Validating SHACL constraints over a SPARQL endpoint

Julien Corman[1], Fernando Florenzano[2], Juan L. Reutter[2], and Ognjen Savković[1]

[1] Free University of Bozen-Bolzano, Bolzano, Italy
[2] PUC Chile and IMFD, Chile

Abstract. SHACL (Shapes Constraint Language) is a specification for describing and validating RDF graphs that has recently become a W3C recommendation. While the language is gaining traction in the industry, algorithms for SHACL constraint validation are still at an early stage. A first challenge comes from the fact that RDF graphs are often exposed as SPARQL endpoints, and therefore only accessible via queries. Another difficulty is the absence of guidelines about the way recursive constraints should be handled. In this paper, we provide algorithms for validating a graph against a SHACL schema, which can be executed over a SPARQL endpoint. We first investigate the possibility of validating a graph through a single query for non-recursive constraints. Then for the recursive case, since the problem has been shown to be NP-hard, we propose a strategy that consists in evaluating a small number of SPARQL queries over the endpoint, and using the answers to build a set of propositional formulas that are passed to a SAT solver. Finally, we show that the process can be optimized when dealing with recursive but tractable fragments of SHACL, without the need for an external solver. We also present a proof-of-concept evaluation of this last approach.

1 Introduction

SHACL (for SHApes Constraint Language),[3] is an expressive constraint language for RDF graph, which has become a W3C recommendation in 2017. A SHACL *schema* is a set of so-called *shapes*, to which some nodes in the graph must conform. Figure 1 presents two simple SHACL shapes. The left one, called `:MovieShape`, is meant to define movies in DBPedia. The triple `:MovieShape sh:targetClass dbo:Film` is the *target definition* of this shape, and specifies that all instances of the class `dbo:Film` must conform to this shape. These are called the *target nodes* of a shape. The next triples specify the constraints that must be satisfied by such nodes, namely that they must have an Imdb identifier, and that their directors (i.e. their `dbo:director`-successors in the graph), if any, must conform to the shape `:DirectorShape`. The rightmost shape, called `:DirectorShape`, is meant to define movie directors in DBPedia. It does not have a target definition (therefore no target node either), and states that a director must have exactly one birth date, and can only direct movies that conform to the shape `:MovieShape`.

[3] `https://www.w3.org/TR/shacl/`

```
:MovieShape                          :DirectorShape
  a sh:NodeShape ;                     a sh:NodeShape ;
  sh:targetClass dbo:Film ;            sh:property [
  sh:property [                          sh:path dbo:birthDate;
    sh:path dbo:imdbId ;                 sh:minCount 1 ;
    sh:minCount 1 ] ;                    sh:maxCount 1 ];
  sh:property [                        sh:property [
    sh:path dbo:director ;               sh:inversePath dbo:director ;
    sh:node :DirectorShape ] .           sh:node :MovieShape ] .
```

Fig. 1: Two SHACL shapes, about movies and directors

```
:PulpFiction a dbo:Film .
:PulpFiction dbo:imdbId 24451 .
:PulpFiction dbo:director :QuentinTarantino .
:QuentinTarantino dbo:birthDate "1963-03-27" .
```

```
:Brazil a dbo:Film .
:Brazil dbo:imdbId 15047 .
:Brazil dbo:director :TerryGilliam .
```

```
:CitizenKane dbo:director :OrsonWelles .
```

Fig. 2: Three RDF graphs, respectively valid, invalid and valid against the shapes of Figure 1

The possibility for a shape to refer to another (like `MovieShape` refers to `:DirectorShape` for instance), or to itself, is a key feature of SHACL. This allows designing schemas in a modular fashion, but also reusing existing shapes in a new schema, thus favoring semantic interoperability.

The SHACL specification provides a semantics for *graph validation*, i.e. what it means for a graph to conform to a set of shapes: a graph is *valid* against a set of shapes if each target node of each shape satisfies the constraints associated to it. If these constraints contain shape references, then the propagated constraints (to neighbors, neighbors of neighbors, etc.) must be satisfied as well.

Unfortunately, the SHACL specification leaves explicitly undefined the semantics of validation for schemas with circular references (called *recursive* below), such as the one of Figure 1, where `:MovieShape` and `:DirectorShape` refer to each other. Such schemas can be expected to appear in practice though, either by design (e.g. to characterize relations between events, or a structure of unbounded size, such as a tree), or as a simple side-effect of the growth of the number of shapes (like an object-oriented program may have cyclic references as its number of classes grows). A semantics for graph validation against possibly recursive shapes was later proposed in [11] (for the so-called "core constraint components" of the SHACL specification). It complies with the specification in the non-recursive case. Based on to this semantics, the first graph in Figure **??** is valid against the shapes of Figure 1. The second graph is not, because the director misses a birth date. The third graph is trivially valid, since there is no target node to initiate validation.

Negation is another important feature of the SHACL specification (allowing for instance to state that a node cannot conform to two given shapes at the same time, or to express functionality, like "exactly one birth date" in Figure 1). But as shown in [11], the interplay between recursion and negation makes the

graph validation problem significantly more complex (NP-hard in the size of the graph, for stratified constraints already).

As SHACL is gaining traction, more validation engines become available.[4] However, guidance about the way graph validation may be implemented is still lacking. In particular, to our knowledge, existing implementations deal with recursive schemas in their own terms, without a principled approach to handle the interplay between recursion and negation.

Another key aspect of graph validation is the way the data can be accessed. RDF graphs are generally exposed as SPARQL endpoints, i.e. primarily (and sometimes exclusively) accessible via SPARQL queries. This is often the case for large graphs that may not fit into memory, exposed via triple stores. Therefore an important feature of a SHACL validation engine is the possibility to check conformance of a graph by issuing SPARQL queries over it. This may also be needed when integrating several data sources not meant to be materialized together, or simply to test conformance of data that one does not own.

Several engines can already perform validation via SPARQL queries for fragments of SHACL but, to our knowledge, not in the presence of recursive constraints.[5] This should not come as a surprise: as will be shown in this article, recursive shapes go beyond the expressive power of SPARQL, making validation via SPARQL queries significantly more involved: if the schema is recursive, it is not possible in general to retrieve target nodes violating a given shape by issuing a single SPARQL query. This means that some extra computation (in addition to SPARQL query evaluation) needs to be performed, in memory.

This article provides a theoretical and empirical investigation of graph validation against (possibly recursive) SHACL schemas, when the graph is only accessible via SPARQL queries, and based on the semantics defined in[11]. First, we show that validation can be performed via SPARQL queries only (without extra computation) if the schema is non-recursive, and that some recursive fragments can (in theory) be handled this way if one extends SPARQL with fixed-point iteration. We also show that this strategy cannot be applied for arbitrary SHACL schemas.

Therefore we investigate a second strategy, allowing some in-memory computation, while still accessing the endpoint via queries only. Because the validation problem is NP-hard (in the size of the graph) for the full language, we first define a robust validation approach, that evaluates a limited number of queries over the endpoint, and reduces validation to satisfiability of a propositional formula, potentially leveraging the mature optimization techniques of SAT solvers. We then focus on recursive but tractable fragments of SHACL. For these, we devise an efficient algorithm that relies on the same queries as previously, but performs all the necessary inference on the fly. Finally, we describe a proof-of-concept evaluation of this last approach, performed by validating DBPedia against different schemas, and, for the non-recursive ones, comparing its performance with full delegation to the endpoint.

[4] `https://w3c.github.io/data-shapes/data-shapes-test-suite/`

[5] with the exception of *Shaclex* [5], which can handle recursion, but not recursion and negation together in a principled way.

