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

Renzo Angles
Universidad de Talca, Chile

Georg Gottlob
University of Oxford, UK

Aleksandar Pavlović
TU Wien, Austria

Reinhard Pichler
TU Wien, Austria

Emanuel Sallinger
TU Wien, Austria

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 Database area , a different set of languages such as SPARQL and OWL is used in the Semantic Web area. Interoperability between such system is still a challenge. The key interoperability challenge we take on in this paper is how to bring together systems that allow full recursion (a typical Databases requirement) and existential quantification as required by OWL 2 (a typical SW requirement, taken up by DB-languages in the Datalog+/- family of languages) and at the same time allow querying via both Datalog with existential quantification (DB) and SPARQL (SW).

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, 14(1): XXX-XXX, 2022. doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at iURL_TO_YOUR_ARTIFACTS.

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. Below we propose several requirements that, based on our experience, play a prominent role in the efforts of the two communities.

Of major importance to the *Semantic Web* community is the compliance with the relevant W3C standards. We focus here on the widely-used SPARQL over OWL 2 QL ontologies, which has been a good yardstick of what can be reasonably achieved [4]:

[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.

[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] Existential Quantification. Existential quantification is necessary to perform ontological reasoning. In this paper, we require the support of existential quantification at least to the extent required by OWL 2 QL, a major Semantic Web standard.

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

[RQ4] Full Recursion. Full recursion is vital to model many characteristics of the real-world and it is the main feature of the relational query language Datalog. Also SQL-based relational database management systems have meanwhile incorporated recursion capabilities to increase their expressive power.

[RQ5] Query Independent Schema. This means that the result schema of a query (i.e., the relation representing the result) solely depends on the selected variables and not on the structure and operations of the query. In this respect, traditional database management systems are less flexible than Semantic Web approaches.

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

[RQ6] 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 Semantic Web and Database community, namely: both communities address different requirements when studying the foundations of Knowledge Graphs and developing Knowledge Graph management systems.

There have already 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 build a bridge between the Semantic Web and Database communities by developing one uniform

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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–RQ6. 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 [2, 24, 26]. 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.

The contribution of this paper can be viewed in two ways:

- SparqLog as a **stand-alone translation engine** for SPARQL into Warded Datalog$^\pm$: a fully independent system that can be used with any underlying engine that supports Datalog, and where required by the features used in SPARQL, at least existential quantification as defined by Warded Datalog$^\pm$.
- SparqLog as a full-fledged **Knowledge Graph engine** obtained by using our translation engine together with the engine Vadalog [8]: bringing the ability to express KG ontologies and to query KGs via both Warded Datalog$^\pm$ *and* SPARQL together. More specifically, Vadalog fulfills RQ3, RQ4, and RQ6 while our novel translation engine adds the functionality required by RQ1, RQ2 and RQ5.

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 translation from SPARQL to Warded Datalog$^\pm$ in Section 4, a more detailed look into the translation engine in Section 5, and the experimental evaluation of the SparqLog system in Section 6. A conclusion and an outlook to future work is given in Section 7. Further details on our theoretical translation as well as 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] of this submission. All resources will be made publicly available in case of acceptance (on Github and on arXiv, respectively), the only exception being the Vadalog system, which may not be made available for license reasons.

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

In this section, we outline several important theoretical research efforts that aimed at bridging the gap between the Database and Semantic Web community.

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 [24] 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 [26]. Thereby, Polleres and Wallner created a nearly complete translation of SPARQL 1.1 queries to Datalog with disjunction (DLV) programs. However, the chosen target language had two major drawbacks: On the one hand, 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 translation maintains duplicates by increasing the arity of result predicates for operations that can generate duplicates. This is however problematic, as the query response changes its schema in dependence of the structure of the query, conflicting with the query-independent schema requirement (RQ5) of the Database community. Also the requirement of an implemented system (RQ6) is only partially fulfilled, since the prototype implementation *DLVhex-SPARQL Plugin* [25] of the SPARQL to Datalog translation of [24] 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 [1]. The peculiarities of negation in SPARQL were treated in a separate paper [3]. 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 [2]. They translated a few SPARQL features into a Datalog dialect with bag semantics (multiset non-recursive Datalog with safe negation). This work fulfilled the query-independent schema requirement (RQ5) but considered only a small set of SPARQL functionality on a very abstract level and used again a target language

[1]https://github.com/joint-kg-labs/SparqLog

that does not support ontological reasoning, failing to meet important requirements (RQ1, RQ3) of the Semantic Web community. Most importantly, no implementation exists of the translations provided by Angles and Gutierrez, thus failing to fulfil RQ6.

Supporting Ontological Reasoning via Existential Rules. In [12], 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 [20]. Hence, numerous restrictions have been proposed [6, 7, 11, 13, 14, 18] to ensure decidability of such tasks, which led to the "–" in Datalog$^{\pm}$. Of all variants of Datalog$^{\pm}$, Warded Datalog$^{\pm}$ [4] ultimately turned out to constitute the best compromise between complexity and expressiveness in order to combine ontological reasoning with efficient query answering. Warded Datalog$^{\pm}$ has been implemented in an industrial-strength system – the Vadalog system [8], thus fulfilling requirement RQ6. However, the requirement of supporting SPARQL (RQ1) with or without bag semantics (RQ2) have not been fulfilled by Vadalog up to now.

Warded Datalog$^{\pm}$ with Bag Semantics. In [9], 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 (RQ6) by extending Vadalog were left open in [9] and considered of primary importance for future work.

2.2 Practical Approaches

In this section, we discuss several systems that aim at bridging the gap between Database technologies such as Datalog and Semantic Web technologies such as SPARQL. The World Wide Web Consortium (W3C) lists *StrixDB*, *DLVhex SPARQL-engine* and *RDFox* as systems that support 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. As mentioned above, the DLVhex-SPARQL Plugin [25] is a prototype implementation of the SPARQL to Datalog translation in [24]. 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 underlying logic programming language DLV provides only domain specific existential quantification (described in [17]), produced e.g. by hash-functions. Hence, it only provides very limited support of existential quantification, which 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 [23]. It reasons over OWL 2 RL ontologies in Datalog and computes/stores materialisations of the inferred consequences for efficient query answering [23]. 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 [23]. However, as that work does not mention any translation of SPARQL to Datalog at any point, we assume that Datalog is used solely for OWL 2 RL reasoning and that no high-level translation from SPARQL to Datalog was conducted. Moreover, RDFox does currently not support property paths and some other SPARQL 1.1 features[4].

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[5]. To the best of our knowledge, there is no academic paper or technical report that explains the capabilities of the system in greater detail – leaving us with the web page as the only source of information on StrixDB. The *StrixStore* documentation page[6] lists examples of how to integrate Datalog rules into SPARQL queries, to query graphs enhanced by Datalog ontologies. However, no translation from SPARQL to Datalog is mentioned at any of these documentation pages. Thus we have to assume that *StrixDB* does not translate SPARQL queries to Datalog rules and that it employs Datalog for OWL reasoning tasks only. Moreover, important SPARQL 1.1 features such as aggregation and property paths are not supported by StrixDB.

Graal. Graal was developed as a toolkit for querying ontologies with existential rules [5]. The system does not focus on a specific storage system, however specializes in algorithms that can answer queries regardless of the underlying database type [5]. 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 [5] and therefore supports merely a small subset of SPARQL features[7] — e.g. not even being able to express basic features such as *UNION* or *MINUS*. Clearly, the goal of developing a uniform and consistent framework for both, the Semantic Web and Database communities, cannot be achieved without supporting at least the most vital features of SPARQL (RQ1).

VLog. VLog is a rule engine, developed at the TU Dresden [15]. 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 [15]. 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 (RQ6).

The Vadalog system [8] 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

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

of Datalog, it supports full recursion. Although Warded Datalog$^{\pm}$ was shown to have the capabilities to support SPARQL 1.1 under the OWL 2 QL entailment regime [4] (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), SPARQL feature coverage (RQ1), and query independent schema (RQ5) requirements are not met.

3 PRELIMINARIES

3.1 RDF and SPARQL

RDF [16] 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 $(subject, predicate, object)$ where: 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 [19, 27] is the standard query language for RDF. In general terms, the input of a SPARQL query is an RDF dataset and the output can be a multiset of solution mappings (i.e. a table), an RDF graph or a Boolean value. In order to explain the components of a SPARQL query, we will use the example shown in Figure 1.

