Rule Applicability on RDF Triplestore Schemas

Paolo Pareti[1] , **George Konstantinidis**[1] , **Timothy J. Norman**[1] and **Murat Şensoy**[2]
[1]University of Southampton, Southampton, United Kingdom
[2]Özyeğin University, Istanbul, Turkey

Abstract

Rule-based systems play a critical role in health and safety, where policies created by experts are usually formalised as rules. When dealing with increasingly large and dynamic sources of data, as in the case of Internet of Things (IoT) applications, it becomes important not only to efficiently apply rules, but also to reason about their applicability on datasets confined by a certain schema. In this paper we define the notion of a triplestore schema which models a set of RDF graphs. Given a set of rules and such a schema as input we propose a method to determine rule applicability and produce output schemas. Output schemas model the graphs that would be obtained by running the rules on the graph models of the input schema. We present two approaches: one based on computing a canonical (critical) instance of the schema, and a novel approach based on query rewriting. We provide theoretical, complexity and evaluation results that show the superior efficiency of our rewriting approach.

1 Introduction

Inference rules are a common tool in many areas where they are used, for example, to model access control policies [3] and business rules [10]. In this paper we are motivated by their use in Internet of Things (IoT) applications, where rules are often used to capture human decision making in a simple and straightforward way [21]. This is especially true in safety-critical domains, such as in the field of Occupational Health and Safety (OHS). OHS knowledge is codified by experts into policies, which are then translated into rules to monitor and regulate workplaces. For example, OHS regulations set limits on human exposure to certain gases. Monitoring systems can use these rules to determine, from sensor data, whether dangerous gas concentration levels have been reached, and trigger warnings or perform actions such as increasing ventilation. Real-world use cases have been provided by industrial partners, such as a wireless and electronic solutions company, and OHS policies from the International Labour Organisation [16].

An important limitation of current inference rule applications in the IoT domain is that they require expert human interventions not only to create rules, but also to manage them. This includes determining when they are applicable and what types of facts we can ultimately infer. In the gas-concentration example above, human experts would be employed to answer questions such as: could the rule that aggregates sensor data be used, maybe in conjunction with other rules, to determine whether an area should be evacuated? Is this rule applicable to the available data sources? And will this rule still be applicable after the underlying data sources change (e.g., in case sensors stop working or are substituted with others)? Knowing which rules can be applied and what type of facts can be inferred on a dataset can have safety critical implications, as OHS policies might depend on the availability of certain pieces of information. It is important to note that by executing rules on a specific dataset we only discover the facts that are currently entailed. To predict which facts could potentially be inferred in future versions of the dataset, we need to reason about its schema.

As IoT scenarios become increasingly complex and dynamic, managing rules in a timely and cost effective way requires improvements in automation. In this paper we present an approach that can answer these questions automatically, by reasoning about an abstraction of the available data sources, called the *triplestore schema*. We define triplestore schemas as abstract signatures of underlying data, similar to database schemas. Triplestore schemas can be defined by experts or, as we will see later, can be derived from the types of sensor available. Such schemas are *dynamic*, changing as the data sources change; e.g., a new sensor is added to the system creating triples with new predicates, and entailing other facts.

We consider `RDF` [8] triplestores and we model triplestore schemas as sets of `SPARQL` [13] *triple patterns*, which in some formal sense restrict or model the underlying `RDF` data. We express rules as `SPARQL construct` queries. This type of rules model `SPIN` [17] inference rules, which correspond to Datalog rules [6], and are also compatible with the monotonic subsets of other rule languages, such as `SWRL` [15]. Given an input triplestore schema and a set of rules, our objective is to decide whether the rules would apply on some `RDF` dataset modelled by this schema. We do so by computing the "output" or "consequence" schema of these hypothetical rule applications: this is the schema that models all possible `RDF` datasets that can be obtained by executing the rules on all datasets modeled by the input schema. It is worth not-

ing that our approach is only concerned with schemas, and it is compatible with relational datasets, as long as their schema and and rules can be expressed using RDF and SPARQL [5].

