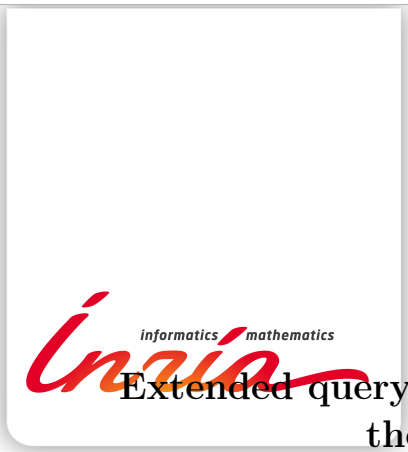
Extended query language and composition for the XR data model

Alin Deutsch, François Goasdoué, Julien Leblay, Ioana Manolescu

RESEARCH REPORT
N° 8391
November 2013
Project-Teams OAKSAD

ISSN 0249-6399 ISRN INRIA/RR--8391--FR+ENG



Extended query language and composition for the XR data model

Alin Deutsch, François Goasdoué, Julien Leblay, Ioana Manolescu

Project-Teams OAKSAD

Research Report n° 8391 — November 2013 — 27 pages

Abstract:

Key-words:

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Langage de requête étendu et composition pour le modèle de données XR

Résumé :

Mots-clés :

1 Introduction

XR is a data model from representing RDF-annotated and XML data. The model treats XML nodes as RDF resources, by assigning URIs to them. XRQ is a query language allowing to query XR data both on the structure of an XML instance and the semantics of its RDF annotations. It combines tree patterns and triple patterns, and in its original form, the language returns tuples of bindings. The model and query language are fully described in [10], along with an implementation and in-depth study of query evaluation over existing XML and RDF data management systems.

In this work, we extend the query language such that queries yield XR data. This extension allows one to see XR queries as views, i.e., intensionally defined XR instances. The problem of evaluating a query over a view rather than an extensional instance naturally emerges from this settings. More precisely, we want to know, given a query q and a view v, if q can be evaluated over the v and how.

1.1 Motivations

Views are an essential tool in the data management literature and they are used in a wide range of applications. They can be used to hide some of the complexity of the data, restrict visibility to the data depending on one's privileges. Views are widely used in data exchange and data integration. e.g., to describe the relationship between local and global schemata. Once materialized, views can be seen as pre-computed sub-queries and serve as powerful instruments for query optimization.

Many problems pertaining to view-based data management, such as answering queries using views and view selection, are still actively studied today. In this section, we study the problem of query-view composition in XR, i.e., evaluating a query over the result of a view.

We present three real-world scenarios, where the query-view composition problem can arise.

Restricted access. A news agency uses a centralized XR instance featuring a large corpus of annotated news articles. These articles are made available to the public and automatically integrated to the on-line issues. Some of the annotations, such as sources of sensitive issues, are however hidden to the public and only accessible to the employees. The public-facing annotated articles could, in this case, be expressed as XR views, by careful pruning out source-related contents.

Data exchange. A Business Process Outsourcing (BPO) company manages payroll information for several clients using XR data. Payslips are typically represented as XML data, while HR-related information is represented as RDF annotations. For historical or regulatory reasons, each client relies on slightly different schema, e.g., weekly vs. monthly income or country-specific social security information fields. All incoming data must be converted to a common schema accommodating the specificities of each client. These transformations can be done with XR views.

Data integration. There exists a large body of XML and RDF data on the Web that cover related topics or contain complementary information. For instance, every year, the OECD publishes very fine-grained economic and social data on a wide range of countries in XML. This data does not, however, incorporate the historical background of these countries or information about the current political situation. A RDF dataset such as DBpedia does, however, contain that type of information. But since these data sets are stored in pure XML and RDF models, querying both of them concurrently requires insight about the data on either side. For example, to join the data on the country, one would need to know which OECD and DBpedia fields to use. An elegant way to simplify that type of queries would be to integrate the two datasets into an XR-ready form, where each OECD yearly record is annotated with country-specific data coming from DBpedia. We give a concrete example of that kind of data integration at the end of this section.

1.2 Contributions

In this report, we present an extension of the query language introduced in [10], where an XRQ query produces XR data. We provide an algorithm for composing an XR query over a view and prove its correctness and completeness. A direct consequence of this results is that our extended query language is closed under composition.

Section 2 presents a brief overview of the data model and core query language. Section 3 introduces the extension to the XRQ query language. In Section 4, we discusses the notion of containment in this language. Section 5 provides an algorithm for query-view composition in XRQ. Finally, Section 6 discusses related works.

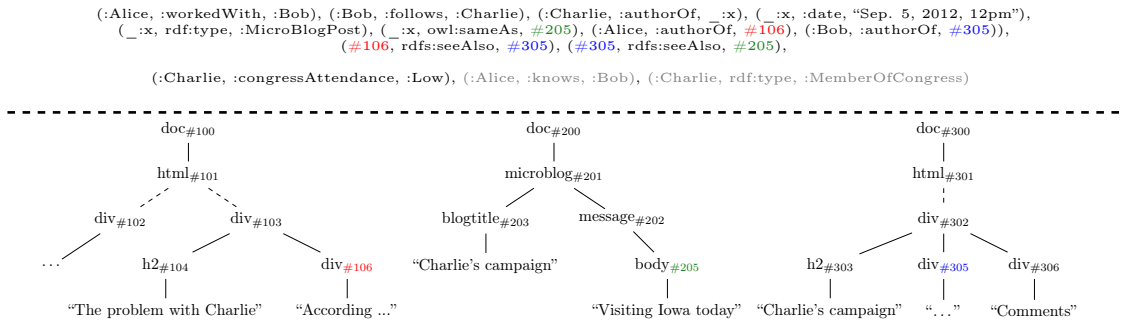


Figure 1: Sample XR instance

2 Preliminaries

The following definitions are given as a quick references for the XR data model and its core query language. The interested is invited to refer to [10] for further details and examples.

Definition 2.1 (The data model)**.**

- $\mathcal{U}$ *is the set of URIs as defined in [25], and a subset* $\mathcal{I} \subseteq \mathcal{U}$ *of* document URIs *acting as document identifiers.*
- $\mathcal{L}$ *is the set of literals [24].*
- $\mathcal{N}$ *is the set of possible XML element and attribute names, to which we add the empty name* ϵ*.*
- $\mathcal{B}$ *is the set of blank nodes.*
- *An* ***XML tree*** *is a finite, unranked, ordered, labeled tree* $T = (N, E)$ *with nodes* N *and edges* E*. We define a set of functions on XML nodes to model certain properties that may carry.*

  1) $\tau : N \to \{document, attribute, element\}$ *assigns a label to every node,*

  2) $\lambda : N \to \mathcal{N}$ *is a partial function assigning a label to every node of type element or attribute,*

  3) $\gamma : N \to \mathcal{U}$ *is a partial functions assigning a URI to every node of type document, element or attribute,*
- *An* ***XML instance*** I_X *is a finite set of XML trees.*
- *An* ***RDF instance*** I_R *is a set of triples of the form* (s, p, o)*, where* $s \in (\mathcal{U} \cup \mathcal{B})$*,* $p \in \mathcal{U}$*, and* $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$*.*
- *An* ***XR instance*** *is a pair* (I_X, I_R)*, where* I_X *and* I_R *are an XML and an RDF instance, respectively, built upon the same set of URIs.*

Example Figure 1 represents an XR data instance, with the RDF sub-instance at the top and the XML sub-instance at the bottom. Node URIs are indicated as subscripts of XML nodes (prefixed with "#").

Definition 2.2 (XRQ core syntax)**.**

- *A* ***tree pattern*** *is a finite, unordered, unranked,* $\mathcal{N}$*-labelled tree with two types of edges, namely child and descendant edges. We may attach to each node at most one* uri *variable, one* val *variable and one* cont *variable. We may also attach to a node one equality predicate of the form* $t : v = c$*, where* t *is a variable type,* v *is a variable and* c *is a constant, denoting a selection on the variable* v*, i.e., it must be bound to* c*. The variable name may be omitted if it is not used anywhere else in the query, leading to a label of the form* $t : c$*.*
- *A* ***triple pattern*** *is a triple* (s, p, o)*, where* s, p *are URIs or variables, whereas* o *is a URI, a literal, or a variable.*
- *An* ***XRQ query*** *consists of a* head *and a* body*. The body is a set* Q_X *of tree and a set* Q_R *triple patterns built over the same set of variables, whereas the head* h *is an ordered list of variables appearing also in the body. We denote such a query by* $Q = (h, Q_X, Q_R)$*.*

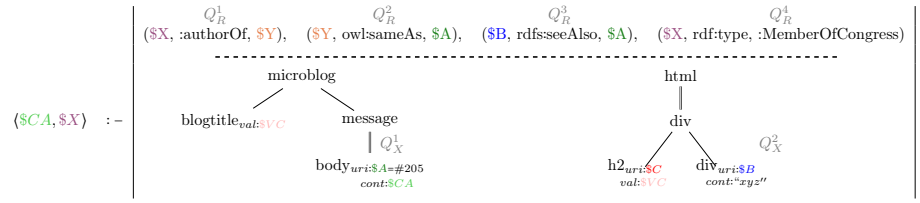


Figure 2: Sample core XRQ query

Example Figure 2 represents an XR query, with its tuple of variables in the head at the left hand, the triple patterns at the top-right hand and the tree patterns at the bottom-right hand.