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), 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.

```
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.

E.g., the evaluation of the query in Figure 1 over the above RDF graph returns two mappings: a total assignment `?N` $\longrightarrow$ `"George"` and `?L` $\longrightarrow$ `"Lucas"` and a partial assignment: `?N` $\longrightarrow$ `"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 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 [12], Datalog$^{\pm}$ was presented as a family of languages that, on one hand, extend Datalog (whence the +) to increase its expressive power and, on the other hand, impose restrictions (whence the −) to ensure decidability or even tractability 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.

$\Pi \cup D$ can have many models. A canonical model is obtained via the *chase*, which is defined as follows: We say that a rule $\rho \in \Pi$ with head $p(\vec{z})$ is *applicable* to an instance I if there exists a homomorphism h from the body of ρ to I. We may then carry out a *chase step*, which consists in adding atom $h'(p(\vec{z}))$ to instance I, where h' coincides with h on all variables occurring in the body of ρ and h' maps each existential variable in $p(\vec{z})$ to a fresh *labelled null* not occurring in I. A *chase sequence* for database D and program Π is a sequence of instances $I_0, I_1, \ldots$ obtained by applying a sequence of chase steps, starting with $I_0 = D$. The union of instances obtained by all possible chase sequences is referred to as $Chase(D, \Pi)$. The labelled nulls in $Chase(D, \Pi)$ play the same role as blank nodes in an RDF graph, i.e., resources for which the concrete value is not known. The importance of $Chase(D, \Pi)$ comes from the equivalence $\Pi \cup D \models Q(\vec{a}) \Leftrightarrow Chase(D, \Pi) \models Q(\vec{a})$ [18]. Note however that, in general, $Chase(D, \Pi)$ is infinite. Hence, the previous equivalence does not yield an algorithm to evaluate a CQ $Q(\vec{z})$ w.r.t. database D and program Π. In fact, without restriction, this is an undecidable problem [20]. Several subclasses of Datalog$^{\pm}$ have thus been presented [4, 6, 7, 11, 13, 14, 18] that ensure decidability of CQ answering (see [11] for an overview).

One such subclass is *Warded* Datalog$^{\pm}$ [4], which makes CQ answering even tractable (data complexity). For a formal definition of *Warded* Datalog$^{\pm}$, see [4]. 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 [8]. Originally developed at the University of Oxford, it combines full support of Warded Datalog$^{\pm}$ plus a number of extensions needed for practical use, including (decidable) arithmetics and aggregation, and other features needed in practice. It has been deployed in numerous industrial scenarios, including at the Central Bank of Italy for applications such as detecting money laundering, detecting hostile company takeovers, assessing the (economic) impact of COVID-19, and many more. Besides the finance sector, it has been used in the supply chain and logistics sector, in the media intelligence sector, and many other areas of industry.

3.3 Datalog: From Bag Semantics to Set Semantics

In [21], a bag semantics of Datalog was introduced based on *derivation trees*. Given a database D and Datalog program Π, a *derivation tree* (DT) is a tree T with node and edge labels, such that either (1) T consists of a single node labelled by an atom from D or (2) Π contains a rule $\rho\colon\ H \leftarrow A_1, A_2, \ldots, A_k$ with $k > 0$, and there exist DTs $T_1, \ldots, T_k$ whose root nodes are labelled with atoms $C_1, \ldots, C_k$ such that $A_1, \ldots, A_k$ are simultaneously matched to $C_1, \ldots, C_k$ by applying some substitution θ, and T is obtained as follows: T has a new root node r with label $H\theta$ and the k root nodes of the DTs $T_1, \ldots, T_k$ are appended as child nodes of r in this order. All edges from r to its child nodes are labelled with ρ. Then the bag semantics of program Π over database D consists of all ground atoms derivable from D by Π, and the multiplicity $m \in \mathbb{N} \cup \{\infty\}$ of each such atom A is the number of possible DTs with root label A. Datalog with bag semantics is readily extended by stratified negation [22]: the second condition of the definition of DTs now has to take negative body atoms in a rule $\rho\colon H \leftarrow A_1, A_2, \ldots, A_k, \neg B_1, \ldots \neg B_\ell$ with $k > 0$ and $\ell \geq 0$ with head atom H from some stratum i into account in that we request that none of the atoms $B_1\theta, \ldots B_\ell\theta$ can be derived from D via the rules in Π from strata less than i.

In [9] it was shown how Datalog with *bag* semantics can be transformed into Warded Datalog$^{\pm}$ with *set* semantics. The idea is to replace every predicate $P(\ldots)$ by a new version $P(.\,;\ldots)$ with an extra, first argument to accommodate a labelled null which is interpreted as tuple ID (TID). Each rule in Π of the form

$$\rho\colon H(\bar{x}) \;\leftarrow\; A_1(\bar{x}_1), A_2(\bar{x}_2), \ldots, A_k(\bar{x}_k), \text{with } k > 0, \bar{x} \subseteq \cup_i \bar{x}_i$$

is then transformed into the Datalog$^{\pm}$ rule

$$\rho'\colon\ \exists z\, H(z;\bar{x}) \leftarrow A_1(z_1;\bar{x}_1), A_2(z_2;\bar{x}_2), \ldots, A_k(z_k;\bar{x}_k),$$

with fresh, distinct variables $z, z_1, \ldots, z_k$. Some care (introducing auxiliary predicates) is required for rules with negated body atoms so as not to produce unsafe negation. A Datalog rule $\rho\colon H(\bar{x}) \;\leftarrow\; A_1(\bar{x}_1), \ldots, A_k(\bar{x}_k), \neg B_1(\bar{x}_{k+1}), \ldots, \neg B_\ell(\bar{x}_{k+\ell})$ with, $\bar{x}_{k+1}, \ldots, \bar{x}_{k+\ell} \subseteq \bigcup_{i=1}^{k} \bar{x}_i$ is replaced by $\ell+1$ rules in the corresponding Datalog$^{\pm}$ program Π':

$$\begin{aligned}\rho'_0\colon \exists z H(z;\bar{x}) \leftarrow\ & A_1(z_1;\bar{x}_1), \ldots, A_k(z_k;\bar{x}_k),\\ & \neg Aux_1^{\rho}(\bar{x}_{k+1}), \ldots, \neg Aux_\ell^{\rho}(\bar{x}_{k+\ell}),\\ \rho'_i\colon Aux_i^{\rho}(\bar{x}_{k+i}) \leftarrow\ & B_i(z_i;\bar{x}_{k+i}),\ \ i = 1, \ldots, \ell.\end{aligned}$$

The resulting Datalog$^{\pm}$ program Π' is trivially warded since the rules thus produced contain no dangerous variables at all. Moreover, it is proved in [9] that an atom $P(\vec{a})$ is in the DT-defined bag semantics of Datalog program Π over database D with multiplicity $m \in \mathbb{N} \cup \{\infty\}$, iff $Chase(D, \Pi')$ contains atoms of the form $P(t;\vec{a})$ for m distinct labelled nulls t (i.e., the tuple IDs).

4 INTEGRATION OF SPARQL AND DATALOG$^{\pm}$

As discussed in Section 2, several theoretical works [1–3, 24, 26] have studied the translation of SPARQL into Datalog. However, to the best of our knowledge, this is the first work that describes and evaluates a practical translation from SPARQL 1.1 with bag semantics to Datalog. In this section, we summarize the status of the SPARQL features supported by our SparqLog system and describe some general principles of SparqLog. A more detailed dive into the translation engine is deferred to Section 5

4.1 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. [10], 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 [10] and its current implementation status in our SparqLog system. The table represents the real-world usage by a percentage value (drawn from [10]) in the feature usage field, if [10] covers the feature, "Unknown" if [10] 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 (except OFFSET, which is currently not translated as will be briefly explained below). 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 [10], 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.

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 those presented in [26]; we provide translations for both bag and set semantics, showcasing the respective translations 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 its dependence on a very specific chase algorithm 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 our SparqLog system. Among these features are CONSTRUCT, DESCRIBE, OFFSET and FILTER NOT EXISTS. We do not support CONSTRUCT and DESCRIBE so far, as these solution modifiers do not yield any interesting theoretical challenges and they did not occur in any of the benchmarks chosen for our experimental evaluation. OFFSET is currently not supported by Vadalog; thus a translation attempt from SPARQL to Datalog$^{\pm}$ at the moment would introduce unnecessary complexity that can be easily avoided in the future when Vadalog itself gets extended. The features for query federation are placed out of considered 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, according to [10].

4.2 SPARQL to Datalog$^{\pm}$ Translation

SparqLog allows to run SPARQL queries as Datalog$^{\pm}$ programs. To this end, SparqLog includes a translation engine with three methods:

- a *data translation method* T_D which generates Datalog$^{\pm}$ rules from an RDF Dataset;
- a *query translation* method T_Q which generates Datalog$^{\pm}$ rules from a SPARQL query;
- 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 transforms the resulting Datalog$^{\pm}$ solution into a SPARQL solution by using T_S.

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		✓
	Arithmetic Comparison		✓
	bound, isIRI, isBlank, isLiteral	All Constraints 40.15%	✓
	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	✗

Table 1: Selected SPARQL features, including their real-world usage according to [10] and the current status in SparqLog.

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`.

Data translation method. Consider the RDF graph G presented in Section 3.1. First, the data translation method 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 G, T_D generates a fact of the form `triple(s,p,o,g)` where g is the IRI of 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").
```

Query translation method. This is the most elaborate part of SparqLog. It produces rules for each language construct of SPARQL 1.1 that is supported by SparqLog plus rules defining several auxiliary predicates (such as compatibility of terms when joining two triples). In addition, also system instructions (e.g., to indicate the answer predicate or ordering requirements) are generated. The query translation method T_Q begins with the `WHERE` clause, then continues with the `SELECT` clause, and finalizes with the `ORDER BY` clause. Below we highlight some aspects of T_Q. A more detailed look into the query translation will be provided in Section 5.

General strategy of the translation. Analogously to [24, 26], also 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.