Organization. Section 2 introduces preliminary notions and Section 3 presents the graph validation problem. Section 4 studies the usage of a single query, whereas Sections 5 and 6 focus on the strategy with in-memory computation, first for the full language, and then for recursive but tractable fragments. Section 7 provides empirical results for this algorithm and full delegation, while Sections 8 and 9 discuss related work and perspectives. Due to space limitations, proofs of propositions are provided in the extended version of this paper, available at [**?**].

2 Preliminaries

We assume familiarity with RDF and SPARQL. We abstract away from the concrete RDF syntax though, representing an RDF graph $\mathcal{G}$ as a labeled oriented graph $\mathcal{G} = \langle V_\mathcal{G}, E_\mathcal{G} \rangle$, where $V_\mathcal{G}$ is the set of nodes of $\mathcal{G}$, and $E_\mathcal{G}$ is a set of triples of the from (v_1, p, v_2), meaning that there is an edge in $\mathcal{G}$ from v_1 to v_2 labeled with property p. We make this simplification for readability, since distinctions such as RDF term types are irrelevant for the content of this paper.

We use $[\![Q]\!]^\mathcal{G}$ to denote the evaluation of a SPARQL query Q over an RDF graph $\mathcal{G}$. As usual, this evaluation is given as a set of *solution mappings*, each of which maps variables of Q to nodes of $\mathcal{G}$. All solution mappings considered in this article are *total* functions over the variables projected by Q. We use $\{?x_1 \mapsto v_1, \ldots, ?x_n \mapsto v_n\}$ to denote the solution mapping that maps $?x_i$ to v_i for $i \in [1..n]$. However, if Q is a *unary* query (i.e. if it projects only one variable), we may also represent $[\![Q]\!]^\mathcal{G} = \{\{?x \mapsto v_1\}, .., \{?x \mapsto v_m\}\}$ as the set of nodes $\{v_1, .., v_m\}$.

SHACL. This article follows the abstract syntax for SHACL core constraint components introduced in [11]. In the following, we review this syntax and the associated semantics for graph validation.

A *shape schema* $\mathcal{S}$ is represented as a triple $\langle S, \text{targ}, \text{def} \rangle$, where S is a set of *shape names*, targ is a function that assigns a *target query* to each $s \in S$, and def is a function that assigns a *constraint* to each $s \in S$.

For each $s \in S$, $\text{targ}(s)$ is a unary query, which can be evaluated over the graph under validation in order to retrieve the *target nodes* of s. The SHACL specification only allows target queries with a limited expressivity, but for the purpose of this article, $\text{targ}(s)$ can be assumed to be an arbitrary unary SPARQL query. If a shape has no target definition (like the shape `:DirectorShape` in Figure 1), we use an arbitrary empty SPARQL query (i.e. with no answer, in any graph), denoted with $\bot$.

The constraint $\text{def}(s)$ for shape s is represented as a formula ϕ verifying the following grammar:

$$\phi \quad ::= \quad \top \mid s \mid I \mid \phi \wedge \phi \mid \neg\phi \mid \geq_n r.\phi \mid \text{EQ}(r_1, r_2)$$

where s is a shape name, I is an IRI,[6] r is a SHACL path[7], and $n \in \mathbb{N}^+$. As syntactic sugar, we use $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\leq_n r.\phi$ for $\neg(\geq_{n+1} r.\phi)$,

[6] More exactly, I is an abstraction, standing for any syntactic constraint over an RDF term: exact value, datatype, regex, etc.

[7] SHACL paths are built like SPARQL property paths, but without the *NegatedPropertySet* operator

$$
\begin{aligned}
[\top]^{\mathcal{G},v,\sigma} &= 1 \\
[\neg\phi]^{\mathcal{G},v,\sigma} &= 1 - [\phi]^{\mathcal{G},v,\sigma} \\
[\phi_1 \wedge \phi_2]^{\mathcal{G},v,\sigma} &= \min\{[\phi_1]^{\mathcal{G},v,\sigma}, [\phi_2]^{\mathcal{G},v,\sigma}\} \\
[\mathrm{EQ}(r_1, r_2)]^{\mathcal{G},v,\sigma} &= 1 \text{ iff } \{v' \mid (v,v') \in [\![r_1]\!]^{\mathcal{G}}\} = \{v' \mid (v,v') \in [\![r_2]\!]^{\mathcal{G}}\} \\
[I]^{\mathcal{G},v,\sigma} &= 1 \text{ iff } v \text{ is the IRI I} \\
[s]^{\mathcal{G},v,\sigma} &= 1 \text{ iff } s(v) \in \sigma \\
[\geq_n r.\phi]^{\mathcal{G},v,\sigma} &= 1 \text{ iff } |\{v' \mid (v,v') \in [\![r]\!]^{\mathcal{G}} \text{ and } [\phi]^{\mathcal{G},v',\sigma} = 1\}| \geq n
\end{aligned}
$$

Table 1: Evaluation of constraint ϕ at node v in graph $\mathcal{G}$ given total assignment σ. We use $(v, v') \in [\![r]\!]^{\mathcal{G}}$ to say that v and v' are connected via SHACL path r.

and $=_n r.\phi$ for $(\geq_n r.\phi) \wedge (\leq_n r.\phi)$. A translation from SHACL core constraint components to this grammar and conversely can be found in [12].

Example 1. The shapes of Figure 1 are abstractly represented as follows:

$$\mathrm{targ}(\texttt{:MovieShape}) = \texttt{SELECT ?x WHERE } \{\texttt{?x a dbo:Film}\}$$
$$\mathrm{targ}(\texttt{:DirectorShape}) = \bot$$
$$\mathrm{def}(\texttt{:MovieShape}) = (\geq_1 \texttt{ dbo:imdbId.}\top) \wedge (\leq_0 \texttt{ dbo:director.}\neg\texttt{:DirectorShape})$$
$$\mathrm{def}(\texttt{:DirectorShape}) = (=_1 \texttt{ dbo:birthDate.}\top) \wedge (\leq_0 \texttt{ dbo:director.}\neg\texttt{:MovieShape})$$

The *dependency graph* of a schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def}\rangle$ is a graph whose nodes are S, and such that there is an edge from s_1 to s_2 iff s_2 appears in $\mathrm{def}(s_1)$. This edge is called negative if such reference is in the scope of at least one negation, and positive otherwise. A schema is *recursive* if its dependency graph contains a cycle, and *stratified* if the dependency graph does not contain a cycle with at least one negative edge. In Example 1, we see that shapes are recursive, since `:MovieShape` references `:DirectorShape` and vice-versa. Since this reference is in the scope of a negation, the schema is not stratified.

Semantics. Since the semantics for recursive schemas is left undefined in the SHACL specification, we use the framework proposed in [11]. The evaluation of a formula is defined with respect to a given assignment, i.e. intuitively a labeling of the nodes of the graph with sets of shape names.

Formally, an *assignment* σ for a graph $\mathcal{G}$ and a schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def}\rangle$ can be represented as a set of atoms of the form $s(v)$ or $\neg s(v)$, with $s \in S$ and $v \in V_{\mathcal{G}}$, that does not contain both $s(v)$ and $\neg s(v)$ for any $s \in S$ or $v \in V_{\mathcal{G}}$. An assignment σ is *total* if for every $s \in S$ and $v \in V_{\mathcal{G}}$, one of $s(v)$ or $\neg s(v)$ belongs to σ. Otherwise (if there are s, v such that neither $s(v)$ not $\neg s(v)$ belong to σ), the assignment is *partial*.