Reasoning at the schema level has been explored previously for databases [20] and Description Logics [11]. In fact, for a different but closely related problem of reasoning on database schemas (called *chase termination*), Marnette [20] employed a canonical database instance, called the *critical instance*, which is representative of all database instances of the given schema, on which we base one of our solutions.

We propose two approaches to reason about the applicability of inference rules on triplestore schemas. First, we re-use the critical instance for our triplestore schemas and develop an approach based on this representative RDF graph: running the rules on this graph produces evaluation mappings which, after careful manipulation in order to account for peculiarities of RDF literals, help produce our consequence schemas. When constructing the critical instance, as in the original case of relational databases, we need to place all constants appearing in our schema and rules in the constructed instance in many possible ways. This leads to a blowup in its size and so we turn our attention to finding a much smaller representative RDF graph, that we call the *sandbox* graph, and which we populate with only one "representative" element. We then develop a novel query rewriting algorithm that can compute the consequence schema on the sandbox graph. We provide correctness, complexity, and evaluation results and experimentally exhibit the efficiency and scalability of our rewriting-based approach: it surpasses the critical-instance methods by orders of magnitude while scaling to hundreds of rules and schema triples in times ranging from milliseconds to seconds.

2 Background

We consider triplestores containing a single RDF *graph*, without blank nodes. 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 URIs, $\mathbb{L}$ the set of all literals and $\mathbb{V}$ the set of all variables. We use the term *constants* to refer to both literals and URIs. A *graph pattern* is a set of *triple patterns* defined in: $(\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$. Given a pattern P, $vars(P)$ and $const(P)$ are the sets of variables and constants in the elements of P, respectively. We represent URIs as namespace-prefixed strings of characters, where a namespace prefix is a sequence of zero or more characters followed by a column 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 τ denoting throughout the paper indexes $\tau = \{1, 2, 3\}$.

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 $m(p)$).

Given a graph pattern P and a graph G, the SPARQL evaluation of P over G, denoted with $[\![P]\!]_G$, is a set of mappings as defined in [22]. 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 a 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. A single application of a rule $r : A \rightarrow C$ to a dataset I, denoted by $r(I)$, is $I \cup \bigcup_{m \in [\![A]\!]_I} \{m(C), \text{ if } m(C) \text{ is a valid RDF a graph}\}$. Rule notations such as SPIN and SWRL can be represented in this format [2]. The closure, or saturation, 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 $I_{i+1} = \bigcup_{r \in R} \{r(I_i)\}$.

3 Problem Description

To reason about schemas we need a simple language to model, and at the same time restrict, the type of triples that an RDF graph can contain. It should be noted that, despite the similarity in the name, the existing RDF *schema* (RDFS) vocabulary is used to describe ontological properties of the data, but not designed to restrict the type of triples allowed in the dataset. In this paper we define a *triplestore schema* (or just *schema*) S as a pair $\langle S^G, S^\Delta \rangle$, where S^G is a set of triple patterns, and S^Δ is a subset of the variables in S^G which we call the *no-literal* set. Intuitively, S^G defines the type of triples allowed in a database, 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^Δ defines which variables cannot be instantiated with literals, thus S^Δ must at least include all variables that occur in the subject or predicate position in S^G. For example, if $\langle ?v1, :a, ?v2 \rangle \in S'^G$ and $?v2 \notin S'^\Delta$, then the instances of schema S' can contain any triple that has :a as a predicate. If $\langle :b, :c, ?v3 \rangle \in S'^G$ and $?v3 \in S'^\Delta$, the instances of S' can contain any triple that has :b as a subject, :c as a predicate, and a URI as an object. To prevent the occurrence of complex interdependencies between variables, we restrict each variable to occur only once (both across triples, and within each triple) in S^G and in the rule consequents.

A graph I *is an instance of a schema* S if for every triple t^I in I there exists a triple pattern t^S in S^G, and a mapping m such that (1) $m(t^S) = t^I$ and (2) m does not bind any variable in S^Δ to a literal. In this case we say that S *models* graph I (and that each triple t^S models triple t^I). All instances of S are denoted by $\mathbb{I}(S)$. We say that two schemas S and S' are semantically equivalent if they model the same set of instances (formally, if $\mathbb{I}(S) = \mathbb{I}(S')$). Notice that any subset of an instance of a schema is still an instance of that schema. A rule r within a set of rules R is *applicable* with respect to a triplestore schema S if there exists a graph I instance of S, such that the precondition of r matches $clos(I, R - r)$.

