# SHACL Constraints with Inference Rules

Paolo Pareti[(✉)1], George Konstantinidis[1], Timothy J. Norman[1], and Murat Şensoy[2]

[1] University of Southampton, Southampton, United Kingdom
`pp1v17@soton.ac.uk`
[2] Özyeğin University, Istanbul, Turkey

**Abstract.** The Shapes Constraint Language (SHACL) has been recently introduced as a W3C recommendation to define constraints that can be validated against `RDF` graphs. Interactions of SHACL with other Semantic Web technologies, such as ontologies or reasoners, is a matter of ongoing research. In this paper we study the interaction of a subset of SHACL with inference rules expressed in datalog. On the one hand, SHACL constraints can be used to define a "schema" for graph datasets. On the other hand, inference rules can lead to the discovery of new facts that do not match the original schema. Given a set of SHACL constraints and a set of datalog rules, we present a method to detect which constraints could be violated by the application of the inference rules on some graph instance of the schema, and update the original schema, i.e, the set of SHACL constraints, in order to capture the new facts that can be inferred. We provide theoretical and experimental results of the various components of our approach.

## 1 Introduction

Information about the type of data contained in a dataset is a critical piece of information both to understand data, and to interface with databases. While the relational model explicitly defines a schema, graph data representations are inherently schemaless, in the sense that any `RDF` triple could, in principle, be stored in any `RDF` triplestore. The Shapes Constraint Language (SHACL) [10], is a W3C recommendation recently introduced to define properties of `RDF` datasets. SHACL allows the definition of constraints that can be validated against `RDF` graphs. Such constraints can be seen as the schema of the graphs that do not violate them. Schemas are not static objects, and they can evolve over time to reflect changes in the datasets they model. One important source of change in graph datasets comes from the application of inference rules. Inference rules can be used to reason about ontological properties, such as class membership. They can also be used for non-ontological types of inference, such as aggregating sensor data to detect important facts such as the presence of a fire, or the temperature of a room. This paper focuses on datalog rules [6] without negation (the exact subset of datalog that we consider is defined in Section 2).

The application of inference rules might generate new facts, not captured by the original schema definition. Given a set of SHACL constraints and a set of inference rules, we would like to determine whether a graph, initially valid with respect to the SHACL constraints, remains valid after computing its closure with the inference rules. If constraint violations can occur, a domain expert could decide whether to remove the

inference rules that cause these violations, or to update the violated SHACL constraints. Updating the violated constraints to account for the new facts that can be produced via inference effectively creates a new schema.

This research is motivated by use cases in the area of Occupational Health and Safety (OHS), and in particular in the mining sector. In these areas, schemas are used to model and understand the underlying data sources, and to ensure interoperability between different applications. Inference rules, usually developed separately, are used to aggregate raw sensor data into more useful abstractions and encode OHS policies (e.g. to detect unsafe working environments). At the moment, domain experts are needed to define such rules. However this process is slow, expensive and error prone. Our research aims to better inform experts about the effects of applying certain rules (which could affect the schema, and thus interoperability) and automatically detect conflicts between rules and schemas. For example, as schemas frequently change (e.g. sensors malfunction, or new ones are deployed), it is essential to automatically detect schema changes that render important rules (and policies) no longer applicable, on unforeseen datasets.

In this paper we present an approach that models *triplestore schemas* as triplets of sets: a set of triple patterns that can be appropriately instantiated by RDF triples, a set of positions in those triples that cannot be instantiated by literal values (e.g. object positions in triples), and a set of existential validity rules (such as tuple-generating dependencies [8]) which must hold on the instatiated triples in order for our graph to be valid. Our triplestore schema captures a fragment of SHACL, but abstracts away from its particular syntax and can be used as a logical tool to model properties of RDF graphs in general. However, it is not meant to provide a complete formal semantic representation of the core SHACL components, such as the one presented in [7].

Furthermore, we investigate how our triplestore schemas interact with inference rules and evolve into new schemas, that we call *schema consequences*; these are schemas that model all possible RDF graphs extended with the inferred facts. Given an input schema $S$, we want to reason about the applicability of inference rules on all potential instances of $S$, and compute the schema consequence. This problem proves challenging even without taking existential validity rules into account in our schemas; i.e., for what we call our *simple* schema consequence. To reason with inference rules in this version of the problem we have to make use of the notion of a "canonical" instance of $S$, representative of all other instances. For this, we first explore such an instance known as the *critical instance* and investigated in relational databases [13]; running the inference rules on this graph enables us to produce our schema consequence. However, the critical instance is inefficient, as it has a very large size, and so we turn our attention to finding a much smaller representative instance, that we call the *sandbox* graph. We then present a novel query rewriting algorithm that can compute the simple schema consequence on the sandbox graph, much more efficiently than in the critical instance case.

Building on top of our simple schema consequence we use a novel combination of techniques, variations of datalog rewriting [1] and the Chase algorithm [4], to produce our *existential-preserving* schema consequence, a triplestore schema the identifies and removes from its description the existential validity rules that could potentially be violated on some instance produced by the inference rules. We provide both theoretical and experimental evaluations of our approach.

## 2   Background

We consider triplestores containing a single RDF *graph*. Such a graph is a set of *triples* $\mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ where $\mathbb{U}$ is the set of all IRIs and $\mathbb{L}$ the set of all literals. Although we do not explicitly discuss blank nodes, it should be noted that, for the purpose of this paper, when they occur in a graph they can be treated exactly as IRIs. We use the term *constants* to refer to both literals and IRIs. A *graph pattern* is a set of *triple patterns* defined in: $(\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$, where $\mathbb{V}$ the set of all variables. Given a pattern $P$, *vars*$(P)$ and *const*$(P)$ are the sets of variables and constants in the elements of $P$, respectively. We represent IRIs as namespace-prefixed strings of characters, where a namespace prefix is a sequence of zero or more characters followed by a colon e.g. :a; literals as strings of characters enclosed in double-quotes, e.g. "l"; and variables as strings of characters prefixed by a question-mark, e.g. ?v. The first, second and third elements of a triple $t$ are called, respectively, *subject*, *predicate* and *object*, and are denoted by $t[x]$, $x \in \tau$ with $\tau = \{1,2,3\}$ throughout the paper.

A *variable substitution* is a partial function $\mathbb{V} \nrightarrow \mathbb{V} \cup \mathbb{U} \cup \mathbb{L}$. A *mapping* is a variable substitution defined as $\mathbb{V} \nrightarrow \mathbb{U} \cup \mathbb{L}$. Given a mapping $m$, if $m(?v) = n$, then we say $m$ contains *binding* $?v \rightarrow n$. The domain of a mapping $m$ is the set of variables *dom*$(m)$. Given a triple or a graph pattern $p$ and a variable substitution $m$ we abuse notation and denote by $m(p)$ the pattern generated by substituting every occurrence of a variable $?v$ in $p$ with $m(?v)$ if $?v \in dom(m)$ (otherwise $?v$ remains unchanged in $m(p)$). A *grounding* is a mapping that transforms a graph pattern into a graph.