The semantics of a constraint ϕ is given in terms of a function $[\phi]^{\mathcal{G},v,\sigma}$, for a graph $\mathcal{G}$, node v and assignment σ. This function evaluates whether v satisfies ϕ given σ. This semantics depends on which type of assignments is considered. If we only consider *total* assignments, then $[\phi]^{\mathcal{G},v,\sigma}$ is always true or false, and its semantics is defined in Table 1.

We remark that [11] provides a semantics in terms of *partial* assignments. In this case, the inductive evaluation of $[\phi]^{\mathcal{G},v,\sigma}$ is based on Kleene's 3-valued logic. We omit this definition for simplicity, since it is not required in this article, and refer to [11] instead.

3 Validation and tractable fragments of SHACL

In this section, we define what it means for a graph to be valid against a schema. Then we identify tractable fragments of SHACL (including some recursive ones) for which we will introduce either a full SPARQL rewriting (in Section 4) or a validation algorithm (in Section 6).

Validation problem. A graph $\mathcal{G}$ satisfies a schema $\mathcal{S}$ if there is a way to assign shapes names to nodes of $\mathcal{G}$ such that all targets and constraints in $\mathcal{S}$ are satisfied. For instance, in Figure **??** (first graph), one may assign shape `:MovieShape` to node `:PulpFiction`, and shape `:DirectorShape` to node `:QuentinTarantino` while satisfying all targets and constraints. Since we consider two kinds of assignments (total and partial), we also define two types of validation.

Definition 1. *A graph $\mathcal{G}$ is* valid *against a shape schema $\mathcal{S} = \langle S, \text{targ}, \text{def}\rangle$ with respect to total (resp. partial) assignments iff there is a total (resp. partial) assignment σ for $\mathcal{G}$ and $\mathcal{S}$ that verifies the following, for each shape name $s \in S$:*
- *$s(v) \in \sigma$ for each node v in $[\![\text{targ}(s)]\!]^{\mathcal{G}}$, and*
- *if $s(v) \in \sigma$, then $[\text{def}(s)]^{\mathcal{G},v,\sigma} = 1$, and if $\neg s(v) \in \sigma$, then $[\text{def}(s)]^{\mathcal{G},v,\sigma} = 0$.*

The first condition ensures that all targets of a shape are assigned this shape, and the second condition that the assignment is consistent w.r.t. shape constraints.

We note that a total assignment is a specific case of partial assignment. So if $\mathcal{G}$ is valid against $\mathcal{S}$ with respect to total assignments, it is also valid with respect to partial assignments. The converse does not necessarily hold though. But as we see below, it holds for all the tractable fragments considered in this paper. We use this property several times in the following sections.

Tractable fragments of SHACL. As is usual in the database literature, we measure complexity in the size of the graph only (*data complexity*), and not of the schema, given that the size of the graph is likely to grow much faster. The Validation problem then asks, given a graph $\mathcal{G}$ and a fixed schema $\mathcal{S}$, whether $\mathcal{G}$ is valid against $\mathcal{S}$ with respect to total assignments. We also define the Partial-Validation problem, by focusing instead on partial assignments. Unfortunately, both problems have been shown to be NP-complete in [11] for full SHACL.

Two tractable recursive fragments of SHACL were identified in [11] and [**?**] though. The first fragment simply disallows negated constraints, and allows disjunction ($\vee$) as a native operator. We call this fragment $\mathcal{L}^{+}_{\vee}$ below. The second fragment allows all operators, but restricts interplay between recursion and negation. Due to the lack of space, we refer to [**?**] for a formal definition. We call this fragment $\mathcal{L}^{s}$ below. Finally, we also consider non-recursive shapes, the only fragment whose semantics is fully described by the SHACL specification. We call this fragment $\mathcal{L}^{\text{non-rec}}$ below. All these fragments share a property that is key for the correctness of our validation algorithms:

Proposition 1. *The Validation and Partial-Validation problems coincide for $\mathcal{L}^{+}_{\vee}$, $\mathcal{L}^{s}$ and $\mathcal{L}^{non\text{-}rec}$ schemas.*

Complexity. Table 2 summarizes data complexity for full SHACL and all three fragments. All results are new (to our knowledge), aside from the one for full SHACL, borrowed from [11]. Proofs are provided in the online appendix.

	$\mathcal{L}^{\text{non-rec}}$	$\mathcal{L}^{+}_{\vee}$	$\mathcal{L}^{s}$	full SHACL
Complexity of VALIDATION	NL-c	PTIME-c	PTIME-c	NP-c

Table 2: Data complexity of the validation problem.

Such complexity results do not guarantee that efficient algorithms for the tractable fragments can be found though. Moreover, none of the results considers validation over an endpoint. One can nonetheless use these bounds as a guideline, to devise validation procedures for each fragment. In particular, the NP upper bound for the general case suggests that one can take advantage of existing tools optimized for NP-complete problems. And it can indeed be shown that each of the algorithms below is worst-case optimal for the fragments that it addresses.

4 Validation via a single query for non-recursive SHACL

In this section, we address the question of whether validation can be performed by evaluating a single SPARQL query. To state our results, we say that a schema $\mathcal{S}$ can be *expressed* in SPARQL if there is a SPARQL query $q_{\mathcal{S}}$ such that, for every graph $\mathcal{G}$, it holds that $[\![q_S]\!]^{\mathcal{G}} = \emptyset$ iff $\mathcal{G}$ is valid against $\mathcal{S}$

We start with negative results. As shown above, validation for full SHACL is NP-hard in data complexity, whereas SPARQL query evaluation is tractable, which immediately suggests that the former cannot be reduced to the latter. We provide a stronger claim, namely that inexpressibility still holds for much milder classes of schemas, and without complexity assumptions.

Proposition 2. *There is a schema that is in both $L^{+}_{\vee}$ and $\mathcal{L}^{s}$, and cannot be expressed in* SPARQL

On the positive side, one can express non-recursive SHACL schemas in SPARQL:

Proposition 3. *Every schema in $\mathcal{L}^{non\text{-}rec}$ can be expressed in* SPARQL

We provide the main intuition behind this observation (the full construction can be found in appendix). Given a non-recursive shape schema $\mathcal{S} = \langle S, \text{targ}, \text{def}\rangle$, it is possible to associate to each shape $s \in S$ a SPARQL query that retrieves the target nodes of s violating the constraints for s. The query is of the form:

`SELECT ?x WHERE { `$\mathcal{T}(\text{targ}(s), \texttt{?x})$` FILTER NOT EXISTS { `$\mathcal{C}(\text{def}(s), \texttt{?x})$` } }`

where $\mathcal{T}(\text{targ}(s), \texttt{?x})$ is a BGP identical to $\text{targ}(s)$ (with target nodes bound to variable `?x`), and $\mathcal{C}(\text{def}(s), \texttt{?x})$ is a BGP retrieving all nodes verifying $\text{def}(s)$ (again bound to variable `?x`), defined by induction on the structure of $\text{def}(s)$. Then the query $q_{\mathcal{S}}$ above is defined as the union of all such queries (one for each $s \in S$) so that $[\![q_S]\!]^{\mathcal{G}} = \emptyset$ iff $\mathcal{G}$ is valid.