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).
```

Skolem functions to realise bag semantics. We have developed a novel duplicate preservation model based on the abstract theories of ID generation of [9]. 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.

Handling of filter constraints. We note that some complication with filter constraints may arise from bag semantics: For instance, let P_1, P_2 be graph patterns, and C_1, C_2 be filter constraints. Then the equivalences $\{P_1 \text{ FILTER } C_1 \wedge C_2\} \equiv \{\{P_1 \text{ FILTER } C_1\} \text{ FILTER } C_2\}$ and $\{P_1 \text{ FILTER } C_1 \vee C_2\} \equiv \{P_1 \text{ FILTER } C_1\} \text{ UNION} \{P_1 \text{ FILTER } C_2\}$ hold for *set* semantics. However, the latter equivalence breaks in case of *bag* semantics. The problem arises from the duplication of cardinalities when a solution mapping satisfies both conditions C_1 and C_2. Although there exist equivalences that work for bag semantics, the translations are large expressions with a costly evaluation in many cases as can be seen in [26]. To overcome this problem and in contrast to [26], SparqLog does not translate complex filter constraints. Instead, they are evaluated by using the internal evaluator of the underlying Vadalog system for Boolean conditions.

Example. We now illustrate our SPARQL-to-Datalog$^\pm$ translation by considering the SPARQL query Q from Figure 1. The resulting Datalog$^\pm$ rules are shown in Figure 2. Note that, in the interest of readability, we slightly simplify the presentation, e.g., by omitting language tags and type definitions, using simple (intuitive) variable names (rather than more complex ones as would be generated by SparqLog to rule out name clashes), etc.

Let the graph pattern defined by the `WHERE` clause be 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 27) is used to define the literal of the goal rule `ans`. It realises the projection defined by the `SELECT` clause. The `ORDER BY` clause is translated to the instruction `@post("ans","orderby(2)")` (line 26), which 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 1–3) by projecting out the `X` variable from the `ans1` relation, which contains the result of evaluating pattern P_1. As mentioned above, the tuple IDs are generated as Skolem terms (line 3 for `ans`; likewise lines 12, 17, 21, 25). 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 20, 24) 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 4–7) 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 8–12) 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 13–17) 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 16). The two basic graph patterns P_2 and P_3 are translated to rules for the predicates `ans2` (lines 18–21) and `ans3` (lines 22–25) in the obvious way.

Solution translation method. 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* of solutions mappings in this case. Its tabular representation is the following:

?N	?L
"George"	"Lucas"
"Steven"	

5 TRANSLATION ENGINE

We now provide more details on our translation of SPARQL to Warded Datalog$^\pm$. We thus first present some basic rules and we will then put particular emphasis on the most challenging part of

