Access Control Enforcement for Selective Disclosure of Linked Data

Tarek Sayah, Emmanuel Coquery, Romuald Thion, Mohand-Saïd Hacid

Access Control Enforcement for Selective Disclosure of Linked Data – Authors' Version

Tarek Sayah, Emmanuel Coquery, Romuald Thion, and Mohand-Saïd Hacid

Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205 F-69622, France
{tarek.sayah,emmanuel.coquery,
romuald.thion,mohand-said.hacid}@liris.cnrs.fr

Abstract. The Semantic Web technologies enable Web-scaled data linking between large RDF repositories. However, it happens that organizations cannot publish their whole datasets but only some subsets of them, due to ethical, legal or confidentiality considerations. Different user profiles may have access to different authorized subsets. In this case, selective disclosure appears as a promising incentive for linked data. In this paper, we show that modular, fine-grained and efficient selective disclosure can be achieved on top of existing RDF stores. We use a data-annotation approach to enforce access control policies. Our results are grounded on previously established formal results proposed in [3]. We present an implementation of our ideas and we show that our solution for selective disclosure scales, is independent of the user query language, and incurs reasonable overhead at runtime.

Keywords: RDF; Authorization; Enforcement; Linked Data

1 Introduction

The Linked Data movement [6] (aka Web of Data) is about using the Web to create typed links between data from different sources. Technically, Linked Data refers to a set of best practices for publishing and connecting structured data on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets [5]. Linking data distributed across the Web requires a standard mechanism for specifying the existence and meaning of connections between items described in this data. This mechanism is provided by the Resource Description Framework (RDF). Multiple datastores that belong to different thematic domains (government, publications, life sciences, etc) publish their RDF data on the web[1]. The size of the Web of Data is estimated to about 85 billions of RDF triples (statements) from more than 3400 open data sets [2]. One of the challenges of the Linked Data is to encourage businesses and organizations worldwide to publish their RDF data into the linked data global space.

[1] http://lod-cloud.net/
[2] http://stats.lod2.eu

Indeed, the published data may be sensitive, and consequently, data providers may avoid to release their sensitive information, unless they are certain that the desired access rights of different accessing entities are enforced properly. Hence the issue of securing RDF content and ensuring the selective exposure of information to different classes of users is becoming all the more important. Several works have been proposed for controlling access to RDF data. In [3], the authors proposed a fine-grained access control model with a declarative language for defining authorization policies (we call this model `AC4RDF` in the rest of this paper).

Our enforcement framework allows to define multi-subject policies with a global set of authorizations $\mathscr{A}$. A subset $\mathscr{A}_s \subseteq \mathscr{A}$ of authorizations is associated to each subject S who executes a (SPARQL) query. The subject's policy is then enforced by `AC4RDF` which computes the positive subgraph of the authenticated subject. We use an annotation based approach to enforce multi-subject policies: the idea is to materialize every triple's applicable authorizations of the global policy, into a bitset which is used to annotate the triple. The base graph G is materialized into a graph $G^{\mathscr{A}}$ by annotating every triple $t \in G$ with a bitset representing its set of applicable authorizations $\mathsf{ar}(G, \mathscr{A})(t) \subseteq \mathscr{A}$. The subjects are similarly assigned to a bitset which represents the set of authorizations assigned to them. When a subject sends a query, the system evaluates it over the her/his positive subgraph. In Section 3 we give an overview about RDF data model and SPARQL query language. In Section 4 we give the semantics of `AC4RDF` model which are defined using positive subgraph from the base graph. In Section 5 we propose an enforcement approach of `AC4RDF` model in multiple-subject context. We present and prove the correctness of our encoding approach. In Section 6 we give details about the implementation and experimental results.

2 Related work

The enforcement techniques can be categorized into three approaches: *pre-processing*, *post-processing* and *annotation based*.

- The *pre-processing* approaches enforce the policy before evaluating the query. For instance, the query rewriting technique consists of reformulating the user query using the access control policy. The new reformulated query is then evaluated over the original data source returning the accessible data only. This technique was used by Costabello et *al.* [7] and Abel et *al.* [1].
- In the *post-processing* approaches, the query is evaluated over the original data source. The result of the query is then filtered using the access control policy to return the accessible data. Reddivari et *al.* [16] use a post-processing approach to enforce their models.
- In the *annotation based* approaches, every triple is annotated with the access control information. During query evaluation, only the triples annotated with a permit access are returned to the user. This technique is used by Papakonstantinou et *al.* [13], Jain et *al.* [11], Lopes et *al.* [12] and Flouris et *al.* [8].

The advantage of the pre-processing approaches such as query rewriting, is that the policy enforcer is independent from RDF data. In other words, any updates on data would not affect the policy enforcement. On the other hand, this technique fully depends on the the query language. Moreover, the query evaluation time may depend on the policy. The experiments in [1] showed that the query evaluation overhead grows when the number of authorization grows, in contrast to our solution which does not depend on the number of authorizations. In the post-processing approaches, the query response time may be considerably longer since policies are enforced after all data (allowed and not allowed) have been processed. The data-annotation approach gives a fast query answering, since the triples are already annotated with the access information and only the triples with a grant access can be used to answer the query. On the other hand, any updates in the data would require the re-computation of annotations.

Some works [13] support incremental re-computation of the annotated triples after updates. In this paper, we do not handle updates and we leave the incremental re-computation as future work.

In the data-annotation based approaches that hard-code the conflict resolution strategy [8], annotations are fully dependent on the used strategy so they need to be recomputed in case of change of the strategy. Our encoding is independent of the conflict resolution strategy function which is evaluated at query time, which means that changing the strategy does not impact the annotations.

As the semantics of an RDF graph are given by its closure, it is important for an access control model to take into account the implicit knowledge held by this graph. In the Semantic Web context, the policy authorizations deny or allow access to triples whether they are implicit or not. In [16] the implicit triples are checked at query time. Inference is computed during every query evaluation, and if one of the triples in the query result could be inferred from a denied triple, then it is not added to the result. Hence the query evaluation may be costly since there is a need to use the reasoner for every query to compute inferences. To protect implicit triples, [12], [11] and [13] proposed a propagation technique where the implicit triples are automatically labeled on the basis of the labels assigned to the triples used for inference. Hence if one of the triples used for inference is denied, then the inferred triple is also denied. This introduces a new form of inference anomalies where if a triple is explicit (materialized) then it is allowed, however, if the triple is inferred then it is denied. We illustrate with the following example.