Consider the following example scenario of a mine

where sensors produce data modelled according to the Semantic Sensor Network Ontology (SSN) [19], with namespace sosa. In SNN, sensor measurements are called *observations*. A simple approach to create the schema of a dataset is the following. A dataset that can be populated with sensor measurements of a property :x (e.g., temperature) can be defined with triple pattern [?v1, sosa:observedProperty, :x]. Pattern [?v1, sosa:hasResult, ?v2] indicates that the results of these measurements are collected and pattern [?v1, sosa:hasFeatureOfInterest, :y] indicates that the results are applicable to a specific entity :y (e.g., a room). Similar patterns are presented in [7] in the context of converting CSV data into the SNN format . In this example, the sensors are deployed only in one tunnel, namely :TunnelA, and schema S_1 is:

S_1^G = {[?v1, sosa:observedProperty, :CO_Danger],
[?v2, sosa:observedProperty, :WorkerTag],
[?v3, sosa:hasFeatureOfInterest, :TunnelA],
[?v5, sosa:hasResult, ?v4]}
S_1^Δ = {?v1, ?v2, ?v3, ?v5}

We now consider instance I_1 of schema S_1. In this instance, the sensors in tunnel A observed both a dangerous gas concentration, represented by the value "1", and the presence of worker :John.

I_1 = {⟨:o1, sosa:observedProperty, :CO_Danger⟩,
⟨:o1, sosa:hasFeatureOfInterest, :TunnelA⟩,
⟨:o1, sosa:hasResult, "1"⟩,
⟨:o2, sosa:observedProperty, :WorkerTag⟩,
⟨:o2, sosa:hasFeatureOfInterest, :TunnelA⟩,
⟨:o2, sosa:hasResult, :John⟩}

Consider two rules r_1 and r_2. The first one detects when workers trespass on an "off-limit" area, and the second one labels areas with dangerous gas concentrations as "off-limit".

r_1 = {[?v1, sosa:observedProperty, :WorkerTag],
[?v1, sosa:hasFeatureOfInterest, ?v2],
[?v1, sosa:hasResult, ?v3],
[?v2, rdf:type, :OffLimitArea]}
→ {[?v2, rdf:type, :TrespassedArea]}
r_2 = {[?v1, sosa:observedProperty, :CO_Danger],
[?v1, sosa:hasFeatureOfInterest, ?v2],
[?v1, sosa:hasResult, "1"]}
→ {[?v2, rdf:type, :OffLimitArea]}

Since the precondition of rule r_2 matches dataset I_1, we can apply the rule and derive a new fact: [:TunnelA, rdf:type, :OffLimitArea]. On the instance extended by this new fact, rule r_1 is applicable and adds [:TunnelA, rdf:type, :TrespassedArea].

Our approach relies on being able to decide which rules are applicable on a specific triple store schema, e.g., S_1, in absence of any particular instance, e.g., I_1. Since the precondition of rule r_2 matches dataset I_1, this rule is directly applicable on schema S_1, and we would like to be able to decide this by only looking at S_1. Moreover if we can decide this and extend schema S_1 with a triple pattern that is the schema of [:TunnelA, rdf:type, :OffLimitArea] (in this case that schema would be the same triple itself), then we would able to reason with this new schema and decide that rule r_1 is also applicable. In practice, what we would like to do is to compute a schema that captures all consequences of applying our set of rules on any potential instance.

The following definition captures this intuition. Given a schema S and a set of rules R, a schema S' is a schema consequence of S with respect to R, denoted $con(S,R)$, if $\mathbb{I}(S') = \bigcup_{I\in\mathbb{I}(S)}\{I' \mid I' \subseteq clos(I,R)\}$. We can notice that since every subset of an instance of a schema is still an instance of that schema, a dataset can contain the consequence of a rule application without containing a set of triples matching the antecedent. This situation is commonly encountered when some triples are deleted after an inference is made.

