Data Provenance for SHACL

Thomas Delva
IDLab, Ghent University, imec
Ghent, Belgium
thomdelva@gmail.com

Anastasia Dimou
Dept. Computer Science, KU Leuven
Leuven, Belgium
anastasia.dimou@kuleuven.be

Maxime Jakubowski
DSI, Hasselt University
Hasselt, Belgium
maxime.jakubowski@uhasselt.be

Jan Van den Bussche
DSI, Hasselt University
Hasselt, Belgium
jan.vandenbussche@uhasselt.be

ABSTRACT

In constraint languages for RDF graphs, such as ShEx and SHACL, constraints on nodes and their properties are known as "shapes". Using SHACL, we propose in this paper the notion of *neighborhood* of a node v satisfying a given shape in a graph G. This neighborhood is a subgraph of G, and provides data provenance of v for the given shape. We establish a correctness property for the obtained provenance mechanism, by proving that neighborhoods adhere to the Sufficiency requirement articulated for provenance semantics for database queries. As an additional benefit, neighborhoods allow a novel use of shapes: the extraction of a subgraph from an RDF graph, the so-called shape fragment. We compare shape fragments with SPARQL queries. We discuss implementation strategies for computing neighborhoods, and present initial experiments demonstrating that our ideas are feasible.

1 INTRODUCTION

An important functionality expected of modern data management systems [1] is that they can provide *provenance* for the results they produce in response to queries or constraint checks. Intuitively, the provenance of a query result explains why the result was produced. Provenance typically takes the form of a subinstance, containing the data on which the produced result depends, or the data that is responsible for the result.

Provenance semantics have been proposed for a variety of data models and query languages, as surveyed by Glavic [27], even with many different proposals for the standard relational model and conjunctive queries. For the Shapes Constraint Language, SHACL [54], however, a provenance semantics has been lacking so far. Our goal in this paper is to fill this gap.

SHACL is the W3C-recommended language for formulating *constraints* (called "shapes") on nodes in graph data, more specifically, RDF graphs [52]. In RDF, a framework often used on the Web, data is represented as sets of subject–property–object triples. Viewing properties as labeled edges, such a set of triples is indeed naturally interpreted as a labeled graph over the subjects and objects.

Example 1.1. Consider a publication graph (like the DBLP database) in RDF, where nodes represent papers, authors, and classes. We have :author-labeled edges from papers to their authors, and rdf:type-labeled edges from nodes to their class (e.g., paper, student, professor). A node on which a constraint is checked will be referred to as a *focus node*. Consider the constraint "the focus node has at least one author of type student". In the language SHACL, this constraint is expressed by the following shape:

```
:WorkshopShape sh:property [
  sh:path :author ; sh:qualifiedMinCount 1 ;
  sh:qualifiedValueShape [ sh:class :Student ] ] .
```

The first line introduces the shape and names it :WorkshopShape; the other two lines define the actual constraint in SHACLs RDF-based syntax.

Provenance semantics is normally defined for query languages, not for constraint languages. Yet, any shape (constraint) ϕ can be naturally treated as the query that returns the set of nodes from the input graph that conform to ϕ. Following this idea, we will propose a provenance semantics for SHACL that returns, for any shape ϕ, any RDF graph G, and any focus node v from G that conforms to the shape, a certain subset of G. This subset, which we call the *neighborhood* of v in G with respect to ϕ, intuitively consists of the triples from G that contribute to v conforming to ϕ.

Example 1.2. For our example :WorkshopShape, we will define the neighborhood of a conforming node v to consist of all triples (v :author x) from the graph where the graph also has the triple (x rdf:type Student), and that triple is also included in the neighborhood.

The above example involves a simple positive-existential constraint, but SHACL has quite powerful logical constructs, including negation, universal and counting quantifiers, path expressions, and primitives for equality and disjointness. This means that giving a nontrivial definition of neighborhood is challenging, if we want neighborhoods to satisfy an essential criterion known as *sufficiency* [27]. Simply put, a neighborhood N of a node v with respect to a shape ϕ is sufficient if v still conforms to ϕ when evaluated in the subgraph N. We will prove sufficiency for our provenance semantics for SHACL.

For conjunctive queries or positive-existential queries, sufficiency is easy to satisfy. For a language with the logical constructs mentioned above, however, we are the first to present a nontrivial provenance semantics for which sufficiency can be proved. We specify "nontrivial" here, as one can always define the neighborhood to be the entire graph and obtain sufficiency trivially. Indeed, the challenge is to keep only the relevant triples, without throwing out too much. Also, thanks to negation, we obtain both "why" and "why not" provenance [34]: if v does not conform to a shape ϕ, then its neighborhood for the shape $\neg\phi$ provides the explanation.

Interestingly, neighborhoods suggest an opportunity to leverage shapes beyond conformance checking, and use them also to

retrieve data. Specifically, given a shape ϕ and an input graph G, we can retrieve the subgraph of G formed by the union of all neighborhoods of all nodes in G that conform to ϕ. We refer to the result as the *shape fragment* of G with respect to ϕ. We will actually prove a stronger version of sufficiency, to the effect that a node v conforms to ϕ in G if and only if conforms to ϕ in the shape fragment of G with respect to ϕ.

Readers familiar with the language will point out that in standard usage of SHACL, shapes are associated with *targets*, which are simple kinds of node-returning queries. Such a target–shape pair represents an inclusion statement, to the effect that all nodes returned by the target must satisfy the shape. Thus, in SHACL one specifies a collection of inclusion statements, which we refer to as a *shape schema*.[1] The task of *validation* then amounts to checking whether an input graph conforms to the schema, i.e., satisfies all inclusions.

In our work, we will duly generalize the notion of shape fragments to shape schemas, and also extend the sufficiency result to them.

Example 1.3. We may associate to our shape :WorkshopShape the target that retrieves all papers. In SHACL syntax this is expressed by adding the statement :WorkshopShape sh:targetClass :Paper to the shapes graph. An RDF graph G validates against the resulting schema if for every triple (v rdf:type :Paper) in G, node v conforms to :WorkshopShape.

The shape fragment of G for this schema, as we will define it, consists of all the above triples (v rdf:type :Paper) plus all triples from the neighborhoods of these nodes v with respect to shape :WorkshopShape. Sufficiency for shape fragments will guarantee that the resulting shape fragment still validates against the schema (as we can indeed also verify in this example).

The further contents of this paper can be summarized as follows. Section 2 presents preliminaries on SHACL, defining shapes and shape schemas formally.

Section 3 motivates and defines our notion of neighborhood, and establishes sufficiency.

Section 4 develops the notion of shape fragments.

Section 5 explores how neighborhoods can be computed, either by translation to SPARQL, or by instrumenting an existing SHACL validator. We present initial experiments showing that computing neighborhoods is feasible.

Section 6 compares our work to related work on data provenance, and to a recent independent proposal, similar to shape fragments, made by Labra Gayo [36]. We also compare the expressive power of shape fragments to Triple Pattern Fragments [62], a popular existing subgraph retrieval mechanism based on single triple patterns [10, 31, 38].

Section 7 concludes the paper by discussing possible new applications and topics for further research.

Due to space limitations, some proofs have been omitted or abbreviated; a full version is available on arXiv.

2 PRELIMINARIES ON SHACL

In this section, we give self-contained definitions of shapes, their syntax and their semantics, and of shape schemas. It will be convenient here to work not with the actual SHACL syntax, but to build upon the logical syntax proposed by Corman, Reutter and Savkovic [19], which is gaining traction [3, 5, 39, 45]. We extend their proposal to cover all features of SHACL, such as disjointness, zero-or-one property paths, closedness, language tags, node tests, and literals. We have verified that our definitions given here fully cover real SHACL.

From the outset, we assume three pairwise disjoint infinite sets I, L, and B of *IRIs*, *literals*, and *blank nodes*, respectively. We use N to denote the union $I \cup B \cup L$; all elements of N are referred to as *nodes*. Literals may have a "language tag" [52]. We abstract this by assuming an equivalence relation $\sim$ on L, where $l \sim l'$ represents that l and l' have the same language tag. Moreover, we assume a strict partial order $<$ on L that abstracts comparisons between numeric values, strings, dateTime values, etc.

An *RDF triple* (s, p, o) is an element of $(I \cup B) \times I \times N$. We refer to the elements of the triple as the subject s, the property p, and the object o. An *RDF graph* G is a finite set of RDF triples. It is natural to think of an RDF graph as an edge-labeled, directed graph, viewing a triple (s, p, o) as a p-labeled edge from node s to node o.

We formalize SHACL property paths as *path expressions* E. Their syntax is given by the following grammar, where p ranges over I:

$$E ::= p \mid E^- \mid E/E \mid E \cup E \mid E^* \mid E?$$

SHACL can do many tests on individual nodes, such as testing whether a node is a literal, or testing whether an IRI matches some regular expression. We abstract this by assuming a set Ω of *node tests*; for any node test t and node a, we assume it is well-defined whether or not *a satisfies t*.

The formal syntax of *shapes* ϕ is now given by the following grammar.

$$\begin{aligned}
F &::= E \mid \mathrm{id} \\
\phi &::= \top \mid \bot \mid \mathit{hasShape}(s) \mid \mathit{test}(t) \mid \mathit{hasValue}(c) \\
&\quad \mid \mathit{eq}(F, p) \mid \mathit{disj}(F, p) \mid \mathit{closed}(P) \\
&\quad \mid \mathit{lessThan}(E, p) \mid \mathit{lessThanEq}(E, p) \mid \mathit{uniqueLang}(E) \\
&\quad \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \\
&\quad \mid \geq_n E.\phi \mid \leq_n E.\phi \mid \forall E.\phi
\end{aligned}$$

with E a path expression; $s \in I \cup B$; $t \in \Omega$; $c \in N$; $p \in I$; $P \subseteq I$ finite; and n a natural number.

Remark 2.1. In shapes of the form $\mathit{eq}(F, p)$ or $\mathit{disj}(F, p)$, the argument expression F can be either a path expression E or the keyword 'id'. We will see soon that 'id' stands for the focus node. We need to include these id-variants in order to reflect the distinction, made in the SHACL recommendation, between "node shapes" (expressing constraints on the focus node itself) and "property shapes" (expressing constraints on nodes reachable from the focus node by a path expression). □

We formalize SHACL shapes graphs as *schemas*. We first define the notion of *shape definition*, as a triple (s, ϕ, τ) where $s \in I \cup B$, and ϕ and τ are shapes. The elements of the triple are referred to as the *shape name*, the *shape expression*, and the *target expression*, respectively.[2]

Now a *schema* is a finite set H of shape definitions such that no two shape definitions have the same shape name. Moreover, as in the current SHACL recommendation, in this paper we consider only *nonrecursive* schemas. Here, H is said to be recursive if there is a directed cycle in the directed graph formed by the shape names, with an edge $s_1 \to s_2$ if $\mathit{hasShape}(s_2)$ occurs in the shape expression defining s_1.

In order to define the semantics of shapes and shape schemas, we first recall that a path expression E evaluates on an RDF

[1] The official SHACL terminology is "shapes graph" instead of shape schema.

[2] Real SHACL only supports specific shapes for targets, but our development works equally well when allowing any shape for a target.

Table 1: Conditions for conformance of a node to a shape.

ϕ	$H, G, a \models \phi$ if:
$hasValue(c)$	$a = c$
$test(t)$	a satisfies t
$hasShape(s)$	$H, G, a \models def(s, H)$
$\geq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} \geq n$
$\leq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} \leq n$
$\forall E.\psi$	every $b \in [\![E]\!]^G(a)$ satisfies $H, G, b \models \psi$
$eq(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are equal
$disj(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are disjoint
$closed(P)$	for all triples $(a, p, b) \in G$ we have $p \in P$
$lessThan(E, p)$	$b < c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$lessThanEq(E, p)$	$b \leq c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$uniqueLang(E)$	$b \nsim c$ for all $b \neq c \in [\![E]\!]^G(a)$.

graph G to a binary relation on N, denoted by $[\![E]\!]^G$ and defined as follows. $[\![p]\!]^G = \{(a, b) \mid (a, p, b) \in G\}$; $[\![E^-]\!]^G = \{(b, a) \mid (a, b) \in [\![E]\!]^G\}$; $[\![E?]\!]^G = \{(a, a) \mid a \in N\} \cup [\![E]\!]^G$; $[\![E_1 \cup E_2]\!]^G = [\![E_1]\!]^G \cup [\![E_2]\!]^G$; $[\![E_1/E_2]\!]^G = \{(a, c) \mid \exists b : (a, b) \in [\![E_1]\!]^G \ \&\ (b, c) \in [\![E_2]\!]^G\}$; and $[\![E^*]\!]^G$ = the reflexive-transitive closure of $[\![E]\!]^G$. Finally, we also define $[\![\mathrm{id}]\!]^G$, for any G, to be simply the identity relation on N.

We are now ready to define when a focus node a *conforms* to a shape ϕ in a graph G, in the context of a schema H, denoted by $H, G, a \models \phi$. For the boolean operators $\top$ (true), $\bot$ (false), $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), the definition is obvious. For the other constructs, the definition is shown in Table 1. In this table, we employ the following notations:

- $def(s, H)$ denotes the shape expression defining shape name s in H. If s does not have a definition in H, we let $def(s, H)$ be $\top$ (this is the behavior in real SHACL).
- We use the notation $R(x)$, for a binary relation R, to denote the set $\{y \mid (x, y) \in R\}$. We apply this notation to the case where R is of the form $[\![E]\!]^G$ and x is a node. For example, $[\![\mathrm{id}]\!]^G(a)$ equals the singleton $\{a\}$.
- We also use the notion $\sharp X$ for the cardinality of a set X.

Note that the conditions for $lessThan(E, p)$ and $lessThanEq(E, p)$ imply that b and c must be literals.

In Appendix A, we show that our formalization fully covers real SHACL.

Example 2.2.

- The example shape :WorkshopShape from the Introduction is expressed as follows:

$\geq_1$:author.$\geq_1$ rdf:type/rdfs:subclassOf*.$hasValue$(:Student)

- The shape $\neg disj$(:friend, :colleague) expresses that the focus node has at least one friend who is also a colleague. In real SHACL syntax, it would be expressed as

```
:HappyAtWork a sh:NodeShape ;
  sh:not [
    sh:path :friend ;
    sh:disjoint :colleague ; ] .
```

- For an IRI p, the shape $\neg disj(\mathrm{id}, p)$ expresses that the focus node has a p-labeled self-loop, and the shape $eq(\mathrm{id}, p)$ expresses that its *only* outgoing p-edge is a self-loop. □

Finally, we can define conformance of a graph to a schema as follows. RDF graph G *conforms* to schema H if for every shape definition $(s, \phi, \tau) \in H$ and for every $a \in N$ such that $H, G, a \models \tau$, we have $H, G, a \models \phi$.