Example 1. Let us consider the graph G_0 of Fig. 1. Suppose we want to protect G_0 by applying the policy $P = \{$*deny access to triples with type*$:$*Cancerous*, *allow access to all resources which are instance of*$:$*Patient*$\}$. The triple t_9 is inferred from t_2 and t_7 using the RDFS subClassOf inheretence rule. With the propagation approaches which consider inference [12,13,11], the triple $t_9 = (:alice\ ;\ rdf:type\ ;:Patient)\}$ will be denied since it is inferred from denied triples (t_7). Hence the fact that alice is a patient will not be returned in the result even though the policy clearly allows access to it.

In our model, explicit and implicit triples are handled homogeneously to avoid this kind of inference anomalies.

3 RDF data model

"Graph database models can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors" [2]. The graph data model used in the semantic web is RDF (*Resource Description Framework*) [9]. RDF allows decomposition of knowledge in small portions called *triples*. A triple has the form "(*subject* ; *predicate* ; *object*)" built from pairwise disjoint countably infinite sets I, B, and L for IRIs (*Internationalized Resource Identifiers*), blank nodes, and literals respectively. The subject represents the resource for which information is stored and is identified by an IRI. The predicate is a property or a characteristic of the subject and is identified by an IRI. The object is the value of the property and is represented by an IRI of another resource or a literal. In RDF, a resource which do not have an IRI can be identified using a blank node. Blank nodes are used to represent these unknown resources, and also used when the relationship between a subject node and an object node is n-ary (as is the case with collections). For ease of notation, in RDF, one may define a *prefix* to represent a namespace, such as `rdf` : `type` where `rdf` represents the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns`.

Note 1. In this paper, we explicitly write `rdf` and `rdfs` when the term is from the RDF or the RDFS standard vocabulary. However, we do not prefix the other terms for the sake of simplicity.

For instance the triple (:*alice* ;:*hasTumor* ;:*breastTumor*) states that alice has a breast tumor. A collection of RDF triples is called an *RDF Graph* and can be intuitively understood as a directed labeled graph where resources represent the nodes and the predicates the arcs as shown by the example RDF graph G_0 in Fig. 1.

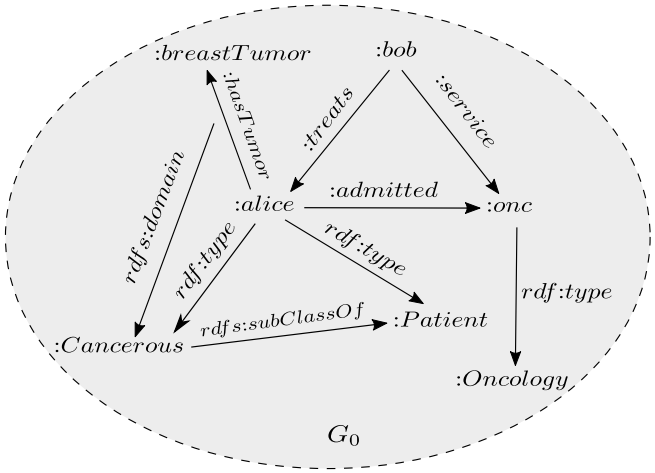
Definition 1. *(RDF graph) An RDF graph (or simply "graph", where unambiguous) is a finite set of RDF triples.*

Example 2. Fig. 1 depicts a graph G_0 constituted by triples t_1 to t_9, both pictorially and textually.

We reuse the formal definitions and notation used by Pérez and Gutierrez [14]. Throughout this paper, $\mathscr{P}(\mathsf{E})$ denotes the *finite powerset* of a set E and $\mathsf{F} \subseteq \mathsf{E}$ denotes a *finite subset* F of a set E.

3.1 SPARQL

An RDF query language is a formal language used for querying RDF triples from an *RDF store* also called *triple store*. An RDF store is a database specially



	Subject	Predicate	Object
t_1	$:hasTumor$	$rdfs:domain$	$:Cancerous$
t_2	$:Cancerous$	$rdfs:subClassOf$	$:Patient$
t_3	$:onc$	$rdf:type$	$:Oncology$
t_4	$:alice$	$:hasTumor$	$:breastTumor$
t_5	$:bob$	$:service$	$:onc$
t_6	$:bob$	$:treats$	$:alice$
t_7	$:alice$	$rdf:type$	$:Cancerous$
t_8	$:alice$	$:admitted$	$:onc$
t_9	$:alice$	$rdf:type$	$:Patient$

Fig. 1: An example of an RDF graph G_0

designed for storing an retrieving RDF triples. SPARQL (*SPARQL Protocol and RDF Query Language*) is a W3C recommendation which has established itself as the *de facto* language for querying RDF data. SPARQL borrowed part of its syntax from the popular and widely adopted SQL (*Structured Query Language*). The main mechanism for computing query results in SPARQL is subgraph matching: RDF triples in both the queried RDF data and the query patterns are interpreted as nodes and edges of directed graphs, and the resulting query graph is matched to the data graph using variables.

Definition 2. *(Triple Pattern, Graph Pattern) A* term *t is either an IRI, a variable or a literal. Formally $t \in \mathsf{T} = \mathsf{I} \cup \mathsf{V} \cup \mathsf{L}$. A tuple $t \in \mathsf{TP} = \mathsf{T} \times \mathsf{T} \times \mathsf{T}$ is called a* Triple Pattern (TP)*. A* Basic Graph Pattern (BGP)*, or simply a* graph*, is a finite set of triple patterns. Formally, the set of all BGPs is $\mathsf{BGP} = \mathscr{P}(\mathsf{TP})$.*

Given a triple pattern $tp \in \mathsf{TP}$, $\mathsf{var}(tp)$ is the set of variables occurring in tp. Similarly, given a basic graph pattern $B \in \mathsf{BGP}$, $\mathsf{var}(B)$ is the set of variables occurring in the BGP defined by $\mathsf{var}(B) = \{v \mid \exists tp \in B \wedge v \in \mathsf{var}(tp)\}$.

In this paper, we do not make any formal difference between a basic graph pattern and a graph. Blank nodes are replaced by variables since they are semantically equivalent to existentially quantified variables [15]. Moreover, we use an extended version of RDF [10] which allows variables in property position. When graph patterns are considered as instances stored in an RDF store, we simply call them *graphs*.

The evaluation of a graph pattern B on another graph pattern G is given by mapping the variables of B to the terms of G such that the structure of B is preserved. First, we define the substitution mappings as usual. Then, we define the evaluation of B on G as the set of substitutions that embed B into G.