Keeping track of the schema consequences allows us to directly see which rules are applicable to instances of a schema without running the rules on the data. In correspondence to a single rule application $r(I)$, of a rule r on an instance I, we define a *basic consequence* of a schema S by a 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 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 the following section we focus on the core of our problem which is 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)).

4 Computing the Basic Schema Consequence

Given a schema S and a rule $r : A \rightarrow C$, 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 [20] (and similar notions in the area of Description Logics [11]). 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 fresh constant not in S^G or A. In [20] this instance is used in order to decide Chase termination; Chase is referred to rule inference with *existential* rules, 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 actually 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 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 fresh URI :λ such that :$\lambda \notin const(S^G) \cup const(A)$. Formally, the critical instance $\mathbb{C}(S, A \rightarrow C)$ is the

set of triples:

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

The critical instance replaces variables with URIs 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^Δ with a literal). In order to compute the triples of our basic schema consequence for 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 stretch 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 above we present a novel alternative solution based on query rewriting, called score. This alternative 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 fresh URI $:\lambda$. This results in an instance with the same number of triples as S^G. Formally, a sandbox graph $\mathbb{S}(S)$ is the set of triples:

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

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 $\mathbb{S}(S) \in \mathbb{I}(S)$ and $\mathbb{S}(S) \subseteq \mathbb{C}(S,r)$. As an example, consider the sandbox graph of schema S_1 from Section 3:

$\mathbb{S}(S_1)$ = {[:λ, sosa:observedProperty, :CO_Danger],
[:λ, sosa:observedProperty, :WorkerTag],
[:λ, sosa:hasFeatureOfInterest, :TunnelA],
[:λ, sosa:hasResult, :λ]}

The critical instances $\mathbb{C}(S_1, r_1)$ and $\mathbb{C}(S_1, r_2)$ from our example would contain all the triples in $\mathbb{S}(S_1)$, plus any other triple obtained by substituting some variables with constants other than $:\lambda$, such as the triple: [:λ, sosa:hasResult, :OffLimitArea]}. A complete example of $\mathbb{C}(S_1, r_2)$ is available in an external appendix.[1]

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 $:\lambda$. Formally, $\mathbb{Q}(A)$ is the conjunction of disjunctions of triple patterns:

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

[1] https://github.com/paolo7/ap1/blob/master/Ap.pdf

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 $A_1 = \{\langle ?v3, :a, ?v4\rangle, \langle ?v3, :b, :c\rangle\}$, 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 $q \in \mathbb{Q}(A)$ a conjunctive query within it.

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 of the mappings This phase deals with processing the mappings computed by either critical or score, namely $[\![A]\!]_{\mathbb{C}(S,r)}$ or $[\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$. It should be noted that it is not possible to simply apply the resulting mappings on the consequent of the 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 (i.e. $S'^G = S^G$, $S'^\Delta = S^\Delta$). We then incrementally extend S' on a mapping-by-mapping basis until all the mappings have been considered, at which point, S' represents the final output of our approach.

For each mapping m in $[\![A]\!]_{\mathbb{C}(S,r)}$ or $[\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$, we do the following. We create a temporary no-literal set Δ^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 Δ^m with all the variables of our rule $A \rightarrow C$ 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.

Then, we 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 ($\mathbb{S}(S)$ 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 compute the set of schema triples t^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^Δ). If such triple pattern doesn't exist, then $m(A)$ cannot be an instance of S since it has a literal in

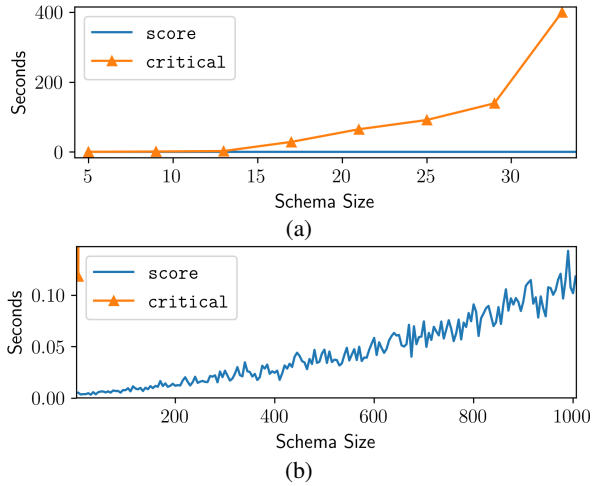

Figure 1: Average time to compute 20 schema consequences using `score` and `critical` as the schema size $|S|$ grows. The other parameters are: $|P| = 1.5|S|$, $\pi_C = 0.1$, $|U| = |L| = |S|$, $|R| = 4$, $n_A = 2$. Due to large difference in performance, subplots (a) and (b) focus, respectively, on `critical` and `score`.