Remark 2.3. Curiously, SHACL provides shapes *lessThan* and *lessThanEq* but not their variants *moreThan* and *moreThanEq* (with the obvious meaning). Note that $moreThan(E, p)$ is not equivalent to $\neg lessThanEq(E, p)$. In this paper we stay with the SHACL standard, but our treatment is easily extended to *moreThan* and *moreThanEq*.

Remark 2.4. Note that shapes are *unary* constraints: they are satisfied, or not, by any single node at a time (in the context of a graph).[3] N-ary constraints over nodes in a graph, while natural, are outside the scope of SHACL.

3 DATA PROVENANCE FOR SHACL

In this section, we propose a provenance semantics for SHACL by defining the fundamental notion of the *neighborhood* of a node v for a shape ϕ in a graph G. The intuition is that this neighborhood consists of those triples in G that show that v conforms to ϕ; if v does not conform to ϕ, the neighborhood is set to be empty. We want a generic, tractable, deterministic definition that formalizes this intuition.

3.1 Neighborhoods

Before developing the definition formally, we discuss the salient features of our approach.

Negation Following the work by Grädel and Tannen on supporting where-provenance in the presence of negation [28], we assume shapes are in *negation normal form*, i.e., negation is only applied to atomic shapes. This is no restriction, since every shape can be put in negation normal form, preserving the overall syntactic structure, simply by pushing negations down. We push negation through conjunction and disjunction using De Morgan's laws. We push negation through quantifiers as follows:

$$\neg \geq_{n+1} E.\psi \equiv \leq_n E.\psi \quad \neg \leq_n E.\psi \equiv \geq_{n+1} E.\psi \quad \neg\forall E.\psi \equiv \geq_1 E.\neg\psi$$

The negation of $\geq_0 E.\psi$ is simply false.

Node tests We leave the neighborhood for *hasValue* and *test* shapes empty, as these involve no properties, i.e., no triples.

Closedness We also define the neighborhood for $closed(P)$ to be empty, as this is a minimal subgraph in which the shape is indeed satisfied. A reasonable alternative approach would be to return all properties of the node, as "evidence" that these indeed involve only IRIs in P. Indeed, we will show in Section 3.4 that our definitions, while minimalistic, are taken such that they can be relaxed without sacrificing the sufficiency property.

Disjointness Still according to our minimal approach, the neighborhood for disjointness shapes is empty. Analogously, the same holds for *lessThan* and *uniqueLang* shapes.

Equality The neighborhood for a shape $eq(E, p)$ consists of the subgraph traced out by the E-paths and p-properties of the node under consideration, evidencing that the sets of end-nodes are indeed equal. Here, we can no longer afford to return the empty neighborhood, although equality would hold trivially there. Indeed, this would destroy the relaxation property promised above. For example, relaxing by adversely adding just one E-path and one p-property with distinct end-nodes, would no longer satisfy equality.

[3] Of course, to check satisfaction, we need to inspect properties of the node, properties of these properties, etc. Indeed our goal in this paper is to pinpoint exactly which subgraph needs to be inspected.

Nonclosure The neighborhood for a shape $\neg closed(P)$ consists of those triples from the node under consideration that involve properties outside P, as expected.

Nonequality For $\neg eq(E, p)$ we return the subgraph traced out by the E-paths from the node v under consideration that end in a node that is *not* a p-property of v, and vice versa. A similar approach is taken for nondisjointness and negated *lessThan* shapes.

Quantifiers The neighborhood for a shape $\forall E.\psi$ consists, as expected, of the subgraph traced out by all E-paths from the node under consideration to nodes x, plus the ψ-neighborhoods of these nodes x. For $\geq_n E.\psi$ we do something similar, but we take only those x that conform to ψ. Given the semantics of the $\geq_n$ quantifier, it seems tempting to instead just take a selection of n of such nodes x. However, we want a deterministic definition of neighborhood, so we take all x. Dually, for $\leq_n E.\psi$, we return the subgraph traced out by E-paths from the current node to nodes *not* conforming to ψ, plus their $\neg\psi$-neighborhoods.

3.2 Formal definition

Towards a formalization of the above ideas, we first make precise the intuitive notion of a path in an RDF graph, and of the subgraph traced out by a path. Paths are finite sequences of adjacent steps. Each step either moves forward from the subject to the object of a triple, or moves backward from the object to the subject. We make backward steps precise by introducing, for each property $p \in I$, its *reverse*, denoted by p^-. The set of reverse IRIs is denoted by I^-. We assume I and I^- are disjoint, and moreover, we also define $(p^-)^-$ to be p for every $p \in I$.

For any RDF triple $t = (s, p, o)$, the triple $t^- := (o, p^-, s)$ is called a *reverse triple*. As for IRIs, we define $(t^-)^-$ to be t. A *step* is an RDF triple (a forward step) or a reverse triple (a backward step). For any step $t = (x, r, y)$, we refer to x as the *tail*, denoted by $tail(t)$, and to y as the *head*, denoted by $head(t)$. A *path* is a nonempty finite sequence π of steps so that $head(t_1) = tail(t_2)$ for any two subsequent steps t_1 and t_2 in π. The *tail* of π is the tail of its first step; the *head* of π is the head of its last step. Any two paths π and π' where $head(\pi) = tail(\pi')$ can be concatenated; we denote this by $\pi \cdot \pi'$.

The *graph* traced out by a path π, denoted by $graph(\pi)$, is simply the set of RDF triples underlying the steps of the path. Thus, backward steps must be reversed. Formally,

$$graph(\pi) = \{t \mid t \text{ forward step in } \pi\} \cup \{t^- \mid t \text{ backward step in } \pi\}.$$

For a set Π of paths, we define $graph(\Pi) = \bigcup\{graph(\pi) \mid \pi \in \Pi\}$.

We are not interested in arbitrary sets of paths, but in the set of paths generated by a path expression E in an RDF graph G, denoted by $paths(E, G)$ and defined in a standard manner as follows. $paths(p, G) = \{(a, r, b) \in G \mid r = p\}$; $paths(E/E', G) = \{\pi \cdot \pi' \mid \pi \in paths(E, G)\ \&\ \pi' \in paths(E', G)\ \&\ tail(\pi) = head(\pi')\}$; $paths(E \cup E', G) = paths(E, G) \cup paths(E', G)$; $paths(E?, G) = paths(E, G)$; $paths(E^*, G) = \bigcup_{i=1}^{\infty} paths(E^i, G)$; and

$$paths(E^-, G) = \{\pi^- \mid \pi \in paths(E, G)\}.$$

Here, E^i abbreviates $E/\cdots/E$ (i times), and $\pi^- = t_l^-, \ldots, t_1^-$ for $\pi = t_1, \ldots, t_l$. Note that $paths(p, G)$ is a set of length-one paths.

In order to link E-paths to the evaluation of shapes below, we introduce some more notation, for any two nodes a and b:

$$paths(E, G, a, b) := \{\pi \in paths(E, G) \mid tail(\pi) = a\ \&\ head(\pi) = b\}$$

Note that $graph(\pi)$, for every $\pi \in paths(E, G)$, is a subgraph of G. This will ensure that neighborhoods and shape fragments are always subgraphs of the original graph. Moreover, the following observation ensures that path expressions will have the same semantics in the neighborhood as in the original graph:

Proposition 3.1. *Let* $F = graph(paths(E, G, a, b))$. *Then*

$$(a, b) \in [\![E]\!]^G \quad \Leftrightarrow \quad (a, b) \in [\![E]\!]^F.$$

Note that $paths(E, G)$ may be infinite, due to the use of Kleene star in E and cycles in G. However $graph(paths(E, G))$ is always finite, because G is finite.

We are now ready to define neighborhoods in the context of an arbitrary but fixed schema H. To avoid clutter we will omit H from the notation.

Definition 3.2. Let v be a node, G be a graph, and ϕ be a shape. We define the *ϕ-neighborhood of v in G*, denoted by $B(v, G, \phi)$, as the empty RDF graph whenever v does not conform to ϕ in G. When v does conform, the definition is given in Table 2.

In the table, as already discussed above, by pushing negations down, we can and do assume that ϕ is put in *negation normal form*, meaning that negation is only applied to atomic shapes. (Atomic shapes are those from the first three lines in the production for ϕ, in the grammar for shapes given in Section 2.)

Example 3.3. Recall the "happy at work" shape

$$\neg disj(\text{:friend}, \text{:colleague})$$

from Example 2.2. The neighborhood of a conforming node v consists of the union of all pairs of triples (v :friend x) and (v :colleague x) for each common x that exists in the data graph.

3.3 Algorithms for neighborhoods

Table 2 defines neighborhoods by set-theoretic expressions which are constructive, comparable to safe relational calculus formulas in the relational model [60]. As such, these expressions immediately yield a naive algorithm for computing neighborhoods.

Consider, for example, the computation of $B(v, G, \geq_n E.\psi)$. We proceed as follows, following the set-theoretic expression provided. Run through all nodes x for which there is an E-path from v to x. Algorithms for such regular path queries are well understood [9] and are supported by SPARQL query processors. For each such x, test whether $G, x \models \psi$. This test is according to the semantics of shapes defined in Table 1, which is again constructive and algorithmic. Now for each x passing the test, recursively compute $B(x, G, \psi)$, and also compute $graph(paths(E, G, v, x))$. Collect the results for all x, and return their union.

For another example, the computation of $B(v, G, \neg eq(E, p))$ proceeds as follows. Again run through all nodes x for which there is an E-path from v to x. For each x we test if (v, p, x) is in G; if it is not, we compute $graph(paths(E, G, v, x))$. We collect the resulting triples for all x in a temporary result set T_1. Secondly, we run through all nodes x for which the triple $t = (v, p, x)$ is in G. For each x we test if x is reachable from v by an E-path; if it is not, add t to the temporary result set T_2. We finally return the union $T_1 \cup T_2$.

All cases from Table 2 likewise can be given an algorithmic reading, so together they provide a (naive) algorithm for computing neighborhoods.

Table 2: Neighborhood $B(v, G, \phi)$ in the context of a schema H, when $G, v \models \phi$ and ϕ is in negation normal form. In particular, in rules 2 and 6, we assume that $\neg def(s, H)$ and $\neg\psi$ are put in negation normal form. In the omitted cases, and when $G, v \not\models \phi$, the neighborhood is defined to be empty.

ϕ	$B(v, G, \phi)$
$hasShape(s)$	$B(v, G, def(s, H))$
$\neg hasShape(s)$	$B(v, G, \neg def(s, H))$
$\phi_1 \wedge \phi_2$	$B(v, G, \phi_1) \cup B(v, G, \phi_2)$
$\phi_1 \vee \phi_2$	$B(v, G, \phi_1) \cup B(v, G, \phi_2)$
$\geq_n E.\psi$	$\bigcup\{graph(paths(E, G, v, x)) \cup B(x, G, \psi) \mid (v, x) \in [\![E]\!]^G \,\&\, G, x \models \psi\}$
$\leq_n E.\psi$	$\bigcup\{graph(paths(E, G, v, x)) \cup B(x, G, \neg\psi) \mid (v, x) \in [\![E]\!]^G \,\&\, G, x \models \neg\psi\}$
$\forall E.\psi$	$\bigcup\{graph(paths(E, G, v, x)) \cup B(x, G, \psi) \mid (v, x) \in [\![E]\!]^G\}$
$eq(E, p)$	$\bigcup\{graph(paths(E \cup p, G, v, x)) \mid (v, x) \in [\![E \cup p]\!]^G\}$
$eq(\mathrm{id}, p)$	$\{(v, p, v)\}$
$\neg eq(E, p)$	$\bigcup\{graph(paths(E, G, v, x)) \mid (v, x) \in [\![E]\!]^G \,\&\, (v, p, x) \notin G\} \cup \{(v, p, x) \in G \mid (v, x) \notin [\![E]\!]^G\}$
$\neg eq(\mathrm{id}, p)$	$\{(v, p, x) \in G \mid x \neq v\}$
$\neg disj(E, p)$	$\bigcup\{graph(paths(E, G, v, x)) \cup \{(v, p, x)\} \mid (v, x) \in [\![E]\!]^G \,\&\, (v, p, x) \in G\}$
$\neg disj(\mathrm{id}, p)$	$\{(v, p, v)\}$
$\neg lessThan(E, p)$	$\bigcup\{graph(paths(E, G, v, x)) \cup \{(v, p, y)\} \mid (v, x) \in [\![E]\!]^G \,\&\, (v, p, y) \in G \,\&\, x \not< y\}$
$\neg lessThanEq(E, p)$	$\bigcup\{graph(paths(E, G, v, x)) \cup \{(v, p, y)\} \mid (v, x) \in [\![E]\!]^G \,\&\, (v, p, y) \in G \,\&\, x \not\leq y\}$
$\neg uniqueLang(E)$	$\bigcup\{graph(paths(E, G, v, x)) \mid (v, x) \in [\![E]\!]^G \,\&\, \exists y \neq x : (v, y) \in [\![E]\!]^G \,\&\, y \sim x\}$
$\neg closed(P)$	$\{(v, p, x) \in G \mid p \notin P\}$

Computing $graph(paths(E, G, v, x))$. A key ingredient in the neighborhood algorithm is the computation of the subgraph $graph(paths(E, G, v, x))$. For simple path expressions E which are just a property p or an inverse property p^-, these are simply the singletons $\{(v, p, x)\}$ and $\{(x, p, v)\}$, respectively. For more complex path expressions E, however, it is not obvious how $graph(paths(E, G, v, x))$ can be computed. We will actually show this later in Lemma 5.1, by effectively reducing the problem to the computation of a SPARQL query.

Complexity. In Section 5.1 we will see more generally that, in fact, the entire neighborhood can be computed by a single SPARQL query. Since SPARQL (without the need for regular expressions with counting, and using standard regular path semantics) has polynomial-time data complexity [48], we obtain polynomial-time complexity of neighborhood computation.

3.4 The sufficiency property

We can prove that neighborhoods indeed provide us with an adequate provenance semantics for shapes. Specifically, we want to show that the neighborhood $B(v, G, \phi)$ is sufficient in the sense of providing provenance for the conformance of v to ϕ in G. Thinking of a shape as a unary query, returning all nodes that conform to it, the following theorem states exactly the "sufficiency property" that has been articulated in the theory of data provenance [27].