Definition 2.3 (XRQ core semantics)**.**

- *A **match of** Q **against** I_X is a mapping ϕ from the nodes of Q to the nodes of I_X that preserves (i) node labels, i.e., for every node $n \in Q$, $\phi(n) \in I_X$ has the same label as n, (ii) structural relationships, that is: if n_1 is a /-child of n_2 in Q, then $\phi(n_1)$ is a child of $\phi(n_2)$, while if n_1 is a //-child of n_2, then $\phi(n_1)$ must be a descendant of $\phi(n_2)$, and (ii) structural relationships, that is: if n_1 is a /-child of n_2 in Q, then $\phi(n_1)$ is a child of $\phi(n_2)$, while if n_1 is a //-child of n_2, then $\phi(n_1)$ must be a descendant of $\phi(n_2)$. Moreover, ϕ satisfies equality constraints, i.e., if n has a label of the form $t : v = c$ or $t : c$, where t is a type, v is a variable and c is a constant, then (i) if t is* uri*, the URI of $\phi(n)$ is c, (ii) if t is* val*, the text value of $\phi(n)$ is c, (iii) if t is* cont*, the serialization of $\phi(n)$ is c .*

- *Let ϕ be a match of a tree pattern Q against an XML instance I_X and V the set of variables in Q. Let $v \in V$ be a variable associated with a node n. Then the **variable binding** f **of** Q **against** I_X corresponding to ϕ is a function over V such that: (i) if v is a* uri *variable, then $f(v)$ is the URI of $\phi(n)$ in I_X, (ii) if v is a* val *variable, then $f(v)$ is the value of $\phi(n) \in I_X$, and (iii) if v is a* cont *variable, then $f(v)$ is the serialization of the subtree of I_X rooted at $\phi(n)$.*

- *Let Q be a triple pattern (s, p, o), I_R an RDF instance and I_R^∞ the saturation of I_R. A **match of** Q **against** I_R is a mapping from $\{s, p, o\}$ to the components of a single triple $t_\phi = (s_\phi, p_\phi, o_\phi) \in I_R^\infty$, such that $\phi(s) = s_\phi$, $\phi(p) = p_\phi$ and $\phi(o) = o_\phi$, and for any URI or literal ul appearing in s, p or o, we have $\phi(ul) = ul$ (ϕ maps any URI or literal only to itself).*

- *Let ϕ be a match of a triple pattern Q against an RDF instance I_R. Then the **variable binding of** Q **against** I_R corresponding to ϕ is the function $\phi|_V$, where V is the set of variables in Q.*

- *Let Q be an XRQ query, V be its set of variables, and $\langle v_1, v_2, \ldots, v_n \rangle$ the head variables of Q. Let $I = (I_X, I_R)$ be an XR instance.*

  *A **variable binding** f **of** Q **against** I is a function over V, such that for every tree (resp., triple) pattern $P \in Q$ whose variables we denote V_P, $V_P \subseteq V$, $f|_{V_P}$ is a variable binding of P against I_X (resp., I_R).*

- *The **result of** Q **over** I, denoted $Q(I)$, is the set of tuples:*

$$\{\langle f(v_1), f(v_2), ..., f(v_n)\rangle \mid f \text{ is a variable binding of } Q \text{ against } I\}$$

  In case of a boolean query, the singleton set $\{\langle\rangle\}$ containing the empty tuple corresponds to true and the empty set of tuples $\{\}$ to false.

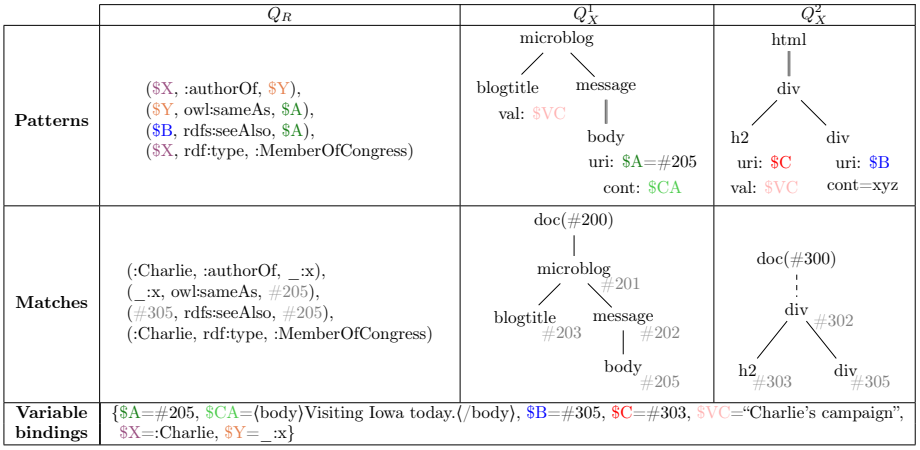


Figure 3: Pattern matches and variable binding of the query of Figure 2 on the XR instance of Figure 1.

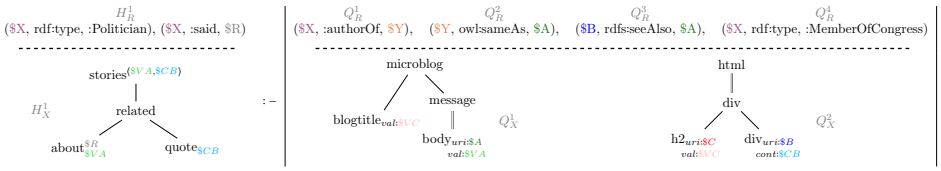


Figure 4: Sample XRQ$_{ext}$ query a grouping label on the top node

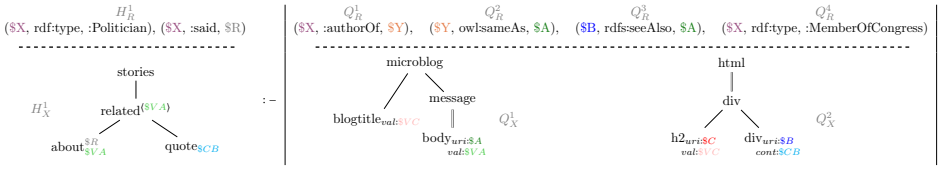


Figure 5: Sample XRQ$_{ext}$ query grouping labels on intermediate nodes

3 The XRQ$_{ext}$ query language

XRQ$_{ext}$ allows querying an XR instance w.r.t. both its structure (described in the XML sub-instance) and its semantic annotations (modeled in the RDF sub-instance), and in turn producing XR data as results. In the sequel, we present the syntax and semantics of the extended language, reusing as much as possible the syntax and semantics of the core one.

3.1 XRQ$_{ext}$ syntax

An XRQ query returns a set of tuples. Since the input is an XR instance, one should ideally be able to create such an instance as the output of the query. To this end, we extend our query language by augmenting it with data constructors.

The head of an extended query not only allows the generation of trees and triples in the output but also allows triples that annotate fresh nodes.

Definition 3.1 (Tree Constructor)**.** *A **tree constructor** is a finite, ordered, unranked, $\mathcal{N}$-labeled tree child edges only. We may attach to each node at most one* assignment label *consisting of a single variable and one* grouping label *consisting of a tuple of variables and constants. This tuple may not contain any variable already present in the grouping label of an ancestor. When omitted, the empty tuple is assumed. We may also attach to each leaf node a* value label *consisting of a single constant or variable.*

In practice, value labels are presented in subscript, while assignment labels and grouping labels are in superscript, with the latter between angle brackets. When a node features both an assignment label and a grouping label, the assignment label comes first and the two labels are separated with the ":=" symbol. Note that the constraints imposed on grouping labels imply that any variable may only appear once on a root-to-leaf path in a tree.

Example. The trees at the bottom left hand side of Figures 4 and 5 are tree constructors. The constructors are identical except in the way they are labeled. In Figure 4, the top *stories* node has a grouping label featuring variables $\$VA$ and $\$CB$, the *about* node at the bottom left is labeled with an assignment label $\$R$ and a value label $\$VA$, while the *quote* node at the bottom right has the value label $\$CB$. In contrast, the top *stories* node in Figure 5 does not have any label, but its child *related* has the grouping label featuring variable $\$VA$. The "about" and "quote" nodes are labeled similarly to Figure 4.

Definition 3.2 (XRQ$_{ext}$query head)**.** *An XRQ$_{ext}$ is a tuple (H_X, H_R, h), where H_X is a set of tree constructors, H_R is a set of triple patterns, and h is a tuple of variables or constants in $\{\mathcal{U} \cup \mathcal{L}\}$.*

For each tree constructor $t_X \in H_X$, and for each node $n_X \in t_X$:

- *if n_X has a* assignment label *v, then v is a variable of type* URI *s.t. $v \notin h$,*
- *if n_X has a* grouping label *of the form $(l_1, \ldots, l_k)$, then $l_i \in \{\mathcal{L} \cup \mathcal{U} \cup h\}$,*
- *if n_X has a* value label *l, then $l \in \{\mathcal{L} \cup \mathcal{U} \cup h\}$.*

Finally, each triple pattern $t_R \in H_R$ complies to Definition 2.1 with the restriction that variables may only belong to h or be assignment labels appearing in H_X.

Example. The left hand side of Figures 4 and 5 are both query heads made of two triple patterns and a single tree constructor, with $h = (\$X, \$VA, \$CB)$ and the assignment label \$R. The head triple patterns feature \$R and the variable \$X appearing in h.

Let $\mathcal{S}$ be an infinite set of Skolem functions, such that $\forall s \in \mathcal{S}$, $s : (\mathcal{U} \cup \mathcal{L})^{\mathbb{N}} \to \mathcal{U}$ is a function returning a fresh URI for any new input tuple of URIs or literals. As customary, we impose that the ranges of Skolem functions in $\mathcal{S}$ are disjoint, i.e., $\forall t, t' \in (\mathcal{U} \cup \mathcal{L})^{\mathbb{N}}$ and $\forall s^i, s^j \in \mathcal{S}$ where $s^i \neq s^j$, the following holds: $s^i(t) \neq s^j(t')$.

Definition 3.3 (XRQ$_{ext}$ Syntax)**.** *An XRQ$_{ext}$ query, denoted $Q = (H_X, H_R, h, Q_X, Q_R, \text{SK})$, consists of a head (H_X, H_R, h), a core query (h, Q_X, Q_R) and a bijective mapping $\text{SK} : H_X \to \mathcal{S}$, assigning a distinct Skolem function to each node of H_X.*

Example. Figures 4 and 5 depicted XRQ$_{ext}$ queries whose body is similar to that of the query in Figure 2. The two queries only differ in their grouping labels. Figure 4 has a grouping label $\langle \$VA, \$CB \rangle$ on the top node and none on the "related" node, while Figure 5 has the grouping label $\langle \$VA \rangle$ on the "related" node and none on the top node.

3.2 XRQ$_{ext}$ semantics

We now formalize the semantics of the extended language. In the following definition, a variable binding f of Q against an XR instance I is defined as for XRQ queries in Definition 2.3.