an non-allowed positions, and therefore we filter out or *disregard m*. If $t_A[3]$ is a variable mapped to $:\lambda$ in m, we check whether there exists a t^S such that $t^S[3]$ is a variable that allows literals (not in S^Δ). If such t^S cannot be found, we add variable $t_A[3]$ to Δ^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 Δ^m to a literal. If m hasn't been filtered out, we say that 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 $?v \rightarrow :\lambda$ in m, a variable substitution $?v \rightarrow ?v^*$ where $?v^*$ is a fresh 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'^Δ.

Although the schema consequences produced by `score`(S, r) and `critical`(S, r) might not be identical, they are semantically equivalent. This notion of equivalence is captured by the following theorem.

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

The `score` approach (and by extension also `critical`, by Theorem 1) is sound and complete. The following theorem captures this notion by stating the semantic equivalence of `score`(S, r) and $r(S)$.

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

For our proofs, we refer the reader to an external appendix.[1]

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

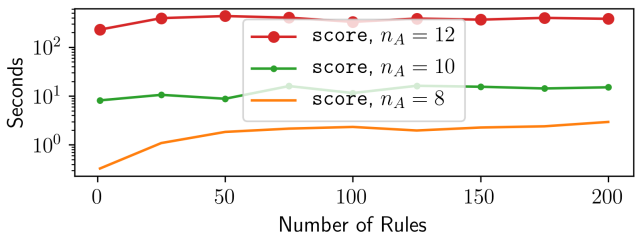

Figure 2: Average time to compute 20 schema consequences using `score` as the number of rules $|R|$ grows, for three configuration of n_A. The other parameters are set as follows: $|S| = 50$, $|P| = 60$, $\pi_C = 0.1$, $|U| = |L| = |S|$.

Complexity. Our central problem in this paper, computing the schema consequence for a set of rules, can be seen as a form of datalog evaluation [1] on our critical or sandbox instance. Datalog has been extensively studied in databases and its corresponding evaluation decision problem (whether a given tuple is in the answer of a datalog program) is known to be EXPTIME-complete in the so called combined complexity [25], and PTIME-complete in data complexity [25; 9]. Data complexity in databases refers to the setting in which the query (or datalog program) is considered fixed and only the data is considered an input to the problem. In our setting, data complexity refers to the expectation that the overall number and complexity of the rules remains small. It is not hard to see that the corresponding decision problem, stated below, remains PTIME-complete in data complexity. The intuition is that once we construct the critical instance in polynomial time (or alternatively, we grow our set of rules by a polynomial rule rewriting) we have essentially an equivalent problem to datalog evaluation (our rules being the datalog program, and the canonical instance being the database).

Theorem 3 *Given triple pattern $t_{S'}$ and schema S as inputs, the problem of deciding whether $t_{S'}$ is in the consequence schema of S for a fixed set of rules R is PTIME-complete.*

5 Experimental Evaluation

We developed a Java implementation of the `score` and `critical` approaches and evaluated them on synthetic datasets to compare their scalability. We developed a synthetic schema and rule generator that is configurable with 7 parameters: $\pi_C, |P|, |U|, |L|, |S|, |R|, n_A$, 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 `URI`s are randomly selected from a set of `URI`s P. Elements in the subject and object position are instantiated as constants with probability π_C, or else as new variables. Constants in the subject positions are instantiated with a random `URI`, and constants in the object position with a random `URI` with 50% probability, or otherwise with a random literal. Random `URI`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 rule generated by our experiment is: $\{\langle ?v0, :m1, ?v1\rangle, \langle ?v1, :m3, ?v2\rangle\} \rightarrow \{\langle ?v0, :m2, ?v2\rangle\}$ In each run of the experiment we populate a schema S and a set of rules R having n_A triples in the antecedent. To ensure that some rules in each set are applicable, half of the schema is initialized with the antecedents triples of randomly selected rules. The other half is populated with random triple patterns. We initialize S^{Δ} with all the variables in the subject and predicate position in the triples of S. The code used in this experiments is avaliable on GitHub;[2] it uses Apache Jena[3] to handle query execution. We run the experiments on a standard Java virtual machine running on an Ubuntu 16.04 computer with 15.5 GB RAM, an Intel Core i7-6700 Processor. Average completion times of over 10 minutes have not been recorded.