Definition 3. *(Substitution Mappings) A* substitution (mapping) η *is a partial function* $\eta : \mathsf{V} \to \mathsf{T}$. *The domain of* η, $\mathsf{dom}(\eta)$, *is the subset of* V *where* η *is defined. We overload notation and also write* η *for the partial function* $\eta^\star : \mathsf{T} \to \mathsf{T}$ *that extends* η *with the identity on terms. Given two substitutions* η_1 *and* η_2, *we write* $\eta = \eta_1\eta_2$ *for the substitution* $\eta : ?v \mapsto \eta_2(\eta_1(?v))$ *when defined.*

Given a triple pattern $tp = (s\,;p\,;o) \in \mathsf{TP}$ *and a substitution* η *such that* $\mathsf{var}(tp) \subseteq \mathsf{dom}(\eta)$, $(tp)\eta$ *is defined as* $(\eta(s)\,;\eta(p)\,;\eta(o))$. *Similarly, given a graph pattern* $B \in \mathsf{BGP}$ *and a substitution* η *such that* $\mathsf{var}(B) \subseteq \mathsf{dom}(\eta)$, *we extend* η *to graph pattern by defining* $(B)\eta = \{(tp)\eta \mid tp \in B\}$.

Definition 4. *(BGP Evaluation) Let* $G \in \mathsf{BGP}$ *be a graph, and* $B \in \mathsf{BGP}$ *a graph pattern. The evaluation of* B *over* G *denoted by* $[\![B]\!]_G$ *is defined as the following set of substitution mappings:*

$$[\![B]\!]_G = \{\eta : V \to T \mid \mathsf{dom}(\eta) = \mathsf{var}(B) \wedge (B)\eta \subseteq G\}$$

Example 3. Let B be defined as $B = \{(?d\,;\!:\!service\,;?s),\ (?d\,;\!:\!treats\,;?p)\}$. B returns the doctors, their services and the patients they treat. The evaluation of B on the example graph G_0 of Fig. 1 is $[\![B]\!]_{G_0} = \{\eta\}$, where η is defined as $\eta : ?d \mapsto :bob$, $?s \mapsto :onc$ and $?p \mapsto :alice$.

Formally, the definition of BGP evaluation captures the semantics of SPARQL restricted to the conjunctive fragment of `SELECT` queries that do not use `FILTER`, `OPT` and `UNION` operators (see [14] for further details).

Another key concept of the Semantic Web is *named graphs* in which a set of RDF triples is identified using an IRI forming a *quad*. This allows to represent meta-information about RDF data such as provenance information and context. In order to handle named graphs, SPARQL defines the concept of *dataset*. A dataset is a set composed of a distinguished graph called the *default graph* and pairs comprising an IRI and an RDF graph constituting named graphs.

Definition 5. *(RDF dataset) An RDF dataset is a set:*

$$\mathscr{D} = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$$

where $G_i \in \mathsf{BGP}$, $u_i \in \mathsf{I}$, *and* $n \geq 0$. *In the dataset,* G_0 *is the default graph, and the pairs* $\langle u_i, G_i\rangle$ *are named graphs, with* u_i *the name of* G_i.

4 Access control semantics

AC4RDF semantics is defined using *authorization policies*. An authorization policy P is defined as a pair $P = (\mathscr{A}, \mathsf{ch})$ where $\mathscr{A}$ is a set of authorizations and $\mathsf{ch} : \mathscr{P}(\mathscr{A}) \to \mathscr{A}$ is a so called (abstract) *conflict resolution function* that picks out a unique authorization when several ones are applicable. The semantics of the access control model are given by means of the positive (authorized) subgraph G^+ obtained by evaluating P on a base RDF graph G.

4.1 Authorization semantics

Authorizations are defined using basic SPARQL constructions, namely basic graph patterns, in order to facilitate the administration of access control and to include homogeneously authorizations into concrete RDF stores without additional query mechanisms.

Definition 6. *(Authorization) Let* $\mathsf{Eff} = \{+, -\}$ *be the set of applicable* effects. *Formally, an* authorization $@a = (e, h, b)$ *is a element of* $\mathsf{Auth} = \mathsf{Eff} \times \mathsf{TP} \times \mathsf{BGP}$. *The component* e *is called* the effect *of the authorization* $@a$, h *and* b *are called its* head *and* body *respectively. The function* $\mathsf{effect} : \mathsf{Auth} \to \mathsf{Eff}$ *(resp.,* $\mathsf{head} : \mathsf{Auth} \to \mathsf{TP}$, $\mathsf{body} : \mathsf{Auth} \to \mathsf{BGP}$*) is used to denote the first (resp., second, third) projection function. The set* $\mathsf{hb}(@a) = \{\mathsf{head}(@a)\} \cup \mathsf{body}(@a)$ *is called the* underlying graph pattern *of the authorization* $@a$.

The concrete syntax "`GRANT`/`DENY` h `WHERE` b" is used to represent an authorization $@a = (e, h, b)$. The `GRANT` keyword is used when $e = +$ and the `DENY` keyword when $e = -$. Condition `WHERE` $\emptyset$ is elided when b is empty.

Example 4. Consider the set of authorizations shown in Table. 1. Authorization $@a_1$ grants access to triples with predicate : *hasTumor*. Authorization $@a_2$ states that all triples of type : *Cancerous* are denied. Authorizations $@a_3$ and $@a_4$ state that triples with predicate : *service* and : *treats* respectively are permitted. Authorization $@a_5$ states that triples about admission to the oncology service are specifically denied, whereas the authorization $@a_6$ states that such information are allowed in the general case. Finally, authorization $@a_9$ denies access to any triple, it is meant to be a default authorization.

Given an authorization $@a \in \mathsf{Auth}$ and a graph G, we say that $@a$ is *applicable to* a triple $t \in G$ if there exists a substitution θ such that the head of $@a$ is mapped to t and all the conditions expressed in the body of $@a$ are satisfied as well. In other words, we evaluate the underlying graph pattern $\mathsf{hb}(@a) = \{\mathsf{head}(@a)\} \cup \mathsf{body}(@a)$ against G and we apply all the answers of $[\![\mathsf{hb}(@a)]\!]_G$ to $\mathsf{head}(@a)$ in order to know which $t \in G$ the authorization $@a$ applies to. In the concrete system we implemented, this evaluation step is computed using the mechanisms used to evaluate SPARQL queries. In fact, given an authorization $@a$, the latter is translated to a SPARQL `CONSTRUCT` query which is evaluated over G. The result represents the triples over which $@a$ is applicable.