Definition 3.4 (XRQ$_{ext}$ semantics)**.** *The result of an XRQ$_{ext}$ query $Q = (H_X, H_R, h, Q_X, Q_R, \text{SK})$ over $I = (I_X, I_R)$, is an XR instance (I'_X, I'_R).*

I'_X is a forest of XML trees resulting from the replication of H_X for each binding f of Q against I. The URI of a node n^f_X, corresponding to a binding f and a node $n_X \in H_X$, is given by the Skolem function $\text{SK}(n_X)$. The input tuple of the function is obtained by appending constants and images of f for each variables appearing in grouping labels of n_X's ancestors (from the root to n_X), followed by the value bound to the node value label's variable, if any. Nodes with identical URIs coincide. If n_X has an assignment label \$w, the URI of n^f_X is bound to \$w in f. If n_X has a value label \$v,n_X is endowed with the text value $f(\$v)$.

I'_R is a union of triples obtained by replicating H_R for each binding f, replacing each variable \$v by $f(\$v)$, and each blank node _:b *with a fresh blank node.*

Example. Figures 6 and 7 show the results of the queries depicted in Figures 4 and 5 respectively, with the following bindings (we omitted variables that do not appear in the head):

$$
\begin{array}{lllll}
f_1 = & \{\$X = :Alice, & \$VA = Message1, & \$CB = \langle div \rangle StoryA\langle / \rangle\} \\
f_2 = & \{\$X = :Alice, & \$VA = Message1, & \$CB = \langle div \rangle StoryB\langle / \rangle\} \\
f_3 = & \{\$X = :Bob, & \$VA = Message2, & \$CB = \langle div \rangle StoryB\langle / \rangle\} \\
f_4 = & \{\$X = :Alice, & \$VA = Message1, & \$CB = \langle div \rangle StoryB\langle / \rangle\}
\end{array}
$$

XML node URIs are indicated in gray subscripts, each are prefixed with $\#sk_i$: to indicate that the URI was obtained through the i^{th} Skolem function of $\mathcal{S}$.

In Figure 4, the root node has grouping label $\langle \$VA, \$CB \rangle$. Its result is obtained as follows. Let us consider the first binding f_1. After copying the root node, its URI is assigned through the function call $sk_1((f_1(\$VA), f_1(\$CB)))$, where sk_1 is the Skolem function assigned to the root node in H_X. This returns URI $\#sk_1$:1. The URI of the node label "related" will be obtained through the call $sk_2((f_1(\$VA), f_1(\$CB)))$, which returns $\#sk_2$:1. Since the node has no grouping label, the empty list is assumed. The input of sk_2 is a concatenation of the variable bindings declared in the grouping label of all the node's ancestors down to the current one. Since the root node's grouping label already contained all possible variables, the input tuple of sk_2 will be the same as the input tuple of sk_1. This applies to all nodes in H_X, to produced R_1, the left-most tree on Figure 6. The second and third trees R_2 and R_3 are built in a similar manner. Since, the variable bindings of f_1, f_2 and f_3 are all different, the sets of URIs for nodes of each tree are disjoint. However, the binding f_4 is identical to f_2, thus Skolem function calls for each node of the tree associated with f_4 return the same URIs as in R_2, eventually merging the two trees into a single one. The URIs of the "about" nodes, $\#sk_3$:1, $\#sk_3$:2, $\#sk_3$:3 and $\#sk_3$:2, are

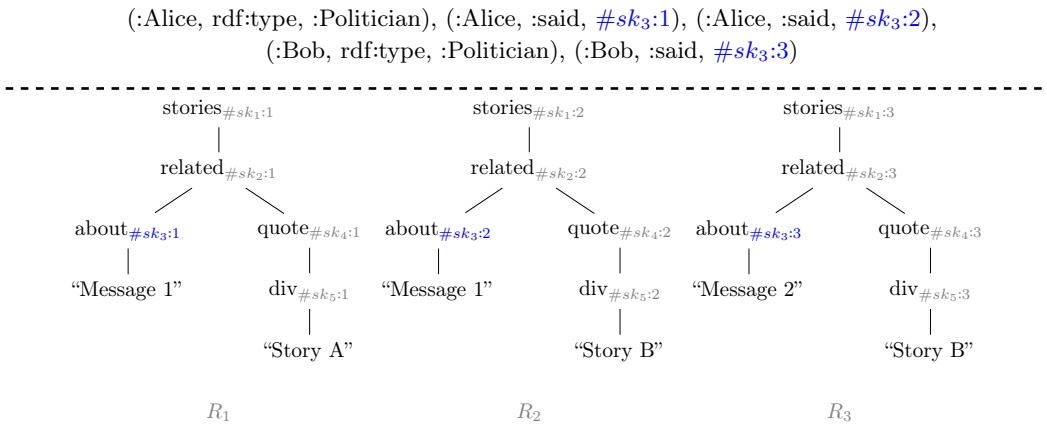


Figure 6: Result of an XRQ$_{ext}$ query without grouping nodes

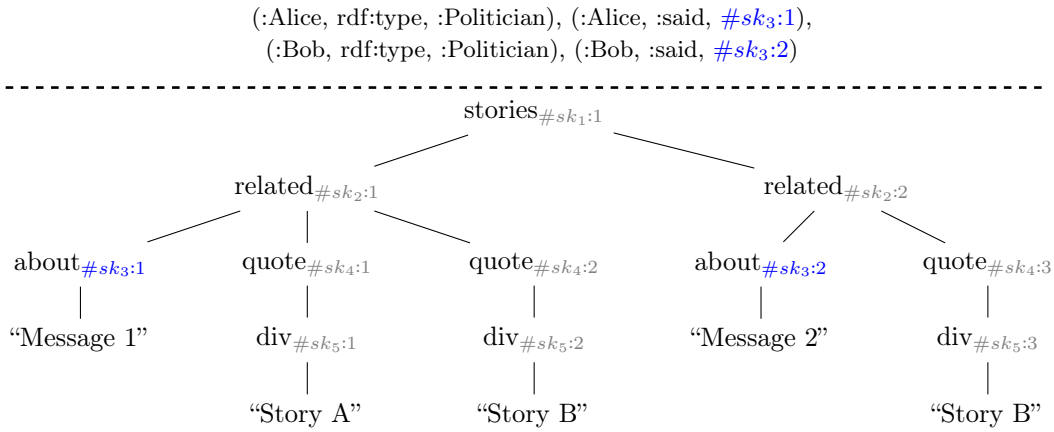


Figure 7: Result of an XRQ$_{ext}$ query with grouping nodes

bound to \$R and added to bindings f_1, f_2, f_3 and f_4 respectively. These are necessary to produce the triples shown at the top of Figure 6.

Next, we explain Figure 7, which represents the result of the query depicted on Figure 5. Since no grouping label is defined on the root node, the empty list is assumed, then $sk_1(())$ is called to assign a URI to the first node as well as every subsequent copies of the root node. Consequently, all the top nodes in the results will be merged into a single one. Next, the "related" nodes are produced. Their corresponding grouping label includes \$VA, therefore the input of sk_2 is $(f_i(\$VA))$, for each finding f_i, with $1 \leq i \leq 4$. Calls to Skolem function for this node with bindings f_2 and f_4 will return the same URIs, since the same value is bound to \$VA in those bindings. However, for binding f_3 a fresh node will be generated. The input of the Skolem function for every child of the "related" nodes will be the same, as none of them specifies any additional variable. The node label "quote" however, features the variable \$CB as a value label, therefore the input of the Skolem function for this node will be $(f_i(\$VA), f_i(\$CB))$, i.e., the concatenation of the input of its ancestors and \$CB, resulting in the grouping shown in the Figure.

The construction of XML nodes affects the construction RDF triples. In Figure 6, the three distinct "related" nodes led to the creation of three corresponding triples. In Figure 7, only two "related" nodes were ultimately generated. The variable \$R was bound to the same URI sk_3:1 in the three bindings f_1, f_2 and f_4, leading to three identical triples (:Alice, :said, $\#sk_3 : 1$). Due to the set semantics of RDF, these coincide into a single one in the RDF sub-instance.

Note that the type of nodes created from a value label v depends on the type of v. If v is a *uri* or *val* variable, newly created nodes will be text nodes. If the v is a *Cont* variable, the node will be an *element* or *attribute* node, possibly with descendant nodes itself. Besides this difference, the semantics of value labels is essentially the same regardless of the variable types and, wlog, we will only refer to *uri* and *val* variables hereafter.

We call XRQ$^{\cup}_{ext}$ a variant of the query language allowing *unions* of XRQ$_{ext}$ queries.

Definition 3.5 (Unions of XRQ_{ext})**.** *An $XRQ^{\cup}_{ext}$ query is a set of XRQ_{ext} queries. For any data instance $\mathcal{I}$, the result of an $XRQ^{\cup}_{ext}$ query is the union of the results of its sub-queries over $\mathcal{I}$.*

Among others, the language allows the empty union, whose semantics is the empty result.

The semantics of $\mathrm{XRQ}^{\cup}_{ext}$ follows that of XRQ_{ext}. Note that although the node-to-Skolem function mapping Sk of Definition 3.4 is bijective, nodes from tree constructors of distinct sub-queries in a union may map to the same Skolem function. As a consequence, if data nodes yielded by distinct sub-queries have the same URI, they will coincide in the result of the union. The interest of unions of XR queries will become clear when we tackle the problem of composition in the next section.

4 Containment & equivalence

We now define various notions of containment and equivalence in the core and extended XR query language. Let q and q' be two XR queries. We denote by $q_{core}(\mathcal{I})$, the set of tuples resulting from the evaluation q over $\mathcal{I}$, and $q(\mathcal{I})$, the instance resulting from the evaluation q over $\mathcal{I}$.

The first notion refers to the containment between queries in the core language.

Definition 4.1 (Core XRQ containment ($\sqsubseteq_c$))**.** *q' is* core-contained *in q, noted $q' \sqsubseteq_c q$, iff for any instance $\mathcal{I}$, then $q'_{core}(\mathcal{I}) \subseteq q_{core}(\mathcal{I})$.*

The second notion relates to the extended semantics of XRQ, and considers a "weak" notion of containment, in which results of two queries do not need to share URIs to be contained into one another.