Example 2. As a simple example, consider the schema from Figure 1, To make it non-recursive, the triples `sh:property [ sh:inversePath dbo:director ; sh:Node :MovieShape ]` can be dropped from shape `:DirectorShape`. Then we

get:

$\mathcal{T}(\text{targ}(\texttt{:MovieShape}), \texttt{?x}) = \texttt{\{?x a dbo:Film\}}$

$\mathcal{C}(\text{def}(\texttt{:MovieShape}), \texttt{?x}) = \texttt{\{?x dbo:imdbId ?y0 .?x dbo:director ?y1 .}$

```
?y1 dbo:birthDate ?y2 .FILTER NOT EXISTS{
?y1 dbo:birthDate ?y3 .FILTER(?y2 != ?y3)}}
```

Interestingly, if one uses the recursive SPARQL extension introduced in [15], then both $L^+_\vee$ and $\mathcal{L}^s$ can be expressed:

Proposition 4. *Every schema in $\mathcal{L}^+_\vee$ or $\mathcal{L}^s$ can be expressed in recursive* SPARQL.

5 Validation via multiple queries for full SHACL

This section provides an algorithm for validating arbitrary SHACL shapes over a SPARQL endpoint. The approach reduces validation to satisfiability of a propositional formula, possibly leveraging the optimization techniques of a SAT solver.

Given a graph $\mathcal{G}$ to validate against a shape schema $\mathcal{S}$, the roadmap of this solution is as follows. First, we define a *normal form* for shape schemas. This will allow us to simplify the exposition. Next, we associate one SPARQL query to each shape in a normalized schema. From the evaluation of these queries we construct a set of rules of the form $l_0 \wedge .. \wedge l_n \rightarrow s(v)$, where each l_i is either $s_i(v_i)$ or $\neg s_i(v_i)$, for some $s_i \in S$ and $v_i \in V_\mathcal{G}$. Intuitively, a rule such as $s_1(v_1) \wedge \neg s_2(v_2) \rightarrow s(v)$ means that, if node v_1 conforms to shape s_1 and node v_2 does not conform to shape s_2, then node v conforms to shape s. These rules alone are not sufficient for a sound validation algorithm, so we complement them with additional rules (encoding in particular the targets, and the fact that a node cannot be inferred to conform to a given shape). Finally, we show that $\mathcal{G}$ satisfies $\mathcal{S}$ if and only if the set of constructed formulas is satisfiable.

The approach can handle validations with respect to either total or partial assignments. For validation with respect to partial assignments the set of rules must be satisfiable under 3-valued (Kleene's) logic. For validation with respect to total assignments the set of rules must be satisfiable under standard (2-valued) propositional logic. And as shown in [11], if the schema is stratified, then both notions of validation coincide.

Interestingly, the machinery presented in this section can also be use to design a more efficient algorithm, for the three tractable fragments of SHACL identified in Section 3. This algorithm will be presented in Section 6.

Normal form. A shape schema $\langle S, \text{targ}, \text{def}\rangle$ is *in normal form* if the set S of shape names can be partitioned into two sets S^+ and S^{NEQ}, such that for each $s \in S^+$ (resp. $s \in S^{\text{NEQ}}$), $\text{def}(s)$ verifies ϕ_{s^+} (resp. $\phi_{s^{\text{NEQ}}}$) in the following grammar:

$$\begin{aligned}
\phi_{s^+} &::= \alpha \mid \geq_n r.\alpha \mid \phi_{s^+} \wedge \phi_{s^+} \\
\phi_{s^{\text{NEQ}}} &::= \neg\text{EQ}(r_1, r_2) \\
\alpha &::= \beta \mid \neg\beta \\
\beta &::= \top \mid I \mid s
\end{aligned}$$

If $\phi = \neg\text{EQ}(r_1, r_2)$, then:
q_ϕ = `SELECT ?x WHERE {?x` r_1 w_1 `. ?x` r_2 w_2 `. FILTER (`w_1 `!=` w_2 `)}`

otherwise:
q_ϕ = `SELECT` vars(ϕ) `WHERE {`triples(ϕ) `FILTER (`filters(ϕ)`)}`

with:

vars($\top$)	= {`?x`}	triples($\top$)	= V	filters($\top$)	= {`?x = ?x`}
vars(I)	= {`?x`}	triples(I)	= V	filters(I)	= {`?x =` I}
vars(s)	= {`?x`}	triples(s)	= V	filters(s)	= {`?x = ?x`}
vars($\neg\beta$)	= {`?x`}	triples($\neg\beta$)	= V	filters($\neg\beta$)	= {`!`$f \mid f \in$ filters(β)}

If ϕ is of the form $\phi_1 \wedge \phi_2$, then:

vars(ϕ)	=	vars(ϕ_1) $\cup$ vars(ϕ_2)
triples(ϕ)	=	triples(ϕ_1) $\cup$ triples(ϕ_2)
filters(ϕ)	=	filters(ϕ_1) $\cup$ filters(ϕ_2)

If ϕ is of the form $\geq_n r.\phi'$, then:

vars(ϕ)	=	vars(ϕ') $\cup \{w_1, \ldots, w_n\}$
triples(ϕ)	=	$\{$(`?x` r w_i) $\mid 1 \leq i \leq n\}$
filters(ϕ)	=	$\{f[\texttt{?x}/w_i] \mid f \in \text{filters}(\phi'), 1 \leq i \leq n\} \cup \{w_i \texttt{!=} w_j \mid i \neq j\}$

Fig. 3: Inductive definition of the SPARQL query $q_{\text{def}(s)}$, for each shape s in a normalized schema, where V is a SPARQL subquery that retrieves all nodes in the graph and $f[w/w']$ designates filter expression f, where each occurrence of variable w is replaced by variable w'. SPARQL connectors (`"."` for triples and `AND` for filters) are omitted for readability. All w_i are fresh variables for each occurrence.

It is easy to verify that a shape schema can be transformed in linear time into an equivalent normalized one, by introducing fresh shape names (without target). "Equivalent" here means that both schemas validate exactly the same graphs, with exactly the same target violations.

SPARQL queries. Such normalization allows us to associate a SPARQL query $q_{\text{def}(s)}$ to each shape name in the normalized schema. Intuitively, the query $q_{\text{def}(s)}$ retrieves nodes that may validate def(s), and also the neighboring nodes to which constraints may be propagated in order to satisfy def(s).

For instance, let def(s_0) = ($\geq_1$ $p_1.s_1$) $\wedge$ ($\geq_1$ $p_2.s_2$). Then:[8]

$$q_{\text{def}(s_0)} = \texttt{SELECT ?x ?y1 ?y2 WHERE } \{\texttt{?x } p_1 \texttt{ ?y1 . ?x } p_2 \texttt{ ?y2 }\}$$

Figure 2 provides the definition of $q_{\text{def}(s)}$, by induction on the structure of def(s) (over each occurrence of a formula), based on the normal form.