Table 1: Example of authorizations

$$
\begin{aligned}
&@_1 = \texttt{GRANT}(?p\ ;:hasTumor\ ;?t)\\
&@_2 = \texttt{DENY}\ \ (?p\ ;rdf:type\ ;:Cancerous)\\
&@_3 = \texttt{GRANT}(?d\ ;:service\ ;?s)\\
&@_4 = \texttt{GRANT}(?d\ ;:treats\ ;?p)\\
&@_5 = \texttt{DENY}\ \ (?p\ ;:admitted\ ;?s)\\
&\qquad \texttt{WHERE}\ \{(?s\ ;rdf:type\ ;:Oncology)\}\\
&@_6 = \texttt{GRANT}(?p\ ;:admitted\ ;?s)\\
&@_7 = \texttt{GRANT}(?p\ ;rdfs:domain\ ;?s)\\
&@_8 = \texttt{DENY}\ \ (?s\ ;?p\ ;:Cancerous)\\
&@_9 = \texttt{DENY}\ \ (?s\ ;?p\ ;?o)
\end{aligned}
$$

Definition 7. *(Applicable Authorizations) Given a finite set of authorizations* $\mathscr{A} \in \mathscr{P}(\mathsf{Auth})$ *and a graph* $G \in \mathsf{BGP}$*, the function* ar *assigns to each triple* $t \in G$*, the subset of* applicable authorizations *from* $\mathscr{A}$ *:*

$$\mathsf{ar}(G, \mathscr{A})(t) = \{@ \in \mathscr{A} \mid \exists \theta \in [\![\mathsf{hb}(@)]\!]_G, t = (\mathsf{head}(@))\theta\}$$

Example 5. Consider the graph G_0 shown in Fig. 1 and the set of authorizations $\mathscr{A}$ shown in Table 1. The applicable authorizations on triple t_8 are computed to $\mathsf{ar}(G_0, \mathscr{A})(t_8) = \{@_5, @_6, @_9\}$.

The set of triples in a given graph G to which an authorization $@$ is applicable, is called the *scope* of $@$ in G.

Definition 8. *(Authorization scope) Given a graph* $G \in \mathsf{BGP}$ *and an authorization* $@ \in \mathsf{Auth}$*, the scope of* $@$ *in* G *is defined by the following function* $\mathsf{scope} \in \mathsf{BGP} \times \mathsf{Auth} \to \mathsf{BGP}$*:*

$$\mathsf{scope}(G)(@) = \{t \in G \mid \exists \theta \in [\![\mathsf{hb}(@)]\!]_G, t = (\mathsf{head}(@))\theta\}$$

Example 6. Consider authorization $@_8$ in Table 1, and the graph G_0 in Fig. 1. The scope of $@_8$ is computed as follows : $\mathsf{scope}(G_0)(@_8) = \{t_1, t_7\}$.

4.2 Policy and conflict resolution function

In the context of access control with both positive (grant) and negative (deny) authorizations, policies must deal with two issues: *inconsistency* and *incompleteness*. Inconsistency occurs when multiple authorizations with different effects are applicable to the same triple. Incompleteness occurs when triples have no applicable authorizations. Inconsistency is resolved using a *conflict resolution strategy* which selects one authorizations when more than one applies. Incompleteness is resolved using a *default strategy* which is an effect that is applied to the triples with no applicable authorizations. In [3], the authors abstracted from the details of the concrete resolution strategies by assuming that there exists a choice function that, given a finite set of possibly conflicting authorizations, picks a unique one out.

Definition 9. *(Authorization Policy) An (authorization) policy P is a pair $P = (\mathscr{A}, \mathsf{ch})$, where $\mathscr{A}$ is a finite set of authorizations and $\mathsf{ch} : \mathscr{P}(\mathscr{A}) \to \mathscr{A}$ is a conflict resolution function.*

Example 7. An example policy is $P = (\mathscr{A}, \mathsf{ch})$ where $\mathscr{A}$ is the set of authorizations in Table 1 and ch is defined as follows. For all non-empty subset $\mathscr{B}$ of $\mathscr{A}$, $\mathsf{ch}(\mathscr{B})$ is the first authorization (using syntactical order of Table 1) of $\mathscr{A}$ that appears in $\mathscr{B}$. For $\mathscr{B} = \emptyset$, $\mathsf{ch}(\emptyset) = a_9$.

The semantics of policies are given by composing the functions ar, ch and then effect in order to compute the authorized subgraph of a given graph.

Definition 10. *(Policy Evaluation, Positive Subgraph) Given a policy $P = (\mathscr{A}, \mathsf{ch}) \in \mathsf{Pol}$ and a graph $G \in \mathsf{BGP}$, the set of authorized triples that constitutes the* positive subgraph *of G according to P is defined as follows, writing G^+ when P is clear from the context:*

$$G_P^+ = \{t \in G \mid (\mathsf{effect} \circ \mathsf{ch})(\mathsf{ar}(G, \mathscr{A})(t)) = +\}$$

Example 8. Let us consider the policy $P = (\mathscr{A}, \mathsf{ch})$ defined in Example 7 and the graph G_0 of Fig. 1. Regarding the triple $t_8 = (:alice\ ;::admitted\ ;::onc)$, $\mathsf{ar}(G_0, \mathscr{A})(t_8) = \{a_5, a_6, a_9\}$. Since a_5 is the first among authorization in Table 1 and its effect is $-$, we deduce that $t_8 \notin {G_0}_P^+$. By applying a similar reasoning on all triples in G_0, we obtain ${G_0}_P^+ = \{t_1, t_4, t_5, t_6\}$.

5 Policy enforcement

To enforce `AC4RDF` model, we use an annotation approach which materializes the applicable authorizations in an annotated graph denoted by $G^{\mathscr{A}}$. The latter is computed once and for all at design time. The subjects' queries are evaluated over the annotated graph with respect to their assigned authorizations. In the following, we show how the base graph triples are annotated and how the subjects queries are evaluated.

5.1 Graph annotation

From a conceptual point of view, an annotated triple can be represented by adding a fourth component to a triple hence obtaining a so called *quad*. From a physical point of view, the annotation can be stored in the graph name of the SPARQL dataset (Definition 5). To annotate the base graph, we use the graph name IRI of the dataset to store a bitset representing the applicable authorizations of each triple. First we need a bijective function $\mathsf{authToBs}$ which maps a set of authorizations to an IRI representing its bitset. Authorizations are simply mapped to their position in the syntactical order of authorization definitions. In other words, given an authorization a_i and a set authorizations $\mathscr{A}_S$ to be mapped, the i-th bit is set to 1 in the generated bitset if $a_i \in \mathscr{A}_S$. $\mathsf{authToBs}^{-1}$ is the inverse function of $\mathsf{authToBs}$.