```
ans(ID, L, N, D) :-
   ans1(ID1, L, N, X, D),
   ID = ["f", L, N, X, ID1].
ans_opt1(N, X, D) :-
   ans2(ID2, N, X, D),
   ans3(ID3, V2_L, V2_X, D),
   comp(X, V2_X, X).
ans1(ID1, V2_L, N, X, D) :-
   ans2(ID2, N, X, D),
   ans3(ID3, V2_L, V2_X, D),
   comp(X, V2_X, X),
   ID1 = ["f1a", X, N, V2_X, V2_L, ID2, ID3].
ans1(ID1, L, N, X, D) :-
   ans2(ID2, N, X, D),
   not ans_opt1(N, X, D),
   null(L),
   ID1 = ["f1b", L, N, X, ID2].
ans2(ID2, N, X, D) :-
   triple(X, "<http://ex.org/name>",  N, D),
   D = "default",
   ID2 = ["f2", X, "<http://ex.org/name>",  N, D].
ans3(ID3, L, X, D) :-
   triple(X, "<http://ex.org/lastname>",  L, D),
   D = "default",
   ID3 = ["f3", X, "<http://ex.org/lastname>",  L, D].
@post("ans", "orderby(2)").
@output("ans").
```

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

the translation, which has not been handled properly by previous work, namely property paths.

5.1 Basic Translation Rules

We introduce some useful notation first: 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 [26]. We write $\overline{var}(P)$ to denote the lexicographically ordered tuple of variables of P. Moreover a renaming function $v_j : V \longrightarrow V$ is defined.

Recall the assignment of indices to the subpatterns of a graph described in Section 4.2. Then, for a given graph pattern P and a given dataset $D = \langle G, G_{named} \rangle$ (where G is the default graph and G_{named} is the set of named graphs) the translation function $\tau(P, dst, D, NodeIndex)$ is defined as follows:

(1) P is the graph pattern that should be translated next.
(2) dst (short for "distinct") is a Boolean value that describes whether the result should have set semantics ($dst = True$) or bag semantics ($dst = False$).
(3) D is the graph on which the pattern should be evaluated.
(4) $NodeIndex$ is the index of pattern P to be translated.

In the sequel, we concentrate on bag semantics (i.e., $dst = False$), since this is the more complex case. In Figure 3, we showcase the translation for some basic SPARQL constructs. The handling of property paths will be dealt with separately in Section 5.2. The complete translation – covering all SPARQL 1.1 features omitted here (such as the OPTIONAL and MINUS operator) as well as set semantics – is given in Appendix A.

To improve readability, we apply the simplified notation used in Figure 2 now also to Figure 3. 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.

We briefly highlight some interesting aspects of the translation in Figure 3. In all answer predicates, 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 in the parse tree. Moreover, 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 [26], we can thus distinguish duplicates without the need to increase the arity of the answer predicate. Finally, note how we treat filter conditions in FILTER or OPTIONAL 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 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 (Section 4.2). The regex functionality uses the corresponding Vadalog function, which makes direct use of the Java regex library.

5.2 Translation of Property Paths

Property paths are an important feature that was introduced in SPARQL 1.1. A translation of property paths to Datalog was presented in [26] – but not fully compliant with the SPARQL 1.1 standard: the main problem in [26] 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 [26]. Another challenge arises from the fact that property paths can of course introduce duplicates. In contrast to the translation in [26], we avoid the increase of the arity of the answer predicate by the use of existential variables in the head of Datalog$^{\pm}$ rules.

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: $^\wedge p_1$ (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). The challenge of the translation of a property path pattern s, p, o is that first the property path expression p has to be translated into rules for each subexpression of p, using auxiliary variables for intermediate nodes along a path. 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 SPARQL patterns, we now also introduce a translation function $\tau_{PP}(P, dst, S, O, D, NodeIndex)$ for property path expressions, 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 of property paths to Datalog$^{\pm}$ in Figure 4.

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

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

Graph. Let P_i be the i-th subpattern of P and let P_i be of the form (GRAPH g P_1), then $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

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

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

Here (and also for Optional Filter further below), we are using the following notation:

- $\overline{var}(P_i) = \overline{var(P_1) \cup var(P_2)}$
- $\{x_1, \ldots, x_n\} = var(P_1) \cap var(P_2)$
- $v_1, v_2 : var(P_1) \cap var(P_2) \to V$, 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 FILTER C). Then $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

Optional Filter. Let P_i be the i-th subpattern of P of the form $P_i =$ (P_1 OPT(P_2 FILTER C)). Then $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

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

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

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

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

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

Here, we are using the following notation:

- $\{x_1, \ldots, x_n\} = var(P_2) \setminus var(P_1)$
- $\{y_1, \ldots, y_m\} = var(P_1) \setminus var(P_2)$

Figure 3: Translation of some basic SPARQL constructs.

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 $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

Link property path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ \ p_1\ \ Y$ be a link property path pattern, then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

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

Inverse path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ \ \hat{}P_1\ \ Y$ be an inverse property path pattern, then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$ans_i(Id, X, Y, D)$:- $ans_{2i}(Id_1, Y, X, D)$.
$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$

Alternative path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ \ (P_1|P_2)\ \ Y$ be a alternative property path expression. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$ans_i(Id, X, Y, D)$:- $ans_{2i}(Id_1, X, Y, D)$.
$ans_i(Id, X, Y, D)$:- $ans_{2i+1}(Id_1, X, Y, D)$.
$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$
$\tau_{PP}(P_2, \mathit{false}, S, O, D, 2i+1)$

Sequence path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ \ P_1/P_2\ \ Y$ be a sequence property path pattern. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$ans_i(Id, W, Z, D)$:- $ans_{2i}(Id_1, W, X, D)$,
$ans_{2i+1}(Id_2, Y, Z, D), comp(X, Y, Y)$.
$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$
$\tau_{PP}(P_2, \mathit{false}, S, O, D, 2i+1)$

Zero-or-one path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ \ P_1?\ \ Y$ be a zero-or-one property path pattern. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$ans_i(Id, X, X, D)$:- $subjectOrObject(X), Id = []$.
$ans_i(Id, X, Y, D)$:- $ans_{2i}(Id_1, X, Y, D), Id = []$.
$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$

Moreover, if either one of S and O is a variable and the other is a non-variable t or both S and O are the same non-variable t, then the following rule is added:

$ans_i(Id, X, X, D)$:- $not\ \ Term(X), X = t, Id = []$.

One-or-more path. Let P_i be the i-th subpattern of a property path PP and let $P_i = X\ P_1+\ \ Y$ be a one-or-more property path pattern. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$ans_i(Id, X, Y, D)$:- $ans_{2i}(Id_1, X, Y, D), Id = []$.
$ans_i(Id, X, Z, D)$:- $ans_{2i}(Id_1, X, Y, D), ans_i(Id_2, Y, Z, D), Id = []$.
$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$

Figure 4: Translation of SPARQL property paths.

Again we restrict ourselves to the more interesting case of bag semantics (i.e., $dst = False$). 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 the translation function τ_{PP} is a link property path $P_i = X \;\; p_1 \;\; Y$, which returns all pairs X, Y that occur as subject and object in a triple. Inverse paths are translated in the obvious way by swapping start point and end point. Alternative paths are treated similarly to UNION in Figure 3. Again, it is very convenient that we may use existential variables in the rule heads to keep apart duplicates. Sequence paths behave similarly to joins in Figure 3. That is, two paths are combined by joining the end points of the first path with the start points 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` defined in the obvious way:

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

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 these zero-length paths which were overseen in previous works. Moreover, 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 P_1 if P_i is of the form $P_i = X \;\; P_1? \;\; Y$.

The case of zero-length paths is treated analogously for zero-or-more paths. Finally, we show in Figure 4 how paths of arbitrary length are treated in case of one-or-more path expressions. In contrast to sequence paths, now the join of end point and start point of the two subpaths (in the recursive rule for ans_i) indeed must be identical and not just compatible.

It should be noted that, according to the SPARQL semantics of property paths[8], *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.

6 EXPERIMENTAL EVALUATION

In this section, we report on the experimental evaluation of our SparqLog system. We thus 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 to 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 Appendix C.

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 [29] 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. More details on the setup of our benchmark analysis can be found in Appendix C.1.

	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: Feature Coverage of SPARQL Benchmarks [29]

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, 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)* [28]

[8]https://www.w3.org/TR/SPARQL11-query/#defn_PropertyPathExpr

as the real-world benchmark of choice, as it produces the most diverse test cases [29] and covers the highest amount of features; (2) *SP2Bench* [30] is identified as the synthetic benchmark of choice, since it produces synthetic datasets with the most realistic characteristics [29]; (3) finally, since no benchmark that employs real-world settings provides satisfactory coverage of property paths, we have additionally chosen *BeSEPPI* [31] – 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 hand-crafted 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 extended the SP2Bench benchmark with property path queries – thus covering also complex recursion. 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. Since Virtuoso exhibits non-standard behaviour in many cases (as will be detailed in Section 6.2), we restrict our performance comparison of SparqLog on general SPARQL queries and on recursive property paths to Fuseki. However, since Fuseki does not support ontological reasoning, we switch to the state-of-the-art query answering and reasoning system Stardog for our performance tests with property paths in the presence of ontologies.

