SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog

Renzo Angles
Universidad de Talca
Talca, Chile
rangles@utalca.cl

Georg Gottlob
University of Oxford
Oxford, United Kingdom
georg.gottlob@cs.ox.ac.uk

Aleksandar Pavlović
TU Wien
Vienna, Austria
apavlovi@dbai.tuwien.ac.at

Reinhard Pichler
TU Wien
Vienna, Austria
pichler@dbai.tuwien.ac.at

Emanuel Sallinger
TU Wien
Vienna, Austria
sallinger@dbai.tuwien.ac.at

ABSTRACT

Over the past decade, Knowledge Graphs have received enormous interest both from industry and from academia. Research in this area has been driven, above all, by the Database (DB) community and the Semantic Web (SW) community. However, there still remains a certain divide between approaches coming from these two communities. For instance, while languages such as SQL or Datalog are widely used in the DB area, a different set of languages such as SPARQL and OWL is used in the SW area. Interoperability between such technologies is still a challenge. The goal of this work is to present a uniform and consistent framework meeting important requirements from both, the SW and DB field.

PVLDB Reference Format:
Renzo Angles, Georg Gottlob, Aleksandar Pavlović, Reinhard Pichler, and Emanuel Sallinger. SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog. PVLDB, 16(13): 4240 - 4253, 2023.
doi:10.14778/3625054.3625061

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/joint-kg-labs/SparqLog.

1 INTRODUCTION

Since Google launched its Knowledge Graph (KG) roughly a decade ago, we have seen intensive work on this topic both in industry and in academia. However, there are two research communities working mostly isolated from each other on the development of KG management systems, namely the *Database* and the *Semantic Web* community. Both of them come with their specific key requirements and they have introduced their own approaches.

Of major importance to the *Semantic Web* (SW) community is the compliance with the relevant W3C standards:

[RQ1] SPARQL Feature Coverage. The query language SPARQL is one of the major Semantic Web standards. Therefore, we require the support of the most commonly used SPARQL features.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625061

[RQ2] Bag Semantics. SPARQL employs per default *bag semantics* (also referred to as *multiset semantics*) unless specified otherwise in a query. We therefore require the support of this.

[RQ3] Ontological Reasoning. OWL 2 QL to support ontological reasoning is a major Semantic Web standard. Technically, for rule-based languages, this means that existential quantification (i.e., “object invention”) in the rule heads is required.

The *Database* (DB) community puts particular emphasis on the expressive power and efficient evaluation of query languages. This leads us to the following additional requirement:

[RQ4] Full Recursion. Full recursion is vital to provide the expressive power needed to support complex querying in business applications and sciences (see e.g., [30]) and it is the main feature of the relational query language Datalog [35]. Starting with SQL-99, recursion has also been integrated into the SQL standard and most relational database management systems have meanwhile incorporated recursion capabilities to increase their expressive power.

Finally, for an approach to be *accepted and used in practice*, we formulate the following requirement for both communities:

[RQ5] Implemented System. Both communities require an implemented system. This makes it possible to verify if the theoretic results are applicable in practice and to evaluate the usefulness of the approach under real-world settings.

The above listed requirements explain why there exists a certain gap between the SW and DB communities. There have been several attempts to close this gap. However, as will be detailed in Section 2, no approach has managed to fulfil the requirements of both sides so far. Indeed, while existing solutions individually satisfy some of the requirements listed above, all of them fail to satisfy central other requirements. The goal of this work is to develop one uniform and consistent framework that satisfies the requirements of both communities. More specifically, our contributions are as follows:

Theoretical Translation. We provide a uniform and complete framework to integrate SPARQL support into a KG language that meets all of the above listed requirements RQ1–RQ5. We have thus extended, simplified and – in some cases – corrected previous approaches of translating SPARQL queries (under both set and bag semantics) to various Datalog dialects [3, 26, 28]. For instance, to the best of our knowledge, all previous translations have missed or did not consider correctly certain aspects of the SPARQL standard of the zero-or-one and zero-or-more property paths.

Translation Engine. We have developed the translation engine SparqLog on top of the Vadalog system that covers most of the considered SPARQL 1.1 functionality. We thus had to fill several gaps between the abstract theory and the practical development of the translation engine. For instance, to support bag semantics, we have designed specific Skolem functions to generate a universal duplicate preservation process. On the other hand, the use of the Vadalog system as the basis of our engine made significant simplifications possible (such as letting Vadalog take care of complex filter constraints) and we also get ontological reasoning "for free". SparqLog therefore supports both query answering and ontological reasoning in a single uniform and consistent system.

Experimental Evaluation. We carry out an extensive empirical evaluation on multiple benchmarks with two main goals in mind: to verify the compliance of SparqLog with the SPARQL standard as well as to compare the performance of our system with comparable ones. It turns out that, while SparqLog covers a great part of the selected SPARQL 1.1 functionality in the correct way, some other systems (specifically Virtuoso) employ a non-standard behaviour on queries containing property paths. As far as query-execution times are concerned, the performance of SparqLog is, in general, comparable to other systems such as the SPARQL system Fuseki or the querying and reasoning system Stardog and it significantly outperforms these systems on complex queries containing recursive property paths and/or involving ontologies.

Structure of the paper. After a review of existing approaches in Section 2, and the preliminaries in Section 3, we present our main results: the general principles of our SparqLog system in Section 4, a more detailed look into the translation engine in Section 5, and an experimental evaluation in Section 6. We conclude with Section 7. Further details on our theoretical translation, implementation, and experimental results are provided in the full technical report of this paper [1]. The source code of SparqLog and all material (queries, input and output data, performance measurements) of our experimental evaluation are provided in the supplementary material[1].

2 RELATED APPROACHES

We review several approaches – both from the Semantic Web and the Database community. This discussion of related approaches is divided into theoretical and practical aspects of our work.

2.1 Theoretical Approaches

Several theoretical research efforts have aimed at bridging the gap between the DB and SW communities.

Translations of SPARQL to Answer Set Programming. In a series of papers, Polleres et al. presented translations of SPARQL and SPARQL 1.1 to various extensions of Datalog. The first translation from SPARQL to Datalog [26] converted SPARQL queries into Datalog programs by employing negation as failure. This translation was later extended by the addition of new features of SPARQL 1.1 and by considering its bag semantics in [28]. Thereby, Polleres and Wallner created a nearly complete translation of SPARQL 1.1 queries to Datalog with disjunction (DLV) programs. However, the translation had two major drawbacks: On the one hand, the chosen target language DLV does not support ontological reasoning as it does not contain existential quantification, thereby missing a key requirement (RQ3) of the Semantic Web community. On the other hand, the requirement of an implemented system (RQ5) is only partially fulfilled, since the prototype implementation *DLVhex-SPARQL Plugin* [27] of the SPARQL to Datalog translation of [26] has not been extended to cover also SPARQL 1.1 and bag semantics.

Alternative Translations of SPARQL to Datalog. An alternative approach of relating SPARQL to non-recursive Datalog with stratified negation (or, equivalently, to Relational Algebra) was presented by Angles and Gutierrez in [2]. The peculiarities of negation in SPARQL were treated in a separate paper [4]. The authors later extended this line of research to an exploration of the bag semantics of SPARQL and a characterization of the structure of its algebra and logic in [3]. They translated a few SPARQL features into a Datalog dialect with bag semantics (multiset non-recursive Datalog with safe negation). This work considered only a small set of SPARQL functionality on a very abstract level and used again a target language that does not support ontological reasoning, failing to meet important requirements (RQ1, RQ3) of the SW community. Most importantly, no implementation exists of the translations provided by Angles and Gutierrez, thus failing to fulfil RQ5.

Supporting Ontological Reasoning via Existential Rules. In [15], Datalog$^\pm$ was presented as a family of languages that are particularly well suited for capturing ontological reasoning. The "+" in Datalog$^\pm$ refers to the crucial extension compared with Datalog by *existential rules*, that is, allowing existentially quantified variables in the rule heads. However, without restrictions, basic reasoning tasks such as answering Conjunctive Queries w.r.t. an ontology given by a set of existential rules become undecidable [23]. Hence, numerous restrictions have been proposed [9, 10, 14, 16, 17, 21] to ensure decidability of such tasks, which led to the "−" in Datalog$^\pm$. Of all variants of Datalog$^\pm$, Warded Datalog$^\pm$ [5] ultimately turned out to constitute the best compromise between complexity and expressiveness and it has been implemented in an industrial-strength system – the Vadalog system [11], thus fulfilling requirement RQ5. However, the requirement of supporting SPARQL (RQ1) with or without bag semantics (RQ2) have not been fulfilled up to now.

Warded Datalog$^\pm$ with Bag Semantics. In [12], it was shown that Warded Datalog$^\pm$ using set semantics can be used to represent Datalog using bag semantics by using existential quantification to introduce new tuple IDs. It was assumed that these results could be leveraged for future translations from SPARQL with bag semantics to Warded Datalog$^\pm$ with set semantics. However, the theoretical translation of SPARQL to Vadalog (RQ1) using these results and also implementation (RQ5) by extending Vadalog were left open in [12] and considered of primary importance for future work.

2.2 Practical Approaches

Several systems have aimed at bridging the gap between DB and SW technologies. The World Wide Web Consortium (W3C) lists *StrixDB*, *DLVhex SPARQL-engine* and *RDFox* as systems that support

[1]https://github.com/joint-kg-labs/SparqLog (last visited 09/25/2023)

SPARQL in combination with Datalog[2]. Furthermore, we also have a look at ontological reasoning systems *Vadalog*, *Graal* and *VLog*, which either understand SPARQL to some extent or, at least in principle, could be extended in order to do so.

DLVhex-SPARQL Plugin. The DLVhex-SPARQL Plugin [27] is a prototype implementation of the SPARQL to Datalog translation in [26]. According to the repository's ReadMe file[3], it supports basic graph patterns, simple conjunctive *FILTER* expressions (such as *ISBOUND*, *ISBLANK*, and arithmetic comparisons), the *UNION*, *OPTIONAL*, and *JOIN* operation. Other operations, language tags, etc. are not supported and query results do not conform to the SPARQL protocol, according to the ReadMe file. Moreover, the limited support of existential quantification (see [20]) by DLV does not suffice for ontological reasoning as required by the OWL 2 QL standard (RQ3). Also the support of bag semantics is missing (RQ2).