Definition 4.2 (Weak XRQ containment ($\sqsubseteq_w$))**.** *q' is* weakly contained *in q, noted $q' \sqsubseteq_w q$, iff for any instance $\mathcal{I}$, there is a function μ, such that:*

- *for all XML nodes $n \in q'(\mathcal{I})$, $\mu(n) \in q(\mathcal{I})$, where $\lambda(n) = \lambda(\mu(n))$ and $\tau(t) = \tau(\mu(n))$, i.e., μ maps any XML node to a node with the identical label and type,*
- *for all XML edges $(n_1, n_2) \in q'(\mathcal{I})$, $(\mu(n_1), \mu(n_2)) \in q(\mathcal{I})$,*
- *for all RDF triples $(s, p, o) \in q'(\mathcal{I})$:*
  - *if $\exists n \in N$ such that $s = \gamma(n)$, then $(\gamma(\mu(n)), p, o) \in q(\mathcal{I})$,*
  - *if $\exists m \in N$ such that $o = \gamma(m)$, then $(s, p, \gamma(\mu(m))) \in q(\mathcal{I})$,*
  - *if $\exists n, m \in N$ such that $s = \gamma(n)$ and $o = \gamma(m)$, then $(\gamma(\mu(n)), p, \gamma(\mu(m))) \in q(\mathcal{I})$,*
  - *otherwise $(s, p, o) \in q(\mathcal{I})$.*

Note that μ is an injective homomorphism, i.e., it defines an embedding from $q'(\mathcal{I})$ to $q(\mathcal{I})$ up to URI renaming.

The last notion further restricts the containment by requiring, node URIs of one query to coincide with the node URI of the images through the homomorphism μ.

Definition 4.3 (Strong XRQ containment ($\sqsubseteq_s$))**.** *q' is* strongly contained *in q, noted $q' \sqsubseteq_s q$, iff for any instance $\mathcal{I}$, $q' \sqsubseteq_w q$, where μ is a function as described in 4.2, with the additional property that for all XML node $n \in q'(I)$, $\mu(n) \in q(\mathcal{I})$ and $\gamma(n) = \gamma(\mu(n))$.*

As customary, equivalence between two queries is defined as two-ways containment:

Definition 4.4 (Core, weak and strong equivalence)**.**

- *$q \equiv_c q'$ iff $q \sqsubseteq_c q'$ and $q' \sqsubseteq_c q$*
- *$q \equiv_w q'$ iff $q \sqsubseteq_w q'$ and $q' \sqsubseteq_w q$*
- *$q \equiv_s q'$ iff $q \sqsubseteq_s q'$ and $q' \sqsubseteq_s q$*

Note that if $q \equiv_s q'$, then for any instance $\mathcal{I}$, $q(\mathcal{I}) = q'(\mathcal{I})$

5 View composition

5.1 Problem statement

Given a view v, without fresh blank nodes in the head, and a query q, the *view composition* problem consists in finding a query q' such that for any XR instance $\mathcal{I}$, $q'(\mathcal{I}) = (q \circ v)(\mathcal{I})$.

The notations $(q \circ v)(\mathcal{I})$ and $q(v(\mathcal{I}))$ are equivalent and may be used interchangeably hereafter.

5.2 Preliminaries

We introduce the notion of normalization, which is used in the composition algorithm. The normalization $\overline{q}$ of an XRQ query q is a First-Order formula, whose predicates are among $\{Triple, Node, Val, Edge, Path, =\}$, where $Triple$ is ternary predicate and all others are binary predicates.

Normalization serves two purposes. On the one hand, it is used in the composition algorithm to find matches between the query body and the view head at a fine granularity. On the other hand, being based on First-Order Logic, it provides a very general framework to reason about the algorithm. In particular, we prove the soundness and completeness of the algorithm using the normalized form of XR queries and instances.

The normalization procedure consists of a set of rules, described below, to be applied to an XR query. Each rule looks at a particular syntactic item of the body or head of the input XR query, and produces one or two atoms respectively in the body or head of the normalized one. The rules are slightly different for the head and body of the input query. First, we describe the normalization rules of the latter:

Definition 5.1 (Body normalization)**.** *Let $Q = (H_X, H_R, q, Q_X, Q_R, \text{SK})$ be an XR query, where $n_1, \ldots, n_n$ are XML nodes appearing in Q_X and $(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)$ are triples patterns appearing in Q_R.*

$$\frac{n_i \in Q_X}{Node(x, t_i)}, \tag{1}$$

where t_i is the tag of node n_i, x is the URI variable labeling n_i if any, a fresh variable otherwise.

$$\frac{n_i \in Q_X,\ s.t.\ n_i\ has\ a\ selection\ label\ of\ the\ form\ uri = c}{x = c} \tag{2}$$

$$\frac{n_i \in Q_X,\ s.t.\ n_i\ has\ a\ Val\ variable\ label\ of\ the\ form\ val : v}{Val(x, v)} \tag{3}$$

$$\frac{n_i \in Q_X,\ s.t.\ n_i\ has\ a\ selection\ label\ of\ the\ form\ val = c}{Val(x, y), y = c} \tag{4}$$

where x is the URI variable labeling n_i if any, a fresh variable otherwise, and y is the Val variable labeling n_i if any, a fresh variable otherwise.

$$\frac{n_i, n_j \in Q_X,\ such\ that\ n_i\ is\ a\ single\text{-}edge\ parent\ of\ n_j}{Edge(x, y)} \tag{5}$$

$$\frac{n_i, n_j \in Q_X,\ such\ that\ n_i\ is\ a\ double\text{-}edge\ parent\ of\ n_j}{Path(x, y)} \tag{6}$$

where x (resp. y) matches the first attribute of the $Node$ predicate produced by rule 1 for the node n_i (resp. n_j).

$$\frac{(s_i, p_i, o_i) \in Q_R}{Triple(s'_i, p_i, o'_i)} \tag{7}$$

where p_i is a variable or a constant and $s_i = s'_i$ (resp. $o_i = o'_i$) if s_i is a variable or a constant and s'_i (resp. o'_i) is a fresh variable if s_i (resp. o_i) is a blank node.

The restrictions on the rules impose that rule 1 be applied exhaustively before any other rule.

Intuitively, normalizing the body of an XR query produces a $Triple$ atom for each triple pattern in the query body, a $Node$ atom for each XML node, an $Edge$ atom for each parent-child edge and a $Path$ for each ancestor-descendant edge. The presence of Val variables and selections on tree patterns gives rise to Val and = predicates. The arguments of the $Edge$ and $Path$ predicates, as well as the first

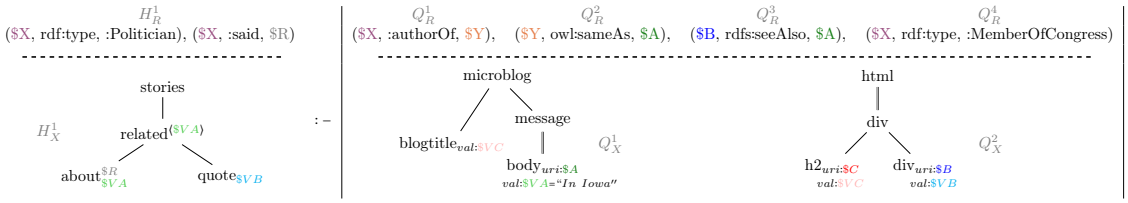


Figure 8: Variant of the query depicted on Figure 5

XML nodes	XML edges & Paths	RDF triple patterns
Node(v_1, microblog)	Edge(v_1, v_2)	Triple($X, :*authorOf*, $Y)
Node(v_2, blogtitle)	Edge(v_1, v_3)	Triple($Y, *owl:sameAs*, $A)
Node(v_3, message)	Edge(v_3, $A)	Triple($B, rdfs:seeAlso, $A)
Node($A, body)	Edge(v_4, v_5)	Triple($X, rdf:type, :*MemberOfCongress*)
Node(v_4, html)	Edge(v_5, $C)	
Node(v_5, div)	Edge(v_5, $B)	
Node($C, h2)	Path(v_3, $A)	
Node($B, div)	Path(v_4, v_5)	
Val(v_2, $VC))		
Val($A, "In Iowa")		
Val($B, $VB)		
Val($C, $VC)		

Table 1: Normalization for the body of the query described in Figure 8

argument of $Node$ and Var admit a variable that represents a node's URI. Since, all XML nodes of an XR instance have distinct URIs, this allows our set of atoms to represent the relationships among the nodes of tree patterns. The second arguments of the $Node$ and Val act as selections on a node's tag and string value respectively.

Note that document nodes are not considered here. Adding normalization rules to produce *Document* predicates is trivial, yet, document nodes are not allowed in the head of views. Therefore in practice, this would render any composed query empty when the input query has document nodes in the body.

Example. As an example, consider the query depicted on Figure 8, a variant of the one depicted on 5, where a selection has been added on the *body* node of the rightmost tree pattern. Table 1 lists the atoms obtained by normalizing its body. Fresh variables are all of the form $\$v_i$. Constants and variables that appeared in the input query are left intact and are all present in the normalization.