Figure 1 shows the time to compute the schema consequence for different schema sizes $|S|$ using `critical` and `score`. The parameters have been chosen to be small enough to accommodate for the high computational complexity of the `critical` approach. This figure shows that `score` is orders of magnitude faster, especially on large schema sizes. The `critical` approach, instead, times out for schema sizes of over 33 triples.

Figure 2 shows the time to compute the schema consequence for different antecedent sizes n_A and rule numbers $|R|$. The `critical` approach is not present in this figure, as it timed out in all the configurations. As this figure shows, the `score` approach can easily scale to a large set of rules. Given the complexity of `SPARQL` query answering [22], we can also notice an exponential increase in computation time as more triples are added to the antecedent of a rule. In our experiment setup, the `score` approach scales to rules with antecedent sizes of up to 12 triples, before timing out.

6 Related Work

To the best of the authors' knowledge, our approach is the first to determine the applicability of inference rules to types of `RDF` triplestores specified by their schema, and to expand their schema with the potential consequences of such rules. Unlike related work on provenance paths for query inferences [14], we do not explicitly model the dependencies between different rules. Instead, we compute their combined potential set of inferences by expanding the original schema on a rule-by-rule basis, through multiple iterations, following the basic principles of the chase algorithm. We choose to follow a subset of the `RDF` data model, and not a simpler graph model such as *generalised* `RDF` [8], to make our approach more applicable in practice, and compatible with existing tools. We pay particular attention to literals, as they are likely to feature prominently in IoT sensor measurements.

A possible application of our approach is to facilitate the sharing and reusing of inference rules. A relevant initiative in the IoT domain is Sensor-based Linked Open Rules (`S-LOR`) [12], which provides a comprehensive set of tools to deal with rules, including a rule discovery mechanism. By classifying rules according to sensor types, domain experts can discover and choose which inference rules are most relevant in a given scenario. Our approach could automate parts of this process, by selecting rules applicable to the available data sources. We refer to [24] for a comprehensive review of rule-based reasoning systems applicable to IoT.

Our approach to define a triplestore schema is related to a number of similar languages, and in particular to Shape Expressions (ShEx) [23] and the Shapes Constraint Language (SHACL) [18]. The term *shape*, in this case, refers to a particular constraint on the structure of an `RDF` graph. ShEx and SHACL can be seen as very expressive schema languages, and computing schema consequences using such schemas would be impractical. In fact, each inference step would need to consider complex interpendencies between shapes and the values allowed in each triple element, and thus we would generate increasingly larger sets of contrainsts. The triplestore schema proposed in this paper is a simpler schema language and, if we disallow variables in the predicate position, can be modelled as a subset of both ShEx and SHACL.

7 Conclusion

As its main contribution, this paper presented two approaches to determine the applicability of a set rules with respect to a database schema (i.e. if the rule could ever match on any dataset modelled by the schema), by expanding such schema to model the potential facts that can be inferred using those rules, which we call *schema consequence*. This can be applied in IoT scenarios, where inference rules are used to aggregate sensor readings from diverse data sources in order to automate health and safety policies. As the underlying data sources evolve, it is important to determine whether rules are still applicable, and what they can infer.

We focused on `RDF` triplestores, and on inference rules that can be modelled as `SPARQL` queries, such as `SPIN` and `SWRL`. To do so, we defined a notion of a *triplestore schema* that constrains the type of triples allowed in a graph. This differs from the `RDF` *schema* (`RDFS`) specification, which is designed to define vocabularies. While we provide an example on how to describe the schema of simple sensor networks, we think extending this approach to more expressive schema languages could be an interesting venue for future work.