RDFox. RDFox is an RDF store developed and maintained at the University of Oxford [25]. It reasons over OWL 2 RL ontologies in Datalog and computes/stores materialisations of the inferred consequences for efficient query answering [25]. The answering process of SPARQL queries is not explained in great detail, except stating that queries are evaluated on top of these materialisations, by employing different scanning algorithms [25]. However, translating SPARQL to Datalog – one of the main goals of this paper – is not supported[4]. Moreover, RDFox does currently not support property paths and some other SPARQL 1.1 features[5] (RQ1).

StrixDB. StrixDB is an RDF store developed as a simple tool for working with middle-sized RDF graphs, supporting SPARQL 1.0 and Datalog reasoning capabilities[6]. To the best of our knowledge, there is no academic paper or technical report that explains the capabilities of the system in greater detail. The *StrixStore* documentation page[7] lists examples of how to integrate Datalog rules into SPARQL queries, to query graphs enhanced by Datalog ontologies. However, translating SPARQL to Datalog is not supported[8]. Moreover, important SPARQL 1.1 features such as aggregation and property paths are not supported by StrixDB (RQ1).

Graal. Graal was developed as a toolkit for querying ontologies with existential rules [8]. The system does not focus on a specific storage system, however specializes in algorithms that can answer queries regardless of the underlying database type [8]. It reaches this flexibility, by translating queries from their host system language into Datalog$^{\pm}$. However, it pays the trade-off of restricting itself to answering conjunctive queries only [8] and therefore supports merely a small subset of SPARQL features[9] — e.g. basic features such as *UNION* or *MINUS* are missing (RQ1).

VLog. VLog is a rule engine, developed at the TU Dresden [18]. The system transfers incoming SPARQL queries to specified external SPARQL endpoints such as Wikidata and DBpedia and incorporates the received query results into their knowledge base [18]. Therefore, the responsibility of query answering is handed over to RDF triple stores that provide a SPARQL query answering endpoint, thus failing to provide a uniform, integrated framework for combining query answering with ontological reasoning (RQ5).

```
SELECT ?N ?L
FROM <http://example.org/graph.rdf>
WHERE { ?X <http://ex.org/name> ?N
        . OPTIONAL { ?X <http://ex.org/lastname> ?L }}
ORDER BY ?N
```

Figure 1: Example of SPARQL query.

The Vadalog system [11] is a KG management system implementing the logic-based language Warded Datalog$^{\pm}$. It extends Datalog by including existential quantification necessary for ontological reasoning, while maintaining reasonable complexity. As an extension of Datalog, it supports full recursion. Although Warded Datalog$^{\pm}$ has the capabilities to support SPARQL 1.1 under the OWL 2 QL entailment regime [5] (considering set semantics though!), no complete theoretical nor any practical translation from SPARQL 1.1 to Warded Datalog$^{\pm}$ exists. Therefore, the bag semantics (RQ2) and SPARQL feature coverage (RQ1) requirements are not met.

3 PRELIMINARIES

3.1 RDF and SPARQL

RDF [19] is a W3C standard that defines a graph data model for describing Web resources. The RDF data model assumes three data domains: *IRIs* that identify Web resources, *literals* that represent simple values, and *blank nodes* that identify anonymous resources. An *RDF triple* is a tuple (s, p, o), where s is the subject, p is the predicate, o is the object, all the components can be IRIs, the subject and the object can alternatively be a blank node, and the object can also be a literal. An *RDF graph* is a set of RDF triples. A *named graph* is an RDF graph identified by an IRI. An *RDF dataset* is a structure formed by a default graph and zero or more named graphs.

For example, consider that <http://example.org/graph.rdf> is an IRI that identifies an RDF graph with the following RDF triples:

```
<http://ex.org/glucas> <http://ex.org/name> "George"
<http://ex.org/glucas> <http://ex.org/lastname> "Lucas"
_:b1 <http://ex.org/name> "Steven"
```

This graph describes information about film directors. Each line is an RDF triple, <http://ex.org/glucas> is an IRI, "George" is a literal, and _:b1 is a blank node.

SPARQL [22, 29] is the standard query language for RDF. The general structure of a SPARQL query is shown in Figure 1, where: the SELECT clause defines the output of the query, the FROM clause defines the input of the query (i.e. an RDF dataset), and the WHERE clause defines a graph pattern.

The evaluation of a query begins with the construction of the RDF dataset to be queried, whose graphs are defined by one or more dataset clauses. A *dataset clause* is either an expression FROM u or FROM NAMED u, where u is an IRI that refers to an RDF graph. The former clause merges a graph into the default graph of the dataset, and the latter adds a named graph to the dataset.

The WHERE clause defines a graph pattern (GP). There are many types of GPs: triple patterns (RDF triples extended with variables),

[2] https://www.w3.org/wiki/SparqlImplementations (last visited 09/25/2023)
[3] https://sourceforge.net/p/dlvhex-semweb/code/HEAD/tree/dlvhex-sparqlplugin/trunk/README (last visited 09/25/2023)
[4] see https://docs.oxfordsemantic.tech/reasoning.html (last visited 09/25/2023)
[5] https://docs.oxfordsemantic.tech/3.1/querying-rdfox.html#query-language (last visited 09/25/2023)
[6] http://opoirel.free.fr/strixDB/ (last visited 09/25/2023)
[7] http://opoirel.free.fr/strixDB/DOC/StrixStore_doc.html (last visited 09/25/2023)
[8] see http://opoirel.free.fr/strixDB/dbfeatures.html (last visited 09/25/2023)
[9] https://graphik-team.github.io/graal/ (last visited 09/25/2023)

basic GPs (a set of GPs), optional GPs, alternative GPs (UNION), GPs on named graphs (GRAPH), negation of GPs (NOT EXISTS and MINUS), GPs with constraints (FILTER), existential GPs (EXISTS), and nesting of GPs (SubQueries). A property path is a special GP which allows to express different types of reachability queries.

The result of evaluating a graph pattern is a multiset of solution mappings. A *solution mapping* is a set of variable-value assignments. E.g., the evaluation of the query in Figure 1 over the above RDF graph returns two mappings $\{\mu_1, \mu 2\}$ with μ_1(`?N`) = `"George"`, μ_1(`?L`) = `"Lucas"` and μ_1(`?N`) = `"Steven"`.

The graph pattern matching step returns a multiset whose solution mappings are treated as a sequence without specific order. Such a sequence can be arranged by using solution modifiers: ORDER BY allows to sort the solutions; DISTINCT eliminates duplicate solutions; OFFSET allows to skip a given number of solutions; and LIMIT restricts the number of output solutions.

Given the multiset of solution mappings, the final output is defined by a *query form*: SELECT projects the variables of the solutions; ASK returns *true* if the multiset of solutions is non-empty and *false* otherwise; CONSTRUCT returns an RDF graph whose content is determined by a set of triple templates; and DESCRIBE returns an RDF graph that describes the resources found.

3.2 Warded Datalog$^\pm$ and the Vadalog System

In [15], Datalog$^\pm$ was presented as a family of languages that extend Datalog (whence the +) to increase its expressive power but also impose restrictions (whence the −) to ensure decidability of answering Conjunctive Queries (CQs). The extension most relevant for our purposes is allowing *existential rules* of the form

$$\exists \bar{z} P(\bar{x}', \bar{z}) \ \leftarrow\ P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n),$$

with $\bar{x}' \subseteq \bigcup_i \bar{x}_i$, and $\bar{z} \cap \bigcup_i \bar{x}_i = \emptyset$. Datalog$^\pm$ is thus well suited to capture ontological reasoning. Ontology-mediated query answering is defined by considering a given database D and program Π as logical theories. The answers to a CQ $Q(\vec{z})$ with free variables $\vec{z}$ over database D under the ontology expressed by Datalog$^\pm$ program Π are defined as $\{\vec{a} \mid \Pi \cup D \models Q(\vec{a})\}$, where $\vec{a}$ is a tuple of the same arity as $\vec{z}$ with values from the domain of D.

Several subclasses of Datalog$^\pm$ have been presented [5, 9, 10, 14, 16, 17, 21] that ensure decidability of CQ answering (see [14] for an overview). One such subclass is *Warded* Datalog$^\pm$ [5], which makes CQ answering even tractable (data complexity). For a formal definition of *Warded* Datalog$^\pm$, see [5]. We give the intuition of *Warded* Datalog$^\pm$ here. First, for all positions in rules of a program Π, distinguish if they are *affected* or not: a position is affected, if the chase may introduce a labelled null here, i.e., a position in a head atom either with an existential variable or with a variable that occurs only in affected positions in the body. Then, for variables occurring in a rule ρ of Π, we identify the *dangerous* ones: a variable is dangerous in ρ, if it may propagate a null in the chase, i.e., it appears in the head and all its occurrences in the body of ρ are at affected positions. A Datalog$^\pm$ program Π is *warded* if all rules $\rho \in \Pi$ satisfy: either ρ contains no dangerous variable or all dangerous variables of ρ occur in a single body atom A (= the "ward") such that the variables shared by A and the remaining body occur in at least one non-affected position (i.e., they cannot propagate nulls).

Apart from the favourable computational properties, another important aspect of Warded Datalog$^\pm$ is that a full-fledged engine (even with further extensions) exists: the Vadalog system [11]. It combines full support of Warded Datalog$^\pm$ plus a number of extensions needed for practical use, including (decidable) arithmetics, aggregation, and other features. It has been deployed in numerous industrial scenarios, including the finance sector as well as the supply chain and logistics sector.

4 THE SPARQLOG SYSTEM

This section introduces SparqLog, a system that allows to evaluate SPARQL 1.1 queries on top of the Vadalog system. To the best of our knowledge, SparqLog is the first system that provides a complete translation engine from SPARQL 1.1 with bag semantics to Datalog. In order to obtain a functional and efficient system, we combined the knowledge provided by the theoretical work with database implementation techniques.

