINCT * WHERE { :a0 (:p)* :a1 }
Cliq-2D:  SELECT DISTINCT * WHERE { :a0 ((:p)*)* :a1 }
Cliq-3D:  SELECT DISTINCT * WHERE { :a0 (((:p)*)*)* :a1 }
Foaf-1D:  SELECT DISTINCT *
          WHERE { axel:me (foaf:knows)* ?x }
```

Tables 2 and 8 show the results of this experiment. Although the tested systems return the same results for these queries, the differences in efficiency between the SPARQL 1.1 implementations and the implementations that use an existential semantics are dramatic.

8. OUR RESULTS IN PERSPECTIVE: A PROPOSAL

Our results pose a strong argument against the current semantics of property paths, from both, theory and practice. We have made clear that the main problem is the necessity of counting paths imposed by the current SPARQL 1.1 specification. Our investigation raises several questions, being one of the most important whether there exists such a strong use case for counting paths that will make the designers of the language to stick with the current semantics, even knowing that in simple and natural cases it will lead to completely impractical evaluation procedures. We have searched in the official document and also in the discussions around the design of the language, and to the best of our knowledge, there is no strong use case for counting paths. It should also be noticed that this counting functionality has not been used as a primitive in previous navigational languages for graph structured data.

On the positive side, we have shown that a semantics based on checking the existence of paths (without counting them), has several advantages: it can be easily defined and understood, it is based on years of research and practical experience, and, most importantly, it can be efficiently evaluated. In view of our result that DISTINCT can be used to go from the counting semantics to the existential semantics, one might be tempted to think that users not interested in counting paths can use DISTINCT in queries. We strongly disagree with this view. Bag semantics for relational queries has proved to be essential in practice. Thus, a good language should be able to deal with both characteristics: being able to search for paths using an existential semantics, and, at the same time, having bag semantics for relational-like operators (such as AND and OPT). Our proposal is to have such a semantics for SPARQL 1.1, which corresponds to the semantics $[\![\cdot]\!]^{\exists}_{G}$ defined in Section 7.1, and provide a special feature such as ALL-PATHS, for users that would like to count paths and know the implications of this action.

Although SPARQL is still in its infancy, the increasing interest in managing RDF data is making this language to become more and more popular. To maintain and even increase its adoption rate, the group in charge of the upcoming version of SPARQL should consider all the possible inputs regarding the language. We do think that for the case of property-path evaluation, our proposal, as opposed to the current official semantics, would lead to a wide adoption of the language by practitioners, developers and theoreticians.

Acknowledgments Arenas was supported by Fondecyt-1090565, and Pérez by Fondecyt-11110404 and by VID grant U-Inicia 11/04, Universidad de Chile.

9. REFERENCES

[1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *JWS*, 7(2):57–73, 2009.

[2] C. Àlvarez and B. Jenner. A very hard log-space counting class. *Theor. Comput. Sci.*, 107(1):3–30, 1993.

[3] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3–14, 2010.

[4] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *PODS*, pages 194–204, 1999.

[5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *WWW (Alternate Track Papers & Posters)*, pages 74–83, 2004.

[6] O. Corby and C. Faron-Zucker. The KGRAM abstract machine for knowledge graph querying. In *Web Intelligence*, pages 338–341, 2010.

[7] L. Eggan. Transition graphs and the star-height of regular events. *The Michigan mathematical journal*, 10(4):385–397, 1963.

[8] J. Gantz et al. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. International Data Corporation, White Paper, 2008.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *TODS*, 30(2):444–491, 2005.

[10] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C Working Draft 12 May 2011, http://www.w3.org/TR/2011/WD-sparql11-query-20110512/.

[11] O. Hartig, C. Bizer, and J. C. Freytag. Executing SPARQL queries over the Web of Linked Data. In *ISWC*, pages 293–309, 2009.

[12] M. Marx. Conditional XPath. *TODS*, 30(4):929–959, 2005.

[13] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[14] J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Technical Report, U. de Chile TR/DCC-2006-17, October 2006.

[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.

[16] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *JWS*, 8(4):255–270, 2010.

[17] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation 15 January 2008, http://www.w3.org/TR/rdf-sparql-query/.

[18] A. L. Selman. A taxonomy of complexity classes of functions. *J. Comput. Syst. Sci.*, 48(2):357–381, 1994.

[19] L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[20] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[21] K. W. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theor. Comput. Sci.*, 51:53–80, 1987.

[22] ARQ. http://sourceforge.net/projects/jena/files/ARQ/.

[23] KGRAM. http://www-sop.inria.fr/edelweiss/software/corese/.

[24] RDF::Query. http://search.cpan.org/~gwilliams/RDF-Query/.

[25] Sesame. http://sourceforge.net/projects/sesame/.

[26] Psparql. http://exmo.inrialpes.fr/software/psparql/.

[27] Gleen. http://sig.biostr.washington.edu/projects/ontviews/gleen/.

[28] Semantic Web Client Library. http://www4.wiwiss.fu-berlin.de/bizer/ng4j/semwebclient/.