Theorem 3.4 (Sufficiency). *If $G, v \models \phi$ then also $G', v \models \phi$ for any RDF graph G' such that $B(v, G, \phi) \subseteq G' \subseteq G$.*

Proof. By induction on the structure of ϕ. Due to space limitations we only present one representative base case and one inductive case. The full proof can be found in Appendix B.

Let ϕ be of the form $eq(E, p)$. We must show that $[\![E]\!]^{G'}(v) = [\![p]\!]^{G'}(v)$. For the containment from left to right, let $x \in [\![E]\!]^{G'}(v)$. Since $G' \subseteq G$, also $x \in [\![E]\!]^G(v)$. Since $G, v \models \phi$, it follows that $x \in [\![p]\!]^G(v)$. Now let $F = graph(paths(p, G, v, x))$; note that $F \subseteq B$. By Proposition 3.1, we have $x \in [\![p]\!]^F(v)$. Then since $F \subseteq B \subseteq G'$, also also $x \in [\![p]\!]^{G'}(v)$ as desired. The containment from right to left is analogous.

Let ϕ be of the form $\leq_n E.\psi$. We begin by verifying that every $x \in [\![E]\!]^{G'}(v)$ that conforms to ψ in G' must also conform to ψ in G. For, suppose to the contrary that x would conform to $\neg\psi$ in G. Then, by definition, $B(x, G, \neg\psi) \subseteq B(v, G, \phi)$, which is itself included in G', so $B(x, G, \neg\psi) \subseteq G'$. But then by induction, and we know $B(v, G, \phi) \subseteq G'$. Therefore, by induction, x conforms to $\neg\psi$ in G', a contradiction.

Now because of the claim just verified, the number of nodes reachable from v through E that satisfy ψ in G' must be smaller or equal to the number of nodes reachable from v through E that satisfy ψ in G. Hence, since $G, v \models \phi$, certainly $G', v \models \phi$, as desired. □

Note that the Sufficiency property is stated not just for the neighborhood, but more strongly for all subgraphs G' that encompass the neighborhood. This stronger statement serves both a technical and a practical purpose. The technical purpose is that it is needed to deduce our results on shape fragments (cf. the next Subsection). The practical advantage is that it allows some leeway for provenance engines. Indeed, even if the engine, for reasons of efficiency or ease of implementation, return *larger* neighborhoods than the ones we strictly define, Sufficiency will continue to hold.

Example 3.5. Let us consider a variation to the schema from Example 1.3 in the Introduction. We now require that each paper must have at least one author, but can have at most one author who is *not* of type student. These two constraints are captured by a schema H with two shape definitions. One has the shape expression $\geq_1 \mathsf{author}.\top$, and the other has the shape expression

$$\leq_1 \mathsf{author}.\neg \geq_1 \mathsf{type}.hasValue(\mathsf{student}),$$

which in negation normal form becomes

$$\leq_1 \mathsf{author}.\leq_0 \mathsf{type}.hasValue(\mathsf{student}).$$

Both shape definitions have target $\geq_1$ type.*hasValue*(paper). We denote the two shape expressions by ϕ_1 and ϕ_2, and the target by τ.

Consider the simple graph G consisting of a single paper, say p1. This paper has two authors: Anne, who is a professor, and Bob, who is a student. Formally, G consists of the five triples (p1, type, paper), (p1, auth, Anne), (p1, auth, Bob), (Anne, type, prof) and (Bob, type, student).

Let us consider the neighborhood of p1 for the shape $\phi_1 \wedge \tau$. This neighborhood consists of the three triples (p1, type, paper), (p1, auth, Anne) and (p1, auth, Bob). On the other hand, the neighborhood of p1 for $\phi_2 \wedge \tau$ consists of the three triples (p1, type, paper), (p1, auth, Bob) and (Bob, type, student).

Note that the triple (Bob, type, student) is essential in the neighborhood for $\phi_2 \wedge \tau$; omitting it would break Sufficiency. On the other hand, we are free to add the triple (Anne, type, prof) to any of the neighborhoods without breaking Sufficiency.

Finally, note that we could add to G various other triples unrelated to the shapes ϕ_1, ϕ_2 and τ. The neighborhoods would omit all this information, as desired.

We conclude this section with a number of remarks.

Remark 3.6. A natural question is whether $B(v, G, \phi)$, as we have defined it, is *minimal* while still being sufficient in the sense of Theorem 3.4. Our discussion on quantifiers in Section 3.1 already indicated non-minimality. Assume, for example, that ϕ is "the focus node must have an a-property" (say, an address), expressed as $\geq_1 a.\top$. In a graph G with two triples (v, a, x) and (v, a, y), node v has two addresses. Any of the two triples in itself would be sufficient as a neighborhood of v for ϕ. Choosing between the two addresses x and y, however, leads to a nondeterministic behavior.

Remark 3.7. In Theorem 3.1, what can we say if v does not conform to ϕ in G? In this case, v conforms to $\neg\phi$ in G. Hence, the provenance for the non-conformance will be provided by $B(v, G, \neg\phi)$. This point was first made by Köhler, Ludäscher and Zinn [34], who, however, do not define neighborhood subgraphs and do not prove any Sufficiency property.

Remark 3.8. The neighborhood of a node, for whatever shape, is always a subset of its connected component in the graph. Thus, Sufficiency implies that only the connected component (indeed, only the neighborhood!) is needed to check conformance of a node.

4 SHAPE FRAGMENTS

In this section we define and illustrate the idea of shape fragments as a novel mechanism to retrieve subgraphs.

The *shape fragment* of an RDF graph G, for a finite set S of shapes, is the subgraph of G formed by the neighborhoods of all nodes in G for the shapes in S. Formally:

$$Frag(G, S) = \bigcup \{B(v, G, \phi) \mid v \in N \ \& \ \phi \in S\}.$$

Here, v ranges over the universe N of all nodes, but since neighborhoods are always subgraphs of G, it is equivalent to let v range over all subjects and objects of triples in G. So, to compute $Frag(G, S)$, we run over v, retrieve the neighborhoods for each v independently, and collect and return the set of resulting triples.

The shapes in S can be interpreted as arbitrary "request shapes". An interesting special case, however, is when S is derived from a shape schema H. Formally, we define the shape fragment of G for H as $Frag(G, H) := Frag(G, S)$, where $S = \{\phi \wedge \tau \mid \exists s : (s, \phi, \tau) \in H\}$. Thus, the shape fragment for a schema requests the conjunction of each shape in the schema with its associated target.

In order to state our main results concerning these two types of shape fragments, we need to revisit the definition of schema. Recall that a schema is a set of shape definitions, where a shape definition is of the form (s, ϕ, τ). Until now, we allowed both the shape expression ϕ and the target τ to be arbitrary shapes. In real SHACL, however, only shapes of the following specific forms can be used as targets:

- *hasValue*(c) (node targets);
- $\geq_1 p/r^*.$*hasValue*(c) (class-based targets: p and r stand for type and subclass from the RDF Schema vocabulary [52], and c is the class name);
- $\geq_1 p.\top$ (subjects-of targets); and
- $\geq_1 p^-.\top$ (objects-of targets).

For our purposes, however, what counts is that real SHACL targets τ are *monotone*, in the sense that if $G, v \models \tau$ and $G \subseteq G'$, then also $G', v \models \tau$.

We establish:

Theorem 4.1 (Conformance). *Assume that schema H has monotone targets, and assume RDF graph G conforms to H. Then $Frag(G, H)$ also conforms to H.*

The proof is a straightforward application of Theorem 3.4. Moreover, Sufficiency carries over to shape fragments defined by arbitrary request shapes as follows:

Corollary 4.2. *Let G be an RDF graph, let S be a finite set of shapes, let ϕ be a shape in S, and let v be a node. If $G, v \models \phi$, then also $Frag(G, S), v \models \phi$.*

Example 4.3. For monotone shapes, the converse of Corollary 4.2 clearly holds as well. In general, however, the converse does not always hold. For example, consider the shape $\phi = \leq_0 p.\top$ ("the node has no property p"), and the graph $G = \{(a, p, b)\}$. Then the fragment $Frag(G, \{\phi\})$ is empty, so a trivially conforms to ϕ in the fragment. However, a clearly does not conform to ϕ in G.

Draft specification. We have defined a complete specification of shape fragments which closely follows the existing W3C SHACL recommendation. Our specification explains in detail how each construct of core SHACL contributes to the formation of the shape fragment [55].

4.1 Applicability of shape fragments

In order to assess the practical applicability of shape fragments, we simulated a range of SPARQL queries by shape fragments. Queries were taken from the SPARQL benchmarks BSBM [13] and WatDiv [4]. Unlike a shape fragment, a SPARQL select-query does not return a subgraph but a set of variable bindings. SPARQL construct-queries do return RDF graphs directly, but not necessarily subgraphs. Hence, we followed the methodology of modifying SPARQL select-queries to construct-queries that return all *images* of the pattern specified in the where-clause.

For tree-shaped basic graph patterns, with given IRIs in the predicate position of triple patterns, we can always simulate the corresponding subgraph query by a shape fragment. Indeed, a typical query from the benchmarks retrieves nodes with some specified properties, some properties of these properties, and so on. For example, a slightly simplified WatDiv query, modified

into a subgraph query, would be the following. (To avoid clutter, we forgo the rules of standard IRI syntax.)

```
CONSTRUCT WHERE {
?v0 caption ?v1 . ?v0 hasReview ?v2 . ?v2 title ?v3 .
?v2 reviewer ?v6 . ?v7 actor ?v6 }
```

(Here, CONSTRUCT WHERE is the SPARQL notation for returning all images of a basic graph pattern.) We can express the above query as the fragment for the following request shape:

$$\geq_1 \text{caption}.\top \wedge \geq_1 \text{hasReview}.(\geq_1 \text{title}.\top \wedge \geq_1 \text{reviewer}.\geq_1 \text{actor}^-.\top)$$

Of course, patterns can involve various SPARQL operators, going beyond basic graph patterns. Filter conditions on property values can be expressed as node tests in shapes; optional matching can be expressed using $\geq_0$ quantifiers. For example, consider a simplified version of the pattern of a typical BSBM query:

```
?v text ?t . FILTER langMatches(lang(?t),"EN")
OPTIONAL { ?v rating ?r }
```

The images of this pattern can be retrieved using the shape

$$\geq_1 \text{title}.\mathit{test}(\text{lang} = \text{“EN”}) \wedge \geq_0 \text{rating}.\top.$$

Interestingly, the BSBM workload includes a pattern involving a combination of optional matching and a negated bound-condition to express absence of a certain property (a well-known trick [6, 7]). Simplified, this pattern looks as follows:

```
?prod label ?lab . ?prod feature 870
OPTIONAL { ?prod feature 59 . ?prod label ?var }
FILTER (!bound(?var))
```

The images of this pattern can be retrieved using the shape

$$\geq_1 \text{label}.\top \wedge \geq_1 \text{feature}.\mathit{hasValue}(870) \wedge \leq_0 \text{feature}.\mathit{hasValue}(59).$$

A total of 39 out of 46 benchmark queries, modified to return subgraphs, could be simulated by shape fragments in this manner. The remaining seven queries involved features not supported by SHACL, notably, variables in the property position, or arithmetic.

5 IMPLEMENTATION AND EXPERIMENTAL VALIDATION

In this section we show that neighborhoods can be effectively computed, and report on initial experiments.

5.1 Translation to SPARQL

Our first approach to computing neighborhoods is by translation into SPARQL, the recommended query language for RDF graphs [30]. SPARQL select-queries return sets of *solution mappings*, which are maps μ from finite sets of variables to N. Variables are marked using question marks. Different mappings in the result may have different domains [8, 47].

Neighborhoods in an RDF graph G are unions of subgraphs of the form $\mathit{graph}(\mathit{paths}(E, G, a, b))$, for path expressions E mentioned in the shapes, and selected nodes a and b. Hence, the following lemma is important. For any RDF graph G, we denote by $N(G)$ the set of all subjects and objects of triples in G.

Lemma 5.1. *For every path expression E, there exists a SPARQL select-query $Q_E(?t, ?s, ?p, ?o, ?h)$ such that for every RDF graph G:*

1. *The binary relation $\{(\mu(?t), \mu(?h)) \mid \mu \in Q_E(G)\}$ equals $[\![E]\!]^G$, restricted to $N(G)$.*
2. *For all $a, b \in N(G)$, the RDF graph*

$$\{(\mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_E(G) \;\&\; (\mu(?t), \mu(?h)) = (a, b) \;\&\; \mu \text{ is defined on } ?s, ?p \text{ and } ?o\}$$

equals $\mathit{graph}(\mathit{paths}(E, G, a, b))$.

Proof. The difficulty is that we do not merely have to test if $(a, b) \in [\![E]\!]^G$, which is readily expressed using SPARQL property paths, but that we actually have to return $\mathit{graph}(\mathit{paths}(E, G, a, b))$. We construct Q_E by induction on the structure of E. We list Q_E in each of the cases of the syntax of path expressions. In the base case, when E is a property name p:

```
SELECT (?s AS ?t) ?s (p AS ?p) ?o (?o AS ?h) WHERE { ?s p ?o }
```

When E is of the form E_1^-, we obtain Q_{E_1} by induction, and construct Q_E as follows:

```
SELECT (?h AS ?t) ?s ?p ?o (?t AS ?h) WHERE { Q_E1 }
```

When E is of the form $E_1?$:

```
SELECT ?t ?s ?p ?o ?h WHERE {
{ Q_E1 } UNION
{ SELECT (?v AS ?t) (?v AS ?h)
  WHERE { { ?v ?_p1 ?_o1 } UNION { ?_s2 ?_p2 ?v } } } }
```

When E is of the form $E_1 \cup E_2$:

```
SELECT ?t ?s ?p ?o ?h WHERE { { Q_E1 } UNION { Q_E2 } }
```

When E is of the form E_1/E_2:

```
SELECT ?t ?s ?p ?o ?h WHERE {
{ { SELECT ?t ?s ?p ?o (?h AS ?h1) WHERE { Q_E1 } } .
  { SELECT (?t AS ?h1) ?h WHERE { ?t E2 ?h } } }
UNION {
{ SELECT ?t (?h AS ?h1) WHERE { ?t E1 ?h } } .
{ SELECT (?t AS ?h1) ?s ?p ?o ?h WHERE { Q_E2 } } } }
```

Finally, when E is of the form E_1^*:

```
SELECT ?t ?s ?p ?o ?h WHERE {
{ ?t E1* ?x1 . ?x2 E1* ?h .
  { SELECT (?t AS ?x1) ?s ?p ?o (?h AS ?x2) WHERE { Q_E1 } } }
UNION {
  SELECT (?v AS ?h) (?v AS ?t)
  WHERE { { ?v ?_p1 ?_o1 } UNION { ?_s2 ?_p2 ?v } } } }
```

□

The following example illustrates the lemma, but using a more readable query than the one that would be literally generated by the above proof.