SparqLog implements three translation methods: (i) a *data translation method* T_D which generates Datalog$^\pm$ rules from an RDF Dataset; (ii) a *query translation* method T_Q which generates Datalog$^\pm$ rules from a SPARQL query; and (iii) a *solution translation method* T_S which generates a SPARQL solution from a Datalog$^\pm$ solution. Hence, given an RDF dataset D and a SPARQL query Q, SparqLog generates a Datalog$^\pm$ program Π as the union of the rules returned by T_D and T_Q, then evaluates the program Π, and uses T_S to transform the resulting Datalog$^\pm$ solution into a SPARQL solution.

4.1 Example of Graph Pattern Translation

In order to give a general idea of the translation, we will sketch the translation of the RDF graph and the SPARQL query presented in Section 3.1. To facilitate the notation, we will abbreviate the IRIs by using their prefix-based representation. For example, the IRI `http://ex.org/name` will be represented as `ex:name`, where `ex` is a prefix bound to the namespace `http://ex.org/`. Additionally, we will use `graph.rdf` instead of `http://example.org/graph.rdf`.

4.1.1 Data translation. Consider the RDF graph G presented in Section 3.1. First, the data translation method T_D generates a special fact for every RDF term (i.e., IRI, literal, and blank node) in G:

```
iri("ex:glucas"). iri("ex:name"). iri("ex:lastname").
literal("George"). literal("Lucas"). literal("Steven").
bnode("b1").
```

These facts are complemented by the following rules, which represent the domain of RDF terms:

```
term(X) :- iri(X).
term(X) :- literal(X).
term(X) :- bnode(X).
```

For each RDF triple `(s,p,o)` in graph G with IRI `g`, T_D generates a fact `triple(s,p,o,g)`. Hence, in our example, T_D produces:

```
triple("ex:glucas", "ex:name", "George", "graph.rdf").
triple("ex:glucas", "ex:lastname", "Lucas", "graph.rdf").
triple("b1", "ex:name", "Steven", "graph.rdf").
```

4.1.2 Query translation. Assume that Q is the SPARQL query presented in Figure 1. The application of the query translation method T_Q over Q returns the Datalog$^\pm$ rules shown in Figure 2. The general principles of the translation will be discussed in Section 5.1. In

```
1  // SELECT ?N ?L
2  ans(ID, L, N, D) :- ans1(ID1, L, N, X, D),
3    ID = ["f", L, N, X, ID1].
4  // P1 = { P2 . OPTIONAL { P3 } }
5  ans1(ID1, V2_L, N, X, D) :- ans2(ID2, N, X, D),
6     ans3(ID3, V2_L, V2_X, D), comp(X, V2_X, X),
7     ID1 = ["f1a", X, N, V2_X, V2_L, ID2, ID3].
8  ans1(ID1, L, N, X, D) :- ans2(ID2, N, X, D),
9     not ans_opt1(N, X, D), null(L),
10    ID1 = ["f1b", L, N, X, ID2].
11 ans_opt1(N, X, D) :- ans2(ID2, N, X, D),
12    ans3(ID3, V2_L, V2_X, D), comp(X, V2_X, X).
13 // P2 = ?X ex:name ?N
14 ans2(ID2, N, X, D) :-
15    triple(X, "ex:name",  N, D),
16    D = "default",
17    ID2 = ["f2", X, "ex:name",  N, D].
18 // P3 = ?X ex:lastname ?L
19 ans3(ID3, L, X, D) :-
20    triple(X, "ex:lastname",  L, D),
21    D = "default",
22    ID3 = ["f3", X, "ex:lastname",  L, D].
23 @post("ans", "orderby(2)").
24 @output("ans").
```

Figure 2: Datalog$^{\pm}$ rules for SPARQL query Q in Figure 1.

the interest of readability, we slightly simplify the presentation, e.g., by omitting language tags and type definitions and using simple (intuitive) variable names (rather than more complex ones as would be generated by SparqLog to rule out name clashes).

The query translation method T_Q produces rules for each language construct of SPARQL 1.1 plus rules defining several auxiliary predicates. In addition, also system instructions (e.g., to indicate the answer predicate or ordering requirements) are generated. The translation begins with the `WHERE` clause, then continues with the `SELECT` clause, and finalizes with the `ORDER BY` clause.

The most complex part of T_Q is the translation of the graph pattern defined in the `WHERE` clause. In our example, the graph pattern defined by the `WHERE` clause is of the form $P_1 = P_2$ `OPTIONAL` P_3 with triple patterns P_2 = `?X ex:name ?N` and P_3 = `?X ex:lastname ?L`. The instruction `@output` (line 24) is used to define the literal of the goal rule `ans`. It realises the projection defined by the `SELECT` clause. The instruction `@post("ans","orderby(2)")` (line 23) realises the `ORDER BY` clause; it indicates a sort operation over the elements in the second position of the goal rule `ans(ID,L,N,D)`, i.e. sorting by `N` (note that `ID` is at position 0). The `ans` predicate is defined (lines 2–3) by projecting out the `X` variable from the `ans1` relation, which contains the result of evaluating pattern P_1. The tuple IDs are generated as Skolem terms (line 3 for `ans`; likewise lines 7, 10, 17, 22). In this example, we assume that the pattern P_1 and its subpatterns P_2 and P_3 are evaluated over the default graph. This is explicitly defined for the basic graph patterns (lines 15, 20) and propagated by the last argument `D` of the answer predicates.

The OPTIONAL pattern P_1 gives rise to 3 rules defining the predicate `ans1`: a rule (lines 11–12) to define the predicate `ans_opt1`, which computes those mappings for pattern P_2 that can be extended to mappings of P_3; a rule (lines 5–7) to compute those tuples of `ans1` that are obtained by extending mappings of P_2 to mappings of P_3; and finally a rule (lines 8–10) to compute those tuples of `ans1` that are obtained from mappings of P_2 that have no extension to mappings of P_3. In the latter case, the additional variables of P_3 (here: only variable `L`) are set to `null` (line 9). The two basic graph patterns P_2 and P_3 are translated to rules for the predicates `ans2` (lines 14–17) and `ans3` (lines 19–22) in the obvious way.

4.1.3 Solution translation. The evaluation of the program Π produced by the data translation and query translation methods yields a set of ground atoms for the goal predicate p. In our example, we thus get two ground atoms: `ans(id1, "George","Lucas","graph.rdf")` and `ans(id2, "Steven","null","graph.rdf")`. Note that the ground atoms are guaranteed to have pairwise distinct tuple IDs. These ground atoms can be easily translated to the *multiset* of solution mappings by projecting out the tuple ID. Due to the simplicity of our example, we only get a *set* $\{\mu_1, \mu_2\}$ of solution mappings with μ_1(?N) = "George", μ_1(?L) = "Lucas" and μ_2(?N) = "Steven".

4.2 Example of Property Path Translation

A property path is a feature of the SPARQL query language that allows the user to query for complex paths between nodes, instead of being limited to graph patterns with a fixed structure. SPARQL defines different types of property path, named: PredicatePath, InversePath, SequencePath, AlternativePath, ZeroOrMorePath, OneOrMorePath, ZeroOrOnePath and NegatedPropertySet. Next we present an example to show the translation of property paths.

Assume that `<http://example.org/countries.rdf>` identifies an RDF graph with the following prefixed RDF triples:

```
@prefix ex: <http://ex.org/> .
ex:spain ex:borders ex:france .
ex:france ex:borders ex:belgium .
ex:france ex:borders ex:germany .
ex:belgium ex:borders ex:germany .
ex:germany ex:borders ex:austria
```

Note that each triple describes two bordered countries in Europe. Recall that `ex` is a prefix for the namespace `http://ex.org/`, meaning, e.g., that `ex:spain` is the abbreviation of `http://ex.org/spain`.

A natural query could be asking for the countries than can be visited by starting a trip in *Spain*. In other terms, we would like the get the nodes (countries) reachable from the node representing *Spain*. Although the above query could be expressed by computing the union of different fixed patterns (i.e. one-country trip, two-country trip, etc.), the appropriate way is to use the SPARQL query shown in Figure 3. The result of this query is the set $\{\mu_1, \mu_2, \mu_3, \mu_4\}$ of mappings with $\mu_1(?B)$ = `ex:france`, $\mu_2(?B)$ = `ex:germany`, $\mu_3(?B)$ = `ex:austria`, and $\mu_4(?B)$ = `ex:belgium`.

A property path pattern is a generalization of a triple pattern (s, p, o) where the predicate p is extended to be a regular expression called a property path expression. Hence, the expression `?A`

```
PREFIX ex: <http://ex.org/>
SELECT ?B
FROM <http://example.org/countries.rdf>
WHERE { ?A ex:borders+ ?B . FILTER (?A = ex:spain) }
```

Figure 3: Example of SPARQL property path query.

```
1  // P1 = "{?A ex:borders+ ?B . FILTER (?A = ex:spain)}"
2  ans1(ID1,A,B,D) :- ans2(ID2,A,B,D),
3                     X = "ex:spain", ID1 = [...].
4  // P2 = "?A ex:borders+ ?B"
5  ans2(ID2,X,Y,D) :- ans3(ID3,X,Y,D), ID2 = [...].
6  // PP3 = "ex:borders+"
7  ans3(ID3,X,Y,D) :- ans4(ID4,X,Y,D), ID4 = [].
8  ans3(ID3,X,Z,D) :- ans4(ID4,X,Y,D),
9                     ans3(ID31,Y,Z,D), ID4 = [].
10 // PP4 = "ex:borders"
11 ans4(ID4,X,Y,D) :- triple(X,"ex:borders",Y,D),
12                    D = "default", ID4 = [...].
13 @output("ans1").
```

Figure 4: Datalog$^{\pm}$ rules obtained after translating the SPARQL property path pattern shown in Figure 3.

ex:borders+ ?B shown in Figure 3 is a property path pattern, where the property path expression ex:borders+ allows to return all the nodes ?B reachable from node ?A by following one or more matches of edges with ex:borders label. The FILTER condition restricts the solution mappings to those where variable ?*A* is bound to ex:spain, i.e. pairs of nodes where the source node is *spain*. Finally, the SELECT clause projects the result to variable ?B, i.e., the target nodes.