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 Appendix C. Below, we summarize the results:

The FEASIBLE(S) benchmark contains 77 queries out of which 9 contain features not supported by SparqLog such as complex arguments (i.e., not just a variable) in either an ORDER BY or a COUNT construct. We have therefore restricted the test for standard-conformant behaviour to the remaining 68 queries. It turned out that both SparqLog and Fuseki fully comply to the standard, 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 in total, specifically designed to test the scalability of SPARQL engines. All 3 considered systems produce the correct result for each of the 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.

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

Table 3: Compliance Test Results with BeSEPPI

To conclude, while SparqLog and Fuseki handle all considered queries from the 3 chosen benchmarks correctly, Virtuoso produces a significant number of errors. Thus, to allow for a fair comparison of systems, we restrict in the following sections our performance measurements to SparqLog and Fuseki, as both are SPARQL standard-compliant, while Virtuoso exhibits non-standard behaviour and may thus achieve faster query execution times by trading standard-compliance for faster query execution.

6.3 Performance Measurements

Experimental Setup. For our experiments we use Apache Jena Fuseki 3.15.0 and Stardog 7.7.1 and run the experiments on a Windows 10 machine with 8GB of main memory. 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). Before the execution of any benchmark query, we additionally run a simple warm-up query 3 times to warm-up the cache, i.e., populate the cache with relevant data to achieve realistic query evaluation conditions. Finally, we repeat the execution of benchmark queries 10 times and average the total loading and querying times.

Performance on general SPARQL queries. SP2Bench is a benchmark that particularly targets query optimization and computation-intensive queries. Figure 5 reveals that our system performs comparably well as Fuseki, being slightly slower on a few queries (8, 10, and 14) but dramatically faster on some queries (5, 6 and 9).

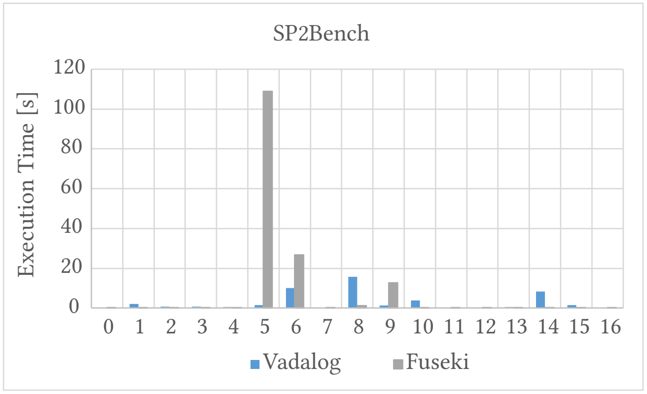

Figure 5: SP2Bench Benchmark

Property paths. Since current SPARQL benchmarks provide only rudimentary coverage of property path expressions, we have created benchmark queries ourselves for SP2Bench that contain recursive property paths. Specifically, queries 0 and 1 contain each one recursive property path with a variable on the left side, queries 2 and 3 one with a variable on the right side and queries 4 and 5 one with two variables. Figure 6 reveals that our SparqLog system particularly excels at recursive property paths with two variables (queries 4 and 5). That is, SparqLog only needs about a fifth of the time that Fuseki needs to answer these queries.

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 also wanted to measure the performance of query answering in the presence of an ontology. Since Fuseki does not provide such support, we now compare SparqLog with Stardog, which is a commonly accepted state-of-the-art system for reasoning and querying within

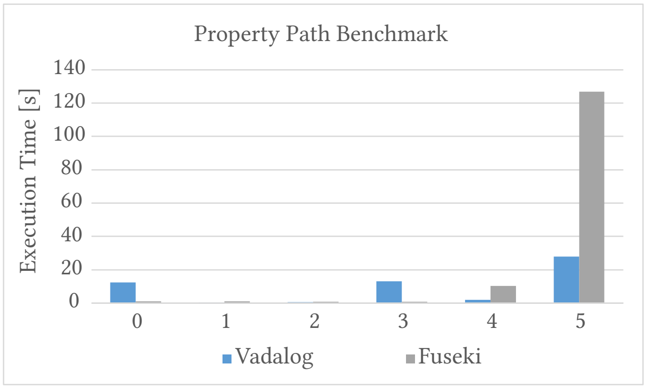

Figure 6: Property Path Benchmark

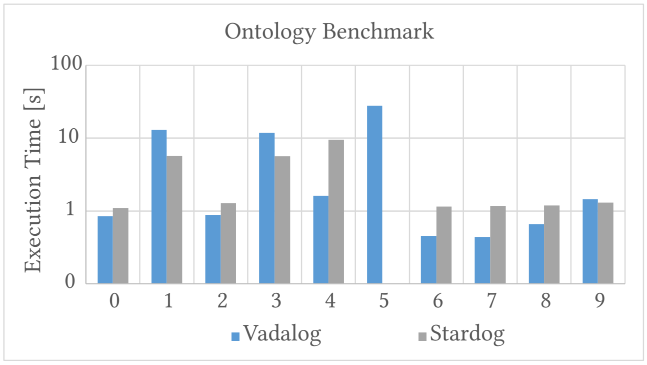

Figure 7: Ontology Benchmark

the Semantic Web. Furthermore, we have also extended the previous property path benchmark by ontological concepts such as subPropertyOf and subClassOf in the additional queries 6-9.

Figure 7 shows the outcome of these experiments. We see that SparqLog is faster than Stardog on most queries. Particularly interesting are queries 4 and 5, which (as previously discussed) 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 competitive to state-of-the-art SPARQL engines and reasoners and even beats them on 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 contribution of the engine SparqLog we provided in this paper 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.

However, our work does not stop here. 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.

REFERENCES

[1] 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. ihttps://doi.org/10.1007/978-3-540-88564-1_8

[2] 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. ihttps://doi.org/10.1007/978-3-319-46523-4_2

[3] 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.

[4] 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.

[5] 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. ihttps://doi.org/10.1007/978-3-319-21542-6_21

[6] 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.

[7] 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.

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

[9] 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. ihttps://doi.org/10.4230/LIPIcs.ICDT.2019.16

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

[11] 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.

[12] 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. ihttps://doi.org/10.1145/1514894.1514897

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

[14] 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. ihttps://doi.org/10.1007/978-3-642-15918-3_1

[15] 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. ihttps://doi.org/10.1007/978-3-030-30796-7_2

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

[17] 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. ihttps://doi.org/10.1007/978-3-319-08909-6_7

[18] 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.

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

[20] 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.

[21] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The Magic of Duplicates and Aggregates. In *International Conference on Very Large Databases (VDLB)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 264–277.

[22] Inderpal Singh Mumick and Oded Shmueli. 1993. Finiteness Properties of Database Queries. In *Proc. ADC '93*. World Scientific, Toh Tuck Link Singapore, 274–288.

[23] 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. ihttps://doi.org/10.1007/978-3-319-25010-6_1

[24] 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. ihttps://doi.org/10.1145/1242572.1242679

[25] 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.

[26] 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.

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

[28] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In *International Semantic Web Conference*.

[29] 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. ihttps://doi.org/10.1145/3308558.3313556

[30] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. 2009. SP2Bench: A SPARQL Performance Benchmark. *ArXiv* abs/0806.4627 (2009).

[31] 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. ihttps://doi.org/10.1007/978-3-030-21348-0_31

Appendix

A TRANSLATING SPARQL 1.1 TO WARDED DATALOG$^{\pm}$

In this section, we provide a detailed description of our translation from SPARQL 1.1 to Warded Datalog$^{\pm}$. Note that many of the rules thus generated are simple Datalog rules, i.e., they do not have existentially quantified variables in the head. In such cases, we shall interchangeably refer to these rules as "'Datalog rules" or "Datalog$^{\pm}$ rules". Of course, if existentially quantified variables are indeed used in the head, we shall always speak of "Datalog$^{\pm}$ rules".

We start with the translation of RDF graphs to Datalog rules in Section A.1. We then detail our translation of graph patterns and the specific translation rules for property path expressions in Sections A.2 and A.3. Finally, in Section A.4, we consider query forms.

A.1 Translation of RDF Graphs

Assume that I, L and B are disjoint infinite sets corresponding to IRIs, literals and blank nodes. An RDF term is an element in the set $T = I \cup L \cup B$. An *RDF triple* is a tuple $(s, p, o) \in T \times I \times T$ where s is called the *subject*, p is the *predicate* and o is the *object*. An *RDF graph* G is a set of RDF triples. An *RDF dataset* D is a collection of graphs including a default graph G_0 and zero or more named graphs, such that a named graph is a pair (u, G) where u is an IRI which identifies the RDF graph G.

Let G be a given RDF graph, the translation of the graph to Datalog facts is defined as follows:

(1) For each IRI, constant and blank node in G the corresponding facts $iri(X)$, $literal(X)$ and $bnode(X)$ are generated.
(2) For each named graph g a tuple $named(g)$ and for each triple (s, p, o) of graph g a fact $triple(s, p, o, g)$ where g is either "default" for the default graph or the IRI of a named graph is created.
(3) A term is either an IRI, a literal or a blank node: The set of terms is represented by the predicate $terms$.

Definition A.1 (Terms). The predicate $terms$ is defined as follows:

$$\begin{aligned} term(X) &:-\ iri(X). \\ term(X) &:-\ literal(X). \\ term(X) &:-\ bnode(X). \end{aligned}$$

A.2 Translation of SPARQL graph patterns

Assume the existence of an infinite set V of variables disjoint from T. We will use var(α) to denote the set of variables occurring in any structure α. A *graph pattern* is defined recursively as follows: a tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$ is a triple pattern; if P_1 and P_2 are graph patterns, C is a filter constraint, and $g \in I$ then $\{P_1 \,.\, P_2\}$, $\{P_1$ UNION $P_2\}$, $\{P_1$ OPT $P_2\}$, $\{P_1$ MINUS $P_2\}$, $\{P_1$ FILTER $C\}$, and; $\{$GRAPH $g\ P_1\}$ are graph patterns. A *filter constraint* is defined recursively as follows: (i) If $?X, ?Y \in V$, $c \in I \cup L$ and r is a regular expression then *true*, *false*, $?X = c$, $?X = ?Y$, bound$(?X)$, isIRI$(?X)$, isBlank$(?X)$, isLiteral$(?X)$ and regex$(?X, r)$ are *atomic filter constraints*; (ii) If C_1 and C_2 are filter constraints then $(!C_1)$, $(C_1\ \&\&\ C_2)$ and $(C_1\ ||\ C_2)$ are *Boolean filter constraints.*

A *subpattern* P' of a graph pattern P is defined to be any substring of P that is also a graph pattern. Furthermore P' is defined to be an immediate subpattern of P if it is a subpattern of P and if there is no other subpattern of P, different from P, that contains P'. A *parse tree* is specified as a tree $< V, E >$ with the set of nodes V being the subpatterns of a graph pattern P and the set of edges E containing an edge (P_1, P_2) if P_2 is an immediate subpattern of P_1.

The evaluation of a graph pattern results in a multiset of solution mappings. A *solution mapping* is a partial function $\mu : V \to T$, i.e. an assignment of variables to RDF terms. The domain of μ, denoted dom(μ), is the subset of V where μ is defined. The *empty mapping*, denoted μ_0, is the mapping satisfying that dom$(\mu_0) = \emptyset$. A *multiset of solution mappings* Ω is an unsorted list of solution mappings where duplicates are allowed. The domain of Ω is the set of variables occurring in the solution mappings of Ω.