Definition 5.2 (Head normalization)**.** *Let $Q = (Q = (H_X, H_R, q, Q_X, Q_R, \text{SK}))$ be an XR query, where $n_1, \ldots, n_n$ are XML nodes appearing in H_X and $(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)$ are triples patterns appearing in H_R.*

$$\frac{n_i \in H_X}{Node(x, t_i)}, \tag{8}$$

where t_i is the tag of node n_i, x is the assignment label of n_i if any, a fresh variable otherwise.

$$\frac{n_i \in H_X,\ s.t.\ n_i}{x = sk_i} \tag{9}$$

where x is the assignment label of n_i, if any, a fresh variable otherwise and sk_i is the URI assignment Skolem function call for n_i.

$$\frac{n_i \in H_X,\ s.t.\ n_i\ has\ a\ value\ label\ y}{Val(x, y)} \tag{10}$$

$$\frac{n_i, n_j \in H_X,\ such\ that\ n_i\ is\ a\ single\text{-}edge\ parent\ of\ n_j}{Edge(x, y)} \tag{11}$$

$$\frac{n_i, n_j \in H_X,\ such\ that\ n_i\ is\ an\ ancestor\ of\ n_j}{Path(x, y)} \tag{12}$$

XML nodes and values	XML paths and edges	RDF triple patterns
Node($f_{stories}$(), stories)	Edge($f_{stories}$(), $f_{related}$(*"In Iowa"*))	Triple($\$X$, rdf:type, *:Politician*)
Node($f_{related}$(*"In Iowa"*), related)	Edge($f_{related}$(*"In Iowa"*), f_{about}(*"In Iowa"*))	Triple($\$X$, *:said*, f_{about}(*"In Iowa"*))
Node(f_{about}(*"In Iowa"*), about)	Edge($f_{related}$(*"In Iowa"*), f_{quote}(*"In Iowa"*, $\$VB$))	
Node(f_{quote}(*"In Iowa"*, $\$VB$), quote)	Path($f_{stories}$(), $f_{related}$(*"In Iowa"*))	
Val(f_{about}(*"In Iowa"*), *"In Iowa"*)	Path($f_{related}$(*"In Iowa"*), f_{about}(*"In Iowa"*))	
Val(f_{quote}(*"In Iowa"*, $\$VB$), $\$VB$)	Path($f_{related}$(*"In Iowa"*), f_{quote}(*"In Iowa"*, $\$VB$))	
	Path($f_{stories}$(), f_{about}(*"In Iowa"*))	
	Path($f_{stories}$(), f_{quote}(*"In Iowa"*, $\$VB$))	

Table 2: Normalization for the head of the query depicted on Figure 8

where x (resp. y) matches the first attribute of the Node predicate produced by rule 8 for the node n_i (resp. n_j).

$$\frac{(s_i, p_i, o_i) \in H_R}{Triple(s_i, p_i, o_i)} \tag{13}$$

where s_i, p_i and o_i are variables or constants.

As for Procedure 5.1, rule 8 must be applied exhaustively before all the others.

After applying the normalization rules, the resulting expressions can be reduced by propagating equality constraints, i.e., for each equality of the form *var* = *const*, replacing all occurrences of *var* by *const*, then discard the constraint.

The main differences with body normalization procedure are the following: (i) *Path* atoms are produced for all pairs of nodes that reside on the same path (top-down), (ii) the Skolem function calls that happen implicitly upon query evaluation are revealed in every places they effectively occur.

Example. In Table 2, we present the normalization of the query depicted on Figure 8. Equality constraints have already been propagated. Notice how the selection applied in the body to variable $VA propagates to the body and head atoms. Likewise, although variable $\$R$ initially appears in the first stages of the normalization, it was ultimately replaced by the constant f_{about}(*"In Iowa"*), due to the equality constraint $\$R = f_{about}$(*"In Iowa"*) (generated by Rule 4).

We now extend the concept of normalization to XR instances.

Definition 5.3 (Instance normalization)**.** *We call $\overline{\mathcal{I}}$, the normalization of an instance $\mathcal{I}$, a set of atoms whose parameters are constants or blank nodes, obtained by applying the following rules onto $\mathcal{I}$:*

$$\frac{(s, p, o) \in \mathcal{I}}{Triple(s', p, o')}, \tag{14}$$

where p is a constant, and $s = s'$ (resp. $o = o'$) if s (resp. o) is a constant, s' (resp. o') is a fresh variable if s (resp. o) is a blank node.

$$\frac{n \in \mathcal{I}}{Node(u_n, t_n)}, \tag{15}$$

where u_n is the URI of n and t_n its tag.

$$\frac{n, m \in \mathcal{I},\ s.t.\ n \text{ and } m \text{ are parent and child nodes of an edge}}{Edge(u_n, u_m)}, \tag{16}$$

where u_n,u_m are the URIs of n and m respectively.

$$\frac{n, m \in \mathcal{I},\ s.t.\ n \text{ and } m \text{ are ancestor / descendant node pair of any path}}{Path(u_n, u_m)}, \tag{17}$$

where u_n,u_m are the URIs of n and m respectively.

$$\frac{n \in \mathcal{I},\ s.t.\ n_i \text{ has a text child}}{Val(u_n, c)}, \tag{18}$$

where c is the value of the text node.

	Head of $\overline{q}$	**Body of $\overline{q}$**
# *Triple* predicates	$\lvert H_R \rvert$	$\lvert Q_R \rvert$
# *Node* predicates	$\lvert H_X \rvert$	$\lvert Q_X \rvert$
# *Val* predicates	$\lvert H_X \rvert$	$\lvert Q_X \rvert$
# *Edge* predicates	$\leq \lvert H_X \rvert - 1$	$\leq \lvert Q_X \rvert - 1$
# *Path* predicates	$\leq \binom{\lvert H_X \rvert}{2}$	$\leq \lvert Q_X \rvert - 1$

Table 3: Number of atoms produced by normalizing of an XR query q into a FOL formula $\overline{q}$

Complexity of the normalization. The size of a normalized query is dominated by the number of *Path* predicates generated by rule 12 making the normalization result complexity quadratic in the size of the original query. Let $q = (H_X, H_R, Q_X, Q_R)$ be an XRQ$_{ext}$ query and $\overline{q}$ its normalization. We note $|Q_R|$ and $|H_R|$ the number of triple patterns in the head and body of q respectively, $|Q_X|$ and $|H_X|$ the number of XML nodes in q. The number of atoms produced by the normalization is given in Table 3.

Substitution & unification. Recall that a *substitution* is a mapping from variables to terms (i.e., in our case, variables, constants and function calls). Let σ be a substitution and ϕ a FO formula, we write ϕ^σ to designate a copy of ϕ in which each variable $v \in \phi$ has been replaced with $\sigma(v)$. By extension, we note q^σ the extended XR query q that has undergone the substitution σ. We may use the set notation $\sigma = \{x_1 \to y_1, \ldots, x_n \to y_n\}$, i.e., x_1 maps to y_1, …, x_n maps to y_n, to describe σ completely. Inconsistent mappings are noted $\perp$.

We call *unification*, the process of finding the *most general unifier (MGU)* between two formulas ϕ_1 and ϕ_2, i.e., a unifier $\hat{\sigma}$ such that $\phi_1^{\hat{\sigma}} = \phi_2^{\hat{\sigma}}$ and $\nexists \sigma'$ where σ' is a unifier and σ' subsumes $\hat{\sigma}$.

5.3 Composition algorithm

Algorithm 1 takes as input an XRQ$_{ext}$ view v and an XRQ$_{ext}$ query q, and returns an XRQ$^{\cup}_{ext}$ query q'. We briefly present the main functions used by the algorithm.

- *head* and *body* respectively return the head and body of the input query or view;
- **normalize** takes a query or view as input and returns its normalization;
- **assignFreshVars** replaces the name of each distinct variable of the input expression with a fresh one;
- The function **mgu** takes two atoms as inputs and returns their *MGU* if one exists, and $\perp$ otherwise.

Algorithm 1 has two phases:

1. During the matching phase (lines 4-10), we attempt to unify each atom p_i in the body of the normalized query $\overline{q}$, with each head atom from the normalized view $\overline{v}$. For each atom p_i, we also keep a copy of $\overline{v}$, whose variables have been replaced with fresh ones. Whenever a valid substitution is found, it is added to a set of valid substitutions Σ_i associated with p_i. At the end of the matching phase, n sets of substitutions have been created, one for each atom from the normalized query body.
2. During the building phase (lines 12-18), we iterate over the Cartesian product of these sets, and we form the union of the substitutions in each entry. We call this union a *compound substitution*, noted $\sigma_\cup$. Each compound substitution covers all the atoms in the body of q. A compound substitution may be inconsistent, in which case it is discarded. Consistency can be checked applying simple rules. For instance, if a single variable maps to two distinct constants, the substitution is inconsistent.

   For each consistent compound substitution $\sigma_\cup$, we build a sub-query q'_σ, whose head is obtained by applying $\sigma_\cup$ onto the head of q, and whose body is built by joining $body(v_1)$ with $body(v_2)$, …, $body(v_n)$, and applying $\sigma_\cup$ on the join result. The final query q' is the union of the sub-queries obtained for all compound substitutions.

When two atoms feature distinct constants in a given position, they cannot be unified. Hence, if an atom from the query cannot be unified with any normalized view head atoms, the algorithm will return the empty union. In Algorithm 1, this is apparent in the fact that if a single set Σ_i is empty, the Cartesian product in the build phase is also empty.

Algorithm 1: XRComp

Input : view v, query q
Output: a composed query q', such that for any instance $\mathcal{I}$, $q'(\mathcal{I}) = (q \circ v)(\mathcal{I})$

$\overline{v} \leftarrow$ **normalize**(v)
$\overline{q} \leftarrow$ **normalize**(q)
Let n be the number of atoms in $body(\overline{q})$
//Matching stage
foreach *atom* $p_i \in body(\overline{q})$ **do**
  $\overline{v_i} \leftarrow$ **assignedFreshVars**$(\overline{v})$
  let Σ_i be a set of substitutions, $\Sigma_i \leftarrow \varnothing$
  foreach *atom* $p_j \in head(\overline{v})$ **do**
    $\sigma_j^i \leftarrow$ **mgu**(p_i, p_j)
    if $\sigma_j^i \neq \perp$ **then**
      $\Sigma_i \leftarrow \Sigma_i \cup \{\sigma_j^i\}$
//Building stage
$q' \leftarrow \varnothing$
foreach *set* $\sigma \in \Sigma_1 \times \cdots \times \Sigma_n$ **do**
  $\sigma_\cup = \bigcup_{\sigma' \in \sigma} \sigma'$
  if $\sigma_\cup \neq \perp$ **then**
    $q'_\sigma \leftarrow \varnothing$
    $head(q'_\sigma) \leftarrow head(q)^{\sigma_\cup}$
    $body(q'_\sigma) \leftarrow (body(v_1), \ldots, body(v_n))^{\sigma_\cup}$
  $q' \leftarrow q' \cup q'_\sigma$
return q'

5.3.1 Properties

We now discuss the properties of Algorithm 1. We want to show that, for any given input query q, view v and instance $\mathcal{I}$, Algorithm 1 builds an XRQ$^\cup_{ext}$ query q', such that $q'(\mathcal{I}) = q(v(\mathcal{I}))$. This can be proved by showing that q' is a *sound and complete* composition of q over v. More precisely, q' is *sound*, iff for every instance $\mathcal{I}$, $q'(\mathcal{I}) \subseteq q(v(\mathcal{I}))$, and *complete*, iff for every instance $\mathcal{I}$, $q(v(\mathcal{I})) \subseteq q'(\mathcal{I})$.