Rule patterns. Let $\mathcal{S} = \langle S, \text{targ}, \text{def}\rangle$ be a normalized schema. The next step consists in generating a set of propositional rules, based on the evaluation of the queries that have just been defined. To generate such formulas, we associate a *rule pattern* $p_{\text{def}(s)}$ to each shape $s \in S$. This rule pattern is of the form $l_1 \wedge .. \wedge l_n \rightarrow s(\texttt{?x})$, where each l_i is either $\top$, $s_i(w_i)$ or $\neg s_i(w_i)$, for some shape

[8] We omit the trivial `FILTER (?y1 = ?y1 AND ?y2 = ?y2)` for readability.

$$p_{\mathrm{def}(s)} = \left(\bigwedge \mathrm{body}(\mathrm{def}(s)) \right) \rightarrow s(\texttt{?x}), \text{ with:}$$

$$\begin{array}{lcllcl}
\mathrm{body}(\neg \mathrm{EQ}(r_1, r_2)) & = & \{\top\} & & & \\
\mathrm{body}(\top) & = & \{\top\} & \mathrm{body}(\neg\top) & = & \{\top\} \\
\mathrm{body}(I) & = & \{\top\} & \mathrm{body}(\neg I) & = & \{\top\} \\
\mathrm{body}(s') & = & \{s'(\texttt{?x})\} & \mathrm{body}(\neg s') & = & \{\neg s'(\texttt{?x})\} \\
\mathrm{body}(\phi_1 \wedge \phi_2) & = & \mathrm{body}(\phi_1) \cup \mathrm{body}(\phi_2) & & & \\
\mathrm{body}(\geq_n r.\phi) & = & \left\{ \ell[\texttt{?x}/w] \mid \ell \in \mathrm{body}(\phi) \text{ and } w \in \mathrm{vars}(\phi) \right\} & & &
\end{array}$$

Fig. 4: Inductive definition of the rule pattern $p_{\mathrm{def}(s)}$. $l[w_1/w_2]$ designates literal l, where each occurrence of variable w_1 is replaced by variable w_2. $\mathrm{vars}(\phi)$ is defined in Figure 2

$s_i \in S$ and variable w. Figure 3 provides the definition of $p_{\mathrm{def}(s)}$, by induction on the structure of $\mathrm{def}(s)$.

Continuing the example above, if $\mathrm{def}(s_0) = (\geq_1 p_1.s_1) \wedge (\geq_1 p_2.s_2)$, then:

$$q_{\mathrm{def}(s_0)} = \texttt{SELECT ?x ?y1 ?y2 WHERE } \{\texttt{?x } p_1 \texttt{ ?y1 . ?x } p_2 \texttt{ ?y2 }\}$$
$$p_{\mathrm{def}(s_0)} = s_1(\texttt{?y1}) \wedge s_2(\texttt{?y2}) \rightarrow s_0(\texttt{?x})$$

Each rule pattern $p_{\mathrm{def}(s)}$ is then instantiated with the answers to $q_{\mathrm{def}(s)}$ over the SPARQL endpoint, which yields a set $[\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ of propositional rules. For instance, assume that the endpoint returns the following mappings for $q_{\mathrm{def}(s_0)}$:

$$\begin{aligned}
[\![q_{\mathrm{def}(s_0)}]\!]^{\mathcal{G}} = \{ &\{\texttt{?x} \mapsto v_0, \texttt{?y1} \mapsto v_1, \texttt{?y2} \mapsto v_2\}, \\
&\{\texttt{?x} \mapsto v_0, \texttt{?y1} \mapsto v_3, \texttt{?y2} \mapsto v_4\}\}
\end{aligned}$$

Then the set $[\![p_{\mathrm{def}(s_0)}]\!]^{\mathcal{G}}$ of propositional rules is:

$$[\![p_{\mathrm{def}(s_0)}]\!]^{\mathcal{G}} = \{s_1(v_1) \wedge s_2(v_2) \rightarrow s_0(v_0), s_1(v_3) \wedge s_2(v_4) \rightarrow s_0(v_0)\}$$

Formally, $[\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ is the set of propositional formulas obtained by replacing, for each solution mapping $\gamma \in [\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$, every occurrence of a variable w in $p_{\mathrm{def}(s)}$ by $\gamma(w)$.[9] Then we use $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$ to designate the set of all generated rules, i.e.:

$$[\![p_{\mathcal{S}}]\!]^{\mathcal{G}} = \bigcup_{s \in S} [\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$$

We need more terminology. For each rule $r = l_1, .., l_n \rightarrow s(v)$, we call $s(v)$ the *head* of r, and $\{l_1, .., l_n\}$ the *body* of r. Finally, if l is a literal, we use $\neg l$ to designate its negation, i.e. $\neg l = \neg s(v)$ if $l = s(v)$, and $\neg l = s(v)$ if $l = \neg s(v)$.

[9] For some normalized schemas, it could happen that $[\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ always retrieves all nodes from $\mathcal{G}$. This would be the case for example if $\mathrm{def}(s) = s_1 \wedge s_2$. A simple optimization technique here consists in not executing such queries, and instantiate instead the rule pattern $p_{\mathrm{def}(s)}$ with all nodes retrieved by all other queries (and bound to variable ?x).

Additional formulas. So far, with a rule $s_1(v_1) \wedge s_2(v_2) \rightarrow s_0(v_0)$, we are capturing the idea that v_0 must be assigned shape s_0 whenever v_1 is assigned s_1 and v_2 is assigned s_2. But we also need to encode that the only way for v_0 to be assigned shape s_0 is to satisfy one of these rules. If there is just one rule with $s_0(v_0)$ as its head, we only need to extend our set of rules with $s_0(v_0) \rightarrow s_1(v_1) \wedge s_2(v_2)$. But for more generality, we construct a second set $[\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}}$ of propositional formulas, as follows. For every literal $s(v)$ that appears as the head of a rule $\psi \rightarrow s(v)$ in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$, let $\psi_1 \rightarrow s(v), .., \psi_\ell \rightarrow s(v)$ be all the rules that have $s(v)$ as head. Then we extend $[\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}}$ with the formula $s(v) \rightarrow (\psi_1 \vee .. \vee \psi_\ell)$.

Next, we add the information about all target nodes, with the set $[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$ of (atomic) formulas, defined by $[\![t_{\mathcal{S}}]\!]^{\mathcal{G}} = \{s(v) \mid s \in S, s(v) \in \mathrm{targ}(s)\}$.

Finally, we use a last set of formulas to ensure that the algorithm is sound and complete. Intuitively, the query $q_{\mathrm{def}(s)}$ retrieves all nodes that may verify shape s (bound to variable `?x`). But evaluating $q_{\mathrm{def}(s)}$ also provides information about the nodes that are *not* retrieved: namely that they *cannot* verify shape s. A first naive idea is to extend our set of propositional formulas with every literal $\neg s(v)$ for which $[\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ does not contain any mapping where v is bound to `?x`. But this may require retrieving all nodes in $\mathcal{G}$ beforehand, which is inefficient. One can do better, by considering only combinations of shapes and nodes that are already in our rules. We thus construct another set $[\![a_{\mathcal{S}}]\!]^{\mathcal{G}}$ of facts. It contains all literals of the form $\neg s(v)$ such that: $\neg s(v)$ or $s(v)$ appears in some formula in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}} \cup [\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$, and $s(v)$ is not the head of any formula in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$ (i.e. there is no rule of the form $\psi \rightarrow s(v)$ in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$).

Analysis. Let $\Gamma_{\mathcal{G},\mathcal{S}} = [\![p_{\mathcal{S}}]\!]^{\mathcal{G}} \cup [\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}} \cup [\![t_{\mathcal{S}}]\!]^{\mathcal{G}} \cup [\![a_{\mathcal{S}}]\!]^{\mathcal{G}}$ be the union of all the sets of formulas constructed so far. We treat $\Gamma_{\mathcal{G},\mathcal{S}}$ as a set of propositional formulas over the set $\{s(v) \mid s \in S, v \in V_{\mathcal{G}}\}$ of propositions. A first observation is that this set of formulas is polynomial in the size of $\mathcal{G}$. Perhaps more interestingly, one can show that the set $\Gamma_{\mathcal{G},\mathcal{S}}$ is also polynomial in the size of the evaluation of all queries $\mathrm{def}(s)$ and $\mathrm{targ}(s)$. For a finer-grained analysis, let us measure the size of a rule as the number of propositions it contains. From the construction, we get the following upper bounds.