Example 5.2. For IRIs a, b, q and r, the following SPARQL query, applied to any graph G, returns $\mathit{graph}(\mathit{paths}((q/r)^*, G, a, b))$:

```
SELECT ?s ?p ?o
WHERE { a (q/r)* ?t . ?h (q/r)* b . {
    { SELECT ?t (?t AS ?s) (q AS ?p) ?o ?h
      WHERE { ?t q ?o . ?o r ?h } }
    UNION
    { SELECT ?t ?s (r AS ?p) (?h AS ?o) ?h
      WHERE { ?t q ?s . ?s r ?h } } } }
```

□

Using Lemma 5.1, and expressing the definitions from Table 2 in SPARQL, we obtain that neighborhoods can be uniformly computed in SPARQL as follows.

Proposition 5.3. *For every shape ϕ, there exists a SPARQL select-query $Q_\phi(?v, ?s, ?p, ?o)$ such that for every RDF graph G,*

$$\{(\mu(?v), \mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_\phi(G)\} = \{(v, s, p, o) \in N^4 \mid (s, p, o) \in B(v, G, \phi)\}$$

Moreover, the size of Q_ϕ is linear in the size of ϕ.

The proof of Proposition 5.3 is given in Appendix C.1. The linear-size claim can indeed be verified by inspecting the construction in the proof of Lemma 5.1 and the construction of Q_ϕ.

Remark 5.4. The above result should not be confused with the known result [18, Proposition 3] that SPARQL can compute the set of nodes that *conform* to a given shape. Our result states that also the neighborhoods can be computed. □

Since shape fragments are unions of neighborhoods, we also obtain:

Corollary 5.5. *For every finite set S of shapes, there exists a SPARQL select-query $Q_S(?s, ?p, ?o)$ such that for every RDF graph G,*

$$\{(\mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_S(G)\} = Frag(G, S).$$

Example 5.6. For IRIs p, q and c, consider the request shape $\forall p.\geq_1 q.hasValue(c)$ (e.g., "all my friends like ping-pong", with p, q and c playing the role of friend, like, and ping-pong, respectively). The corresponding shape fragment is retrieved by the following SPARQL query:

```
SELECT ?s ?p ?o WHERE {
{ SELECT ?v WHERE
  { ?v p ?x MINUS { ?v p ?y OPTIONAL { ?y q c . ?v p ?z }
                    FILTER (!bound(?z)) } } } .
{ { SELECT (?v AS ?s) (p AS ?p) (?x as ?o)
    WHERE { ?v p ?x . ?x q c } }
  UNION
  { SELECT (?x AS ?s) (q AS ?p) (c as ?o)
    WHERE { ?v p ?x . ?x q c } } } }
```

The first subselect retrieves nodes $?v$ conforming to the shape; the UNION of the next two subselects then retrieves the neighborhoods. □

The above example illustrates that query expressions for shapes can quickly become quite complex, even for just retrieving the nodes that conform to a shape. Shapes involving equality constraints require nested not-exists subqueries in SPARQL, and would benefit from specific operators for set joins, e.g., [32, 41]. Shapes of the form $\leq_5 p.\top$ requires grouping the p-properties and applying a condition count ≤ 5, plus a union with an outer join to retrieve the nodes without any p-property. Such shapes would benefit from specific operators for group join [22, 42]. Query optimization for queries derived from SHACL is an important topic for further research.

One may wonder about the converse to Corollary 5.5: is every SPARQL select-query expressible as a shape fragment? This does not hold, if only because shape fragments always consist of triples from the input graph, while select-queries can return arbitrary variable bindings. However, also more fundamentally, SHACL is strictly weaker than SPARQL; we give two representative examples.

4-clique Let $p \in I$. There does not exist a shape ϕ such that, on any RDF graph G, the nodes that conform to ϕ are exactly the nodes belonging to a 4-clique of p-triples in G. We can show that if 4-clique would be expressible by a shape, then the corresponding 4-clique query about a binary relation P would be expressible in 3-variable counting infinitary logic $C^3_{\infty\omega}$. The latter is known not to be the case, however [43]. (Infinitary logic is needed here to express path expressions, and counting is needed for the $\geq_n$ quantifier, since we have only 3 variables.)

Majority Let $p, q \in I$. There does not exist a shape ϕ such that, on any RDF graph G, the nodes that conform to ϕ are exactly the nodes v such that $\sharp\{x \mid (v, p, x) \in G\} \geq \sharp\{x \mid (v, q, x) \in G\}$ (think of departments with at least as many employees as projects). We can show that if Majority would be expressible by a shape, then the classical Majority query about two unary relations P and Q would be expressible in first-order logic. Again, the latter is not the case [35]. (Infinitary logic is not needed here, since for this query, we can restrict to a class of structures where all paths have length one.)

5.2 Adapting a validation engine

We have also investigated computing neighborhoods by adjusting a SHACL validator to return the validated RDF terms and their neighborhood, instead of a validation report.

A SHACL validation engine checks whether a given RDF graph conforms to a given schema, and produces a validation report detailing possible violations. A validation engine needs to inspect the neighborhoods of nodes anyway. Hence, it requires only reasonably lightweight adaptations to produce, in addition to the validation report, also the nodes and their neighborhoods that validate the shapes graph, without introducing significant overheads for tracing out and returning these neighborhoods, compared to doing validation alone. Our hypothesis is that the resulting overhead will not be prohibitive.

To test this hypothesis, we extended the open-source, free-license engine pySHACL [49]. This is a main-memory engine and it achieves high coverage for core SHACL [25]. Written in Python, we found it easy to make local changes to the code [46]; starting out with 4501 lines of code, 482 lines were changed, added or deleted.

Our current implementation covers most of SHACL core, with the exception of complex path expressions, i.e., only simple path expressions are supported. The algorithm that is implemented is then essentially the naive algorithm described in Section 3.3.

Our software, called *pySHACL-fragments*, is available open-source [50].

5.3 Experiments

We validated our approach by (i) measuring the overhead of neighborhood extraction, compared to mere validation, using pySHACL-fragments; and (ii) testing the viability of computing neighborhoods by translation to SPARQL. We perform our experiments in the context of computing shape fragments. Indeed, shape fragments offer a natural test case as they require the neighborhoods of all nodes to be retrieved.

5.3.1 Extraction overhead. To evaluate the viability of computing neighborhoods by adapting a validation engine, we measured the overhead of extracting neighborhoods using our system pySHACL-fragments, compared to producing the corresponding validation report using pySHACL alone. Performance evaluation of SHACL engines is not our goal here; see Schaffenrath et al. [53] for this. Yet, we reuse the 57 shapes from their performance benchmark. These shapes are expressed over a 30-million triple dataset

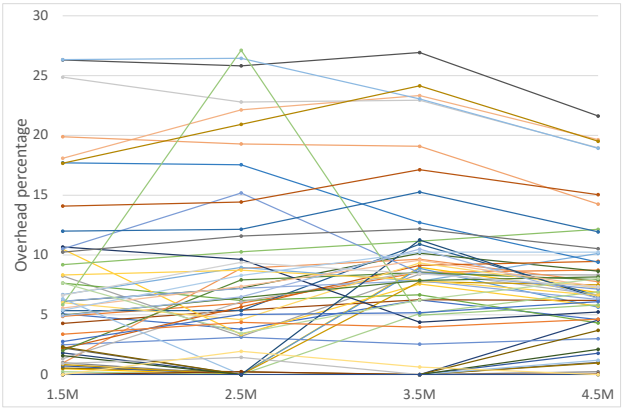

Figure 1: Overhead (percent increase in time to do provenance extraction, over mere validation of a shape) shown for 57 shapes over four graph sizes. Each line represents a shape.

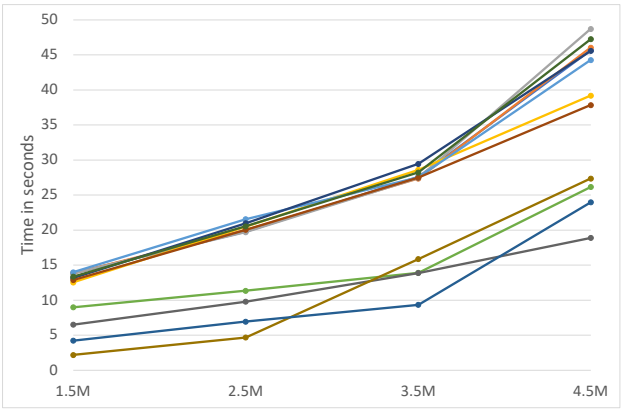

Figure 2: Execution times of provenance computation for 12 shapes by SPARQL queries, over four graph sizes. Each line represents a shape.

known as the "Tyrolean Knowledge Graph". Notably, however, Schaffenrath et al. managed to run their comparative study on a 1-million slice of the knowledge graph only, as common SHACL validation engines are still in their infancy and not very efficient.

For our experiment, instead, we generated a 1.5-million triple induced subgraph of the knowledge graph as follows. We sampled 50 000 individual nodes uniformly at random, and then retrieved all triples involving these individuals as subjects or objects. By sampling a larger number of 100K, 150K and 200K nodes, we similarly obtained subgraphs of (approximately) 2.5, 3.5, and 4.5 million triples.

We used a 12 core AMD EPYC 2.595GHz processor with 16GB DDR4 RAM and 400GB SSD. We executed each of the shapes three times, both on pySHACL and on pySHACL-fragments. Timers were placed around the validator.run() function, so *data loading and shape parsing time is not included.* The average overhead turns out to be well below 10%; if we restrict attention to the shapes where validation on the 1.5M graph takes longer than a second, the average overhead grows to 15.6%. Figure 1 shows that the overhead may vary somewhat going to larger graphs, but stays constant on average. There are some outliers where the overhead fluctuates more wildly, but these happen to be associated with low (below second) runtimes.

The shapes where the overhead is highest are those with existential shapes and many target nodes with large neighborhoods. For some property p and some condition ψ, an existential shape requires that the target node must have at least one p-edge to a node x satisfying ψ (expressed as $\geq_1 p.\psi$). Here, a validator merely needs to check for each target node v that at least one such x exists, while provenance computation must also retrieve all the satisfying triples (v, p, x).

5.3.2 Computing neighborhoods in SPARQL. Instead of modifying an existing SHACL engine, one may compute provenance using SPARQL queries, as presented in Section 5.1. Shapes give rise to complex SPARQL queries which pose quite a challenge to SPARQL query processors. It is outside the scope of the present paper to do a performance study of SPARQL query processors; our goal rather is to obtain an indication of the practical feasibility of computing neighborhoods in SPARQL.

Initial work by Corman et al. has reported satisfying results on doing *validation* for nonrecursive schemas by a single, complex SPARQL query [18]. The question we want to answer is whether we can observe a similar situation when computing neighborhoods, where the queries become even more complex.

We have obtained a mixed picture. We used the main-memory SPARQL engine Apache Jena ARQ. Implementing Corollary 5.5 by following the constructive proof of Proposition 5.3, we translated the shape fragment queries for the benchmark shapes from the previous Section 5.3.1 into large SPARQL queries. The generated expressions can be hundreds of lines long, as our translation procedure is not yet optimized to generate "efficient" SPARQL expressions. However, we then reduced the shapes by substituting $\top$ for node tests, and simplifying the resulting expressions. This reduction preserves the graph-navigational nature of the queries.

After the reduction, 13 out of 57 shapes produced SPARQL queries that ARQ could execute. The other queries were still too long and did not terminate or went out of memory. Figure 2 shows the runtimes on the same test data and the same machine as the overhead experiment; one shape is omitted from the Figure as it does not retrieve any triples at all. Reported timings are averages over three runs.

Finally, to test the extraction of paths in SPARQL, we used the DBLP database [21], and computed the shape fragment for shape $\geq_1 a^-/a/a^-/a/a^-/a.hasValue(\text{MYV})$, where a stands for the property dblp:authoredBy, and MYV stands for the DBLP IRI for Moshe Y. Vardi. This extracts not only all authors at co-author distance three or less from this famous computer scientist, but, crucially, also all a-triples on all the relevant paths. The generated SPARQL query is similar to the query from Example 5.2.

We ran this heavy analytical query on the two secondary-memory engines Apache Jena ARQ on TDB2 store, and GraphDB. The execution times over increasing slices of DBLP, going backwards in time from 2021 until 2010, are comparable between the two engines (see Figure 3). Vardi is a prolific and central author and co-author; just from 2016 until 2021, almost 7% of all DBLP authors are at distance three or less, or almost 145 943 authors. The resulting shape fragment contains almost 3% of all dblpl:authoredBy triples, or 219 085 unique triples. We see that retrieving neighborhoods can be a computationally intensive task for which new methods may be needed.

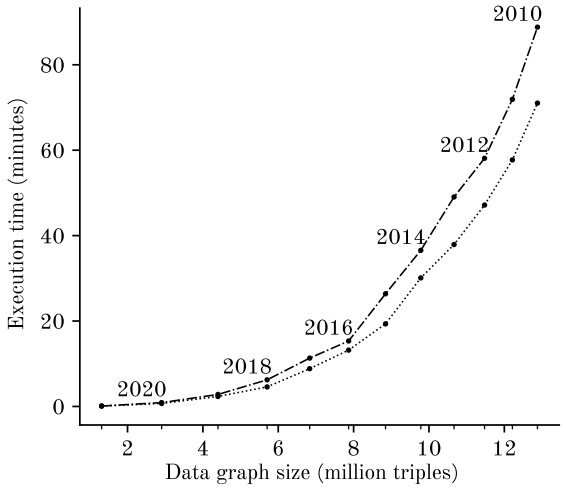

Figure 3: Jena ARQ store-based SPARQL execution time (dotted) and store-based GraphDB execution time (dashed-dotted) for the Vardi-distance-3 shape fragment.

For the Vardi experiment we used a 2x 8core Intel Xeon E5-2650 v2 processor with 48GB DDR3 RAM and a 250GB hard disk.

5.3.3 Discussion. From these experiments, we conclude that computing neighborhoods is viable, but can be computationally intensive. Indeed, provenance for SHACL serves as an interesting new challenge and testbed both for SHACL validators (suitably adapted to retrieve neighborhoods) and SPARQL engines. Advances on either front will also benefit SHACL provenance performance. Interestingly, recent approaches to SHACL validation [18, 23] consider decomposing the task into multiple small SPARQL queries, as opposed to translating to a single large query.