The notion of compatible mappings is fundamental to evaluate SPARQL graphs patterns. Two mappings μ_1 and μ_2, are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all $?X \in$ dom$(\mu_1) \cap$ dom(μ_2) it satisfies that $\mu_1(?X) = \mu_2(?X)$. Note that two mappings with disjoint domains are always compatible, and the empty mapping μ_0 is compatible with any other mapping.

Definition A.2 (Compatibility). The notion of compatible mappings is simulated in Datalog by using the following rules:

$$null(\text{"}null\text{"}).$$

$$\begin{aligned} comp(X, X, X) &:-\ term(X). \\ comp(X, Z, X) &:-\ term(X), null(Z). \\ comp(Z, X, X) &:-\ term(X), null(Z). \\ comp(Z, Z, Z) &:-\ null(Z). \end{aligned}$$

The predicate $comp(X_1, X_2, X_3)$ describes the compatibility of the values X_1 and X_2. The third position X_3 represents the value, that would be used in the result tuple of a join operation.

A given SPARQL graph pattern is translated into Warded Datalog$^{\pm}$ by recursively walking through the parse tree of the query and translating each subpattern into its respective Datalog$^{\pm}$ rules (as outlined in [26]). Subpatterns of the parse tree are indexed. The root has index one, 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. Since the order of variables in predicates is relevant, some variable sets will need to be lexicographically ordered, which is denoted by $\overline{x}$ following the notation of [26]. $\overline{var}(P)$ shall represent the lexicographically ordered tuple of variables of P. Moreover a renaming function $v_j : V \to V$ is defined.

For a given graph pattern P and a given dataset $D = \langle G, G_{named} \rangle$ (where G is the default graph and G_{named} is the set of named graphs) the core translation function $\tau(P, dst, D, NodeIndex)$ is defined as follows:

(1) P is the graph pattern that should be translated next.
(2) dst is a Boolean value that describes whether the results should have set semantics ($dst = True$) or bag semantics ($dst = False$).
(3) D is the graph on which the query should be evaluated.
(4) *NodeIndex* is the index of the pattern P that should be translated.

In Definitions A.3 to A.11 below, we present the translation function τ for the various language constructs of graph patterns in SPARQL 1.1.

Definition A.3 (Triple). Let P_i be the i-th subpattern of P and furthermore let P_i be a triple pattern (s, p, o), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

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

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

Definition A.4 (Graph). Let P_i be the i-th subpattern of P and furthermore let P_i be (GRAPH g P_1), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

$$\begin{aligned} ans_i(\overline{var}(P_1), D) :- \ & ans_{2i}(\overline{var}(P_1), g), \\ & named(g). \\ \tau(P_1, \mathit{true}, g, 2i). & \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

$$\begin{aligned} ans_i(Id, \overline{var}(P_1), D) :- \ & ans_{2i}(Id_1, \overline{var}(P_1), g), \\ & named(g). \\ \tau(P_1, \mathit{false}, g, 2i) & \end{aligned}$$

Definition A.5 (Join). Let P_i be the i-th subpattern of P and furthermore let P_i be $(P_1 \,.\, P_2)$, then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

$$\begin{aligned} ans_i(\overline{x}, D) :- \ & ans_{2i}(v_1(\overline{var}(P_1)), D), \\ & ans_{2i+1}(v_2(\overline{var}(P_2)), D), \\ & comp(v_1(x_1), v_2(x_1), x_1), \\ & \ldots, comp(v_1(x_n), v_2(x_n), x_n). \end{aligned}$$

$$\begin{aligned} \tau(P_1, \mathit{true}, D, 2i) \\ \tau(P_2, \mathit{true}, D, 2i+1) \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

$$\begin{aligned} \tau(P_1, \mathit{false}, D, 2i) \\ \tau(P_2, \mathit{false}, D, 2i+1) \end{aligned}$$

With

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

Definition A.6 (Union). Let P_i be the i-th subpattern of P and furthermore let P_i be (P_1 UNION P_2), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

$$\begin{aligned} ans_i(\overline{var}(P_i), D) :- \ & ans_{2i}(\overline{var}(P_1), D), \\ & null(x_1), \ldots null(x_n). \\ ans_i(\overline{var}(P_i), D) :- \ & ans_{2i+1}(\overline{var}(P_2), D), \\ & null(y_1), \ldots null(y_m). \end{aligned}$$

$$\begin{aligned} \tau(P_1, \mathit{true}, D, 2i) \\ \tau(P_2, \mathit{true}, D, 2i+1) \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

$$\begin{aligned} \tau(P_1, \mathit{false}, D, 2i) \\ \tau(P_2, \mathit{false}, D, 2i+1) \end{aligned}$$

With

- $\{x_1, \ldots, x_n\} = var(P_2) \setminus var(P_1)$
- $\{y_1, \ldots, y_m\} = var(P_1) \setminus var(P_2)$

Definition A.7 (Optional). Let P_i be the i-th subpattern of P and furthermore let P_i be (P_1 OPT P_2), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

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

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

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

$$\begin{aligned} \tau(P_1, \mathit{true}, D, 2i) \\ \tau(P_2, \mathit{true}, D, 2i+1) \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

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

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

$$\tau(P_1, \mathit{false}, D, 2i)$$
$$\tau(P_2, \mathit{false}, D, 2i+1)$$

With

- $\overline{var}(P_i) = \overline{var(P_1) \cup var(P_2)}$
- $\{x_1, \ldots, x_n\} = var(P_1) \cap var(P_2)$
- $\{y_1, \ldots, y_m\} = var(P_2) \setminus var(P_1)$
- $v_1, v_2 : var(P_1) \cap var(P_2) \longrightarrow V$, such that $Image(v_1) \cap Image(v_2) = \{\}$

Definition A.8 (Filter). Let P_i be the i-th subpattern of P and furthermore let P_i be (P_1 FILTER C), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

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

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

Definition A.9 (Optional Filter). Let P_i be the i-th subpattern of P and furthermore let P_i be (P_1 OPT(P_2 FILTER C)), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

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

$$\begin{aligned}
ans_i(\overline{var}(P_i), D) :-\ & ans_{2i}(v_1(\overline{var}(P_1)), D),\\
& ans_{2i+1}(v_2(\overline{var}(P_2)), D),\\
& comp(v_1(x_1), v_2(x_1), x_1),\\
& \ldots, comp(v_1(x_n), v_2(x_n), x_n), C.
\end{aligned}$$

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

$$\tau(P_1, \mathit{true}, D, 2i)$$
$$\tau(P_2, \mathit{true}, D, 2i+1)$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

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

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

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

$$\tau(P_1, \mathit{false}, D, 2i)$$
$$\tau(P_2, \mathit{false}, D, 2i+1)$$

With

- $\overline{var}(P_i) = \overline{var(P_1) \cup var(P_2)}$
- $\{x_1, \ldots, x_n\} = var(P_1) \cap var(P_2)$
- $\{y_1, \ldots, y_m\} = var(P_2) \setminus var(P_1)$
- $v_1, v_2 : var(P_1) \cap var(P_2) \longrightarrow V$, such that $Image(v_1) \cap Image(v_2) = \{\}$

Definition A.10 (Minus). Let P_i be the i-th subpattern of P and furthermore let P_i be (P_1 MINUS P_2), then $\tau(P_i, true, D, i)$ is defined as:

$$\begin{aligned}
ans_{join-i}(\overline{x}, D) :- \;& ans_{2i}(\overline{var}(P_1), D),\\
& ans_{2i+1}(v_2(\overline{var}(P_2)), D),\\
& comp(x_1, v_2(x_1), z_1),\\
& \ldots, comp(x_n, v_2(x_n), z_n).\\
ans_{equal-i}(\overline{x}, D) :- \;& ans_{join-i}(\overline{x}, D),\\
& x_1 = v_2(x_1),\ not\ \ null(x_1).\\
& \ldots\\
ans_{equal-i}(\overline{var}(P_1), D) :- \;& ans_{join-i}(\overline{var}(P_i), D),\\
& x_n = v_2(x_n),\ not\ \ null(x_n).\\
ans_i(\overline{var}(P_1), D) :- \;& ans_{2i}(\overline{var}(P_1), D),\\
& not\ \ ans_{equal-i}(\overline{var}(P_1), D).
\end{aligned}$$

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

And $\tau(P_i, false, D, i)$ is defined as:

$$\begin{aligned}
ans_{join-i}(\overline{x}, D) :- \;& ans_{2i}(Id_1, \overline{var}(P_1), D),\\
& ans_{2i+1}(Id_2, v_2(\overline{var}(P_2)), D),\\
& comp(x_1, v_2(x_1), z_1),\\
& \ldots, comp(x_n, v_2(x_n), z_n).\\
ans_{equal-i}(\overline{x}, D) :- \;& ans_{join-i}(\overline{x}, D),\\
& x_1 = v_2(x_1),\ not\ \ null(x_1).\\
& \ldots\\
ans_{equal-i}(\overline{var}(P_1), D) :- \;& ans_{join-i}(\overline{var}(P_i), D),\\
& x_n = v_2(x_n),\ not\ \ null(x_n).\\
ans_i(Id, \overline{var}(P_1), D) :- \;& ans_{2i}(Id_1, \overline{var}(P_1), D),\\
& not\ \ ans_{equal-i}(\overline{var}(P_1), D).
\end{aligned}$$

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

With

- $\overline{x} = \overline{var(P_1) \cup v_2(var(P_2))}$
- $\{x_1, \ldots, x_n\} = var(P_1) \cap var(P_2)$
- $v_2 : var(P_1) \cap var(P_2) \rightarrow V \setminus var(P_1)$

Property path patterns are given in the form $S\ \ P_1\ \ O$, where P_1 is a property path expression. Due to the complex semantics of property paths, we have introduced a separate translation function τ_{PP} for property path expressions, which we will take a closer look at in Section A.3.

Definition A.11 (Property Path Pattern). Let P_i be the i-th subpattern of P and furthermore let $P_i = S\ \ P_1\ \ O$ be a property path pattern, then $\tau(P_i, true, D, i)$ is defined as:

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

And $\tau(P_i, false, D, i)$ is defined as:

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

With τ_{PP} being the translation function for property path expressions defined next.

A.3 Translation of Property Path Expression

A *property path expression* (or, simply, a *path expression*) is defined recursively as follows:

- if $a \in I$, then a is a predicate path expression;
- If p_1 and p_2 are property path expressions then
  - $^\wedge p_1$ is an inverse path expression;
  - $p_1 \mid p_2$ is an alternative path expression;
  - p_1/p_2 is a sequence path expression;
  - $p_1?$ is a zero-or-one path expression;
  - p_1+ is a one-or-more path expression;
  - p_1* is a zero-or-more path expression.

A *property path triple* is a tuple t of the form (u, p, v), where $u, v \in (I \cup V)$ and p is a property path expression. Analogously to our translation function $\tau(P, dst, D, NodeIndex)$ for SPARQL patterns, we define a translation function $\tau_{PP}(P, dst, S, O, D, NodeIndex)$ for property path expressions, 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

In the following we will only state the translations for property path expressions under bag semantics (i.e. $dst = false$), since for set semantics the IDs are simply left out or set to a constant value (e.g. $Id = []$).

Definition A.12 (Link Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ \ p_1\ \ Y$ be a link property path expression, then $\tau_{PP}(P_i, false, S, O, D, i)$ is defined as:

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

Definition A.13 (Inverse Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ \ \hat{}P_1\ \ Y$ be an inverse property path expression, then $\tau_{PP}(P_i, false, S, O, D, i)$ is defined as:

$$ans_i(Id, X, Y, D) :- \; ans_{2i}(Id_1, Y, X, D).$$
$$\tau_{PP}(P_1, false, S, O, D, 2i)$$

Definition A.14 (Sequence Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ \ P_1/P_2\ \ Y$ be a sequence property path expression, then $\tau_{PP}(P_i, false, S, O, D, i)$ is defined as:

$$\begin{aligned}
ans_i(Id, W, Z, D) :- \;& ans_{2i}(Id_1, W, X, D), ans_{2i+1}(Id_2, Y, Z, D),\\
& comp(X, Y, Y).
\end{aligned}$$
$$\tau_{PP}(P_1, false, S, O, D, 2i)$$
$$\tau_{PP}(P_2, false, S, O, D, 2i+1)$$

Definition A.15 (Alternative Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let