Proposition 5.
- *The sizes of* $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$, $[\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}}$ *and* $[\![a_{\mathcal{S}}]\!]^{\mathcal{G}}$ *are in* $O(\bigcup_{s \in S} [\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}})$.
- *The size of* $[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$ *is in* $O(|\bigcup_{s \in S} [\![\mathrm{targ}(s)]\!]^{\mathcal{G}}|)$.

Hence, the size of the rules we need for inference is not directly dependent on the size of the graph, but rather on the amount of targets and tuples that the SHACL schema selects to be validated.

The next result shows that validation can be reduced to checking whether $\Gamma_{\mathcal{G},\mathcal{S}}$ is satisfiable. "3-valued semantics" here refers to the semantics of Kleene's 3-valued logic (for $\wedge, \vee$ and $\neg$) and where $\psi_1 \rightarrow \psi_2$ is interpreted as $\neg\psi_1 \vee \psi_2$. Then a boolean formula ψ is *satisfiable* under boolean (resp. 3-valued) semantics iff there is a boolean (resp. 3-valued) valuation of the atoms in ψ such that the resulting formula evaluates to *true* under boolean (resp. 3-valued) semantics.

Proposition 6. *For every graph* $\mathcal{G}$ *and schema* $\mathcal{S}$ *we have that:*
- $\mathcal{G}$ *is valid against* $\mathcal{S}$ *with respect to total assignments iff* $\Gamma_{\mathcal{G},\mathcal{S}}$ *is satisfiable under boolean semantics.*

– *$\mathcal{G}$ is valid against $\mathcal{S}$ with respect to partial assignments iff $\Gamma_{\mathcal{G},\mathcal{S}}$ is satisfiable under 3-valued semantics.*

Hence, we can check for validity of schemas over graphs by constructing $\Gamma_{\mathcal{G},\mathcal{S}}$ and checking satisfiability with a SAT solver. This algorithm matches the NP upper bound in data complexity mentioned earlier, since each of $[\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ and $[\![\mathrm{targ}(s)]\!]^{\mathcal{G}}$ can be computed in polynomial time, when $\mathcal{S}$ is considered to be fixed, and thus the set $\Gamma_{\mathcal{G},\mathcal{S}}$ of rules can be computed in polynomial time in data complexity.

6 Optimized algorithm for tractable fragments

The propositional framework described in the previous section applies to arbitrary shape schemas. But it also allows us to devise a more efficient validation algorithm for the tractable fragments $\mathcal{L}^{\text{non-rec}}$, $\mathcal{L}^{+}_{\vee}$ and $\mathcal{L}^{s}$. One could, in theory, feed the same formulas as above to a SAT solver for these fragments. Instead, the algorithm below performs this inference on-the-fly, without the need for a solver. In addition, the validity of the graph may in some cases be decided before evaluating all SPARQL queries (one per shape) against the endpoint.

The key property that enables this algorithm pertains to the notion of *minimal fixed-point assignment* for SHACL, defined in [11]. Due to space limitations, we only rephrase the results relevant for this algorithm in our own terms.

Lemma 1. *For every graph $\mathcal{G}$ and schema $\mathcal{S}$ in $\mathcal{L}^{non\text{-}rec}$, $\mathcal{L}^{+}_{\vee}$ or $\mathcal{L}^{s}$, there is a partial assignment $\sigma^{\mathcal{G},\mathcal{S}}_{\mathrm{minFix}}$ such that:*

1. *$\sigma^{\mathcal{G},\mathcal{S}}_{\mathrm{minFix}}$ can be computed in polynomial time from $\Gamma^{\mathcal{G},\mathcal{S}}$, and*
2. *$\mathcal{G}$ is valid against $\mathcal{S}$ iff $\neg s(v) \notin \sigma^{\mathcal{G},\mathcal{S}}_{\mathrm{minFix}}$ holds for every $s(v) \in [\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$*

Algorithm. The algorithm shares similarities with the one of Section 5. It proceeds shape by shape, materializing the rules $[\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ defined in Section 5. We will see that these rules are sufficient to compute the assignment $\sigma^{\mathcal{G},\mathcal{S}}_{\mathrm{minFix}}$.

The whole procedure is given by Algorithm 1. Variable S' keeps track of the shapes already processed, variable R stores all rules that are known to hold, and σ is the assignment under construction. All arguments are passed by reference. We use procedure SELECTSHAPE (Line 3) to select from S the next shape s to be processed. This selection can be non-deterministic, but as we will see, this choice also opens room for optimization. All the necessary inference is performed by procedure SATURATE, explained below. In the worst case, the loop terminates when all shapes have been processed (i.e. when $S' = S$, Line 7).

We now describe the inference carried out by procedure SATURATE, whose detailed execution is given by Figure 4. heads(R) (Line 3 in procedure NEGATE) designates the sets of all heads appearing in R, whereas $\bigcup$bodies(R) (Line 2 in procedure NEGATE) designates the union of all rule bodies in R. The inference is performed exhaustively by procedures NEGATE and INFER. Procedure NEGATE derives negative information. For any (possibly negated) atom $s(v)$ that is either a target or appears in some rule, we may be able to infer that $s(v)$ cannot hold. This is the case if $s(v)$ has not been inferred already (i.e. $s(v) \notin \sigma$), if the query

Algorithm 1 TRACTABLE ALGORITHM FOR VALIDATION

Input: Graph $\mathcal{G}$, normalized schema $\mathcal{S} = \langle S, \text{def}, \text{targ}\rangle$, set $[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$ of targets.

$\sigma, R, S' \leftarrow \emptyset$
repeat
  $s \leftarrow \text{SELECTSHAPE}(S, S')$
  $S' \leftarrow S' \cup \{s\}$
  $R \leftarrow R \cup [\![p_{\text{def}(s)}]\!]^{\mathcal{G}}$
  $\text{SATURATE}(\sigma, R, S')$
until $S' = S$
$\text{SATURATE}(\sigma, R, S)$

$q_{\text{def}(s)}$ has already been evaluated, and if there is no rule in R with $s(v)$ as its head. In such case, $\neg s(v)$ is added to σ.

Procedure INFER performs two types of inference. First, the obvious one: if R contains a rule $l_1 \wedge \cdots \wedge l_n \rightarrow s(v)$ and each of $l_1, .., l_n$ has already been inferred, then $s(v)$ is inferred, and the rule is dropped. The second inference is negative: if the negation of any l_i has already been inferred, then this rule cannot be applied (to infer $s(v)$), so the entire rule is dropped.

procedure $\text{SATURATE}(\sigma, R, S')$
  repeat
    $\sigma' \leftarrow \sigma$
    $\text{NEGATE}(\sigma, R, S')$
    $\text{INFER}(\sigma, R)$
  until $\sigma = \sigma'$
end procedure

procedure $\text{INFER}(\sigma, R)$
  $R' \leftarrow \emptyset$
  for all $l_1 \wedge .. \wedge l_n \rightarrow s(v) \in R$ **do**
    if $\{l_1, .., l_n\} \subseteq \sigma$ **then**
      $\sigma \leftarrow \sigma \cup \{s(v)\}$
    else if $\{\neg l_1, .., \neg l_n\} \cap \sigma = \emptyset$ **then**
      $R' \leftarrow R' \cup \{l_1 \wedge .. \wedge l_n \rightarrow s(v)\}$
  end for
  $R \leftarrow R'$
end procedure

procedure $\text{NEGATE}(\sigma, R, S')$
  for all $l \in [\![t_{\mathcal{S}}]\!]^{\mathcal{G}} \cup \bigcup \text{bodies}(R)$ **do**
    if $(l = s(v)$ or $l = \neg s(v))$ and $s \in S'$ and $s(v) \notin \sigma \cup \text{heads}(R)$ **then**
      $\sigma \leftarrow \sigma \cup \{\neg s(v)\}$
  end for
end procedure

Fig. 5: Components of in-memory saturation in Algorithm 1

Let $\sigma^{\mathcal{G},\mathcal{S}}_{\text{final}}$ be the state of variable σ after termination. We show:

Proposition 7. $\sigma^{\mathcal{G},\mathcal{S}}_{\text{final}} = \sigma^{\mathcal{G},\mathcal{S}}_{\text{minFix}}$

Interestingly, one can use this result to validate each target $s(v)$ individually: if $\neg s(v) \in \sigma^{\mathcal{G},\mathcal{S}}_{\text{final}}$, then v does not conform to shape s. Otherwise it conforms to it.