Next we define a function arsg which takes a set of authorizations $\mathscr{A}' \subseteq \mathscr{A}$ and a graph G as parameters, and returns the subgraph of G containing triples which have $\mathscr{A}'$ as applicable authorizations. The function arsg is formally defined as follows:

$$\mathsf{arsg}(\mathscr{A}', G) = \{t \in G \mid \mathsf{ar}(G, \mathscr{A})(t) = \mathscr{A}'\}$$

Example 9. Consider the policy $P = (\mathscr{A}, \mathsf{ch})$ defined in Example 7 and the graph G_0 of Fig. 1. $\mathsf{authToBs}(\{a_1, a_9\}) = 100000001$, $\mathsf{arsg}(\{a_1, a_9\}, G_0) = \{t_4\}$.

Now we are ready to define the dataset representing the annotated graph.

Definition 11. *(Annotated graph) Given a set of authorizations $\mathscr{A}$ and a graph G, the dataset that represents the annotated graph denoted by $G^{\mathscr{A}}$, is defined by:*

$$G^{\mathscr{A}} = \big\{\langle \mathsf{authToBs}(\mathscr{A}'), \mathsf{arsg}(\mathscr{A}', G)\rangle \mid \mathscr{A}' \in \mathscr{P}(\mathscr{A}) \wedge \mathsf{arsg}(\mathscr{A}', G) \neq \emptyset\big\}$$

Definition 11 defines how to annotate the base graph G given a set of authorization. The following Lemma 1 ensures that $G^{\mathscr{A}}$ forms a partition of the base graph G.

Lemma 1. *Given an annotated graph $G^{\mathscr{A}} = \{\langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$, the following properties hold:*

- $\forall i, j \in 1..n : i \neq j \implies G_i \cap G_j = \emptyset$
- $\bigcup_{i \in 1..n} G_i = G$

5.2 Subject's query evaluation

The subject is the entity requesting access to the triple store. The determination of the objects accessible by the subject could be based on the subject identity, role or attributes. Given a global set of authorizations $\mathscr{A}$ we suppose that the subset $\mathscr{A}_s$ assigned to the subject is known in advance. The upstream authentication and determination of the authorizations assigned to the subjects is out of the scope of this paper.

Following Definition 10, given a global policy authorization set $\mathscr{A}$, the positive subgraph of a subject having $\mathscr{A}_s \subseteq \mathscr{A}$ as applicable authorizations, is given by the following : $G_s^+ = \{t \in G \mid (\mathsf{effect} \circ \mathsf{ch})(\mathsf{ar}(G, \mathscr{A}_s)(t)) = +\}$. Since we materialized the set of applicable authorizations in $G^{\mathscr{A}}$, we need to define the subject's positive subgraph from the graph annotation, more precisely from $\mathsf{ar}(G, \mathscr{A})$. The following lemma shows that $\mathsf{ar}(G, \mathscr{A}_s)$ can be computed from $\mathscr{A}_s$ and $\mathsf{ar}(G, \mathscr{A})$.

Lemma 2. *Given a graph G, a set of policy authorizations $\mathscr{A}$ and a subset of subject's authorizations $\mathscr{A}_s$, the following holds for any $t \in G$:*

$$\mathscr{A}_s \cap \mathsf{ar}(G, \mathscr{A})(t) = \mathsf{ar}(G, \mathscr{A}_s)(t)$$

Similarly to the triples, subjects are assigned to bitsets representing authorizations applicable to them. If a subject authorization set is $\mathscr{A}_s$, then she/he is assigned a bitset *ubs* where the i-th bit is set to 1 if $a_i \in \mathscr{A}_s$.

Table 2: Example of annotated graph and users bitsets

$G_0^{\mathscr{A}}$		ubs & u	
u	G	Eve	Dave
		100001001	001100001
000000111	$\{t_1\}$	000000001	000000001
000000001	$\{t_2, t_3, t_9\}$	000000001	000000001
100000001	$\{t_4\}$	100000001	000000001
001000001	$\{t_5\}$	000000001	001000001
000100001	$\{t_6\}$	000000001	000100001
010000011	$\{t_7\}$	000000001	000000001
000011001	$\{t_8\}$	000001001	000000001

Example 10. Given the set of authorizations $\mathscr{A}$ in Table 1. Eve is a nurse who can see information about patients having tumors (a_1) and which service they are admitted to (a_6). She is denied anything else (a_9). Her assigned bitset is the bitset 100001001 of Table 2. Dave belongs to the administrative staff, he can access doctors services assignment (a_3) and the patients they treat (a_4). He is denied anything else (a_9). His assigned bitset is the bitset 001100001 of Table 2.

Once the graph is annotated, it is made available to the subjects with a filter function which prunes out the inaccessible triples given the subjects's authorization set. In other words, the filter function returns the subjects's positive subgraph by applying the ch function on the subject's assigned authorizations $\mathsf{ar}(G, \mathscr{A}_s)(t)$. We showed in Lemma 2 that this subset can be obtained from the applicable authorizations in $G^{\mathscr{A}}$ by computing a bitwise logical *and* (denoted by &) between the subject's and triples' bitsets.

Definition 12. *(Filter function) Given a subject's bitset ubs and an annotated graph* $G^{\mathscr{A}}$, filter *is defined as follows:*

$$\mathsf{filter}(G^{\mathscr{A}})(ubs) = \bigcup \{G_i \mid \langle u_i, G_i \rangle \in G^{\mathscr{A}} \wedge (\mathsf{effect} \circ \mathsf{ch})(\mathsf{authToBs}^{-1}(u_i \ \& \ ubs)) = +\}$$

Once the subject's positive subgraph computed with filter, the subject's query Q is then evaluated over it returning $[\![Q]\!]_{\mathsf{filter}(G^{\mathscr{A}})(ubs)}$ to the subject.

Example 11. Let us consider the policy $P = (\mathscr{A}, \mathsf{ch})$ of Example 7. Table 2 illustrates the annotated graph obtained from G_0 shown in Fig. 1, as well as the two users of Example 10 with their assigned authorizations. The filter function will compute the positive subgraph of Eve as follows: $\mathsf{filter}(G_0^{\mathscr{A}})(100001001) = \{t_4, t_8\}$. Similarly, Dave's positive subgraph equals $\{t_5, t_6\}$.

6 Implementation

Our system is implemented using the Jena Java API on top of the Jena TDB[3] (quad) store. Apache Jena is an open source Java framework which provides an API to manage RDF data. ARQ[4] is a SPARQL query engine for Jena which allows querying and updating RDF models through the SPARQL standards. ARQ supports custom aggregation and `GROUP BY`, `FILTER` functions and path queries. Jena TDB is a native RDF store which allows to store and query RDF quads.