In Figure 4, we show the Datalog$^{\pm}$ rules obtained by translating the graph pattern shown in Figure 3. The rule in line 2 corresponds to the translation of the filter graph pattern. The rule in line 5 is the translation of the property path pattern ?A ex:borders+ ?B. The rules shown in lines 8 and 9 demonstrate the use of recursion to emulate the property path expression ex:borders+. The rule in line 11 is the translation of ex:borders which is called a link property path expression. The general principles of the translation of property paths will be discussed in Section 5.2.

4.3 Coverage of SPARQL 1.1 Features

In order to develop a realistic integration framework between SPARQL and Vadalog, we conduct a prioritisation of SPARQL features. We first lay our focus on basic features, such as *terms* and *graph patterns*. Next, we prepare a more detailed prioritisation by considering the results of Bonifati et al. [13], who examined the real-world adoption of SPARQL features by analysing a massive amount of real-world query-logs from different well-established Semantic Web sources. Additionally, we study further interesting properties of SPARQL, for instance SPARQL's approach to support partial recursion (through the addition of property paths) or interesting edge cases (such as the combination of *Filter* and *Optional* features) for which a "special" treatment is required.

The outcome of our prioritisation step is shown in Table 1. For each feature, we present its real-world usage according to [13] and its current implementation status in our SparqLog system. The table represents the real-world usage by a percentage value (drawn from [13]) in the feature usage field, if [13] covers the feature, "Unknown" if [13] does not cover it, and "Basic Feature" if we consider the feature as fundamental to SPARQL. Note that some features are supported by SparqLog with minor restrictions, such as ORDER BY for which we did not re-implement the sorting strategy defined by the SPARQL standard, but directly use the sorting strategy employed by the Vadalog system. Table 1 reveals that our SparqLog engine covers all features that are used in more than 5% of the queries in practice and are deemed therefore to be of highest relevance to SPARQL users. Some of these features have a rather low usage in practice ($< 1\%$), however are still supported by our engine. These features include *property paths* and GROUP BY. We have chosen to add *property paths* to our engine, as they are not only interesting for being SPARQL's approach to support partial recursion but, according to [13], there are datasets that make extensive use of them. Moreover, we have chosen to add GROUP BY and some aggregates (e.g. COUNT), as they are very important in traditional database settings, and thus are important to establish a bridge between the Semantic Web and Database communities.

In addition to these most widely used features, we have covered all features occurring in critical benchmarks (see Section 6.1 for a detailed discussion). Specifically, as used in the FEASIBLE benchmark, we cover the following features: ORDER BY with complex arguments (such as ORDER BY with BOUND conditions), functions on strings such as UCASE, the DATATYPE function, LIMIT, and OFFSET. For the gMark benchmark, we cover the "exactly n occurrences" property path, "n or more occurrences" property path, and the "between 0 and n occurrences" property path.

Among our contributions, concerning the translation of SPARQL to Datalog, are: the available translation methods have been combined into a uniform and practical framework for translating RDF datasets and SPARQL queries to Warded Datalog± programs; we have developed simpler translations for MINUS and OPT, compared with [28]; we provide translations for both bag and set semantics, thus covering queries with and without the DISTINCT keyword; we have enhanced current translations by adding partial support for data types and language tags; we have developed a novel duplicate preservation model based on the abstract theories of ID generation (this was required because plain existential ID generation turned out to be problematic due to some peculiarities of the Vadalog system); and we propose a complete method for translating property paths, including zero-or-one and zero-or-more property paths.

There are also a few features that have a real-world usage of slightly above one percent and which are currently not supported by SparqLog. Among these features are CONSTRUCT, DESCRIBE, and FILTER NOT EXISTS. We do not support features CONSTRUCT and DESCRIBE, as these solution modifiers do not yield any interesting theoretical or practical challenges and they did not occur in any of the benchmarks chosen for our experimental evaluation. The features for query federation are out of the considered the scope, as our translation engine demands RDF datasets to be translated to the Vadalog system for query answering. Furthermore, SPARQL query federation is used in less than 1% of SPARQL queries [13].

5 SPARQL TO DATALOG$^{\pm}$ TRANSLATION

In this section, we present some general principles of our translation from SPARQL queries into Datalog$^{\pm}$ programs. We thus first discuss the translation of graph patterns (Section 5.1), and then treat the translation of a property paths separately (Section 5.2). We conclude this section with a discussion of the correctness of our translation (Section 5.3). Full details of the translation and its correctness proof are given in the full version of the paper [1].

Table 1: Selected SPARQL features, including their real-world usage according to [13] and the current status in SparqLog.

General Feature	Specific Feature	Feature Usage	Status
Terms	IRIs, Literals, Blank nodes	Basic Feature	✓
Semantics	Sets, Bags	Basic Feature	✓
Graph patterns	Triple pattern	Basic Feature	✓
	AND / JOIN	28.25%	✓
	OPTIONAL	16.21%	✓
	UNION	18.63%	✓
	GROUP Graph Pattern	< 1%	✗
Filter constraints	Equality / Inequality	All Constraints 40.15%	✓
	Arithmetic Comparison		✓
	bound, isIRI, isBlank, isLiteral		✓
	Regex		✓
	AND, OR, NOT		✓
Query forms	SELECT	87.97%	✓
	ASK	4.97%	✓
	CONSTRUCT	4.49%	✗
	DESCRIBE	2.47%	✗
Solution modifiers	ORDER BY	2.06%	✓
	DISTINCT	21.72%	✓
	LIMIT	17.00%	✓
	OFFSET	6.15%	✓
RDF datasets	GRAPH ?x { ... }	2.71%	✓
	FROM (NAMED)	Unknown	✗
Negation	MINUS	1.36%	✓
	FILTER NOT EXISTS	1.65%	✗
Property paths	LinkPath (X exp Y)	< 1%	✓
	InversePath (^exp)	< 1%	✓
	SequencePath (exp1 / exp2)	< 1%	✓
	AlternativePath (exp1 | exp2)	< 1%	✓
	ZeroOrMorePath (exp*)	< 1%	✓
	OneOrMorePath (exp+)	< 1%	✓
	ZeroOrOnePath (expr?)	< 1%	✓
	NegatedPropertySet (!expr)	< 1%	✓
Assignment	BIND	< 1%	✗
	VALUES	< 1%	✗
Aggregates	GROUP BY	< 1%	✓
	HAVING	< 1%	✗
Sub-Queries	Sub-Select Graph Pattern	< 1%	✗
	FILTER EXISTS	< 1%	✗
Filter functions	Coalesce	Unknown	✗
	IN / NOT IN	Unknown	✗

5.1 Translation of Graph Patterns

Let P be a SPARQL graph pattern and a D be an RDF dataset $D = \langle G, G_{named} \rangle$ where G is the default graph and G_{named} is the set of named graphs. The translation of graph patterns is realised by the translation function $\tau(P, dst, D, NodeIndex)$ where: P is the graph pattern that should be translated next; dst (short for "distinct") is a Boolean value that describes whether the result should have set semantics ($dst = true$) or bag semantics ($dst = false$); D is the graph on which the pattern should be evaluated; *NodeIndex* is the index of the pattern P to be translated; and the output of function τ is a set of Datalog$^\pm$ rules.

The function τ for different types of graph patterns is presented in Figure 5. In the sequel, we concentrate on bag semantics (i.e., *dst = false*), since this is the more complex case. To improve readability, we apply the simplified notation used in Figure 2 now also to Figure 5. Additionally, we omit the explicit generation of IDs via Skolem functions and simply put a fresh ID-variable in the first position of the head atoms of the rules.

Triple pattern. Let P_i be the i-th subpattern of P and let P_i be a triple pattern (s, p, o). Then $\tau(P_i, false, D, i)$ is defined as:

$ans_i(Id, \overline{var}(P_i), D)$:- $triple(s, p, o, D)$.

$ans_i(Id, \overline{var}(P_i), g)$:- $ans_{2i}(Id_1, \overline{var}(P_1), g), named(g)$.
$\tau(P_1, false, g, 2i)$

Join. Let P_i be the i-th subpattern of P and let P_i be of the form $(P_1 \,.\, P_2)$. Then $\tau(P_i, false, D, i)$ is defined as:

$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i}(Id_1, v_1(\overline{var}(P_1)), D)$,
$ans_{2i+1}(Id_2, v_2(\overline{var}(P_2)), D)$,
$comp(v_1(x_1), v_2(x_1), x_1), \ldots, comp(v_1(x_n), v_2(x_n), x_n)$.
$\tau(P_1, false, D, 2i)$.
$\tau(P_2, false, D, 2i + 1)$.

Here we are using the following notation:

- $\overline{var}(P_i) = \overline{var(P_1) \cup var(P_2)}$
- $\{x_1, \ldots, x_n\} = var(P_1) \cap var(P_2)$
- $v_1, v_2 : var(P_1) \cap var(P_2) \rightarrow V$, such that $Image(v_1) \cap Image(v_2) = \emptyset$

Filter. Let P_i be the i-th subpattern of P and let P_i be of the form $(P_1 \text{ FILTER } C)$. Then $\tau(P_i, false, D, i)$ is defined as:

$ans_i(id, \overline{var}(P_i), D)$:- $ans_{2i}(id_1, \overline{var}(P_1), D), C$.
$\tau(P_1, false, D, 2i)$

Optional. Let P_i be the i-th subpattern of P and furthermore let P_i be $(P_1 \text{ OPT } P_2)$, then $\tau(P_i, false, D, i)$ is defined as:

$ans_{opt-i}(\overline{var}(P_1), D)$:- $ans_{2i}(Id_1, \overline{var}(P_1), D)$,
$ans_{2i+1}(Id_2, v_2(\overline{var}(P_2)), D)$,
$comp(x_1, v_2(x_1), z_1), \ldots, comp(x_n, v_2(x_n), z_n)$.