6 RELATED WORK

Shapes may be viewed as queries on RDF graphs, returning the nodes that conform to the shape. This observation allows us to compare neighborhoods for shapes, with provenance semantics for queries proposed in the literature [17, 27].

A seminal work in the area of data provenance is that on *lineage* by Cui, Widom and Wiener [20]. Like neighborhoods, the lineage of a tuple returned by a query on a database D is a subdatabase of D. Lineage was defined for queries expressed in the relational algebra. In principle, we can express shapes in relational algebra. So, instead of defining our own notion of neighborhood, should we have simply used lineage instead? The answer is no; the following example shows that Sufficiency would fail.

Example 6.1. Recalling Example 3.5, consider a relational database schema with three relation schemes Paper(P), Author(P, A), and Student(A), and the query Q returning all papers with at least one author but without non-student authors. Consider the database D given by

$$D(\text{Paper}) = \{\text{p1}\};$$
$$D(\text{Author}) = \{(\text{p1}, \text{Bob})\};$$
$$D(\text{Student}) = \{\text{Bob}\}.$$

Note that p1 is returned by Q on D. A relational algebra expression for Q is $E = \text{Paper} \bowtie (\pi_P(\text{Author}) - V)$ with $V = \pi_P(\text{Author} - (\text{Author} \bowtie \text{Student}))$. Since V is empty on D, the lineage of p1 for E in D is the database D' where

$D'(\text{Paper}) = \{\text{p1}\}$; $D'(\text{Author}) = \{(\text{p1}, \text{Bob})\}$; $D'(\text{Student}) = \emptyset$.

However, p1 is no longer returned by E on D. □

An alternative approach to lineage is *why-provenance* [16]. This approach is non-deterministic in that it reflects that there may be several "explanations" for why a tuple is returned by a query (for example, queries involving existential quantification). Accordingly, why-provenance does not yield a single neighborhood (called witness), but a set of them. While logical, this approach is at odds with our aim of providing a *deterministic* retrieval mechanism through shapes. Of course, one could take the union of all witnesses, but this runs into similar problems as illustrated in the above example. Indeed, why-provenance was not developed for queries involving negation or universal quantification.

A recent approach to provenance for negation is that by Grädel and Tannen [28, 59] based on the successful framework of provenance semirings [29]. There, provenance is produced in the form of provenance polynomials which give a compact representation of the several possible proof trees showing that the tuple satisfies the query. Thus, like why-provenance, this approach is inherently non-deterministic. Still, we were influenced by Grädel and Tannen's use of negation normal form, which we have followed in this work.

6.1 Triple pattern fragments

Shape fragments return subgraphs: they retrieve a subset of the triples of an input graph. A popular subgraph-returning mechanism is that of *triple pattern fragments* (TPF [62]). A TPF may indeed be viewed as a query that, on an input graph G, returns the subset of G consisting of all images of some fixed triple pattern in G.

While the logic of shapes is, in general, much richer than simple triple patterns, it turns out that not all TPFs are actually expressible by shape fragments.

For example, TPFs of the form (c, p, d), $(c, p, ?x)$, $(?x, p, c)$, or $(?x, p, ?y)$, for IRIs p, c, and d, are easily expressed as shape fragments using request shapes $hasValue(c) \wedge \geq_1 p.hasValue(d)$, $\geq_1 p^-.hasValue(c)$, $\geq_1 p.hasValue(c)$, or $\geq_1 p.\top$, respectively.

The TPF $(?x, p, ?x)$, asking for all p-self-loops in the graph, corresponds to the shape fragment for $\neg disj(\text{id}, p)$.

Furthermore, the TPFs $(?x, ?y, ?z)$ (requesting a full download) and $(c, ?y, ?z)$ are expressible using the request shapes $\neg closed(\emptyset)$ and $hasValue(c) \wedge \neg closed(\emptyset)$. Here, the need to use a "trick" via negation of closedness constraints exposes a weakness of shapes: properties are not treated on equal footing as subjects and objects. Indeed, other TPFs involving variable properties, such as $(?x, ?y, c)$, $(?x, ?y, ?x)$, or $(c, ?x, d)$, are not expressible as shape fragments.

The above discussion can be summarized as follows.

Proposition 6.2. *The TPFs expressible as a shape fragment (uniformly over all input graphs) are precisely the TPFs of the following forms:*

(1) $(?x, p, ?y)$;
(2) $(?x, p, c)$;
(3) $(c, p, ?x)$;
(4) (c, p, d);

(5) $(?x, p, ?x)$;
(6) $(?x, ?y, ?z)$;
(7) $(c, ?y, ?z)$.

The proof of Proposition 6.2 is given in Appendix D.

Remark 6.3. SHACL does not allow *negated properties* in path expressions, while these are supported in SPARQL property paths. Extending SHACL with negated properties would readily allow the expression of *all* TPFs as shape fragments. For example, the TPF $(?x, ?y, c)$, for IRI c, would become expressible by requesting the shape

$$\geq_1 p.hasValue(c) \vee \geq_1 !p.hasValue(c),$$

with p an arbitrary IRI. Here, the negated property $!p$ matches any property different from p.

6.2 Knowledge graph subsets

Recently, the idea of defining subgraphs (or fragments as we call them) using shapes was independently proposed by Labra Gayo [36]. An important difference with our SHACL-based approach is that his approach is based on ShEx, the other shape language besides SHACL that is popular in practice [15, 26]. Shapes in ShEx are quite different from those in SHACL, being based on bag-regular expressions over the bag of properties of the focus node. As a result, the technical developments of our work and Labra Gayo's are quite different. Still, the intuitive and natural idea of forming a subgraph by collecting all triples encountered during conformance checking, is clearly the same in both approaches. This idea, which Labra Gayo calls "slurping", is implemented in our pyshacl-fragments implementation, as well as a "slurp" option in the shex.js implementation of ShEx [56]. Labra Gayo also gives a formal definition of ShEx + slurp, extending the formal definition of ShEx [15].

In our work we make several additional contributions compared to the development by Labra Gayo:

- We make the connection to database provenance.
- We consider the important special case of shape fragments based on schemas with targets.
- We support path expressions directly, which in ShEx need to be expressed through recursion.
- We support negation, universal quantification, and other non-monotone quantifiers and shapes, such as $\leq_n$, equality, disjointness, lessThan.
- We establish formal correctness properties (Sufficiency and Conformance Theorems).
- We investigate the translation of shape fragments into SPARQL. On the other hand, Labra Gayo discusses Pregel-based implementations of his query mechanism.

6.3 Path-returning queries on graph databases

Our definition of neighborhood of a node v for a shape involving a path expression E returns E-paths from v to relevant nodes x (see Table 2). Notably, these paths are returned as a subgraph, using the *graph* constructor applied to a set of paths. Thus, shape fragments are loosely related to path-returning queries on graph databases, introduced as a theoretical concept by Barceló et al. [11] and found in the languages Cypher [24] and G-CORE [37].

However, to our knowledge, a mechanism to return a set of paths in the form of a subgraph is not yet implemented by these languages. We have showed in Section 5.1 that, at least in principle, this is actually possible in any standard query language supporting path expressions, such as SPARQL. Barceló et al. consider a richer output structure whereby an infinite set of paths (or even set of tuples of paths) resulting from an extended regular path query can be finitely and losslessly represented. In contrast, our *graph* constructor is lossy in that two different sets S_1 and S_2 of paths may have $graph(S_1) = graph(S_2)$. However, our Sufficiency property shows that our representation is sufficient for the purpose of validating shapes.

7 CONCLUDING REMARKS

In this paper we have proposed a provenance semantics for SHACL, which was long overdue. In addition to the desirability of supporting provenance from a general database perspective, the utility of a provenance semantics for shapes to support data footprint in Linked Data applications has been pointed out informally by prominent Semantic Web researchers [12, 61]. Moreover, the idea of using shapes to *describe* nodes in a graph has been floating around in the community [58]. Also, the SHACL Recommendation itself anticipates applications for shapes beyond conformance checking. Our work serves to put these ideas on a firm formal footing.

Our notions of shape fragment, developed in Section 4, serve to open up SHACL: initially conceived as a constraint or data validation language, it can now also serve as a data retrieval language. If shapes are available, either in a schema coming from the producer of the data, or as an expression of an application's interest in certain types of information, they can now be used to retrieve data. In such settings we avoid the need to switch to a separate retrieval language, typically SPARQL in this context.

A more specific application made possible by shape fragments is retrieval from data that was populated through forms. For example, in Schímatos [63], Web forms are compiled from shapes. It is conceivable that all data that was entered based on some given shape can be retrieved back as a shape fragment, using that shape as the request shape.

SHACL is a quite powerful language, so an obvious direction for further research is to investigate efficient processing and optimization strategies for SHACL, both just for validation, and for computing provenance. Recent work on validation optimization was done by Figuera et al. [23]. Yet we believe many more insights from database query optimization can be beneficial and specialized to shape processing. (A related direction is to use shapes to inform SPARQL query optimization [2, 51].)

We have seen that shape fragments are strictly less expressive than SPARQL subgraph queries. The relationship between SHACL and SPARQL deserves much further study. Is the complexity of evaluation lower? For those SPARQL subgraph queries that *are* expressible as shape fragments, are queries in practice often easier to write in SHACL? Can we precisely characterise the expressive power of SHACL?

Our approach to defining neighborhoods has been to be deterministic, and to satisfy Sufficiency, while also omitting needless triples, i.e., trying to be minimal. However, as discussed by Glavic [27, Section 2.1], minimality is a requirement for provenance semantics that is challenging, and sometimes impossible to achieve together with determinism and Sufficiency. See also our Remark 3.6 in Section 3.4. Developing good postulates for minimality in provenance notions is an important topic for further research.

Another obvious direction for further research is to extend our work to recursive schemas. The SHACL recommendation

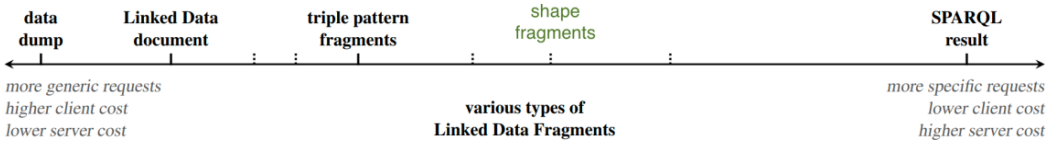

Figure 4: Positioning shape fragments in the LDF Framework (adapted from [62]). This diagram is not to be interpreted as a comparison in expressive power.

only defines the semantics for nonrecursive shape schemas, and we have seen in this paper that defining provenance is already nontrivial for this case. Nevertheless, there is current interest in recursive shape schemas [5, 14, 15, 18, 26].

A final open problem that we mention is to extend shapes so that properties are treated on equal footing as subjects and objects, as is indeed the spirit of RDF [40, 48].

To conclude, we mention that shape fragments fit the framework of Linked Data Fragments [10, 31, 38, 62] (LDF) for publication interfaces to retrieve RDF (sub)graphs. At one end of the spectrum the complete RDF graph is retrieved; at the other end, the results of arbitrary SPARQL queries. Triple Pattern Fragments (TPF) [62], compare Section 6.1, represent an intermediate point where all triples from the graph that match a given SPARQL triple pattern are returned. On this spectrum, shape fragments lie between TPF and arbitrary SPARQL, as depicted in Fig. 4, taking advantage of the merits of both approaches. On the one hand, shape fragments may reduce the server cost, similarly to TPF, but they can also perform fewer requests as multiple TPFs can be expressed as a single shape fragment. On the other hand, shape fragments may also perform quite powerful requests, similarly to SPARQL endpoints, but without reaching the full expressivity of SPARQL.

ACKNOWLEDGMENTS

We are indebted to Bart Bogaerts for helpful suggestions and discussions. This research was supported by the Flanders AI Research programme.