Optimization. An earlier termination condition may apply for Algorithm 1. Indeed, we observe that during the execution, the assignment σ under construction can only be extended. Therefore the algorithm may already terminate if all

		#Queries	Query exec. (ms) max	Query exec. (ms) total	#Query answ. max	#Query answ. total	#Rules max	Total exec. (ms)
VALID$_{\text{single}}$	$\mathcal{S}^2_{\text{non-rec}}$	1	3596	3596	111113	111113	0	**3596**
	$\mathcal{S}^3_{\text{non-rec}}$	1	3976	3976	111629	111629	0	**3976**
	$\mathcal{S}^4_{\text{non-rec}}$	1	5269	5269	111906	111906	0	**5269**
VALID$_{\text{rule}}$	$\mathcal{S}^2_{\text{non-rec}}$	3	858	956	37040	38439	49278	**5305**
	$\mathcal{S}^3_{\text{non-rec}}$	4	827	1149	37040	52122	50774	**5553**
	$\mathcal{S}^4_{\text{non-rec}}$	7	1308	1944	39719	65175	64060	**6857**
	$\mathcal{S}^2_{\text{rec}}$	5	912	1278	37040	59382	59852	**5651**
	$\mathcal{S}^3_{\text{rec}}$	6	1489	3436	61355	146382	146104	**8318**
	$\mathcal{S}^4_{\text{rec}}$	8	1530	4955	61355	186593	159597	**11503**

Table 3: Validation using VALID$_{\text{rule}}$ for all 6 schemas, and VALID$_{\text{single}}$ for non-recursive schemas, on DBP$_{\text{full}}$. Here # **Queries** is the number of executed queries, **Query exec. max** (resp. **total**) is the maximum execution time for a query (resp. total time for all queries) in milliseconds, #**Query answ. max** (resp. **total**) is the max. number of solution mappings for a query (resp. total for all queries) #**Rules max** is the max. number of rules in memory during the execution, and **Total exec.** is the overall execution time in milliseconds

targets have been inferred to be valid or invalid, i.e. if $s(v) \in \sigma$ or $\neg s(v) \in \sigma$ for every target $s(v) \in [\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$. This means that one should also try to process the shapes in the best order possible. For instance, in the experiments reported below, function SELECTSHAPE (Line 3) first prioritizes the shapes that have a target definition, then the shapes referenced by these, and so on, in a depth-first fashion. Such an ordering offers another advantage, which pertains to traceability: when signaling to the user the reason why a given target is violated, it is arguably more informative to return an explanation at depth n than at depth $n + q$. Therefore this breadth-first strategy guarantees that one of the "most immediate" explanations for a constraint violation is always found.

7 Evaluation

We implemented a slightly optimized version of Algorithm 1, A prototype is available online [**?**], together with source code and execution and build instructions.

Shape schemas. We designed two sets of simple shapes, called $M_{\text{non-rec}}$ and M_{rec} below. These shapes pertain to the domain of cinema (movies, actors, etc.), based on patterns observed in DBPedia [1], similarly to the shapes of Figure 1. They were designed to cover several cases discussed in this article (shape reference, recursion, etc.). All shapes are available online ([**?**]). The first set $M_{\text{non-rec}}$ contains shape references, but is non-recursive, whereas the second set M_{rec} is recursive. Out of $M_{\text{non-rec}}$, we created 3 shape schemas $\mathcal{S}^2_{\text{non-rec}}$, $\mathcal{S}^3_{\text{non-rec}}$ and $\mathcal{S}^4_{\text{non-rec}}$, containing 2, 3 and 4 shapes respectively. Similarly for M_2, we created 3 shape schemas $\mathcal{S}^2_{\text{rec}}$, $\mathcal{S}^3_{\text{rec}}$ and $\mathcal{S}^4_{\text{rec}}$.

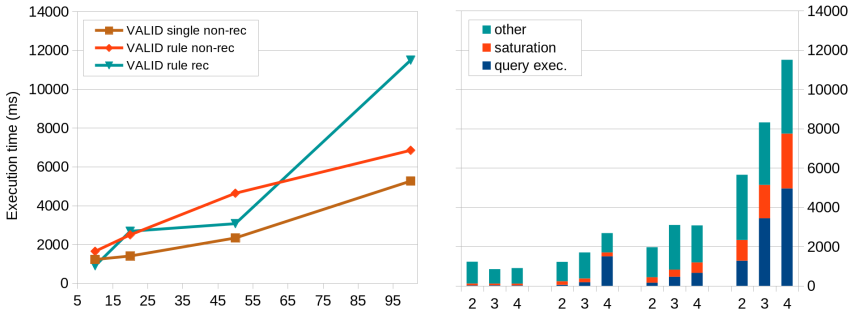
Data. We used the latest version of DBPedia (2016-10), specifically the datasets "Person Data", "Instance Types", "Labels", "Mappingbased Literals" and "Mappingbased Objects" (in English), downloadable from [1], with around 61 million triples (7.7 GB in .ttl format). We denote this dataset as DBP$_{\text{full}}$. The number

of targets to be validated in DBP_{full} is 111938. To test the scalability of the approach, we also produced four samples by randomly selecting 10%, 20% and 50% of triples in DBP_{full}. We denote these datasets as DBP_{10}, DBP_{20} and DBP_{50}.

Setting. We use $\text{VALID}_{\text{rule}}$ to designate our implementation of the rule-based procedure described by Algorithm 1. The implementation is essentially identical, but with a relaxed normal form for the input schema, and improvements geared towards increasing the selectivity of some queries. The ordering of query evaluation (Function SELECTSHAPE in Algorithm 1, Line 3) was based on the dependency graph, in a breadth-first fashion, starting with the only shape with non-empty target definition, then followed by the shapes it references (if not evaluated yet), etc. $\text{VALID}_{\text{single}}$ designates validation performed by executing a single query, as described in Section 4. This approach is only applicable to the non-recursive shape schemas $\mathcal{S}^2_{\text{non-rec}}$, $\mathcal{S}^3_{\text{non-rec}}$ and $\mathcal{S}^4_{\text{non-rec}}$.

We used Virtuoso v7.2.4 as triplestore. Queries were run on a 24 cores Intel Xeon CPU at 3.47 GHz, with a 5.4 TB 15k RPM RAID-5 hard-drive cluster and 108 GB of RAM. Only 1 GB of RAM was dedicated to the triplestore for caching and intermediate operations. In addition, the OS page cache was flushed every 5 seconds, to ensure that the endpoint could only exploit these 1 GB for caching. These precautions ensure that most of the dataset cannot be cached, which would artificially speed up query execution times.