Soundness & completeness of the algorithm. Observe that the notions of mapping and variable binding defined in the semantics, directly translate to the notion of homomorphism and substitution in normalized XR.

Given a query q, an instance $\mathcal{I}$, a match μ of q over $\mathcal{I}$ and its associated binding β of q variables into $\mathcal{I}$, there exists a corresponding homomorphism $\overline{\mu} : body(\overline{q}) \to \overline{\mathcal{I}}$, and a substitution $\overline{\beta}$ mappings variables of $\overline{q}$ to constants of $\overline{\mathcal{I}}$, such that $\forall a \in body(q), a^{\overline{\beta}} = \overline{\mu}(a)$.

From now on, we only consider normalized XR and therefore, we omitted the overline previously used.

Let q be a query, v a view without fresh blank nodes in the head and q' the union of XR queries obtained by Algorithm 1 for q and v. Figure 9 gives an abstract representation of these queries over an instance, and some homomorphism that may exist if the queries are not empty.

Theorem 5.1 (Completeness of Algorithm 1)**.** *For any instance* $\mathcal{I}$*,* $q(v(\mathcal{I})) \subseteq q'(\mathcal{I})$.

Proof. Let t be a tuple of bindings in the result of $(q_{core} \circ v)(\mathcal{I})$

$$t \in (q_{core} \circ v)(\mathcal{I}) \Leftrightarrow \exists h_t : body(q) \to v(\mathcal{I}), \text{ s.t. } h_t(head(q_{core})) = t \tag{19}$$

The existence of the homomorphism h_t implies that each atom of $body(q)$ individually match with an atom of $head(v)$ and that these matches globally agree on the join variables. In other words,

$$\forall p_i \in body(q), \exists p_j \in head(v) \text{ s.t. } \exists \theta_j^i = MGU(p_i, p_j) \tag{20}$$

and

$$\forall x \in Var(p_{i_1}) \cap Var(p_{i_2}), \text{ we have } \theta_{j_1}^{i_1}(x) = \theta_{j_2}^{i_2}(x) \tag{21}$$

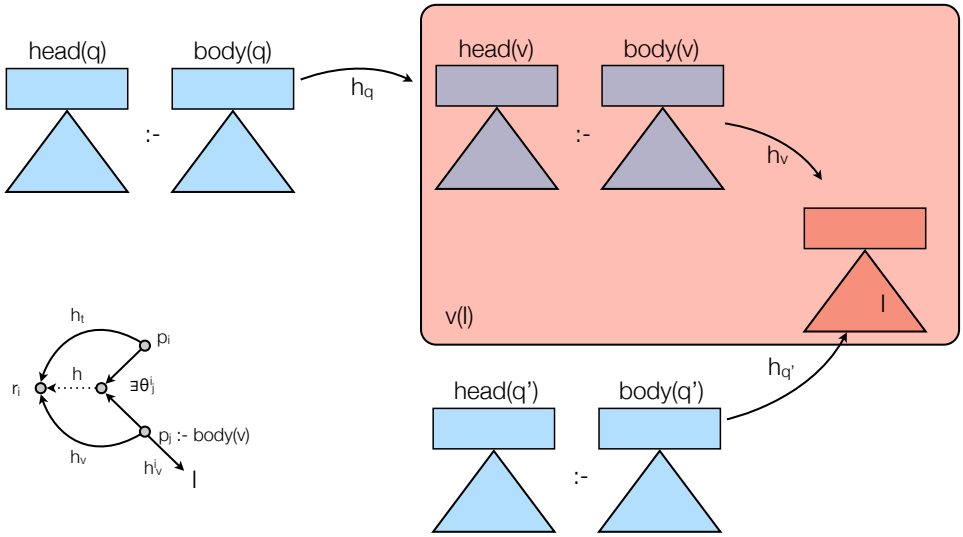


Figure 9: An abstract representation of q, v, q', $\mathcal{I}$, and some homomorphisms that may exist between them.

where function $Var()$ returns the set of variables in a given atom.

Let $r_i = h_t(p_i)$, where each r_i is a projection of t, then

$$\exists h^i_v : body(v) \to \mathcal{I} \text{ s.t. } h^i_v(p_j) = r_i \text{ or } r_i \in NonVar(p_j) \tag{22}$$

where $NonVar()$ is a function that returns the set of constants or function calls in a given atom.

Observe that any constant or function call in the range of θ^i_j results from the presence of a selection in p_i or data creation in p_j. Therefore, from (22), we can conclude that applying θ^i_j to v and projecting on p_j yields r_i.

$$r_i \in (p_j :- body(v))^{\theta^i_j}(\mathcal{I}) \tag{23}$$

Let $h = \bigcup_i h^i_v$ s.t. $(\bigwedge_i (p_j :- body(v_i))^{\theta^i_j}) \overset{h}{\mapsto} \mathcal{I}$, where v_i are variable-renamed copies of v, then any result $r_i \in t$ can be found through h or as a non-variable term in $head(v)$:

$$\forall p_i \in body(q), r_i \in \bigcup_i (h(Var(p_j^{\theta^i_j})) \ \cup \ NonVar(p_j^{\theta^i_j})) \tag{24}$$

The Cartesian product at line 12 explores all possible combinaisons of MGUs from each atom of $body(q)$ to an atom of $head(v)$. Therefore, for any result $r \in t$, there exists a sub-query q'_σ in q', s.t. $r \in q'_{\sigma\ core}(\mathcal{I})$.

□

Theorem 5.2 (Soundness of Algorithm 1)**.** *For any instance $\mathcal{I}$, $q'(\mathcal{I}) \subseteq q(v(\mathcal{I}))$.*

Proof. **Case 1:** $(q \circ v)(\mathcal{I})$ is not empty.

The matching phase of the algorithm (lines 4 to 10) builds, for each atom $p_i \in body(q)$, all valid MGUs with an atom of $head(v_i)$, where v_i is a variable-renamed copy of v. Each entry σ in the Cartesian product of line 12, featuring MGUs from all atoms of $body(q)$ to an atom in $head(v)$, is unioned into a compound substitution $\sigma_\cup$. We discard any $\sigma_\cup$ that is deemed inconsistent, i.e., for any substitution v/c_1 to be added to $\sigma_\cup$, if $\exists v/c_2 \in \sigma_\cup$, where c_1, c_2 are non-variable terms and $c_1 \neq c_2$. It follows that by construction $\sigma_\cup$ satisfies both (20) and (21), i.e.,

$$\forall p_i \in body(q), \exists p_j \in head(v_i) \text{ s.t. } \exists \sigma^i_j \in \sigma_\cup, \text{ where } \sigma^i_j = MGU(p_i, p_j). \tag{25}$$

and all σ^i_j agree on join variables.

The building phase (lines 12 to 18) creates for each consistent $\sigma_\cup$ a sub-query q'_σ by joining each $body(v_i)$ and applying $\sigma_\cup$. In other words,

$$body(q'_\sigma) = \bigwedge_i (body(v_i))^{\sigma^i_j} \tag{26}$$

and

$$head(q'_{\sigma\ core}) = (head(q_{core}))^{\sigma^i_j} \tag{27}$$

Let us consider a tuple $t' \in q'_{\sigma\ core}(\mathcal{I})$. There is an homomorphism $h^{\sigma}_{q'} : body(q'_{\sigma}) \to \mathcal{I}$ and any results $r' \in t'$ can found through $h^{\sigma}_{q'}$.

$$\exists x \in head(q'_{\sigma\ core}),\ \text{s.t.}\ h^{\sigma}_{q'}(x) = r' \tag{28}$$

More precisely, there exists a homormorphism $h^{\sigma^i}_{q'} : body(v_i)^{\sigma^i_j} \to \mathcal{I}$, s.t.

$$\exists x \in p_i,\ \text{s.t.}\ h^{\sigma^i}_{q'}(x) = r' \tag{29}$$

Since $body(v_i)^{\sigma^i_j}$ is a restriction of $body(v)$, it follows that t' is produced by $(q_{core} \circ v)(\mathcal{I})$.

We now show by contradiction that applying $\sigma_\cup$ to $head(q)$ preserves soundness. Observe that $head(q)^{\sigma_\cup}$ is a restriction of $head(q)$. Suppose there is a variable $v \in head(q)$, such that $\sigma_\cup(v) = c$, where c is a constant and there is no binding in the result of $(q \circ v)(\mathcal{I})$ for which c is bound to v. Then, since $\sigma_\cup$ is also applied to the body of q'_σ, necessarily $q'_\sigma(\mathcal{I}) = \varnothing$. Therefore, the following always holds:

$$q'_\sigma(\mathcal{I}) \subseteq (q \circ v)(\mathcal{I}) \tag{30}$$

This extends to all sub-queries of the union q', therefore:

$$q'(\mathcal{I}) \subseteq (q \circ v)(\mathcal{I}) \tag{31}$$

Case 2: $(q \circ v)(\mathcal{I})$ is empty.

If $v(\mathcal{I})$ is empty, then Theorem 5.2 trivially holds, since the body of each sub-query q'_σ made of joins over $body(v)$.

If $(q \circ v)(\mathcal{I})$ is empty while $v(\mathcal{I})$ is not, then there is no homomorphism from $body(q)$ to $v(\mathcal{I})$, i.e. either (20) or (21) is not satisfied. If (20) is not satisfied, at least one of the Σ_i produced in the matching phase will be empty, so will the Cartesian product $\{\Sigma_1 \times \cdots \times \Sigma_n\}$. If (21) is not satisfied, any entry σ in the Cartesian product will yield an inconsistent union of substitutions $\sigma_\cup$. In either case, the algorithm would return the empty union, hence $q'(\mathcal{I}) \subseteq q(v(\mathcal{I}))$.