Given a graph pattern $P$ and a graph $I$, we denote the SPARQL evaluation of $P$ over $I$ as the set of mappings $[\![P]\!]_I$, as defined in [14]. A graph pattern *matches* a graph if its evaluation on the graph returns a non-empty set of mappings. We consider inference rules $A \rightarrow C$, where $A$ and $C$ are graph patterns, and can be expressed as SPARQL construct queries. Note that essentially both $A$ and $C$ in an inference rule are conjunctive queries [1]. The *consequent* $C$ of the rule is represented in the construct clause of the query, which is instantiated using the bindings obtained by evaluating the *antecedent* $A$, expressed in the where clause. For technical reasons, we restrict the subset of datalog that we consider with the requirement that each triple pattern in the consequent $C$ of a rule: (1) has a constant in the predicate position; and (2) does not have the same variable in the subject and object position. A single application of an inference rule $r : A \rightarrow C$ to a dataset $I$, denoted by $r(I)$, is $I \cup \bigcup_{m \in [\![A]\!]_I} \{m(C)$, if $m(C)$ is a valid RDF a graph$\}$. These rules capture datalog [1] and subsets of rule notations such as SPIN and SWRL can be represented in this format [3]. The closure of a dataset $I$ under a set of inference rules $R$, denoted by *clos*$(I,R)$, is the unique dataset obtained by repeatedly applying all the rules in $R$ until no new statement is inferred, that is, $clos(I,R) = \bigcup_{i=0}^{i=\infty} I_i$, with $I_0 = I$, and $I_{i+1} = \bigcup_{r \in R} \{r(I_i)\}$.

The Shapes Constraint Language (SHACL) defines constraints that can be validated against RDF graphs. An example of a constraint is the requirement for an RDF term to be an IRI. The nodes of an RDF graph against which such constraints are validated are called *focus nodes*. At the core of the SHACL language is the concept of *shapes*. A shape groups together a set of constraints, and defines which focus nodes it should apply to. A shape could either directly target specific nodes, such as all the elements of a class, or it could be referenced by other shapes. For example, it is possible to define

the shape of a "well-formed email address", and then specify that every entity of type "person" must have at least one email address that satisfies this shape. In this paper we prefix SHACL terms with the namespace sh:.

Given a schema $S$, we denote with $\mathbb{I}(S)$ the set of instances of $S$, which are the graphs that $S$ models. We say that two schemas $S$ and $S'$ are semantically equivalent if they model the same set of instances; i.e. if $\mathbb{I}(S) = \mathbb{I}(S')$. Naturally, the interpretation of SHACL constraints as a schema is based on SHACL validation. We say that a graph is an instance of a SHACL schema, defined by its set of constraints, if the graph does not violate the SHACL constraints.

## 3    Problem Definition

In this section we are going to present our definition of a *triplestore schema*, a simple representation that captures a powerful fragment of SHACL. A set of SHACL shapes $S$ belongs to this fragment if and only if there exists a triplestore schema $S'$ such that $\mathbb{I}(S) = \mathbb{I}(S')$ (the set of instances of a triplestore schema will be defined later in this section). An important characteristic of this fragment (discussed later) is that its existential validity rules must have atomic antecedents and consequents. This is sufficient to model common constraints for RDF validation, such as the Data Quality Test Patterns TYPEDEP, TYPRODEP, PVT, RDFS-DOMAIN and RDFS-RANGE in the categorisation by Kontokostas et al. [11].

The two main components of triplestore schemas are: (1) a set of abstract triple patterns, that intend to model all different triples/instantiations of those patterns; and (2) a set of existential validity constraints that represent "if-then" statements of SHACL shapes. Abstracting away from the particulars of the SHACL syntax, on one hand, simplifies our approach and, on the other hand, makes it applicable to the fragments of other languages (e.g. ShEx [15]) which can be converted into our triplestore schema. Once we have a triplestore schema in place, we define our problem of how do instances of such a schema interact with a set of datalog inference rules. In particular, we would like to reason at the schema level and decide if there is a potential instance graph of our schema on which our datalog rules would infer facts that violate the validity constraints.

### 3.1    From SHACL to Triplestore Schemas

Our work is inspired by Internet of Things (IoT) settings and as our running example we consider a dataset of a mining company. The SHACL schema, $S_1$, for this mine dataset is presented in Fig. 1. This repository collects data from sensors carried by workers and deployed in the mine. Data is modelled according to the Semantic Sensor Network Ontology (SSN) [12], with namespace prefix s:. In SSN, sensor measurements are called *observations*. The *result* of an observation (e.g. "20") relates to a particular *observed property* (e.g. temperature) of a particular *feature of interest* (e.g. a room). In our example the mine contains two types of sensors. The first is a carbon monoxide (CO) detector, which records value "0" if the CO concentration is within the allowed limits, and "1" otherwise. The second is an RFID reader used to locate personnel in the mine

by sensing the nearby RFID tags carried by the mine workers. SHACL shape `:s0` specifies that the collected sensor data will only refer to those two sensor types. The dataset of the mine is expected to contain a list of known personnel RFID tags, and information on who is currently carrying them. Shape `:s1` specifies that for every personnel tag, we know who it is carried by. Shapes `:s2` and `:s3` restrict features of interest to being `IRI`s and measurement results to be `IRI`s or literals. Shape `:s4` declares that the sensor data contains instances of only two classes, namely sensor observations, and personnel tags.

```
:s0    a                   sh:NodeShape ;
       sh:targetObjectsOf  sn:observedProperty ;
       sh:in               ( :COLevel :TagID ) .
:s1    a                   sh:NodeShape ;
       sh:targetClass      :PersonnelTag ;
       sh:property         [ sh:minCount  1 ;
                              sh:path      :carriedBy ] .
:s2    a                   sh:NodeShape ;
       sh:targetObjectsOf  sn:hasFeatureOfInterest ;
       sh:nodeKind         sh:IRI .
:s3    a                   sh:NodeShape ;
       sh:targetObjectsOf  sn:hasResult ;
       sh:nodeKind         sh:IRIOrLiteral .
:s4    a                   sh:NodeShape ;
       sh:targetObjectsOf  rdf:type ;
       sh:in               ( sn:Observation :PersonnelTag ) .
```

**Fig. 1.** Schema $S_1$.

When using SHACL as a schema language, we would like the constraints to describe the type of data contained in a dataset as accurately as possible. SHACL constraints usually target only a limited number of predicates in an RDF graph, and triples with predicates other than the targeted ones could be present in the graph without causing violations. However, for our purposes we adopt a closed-world view of the available vocabulary of predicates, and we would like to restrict valid graphs to only contain a fixed set of predicates. This vocabulary restriction can be specified by an appropriate SHACL constraint that uses the `sh:closed` component. We assume, therefore, that all SHACL schemas that we work with contain a component that specifies that valid instances of this schema do not contain predicates other than the ones that appear in the SHACL shapes. This is inline with relational databases where the discovery of completely new types of facts (i.e. triples with unforseen predicates) would be reflected by a corresponding change in the original schema.