To generate $G^{\mathscr{A}}$, the dataset of annotated triples, we use SPARQL `CONSTRUCT` queries to obtain authorizations scopes (see Definition 8). An authorization $\mathscr{a}$ is transformed into $Q_{\mathscr{a}} =$ `CONSTRUCT` $\mathsf{head}(\mathscr{a})$ `WHERE` $\mathsf{hb}(\mathscr{a})$. We use an in-memory hash map in which we store the *ids* of the triples and the correspondent bitset. For every authorization $\mathscr{a}_i$, a `CONSTRUCT` query $Q_{\mathscr{a}_i}$ is run over the raw dataset, and the result triples are added/updated to the hash map with the bit i set to 1. Once the hash map is computed, it is written into a dataset which represents $G^{\mathscr{A}}$. Note that we could have used the dataset directly instead of a hash map, but this would be time consuming due to the high number of disk accesses. In case of a high number of triples that can't hold in memory, we could use a hybrid approach by loading the triples partially, but this extension is left for future work.

During query evaluation, on the fly filtering is applied to the accessed triples. Jena TDB provides a low level quad filter hook[5] that we use for implementation. For each accessed quad, let u be the quad's graph IRI, t its triple and *ubs* be the subject's bitset. A bitwise logical *and* is performed between (the bitset represented by) u and *ubs*. The ch function on the authorizations obtained by $\mathsf{authToBs}^{-1}$ is then applied in order to allow or deny access to t. If t is allowed, then it is transmitted to the ARQ engine to be used by query Q. Otherwise, it will be hidden to the ARQ engine. An in-memory cache is used to map quad graph IRIs to grant/deny decisions in order to speedup the filtering process.

6.1 Experiments

The key input factors for the benchmarking of our solution are the sizes of the *base graphs*, the sizes of the *access control policies*, the sizes of positive subgraphs, the sizes of subjects' policies and the subjects' queries. The factors are reported in Table 3. The *base graphs* are synthetic graphs generated by the Lehigh University Benchmark (LUBM)[6]. Their sizes ($|G|$) vary from 126k to 1,591k triples. The *access control policies* are randomly generated using the LUBM vocabulary (about universities and people therein), with three control parameters. The first control parameter is the number of authorizations ($|\mathscr{A}|$)

[3] `https://jena.apache.org/documentation/tdb/`
[4] `https://jena.apache.org/documentation/query/`
[5] `http://jena.apache.org/documentation/tdb/quadfilter.html`
[6] `http://swat.cse.lehigh.edu/projects/lubm/`

Table 3: Summary of notations

in	$\|G\|$	Size of the LUBM dataset
in	$\|\mathscr{A}\|$	Number of authorizations
in	$\|G^+\|_{\|G\|}$	Positive subgraph size w.r.t raw dataset size
in	$\|\mathscr{A}_s\|$	Number of authorizations assigned to the subject
in	Q_s	LUBM test Query
out	t_A	Time to build $G^{\mathscr{A}}$ in memory
out	t_W	Time to write $G^{\mathscr{A}}$ to disk
out	t_{G^+}	Time to evaluate Q on materialized G^+
out	$t_{G^{\mathscr{A}}}$	Time to evaluate Q on $G^{\mathscr{A}}$
out	t_G	Time to evaluate Q on (raw) G

and varies from 50 to 200 authorizations. The second control parameter is the *scope average* of the policy with respect to the G. In other words, the percentage of triples in G which are under the influence of the policy authorizations. The last control parameter is the size of the body of each (atomic) authorization $\mathscr{a} \in \mathscr{A}$. For the sake of brevity, the results we report here are for fixed scope (about 4% by authorization) and fixed sizes of bodies (set to 2 for each authorization). The size of positive subgraph parameter $|G^+|_{|G|}$ varies from 10 to 100% of $|G|$ and the number of subject's authorizations $|\mathscr{A}_s|$ from 50 to 200. Regarding the subject query parameter Q_s, we used a subset of LUBM test queries. We analyzed both the static (creation time) and the dynamic (evaluation time) performance of our solution. Each experiment is run 6 times on 2 cores and 4 GB RAM virtual machines running on OpenStack.

Static performance We distinguish the time needed to compute $G^{\mathscr{A}}$ between the time required for its *building* and the time required for its *writing*. The time to *build* the authorization bitset $\mathsf{ar}(G, t)$ associated with each triple $t \in G$ in memory is referred to as t_A in Table 3. The time to *write* the annotated graph

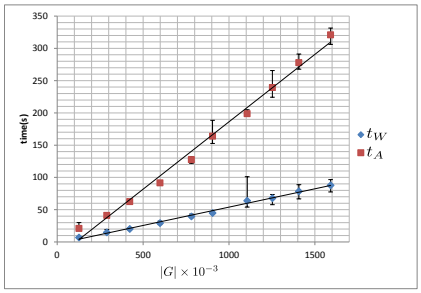


Fig. 2: Annotation and writing times

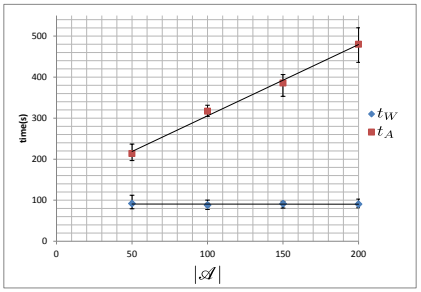


Fig. 3: Annotation and writing times

$G^{\mathscr{A}}$ from the memory to the quad store is referred to as t_W in Table 3. Fig. 2 shows t_A and t_W with $|\mathscr{A}|$ being set to 100 authorizations. Fig. 3 shows t_A and t_W with $|G|$ being set to 1,591k triples. As each $a \in \mathscr{A}$ is mapped to a SPARQL `CONSTRUCT` query, the results show that t_A grows linearly when $|G|$ or $|\mathscr{A}|$ gets bigger. The annotation time is not negligible but we argue that it is not an issue: $G^{\mathscr{A}}$ is computed once, as long as $\mathscr{A}$ is not modified. The ratio t_A/t_W is about 3.4 on average for fixed value of G in Fig. 2. In other words, for 100 authorizations, our method is is amortized if the sum of triples in the materialized named graphs is approximatively 5 times greater then the number of triples in base graph. Fig. 3 shows that t_A grows linearly when $|\mathscr{A}|$ grows. However, as expected the results show that t_W is independent of $|\mathscr{A}|$: the overhead incurred by the growing size of the bitsets is negligible for $|\mathscr{A}| \in \{50, 100, 150, 200\}$. On average, the annotated graph $G^{\mathscr{A}}$ requires 50% more space than G.