$P_i = X\ (P_1|P_2)\ Y$ be a alternative property path expression, then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$$\begin{aligned} ans_i(Id, X, Y, D) &:-\ ans_{2i}(Id_1, X, Y, D).\\ ans_i(Id, X, Y, D) &:-\ ans_{2i+1}(Id_1, X, Y, D).\\ &\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)\\ &\tau_{PP}(P_2, \mathit{false}, S, O, D, 2i+1) \end{aligned}$$

Definition A.16 (One Or More Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ P_1+\ Y$ be a one-or-more property path expression, then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$$ans_i(Id, X, Y, D) :-\ ans_{2i}(Id_1, X, Y, D), Id = [].$$
$$ans_i(Id, X, Z, D) :-\ ans_{2i}(Id_1, X, Y, D), ans_i(Id_2, Y, Z, D), Id = [].$$
$$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$$

Definition A.17 (SubjectOrObject). The subjectOrObject predicate defines intuitively the set of all possible subjects and objects of a graph, i.e.:

$$\begin{aligned} subjectOrObject(X) &:-\ triple(X, P, Y, D).\\ subjectOrObject(Y) &:-\ triple(X, P, Y, D). \end{aligned}$$

Definition A.18 (Zero Or One Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ P_1?\ Y$ be a zero-or-one property property path expression. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ consists of the following rules:

$$\begin{aligned} ans_i(Id, X, X, D) &:-\ subjectOrObject(X), Id = [].\\ ans_i(Id, X, Y, D) &:-\ ans_{2i}(Id_1, X, Y, D), Id = [].\\ &\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i) \end{aligned}$$

Moreover, if either one of S and O is a variable and the other is a non-variable t or both S and O are the same non-variable t, then the following rule is added:

$$ans_i(Id, X, X, D) :-\ not\ \ Term(X), X = t, Id = [].$$

It should be noted that, according to the SPARQL semantics of property paths[9], *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.

Definition A.19 (Zero Or More Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ P_1*\ Y$ be a zero-or-more property property path expression. Then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ consists of the following rules:

$$ans_i(Id, X, X, D) :-\ subjectOrObject(X), Id = [].$$
$$ans_i(Id, X, Y, D) :-\ ans_{2i}(Id_1, X, Y, D), Id = [].$$
$$ans_i(Id, X, Z, D) :-\ ans_{2i}(Id_1, X, Y, D), ans_i(Id_2, Y, Z, D), Id = [].$$
$$\tau_{PP}(P_1, \mathit{false}, S, O, D, 2i)$$

[9]https://www.w3.org/TR/SPARQL11-query/#defn_PropertyPathExpr

Moreover, if either one of S and O is a variable and the other is a non-variable t or both S and O are the same non-variable t, then the following rule is added:

$$ans_i(Id, X, X, D) :-\ not\ \ Term(X), X = t, Id = [].$$

Essentially, the zero-or-more property path is a combination of the zero-or-one and one-or-more property path.

Definition A.20 (Negated Property Path). Let P_i be the i-th subpattern of a property path expression PP and furthermore let $P_i = X\ !(P_1)\ Y$ be a negated property path expression, then $\tau_{PP}(P_i, \mathit{false}, S, O, D, i)$ is defined as:

$$ans_i(Id, X, Y, D) :-\ triple(X, P, Y, D), P! = p_{f_1}, \ldots, P! = p_{f_n}.$$
$$ans_i(Id, Y, X, D) :-\ triple(X, P, Y, D), P! = p_{b_1}, \ldots, P! = p_{b_m}.$$

with

- $p_{f_1}, \ldots, p_{f_n} \in \{p | p \in P_1\}$ ... i.e. the set of negated forward predicates.
- $p_{b_1}, \ldots, p_{b_m} \in \{p | \hat{\ } p \in P_1\}$ ... i.e. the set of negated backward predicates.

A.4 Translation of Query Forms

Let P_1 be a graph pattern and W be a set of variables. We consider two types of query forms: (SELECT W P_1) and (ASK P_1). Their translation is given below.

Definition A.21 (Select). Let P_i be the i-th subpattern of P and furthermore let P_i be (SELECT W P_1), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

$$\begin{aligned} ans_i(\overline{var}(W), D) &:-\ ans_{2i}(\overline{var}(P_1), D).\\ &\tau(P_1, \mathit{true}, D, 2i) \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

$$\begin{aligned} ans_i(Id, \overline{var}(W), D) &:-\ ans_{2i}(Id_1, \overline{var}(P_1), D).\\ &\tau(P_1, \mathit{false}, D, 2i) \end{aligned}$$

Definition A.22 (Ask). Let P_i be the i-th subpattern of P and furthermore let P_i be ASK P_1), then $\tau(P_i, \mathit{true}, D, i)$ is defined as:

$$\begin{aligned} ans_i(HasResult) &:-\ ans\_ask_i(HasResult).\\ ans_i(HasResult) &:-\ not\ \ ans\_ask_i(True), HasResult = false.\\ ans\_ask_i(HasResult) &:-\ ans_{2i}(\overline{var}(P_1), D), HasResult = true.\\ &\tau(P_1, \mathit{true}, D, 2i) \end{aligned}$$

And $\tau(P_i, \mathit{false}, D, i)$ is defined as:

$$\begin{aligned} ans_i(HasResult) &:-\ ans\_ask_i(HasResult).\\ ans_i(HasResult) &:-\ not\ \ ans\_ask_i(True), HasResult = false.\\ ans\_ask_i(HasResult) &:-\ ans_{2i}(Id_1, \overline{var}(P_1), D), HasResult = true.\\ &\tau(P_1, \mathit{false}, D, 2i) \end{aligned}$$

B SOME IMPLEMENTATION DETAILS

Some basic principles. The SPARQL to Warded Datalog$^\pm$ translator was implemented in Java using the library org.apache.jena to parse SPARQL query strings and handle operations, solution modifiers, basic graph patterns, etc. appropriately. The ARQ algebra query parser[10] of Apache Jena parses SPARQL query strings

[10]https://jena.apache.org/documentation/query/

in a top-down fashion. First query forms are parsed, next solution modifiers and in the end operations starting from the outer-most operation going inward.

In contrast, our developed SPARQL to Warded Datalog$^\pm$ parser analyses queries bottom-up. Thus, the translation starts at basic graph patterns and continues upwards. This setup is necessary, as the variables inside an expression need to be kept track of, when parsing it, as rules usually modify results of sub-operations. Therefore, it needs to be known which variables occur in the respective subresult predicates.

Datatypes, languages, and compatibility. Our translation engine partially supports datatypes and language tags by adding two additional arguments to each variable, containing the respective information. This has implications on most SPARQL operations such as *UNION*, *JOIN*, and *FILTER*. For example, in the case of *JOIN* operations, we have extended the existing translation of [2] by developing two additional comparison predicates (*compD* and *compL*). In [2] the predicate *comp* is used for computing the compatibility between two variables. The new predicates *compD* and *compL* are used to compute the respective compatibility for their datatypes and language tags, which is done in the same way as for variables thus far.

Moreover, Vadalog (and Datalog in general) joins variables of the same rule by name, however the semantics of joins in SPARQL is different to the one of Datalog. It is for that reason (1) that we have to prefix/rename variables in such a way that Vadalog's internal join strategy is prevented and (2) that we introduce the join predicate *comp* described in Section A.2 to realise SPARQL join semantics.

Skolem functions (for bag semantics). The Skolem function generator lies at the heart of how we preserve duplicate results. As in the work of [9], we introduce IDs to preserve Datalog bag semantics in Warded Datalog$^\pm$ set semantics. Therefore, each generated result tuple is distinguished from its duplicates by a tuple ID (referred to as TID in [9]). However, instead of simply generating nulls, our ID generation process is abstracted away by the *getSkolF* function of the *skolFG* object. We thus generate IDs as follows. Since the grounding of each positive atom in the rule body is responsible for the generation of a tuple in Datalog$^\pm$, we extract a sorted list of all variables occurring in positive atoms of the rule body *bodyVars*. Finally, the tuple ID is generated by assigning it to a list starting with the string "$f_{<ruleID>}$", followed by the list of positive body variables *bodyVars* and a string *label*. The strings "$f_{<ruleID>}$" and *label* were added, as we use "$f_{<ruleID>}$" to identify the translated rules of the processed operator at the current translation step, while *label* provides additional information when needed.

This setup preserves generated duplicates of SPARQL bag semantics in Warded Datalog$^\pm$ set semantics by utilizing their provenance information to make them distinguishable. Furthermore, it provides information for debugging/explanation purposes of the reasoning process, as each tuple carries the information which rule and grounding has led to its generation. As an added bonus this layer of abstraction may be used to adapt different duplicate generation semantics/strategies as might be necessary for different applications by simply exchanging the *skolFG* by any self-implemented solution.

C FURTHER DETAILS OF THE EXPERIMENTAL EVALUATION

In Section 6.1, we have reported on our analysis of various SPARQL benchmarks. One of the outcomes of this analysis was the selection of three concrete benchmarks, which we then used in Section 6.2 to test the standard-compliance of SparqLog and two state-of-the-art SPARQL engines. In this section, we provide further details on how we have set up the benchmark analysis (Section C.1) and the compliance tests (Section C.2)

C.1 Setup of the Benchmark Analysis

Before we dive into the benchmark analysis, we briefly introduce its setup in this section. Our analysis employs a similar way of counting SPARQL features as is done in [29]. We have counted each feature once per query in which it occurs, with one exception being the *DISTINCT* feature. As in [29], we count the DISTINCT feature only if it is applied to the entire query. This is also in line with our interest of testing bag and set semantics in combination with different SPARQL features.

In contrast to [29] we do not limit our benchmark analysis to *SELECT* queries, but rather analyse all queries provided at the GitHub repository of the dice group[11]. Therefore, we analyse also the *DESCRIBE* queries of e.g. BSBM. Moreover, since the SP2Bench query file does not contain the hand-crafted *ASK* queries provided on its homepage[12], we have chosen to add these to the benchmark to be able to analyse the complete query-set of SP2Bench. For these reasons, the results of overlapping SPARQL features and benchmarks from our analysis in Table 2 and the one of [29] differ slightly.

Furthermore, note that we do not display basic features, such as *Join*, *Basic Graph pattern*, etc. in Table 2, as these features are, of course, covered by each of the considered benchmarks. Furthermore, we have chosen to omit the SPARQL features *ORDER BY*, *LIMIT* and *OFFSET* from the table since these cannot be evaluated currently by SparqLog due to the limitations of Vadalog and our translation engine, as has already been mentioned in our discussion of the SPARQL feature coverage of SparqLog in Table 1. Also *ASK* was left out in the table, as ASK is not an especially challenging feature and was therefore removed for space reasons. Moreover, we have chosen to only include the *REGEX* filter constraint in the feature coverage table and no other specific constraints, as the *REGEX* function is argued to be of vital importance for SPARQL users in [29]. For this reason, we have chosen to cover this feature by our translation engine in addition to the other filter constraints. Finally, we have not included the SPARQL features *MINUS* and the *inverted*, *zero-or-one*, *zero-or-more*, *one-or-more* and *negated property path*, as none of the selected benchmarks covers any of these SPARQL features.

C.2 Setup of the SPARQL Compliance Tests