In our running example, instances of schema $S_1$ would contain triples matching the triples patterns of graph pattern $S_1^G$ displayed in Fig. 2, where each variable can be appropriately instantiated by an IRI or a literal. In fact, exactly such a set of triple patterns will be the first element of our representation of triplestore schemas, called a *schema graph*, defined below. Note that valid instances of our schema might contain

```
?v1 sn:observedProperty :COLevel .   ?v5 :carriedBy ?v6 .
?v2 sn:observedProperty :TagID .     ?v7 sn:hasFeatureOfInterest ?v8 .
?v3 rdf:type sn:Observation .        ?v9 sn:hasResult ?v10 .
?v4 rdf:type :PersonnelTag .
```

**Fig. 2.** Graph pattern $S_1^G$.

multiple instantiations of some, but not necessarily all of the predicates defined in the schema graph, and they cannot contain other kinds of triples (e.g. undefined predicates). We use different variables in $S_1^G$ to denote the fact that variables in a schema graph act as wildcards, and are not meant to join triple patterns together.

In addition to the schema graph, a second part of our schema representation will be the subset of variables from the schema graph, called the *no-literal set*, where literals can not occur in valid instances. For example, we cannot instantiate variables ?v7 and ?v8 of triple pattern [?v7,sn:hasFeatureOfInterest,?v8] from Fig. 2 with a literal; in the case of ?v7, because we would not generate a valid RDF triple, and in the case of ?v8, because it would violate shape :s2.

The last part of our schema representation will translate SHACL constraints to "if-then" statements like the following, which corresponds to shape :s1 of schema $S_1$:

$$e_1 = [\text{?v1,rdf:type,:PersonnelTag}] \rightarrow^\exists [\text{?v1,:carriedBy,?v2}]$$

These constraints are essentially *existential rules* [2], also expressible as tuple-generating dependencies (TGDs) [8]. For all practical purposes, the part of SHACL that we consider, when translatable to existential rules, falls into a language known as linear weakly-acyclic TGDs [8] with a single atom in the consequent. Linear means that these rules have only one atom (one triple pattern) in the antecedent, and *weakly-acyclic* is a property that guarantees that forward-chaining algorithms, such as the Chase [4], terminate. Formally, we define an existential rule as a formula of the form: $a \rightarrow^\exists c$, where $a$ and $c$, respectively the antecedent and the consequent of the rule, are triple patterns. The consequent specifies which triples must exist in a graph whenever the antecedent holds in that graph. We say that an existential rule $a \rightarrow^\exists c$ is *violated* on a graph $I$ if there exists a mapping $m \in [\![\{a\}]\!]_I$ such that $[\![m(c)]\!]_I = \varnothing$ (i.e. $m(c)$ is not in $I$), and *satisfied* otherwise. Note that if $m(c)$ is a ground triple, and $m(c) \in I$, then $[\![m(c)]\!]_I$ is not empty, as it contains the empty mapping [14]. Given a set of existential rules $E$, we use *violations*$(E,I)$ to refer to the set of pairs $\langle m,e \rangle$, where $e \in E$ and mapping $m$ causes $e$ to be violated on instance $I$.

We are now ready to define our triplestore schemas. A triplestore schema (or from now on, just *schema*) $S$, is a tuple $\langle S^G, S^\Delta, S^\exists \rangle$, where $S^G$, called a *schema graph*, is a set of triple patterns where every variable occurs at most once, $S^\Delta$ is a subset of the variables in $S^G$ which we call the *no-literal* set, and $S^\exists$ is a set of existential rules. Intuitively, $S^G$ defines the type of triples that can appear in a graph, where variables act as wildcards, which can be instantiated with any constant element. To account for the restrictions imposed by the RDF data model, the no-literal set $S^\Delta$ defines which variables cannot be instantiated with literals, thus $S^\Delta$ must at least include all variables that occur in the subject or predicate position in $S^G$. For example, if $\langle \text{?v1,sn:hasResult,} \text{?v2} \rangle \in S_r^G$ and $\text{?v2} \notin S_r^\Delta$, then the instances of schema $S_r$ can contain any triple that

has `sn:hasResult` as a predicate. If $\langle$`?v3,rdf:type,:Observation`$\rangle \in S_{I'}^G$ and $?v3 \in S_{I'}^\Delta$, the instances of $S_{I'}$ can contain any entity of type `:Observation`. While $S^G$ and $S^\Delta$ together define the set of all the possible triples that can be found in a graph, not all combinations of such triples are valid instances of the schema. The set of existential rules $S^\exists$ defines further requirements that instances of the schema must satisfy. Formally, a graph $I$ is an *instance* of a triplestore schema $\langle S^G, S^\Delta, S^\exists \rangle$ if and only if *violations*$(S^\exists, I) = \varnothing$ and for every triple $t^I$ in $I$ there exists a triple pattern $t^S$ in $S^G$, such that $t^I$ is an *instantiation* of $t^S$ w.r.t $S^\Delta$; that is, there exists a mapping $m$ such that (1) $m(t^S) = t^I$ and (2) $m$ does not bind any variable in $S^\Delta$ to a literal.

For our SHACL to triplestore schema translation we direct the reader to our external appendix[1] and our implementation in our code repository.[2]

### 3.2   Inference Rules and Schema consequences

We are interested in the effect that inference rules (not to be confused with existential rules) have on RDF graphs, and their interaction with existential rules. Inference rules are used to compute the *closure* of instances of our original schema as defined in Section 2. As an example consider the of inference rules $R_1 = \{r_1, r_2, r_3\}$ below. Rule $r_1$ states that the RFIDs recorded by the sensors should be interpreted as personnel tags, and it records the location of where they are detected. Rule $r_2$ states that locations with a high carbon monoxide (CO) concentration should be off-limits. Rule $r_3$ states that if someone is located in an off-limits area, then they are trespassing in that area.

$$
\begin{aligned}
r_1 &= \big\{\,[\text{?v1,sn:observedProperty,:TagID}],\\
&\qquad [\text{?v1,sn:hasResult,?v2}],\\
&\qquad [\text{?v1,sn:hasFeatureOfInterest,?v3}]\big\}\\
&\quad \rightarrow \big\{[\text{?v2,rdf:type,:PersonnelTag}],[\text{?v2,:isLocatedIn,?v3}]\big\}\\
r_2 &= \big\{\,[\text{?v1,sn:observedProperty,:COLevel}],\\
&\qquad [\text{?v1,sn:hasResult,"1"}],\\
&\qquad [\text{?v1,sn:hasFeatureOfInterest,?v2}]\big\}\\
&\quad \rightarrow \big\{[\text{?v2,rdf:type,:OffLimitArea}]\big\}\\
r_3 &= \big\{\,[\text{?v1,:isLocatedIn,?v2}],\\
&\qquad [\text{?v2,rdf:type,:OffLimitArea}]\big\}\\
&\quad \rightarrow \big\{[\text{?v1,:isTrespassingIn,?v2}]\big\}
\end{aligned}
$$

In our example, an emergency response application might need to know who is carrying each personnel RFID tag, in order to compute an emergency response plan. In this case, it is important to know which existential rules the application of a set of inference rules can violate. Once potential violations are detected, a domain expert could, for example, decide whether to relax (i.e. remove) the violated existential rules, or to remove the inference rules that cause the violations.