□

Complexity of Algorithm 1. As we saw in Section 5.2, the complexity on the normalization steps at lines 1 and 2 are respectively quadratic and linear in the size of q. It directly follows that the nested loops of the matching phase (lines 4-10) run it cubic time. The **mgu** function (line 6) operates in constant time since we only attempt to unify pairs of atoms of arity 2 or 3. In the worst case, the size of the cartesian product explored in the building phase (lines 12-18) is exponential in the size of the q, i.e., $O(|\overline{v}|^{2\times|\overline{q}|})$. The consistency check performed at line 14 can be done in linear time in the size of q. Similarly, the substitutions involved in the creation of the UCQ (lines 15-17) can be performed in linear time. It follows that the time complexity of the building phase is overall exponential q and v, and in the worst case, so is the size of the UCQ produced during this phase.

5.3.2 Examples

In the sequel, we illustrate the composition algorithm through various examples. Unless specified otherwise, variables are of URI type.

Lifting and lowering examples. We now provide some examples of XRQ view, queries, along with their compositions, in the context of social networks. Query Q_1, depicted on Figure 10, on the one hand, evaluates over an XML document featuring message feeds between users. It retrieves the URIs and names of users who have sent messages to their friends, along with the URIs of these messages, of their recipients and the content value of the messages sent. It outputs roughly the same information in RDF, with additional triples stating that the sender knows the recipients and vice versa.

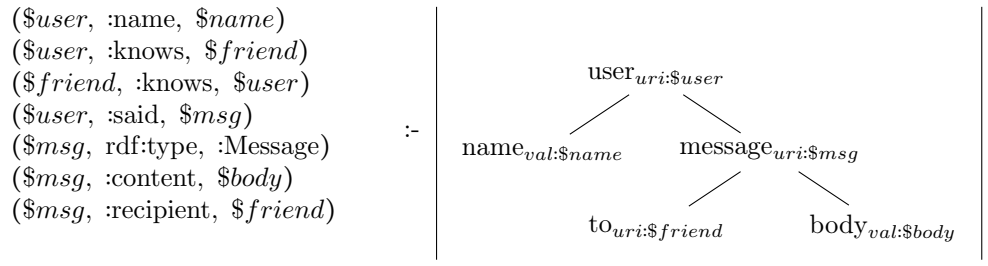


Figure 10: Query Q_1, *lifting* social network information from XML to RDF

Query Q_2, depicted on Figure 11, on the other hand, evaluates over an RDF sub-instance. It retrieves users who have sent an identical message to pairs of people that know each other. In turn, it builds an XML tree in which the top node groups each distinct sender, and append their name as a child node. Then, under each sender, it groups messages sent to multiple people, appends each recipient under a distinct child labeled "to" and finally appends the message itself (once).

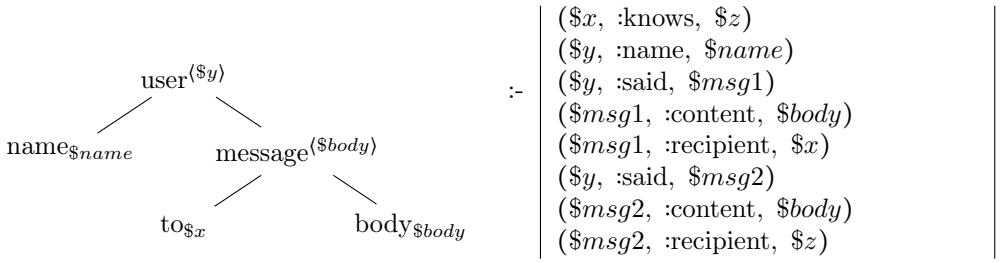


Figure 11: Query Q_2, *lowering* social network information from RDF to XML

These types of queries are sometimes qualified as lowering and lifting queries, as they respectively lower data from RDF to XML or lift it from XML to RDF. These two queries happen to be composable on one another. Composing Q_2 over Q_1 produces a union of two queries, whose respective bodies feature eight self-joins over copies of Q_1 (as there are eight atoms in the normalized body of Q_2, thus eight variable-renamed copies of Q_1 are used in the matching phase). These query bodies can be minimized. For readability, Figure 12 shows the composition of Q_2 over Q_1, where each sub-query of the union has been minimized and features only to three self-joins.

Figure 13 depicts the composition of Q_1 over Q_2. In this case, at the end of the matching phase, only one entry of the Cartesian products of substitutions yields a consistent substitutions after been unioned. This explains why the composed query is not a union. The head of the composed query features multiple calls to Skolem functions. These are the calls that would be used to assign URIs to XML node in the result of Q_1, e.g., if the view had to be materialized.

XR integration. The next scenario shows how one can query interconnected XR, in the absence of such data, for example, on legacy XML and RDF data. Figure 14 shows a view that combines data from an RDF data instance such as DBpedia and XML data instance such as some OECD report, gathering data about countries productivity and GDP per year. The query uses the country code to join data from both sub-instances, and constructs an XML tree, that is similar to the one filtered, up to XML node

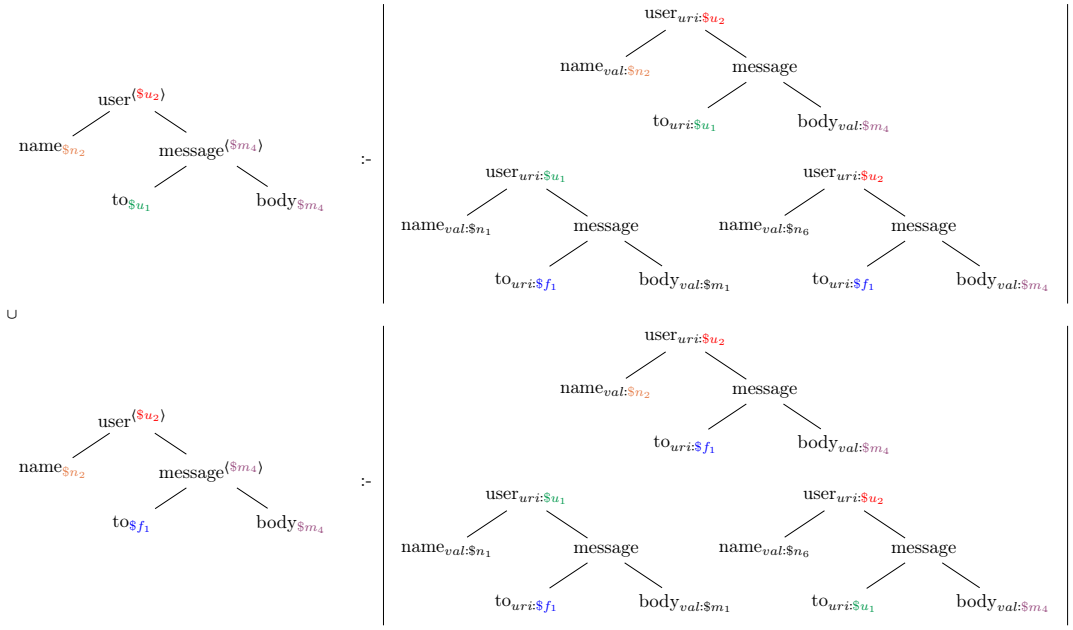


Figure 12: Composition of Q_2 over Q_1

$(f_{user}(\$y),\ \text{:name},\ \$name)$
$(f_{user}(\$y),\ \text{:knows},\ f_{to}(\$y, \$body, \$x))$
$(f_{to}(\$y, \$body, \$x),\ \text{:knows},\ f_{user}(\$y))$
$(f_{user}(\$y),\ \text{:said},\ f_{msg}(\$y, \$body))$
$(f_{msg}(\$y, \$body),\ \text{rdf:type},\ \text{:Message})$
$(f_{msg}(\$y, \$body),\ \text{:content},\ \$body)$
$(f_{msg}(\$y, \$body),\ \text{:recipient},\ f_{to}(\$y, \$body, \$x))$

:-

$(\$x,\ \text{:knows},\ \$z)$
$(\$y,\ \text{:name},\ \$name)$
$(\$y,\ \text{:said},\ \$msg1)$
$(\$msg1,\ \text{:content},\ \$body)$
$(\$msg1,\ \text{:recipient},\ \$x)$
$(\$y,\ \text{:said},\ \$msg2)$
$(\$msg2,\ \text{:content},\ \$body)$
$(\$msg2,\ \text{:recipient},\ \$z)$

Figure 13: Composition of Q_1 over Q_2

URIs. However, the RDF sub-instance produced by the query features new RDF triples that directly link countries to the XML nodes containing informations collected by the OECD.

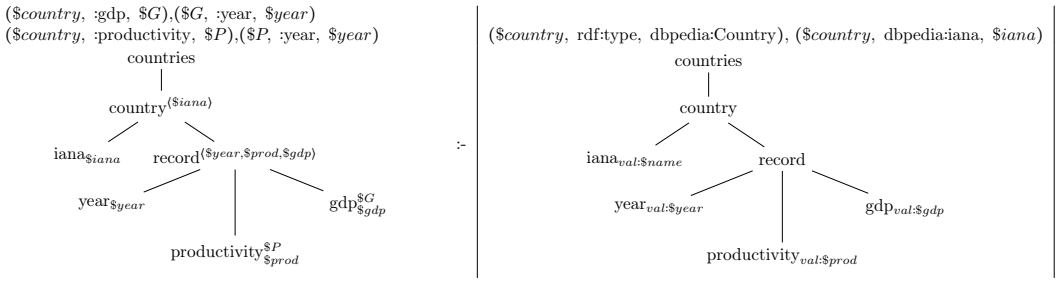


Figure 14: Query Q_3, integrating legacy XML and RDF into fresh XR

Query Q_4 on Figure 15 attempts to find the productivity and GDP of France in 2010, referring directly to its DBpedia URI.