As detailed in Section 6.1, we have selected three benchmarks (BeSEPPI, SP2Bench and FEASIBLE (S)) for evaluating the standard-compliance of the chosen three systems (SparqLog, Jena Fuseki, and OpenLink Virtuoso). For our experiments, we use Apache Jena

[11]https://hobbitdata.informatik.uni-leipzig.de/benchmarks-data/queries/
[12]http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/queries.php

Fuseki 3.15.0 and Virtuoso Open Source Edition, version 7.2.5. The experiments are run on a Windows 10 machine with 8GB of main memory. In this section, we explain the setup of the standard-compliance tests that we performed on our SparqLog system and the two state-of-the-art SPARQL engines Fuseki and Virtuoso. Moreover, we mention some challenges with these tests and we provide further details on the outcome of the compliance tests.

C.2.1 Benchmark Generation. In order to carry out our standard-compliance tests, we first have to make the queries and the data provided by the benchmarks accessible to the tested systems. While this turns out to be an easy task for BeSEPPI and SP2Bench, some care is required for the FEASIBLE (S) benchmark.

BeSEPPI and SP2Bench. The BeSEPPI benchmark contains queries and a dataset for the evaluation of property path queries. Its dataset can be directly loaded into the selected systems and its queries can directly be executed. The SP2Bench benchmark contains 17 hand-crafted queries and a benchmark dataset generator. For the purpose of our compliance tests, we have generated a dataset with 50k triples, which was loaded into each of the considered systems.

FEASIBLE. The FEASIBLE benchmark contains a query generator, which generates queries for an arbitrary dataset that provides a query-log. In the case of the FEASIBLE (S) benchmark, we have chosen the Semantic Web Dog Food (SWDF) dataset and we have generated 100 queries using the SWDF query-log. However, some additional work was required before we could use the FEASIBLE (S) benchmarks for our tests.

The first complication arises from the fact that Vadalog uses Java sorting semantics, whereas the SPARQL standard defines its own ordering semantics. We had to remove *LIMIT* and *OFFSET* from each query of the FEASIBLE (S) benchmark, as queries with these features can only be reasonably evaluated (comparing the generated query results, rather than only checking if their cardinalities are equal) if the results are sorted and if each considered RDF query and storage system provides the same sorting semantics. Some queries of the generated benchmark only differed from each other in the argument of the *LIMIT* or *OFFSET* clause. Thus, after removing all *LIMIT* and *OFFSET* clauses, we ended up with duplicate queries. These duplicate queries were eliminated, leaving the FEASIBLE (S) benchmark with a total number of 77 unique queries.

Moreover, Vadalog does currently not support UTF-8 characters. We were therefore faced with the necessity of changing the encoding of the SWDF dataset of the FEASIBLE (S) benchmark. We have made the plausible assumption that dropping non-ASCII characters from RDF strings would not lead to vastly different results and we have therefore simply deleted all non-ASCII characters from the SWDF dataset.

Furthermore, since the FEASIBLE (S) benchmark includes queries with the *GRAPH* feature (which selects the graph IRI of RDF triples), we have loaded the SWDF dataset both into the default graph of each tested system and into a named graph for the FEASIBLE benchmark to be able to test the GRAPH feature.

C.2.2 Challenges of the Evaluation Process. The evaluation of the standard compliance of the three chosen systems requires the comparison of query results. In case of BeSEPPI, the benchmark also provides the expected result for each query. We therefore have to compare the result produced by each of the three systems with the correct result defined by the benchmark itself. In contrast, FEASIBLE and SP2Bench do not provide the expected results for their queries. We therefore use a majority voting approach to determine the correct answer. That is, we compare the query results produced by the three considered systems and accept a result as the expected query answer if it is equal to the generated query result of at least two of the tested systems.

A major challenge for comparing query results (both, when comparing the result produced by one system with the expected result defined by the benchmark itself and when comparing the results produced by two systems) comes from *blank nodes.* On the one hand, each system employs its own specific functionality for assigning blank node names. Therefore, to compare blank nodes between the different result multisets, a mapping between the internal system-specific blank node names has to be found. However, finding such a mapping comes down to finding an isomorphism between two arbitrarily sized tables, containing only blank nodes, which requires exponential time in the worst case and which poses a serious problem for large result multisets with many blank nodes. We have therefore tried out a simple heuristic to find a suitable mapping between blank nodes by sorting the query results without considering blank node names first. We then iterate over both results and finally, each time when a new blank node name is encountered, we save the mapping between the system-specific blank node names. Even though this is a very simple heuristic, it has worked quite well in many cases. Nevertheless, there are cases, where this simple procedure infers wrong blank node mappings, even though the results are semantically equivalent. Hence, due to the instability of this efficient blank node checking heuristic, we have chosen to remove the evaluation of blank nodes from our compliance tests. That is, our current evaluation test suite does for this reason not distinguish between different blank node names but, of course, it distinguishes between all other terms.

C.2.3 Outcome of the Compliance Tests. The outcome of our compliance tests is based on the notions of *correctness* and *completeness* of query results, which we define in the same way as done in [31]:

Correctness defines the ratio of correct tuples generated by the tested system for a query. For a *SELECT* query q with $R_{expected}(q)$ being the expected result of q and $R_{sys}(q)$ being the response of system *sys* to the *SELECT* query, [31] defines correctness as follows:

$$correct(q) = \begin{cases} \frac{|R_{expected}(q) \cap R_{sys}(q)|}{|R_{sys}(q)|}, & \text{if } R_{sys}(q) \neq 0 \\ 1, & \text{otherwise} \end{cases}$$

It intuitively accepts a result as correct ($correct(q) = 1$) if the returned result of the considered system $R_{sys}(q)$ is a subset of the expected answer $R_{expected}(q)$. For *ASK* queries we consider a result to be correct only if it exactly matches the expected answer, as done in [31].

Completeness defines the ratio of all accepted result-tuples generated by the tested system for a query. For a *SELECT* query q with $R_{expected}(q)$ being the expected result of q and $R_{sys}(q)$ being the response of system *sys* to the *SELECT* query, [31] defines completeness as follows:

$$complete(q) = \begin{cases} \frac{|R_{expected}(q) \cap R_{sys}(q)|}{|R_{expected}(q)|}, & \text{if } R_{expected}(q) \neq 0 \\ 1, & \text{otherwise} \end{cases}$$

It intuitively accepts a result as complete ($complete(q) = 1$) if the expected answer $R_{expected}(q)$ is a subset of the returned result of the considered system $R_{sys}(q)$. For *ASK* queries we consider a result to be complete only if it exactly matches the expected answer, as done in [31].

FEASIBLE (S). FEASIBLE (S) contains 77 unique queries. From these 77 queries, a total of 68 are accepted by our translation engine and were thus used for the correctness and completeness evaluation of our system. The remaining 9 queries could not be translated into Datalog$^{\pm}$ programs, since they contain features that are currently not supported by our SparqLog system for the following reasons:

Three queries contain a complex expression in an *ORDER BY* statement and two queries contain complex expressions in *COUNT* aggregates. Both features are currently supported by SparqLog only for simple expressions consisting of a single variable. Two queries cannot be translated currently due to the missing support of our engine for the functions *ucase* and *contains*. However, these SPARQL features do not impose any conceptual hurdle and were only left out from the first version of SparqLog since we considered their priority as low. Similarly, two queries are not accepted by our engine at the moment, as they contain the *DATATYPE* feature. Again, this is mainly due to the low priority that we gave this feature. Since our translation engine already tracks datatypes and language tags of RDF terms, it should be no problem to integrate it in later versions of SparqLog.

On the remaing 68 queries that are accepted by our system, our translation engine produces for all executed 68 queries the same result as Apache Jena Fuseki. In contrast, OpenLink Virtuoso returned an erroneous result for 14 queries 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.

SP2Bench. The SP2Bench benchmark contains 17 queries in total and is specifically designed to test the scalability of of SPARQL engines. All three considered systems produce identical results for each of the 17 queries.

BeSEPPI. The BeSEPPI benchmark contains 236 queries, specifically designed to evaluate the correct and complete support of property path features. We have already summarized the outcome of our compliance tests of the 3 considered systems on this benchmark in Table 3 in Section 6.2. In a nutshell, while SparqLog and Fuseki follow the SPARQL standard for the evaluation of property paths, Virtuoso produces errors for 18 queries and returns incomplete results for another 13 queries.

We conclude this section by a more detailed look into the problems Virtuoso is currently facing with property path expressions. As already observed in [31], Virtuoso produces errors for zero-or-one, zero-or-more and one-or-more property paths that contain two variables. Furthermore, the error messages state that the transitive start is not given. Therefore, we come to the same conclusion as [31] that these features were most likely left out on purpose, since Virtuoso is based on relational databases and it would require huge joins to answer such queries. Moreover, we have noticed that the errors for inverse negated property paths (reported in [31]) have been fixed in the current OpenLink Virtuoso release.

Virtuoso produces 10 incomplete results when evaluating one-or-more property path queries. As already discovered in [31], they all cover cases with cycles and miss the starting node of the property path, indicating that the one-or-more property path might be implemented by evaluating the zero-or-more property path first and simply removing the starting node from the computed result. Finally, in contrast to the results of [31], we have found that the current version of OpenLink Virtuoso generates wrong answers for queries that contain alternative property paths. Virtuoso generates for three alternative property path queries incomplete results, which differ from the results of Fuseki and SparqLog by missing all duplicates, which should have been generated.