Thus, an example of the central question we address in this paper is: is $e_1$ guaranteed to remain valid in instances of schema $S_1$ under closure with inference rules $R_1$? The answer to this question is *no*, as demonstrated by graph $I_1$, which is a valid instance of $S_1$. This instance contains two records of miner tags being detected, namely `:WID1` and

---

[1] https://github.com/paolo7/ISWC2019-appendix/raw/master/Appendix.pdf
[2] https://github.com/paolo7/ISWC2019-code

:WID2. While we know that :WID1 is being carried by worker :Alex, we do not have any such information about tag :WID2.

```
:o1 sn:observedProperty :TagID;      :o3 sn:observedProperty :COLevel;
   sn:hasFeatureOfInterest :room1;      sn:hasFeatureOfInterest :room2;
   sn:hasResult :WID1 .                 sn:hasResult "1" .
:o2 sn:observedProperty:TagID;       :WID1 a :PersonnelTag;
   sn:hasFeatureOfInterest:room2;       :carriedBy :Alex .
   sn:hasResult :WID2 .
```

**Fig. 3.** Instance $I_1$.

Rule $r_1$ will deduce that :WID2 is a personnel tag, by inferring triples [:WID2, rdf:type,:PersonnelTag] and [:WID2,:isLocatedIn,:room2] from instance $I_1$. However, since there is no information on who is carrying tag :WID2, existential rule $e_1$ is violated. A domain expert analysing this conflict can then decide to either relax $e_1$, to state that there is not always information on who is carrying a personnel tag, or to remove rule $r_1$, to state that not all RFIDs recorded by the sensors are personnel tags. Rule $r_2$ is triggered by observation :o3, inferring triple [:room2,rdf:type, :OffLimitArea]. The IRI :OffLimitArea is not one of the types allowed in the original schema. Therefore, we might want to either revise rule $r_2$, or extend schema $S_1$ to allow for instances of this type. Facts inferred by rules $r_1$ and $r_2$ together trigger rule $r_3$, which will infer [:WID2,:isTrespassingIn,:room2]; i.e., that the person carrying the RFID tag :WID2 is trespassing in dangerous area :room2. These new facts contain the new predicate :isTrespassingIn, and thus violate our closed-world interpretation of schema $S_1$ (as captured by our schema graph patterns). Hence, if one wants to retain all inference rules $R_1$ in our mine repository, an alteration of the original schema (and its schema graph) is required.

In this paper we deal with predicting these kinds of constraint violation, and computing an updated schema that accounts for them, without looking at specific instances such as $I_1$. Given a schema $S :< S^G, S^\Delta, S^\exists >$ and a set of inference rules $R$, we want to compute a new schema, called *schema consequence*, which captures all the inferences of the set of rules $R$ on any potential instance of $S$. By computing an updated triplestore schema, once a violation is detected, our approach gives preference to maintaining inference rules over maintaining the original schema, essentially choosing to alter the schema graph and/or the existential rules. This is not an inherent limitation of our approach, which could be easily transformed to a method that maintains the original schema and chooses to reject conflicting inference rules.

To present our problem incrementally, we first compute a *simple* schema consequence which does not take existential rules into account (i.e. it only deals with $S^G$ and $S^\Delta$ of our triplestore schema) and then we extend our solution to take $S^\exists$ into account in our *existentially-preserving* schema consequence.

The simple interpretation of a schema consequence captures the type of triples that the closure of an instance of the schema could potentially contain. Given a schema $S$ and a set of inference rules $R$, a schema $S'$ is a *simple schema consequence* of $S$ with

respect to $R$, denoted $con(S,R)$, if $\mathbb{I}(S') = \bigcup_{I \in \mathbb{I}(S)}\{I' \mid I' \subseteq clos(I,R)\}$. It is important to note that every subset of an instance's closure is still an instance of the simple schema consequence. Thus a simple schema consequence can contain the consequence of an inference rule application without containing a set of triples matching the antecedent, or vice versa. This situation is commonly encountered when some triples are deleted after an inference is made. Effectively, this definition does not assume that all the triples in an instance's closure are retained. One use of this schema consequence is to discover whether certain important facts (e.g. personnel trespassing in a dangerous area) can be inferred from the given schema (e.g. available sensor data streams) and set of inference rules (e.g. sensor data aggregation rules). Another use is to compute which inference rules are *applicable* on a schema, which means that they will be triggered on at least one instance of that schema.

Given a schema $S$ and a set of inference rules $R$, a schema $S'$ is an existential-preserving schema consequence of $S$ with respect to $R$, denoted $con^{ex}(S,R)$, if and only if $\mathbb{I}(S') = \bigcup_{I \in \mathbb{I}(S)}\{I' \mid I' \subseteq clos(I,R) \wedge violations(S^{\exists},I') = violations(S^{\exists},clos(I,R))\}$. In other words, instances of an existential-preserving schema consequence are generated by computing the closure of an instance of the original schema under the inference rules, and then discarding a set of triples as long as doing so does not generate new violations of existential rules $S^{\exists}$. This allows us to detect which existential rules can be violated by the application of inference rules (and not just by arbitrary triple deletions). Our approaches to compute simple and existential-preserving schema consequences are presented, respectively, in sections 4 and 5.

## 4   Computing the Simple Schema Consequence

We compute $con(S,R)$ iteratively, on a rule-by-rule basis. In correspondence to a single application $r(I)$, of an inference rule $r$ on an instance $I$, we define a *basic consequence* of a schema $S$ by an inference rule $r$, denoted by $r(S)$, as a finite schema $S'$ for which $\mathbb{I}(S') = \bigcup_{I \in \mathbb{I}(S)}\{I' \mid I' \subseteq r(I)\}$. It is now easy to see that the consequence schema for a set of inference rules $con(S,R)$ is obtained by repeatedly executing $r(S)$ for all $r \in R$ until no new pattern is inferred. Formally, $con(S,R) = \bigcup_{i=0}^{i=n} S_i$, with $S_0 = S$, and $S_{i+1} = \bigcup_{r \in R}\{r(S_i)\}$, and $S_n = S_{n-1}$ (modulo variable names). In this section we focus on computing a single basic schema consequence $r(S)$, and describe two approaches for this, namely Schema Consequence by Critical Instance (`critical(S,r)`), and Schema Consequence by Query Rewriting (`score(S,r)`).