$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i}(Id_1, v_1(\overline{var}(P_1)), D)$,
$ans_{2i+1}(Id_2, v_2(\overline{var}(P_2)), D)$,
$comp(v_1(x_1), v_2(x_1), x_1), \ldots, comp(v_1(x_n), v_2(x_n), x_n)$.

$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i}(Id_1, \overline{var}(P_1), D)$,
$not\ ans_{opt-i}(\overline{var}(P_1), D)$,
$null(y_1), \ldots, null(y_m)$.

$\tau(P_1, false, D, 2i)$.
$\tau(P_2, false, D, 2i + 1)$.

Union. Let P_i be the i-th subpattern of P and let P_i be of the form $(P_1 \text{ UNION } P_2)$. Then $\tau(P_i, false, D, i)$ is defined as:

$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i}(Id_1, \overline{var}(P_1), D)$,
$null(x_1), \ldots null(x_n)$.
$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i+1}(Id_2, \overline{var}(P_2), D)$,
$null(y_1), \ldots null(y_m)$.
$\tau(P_1, false, D, 2i)$
$\tau(P_2, false, D, 2i + 1)$

Figure 5: Translation rules for SPARQL graph patterns.

General strategy of the translation. Analogously to [26, 28], our translation proceeds by recursively traversing the parse tree of a SPARQL 1.1 query and translating each subpattern into its respective Datalog$^\pm$ rules. Subpatterns of the parse tree are indexed. The root has index 1, the left child of the i-th node has index $2*i$, the right child has index $2*i+1$. During the translation, bindings of the

i-th subpattern are represented by the predicate ans_i. In all answer predicates ans_i, we have the current graph as last component. It can be changed by the GRAPH construct; for all other SPARQL constructs, it is transparently passed on from the children to the parent in the parse tree. Since the order of variables in predicates is relevant, some variable sets will need to be lexicographically ordered, which we denote by $\overline{x}$ as in [28]. We write $\overline{var}(P)$ to denote the lexicographically ordered tuple of variables of P. Moreover a variable renaming function $v_j : V \to V$ is defined.

Auxiliary Predicates. The translation generates several auxiliary predicates. Above all, we need a predicate comp for testing if two mappings are *compatible*. The notion of compatible mappings is fundamental for the evaluation of SPARQL graph patterns. Two mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, if for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it is satisfied that $\mu_1(?X) = \mu_2(?X)$. The auxiliary predicate $comp(X_1, X_2, X_3)$ checks if two values X_1 and X_2 are compatible. The third position X_3 represents the value that is used in the result tuple when joining over X_1 and X_2:

```
null("null").
comp(X,X,X) :- term(X).
comp(X,Z,X) :- term(X), null(Z).
comp(Z,X,X) :- term(X), null(Z).
comp(Z,Z,Z) :- null(Z).
```

Bag semantics. For bag semantics, (i.e., $dst = false$) all answer predicates contain a fresh existential variable when they occur in the head of a rule. In this way, whenever such a rule fires, a fresh tuple ID is generated. This is particularly important for the translation of the UNION construct. In contrast to [28], we can thus distinguish duplicates without the need to increase the arity of the answer predicate. We have developed a novel duplicate preservation model based on the abstract theories of ID generation of [12]. As mentioned above, plain existential ID generation turned out to be problematic due to peculiarities of the Vadalog system. Therefore, our ID generation process is abstracted away by using a Skolem function generator and representing nulls (that correspond to tuple IDs) as specific Skolem terms.

Filter constraints. Note how we treat filter conditions in FILTER constructs: building our translation engine on top of the Vadalog system allows us to literally copy (possibly complex) filter conditions into the rule body and let the Vadalog system evaluate them. For instance, the regex functionality uses the corresponding Vadalog function, which makes direct use of the Java regex library. For evaluating filter functions isIRI, isURI, isBlank, isLiteral, isNumeric, and bound expressions, our translation engine uses the corresponding auxiliary predicates generated in our data translation method.

5.2 Translation of Property Paths

Property paths are an important feature, introduced in SPARQL 1.1. A translation of property paths to Datalog was presented in [28] – but not fully compliant with the SPARQL 1.1 standard: the main problem in [28] was the way how *zero-or-one* and *zero-or-more* property paths were handled. In particular, the case that a path of zero length from t to t also exists for those terms t which occur in the query but not in the current graph, was omitted in [28]. A *property path pattern* is given in the form s, p, o, where s, o are the usual subject and object and p is a *property path expression*. That is, p is either an IRI (the base case) or composed from one or two other property path expressions p_1, p_2 as: $\hat{}p_1$ (inverse path expression), $p_1 \mid p_2$ (alternative path expression), p_1/p_2 (sequence path expression), $p_1?$ (zero-or-one path expression), p_1+ (one-or-more path expression), p_1* (zero-or-more path expression), or $!p_1$ (negated path expression). A property path pattern s, p, o is translated by first translating the property path expression p into rules for each subexpression of p. The endpoints s and o of the overall path are only applied to the top-level expression p. Analogously to our translation function $\tau(P, dst, D, NodeIndex)$ for graph patterns, we now also introduce a translation function $\tau_{PP}(PP, dst, S, O, D, NodeIndex)$ for property path expressions PP, where S, O, are the subject and object of the top-level property path expression that have to be kept track of during the entire evaluation as will become clear when we highlight our translation in Figure 6.

Property path. Let P_i be the i-th subpattern of P and let P_i be a property path pattern of the form (S, P_1, O) where P_1 is a property path expression. Then $\tau(P_i, false, D, i)$ is defined as:

$ans_i(Id, \overline{var}(P_i), D)$:- $ans_{2i}(Id_1, S, O, D)$.
$\tau_{PP}(P_1, false, S, O, D, 2i)$.

Link property path. Let PP_i be the i-th subexpression of a property path expression PP and let $PP_i = p_1$ be a link property path expression. Then $\tau_{PP}(PP_i, false, S, O, D, i)$ is defined as:

$ans_i(Id, X, Y, D)$:- $triple(X, p1, Y, D)$.

One-or-more path. Let PP_i be the i-th subexpression of a property path PP and let $PP_i = PP_1+$ be a one-or-more property path expression. Then $\tau_{PP}(PP_i, false, S, O, D, i)$ is defined as:

Zero-or-one path. Let PP_i be the i-th subexpression of a property path expression PP and let $PP_i = PP_1?$ be a zero-or-one property path expression. Then $\tau_{PP}(PP_i, false, S, O, D, i)$ is defined as:

$ans_i(Id, X, X, D)$:- $subjectOrObject(X), Id = []$.
$ans_i(Id, X, Y, D)$:- $ans_{2i}(Id_1, X, Y, D), Id = []$.
$\tau_{PP}(PP_1, false, S, O, D, 2i)$

Figure 6: Translation rules for SPARQL property paths.

Again we restrict ourselves to the more interesting case of bag semantics. The translation of a property path pattern S, P_1, O for some property path expression P_1 consists of two parts: the translation of P_1 by the translation function τ_{PP} and the translation τ of S, P_1, O – now applying the endpoints S and O to the top-level property path expression P_1. The base case of τ_{PP} is a link property path $PP_i = p_1$ (i.e., simply an IRI), which returns all pairs (X, Y) that occur as subject and object in a triple with predicate p_1. Equally simple translations apply to inverse paths (which swap start point and end point), alternative paths (which are treated similarly to UNION in Figure 5), and sequence paths (which combine two paths by identifying the end point of the first path with the start point of the second path).

For zero-or-one paths (and likewise for zero-or-more paths), we need to collect all terms that occur as subjects or objects in the current graph by an auxiliary predicate subjectOrObject:

$subjectOrObject(X)$:- $triple(X, P, Y, D)$.
$subjectOrObject(Y)$:- $triple(X, P, Y, D)$.

This is needed to produce paths of length zero (i.e., from X to X) for all these terms occurring in the current graph. Moreover, if exactly one of S and O is not a variable or if both are the same non-variable, then also for these nodes we have to produce paths of zero length. It is because of this special treatment of zero-length paths that subject S and object O from the top-level property path expression have to be propagated through all recursive calls of the translation function τ_{PP}. In addition to the zero-length paths, of course, also paths of length one have to be produced by recursively applying the translation τ_{PP} to PP_1 if PP_i is of the form $PP_i = PP_1?$. Finally, one-or-more paths are realised in the usual style of transitive closure programs in Datalog.

It should be noted that, according to the SPARQL semantics of property paths[10], zero-or-one, zero-or-more, and one-or-more property paths always have set semantics. This is why the Datalog$^\pm$ rules for these three path expressions contain a body literal $Id = []$. By forcing the tuple ID to the same value whenever one of these rules fires, multiply derived tuples are indistinguishable for our system and will, therefore, never give rise to duplicates.

5.3 Correctness of our Translation

To ensure the correctness of our translation, we have applied a two-way strategy – consisting of an extensive empirical evaluation and a formal analysis. For the empirical evaluation, we have run SparqLog as well as Fuseki and Virtuoso on several benchmarks, which provide a good coverage of SPARQL 1.1. The results are summarized in Section 6.2. In a nutshell, SparqLog and Fuseki turn out to fully comply with the SPARQL 1.1 standard, while Virtuoso shows deviations from the standard on quite some queries.

For the formal analysis, we juxtapose our translation with the formal semantics of the various language constructs of SPARQL 1.1. Below we briefly outline our proof strategy: Following [2, 6, 28] for SPARQL graph patterns and [24, 28] for property path expressions, we first of all provide a formal definition of the semantics of the various SPARQL 1.1 features.[11]Given a SPARQL graph pattern P and a graph D, we write $[\![P]\!]_D$ to denote the result of evaluating P over D. The semantics $[\![PP]\!]_{D,s,o}$ of property path expressions PP is defined in a similar way, but now also taking the top level start and end points s, o of the property path into account.