Dynamic performance To evaluate the performance of our solution at runtime, we compare our approach to two extreme methods. Each method computes the positive subgraph G^+ obtained by filtering the result of query Q on a base graph G according to a set $\mathscr{A}$ of authorizations.

The first extreme (naive) method gives an upper bound on the overhead incurred by the filtering process. Indeed, in the post-processing approaches, the access control consists in two steps : (1) compute the full answer $Q(G)$ and (2) filter out the denied triples from $Q(G)$ as a post-processing step. This method avoids duplication of the base graph G at the price of high overhead at runtime. In our experiments, we considered the step (1) only, by computing the full answer $Q(G)$. We refer to this method as t_G in Table 3. The second extreme method gives a lower bound on the overhead incurred by the filtering process. The idea is to materialize G^+ for each user profile and then compute $Q(G^+)$. We refer to this method as t_{G^+} in Table 3. This method avoids the filtering post-process at the price of massive duplication and storage overhead. In contrast, our approach, namely $t_{G^{\mathscr{A}}}$ in Table 3, is a trade-off between the extreme ones: it needs some static computation while offering competitive runtime performance. Our results are shown in Fig. 4 for varying sizes of $|G|$ with $|\mathscr{A}|$ and $|\mathscr{A}_s|$ set to 100, and

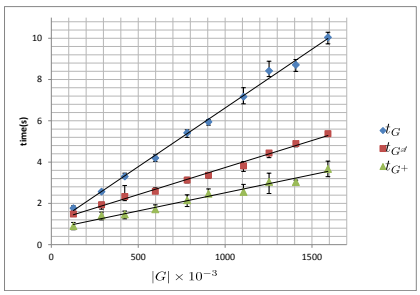


Fig. 4: Query evaluation time

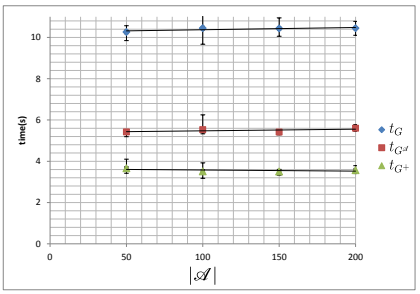


Fig. 5: Query evaluation time

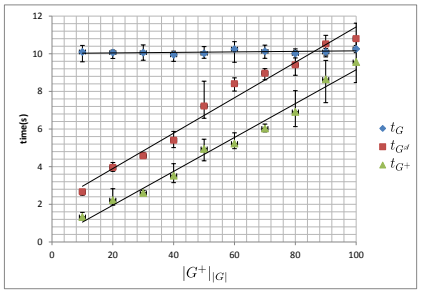


Fig. 6: Query evaluation time

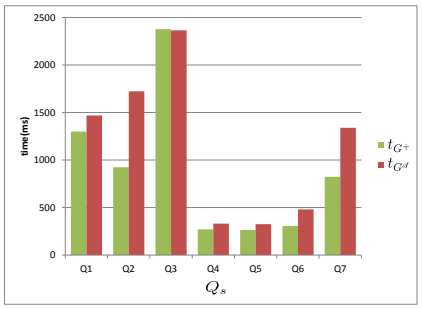


Fig. 7: Query evaluation time

$|G^+|_{|G|}$ set to 40%. The subject query Q_s is set to the worst case which is the select all query. The key insight from these experiments is that the overhead *is independent from* $|G|$ and is about 50% as confirmed by the R statistical software.

Another advantage of our approach is its independence from the number of authorizations of both the policy and those assigned to the subject. In Fig. 5 we vary the number of policy authorizations ($|\mathscr{A}|$) with $|G|$ set to 1,591k triples and Q_s to the select all query. The experiments show a constant overhead while changing $|\mathscr{A}|$.

Regarding $|G^+|_{|G|}$, the size of the positive subgraph with respect to the size of the annotated graph, the experiments in Fig. 6 show that the query answer time $t_{G^{\mathscr{A}}}$ grows linearly when $|G^+|_{|G|}$ grows, with $|G|$ fixed to 1,591k and $|\mathscr{A}|$ and $|\mathscr{A}_s|$ fixed to 100. Q_s being the select all query. This shows that the overhead w.r.t. a materialized $Q(G^+)$ does not depend on the size of the positive subgraph. Note that t_G does not vary since we did not consider the filtering step of post-processing approaches, otherwise it would grow linearly when $|G^+|_{|G|}$ grows.

In Fig. 7 we run experiments on our system with a subset of LUBM test queries used by [4] with $|\mathscr{A}|$ and $|\mathscr{A}_s|$ set to 100, and $|G^+|_{|G|}$ set to 40%.. Q1 and Q3 are more complex queries having a high number of initial triples associated with the triple patterns, but the final number of results is quite small (28 and 0 respectively). Fig. 7 shows that the time to evaluate query Q3 in presence of the filter $t_{G^{\mathscr{A}}}$ is smaller than the evaluation time over materialized positive subgraph t_{G^+}. The reasons could be the empty result of Q3 or different execution plans. In the rest of the queries, the overhead was between 6 and 40%.

7 Conclusion

In this paper, we proposed an enforcement framework to the access control model for RDF defined in [3]. We used an annotation approach where the base graph is annotated at the policy design time. Each triple is annotated with a bitset representing its applicable authorizations. The subjects' queries are evaluated over their positive subgraph constructed using the the her/his bitset and the

triples' bitset. The experiments showed that the annotation time is not negligible, but we argue that it is not an issue since this operation is done once and for all during policy design time. We showed that the overhead of the subject query evaluation is independent from size of the base graph, and it is about 50%. Moreover, we showed that our approach is independent from the number of policy authorizations as well as the used query language in contrast to the query rewriting techniques.

Ongoing work on this platform includes the design of an algorithm for the incremental update of $G^{\mathscr{A}}$ when G is modified, high-level optimizations for the construction of $G^{\mathscr{A}}$ using the partial order between authorizations induced by basic graph pattern containment and new empirical evaluations on both synthetic and real-life data.