Given a schema $S$ and an inference rule $r : A \to C$, our approach to compute the basic schema consequence for $r$ on $S$ is based on evaluating $A$, or an appropriate rewriting thereof, on a "canonical" instance of $S$, representative of all instances modelled by the schema. The mappings generated by this evaluation are then (1) filtered (in order to respect certain literal restrictions in `RDF`) and (2) applied appropriately to the consequent $C$ to compute the basic schema consequence.

We present two approaches, that use two different canonical instances. The first instance is based on the concept of a *critical instance*, which has been investigated in the area of relational databases before [13] (and similar notions in the area of Description Logics [9]). Adapted to our `RDF` setting, the critical instance would be created by

substituting the variables in our schema, in all possible ways, with constants chosen from the constants in $S^G$ and $A$ as well as a new constant not in $S^G$ or $A$. In [13] this instance is used in order to decide Chase termination; Chase is referred to rule inference with *existential* variables, more expressive than the ones considered here and for which the inference might be infinite (see [4] for an overview of the Chase algorithm). Although deciding termination of rule inference is slightly different to computing the schema consequence, we show how we can take advantage of the critical instance in order to solve our problem. Nevertheless, this approach, that we call `critical`, creates prohibitively large instances when compared to the input schema. Thus, later on in this section we present a rewriting-based approach, called `score`, that runs a rewriting of the inference rule on a much smaller canonical instance of the same size as $S^G$.

**The Critical Approach.** For both versions of our algorithms we will use a new IRI $:\lambda$ such that $:\lambda \notin const(S^G) \cup const(A)$. Formally, the critical instance $\mathbb{C}(S, A \to C)$ is the set of triples:

$$\{t|\ \text{triple } t \text{ with } t[i] = \begin{cases} c & \text{if } t^S[i] \text{ is a variable and:} \\ & \quad (1)\ c \text{ is a IRI or} \\ & \quad (2)\ i = 3 \text{ and } t^S[i] \notin S^\Delta \\ t^S[i] & \text{if } t^S[i] \text{ is not a variable} \end{cases},$$
$$t^S \in S^G, i \in \tau, c \in const(S^G) \cup const(A) \cup \{:\lambda\}\}$$

The critical instance replaces variables with IRIs and literals from the set $const(S^G) \cup const(A) \cup \{:\lambda\}$, while making sure that the result is a valid RDF graph (i.e. literals appear only in the object position) and that it is an instance of the original schema (i.e. not substituting a variable in $S^\Delta$ with a literal). In order to compute the triples of our basic schema consequence for inference rule $r$ we evaluate $A$ on the critical instance, and post-process the mappings $[\![A]\!]_{\mathbb{C}(S,r)}$ as we will explain later. Before presenting this post-processing of the mappings we stress the fact that this approach is inefficient and as our experiments show, non scalable. For each triple $t$ in the input schema $S$, up to $|const(S^G) \cup const(A) \cup \{:\lambda\}|^{vars(t)}$ new triples might be added to the critical instance.

**The Score Approach.** To tackle the problem of efficiency we present an alternative solution based on query rewriting, called `score`. This solution uses a small instance called the *sandbox* instance which is obtained by taking all triple patterns of our schema graph $S^G$ and substituting all variables with the same new IRI $:\lambda$. This results in an instance with the same number of triples as $S^G$. The main property that allows us to perform this simplification is the fact that variables in $S^G$ are effectively independent from each other. Formally, a sandbox graph $\$(S)$ is the set of triples:

$$\{t|\ \text{triple } t \text{ with } t[i] = \begin{cases} :\lambda & \text{if } t^S[i] \text{ is a variable,} \\ t^S[i] & \text{else} \end{cases}, t^S \in S^G, i \in \tau\}$$

Contrary to the construction of the critical instance, in our sandbox graph, variables are never substituted with literals (we will deal with RDF literal peculiarities in a post-processing step). Also notice that $\$(S) \in \mathbb{I}(S)$ and $\$(S) \subseteq \mathbb{C}(S,r)$. As an example, consider the sandbox graph $\$(S_1)$ of schema $S_1$ from Section 3.1:

```
:λ sn:observedProperty :COLevel .      :λ :carriedBy :λ .
:λ sn:observedProperty :TagID .        :λ sn:hasFeatureOfInterest :λ .
:λ rdf:type sn:Observation .           :λ sn:hasResult :λ .
:λ rdf:type :PersonnelTag .
```

The critical instances $\mathbb{C}(S_1, r_1)$, $\mathbb{C}(S_1, r_2)$ and $\mathbb{C}(S_1, r_3)$ from our example would contain all the triples in $\$(S_1)$, plus any other triple obtained by substituting some variables with constants other than :λ. For example, $\mathbb{C}(S_1, r_2)$ would contain the triple $[:\lambda,$ sn:hasResult,:OffLimitArea]$\}$.

In order to account for all mappings produced when evaluating $A$ on $\mathbb{C}(S, r)$ we will need to evaluate a different query on our sandbox instance, essentially by appropriately rewriting $A$ into a new query. To compute mappings, we consider a rewriting $\mathbb{Q}(A)$ of $A$, which expands each triple pattern $t_A$ in $A$ into the union of the 8 triple patterns that can be generated by substituting any number of elements in $t_A$ with :λ. Formally, $\mathbb{Q}(A)$ is the following conjunction of disjunctions of triple patterns, where $\bigwedge$ and $\bigvee$ denote a sequence of conjunctions and disjunctions, respectively:

$$\mathbb{Q}(A) = \bigwedge\nolimits_{t \in A} \Big( \bigvee\nolimits_{\substack{x_1 \in \{:\lambda, t[1]\} \\ x_2 \in \{:\lambda, t[2]\} \\ x_3 \in \{:\lambda, t[3]\}}} \langle x_1, x_2, x_3 \rangle \Big)$$

When translating this formula to SPARQL we want to select mappings that contain a binding for all the variables in the query, so we explicitly request all of them in the select clause. For example, consider graph pattern $A_1 = \{\langle \text{?v3}, :a, \text{?v4} \rangle, \langle \text{?v3}, :b, :c \rangle\}$, which is interpreted as query:

```
SELECT ?v3 ?v4 WHERE { ?v3 :a ?v4 . ?v3 :b :c }
```

Query rewriting $\mathbb{Q}(A_1)$ then corresponds to:

```
SELECT ?v3 ?v4 WHERE {
  { {?v3 :a ?v4} UNION {:λ :a ?v4} UNION {?v3 :λ ?v4}
    UNION {?v3 :a :λ} UNION {:λ :λ ?v4} UNION {:λ :a :λ}
    UNION {?v3 :λ :λ} UNION {:λ :λ :λ} }
  { {?v3 :b :c} UNION {:λ :b :c} UNION {?v3 :λ :c}
    UNION {?v3 :b :λ} UNION {:λ :λ :c} UNION {:λ :b :λ}
    UNION {?v3 :λ :λ} UNION {:λ :λ :λ} } }
```

Below we treat $\mathbb{Q}(A)$ as a union of conjunctive queries, or UCQ [1], and denote $q \in \mathbb{Q}(A)$ to be a conjunctive query within it.