REFERENCES

[1] Daniel Abadi et al. 2019. The Seattle report on database research. *SIGMOD Record* 48, 4 (2019), 44–53.
[2] Abdullah Abbas, Pierre Genevès, Cécile Roisin, and Nabil Layaïda. 2018. Selectivity estimation for SPARQL triple patterns with shape expressions. In *Proceedings 18th International Conference on Web Engineering (Lecture Notes in Computer Science, Vol. 10845)*, T. Mikkonen et al. (Eds.). Springer, 195–209.
[3] Shqiponja Ahmetaj, Robert David, Magdalena Ortiz, Axel Polleres, Bojken Shehu, and Mantas Simkus. 2021. Reasoning about explanations for non-validation in SHACL. In *Proceedings 18th International Conference on Principles of Knowledge Representation and Reasoning*, M. Bienvenu, G. Lakemeyer, et al. (Eds.). IJCAI Organization, 12–21.
[4] Güneş Aluç, Olaf Hartig, Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *Proceedings 13th International Semantic Web Conference (Lecture Notes in Computer Science, Vol. 8796)*, P. Mika, T. Tudorache, et al. (Eds.). Springer, 197–212.
[5] Medina Andreşel, Julien Corman, Magdalena Ortiz, Juan L. Reutter, Ognjen Savkovic, and Mantas Simkus. 2020. Stable model semantics for recursive SHACL, See [33], 1570–1580.
[6] Renzo Angles and Claudio Gutierrez. 2008. The expressive power of SPARQL. In *Proceedings 7th International Semantic Web Conference (Lecture Notes in Computer Science, Vol. 5318)*, A. Sheth, S. Staab, et al. (Eds.). Springer, 114–129.
[7] Marcelo Arenas and Jorge Pérez. 2011. Querying semantic web data with SPARQL. In *Proceedings 30st ACM Symposium on Principles of Databases*. ACM, 305–316.
[8] Marcelo Arenas, Jorge Pérez, and Claudio Gutierrez. 2009. On the semantics of SPARQL. In *Semantic Web Information Management—A Model-Based Perspective*, R. De Virgilio, F. Giunchiglia, and L. Tanca (Eds.). Springer, 281–307.
[9] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time- and space-efficient regular path queries. In *Proceedings 38th International Conference on Data Engineering*. IEEE, 3091–3105.
[10] Amr Azzam, Javier D. Fernández, et al. 2020. SMART-KG: Hybrid shipping for SPARQL querying on the Web, See [33], 984–994.
[11] Pablo Barceló, Carlos A. Hurtado, Leonid Libkin, and Peter T. Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems* 37, 4 (2012), 31:1–31:46.
[12] Tim Berners-Lee. 2019. Linked data shapes, forms and footprints. https://www.w3.org/DesignIssues/Footprints.html.
[13] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems* 5, 2 (2009), 1–24.
[14] Bart Bogaerts and Maxime Jakubowski. 2021. Fixpoint semantics for recursive SHACL. In *Proceedings 37th International Conference on Logic Programming (Technical Communications) (Electronic Proceedings in Theoretical Computer Science, Vol. 345)*, A. Formisano, Y.A. Liu, et al. (Eds.). 41–47.
[15] Iovka Boneva, Jose E. Labra Gayo, and Eric G. Prud'hommeaux. 2017. Semantics and validation of shape schemas for RDF. In *Proceedings 16th International Semantic Web Conference (Lecture Notes in Computer Science, Vol. 10587)*, Claudia d'Amato, Miriam Fernandez, Valentina Tamma, et al. (Eds.). Springer, 104–120.
[16] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001 (Lecture Notes in Computer Science, Vol. 1973)*, J. Van den Bussche and V. Vianu (Eds.). Springer, 316–330.
[17] James Cheney, Laura Chiticarius, and Wang-Chiew Tan. 2009. Provenance in Databases: why, how and where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.
[18] Julien Corman, Fernando Florenzano, Juan L. Reutter, and Ognjen Savkovic. 2019. Validating SHACL constraints over a SPARQL endpoint. In *Proceedings 18th International Semantic Web Conference (Lecture Notes in Computer Science, Vol. 11778)*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, et al. (Eds.). Springer, 145–163.
[19] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. 2018. Semantics and validation of recursive SHACL. In *Proceedings 17th International Semantic Web Conference (Lecture Notes in Computer Science, Vol. 11136)*, D. Vrandecic et al. (Eds.). Springer, 318–336. Extended version, technical report KRDB18-01, https://www.inf.unibz.it/krdb/tech-reports/.
[20] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems* 25, 2 (2000), 179–227.
[21] DBLP data in RDF. [n.d.]. http://dblp.org/rdf/.
[22] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *The VLDB Journal* 27, 5 (2018), 617–641.
[23] Mónica Figuera, Philipp D. Rohde, and Maria-Ester Vidal. 2021. Trav-SHACL: Efficiently validating networks of SHACL constraints. In *Proceedings WWW'21*, J. Leskovec et al. (Eds.). ACM, 3337–3348.
[24] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs, See [57], 1433–1445.
[25] Jose E. Labra Gayo, Holger Knublauch, and Dimitris Kontokostas. 2021. SHACL test suite and implementation report. https://w3c.github.io/data-shapes/data-shapes-test-suite/.
[26] Jose E. Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. 2018. Validating RDF Data. *Synthesis Lectures on the Semantic Web: Theory and Technology* 16 (2018).
[27] Boris Glavic. 2021. Data provenance: Origins, Applications, Algorithms, and Models. *Foundations and Trends in Databases* 9, 3–4 (2021), 209–441.
[28] Erich Grädel and Val Tannen. 2017. Semiring provenance for first-order model checking. arXiv:1712.01980.
[29] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings 26th ACM Symposium on Principles of Database Systems*. 31–40.
[30] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 query language. W3C Recommendation.
[31] Olaf Hartig and Carlos Buil-Aranda. 2016. Bindings-restricted triple pattern fragments. In *Proceedings OTM Conference (Lecture Notes in Computer Science, Vol. 10033)*, C. Debruyne, H. Panetto, et al. (Eds.). 762–779.
[32] Sven Helmer and Guido Moerkotte. 1997. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann, 386–395.

[33] Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). 2020. *Proceedings WWW'20*. ACM.

[34] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. 2013. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, V. Tannen, L. Wong, et al. (Eds.). Lecture Notes in Computer Science, Vol. 8000. Springer, 382–399.

[35] Phokion G. Kolaitis. 2007. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*. Springer, Chapter 2.

[36] Jose E. Labra Gayo. 2021. Creating knowledge graph subsets using shape expressions. arXiv:2110.11709.

[37] LDBC Graph Query Language Task Force. 2018. G-CORE: A core for future graph query languages, See [57], 1421–1432.

[38] LDF 2020. Linked Data Fragments. https://linkeddatafragments.org.

[39] Martin Leinberger, Philipp Seifer, et al. 2020. Deciding SHACL shape containment through description logics reasoning, See [44], 366–383.

[40] Leonid Libkin, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2018. TriAL: A navigational algebra for RDF triplestores. *ACM Transactions on Database Systems* 43, 1 (2018), 5:1–5:46.

[41] Nikos Mamoulis. 2003. Efficient processing of joins on set-valued attributes. In *Proceedings ACM SIGMOD International Conference on Management of Data*. 157–168.

[42] Guido Moerkotte and Thomas Neumann. 2011. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment* 4 (2011), 843–851.

[43] Martin Otto. 1997. *Bounded Variable Logics and Counting: A Study in Finite Models*. Lecture Notes in Logic, Vol. 9. Springer.

[44] Jeff Z. Pan et al. (Eds.). 2020. *Proceedings 19th International Semantic Web Conference*. Lecture Notes in Computer Science, Vol. 12506. Springer.

[45] Paolo Pareti, George Konstantinidis, et al. 2020. SHACL satisfiability and containment, See [44], 474–493.

[46] Linda D. Paulson. 2007. Developers shift to dynamic programming languages. *Computer* 40, 2 (2007), 12–15.

[47] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009), article 16.

[48] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2010. nSPARQL: A navigational language for RDF. *Journal of Web Semantics* 8, 4 (2010), 255–270.

[49] pyshacl 2021. RDFLib/pySHACL: A Python validator for SHACL. https://github.com/RDFLib/pySHACL.

[50] pySHACL-fragments software. [n.d.]. https://github.com/shape-fragments/pySHACL-fragments.

[51] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2021. Optimizing SPARQL Queries using Shape Statistics. In *Proceedings 24th International Conference on Extending Database Technology*, Y. Velegrakis, D. Zeinalipour-Yazti, et al. (Eds.). OpenProceedings.org, 505–510.

[52] RDF 2014. RDF 1.1 Primer. W3C Working Group Note.

[53] Robert Schaffenrath, Daniel Proksch, Markus Kopp, Iacopo Albasini, Oleksandra Panasiuk, and Anna Fensel. 2020. Benchmark for Performance Evaluation of SHACL Implementations in Graph Databases. In *International Joint Conference on Rules and Reasoning*. Springer, 82–96.

[54] SHACL 2017. Shapes Constraint Language (SHACL). W3C Recommendation.

[55] Shape Fragments Specification. [n.d.]. https://shape-fragments.github.io/shape-fragments-spec.

[56] shexjs [n.d.]. https://github.com/shexjs/shex.js.

[57] SIGMOD 2018. *Proceedings 2018 International Conference on Management of Data*. ACM.

[58] SPARQL 1.2 community group. [n.d.]. DESCRIBE using shapes. https://github.com/w3c/sparql-12/issues/39.

[59] Val Tannen. 2017. Provenance analysis for FOL model checking. *ACM SIGLOG News* 4, 1 (2017), 24–36.

[60] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. I. Computer Science Press.

[61] Ruben Verborgh. 2019. Shaping linked data apps. https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/.

[62] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, et al. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *Journal of Web Semantics* 37–38 (2016), 184–206.

[63] Jesse Wright et al. 2020. Schímatos: A SHACL-based web-form generator for knowledge graph editing. In *Proceedings 19th International Semantic Web Conference*. 65–80.

APPENDIX

A TRANSLATING REAL SHACL TO FORMAL SHACL

In this section we define the function t which maps a SHACL shapes graph $\mathcal{S}$ to a schema H.

Assumptions about the shapes graph:

- All shapes of interest must be explicitly declared to be a `sh:NodeShape` or `sh:PropertyShape`
- The shapes graph is well-formed

Let the sets $\mathcal{S}_n$ and $\mathcal{S}_p$ be the sets of all node shape shape names, respectively property shape shape names defined in the shapes graph $\mathcal{S}$. Let d_x denote the set of RDF triples in $\mathcal{S}$ with x as the subject. We define $t(\mathcal{S})$ as follows:

$$t(\mathcal{S}) = \{(x, t_{nodeshape}(d_x), t_{target}(d_x)) \mid x \in \mathcal{S}_n\} \cup \{(x, t_{propertyshape}(d_x), t_{target}(d_x)) \mid x \in \mathcal{S}_p\}$$

where we define $t_{nodeshape}(d_x)$ in Section A.1, $t_{propertyshape}(d_x)$ in Section A.3 and $t_{target}(d_x)$ in Section A.4.

Remark A.1. We treat node shapes and property shapes separately. In particular, minCount, maxCount, qualified minCount, qualified maxCount, and uniqueLang constraints are only treated below under property shapes. Strictly speaking, however, these constraints may also be used in node shapes, where they are redundant, as the count equals one in this case. For simplicity, we assume the shapes graph does not contain such redundancies.

A.1 Defining $t_{nodeshape}(d_x)$

This function translates SHACL node shapes to shapes in the formalization. We define $t_{nodeshape}(d_x)$ to be the following conjunction:

$$t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{value}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x) \wedge t_{pair}(\mathrm{id}, d_x) \wedge t_{languagein}(d_x)$$

where we define $t_{shape}(d_x), t_{logic}(d_x), t_{tests}(d_x), t_{value}(d_x), t_{in}(d_x), t_{closed}(d_x), t_{langin}(d_x)$ and $t_{pair}(\mathrm{id}, d_x)$ in the following subsections.

A.1.1 Defining $t_{shape}(d_x)$. This function translates the Shape-based Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:node` and `sh:property`.

We define $t_{shape}(d_x)$ to be the conjunction:

$$\bigwedge_{(x,\mathsf{sh:node},y)\in d_x} hasShape(y) \wedge \bigwedge_{(x,\mathsf{sh:property},y)\in d_x} hasShape(y)$$

A.1.2 Defining $t_{logic}(d_x)$. This function translates the Logical Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:and`, `sh:or`, `sh:not`, `sh:xone`.

We define $t_{logic}(d_x)$ as follows:

$$\begin{aligned}
&\bigwedge_{(x,\mathsf{sh:not},y)\in d_x} (\neg hasShape(y)) \wedge \\
&\bigwedge_{(x,\mathsf{sh:and},y)\in d_x} (\bigwedge_{z\in y} hasShape(z)) \wedge \\
&\bigwedge_{(x,\mathsf{sh:or},y)\in d_x} (\bigvee_{z\in y} hasShape(z)) \wedge \\
&\bigwedge_{(x,\mathsf{sh:xone},y)\in d_x} (\bigvee_{a\in y} (a \wedge \bigwedge_{b\in y-\{a\}} \neg hasShape(b)))
\end{aligned}$$

where we note that the object y of the triples with the predicate sh:and, sh:or, or sh:xone is a SHACL list.

A.1.3 Defining $t_{tests}(d_x)$. This function translates the Value Type Constraint Components, Value Range Constraint Components, and String-based Constraint Components, with exception to the `sh:languageIn` keyword which is handled in Section A.1.5, from d_x to shapes from the formalization. This function covers the SHACL keywords:

`sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:minExclusive`, `sh:maxExclusive`, `sh:minLength`, `sh:maxLength`, `sh:pattern`.

We define $t_{tests}(d_x)$ as follows:

$$t_{tests'}(d_x) \wedge \bigwedge_{(x,\texttt{sh:class},y)\in d_x} \geq_1 \texttt{rdf:type/rdf:subclassOf}^*.hasValue(y)$$

where $t_{tests'}(d_x)$ is defined next. Let Γ denote the set of keywords just mentioned above, except for `sh:class`.

$$t_{tests'}(d_x) = \bigwedge_{c\in\Gamma} \bigwedge_{(x,c,y)\in d_x} test(\omega_{c,y})$$

where $\omega_{c,y}$ is the node test in Ω corresponding to the SHACL constraint component corresponding to c with parameter y. For simplicity, we omit the `sh:flags` for `sh:pattern`.

A.1.4 Defining $t_{pair}(\mathrm{id}, d_x)$. This function translates the Property Pair Constraint Components when applied to a node shape from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`. `sh:lessThanOrEquals`.

We define the function $t_{pair}(\mathrm{id}, d_x)$ as follows:

- If $\exists p : (x, \texttt{sh:lessThan}, p) \in d_x$ or $(x, \texttt{sh:lessThanEq}, p) \in d_x$, then we define $t_{pair}(\mathrm{id}, d_x)$ as $\bot$.
- Otherwise, we define $t_{pair}(\mathrm{id}, d_x)$ as

$$\bigwedge_{(x,\texttt{sh:equals},p)\in d_x} eq(\mathrm{id}, p) \wedge \bigwedge_{(x,\texttt{sh:disjoint},p)\in d_x} disj(\mathrm{id}, p)$$

A.1.5 Defining $t_{languagein}(d_x)$. This function translates the constraint component Language In Constraint Component from d_x to shapes from the formalization. This function covers the SHACL keyword: `sh:languageIn`.

The function $t_{languagein}(E, d_x)$ is defined as follows:

$$t_{languagein}(E, d_x) = \bigwedge_{(x,\texttt{sh:languageIn},y)\in d_x} \forall E. \bigvee_{lang\in y} test(\omega_{lang})$$

where y is a SHACL list and ω_{lang} is the element from Ω that corresponds to the test that checks if the node is annotated with the language tag *lang*.

A.1.6 Defining other constraint components. These functions translate the Other Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:closed`, `sh:ignoredProperties`, `sh:hasValue`, `sh:in`.

We define the following functions:

$$t_{value}(d_x) = \bigwedge_{(x,\texttt{sh:hasValue},y)\in d_x} hasValue(y)$$

$$t_{in}(d_x) = \bigwedge_{(x,\texttt{sh:in},y)\in d_x} (\bigvee_{a\in y} hasValue(a))$$

Let P be the set of all properties $p \in I$ such that $(y, \texttt{sh:path}, p) \in \mathcal{S}$ where y is a property shape such that $(x, \texttt{sh:property}, y) \in d_x$ union the set given by the SHACL list specified by the `sh:ignoredProperties` parameter. Then, we define the function $t_{closed}(d_x)$ as follows:

$$t_{closed}(d_x) = \begin{cases} \top & \text{if } (x, \texttt{sh:closed}, true) \notin d_x \\ closed(P) & \text{otherwise} \end{cases}$$

A.2 Defining $t_{path}(pp)$

In preparation of the next Subsection, this function translates the Property Paths to path expressions. This part of the translation deals with the SHACL keywords:

`sh:inversePath`, `sh:alternativePath`, `sh:zeroOrMorePath`, `sh:oneOrMorePath`, `sh:zeroOrOnePath`, `sh:alternativePath`.