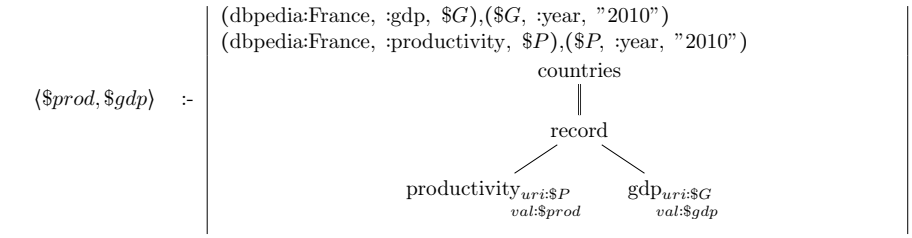


Figure 15: Query Q_4, retrieving information of a virtually integrated XR instance

Composing Query Q_4 over Query Q_3 yields the query depicted on Figure 16.

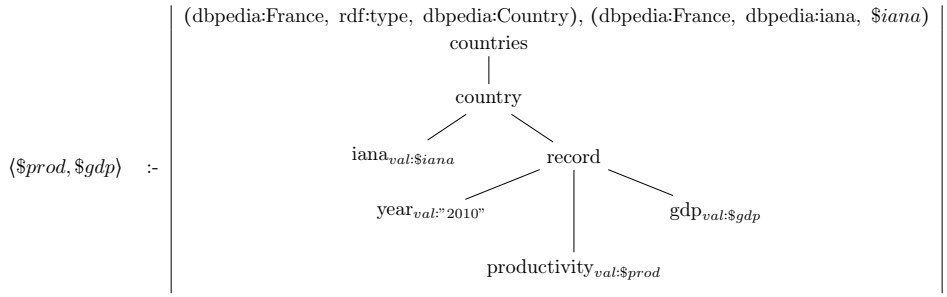


Figure 16: Composition of Q_4 over Q_3

6 Related Works

On the extended language and Skolem functions. Most query languages for semi-structured data such as XML and RDF provide some means to construct data in that same model. In XQuery, and some of its predecessors (e.g., XML-QL [2], XQL [13]), the nesting of XML element and attribute emerges naturally from the language's syntax. In the semi-strcutured data management platform Strudel [6], the StruQL query language relies on Skolem functions to assign internal IDs to node and merge them accordingly. In this language however, calls to Skolem function are explicit, which requires a rather high level of expertise from the user. Moreover, one may easily end up with graph-structured (rather than tree-structured) data in the query result. Strudel relies on a graph data model, however we defined an XML sub-instance as a set of trees. This is avoided this in XRQ, the assignment of Skolem function for node constructors is transparent to the user and the construction of input tuple to those functions is done based on the grouping labels and the structure of trees appearing in queries heads.

SPARQL also allows creating RDF in the result of its queries with the CONSTRUCT and DESCRIBE clauses. The language permits blank nodes in the head, but due to problems caused by (correct or assumed) semantics of blank nodes, the use of Skolem function in place of blank nodes has recently been suggested [17]. Explicit use of Skolem functions in SPARQL has also been proposed in the RDF view definition language vSPARQL [21].

Resorting to Skolem functions for data creation is not restricted to modern semi-structured data models. The idea was introduced for object-relational models in [16], and further studied with the ILOG query language [12], where Skolem functions are used to assign OIDs to "invented" objects.

Query composition in XML. In terms of systems, we have already mentioned Strudel [6], an integration and publishing platform, where a data instance is defined through a set of views over heterogenous data source. Queries over these views are used to produce web sites in a declarative manner. The SilkRoute [8, 7] and Xperanto/XTables [3, 9] systems were designed to let one query relational table with XQuery. Tables were mapped to XML views and a composition algorithm translated queries to SQL in the former case, or a physical execution plan in the latter.

Composition vs. query rewriting using views In the relational data management literature [1], composition is defined as follows: given a *program* P with a final rule S, i.e., a set of rules whose heads define intensional predicates, and whose bodies are made of extensional and intensional predicates, a composition is a query q, s.t. for any database I, $q(I) = [P(I)]S$. For non-recursive programs, the problem is quite straightforward and amounts to expanding the body of the final rule S, i.e., replacing intensional predicates with the body of the corresponding rule, until it contains nothing but extensional predicates.

In XR, both the head and body of queries are made of triple and tree patterns or constructs, preventing the mere expansions that relations allow. The closest parent of our composition is "Query decomposition & algebraic optimization" (QD&AO) algorithm presented in [18], in the context the TSIMMIS project, a mediator for OEM semi-structured data, where views are defined in the "Mediator Specification Language" (MSL) language over heterogenous data sources and query evaluated of these views. The algorithm first applies a normalization step, then matching of query body and view head, and finally applied to the rewritings. In [26, 19, 22], author also give an algorithm for ansering queries using views in the "Tree Specification Language" (TSL).

Our composition algorithm also has strong connections with the problem of answering queries using views, where solutions require to match atom of the query body with that of the body of views. Next, we discuss similarities between our approach and three well-known algorithms to find maximally contained rewriting using materialized views.

The inverse rule algorithm [4] guarantees to find, for a given a (possibly recursive) datalog program P and a set of views V, a maximally contained program P' whose rules body solely relies on V. The central idea of the algorithm is to turn V into a set of rules V', where for each view $v \in V$ and each atom $a \in v$, there is a rule in V' featuring a as head and the head of v as body. Skolem functions are used to replace non-distinguished variables revealed in the head of such rules. P' is obtained by removing from P all rules using extensional predicates that do not appear in any view of V, and adding all rules of V' whose head match with a body atom of P. The final step of the algorithm consists in removing rules that rely on Skolem functions.

The bucket algorithm [15] and one of its direct offsprings, the MiniCon algorithm [20], also share resemblance with our approach. The first phase of these algorithms is essentially the same as the

matching phase, except that in those cases, body atoms of the query are matched with body atoms of the views. A bucket of matching view atoms, similarly to our substitutions sets Σ_i, is created for each query atom. Then, the algorithms diverge in the way these buckets are combined into conjunctive queries that *cover* the whole query. The bucket algorithm considers the whole Cartesian product of the buckets, while the MiniCon algorithm performs additional checks to reduced the number of combinations to explore. One of these optimizations consists in discarding candidates whose body may match some subgoal of the query, but whose head does not features the variables that would allow to join its results with that of views matching other subgoals of the query. Although our algorithm iterates on the Cartesian product of buckets, as in MiniCon, many entries can quickly be discarded by checking the consistency of $\sigma_\cup$.

This work also shares strong connections with an algorithm proposed by Le et al. [14], to rewrite queries on SPARQL views. In particular, the algorithm also comprises two phases, corresponding to our matching and building phases. An external algorithm is proposed to prune unsatisfiable sub-queries in the resulting union. Similarly, in [5, 23], authors tackle the problem of rewriting XPath queries on XPath views. The use a mixed finite state automata to represent queries and avoid the exponential blow-up of rewritings.

7 Conclusion

In the report, we presented an extension of the XRQ query language introduced in [11], where queries are augmented with data constructors, and produce XR data as results. We detailed the language syntax and semantics, and discussed the notions of containment and equivalence.

Then, we focused on the problem of query composition for which we provide a sound and complete algorithm. The closure under composition of the language directly follow from the algorithms properties.

This extension to the language confers to the XR platform important capabilities that put it on par with data management and data integration integration systems that have been devised for XML and RDF. It's cross-model nature however sets it apart from existing systems.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql/, 1998.

[3] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *Proceedings of the International Conference on Very Large Data Bases*, pages 646–648, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In A. O. Mendelzon and Z. M. Özsoyoglu, editors, *PODS*, pages 109–116. ACM Press, 1997.

[5] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 587–598, New York, NY, USA, 2004. ACM.

[6] M. Fernández, D. Florescu, A. Levy, and D. Suciu. Declarative specification of Web sites with STRUDEL. *The VLDB Journal*, 9(1):38–55, Mar. 2000.

[7] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, Dec. 2002.

[8] M. Fernández, W.-C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1-6):723–745, June 2000.

[9] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging relational technology and XML. *IBM Syst. J.*, 41(4):616–641, Oct. 2002.

[10] F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Growing Triples on Trees: an XML-RDF hybrid model for annotated documents. In *The VLDB Journal - Special Issue on Structured, Social and Crowd-sourced Data on the Web*.

[11] F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Growing triples on trees: an XML-RDF hybrid model for annotated documents. *The VLDB Journal*, pages 1–25, 2013.

[12] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 455–468. Morgan Kaufmann, 1990.

[13] H. Ishikawa, K. Kubota, Y. Kanemasa, and Y. Noguchi. The design of a query language for XML data. In *Proceedings of the International Workshop on Database and Expert Systems Applications*, pages 919–922, 1999.

[14] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *Proceedings of the International Conference on World Wide Web*, pages 655–664, New York, NY, USA, 2011. ACM.

[15] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB*, pages 251–262. Morgan Kaufmann, 1996.

[16] D. Maier. *A logic for objects*. Oregon Graduate Center, 1986.

[17] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In *The Semantic Web–ISWC 2011*, pages 421–437. Springer, 2011.

[18] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *VLDB*, volume 96, pages 413–424. Citeseer, 1996.

[19] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. *SIGMOD Record*, 28(2):455–466, June 1999.

[20] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.

[21] M. Shaw, L. T. Detwiler, N. Noy, J. Brinkley, and D. Suciu. vSPARQL: A view definition language for the semantic web. *Journal of Biomedical Informatics*, 44(1):102–117, Feb. 2011.

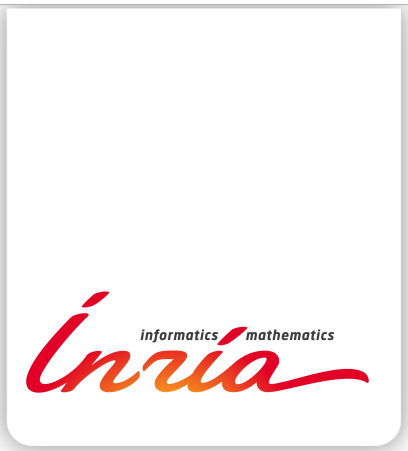
[22] Vasilis Vassalos and Yannis Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *The Journal of Logic Programming*, 43(1):75 – 122, 2000.

[23] Wenfei Fan, F. Geerts, Xibei Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *IEEE International Conference on Data Engineering*, pages 666 –675, April 2007.

[24] RDF Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[25] URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/, 2001.

[26] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data (extended version). Technical Report 1998-10, Stanford InfoLab, 1998.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399