We should point out that in this section we present a generic formulation of both approaches that is applicable to schema graphs having variables in the predicate position. If variables cannot occur in this position, such as in the triplestore schemas representation of SHACL constraints, these approaches could be optimised; for example by removing from $\mathbb{Q}(A)$ all the triples patterns that have :λ in the predicate position.

Having defined how the critical and score approaches compute a set of mappings, we now describe the details of the last two phases required to compute a basic schema consequence.

**Filtering the mappings** This phase deals with processing the mappings computed by either critical or score, namely $[\![A]\!]_{\mathbb{C}(S,r)}$ or $[\![\mathbb{Q}(A)]\!]_{\$(S)}$. It should be noted that

it is not possible to simply apply the resulting mappings on the consequent of the inference rule, as such mappings might map a variable in the subject or predicate position to a literal, thus generating an invalid triple pattern. Moreover, it is necessary to determine which variables should be included in the no-literal set of the basic schema consequence. The schema $S'$ (output of our approaches) is initialised with the same graph and no-literal set as $S$, and with an empty set of existential rules. Formally $S'$ is initialised to $\langle S^G, S^\Delta, \varnothing \rangle$. We then incrementally extend $S'$ on a mapping-by-mapping basis until all the mappings have been considered, at which point, $S'$ is the final output of our basic schema expansion.

For each mapping $m$ in $[\![A]\!]_{\mathbb{C}(S,r)}$ or $[\![\mathbb{Q}(A)]\!]_{\$(S)}$, we do the following. We create a temporary no-literal set $\Delta^m$. This set will be used to keep track of which variables could not be bound to any literals if we evaluated our rule antecedent $A$ on the instances of $S$, or when instantiating the consequence of the rule. We initialise $\Delta^m$ with the variables of our inference rule $A \to C$ that occur in the subject or predicate position in some triple of $A$ or $C$, as we know that they cannot be matched to, or instantiated with literals.

We then consider the elements that occur in the object position in the triples $t_A$ of $A$. We take all the rewritings $t_q$ of $t_A$ in $\mathbb{Q}(A)$ (if using critical, it would be enough to consider a single rewriting $t_q$ with $t_q = t_A$). Since the mapping $m$ has been computed over the canonical instance ($\$(S)$ or $\mathbb{C}(S,r)$ depending on the approach), we know that there exists at least one $t_q$ such that $m(t_q)$ belongs to the canonical instance. We identify the set of schema triples $t^S \in S$ that model $m(t_q)$, for any of the above $t_q$. Intuitively, these are the schema triples that enable $t_A$, or one of its rewritings, to match the canonical instance with mapping $m$. If $t_A[3]$ is a literal $l$, or a variable mapped to a literal $l$ by $m$, we check if there exists any $t^S$ from the above such that $t^S[3] = l$ or $t^S[3]$ is a variable that allows literals (not in $S^\Delta$). If such a triple pattern does not exist, then $m(A)$ cannot be an instance of $S$ since it has a literal in a non-allowed position, and therefore we filter out $m$. If $t_A[3]$ is a variable mapped to $:\lambda$ in $m$, we check whether in any of the above $t^S$, $t^S[3]$ is a variable that allows literals (not in $S^\Delta$). If such $t^S$ cannot be found, we add variable $t_A[3]$ to $\Delta^m$. Intuitively, this models the fact that $t_A[3]$ could not have been bound to literal elements under this mapping. Having considered all the triples $t_A \in A$ we filter out mapping $m$ if it binds any variable in $\Delta^m$ to a literal. If $m$ is not filtered out, we say that inference rule $r$ is applicable, and we use $m$ to expand $S'$.

**Schema Expansion.** For each mapping $m$ that is not filtered out, we compute the substitution $s^m$, which contains all the bindings in $m$ that map a variable to a value other than $:\lambda$, and for every binding $?v \to :\lambda$ in $m$, a variable substitution $?v \to ?v^*$ where $?v^*$ is a new variable. We then add triple patterns $s^m(m(C))$ to $S'^G$ and then add the variables $s^m(\Delta^m) \cap vars(S'^G)$ to $S'^\Delta$. Although the schema consequences produced by score$(S,r)$ and critical$(S,r)$ might not be identical, they are semantically equivalent (i.e. they model the same set of instances). This notion of equivalence is captured by Theorem 1.

**Theorem 1** *For all rules $r : A \to C$ and triplestore schemas $S$,* $\mathbb{I}(\text{score}(S,r)) = \mathbb{I}(\text{critical}(S,r))$.

The score approach (and by extension critical, via Theorem 1) is sound and complete. The following theorem captures the this notion by stating the semantic equivalence of score$(S,r)$ and $r(S)$. For our proofs, we refer the reader to our appendix.[1]

**Theorem 2** *For all rules $r : A \to C$ and triplestore schemas S,* $\mathbb{I}(score(S,r)) = \mathbb{I}(r(S))$.

**Termination.** It is easy to see that our approaches terminate since our datalog rules do not contain existential variables, and do not generate new `IRI`s or literals (but just new variable names). After a finite number of iterations, either approach will only generate isomorphic (and thus equivalent) triple patterns.

## 5 Computing the Existential-Preserving Schema Consequence

In order to compute the existential-preserving schema consequence we are going to build on the result of our simple schema consequence. Recall the definition of schema consequences from Section 3.2 and note that given a schema $S = \langle S^G, S^\Delta, S^\exists \rangle$ and a set of inference rules $R$ such that $con(S,R) = \langle S'^G, S'^\Delta, \varnothing \rangle$ then $con^{ex}(S,R) = \langle S'^G, S'^\Delta, S'^\exists \rangle$ for some $S'^\exists \subseteq S^\exists$; that is, the output schema graph and no-literal set of the existential-preserving schema consequence are the same as those of the simple one.

Our first step is to compute the schema graph and no-literal set of the existential preserving schema consequence, as in Section 4. Next, and in the rest of this section, we want to compute the set of existential rules $S'^\exists$ that are still valid on all possible "closure" instances (instances of the original schema closed under $R$), or complementary, those existential rules that are violated on some "closure" instance.

Starting from an instance $I$ of $S$, which by definition satisfies $S^\exists$, an existential rule might become violated by the inference rules due to new facts added by the closure. Thus, the aim of the algorithm is to find an instance $I$ of $S$, that can "trigger" an existential rule $a \to^\exists c$ by mapping its antecedent $a$ on $clos(I,R)$. For every existential rule, we want to construct $I$ in a "minimal" way, so that if $clos(I,R)$ satisfies the rule $e$ then there is no proper subset $I'$ of $I$ which is still an instance of $S$ and does not satisfy the rule. By considering all such minimal instances $I$ for every existential rule, we can determine if the rule is violated or not on any potential closure of an instance.