For an IRI or blank node pp representing a property path, we define $t_{path}(pp)$ as follows:

$$t_{path}(pp) = \begin{cases} pp & \text{if } pp \text{ is an IRI} \\ t_{path}(y)^- & \text{if } \exists y : (pp, \texttt{sh:inversePath}, y) \in \mathcal{S} \\ t_{path}(y)^* & \text{if } \exists y : (pp, \texttt{sh:zeroOrMorePath}, y) \in \mathcal{S} \\ t_{path}(y)/t_{path}(y)^* & \text{if } \exists y : (pp, \texttt{sh:oneOrMorePath}, y) \in \mathcal{S} \\ t_{path}(y)? & \text{if } \exists y : (pp, \texttt{sh:zeroOrOnePath}, y) \in \mathcal{S} \\ \bigcup_{a \in y} t_{path}(a) & \text{if } \exists y : (pp, \texttt{sh:alternativePath}, y) \in \mathcal{S} \text{ and } y \text{ is a SHACL list} \\ t_{path}(a_1)/\ldots/t_{path}(a_n) & \text{if } pp \text{ represents the SHACL list } [a_1, \ldots, a_n] \end{cases}$$

A.3 Defining $t_{propertyshape}(d_x)$

This function translates SHACL property shapes to shapes in the formalization. Let pp be the property path associated with d_x. Let E be $t_{path}(pp)$. We define $t_{propertyshape}(d_x)$ as the following conjunction:

$$t_{card}(E, d_x) \wedge t_{pair}(E, d_x) \wedge t_{qual}(E, d_x) \wedge t_{all}(E, d_x) \wedge t_{uniquelang}(E, d_x)$$

where we define t_{card}, t_{pair}, t_{qual}, t_{all}, and $t_{uniquelang}$ in the following subsections.

A.3.1 Defining $t_{card}(E, d_x)$. This function translates the Cardinality Constraint Components. from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:minCount`, `sh:maxCount`.

We define the function $t_{card}(E, d_x)$ as follows:

$$\bigwedge_{(x,\texttt{sh:minCount},n) \in d_x} \geq_n E.\top \wedge \bigwedge_{(x,\texttt{sh:maxCount},n) \in d_x} \leq_n E.\top$$

A.3.2 Defining $t_{pair}(E, d_x)$. This function translates the Property Pair Constraint Components when applied to a property shape from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`. `sh:lessThanOrEquals`.

We define the function $t_{pair}(E, d_x)$ as follows:

$$\bigwedge_{(x,\texttt{sh:equals},p) \in d_x} eq(E, p) \wedge \bigwedge_{(x,\texttt{sh:disjoint},p) \in d_x} disj(E, p) \wedge \bigwedge_{(x,\texttt{sh:lessThan},p) \in d_x} lessThan(E, p) \wedge \bigwedge_{(x,\texttt{sh:lessThanOrEquals},p) \in d_x} lessThanEq(E, p)$$

A.3.3 Defining $t_{qual}(E, d_x)$. This function translates the (Qualified) Shape-based Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords:

`sh:qualifiedValueShape`, `sh:qualifiedMinCount`, `sh:qualifiedMaxCount`, `sh:qualifiedValueShapesDisjoint`.

We distinguish between the case where the parameter `sh:qualifiedValueShapesDisjoint` is set to *true*, and the case where it is not:

$$t_{qual}(E, d_x) = \begin{cases} t_{sibl}(E, d_x) & \text{if } (x, \texttt{sh:qualifiedValueShapesDisjoint}, \textit{true}) \in d_x \\ t_{nosibl}(E, d_x) & \text{otherwise} \end{cases}$$

where we define $t_{sibl}(E, d_x)$ and $t_{nosibl}(E, d_x)$ next. Let $ps = \{v \mid (v, \texttt{sh:property}, x) \in \mathcal{S}\}$. We define the set of sibling shapes

$$sibl = \{w \mid \exists v \in ps\, \exists y (v, \texttt{sh:property}, y) : (y, \texttt{sh:qualifiedValueShape}, w) \in \mathcal{S}\}.$$

We also define:

$$\begin{aligned} Q &= \{y \mid (x, \texttt{sh:qualifiedValueShape}, y) \in d_x\} \\ Qmin &= \{z \mid (x, \texttt{sh:qualifiedMinCount}, z) \in d_x\} \\ Qmax &= \{z \mid (x, \texttt{sh:qualifiedMaxCount}, z) \in d_x\} \end{aligned}$$

We now define

$$\begin{aligned} t_{sibl}(E, d_x) = \bigwedge_{y \in Q} \bigwedge_{z \in Qmin} \geq_z E.(hasShape(y) \wedge \bigwedge_{s \in sibl} \neg hasShape(s)) \\ \wedge \bigwedge_{y \in Q} \bigwedge_{z \in Qmax} \leq_z E.(hasShape(y) \wedge \bigwedge_{s \in sibl} \neg hasShape(s)) \end{aligned}$$

and

$$t_{nosibl}(E, d_x) = \bigwedge_{y \in Q} \bigwedge_{z \in Qmin} \geq_z E.hasShape(y) \wedge \bigwedge_{y \in Q} \bigwedge_{z \in Qmax} \leq_z E.hasShape(y).$$

A.3.4 Defining $t_{all}(E, d_x)$. This function translates the constraint components that are not specific to property shapes, but which are applied on property shapes.

We define the function $t_{all}(E, d_x)$ to be:

$$\forall E.(t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x) \wedge t_{languagein}(d_x)) \wedge t_{allvalue}(E, d_x)$$

where

$$t_{allvalue}(E, d_x) = \begin{cases} \top & \text{if } \nexists v : (x, \texttt{sh:hasValue}, v) \in d_x \\ \geq_1 E.t_{value}(d_x) & \text{otherwise} \end{cases}$$

and t_{shape}, t_{logic}, t_{tests}, t_{value}, $t_{languagein}$, and t_{closed} are as defined earlier. Note the treatment of the sh:hasValue parameter when used in a property shape. Unlike the other definitions, it is not universally quantified over the value nodes given by E.

A.3.5 Defining $t_{uniquelang}(E, d_x)$. This function translates the constraint component Unique Lang Constraint Component from d_x to shapes from the formalization. This function covers the SHACL keyword: sh:uniqueLang.

The function $t_{uniquelang}(E, d_x)$ is defined as follows:

$$t_{uniquelang}(E, d_x) = \begin{cases} uniqueLang(E) & \text{if } (x, \texttt{sh:uniqueLang}, true) \in d_x \\ \top & \text{otherwise} \end{cases}$$

A.4 Defining $t_{target}(d_x)$

This function translates the Target declarations to shapes from the formalization. This function covers the SHACL keywords:

sh:targetNode, sh:targetClass, sh:targetSubjectsOf, sh:targetObjectsOf.

We define the function as follows:

$$\begin{aligned} t_{target}(d_x) = \bigvee_{(x, \texttt{sh:targetNode}, y) \in d_x} hasValue(y) \vee \\ \bigvee_{(x, \texttt{sh:targetClass}, y) \in d_x} \geq_1 \texttt{rdf:type/rdf:subclassOf}^*.hasValue(y) \vee \\ \bigvee_{(x, \texttt{sh:targetSubjectsOf}, y) \in d_x} \geq_1 y.\top \vee \\ \bigvee_{(x, \texttt{sh:targetObjectsOf}, y) \in d_x} \geq_1 y^-.\top \end{aligned}$$

If none of these triples are in d_x we define $t_{target}(d_x) = \bot$

B PROOF OF SUFFICIENCY AND CONFORMANCE

Toward a proof of the Sufficiency property, we first prove:

Proof of Proposition 3.1. That $[\![E]\!]^F \subseteq [\![E]\!]^G$ is immediate from $F \subseteq G$ and the monotonicity of path expressions. For the reverse inclusion, we proceed by induction on the structure of E. The base case, where E is a property p, is immediate from the definitions. The inductive cases where E is one of $E_1 \cup E_2$, E_1^-, or E_1/E_2, are clear. Two cases remain:

- E is $E_1?$. If $a = b$, then $(a, b) \in [\![E]\!]^F$ by definition. Otherwise, (a, b) must be in $[\![E_1]\!]^G$. Therefore, by induction, $(a, b) \in [\![E_1]\!]^F \subseteq [\![E]\!]^F$.
- E is E_1^*. If $a = b$, then $(a, b) \in [\![E]\!]^F$ by definition. Otherwise, there exist $i \geq 1$ nodes $c_0, \ldots, c_i$ such that $a = c_0$ and $b = c_i$, and $(c_j, c_{j+1}) \in [\![E_1]\!]^G$ for $0 \leq j < i$. By induction, each $(c_j, c_{j+1}) \in [\![E_1]\!]^F$, whence (a, b) belongs to the transitive closure of $[\![E_1]\!]^F$ as desired.

□

We now give the

Proof of the Sufficiency Theorem. For any shape ϕ, we consider its expansion with relation to the schema H, obtained by repeatedly replacing subshapes of the form $hasShape(s)$ by $def(s, H)$, until we obtain an equivalent shape that no longer contains any subshapes of the form $hasShape(s)$. The proof proceeds by induction on the height of the expansion of ϕ in negation normal form, where the height of negated atomic shapes is defined to be zero. When ϕ is $\top$, $test(t)$, or $hasValue(c)$, and v conforms to ϕ in G, then v clearly also conforms to ϕ in G', as the conformance of the node is independent of the graph. We consider the following inductive cases:

- ϕ is $\phi_1 \wedge \phi_2$. By induction, we know v conforms to ϕ_1 in G' and conforms to ϕ_2 in G'. Therefore, v conforms to $\phi_1 \wedge \phi_2$ in G'.
- ϕ is $\phi_1 \vee \phi_2$. We know v conforms to at least one of ϕ_i for $i \in \{1, 2\}$ in G. Assume w.l.o.g. that v conforms to ϕ_1 in G. Then, our claim follows by induction.
- ϕ is $\geq_n E.\psi$. Here, and in the following cases, we denote $B(v, G, \phi)$ by B. As $G, v \models \phi$, we know there are at least n nodes $x_1, \ldots, x_n$ in G such that $x_i \in [\![E]\!]^G(v)$ and $G, x_i \models \psi$ for all $1 \leq i \leq n$. Let $F = graph(paths(E, G, v, x))$. By Proposition 3.1, $x_i \in [\![E]\!]^F(v)$. By definition of ϕ-neighborhood $F \subseteq B$, and we know $B \subseteq G'$. Therefore, because E is monotone, $x_i \in [\![E]\!]^{G'}(v)$. Furthermore, since $B(x_i, G, \psi) \subseteq B \subseteq G'$, by induction, $G', x_i \models \psi$ as desired.
- ϕ is $\leq_n E.\psi$. First we show that every $x \in [\![E]\!]^{G'}(v)$ that conforms to ψ in G', must also conform to ψ in G.
  Proof by contradiction. Suppose there exists a node $x \in [\![E]\!]^{G'}(v)$ that conforms to ψ in G', but conforms to $\neg\psi$ in G. By definition of ϕ-neighborhood, $B(x, G, \neg\psi) \subseteq B$, and we know $B \subseteq G'$. Therefore, by induction, x conforms to $\neg\psi$ in G', which is a contradiction.
  Because of the claim above, the number of nodes reachable from v through E that satisfy ψ in G' must be smaller or equal to the number of nodes reachable from v through E that satisfy ψ in G. Because we know $G, v \models \phi$, the lemma follows.
- ϕ is $\forall E.\psi$. For all nodes x such that $x \in [\![E]\!]^{G'}(v)$, as E is monotone, $x \in [\![E]\!]^G(v)$. As $G, v \models \phi$, $G, x \models \psi$. By definition of ϕ-neighborhood, $B(x, G, \psi) \subseteq B$. We know $B \subseteq G'$. Thus, by induction, $G', x \models \psi$ as desired.
- ϕ is $eq(E, p)$. We must show that $[\![E]\!]^{G'}(v) = [\![p]\!]^{G'}(v)$. For the containment from left to right, let $x \in [\![E]\!]^{G'}(v)$. Since E is monotone, $x \in [\![E]\!]^G(v)$. Since $G, v \models \phi$, $x \in [\![p]\!]^G(v)$. Let $F = graph(paths(p, G, v, x))$. By Proposition 3.1, $x \in [\![p]\!]^F(v)$. By definition of ϕ-neighborhood, $F \subseteq B$, and we know $B \subseteq G'$. Therefore, because path expressions are monotone, we also have $x \in [\![p]\!]^{G'}(v)$ as desired. The containment from right to left is analogous.
- ϕ is $eq(\mathrm{id}, p)$. We must show that $[\![id]\!]^{G'}(v) = [\![p]\!]^{G'}(v)$, or equivalently we must show that $\{v\} = [\![p]\!]^{G'}(v)$. We know that $G, v \models \phi$, therefore $[\![p]\!]^G(v) = \{v\}$. Now we only need to show that $(v, p, v) \in G'$ as $G' \subseteq G$ (and therefore G' does not contain more p-edges than G). Because by definition of neighborhood $B = \{(v, p, v)\}$, and because $B \subseteq G'$, the claim follows.