Both $[\![P]\!]_D$ and $[\![PP]\!]_{D,s,o}$ are defined inductively on the structure of the expression P or PP, respectively, with triple patterns $P = (s, p, o)$ and link property paths $PP = p_1$ as base cases. For instance, for a join pattern $P_i = (P_1 \,.\, P_2)$ and optional pattern $P_j = (P_1 \text{ OPT } P_2)$, the semantics is defined as follows:

$$
\begin{aligned}
[\![P_i]\!]_D &= \{\!\{\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_D \text{ and } \mu_2 \in [\![P_2]\!]_D \text{ and } \mu_1 \sim \mu_2\}\!\} \\
[\![P_j]\!]_D &= \{\!\{\mu \mid \mu \in [\![P_1 \,.\, P_2]\!]_D \text{ and } \mu \models C\}\!\} \cup \\
&\qquad \{\!\{\mu_1 \mid \mu_1 \in [\![P_1]\!]_D \text{ and for all } \mu_2 \in [\![P_2]\!]_D\colon \mu_1 \nsim \mu_2\}\!\}
\end{aligned}
$$

Now consider the Datalog$^\pm$ rules generated by our translation. For a join pattern, the two body atoms $ans_{2i}(Id_1, v_1(\overline{var}(P_1)), D)$ and $ans_{2i+1}(Id_2, v_2(\overline{var}(P_2)), D)$ yield, by induction, the sets of mappings $[\![P_1]\!]_D$ and $[\![P_2]\!]_D$. The variable renamings v_1 and v_2 make sure that there is no interference between the evaluation of $[\![P_1]\!]_D$ (by the first body atom) and the evaluation of $[\![P_2]\!]_D$ (by the second body atom). The *comp*-atoms in the body of the rule make sure that μ_1 and μ_2 are compatible on all common variables. Moreover, they bind the common variables $\{x_1, \dots, x_n\}$ to the correct value according to the definition of the *comp*-predicate.

The result of evaluating an optional pattern consists of two kinds of mappings: (1) the mappings μ in $[\![(P_1 \,.\, P_2)]\!]_D$ and (2) the mappings μ_1 in $[\![P_1]\!]_D$ which are not compatible with any mapping μ_2 in $[\![P_2]\!]_D$. Analogously to join patterns, the second rule generated by our translation produces the mappings of type (1). The first rule generated by our translation computes those mappings in $[\![P_1]\!]_D$ which are compatible with some mapping in $[\![P_2]\!]_D$. Hence, the third rule produces the mappings of type (2). Here the negated second body literal removes all those mappings from $[\![P_1]\!]_D$ which are compatible with some mapping in $[\![P_2]\!]_D$.

Full details of the semantics definitions $[\![P]\!]_D$ and $[\![PP]\!]_{D,s,o}$ and of the juxtaposition with the rules generated by our translation are provided in the full version of this paper [1].

6 EXPERIMENTAL EVALUATION

In this section, we report on the experimental evaluation of the SparqLog system. We want to give a general understanding of the behaviour of SparqLog in the following three areas: (1) we first analyse various benchmarks available in the area to identify **coverage** of SPARQL features and which benchmarks to use subsequently in our evaluation, (2) we analyse the **compliance** of our system with the SPARQL standard using the identified benchmarks, and set this in context with the two state-of-the-art systems Virtuoso and Fuseki, and, finally, (3) we evaluate the **performance** of query execution of SparqLog and compare it with state-of-the-art systems for SPARQL query answering and reasoning over ontologies, respectively. We thus put particular emphasis on property paths and their combination with ontological reasoning. Further details on our experimental evaluation – in particular, how we set up the analysis of different benchmarks and of the standard-compliance of various systems – are provided in the supplementary material.

6.1 Benchmark Analysis

In this subsection, we analyse current state-of-the-art benchmarks for SPARQL engines. Table 2 is based on the analysis of [32] and represents the result of our exploration of the SPARQL feature coverage of the considered benchmarks. Furthermore, it was adjusted and extended with additional features by us. Particularly heavily used SPARQL features are marked in blue, while missing SPARQL features are marked in orange. The abbreviations of the columns represent the following SPARQL features: DIST[INCT], FILT[ER], REG[EX], OPT[IONAL], UN[ION], GRA[PH], P[roperty Path] Seq[uential], P[roperty Path] Alt[ernative], GRO[UP BY]. Note that, in Table 2, we do not display explicitly basic features, such as *Join*, *Basic Graph pattern*, etc., since these are of course covered by every benchmark considered here. Morerover, we have not included the SPARQL features *MINUS* and the *inverted*, *zero-or-one*, *zero-or-more*, *one-or-more*, and *negated property path* in Table 2, as none of the selected benchmarks covers any of these SPARQL features.

[10]https://www.w3.org/TR/SPARQL11-query/#defn_PropertyPathExpr (last visited 09/25/2023)

[11]We note that the semantics definitions in all of these sources either only cover a rather small subset of SPARQL 1.1 or contain erroneous definitions. The most complete exposition is given in [28] with some inaccuracies in the treatment of Optional Filter patterns and of zero-length property paths.

Table 2: Feature Coverage of SPARQL Benchmarks [32]

	Benchmark	DIST	FILT	REG	OPT	UN	GRA	PSeq	PAlt	GRO
Synthetic	Bowlogna	5.9	41.2	11.8	0.0	0.0	0.0	0.0	0.0	76.5
	TrainBench	0.0	41.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	BSBM	25.0	37.5	0.0	54.2	8.3	0.0	0.0	0.0	0.0
	SP2Bench	35.3	58.8	0.0	17.6	17.6	0.0	0.0	0.0	0.0
	WatDiv	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	SNB-BI	0.0	66.7	0.0	45.8	20.8	0.0	16.7	0.0	100.0
	SNB-INT	0.0	47.4	0.0	31.6	15.8	0.0	5.3	10.5	42.1
Real	FEASIBLE (D)	56.0	58.0	14.0	28.0	40.0	0.0	0.0	0.0	0.0
	FEASIBLE (S)	56.0	27.0	9.0	32.0	34.0	10.0	0.0	0.0	25.0
	Fishmark	0.0	0.0	0.0	9.1	0.0	0.0	0.0	0.0	0.0
	DBPSB	100.0	44.0	4.0	32.0	36.0	0.0	0.0	0.0	0.0
	BioBench	39.3	32.1	14.3	10.7	17.9	0.0	0.0	0.0	10.7

Table 2 reveals that no benchmark covers all SPARQL features. Even more, SNB-BI and SNB-INT are the only benchmarks that contain property paths. Yet, they cover merely the *sequential* (PSeq) and *alternative property path* (PAlt), which in principle correspond to the *JOIN* and *UNION* operator. This means that no existing benchmark covers recursive property paths (though we will talk about the benchmark generator gMark [7] later), which are one of the most significant extensions provided by SPARQL 1.1. Our analysis of SPARQL benchmarks leads us to the following conclusions for testing the compliance with the SPARQL standard and for planning the performance tests with SparqLog and state-of-the-art systems.

Evaluating compliance with the SPARQL standard. Based on the results of Table 2, we have chosen the following three benchmarks to evaluate the compliance of our SparqLog system with the SPARQL standard: (1) We have identified *FEASIBLE (S)* [31] as the real-world benchmark of choice, as it produces the most diverse test cases [32] and covers the highest amount of features; (2) *SP2Bench* [33] is identified as the synthetic benchmark of choice, since it produces synthetic datasets with the most realistic characteristics [32]; (3) finally, since no benchmark that employs real-world settings provides satisfactory coverage of property paths, we have additionally chosen *BeSEPPI* [34] – a simplistic, yet very extensive benchmark specifically designed for testing the correct and complete processing of property paths. We report on the results of testing the compliance of our SparqLog system as well as Fuseki and Virtuoso in Section 6.2.

Performance benchmarking. For the empirical evaluation of query execution times reported in Section 6.3, we have identified SP2Bench as the most suitable benchmark, as it contains handcrafted queries that were specifically designed to target query optimization. Since none of the existing benchmarks for SPARQL performance measurements contains recursive property paths, we have included instances generated by the benchmark generator gMark [7], and report extensive results of this important aspect. In order to include in our tests also the performance measurements for the combination of property paths with ontologies, we have further extended the SP2Bench with an ontology containing subPropertyOf and subClassOf statements.

6.2 SPARQL Compliance

As discussed in the previous section, we have identified three benchmarks (FEASIBLE(S), SP2Bench, BeSEPPI) for the evaluation of the standard compliance of our SparqLog system and two state-of-the-art SPARQL engines. More details on the compliance evaluation as well as some challenges encountered by this evaluation (such as the comparison of results in the presence of null nodes) are discussed in the supplementary material. Below, we summarize the results:

The FEASIBLE(S) benchmark contains 77 queries that we used for testing the standard-conformant behaviour. It turned out that both SparqLog and Fuseki fully comply to the standard on each of the 77 queries, whereas Virtuoso does not. More specifically, for 14 queries, Virtuoso returned an erroneous result by either wrongly outputting duplicates (e.g., ignoring DISTINCTs) or omitting duplicates (e.g., by handling UNIONs incorrectly). Moreover, in 18 cases, Virtuoso was unable to evaluate the query and produced an error.

The SP2Bench benchmark contains 17 queries, specifically designed to test the scalability of SPARQL engines. All 3 considered systems produce the correct result for all 17 queries.

The BeSEPPI benchmark contains 236 queries, specifically designed to evaluate the correct and complete support of property path features. Table 3 shows the detailed results of the experimental evaluation of the 3 considered systems on this benchmark. We distinguish 4 types of erroneous behaviour: correct but incomplete results (i.e., the mappings returned are correct but there are further correct mappings missing), complete but incorrect (i.e., no correct mapping is missing but the answer falsely contains additional mappings), incomplete and incorrect, or failing to evaluate the query and returning an error instead. The entries in the table indicate the number of cases for each of the error types. We see that Fuseki and SparqLog produce the correct result in all 236 cases. Virtuoso only handles the queries with inverse, sequence and negated path expressions 100% correctly. For queries containing alternative, zero-or-one, one-or-more, or zero-or-more path expressions, Virtuoso is not guaranteed to produce the correct result. The precise number of queries handled erroneously is shown in the cells marked red.