The first of the two approaches that we presented is based on the existing notion of a critical instance; the second on query rewriting. We have theoretically demonstrated the functional equivalence of the approaches, as well as their soundness and completeness. Moreover, we have provided experimental evidence of the superior scalability of the second approach, which can be applied over large schemas and rulesets within seconds.

With this paper we intend to provide a first theoretical framework to reason about rule applicability. We hope that our approach will pave the way, on one hand, for efficient and meaningful policy reasoning in IoT systems, and on the other, for new and interesting rewriting-based schema reasoning approaches in the knowledge representation and databases research areas.

[2] https://github.com/paolo7/ap2

[3] https://jena.apache.org/

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc. (1995)

[2] Bassiliades, N.: SWRL2SPIN: A tool for transforming SWRL rule bases in OWL ontologies to object-oriented SPIN rules. CoRR **abs/1801.09061** (2018), http://arxiv.org/abs/1801.09061

[3] Beimel, D., Peleg, M.: Editorial: Using owl and swrl to represent and reason with situation-based access control policies. Data Knowl. Eng. **70**(6), 596–615 (2011)

[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., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL queries over relational databases. Semantic Web **8**(3), 471–487 (2017)

[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] Chaochaisit, W., Sakamura, K., Koshizuka, N., Bessho, M.: CSV-X: A Linked Data Enabled Schema Language, Model, and Processing Engine for Non-Uniform CSV. 2016 IEEE Int. Conf. on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) pp. 795–804 (2016)

[8] Cyganiak, R., Wood, D., Markus Lanthaler, G.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C (2014), http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[9] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys (CSUR) **33**(3), 374–425 (2001)

[10] Fortineau, V., Fiorentini, X., Paviot, T., Louis-Sidney, L., Lamouri, S.: Expressing formal rules within ontology-based models using SWRL: an application to the nuclear industry. International Journal of Product Lifecycle Management **7**(1), 75–93 (2014), pMID: 65458

[11] Glimm, B., Kazakov, Y., Liebig, T., Tran, T.K., Vialard, V.: Abstraction refinement for ontology materialization. In: International Semantic Web Conference. pp. 180–195. Springer (2014)

[12] Gyrard, A., Serrano, M., Jares, J.B., Datta, S.K., Ali, M.I.: Sensor-based Linked Open Rules (S-LOR): An Automated Rule Discovery Approach for IoT Applications and Its Use in Smart Cities. In: 26th International Conference on World Wide Web Companion. pp. 1153–1159. WWW '17 (2017)

[13] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation, W3C (2013), https://www.w3.org/TR/sparql11-query/

[14] Hecham, A., Bisquert, P., Croitoru, M.: On the Chase for All Provenance Paths with Existential Rules. In: Rules and Reasoning. pp. 135–150. Springer International Publishing (2017)

[15] Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., et al.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission, W3C (2004), https://www.w3.org/Submission/SWRL/

[16] International Labour Organization: Act No. 6331 on Occupational Health and Safety (2012), https://www.ilo.org/dyn/natlex/natlex4.detail?p_lang=fr&p_isn=92011

[17] Knublauch, H.: SPIN - SPARQL Syntax. W3C Member Submission, W3C (2011), http://www.w3.org/Submission/spin-sparql/

[18] Knublauch, H., Kontokostas, D.: Shapes constraint language (SHACL). W3C Recommendation, W3C (2017), https://www.w3.org/TR/shacl/

[19] 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/

[20] 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)

[21] Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context Aware Computing for The Internet of Things: A Survey. IEEE Communications Surveys Tutorials **16**(1), 414–454 (2014)

[22] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems **34**(3), 16:1–16:45 (2009)

[23] 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)

[24] Serrano, M., Gyrard, A.: A Review of Tools for IoT Semantics and Data Streaming Analytics. Building Blocks for IoT Analytics p. 139 (2016)

[25] Vardi, M.Y.: The Complexity of Relational Query Languages (Extended Abstract). In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. pp. 137–146. STOC '82, ACM, New York, NY, USA (1982)