Results. Table 3 provides statistics for the validation of DBP_{full} against all schemas. A first observation is that execution times remained very reasonable (less that 12 seconds) for a complete validation, given the high number of targets (111938) and the size of the dataset. Another immediate observation is that for the non-recursive schemas, $\text{VALID}_{\text{single}}$ consistently outperformed $\text{VALID}_{\text{rule}}$. However, execution times for both approaches remain in the same order of magnitude. Based on these results, the rule-based approach appears as a relatively small price to pay for an algorithm that is not only more robust (i.e. can handle recursion), but also guarantees traceability of each shape violation (whereas the single-query approach essentially uses the endpoint as a black-box). Figure 5 (a) illustrates scalability of $\text{VALID}_{\text{single}}$ and $\text{VALID}_{\text{rule}}$. The focus is on scalability w.r.t to the size of the graph (data complexity) rather than in the size of the schema. The execution times are given for the different samples of DBPedia (DBP_{10}, DBP_{20}, DBP_{50} and DBP_{full}) against the largest shapes schemas ($\mathcal{S}^4_{\text{non-rec}}$ and $\mathcal{S}^4_{\text{rec}}$). The main observation is that for $\text{VALID}_{\text{rule}}$, execution time increased significantly faster for the recursive schema than for the non-recursive one. Finally, Figure 5 b describes how execution time was split between query answering, saturation and other tasks (mostly grounding rules with solution mappings), for $\text{VALID}_{\text{rule}}$, for each $\mathcal{S}^i_{\text{rec}}$ and for each sample of DBPedia. An important observation here is that the proportion of execution time dedicated to query answering increased with the data and number of shapes, even when the number of rules in memory was arguably large (≥ 100000 for $\mathcal{S}^3_{\text{rec}}$ and $\mathcal{S}^4_{\text{rec}}$ with). This suggests that the extra cost induced by in-memory inference during the execution of Algorithm 1 may not be a bottleneck.



(a) Execution times against $\mathcal{S}^4_{\text{non-rec}}$ with $\text{VALID}_{\text{single}}$, and against $\mathcal{S}^4_{\text{non-rec}}$ and $\mathcal{S}^4_{\text{rec}}$ with $\text{VALID}_{\text{rule}}$.

(b) Repartition of execution time against $\mathcal{S}_{\text{rec}}$, with $\text{VALID}_{\text{rule}}$. "2, 3" and "4" stand for $\mathcal{S}^2_{\text{rec}}$,$\mathcal{S}^3_{\text{rec}}$ and $\mathcal{S}^4_{\text{rec}}$.

Fig. 6: Scalability over DBP_{10}, DBP_{20}, DBP_{50} and DBP_{full}

8 Related work

TopBraid Composer [8] allows validating an RDF graph against a non-recursive SHACL schema via SPARQL queries, similarly to the approach described in Section 4). The tool was initially developed for the language SPIN, which largely influenced the design of the SHACL specification. A list of other implementations of SHACL validation can be found at [**?**] (together with unit tests for non-recursive shapes). To our knowledge, none of these can validate recursive constraints via SPARQL queries, with the exception of *Shaclex* [5], already mentioned **??**.

ShEx [16, 10] is another popular constraint language for RDF, which shares many similarities with SHACL, but is inspired by XML schema languages. A semantics for (stratified) recursive ShEx schemas was proposed in [10], which differs from the one followed in this article for SHACL. ShEx validation is supported by several open-source implementations (like *shex.js* [6] or *Shaclex* [5]), either in memory or over a triple-store. To our knowledge, no procedure for validating recursive ShEx via SPARQL queries has been defined or implemented yet.

Prior to ShEx or SHACL, a common approach to define expressive constraints over RDF graphs was to use OWL axioms with (some form of) closed-world assumption (CWA) [**?** 14] However, OWL is originally designed to model incomplete knowledge (with the open-world assumption), therefore not well-suited to express constraints. In terms of implementation, [14] proposed an encoding of such constraints into complex logical programs, but the usage made of OWL does not allow for recursive constraints. Similarly, *Stardog* [7] offers the possibility to write constraints as OWL axioms under CWA, which are then converted to SPARQL queries. In contrast to SHACL though, these constraints are "local", i.e cannot refer to other constraints. Stardog also has a limited support for SHACL validation, currently in beta phase.

Finally, writing non-recursive constraints natively as SPARQL queries is a relatively widespread approach, for instance to assess data quality, like in [**?**

], and the SHACL specification also allows defining constraints in this way (in addition to the "core constraint components" considered in this article).

9 Conclusion and perspectives

We hope that this article may provide guidelines for future implementations of (possibly recursive) SHACL constraint validation via SPARQL queries. As for delegating validation to query evaluation, we showed the limitation of the approach, opened up an alternative in terms of recursive SPARQL, and provided (in the extended version of this article) a full translation from non-recursive SHACL to SPARQL. Regarding validation via queries, but with additional (in-memory) computation, we devised and evaluated and algorithm for three tractable fragments of SHACL, with encouraging performances. This strategy can also still be largely optimized, generating more selective queries and/or reducing the cost of in-memory inference. A natural extension of this work is the application to ShEx schemas, even though the semantics for recursive ShEx proposed in [10] differs from the one followed in this paper. Finally, a key feature of a constraint validation engine is the ability to provide explanations for target violations. Their number is potentially exponential though, so a natural continuation of this work is to define some preference over explanations, and devise algorithms that return an optimal one, without sacrificing performance.

Bibliography

[1] DBpedia Knowledge Base. Link: `https://wiki.dbpedia.org/`.
[2] Eli-Validator. Link:`http://labs.sparna.fr/eli-validator/home`.
[3] Extended version of the paper, prototype implementation and experiments material. `https://tinyurl.com/y434tlvl/`.
[4] SHACL Playground. Link:`http://shacl.org/playground/`.
[5] Shaclex. Link:`https://github.com/labra/shaclex/`.
[6] Shex.js. Link:`https://github.com/shexSpec/shex.js/`.
[7] Stardog ICV. Link:`https://www.stardog.com/blog/data-quality-with-icv/`.
[8] TopBraid Composer. Link:`https://www.topquadrant.com/products/topbraid-composer/`.
[9] W3C OWL 2. Link:`https://www.w3.org/TR/owl2-overview/`.
[10] I. Boneva, J. E. Labra Gayo, and E. G. Prud'hommeaux. Semantics and Validation of Shapes Schemas for RDF. In *ISWC*, 2017.
[11] J. Corman, J. L. Reutter, and O. Savkovic. Semantics and validation of recursive SHACL. *ISWC*, 2018.
[12] J. Corman, J. L. Reutter, and O. Savkovic. Semantics and validation of recursive shacl (extended version). *Technical Report KRDB18-1, KRDB Research Center, Free Univ. Bozen-Bolzano*, 2018.
[13] J. Corman, J. L. Reutter, and O. Savkovic. A tractable notion of stratification for SHACL. In *ISWC*, 2018.
[14] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between OWL and relational databases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):74–89, 2009.
[15] J. L. Reutter, A. Soto, and D. Vrgoc. Recursion in SPARQL. In *ISWC*, 2015.
[16] S. Staworko, I. Boneva, J. E. Labra Gayo, S. Hym, E. G. Prud'hommeaux, and H. Solbrig. Complexity and Expressiveness of ShEx for RDF. In *ICDT*, 2015.