Table 3: Compliance Test Results with BeSEPPI

Stores	Virtuoso				Jena Fuseki				Our Solution				
Expressions	Incomp. & Correct	Complete & Incor.	Incomp. & Incor.	Error	Incomp. & Correct	Complete & Incor.	Incomp. & Incor.	Error	Incomp. & Correct	Complete & Incor.	Incomp. & Incor.	Error	Total #Queries
Inverse	0	0	0	0	0	0	0	0	0	0	0	0	20
Sequence	0	0	0	0	0	0	0	0	0	0	0	0	24
Alternative	3	0	0	0	0	0	0	0	0	0	0	0	23
Zero or One	0	0	0	3	0	0	0	0	0	0	0	0	24
One or More	10	0	0	8	0	0	0	0	0	0	0	0	34
Zero or More	0	0	0	7	0	0	0	0	0	0	0	0	38
Negated	0	0	0	0	0	0	0	0	0	0	0	0	73
Total	13	0	0	18	0	0	0	0	0	0	0	0	236

To conclude, while SparqLog and Fuseki handle all considered queries from the 3 chosen benchmarks correctly, Virtuoso produces a significant number of errors.

6.3 Performance Measurements

Experimental Setup. Our benchmarks were executed on a system running openSUSE Leap 15.2 with dual Intel(R) Xeon(R) Silver 4314

16 core CPUs, clocked at 3.4 GHz, with 512GB RAM of which 256GB reserved for the system under test, and 256GB for the operating system. For each system we set a time-out of 900s. We start each benchmark by repeating the same warm-up queries 5 times and by 5 times loading and deleting the graph instance. Furthermore, we did 5 repetitions of each query (each time deleting and reloading the dataset). For our experiments we use Apache Jena Fuseki 3.15.0, Virtuoso Open Source Edition 7.2.5, and Stardog 7.7.1. Vadalog loads and queries the database simultaneously. Hence, to perform a fair comparison with competing systems, we compare their total loading and querying time to the total time that SparqLog needs to answer the query. Since, loading includes index building and many more activities, we delete and reload the database each time, when we run a query (independent of warm-up or benchmark queries).

Performance on general SPARQL queries. SP2Bench is a benchmark that particularly targets query optimization and computation-intensive queries. We have visualized the result in Figure 7 and found that SparqLog reaches highly competitive performance with Virtuoso and significantly outperforms Fuseki on most queries.

gMark. Since current SPARQL benchmarks provide only rudimentary coverage of property path expressions, we have evaluated SparqLog, Fuseki, and Virtuoso using the gMark benchmark generator [7], a domain- and language-independent graph instance and query workload generator which specifically focuses on path queries, i.e., queries over property paths. We have evaluated SparqLog's, Fuseki's, and Virtuoso's path query performance on the *test*[12] and *social*[13] demo scenarios. Each of these two demo scenarios provides 50 SPARQL queries and a graph instance. Further details on the benchmarks that we used for evaluating a system's query execution time and on the experimental results that we obtained are given in the full version of this paper [1]. In the following, we compare the results of the three systems on gMark:

Virtuoso could not (correctly) answer 48 of the in total 100 queries of the gMark Social and Test benchmark. Thus, it could not correctly answer almost half of the queries provided by both gMark benchmarks, which empirically reveals its dramatic limitations in answering complex property path queries. In 20 of these 48 cases, Virtuoso returned an incomplete result. While in solely 3 incomplete result cases Virtuoso missed solely one tuple in the returned result multi-set, in the remaining 17 incomplete result cases; Virtuoso produces either the result tuple *null* or an empty result multi-set instead of the correct non-null/non-empty result multi-set. In the other 28 cases Virtuoso failed either due to a time-, mem-out or due to not supporting a property path with two variables. This exemplifies severe problems with handling property path queries.

Fuseki suffered on 37 of the in total 100 queries of the gMark Social and Test benchmark a time-out (i.e., took longer than 900s for answering the queries). Thus, it timed-out on more than a third of gMark queries, which empirically reveals its significant limitations in answering complex property path queries.

SparqLog managed to answer 98 of gMark's (in total 100) queries within less than 200s and timed out on solely 2 queries. The results on the gMark Social benchmark are shown in Figures 8; the results on the gMark Test benchmark are given in full version of this paper [1]. These results reveal the strong ability of our system in answering queries that contain complex property paths. Furthermore, each time when both Fuseki and SparqLog returned a result, the results were equal, even further empirically confirming the correctness of our system (i.e., that our system follows the SPARQL standard).

In conclusion, these three benchmarks show that SparqLog (1) is highly competitive with Virtuoso on regular queries with respect to query execution time, (2) follows the SPARQL standard much more accurately than Virtuoso and supports more property path queries than Virtuoso, and (3) dramatically outperforms Fuseki on query execution, while keeping its ability to follow the SPARQL standard accurately.

Ontological reasoning. One of the main advantages of our SparqLog system is that it provides a uniform and consistent framework for reasoning and querying Knowledge Graphs. We therefore wanted to measure the performance of query answering in the presence of an ontology. Since Fuseki and Virtuoso do not provide such support, we compare SparqLog with Stardog, which is a commonly accepted state-of-the-art system for reasoning and querying within the Semantic Web. Furthermore, we have created a benchmark based on SP2Bench's dataset that contains property path queries and ontological concepts such as subPropertyOf and subClassOf and provide this benchmark in the supplementary material.

Full details of these experiments are provided in the full version of this paper [1]. In summary, we note that SparqLog is faster than Stardog on most queries. Particularly interesting are queries 4 and 5, which contain recursive property path queries with two variables. Our engine needs on query 4 only about a fifth of the execution time of Stardog and it can even answer query 5, on which Stardog times outs (using a timeout of 900s). On the other queries, Stardog and SparqLog perform similarly.

To conclude, our new SparqLog system does not only follow the SPARQL standard, but it also shows good performance. Even though SparqLog is a full-fledged, general-purpose Knowledge Graph management system and neither a specialized SPARQL engine nor a specialized ontological reasoner, it is highly competitive to state-of-the-art SPARQL engines and reasoners and even outperforms them on answering property path queries and particularly hard cases.

7 CONCLUSION

In this work we have taken a step towards bringing SPARQL-based systems and Datalog$^\pm$-based systems closer together. In particular, we have provided (i) a uniform and fairly complete theoretical translation of SPARQL into Warded Datalog$^\pm$, (ii) a practical translation engine that covers most of the SPARQL 1.1 functionality, and (iii) an extensive experimental evaluation.

We note that the SparqLog engine can be seen in two ways: (1) as a stand-alone translation engine for SPARQL into Warded Datalog$^\pm$, and (2) as a full Knowledge Graph engine by using our translation engine together with the Vadalog system.

As next steps, we envisage of course 100% or close to 100% SPARQL coverage. Possibly more (scientifically) interestingly, we plan to expand on the finding that query plan optimization provides a huge effect on performance, and investigate SPARQL-specific query plan optimization in a unified SPARQL-Datalog$^\pm$ system.

[12] https://github.com/gbagan/gMark/tree/master/demo/test (last visited 09/25/2023)
[13] https://github.com/gbagan/gMark/tree/master/demo/social (last visited 09/25/2023)

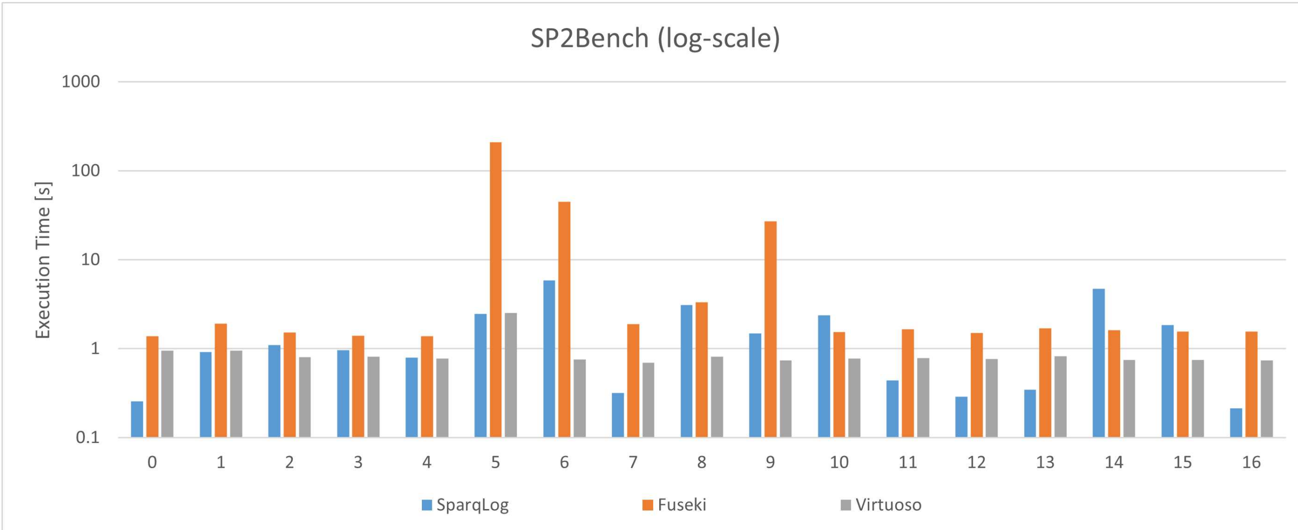

Figure 7: SP2Bench Benchmark (Log Scale)