We can achieve finding these violating instances if they exist, intituitively, by starting from triples that are: (1) groundings of the inference rules' antecedents; (2) instances of the original schema $S$; and (3) which produce, via the closure, a fact on which we can map $a$. To find the minimal number of inference rules' antecedents that we have to ground, we can reason "backwards" starting from an inference rule antecedent $A$ whose consequent can trigger $e$, and compute inference rules' antecedents that can compute $A$. We have implemented this backward-chaining reasoning in a way similar to query rewriting in OBDA [5], and the Query-Sub-Query algorithm in datalog [1]. We don't provide the specifics of the algorithm but emphasize that it terminates by relying on a notion of minimality of the rewritings produced. A rewriting produced by our algorithm is essentially a "transitive" antecedent via our inference rules, which can produce $A$. By instantiating these rule antecedents in one rewriting, that is also an instance of $\langle S^G, S^\Delta, \varnothing \rangle$, and "closing" it existentially[3] with $S^\exists$ we produce a "minimal" instance of the original schema on the closure of which we know we can find $A$. This $A$ is the antecedent of an inference rule that can infer facts matching the antecedent of $e$, and thus,

---

[3] For this step we implement a version of the Chase algorithm [4].

---

**Algorithm 1** Computation of the existential rules in $con^{ex}(S,R)$

---

1:  **procedure** RETAINEDEXISTENTIALS($S : \langle S^G, S^\Delta, S^\exists \rangle, R$)
2:      $V \leftarrow \varnothing$
3:      **for** each $e : a \rightarrow^\exists c \in S^\exists$ **do**
4:          **for** each $r : A \rightarrow C \in R$ **do**
5:              **if** $[\![\mathbb{Q}(a)]\!]_{\mathbb{S}(C)} \neq \varnothing$ **then**
6:                  $W \leftarrow$ all rewritings of the antecedent of $r : A \rightarrow C$ with rules $R$
7:                  **for** each $w : A^w \rightarrow C \in W$ **do**
8:                      $M^w \leftarrow [\![\mathbb{Q}(A^w)]\!]_{\mathbb{S}(S)}$
9:                      **for** each $m^w \in M^w$ **do**
10:                         $\tilde{m}^w \leftarrow$ all mappings in $m^w$ that do not map a variable to :$\lambda$
11:                         $g \leftarrow$ mapping from the $vars(\tilde{m}(A^w))$ to new IRIs
12:                         $I^g \leftarrow g(\tilde{m}(A^w))$
13:                         $I^{g'} \leftarrow \varnothing$
14:                         **while** $I^g \neq I^{g'}$ **do**
15:                             $I^{g'} \leftarrow I^g$
16:                             **for** each $e' : a' \rightarrow^\exists c' \in S^\exists$ **do**
17:                                 $M^{e'} \leftarrow [\![\mathbb{Q}(a')]\!]_{\mathbb{S}(I^g)}$
18:                                 **for** each $m' \in M^{e'}$ **do**
19:                                     **if** $[\![\mathbb{Q}(m'(c'))]\!]_{\mathbb{S}(I^g)} \neq \varnothing$ **then**
20:                                         $g^e \leftarrow$ mapping from $vars(m'(c))$ to new IRIs
21:                                         $I^g \leftarrow I^g \cup g^e(m'(c))$
22:                         $I^g \leftarrow clos(I^g, R)$
23:                         $M^I \leftarrow [\![\mathbb{Q}(a)]\!]_{\mathbb{S}(I^g)}$
24:                         **for** each $m^I \in M^I$ **do**
25:                             **if** $[\![\mathbb{Q}(m^I(c))]\!]_{\mathbb{S}(I^g)} \neq \varnothing$ **then**
26:                                 $V \leftarrow V \cup \{e\}$
        **return** $S^\exists \setminus V$

---

after applying this rule, we can check the satisfaction or violation of $e$. Our rewritings' groundings are representative of all possible instances whose closure can lead to a fact that the antecedent of $e$ maps to. If $e$ is valid in all these instances then $e$ can not be violated in any closure of an instance of $S$, and thus we retain it from $S^\exists$.

The pseudocode for our algorithm can be seen in Algorithm 1. For each existential rule $e$ we consider each inference rule $r : A \rightarrow C$ such that inferring $C$ could trigger $e$ (lines 3-5). We then compute all the rewritings $W$ of $A$ by means of backward-chaining the rules in $R$. For each such rewriting $A^w$, we want to see if we can match it on the instances of $S$. We do so by reusing the score approach, computing the set of mappings $M^w = [\![\mathbb{Q}(A^w)]\!]_{\mathbb{S}(S)}$. If $M^w$ is empty, then there is no instance of $S$ on which $A^w$ would match. Otherwise, we consider whether this potential match could violate the existential rule $e$ (lines 10-26). For each mapping $m^w \in M^w$ we compute the instance $I^g$, grounding of $A^w$, by first applying to $A^w$ all mappings in $m^w$ that do not map a variable to :$\lambda$, and then mapping any remaining variable to new IRIs (lines 10-12). To make sure that $I^g$ is an instance of $S$ we perform the Chase on $I^g$ using the existential rules. Lines 13 to 21 exactly implement the well-known Chase algorithm [4] to compute existential clo-

sure using our own `score` approach. Finally, we compute the closure on $I^g$ with the inference rules $R$ and, if it violates $e$, we add $e$ to the set $V$ of the existential rules that can be violated (lines 22-26). The output of our algorithm $(S'^\exists)$ is $S^\exists \setminus V$.

## 6   Experimental Evaluation

We developed a Java implementation of our algorithms. This allowed us to test their correctness with a number of test cases, and to assess their scalability using synthetic schemas of different sizes. We present here two experiments. In the first, we compare the time to compute the simple schema consequence, on different sized schemas, using the `score` and `critical` approaches. In the second, we show the overhead in computational time to compute the existential-preserving schema consequence. Since this overhead is the same, regardless of which approach we use to compute the simple schema consequence, we only consider `score` in this experiment.