- ϕ is $disj(E, p)$. Let $x \in [\![E]\!]^{G'}(v)$. Since E is monotone, $x \in [\![E]\!]^{G}(v)$. Since $G, v \models \phi$, $x \notin [\![p]\!]^{G}(v)$. Therefore, as p is monotone, $x \notin [\![p]\!]^{G'}(v)$. The case where $x \in [\![p]\!]^{G'}(v)$ is analogous.
- ϕ is $disj(\mathrm{id}, p)$. We must show that $(v, p, v) \notin G'$. Because $G, v \models \phi$, we know that $(v, p, v) \notin G$. As $G' \subseteq G$, the claim follows.
- ϕ is $lessThan(E, p)$. Let $x \in [\![E]\!]^{G'}(v)$. Let $(v, p, y) \in G'$. We must show that $x < y$. Since E is monotone, $x \in [\![E]\!]^{G}(v)$ and since $G' \subseteq G$, $(v, p, y) \in G$. As $G, v \models \phi$, we know that $x < y$ in G and thus also in G'.
- ϕ is $lessThanEq(E, p)$. Analogous to the case where ϕ is $lessThan(E, p)$.
- ϕ is $uniqueLang(E)$. Let $x \in [\![E]\!]^{G'}(v)$. Let $y \in [\![E]\!]^{G'}(v)$ such that $y \neq x$. As E is monotone, $x \in [\![E]\!]^{G}(v)$ and $y \in [\![E]\!]^{G}(v)$. As $G, v \models \phi$, we know $y \nsim x$ in G and thus also in G'.
- ϕ is $closed(P)$. Let $(v, p, x) \in G'$. Then, $(v, p, x) \in G$. Therefore, as $G, v \models \phi$, $p \in P$ as desired.
- ϕ is $\neg eq(E, p)$. Since $G, v \models \phi$, there are two cases. First, there exists a node $x \in [\![E]\!]^{G}(v)$ such that $x \notin [\![p]\!]^{G}(v)$. Let $F = graph(paths(E, G, v, x))$. By Proposition 3.1, $x \in [\![E]\!]^{F}(v)$. By definition of ϕ-neighborhood $F \subseteq B$, and we know $B \subseteq G'$. Therefore, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, since $(v, p, x) \notin G$, we know $(v, p, x) \notin G'$. Thus, $[\![E]\!]^{G}(v) \neq [\![p]\!]^{G}(v)$ as desired. For the other case, there exists a node x such that $(v, p, x) \in G$ and $x \notin [\![E]\!]^{G}(v)$. By definition of ϕ-neighborhood, $(v, p, x) \in B \subseteq G'$. However, because E is monotone $x \notin [\![E]\!]^{G'}(v)$. Therefore $[\![p]\!]^{G}(v) \neq [\![E]\!]^{G}(v)$ as desired.
- ϕ is $\neg eq(\mathrm{id}, p)$. Since $G, v \models \phi$, there are two cases. First, $(v, p, v) \notin G$. We know $G' \subseteq G$, therefore if $(v, p, v) \notin G$, then $(v, p, v) \notin G'$ as desired. Second, $(v, p, v) \in G$ and there exists a node x such that $(v, p, x) \in G$ and $x \neq v$. From the definition of neighborhood, we know that this also holds for B and therefore also in G' as $B \subseteq G'$.
- ϕ is $\neg disj(E, p)$. Since $G, c \models \phi$, we know that there exists a node $x \in [\![E]\!]^{G}(v)$ such that $(v, p, x) \in G$. Let $F = graph(paths(E, G, v, x))$. By Proposition 3.1, $x \in [\![E]\!]^{F}(v)$. By definition of ϕ-neighborhood $F \subseteq B$, and we know $B \subseteq G'$. Then, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, by definition of ϕ-neighborhood, also $(v, p, x) \in B \subseteq G'$. Thus, $x \in [\![E]\!]^{G'}(v) \cap [\![p]\!]^{G'}(v)$ as desired.
- ϕ is $\neg disj(\mathrm{id}, p)$. We need to show that $(v, p, v) \in G'$. By definition of neighborhood, $(v, p, v) \in B$. As $B \subseteq G'$, $(v, p, v) \in G'$ as desired.
- ϕ is $\neg lessThan(E, p)$. Since $G, v \models \phi$, there exists a node $x \in [\![E]\!]^{G}(v)$ and a node $y \in [\![p]\!]^{G}(v)$ with $x \not< y$. If we can show that $x \in [\![E]\!]^{G'}(v)$ and $x \in [\![p]\!]^{G'}(v)$, it will follow that $G', v \models \phi$ as desired. Let $F = graph(paths(E, G, v, x))$. By Proposition 3.1, $x \in [\![E]\!]^{F}(v)$. By definition of ϕ-neighborhood, $F \subseteq B$, and we know $B \subseteq G'$. Then, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, by definition of ϕ-neighborhood, $(v, p, x) \in B$. Since $B \subseteq G'$, also $x \in [\![p]\!]^{G'}(v)$ as desired.
- ϕ is $\neg lessThanEq(E, p)$. Analogous to the case where ϕ is $\neg lessThan(E, p)$.
- ϕ is $\neg uniqueLang(E)$. Since $G, v \models \phi$, there exists $x_1 \neq x_2 \in [\![E]\!]^{G}(v)$ such that $x_1 \sim x_2$. As in the previous case, we must show that x_1 and x_2 are in $[\![E]\!]^{G'}(v)$. By Proposition 3.1, for both $i = 1, 2$, we have $x_i \in [\![E]\!]^{F_i}(v)$ with $F_i = graph(paths(E, G, v, x_i))$. By definition of ϕ-neighborhood $F_i \subseteq B \subseteq G'$. Therefore $x_i \in [\![E]\!]^{G'}(v)$ as desired.
- ϕ is $\neg closed(P)$. As $G, v \models \phi$, there exists a property $p \notin P$ and a node x such that $(v, p, x) \in G$. By definition $(v, p, x) \in B(v, G, \phi) \subseteq G'$. Hence, $G', v \models \phi$ as desired.

□

Finally, the proof of the Conformance Theorem now straightforwardly goes as follows. Let $F = Frag(G, H)$; we must show that F conforms to H. Thereto, consider a shape definition $(s, \phi, \tau) \in H$, and let v be a node such that $F, v \models \tau$. Since $F \subseteq G$ and τ is monotone, also $G, v \models \tau$, whence $G, v \models \phi$ since G conforms to H. Since by definition, F contains $B(v, G, \phi)$, Sufficiency yields $F, v \models \phi$ as desired.

C SHAPE FRAGMENTS IN SPARQL

C.1 Proof of Proposition 5.3

As always we work in the context of a schema H. We assume ϕ is put in negation normal form and proceed by induction as in the proof of the Sufficiency Theorem.

Note that Q_ϕ should not merely check conformance of nodes to shapes, but actually must return the neighborhoods. Indeed, that conformance checking in itself is possible in SPARQL (for nonrecursive shapes) is well known; it was even considered for recursive shapes [18]. Hence, in the constructions below, we use an auxiliary SPARQL query $CQ_\phi(?v)$ (C for conformance) which returns, on every RDF graph G, the set of nodes $v \in N(G)$ such that $G, v \models \phi$.

We now describe Q_ϕ for all the cases in the following.

- ϕ is $\top$: empty
- ϕ is $hasValue(c)$: empty
- ϕ is $test(t)$: empty
- ϕ is $closed(P)$: empty
- ϕ is $hasShape(s)$: $Q_{def(s,H)}$
- ϕ is $\phi_1 \wedge \phi_2$ or $\phi_1 \vee \phi_2$:

```
SELECT ?v ?s ?p ?o xs
WHERE {
  { CQ_φ } .
  { Q_φ1 }
  UNION
  { Q_φ2 }
}
```

- ϕ is $\geq_n E.\phi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  {
    { SELECT (?v AS ?t) WHERE { CQ_φ } } .
    { Q_E } .
    { SELECT (?v AS ?h) WHERE { CQ_φ1 } }
  } UNION
  {
    { SELECT (?v AS ?t) WHERE { CQ_φ } } .
    ?t E ?h .
    {
      SELECT (?v AS ?h) ?s ?p ?o
      WHERE { { Q_φ1 } . { CQ_φ1 }}
    }
  }
}
```

- ϕ is $\leq_n E.\phi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  {
    { SELECT (?v AS ?t) WHERE { CQ_φ } } .
    { Q_E } .
    { SELECT (?v AS ?h) WHERE { CQ_¬φ1 } }
  } UNION
```

```
    {
      { SELECT (?v AS ?t) WHERE { CQ_φ } } .
      ?t E ?h .
      {
        SELECT (?v AS ?h) ?s ?p ?o
        WHERE { { Q_¬φ1 } . { CQ_¬φ1 }}
      }
    }
  }
```

- ϕ is $\forall E.\phi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  {
    { SELECT (?v AS ?t) WHERE { CQ_φ } } .
    { Q_E }
  } UNION
  {
    { SELECT (?v AS ?t) WHERE { CQ_φ } } .
    ?t E ?h .
    {
      SELECT (?v AS ?h) ?s ?p ?o
      WHERE { Q_φ1 }
    }
  }
}
```

- ϕ is $eq(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  {
    { Q_E } UNION { Q_p }
  }
}
```

- ϕ is $eq(\text{id}, p)$:

```
SELECT ?v (?s AS ?v) (p AS ?p) (?v AS ?o)
WHERE {
  { CQ_φ } .
  ?v p ?v
}
```

- ϕ is $disj(E, p)$: empty
- ϕ is $disj(\text{id}, p)$: empty
- ϕ is $lessThan(E, p)$: empty
- ϕ is $lessThanEq(E, p)$: empty
- ϕ is $uniqueLang(E)$: empty
- ϕ is $\neg\top$: empty
- ϕ is $\neg hasValue(c)$: empty
- ϕ is $\neg test(t)$: empty
- ϕ is $\neg hasShape(s)$: $Q_{\neg def(s,H)}$

- ϕ is $\neg closed(P)$):

```
SELECT ?v (?v AS ?s) ?p ?o
WHERE {
  { CQ_φ } .
  ?v ?p ?o .
  FILTER (?p NOT IN P)
}
```

- ϕ is $\neg eq(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  {
    { { Q_E } MINUS { ?t p ?h } }
    UNION
    { { Q_p } MINUS { ?t E ?h } }
  }
}
```

- ϕ is $\neg eq(\text{id}, p)$:

```
SELECT ?v (?v AS ?s) (p AS ?p) (?v AS ?o)
WHERE {
  { CQ_φ } .
  { ?v p ?o }
  FILTER (?o != ?v)
}
```

- ϕ is $\neg disj(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  {
    { { Q_E } . { ?t p ?h } }
    UNION
    { { Q_p } . { ?t E ?h } }
  }
}
```

- ϕ is $\neg disj(\text{id}, p)$:

```
SELECT ?v (?v AS ?s) (p AS ?p) (?v AS ?o)
WHERE {
  { CQ_φ } .
  ?v p ?v
}
```

- ϕ is $\neg lessThan(E, p)$):

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  {
    { { Q_E } . { ?t p ?h2 } FILTER (! ?h < ?h2) }
```

```
    UNION
    { { Q_p } . { ?t E ?h2 } FILTER (! ?h2 < ?h) }
  }
}
```

- ϕ is $\neg lessThanEq(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  {
    { { Q_E } . { ?t p ?h2 } FILTER (! ?h <= ?h2) }
    UNION
    { { Q_p } . { ?t E ?h2 } FILTER (! ?h2 <= ?h) }
  }
}
```

- ϕ is $\neg uniqueLang(E)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
  { SELECT (?v AS ?t) WHERE { CQ_φ } } .
  { Q_E } . { ?t E ?h2 }
  FILTER (?h != ?h2 && lang(?h) = lang(?h2))
}
```

D PROOF OF PROPOSITION 6.2

That the seven forms of TPF mentioned in the proposition can be expressed as shape fragments was already shown in the main body of the paper. So it remains to show that all other forms of TPF are not expressible as shape fragments. Since, for any finite set S of shapes, we can form the disjunction $\bigvee S$ of all shapes in S, and $Frag(G, S) = Frag(G, \{\bigvee S\})$ for any graph G, it suffices to consider single shapes ϕ instead of finite sets of shapes. We abbreviate $Frag(G, \{\phi\})$ to $Frag(G, \phi)$.

Formally, let $Q = (u, v, w)$ be a triple pattern, i.e., u, v and w are variables or elements of N. Let V be the set of variables from $\{u, v, w\}$ to N. A solution mapping is a function $\mu : V \to N$. For any node a, we agree that $\mu(a) = a$. Then the TPF query Q maps any input graph G to the subset

$$Q(G) = \{(\mu(u), \mu(v), \mu(w)) \mid \mu : V \to N \;\&\; (\mu(u), \mu(v), \mu(w)) \in G\}.$$

We now say that a shape ϕ *expresses* a TPF query Q if $Frag(G, \phi) = Q(G)$ for every graph G.

We begin by showing:

Lemma D.1. *Let G be an RDF graph and let ϕ be a shape. Assume $Frag(G, \phi)$ contains a triple (s, p, o) where p is not mentioned in ϕ. Then $Frag(G, \phi)$ contains all triples in G of the form (s, p', o'), where p' is not mentioned in ϕ.*

Proof. Since shape fragments are unions of neighborhoods, it suffices to verify the statement for an arbitrary neighborhood $B(v, G, \phi)$. This is done by induction on the structure of the negation normal form of ϕ. In almost all cases of Table 2, triples from $B(v, G, \phi)$ come from E-paths, with E mentioned in ϕ; from $B(v, G, \psi)$, with ψ a subshape of ϕ or the negation thereof; or involve a property p clearly mentioned in ϕ. Triples of the first kind never have a property not mentioned in ϕ, and triples of the second kind satisfy the statement by induction.

The only remaining case is $\neg closed(P)$. Assume (v, p, x) is in the neighborhood, and let $(v, p', x') \in G$ be a triple such that p' is not mentioned in ϕ. Then certainly $p' \notin P$, so (v, p', x') also belongs to the neighborhoods, as desired. □

Using the above Lemma, we give:

Proof of Proposition 6.2. Consider the TPF $Q = (?x, ?x, ?y)$ and assume there exists a shape ϕ such that $Q(G) = \mathit{Frag}(G, \phi)$ for all G. Consider $G = \{(a, a, b), (a, c, b)\}$, where a and c are not mentioned in ϕ. We have $(a, a, b) \in Q(G)$ so $(a, a, b) \in \mathit{Frag}(G, \phi)$. Then by Lemma D.1, also $(a, c, b) \in \mathit{Frag}(G, \phi)$. However, $(a, c, b) \notin P(G)$, so we arrive at a contradiction, and ϕ cannot exist.

Similar reasoning can be used for all other forms of TPF not covered by the proposition. Below we give the table of these TPFs Q, where c and d are arbitrary IRIs, possibly equal, and $?x$ and $?y$ are distinct variables. The right column lists the counterexample graph G showing that $Q(G) \neq \mathit{Frag}(G, \phi)$. Importantly, the property (a or b) of the triples in G is always chosen so that it is not mentioned in ϕ, and moreover, a, b and e are distinct and also distinct from c and d.

Q	G
$(?x, ?y, ?x)$	$\{(a, b, a), (a, b, c)\}$
$(?x, ?y, ?y)$	$\{(a, b, b), (a, b, c)\}$
$(?x, ?x, ?x)$	$\{(a, a, a), (a, a, b)\}$
$(?x, ?y, c)$	$\{(a, b, c), (a, b, d)\}$
$(?x, ?x, c)$	$\{(a, a, c), (a, a, d)\}$
$(?x, ?y, ?y)$	$\{(a, b, b), (a, b, c)\}$
$(c, ?x, ?x)$	$\{(c, a, a), (c, a, b)\}$
$(c, ?x, d)$	$\{(c, a, d), (c, a, e)\}$

□