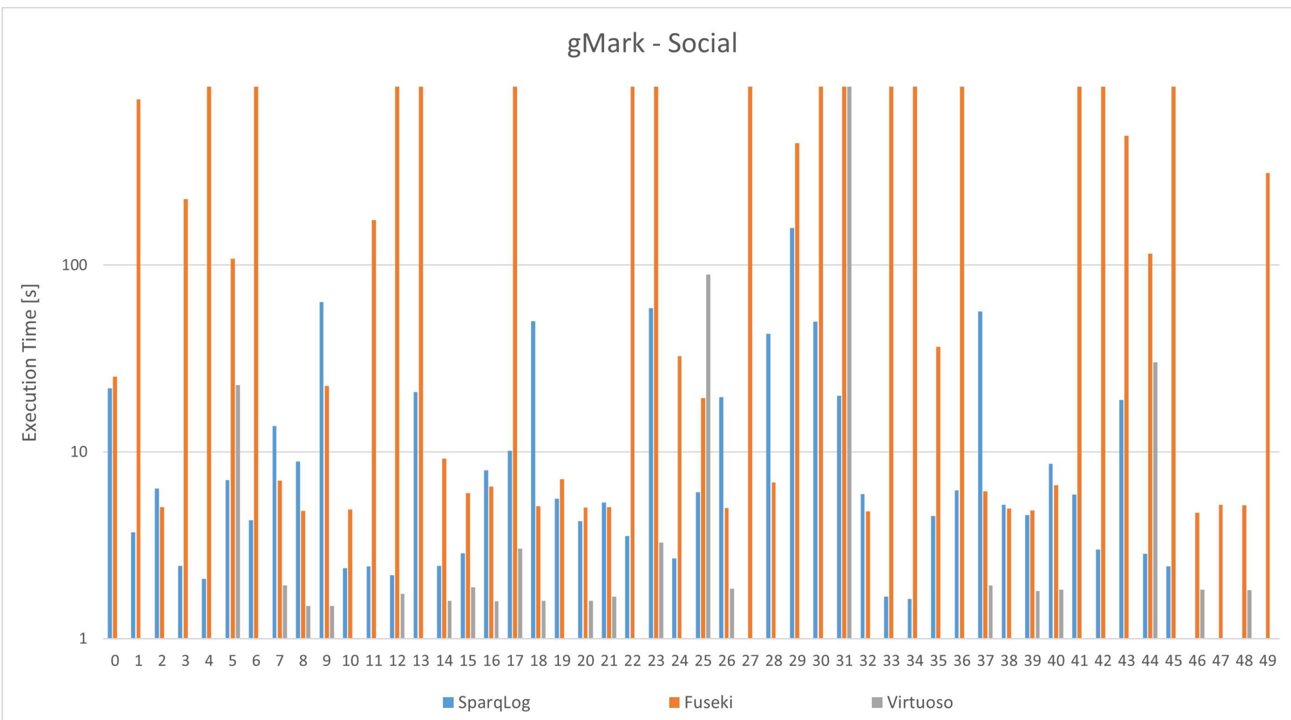

Figure 8: gMark Social Benchmark (Log Scale)

Finally, we note that work on a unified benchmark covering all or close to all of the SPARQL 1.1 features would be desirable. As observed in Section 6.1, no such benchmark currently exists.

ACKNOWLEDGMENTS

This work has been funded by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG18013, 10.47379/NXT22018, 10.47379/ICT2201]; the Christian Doppler Research Association (CDG) JRC LIVE; Renzo Angles was supported by ANID FONDECYT Chile through grant 1221727. Georg Gottlob is a Royal Society Research Professor and acknowledges support by the Royal Society in this role through the “RAISON DATA” project (Reference No. RP\R1\201074).

REFERENCES

[1] Renzo Angles, Georg Gottlob, Aleksandar Pavlović, Reinhard Pichler, and Emanuel Sallinger. 2023. SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog. *CoRR* abs/2307.06119 (2023). https://arxiv.org/abs/2307.06119

[2] Renzo Angles and Claudio Gutiérrez. 2008. The Expressive Power of SPARQL. In *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings (Lecture Notes in Computer Science)*, Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan (Eds.), Vol. 5318. Springer, Berlin, Heidelberg, 114–129. https://doi.org/10.1007/978-3-540-88564-1_8

[3] Renzo Angles and Claudio Gutiérrez. 2016. The Multiset Semantics of SPARQL Patterns. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil (Eds.), Vol. 9981. Springer International Publishing, Cham, 20–36. https://doi.org/10.1007/978-3-319-46523-4_2

[4] Renzo Angles and Claudio Gutiérrez. 2016. Negation in SPARQL. In *10th Alberto Mendelzon International Workshop on Foundations of Data Management (CEUR Workshop Proceedings)*, Vol. 1644. CEUR-WS.org, Aachen.

[5] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2018. Expressive Languages for Querying the Semantic Web. *ACM Trans. Database Syst.* 43 (2018), 13:1–13:45.

[6] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. 2009. On the Semantics of SPARQL. In *Semantic Web Information Management - A Model-Based Perspective*, Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca (Eds.). Springer, 281–307. https://doi.org/10.1007/978-3-642-04329-1_13

[7] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 63–64. https://doi.org/10.1109/ICDE.2017.38

[8] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings (Lecture Notes in Computer Science)*, Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman (Eds.), Vol. 9202. Springer International Publishing, Cham, 328–344. https://doi.org/10.1007/978-3-319-21542-6_21

[9] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2009. Extending Decidable Cases for Rules with Existential Variables. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, Craig Boutilier (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 677–682.

[10] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artif. Intell.* 175, 9-10 (2011), 1620–1654.

[11] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11 (2018), 975–987.

[12] Leopoldo E. Bertossi, Georg Gottlob, and Reinhard Pichler. 2019. Datalog: Bag Semantics via Set Semantics. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal (LIPIcs)*, Pablo Barceló and Marco Calautti (Eds.), Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern, Germany, 16:1–16:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.16

[13] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. An analytical study of large SPARQL query logs. *The VLDB Journal* 29 (2019), 655 – 679.

[14] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *J. Artif. Intell. Res.* 48 (2013), 115–174.

[15] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings (ACM International Conference Proceeding Series)*, Ronald Fagin (Ed.), Vol. 361. Association for Computing Machinery, New York, NY, USA, 14–30. https://doi.org/10.1145/1514894.1514897

[16] Andrea Calì, Georg Gottlob, and Andreas Pieris. 2010. Advanced Processing for Ontological Queries. *PVLDB* 3, 1 (2010), 554–565.

[17] Andrea Calì, Georg Gottlob, and Andreas Pieris. 2010. Query Answering under Non-guarded Rules in Datalog+/-. In *Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings (Lecture Notes in Computer Science)*, Pascal Hitzler and Thomas Lukasiewicz (Eds.), Vol. 6333. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-15918-3_1

[18] David Carral, Irina Dragoste, Larry González, Ceriel J. H. Jacobs, Markus Krötzsch, and Jacopo Urbani. 2019. VLog: A Rule Engine for Knowledge Graphs. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.), Vol. 11779. Springer International Publishing, Cham, 19–35. https://doi.org/10.1007/978-3-030-30796-7_2

[19] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax (W3C Recommendation). https://www.w3.org/TR/rdf11-concepts/.

[20] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. 2013. hex-Programs with Existential Quantification. In *Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Michael Hanus and Ricardo Rocha (Eds.), Vol. 8439. Springer International Publishing, Cham, 99–117. https://doi.org/10.1007/978-3-319-08909-6_7

[21] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124.

[22] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language (W3C Recommendation). https://www.w3.org/TR/sparql11-query/.

[23] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189.

[24] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoc. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.), Vol. 9366. Springer, 3–18. https://doi.org/10.1007/978-3-319-25007-6_1

[25] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.), Vol. 9367. Springer International Publishing, Cham, 3–20. https://doi.org/10.1007/978-3-319-25010-6_1

[26] Axel Polleres. 2007. From SPARQL to rules (and back). In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). Association for Computing Machinery, New York, NY, USA, 787–796. https://doi.org/10.1145/1242572.1242679

[27] Axel Polleres and Roman Schindlauer. 2007. DLVHEX-SPARQL: A SPARQL Compliant Query Engine Based on DLVHEX. In *Proceedings of the ICLP'07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, ALPSWS 2007, Porto, Portugal, September 13th, 2007 (CEUR Workshop Proceedings)*, Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus (Eds.), Vol. 287. CEUR-WS.org, Aachen.

[28] Axel Polleres and Johannes Peter Wallner. 2013. On the relation between SPARQL1.1 and Answer Set Programming. *Journal of Applied Non-Classical Logics* 23 (2013), 159–212.

[29] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF (W3C Recommendation). https://www.w3.org/TR/rdf-sparql-query/.

[30] Piotr Przymus, Aleksandra Boniewicz, Marta Burzanska, and Krzysztof Stencel. 2010. Recursive Query Facilities in Relational Databases: A Survey. In *Database Theory and Application, Bio-Science and Bio-Technology - International Conferences, DTA and BSBT 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings (Communications in Computer and Information Science)*, Yanchun Zhang, Alfredo Cuzzocrea, Jianhua Ma, Kyo-Il Chung, Tughrul Arslan, and Xiaofeng Song (Eds.), Vol. 118. Springer, 89–99. https://doi.org/10.1007/978-3-642-17622-7_10

[31] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.), Vol. 9366. Springer, 52–69. https://doi.org/10.1007/978-3-319-25007-6_4

[32] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2019. How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley,

Ricardo Baeza-Yates, and Leila Zia (Eds.). Association for Computing Machinery, New York, NY, USA, 1623–1633. https://doi.org/10.1145/3308558.3313556

[33] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. 2008. SP2Bench: A SPARQL Performance Benchmark. *CoRR* abs/0806.4627 (2008). arXiv:0806.4627 http://arxiv.org/abs/0806.4627

[34] Adrian Skubella, Daniel Janke, and Steffen Staab. 2019. BeSEPPI: Semantic-Based Benchmarking of Property Path Implementations. In *The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings (Lecture Notes in Computer Science)*, Pascal Hitzler, Miriam Fernández, Krzysztof Janowicz, Amrapali Zaveri, Alasdair J. G. Gray, Vanessa López, Armin Haller, and Karl Hammar (Eds.), Vol. 11503. Springer International Publishing, Cham, 475–490. https://doi.org/10.1007/978-3-030-21348-0_31

[35] Victor Vianu. 2021. Datalog Unchained. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo (Eds.). ACM, 57–69. https://doi.org/10.1145/3452021.3458815