We developed a synthetic schema and an inference rule generator that is configurable with 8 parameters: $\pi_C, |P|, |U|, |L|, |S^G|, |R|, |S^\exists|, n_A$, which we now describe. To reflect the fact that triple predicates are typically defined in vocabularies, our generator does not consider variables in the predicate position. Random triple patterns are created as follows. Predicate `IRI`s are randomly selected from a set of `IRI`s $P$. Elements in the subject and object position are instantiated as constants with probability $\pi_C$, or else as new variables. Constants in the subject positions are instantiated with a random `IRI`, and constants in the object position with a random `IRI` with 50% probability, or otherwise with a random literal. Random `IRI`s and literals are selected, respectively, from sets $U$ and $L$ ($U \cap P = \varnothing$). We consider chain rules where the triples in the antecedent join each other to form a list where the object of a triple is the same as the subject of the next. The consequent of each rule is a triple having the subject of the first triple in the antecedent as a subject, and the object of the last triple as object. An example of such inference rule generated by our experiment is: $\{\langle \texttt{?v0},\texttt{:m1},\texttt{?v1}\rangle, \langle \texttt{?v1}, \texttt{:m3},\texttt{?v2}\rangle\} \rightarrow \{\langle \texttt{?v0},\texttt{:m2},\texttt{?v2}\rangle\}$ In each run of the experiment we populate a schema $S = \langle S^G, S^\Delta, S^\exists \rangle$ and a set of inference rules $R$ having $n_A$ triples in the antecedent. To ensure that some inference rules in each set are applicable, half of the schema is initialized with the antecedents triples of randomly selected inference rules. The other half is populated with random triple patterns. Each existential rule of schema $S$ is created as follows. Its antecedent is selected randomly from all the consequents of the inference rules, while its consequence is selected randomly from all the antecedents of all the inference rules. This is done to ensure the relevance of the existential rules, and increase the likelihood of interactions with the inference rules. We initialize $S^\Delta$ with all the variables in the subject and predicate position in the triples of $S$. The code for these experiments is available on GitHub.[2] We run the experiments on a standard Java virtual machine running on Ubuntu 16.04 with 15.5 GB RAM, an Intel Core i7-6700 Processor. Average completion times of over 10 minutes have not been recorded.

The results of the first experiment are displayed in Fig. 4 (a). This figure shows the time to compute the schema consequence for different schema sizes $|S|$ using `score` and `critical`. The parameters have been chosen to be small enough to accommodate for the high computational complexity of the `critical` approach. This figure
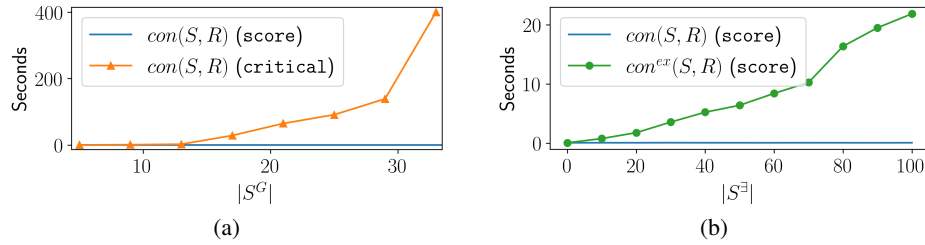
**Fig. 4.** (a) Average time to compute $con(S,R)$ using `score` and `critical` as the schema size $|S^G|$ grows. The other parameters are: $|P| = 1.5|S^G|$, $\pi_C = 0.1$, $|U| = |L| = |S^G|$, $|R| = 4$, $n_A = 2$, $|S^\exists| = 0$. (b) Average time to compute $con(S,R)$ and $con^{ex}(S,R)$ using the `score` approach as the number of existential rules $|S^\exists|$ increases. The other parameters are $|S| = 100$, $|P| = 110$, $\pi_C = 0.1$, $|U| = |L| = |S^G|$, $|R| = 20$, $n_A = 2$.

shows that `score` is orders of magnitude faster, especially on large schema sizes. The `critical` approach, instead, does not scale (times out) beyond schemas with over 33 triples. Fig. 4 (b) shows the increase of computation time as schemas with more existential rules are considered. The results of the second experiment show how our approach to compute the existential-preserving schema consequence can scale to a large number of existential rules on a large input schema in a matter of seconds.

## 7   Conclusion

SHACL constraints can can be used to define the schema of graph datasets. However, the application of inference rules could cause the violation of such constraints, and thus require a change in the schema. In this paper we address the problem of computing the *schema consequence* of a schema $S$ and a set of rules $R$; that is, the evolved schema of the graphs, instances of $S$, closed under inference rules $R$. To address this problem we introduced our notion of a *triplestore schema*, which captures a fragment of SHACL, and can also be used as a standalone logical tool to model properties of RDF graphs in general. We addressed the problem incrementally, first by computing a *simple* schema consequence that does not consider existential constraints. We presented two approaches to compute the simple schema consequence. The first is based on the pre-existing concept of a *critical instance*, while the second is a novel approach based on query rewriting and which our experiments showed to be significantly more efficient. We have then provided an approach to deal with existential constraints based on backward-chaining reasoning, which computes what we call an *existential-preserving* schema consequence. This can be considered the final output of our approach, which a domain expert can use to update the schema of an RDF dataset, if they choose to retain all the inference rules considered. The machinery we developed in the form of the simple schema consequence, can also have other applications, such as determining which rules are applicable on a dataset and, if they are, what kind of triples they can infer.

## Acknowledgments

## References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc. (1995)
2. Baget, J.F., Leclère, M., Mugnier, M.L., Salvat, E.: On rules with existential variables: Walking the decidability line. Artificial Intelligence **175**(9-10), 1620–1654 (2011)
3. Bassiliades, N.: SWRL2SPIN: A tool for transforming SWRL rule bases in OWL ontologies to object-oriented SPIN rules. CoRR (2018), http://arxiv.org/abs/1801.09061
4. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 37–52. ACM (2017)
5. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R.: Ontologies and Databases: The DL-Lite Approach, pp. 255–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
6. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering **1**(1), 146–166 (1989)
7. Corman, J., Reutter, J.L., Savković, O.: Semantics and Validation of Recursive SHACL. In: The Semantic Web – ISWC 2018. pp. 318–336 (2018)
8. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theoretical Computer Science **336**(1), 89–124 (2005)
9. Glimm, B., Kazakov, Y., Liebig, T., Tran, T.K., Vialard, V.: Abstraction refinement for ontology materialization. In: International Semantic Web Conference. pp. 180–195 (2014)
10. Knublauch, H., Kontokostas, D.: Shapes constraint language (SHACL). W3C Recommendation, W3C (2017), https://www.w3.org/TR/shacl/
11. Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R., Zaveri, A.: Test-driven Evaluation of Linked Data Quality. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 747–758. WWW '14, ACM (2014)
12. Lefrançois, M., Cox, S., Taylor, K., Haller, A., Janowicz, K., Phuoc, D.L.: Semantic Sensor Network Ontology. W3C Recommendation, W3C (2017), https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/
13. Marnette, B.: Generalized schema-mappings: from termination to tractability. In: Proc. of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symp. on Principles of database systems. pp. 13–22. ACM (2009)
14. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems **34**(3), 16:1–16:45 (2009)
15. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape Expressions: An RDF Validation and Transformation Language. In: Proceedings of the 10th International Conference on Semantic Systems. pp. 32–40. SEM '14, ACM (2014)