References

1. Abel, F., De Coi, J.L., Henze, N., Koesling, A.W., Krause, D., Olmedilla, D.: Enabling advanced and context-dependent access control in RDF stores. In: The Semantic Web, pp. 1–14. Springer (2007)
2. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1) (2008)
3. Anonymous 2015: Details omitted for double-blind reviewing.
4. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: WWW. pp. 41–50 (2010)
5. Berners-Lee, T.: Linked data-design issues (2006), `https://www.w3.org/DesignIssues/LinkedData.html`
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)
7. Costabello, L., Villata, S., Delaforge, N., Gandon, F., et al.: Linked data access goes mobile: Context-aware authorization for graph stores. In: LDOW-5th WWW Workshop (2012)
8. Flouris, G., Fundulaki, I., Michou, M., Antoniou, G.: Controlling access to rdf graphs. In: Future Internet-FIS 2010, pp. 107–117. Springer (2010)
9. Hayes, P.J., Patel-Schneider, P.F.: Rdf 1.1 semantics. W3C Recommendation (2014), `http://www.w3.org/TR/rdf11-mt/`
10. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. J. Web Sem. 3(2-3), 79–115 (2005)
11. Jain, A., Farkas, C.: Secure resource description framework: an access control model. In: SACMAT. pp. 121–129. ACM (2006)
12. Lopes, N., Kirrane, S., Zimmermann, A., Polleres, A., Mileo, A.: A logic programming approach for access control over RDF. In: ICLP. pp. 381–392 (2012)
13. Papakonstantinou, V., Michou, M., Fundulaki, I., Flouris, G., Antoniou, G.: Access control for RDF graphs using abstract models. In: SACMAT. pp. 103–112 (2012)
14. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34(3), 16:1–16:45 (Sep 2009)
15. Polleres, A.: From SPARQL to rules (and back). In: WWW. pp. 787–796 (2007)
16. Reddivari, P., Finin, T., Joshi, A.: Policy-based access control for an RDF store. In: WWW. pp. 78–81 (2005)

A Appendix

This appendix presents the proofs of Lemma 1 and 2 as well as the LUBM queries used in experiments.

A.1 Proofs

Proof (Proof of Lemma 1). In the following, we prove that $\forall i, j \in 1..n : i \neq j \implies G_i \cap G_j = \emptyset$.

We prove that $\forall i, j \in 1..n : G_i \cap G_j \neq \emptyset \implies i = j$. $G_i \cap G_j \neq \emptyset$ means that there exists t such that $t \in G_i$ and $t \in G_j$. Since $t \in G_i$ then $u_i = \mathsf{authToBs}(\mathsf{ar}(G, \mathscr{A})(t))$. Similarly, since $t \in G_j$ then $u_j = \mathsf{authToBs}(\mathsf{ar}(G, \mathscr{A})(t))$. Which means that $u_i = u_j$, hence $i = j$.

In the following we prove that $\bigcup_{i \in 1..n} G_i = G$

First we prove that $\forall t \in \bigcup_{i \in 1..n} G_i \implies t \in G$.

$\forall t \in \bigcup_{i \in 1..n} G_i$ means that $\exists \langle u, G' \rangle \in G^{\mathscr{A}} \mid t \in G'$. Since by definition, $G' \subseteq G$ then $t \in G$.

We prove that $\forall t \in G \implies t \in \bigcup_{i \in 1..n} G_i$.

Since $@_u \in \mathscr{A}$ then $\forall t \in G$, $\mathsf{ar}(G, \mathscr{A})(t) \neq \emptyset$, hence $\exists \langle u', G' \rangle \in G^{\mathscr{A}}$ s.t. $t \in G'$. Which means that $t \in \bigcup_{i \in 1..n} G_i$.

Proof (Proof of Lemma 2). By Definition 7, $\mathsf{ar}(G, \mathscr{A})(t) = \{@ \in \mathscr{A} \mid \exists \theta \in [\![\mathsf{hb}(@)]\!]_G.t = (\mathsf{head}(@))\theta\}$, hence $\mathscr{A}_s \cap \mathsf{ar}(G, \mathscr{A})(t) = \{@ \in \mathscr{A}_s \cap \mathscr{A} \mid \exists \theta \in [\![\mathsf{hb}(@)]\!]_G.t = (\mathsf{head}(@))\theta\}$. Since $\mathscr{A}_s \subseteq \mathscr{A}$ then $\mathscr{A}_s \cap \mathscr{A} = \mathscr{A}_s$. Which means that $\mathscr{A}_s \cap \mathsf{ar}(G, \mathscr{A})(t) = \{@ \in \mathscr{A}_s \mid \exists \theta \in [\![\mathsf{hb}(@)]\!]_G.t = (\mathsf{head}(@))\theta\} = \mathsf{ar}(G, \mathscr{A}_s)(t)$.

A.2 LUBM Queries

Q1: `SELECT` *?x ?y ?z* `WHERE`
{ *?z ub:subOrganizationOf ?y. ?y rdf:type ub:University.*
?z rdf:type ub:Department. ?x ub:memberOf ?z.
?x rdf:type ub:GraduateStudent. ?x ub:undergraduateDegreeFrom ?y. }

Q2: `SELECT` *?x* `WHERE`
{ *?x rdf:type ub:Course. ?x ub:name ?y.* }

Q3: `SELECT` *?x ?y ?z* `WHERE`
{ *?x rdf:type ub:UndergraduateStudent. ?y rdf:type ub:University.*
?z rdf:type ub:Department. ?x ub:memberOf ?z.
?z ub:subOrganizationOf ?y. ?x ub:undergraduateDegreeFrom ?y. }

Q4: `SELECT` *?x* `WHERE`
{ *?x ub:worksFor* `<http://www.Department0.University0.edu>`.
?x rdf:type ub:FullProfessor. ?x ub:name ?y1.

?*x* *ub*:*emailAddress* ?*y2*. ?*x* *ub*:*telephone* ?*y3*. }

Q5: SELECT ?*x* WHERE
{ ?*x* *ub*:*subOrganizationOf* <http://www.Department0.University0.edu>.
?*x* *rdf*:*type* *ub*:*ResearchGroup* }

Q6: SELECT ?*x* ?*y* WHERE
{ ?*y* *ub*:*subOrganizationOf* <http://www.University0.edu>.
?*y* *rdf*:*type* *ub*:*Department*. ?*x* *ub*:*worksFor* ?*y*.
?*x* *rdf*:*type* *ub*:*FullProfessor*. }

Q7: SELECT ?*x* ?*y* ?*z* WHERE
{ ?*y* *ub*:*teacherOf* ?*z*. ?*y* *rdf*:*type* *ub*:*FullProfessor*.
?*z* *rdf*:*type* *ub*:*Course*. ?*x* *ub*:*advisor* ?*y*.
?*x* *rdf*:*type* *ub*:*UndergraduateStudent*. ?*x* *ub*:*takesCourse* ?*z* }