# Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents

François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, Stamatis Zampetakis

**Noname manuscript No.**
(will be inserted by the editor)

# Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents

**François Goasdoué · Konstantinos Karanasos · Yannis Katsis · Julien Leblay · Ioana Manolescu · Stamatis Zampetakis**

Received: date / Accepted: date

**Abstract** Since the beginning of the Semantic Web initiative, significant efforts have been invested in finding efficient ways to publish, store and query metadata on the Web. RDF and SPARQL have become the standard data model and query language, respectively, to describe resources on the Web. Large amounts of RDF data are now available either as stand-alone datasets or as metadata over semi-structured (typically XML) documents. The ability to apply RDF annotations over XML data emphasizes the need to represent and query data and metadata simultaneously.

We propose XR, a novel hybrid data model capturing the structural aspects of XML data and the semantics of RDF, also enabling us to reason about XML data. Our model is general enough to describe pure XML or RDF datasets, as well as RDF-annotated XML data, where any XML node can act as a resource. This data model comes with the XRQ query language that combines features of both XQuery and SPARQL. To demonstrate the feasibility of this hybrid XML-RDF data management setting, and to validate its interest, we have developed an XR platform on top of well-known data management systems for XML and RDF. In particular, the platform features several XRQ query processing algorithms, whose performance is experimentally compared.

F. Goasdoué, J. Leblay, I. Manolescu, S. Zampetakis
INRIA Saclay & Univ. Paris Sud, Orsay, France
E-mail: firstname.lastname@inria.fr

K. Karanasos
Almaden Research Center, San Jose, CA
E-mail: kkarana@us.ibm.com
This work was done while the author was at Inria Saclay.

Y. Katsis
UC San Diego, San Diego, CA
E-mail: ikatsis@cs.ucsd.edu
This work was done while the author was at Inria Saclay and ENS Cachan.

## 1 Introduction

The XML format [1] is by now universally used to represent structured documents. Initially employed primarily for Web pages, it soon became the standard for text documents produced by major office suites, and the go-to solution for most structured documents at large, be it bills, bank account data, contracts, content produced and shared in the workspace, social network and blog data, etc.

In parallel, W3C's Resource Description Framework (RDF, in short) [2] is becoming the *de facto* standard for describing data rich in semantics. Its provisions (embodied in the RDF Schema language [3]) for defining semantic relationships (e.g., subsumption relations or typing), which are used for reasoning over the data and deriving new knowledge, make it an ideal candidate for representing such data. RDF adoption has recently registered an additional boost due to the Linked Open Data (LOD)[1] movement. Under the LOD vision, users independently author and share information, which they can then link to already existing data published by others. Linking the data is facilitated by assigning each data item a unique identifier, a.k.a. URI [4], which is one of the cornerstones of the RDF data model. Representative examples are government-issued open data portals, such as `http://data.gov` in USA, `http://data.gov.uk` in the UK and `http://data.gouv.fr` in France. Another famous source of RDF linked open data is DBPedia [5], a corpus of facts extracted from Wikipedia.

While XML and RDF are primarily aimed at different types of data (the first at structurally rich data and the second at semantically rich data), there are emerging applications at their juncture that need formal models and semantics. The main objective of this work is to show that combining XML and RDF yields more than the juxtaposition thereof. Recent initiatives such as the Open Annotation Collaboration[2] show

[1] http://linkeddata.org
[2] http://openannotation.org

that using RDF to compensate for the lack of semantics in XML is a promising research direction. Although, in theory, one could simply convert RDF into XML or vice-versa, the tremendous amount of data available on the Web and the frequency at which it is updated plead for efficient techniques for managing this data in its native formats. Moreover, converting RDF to XML (or XML to RDF) would lose the opportunity to exploit existing research and systems on efficiently storing and querying RDF (resp., XML) data. Below we present three scenarios that highlight the interest of combining XML with RDF data.

**Scenario 1: semi-automated fact-checker**

The internet has reshaped journalism in important ways, one of the most important being the instant dissemination capabilities of the Web. Moreover, journalists have suddenly had to compete with bloggers, activists and other concerned citizens, establishing themselves as alternative sources of information, and reaching out, collectively, to a far wider reality on the ground than a news agency (let alone a single journalist) could hope to have access to. This has led to the emergence of new professionals, called data-journalists, and online fact-checkers. These specialists are trained to examine and aggregate data from many sources ("official" or not, such as Data.gov or WikiLeaks[3]) and use online services (such as Twitter[4] or Google Maps[5]) to integrate and corroborate facts found online. Journalists have become data publishers themselves as witnessed in sites such as The Guardian[6], FactCheck[7], and Politifact[8]. However, as skilful as these professionals may be, their work is still very manual as demonstrated by Storyful[9] founder in a recent presentation[10] and, as of today, they lack powerful tools for analyzing, consuming and producing data.

As a concrete example, consider an election campaign, where candidate :Joe publishes on his Web site transcripts of his speeches, expressing his opinions on the situation in :Turkey or :Japan, or the local economy, citing a :MonthlyUnemploymentRate for :July2012 as being "8%" [11]. Using an officially-issued database such as `http://data.gov`, one can automatically check whether the cited number is correct. Moreover, archiving the candidate's speeches allows finding, e.g, "the earliest and latest date at which his discourses mentioned an Asian country", or (if the candidate's official agenda is also added to the analysis) "for each foreign country, the visits the candidate received from or made to that country, and the mentions he subsequently made of the country in his speeches". Although such queries are too ambiguous to yield any valuable results if posed in natural language, with proper knowledge of the datasets in hand and their semantics, an expert should be able to express them in a structured language through a single query or some composition of queries.

**Scenario 2: focused Web warehouse**

The ACME company wants to keep up with the image of its products as reflected by content published on the Web (on news sites, blogs, social networks etc.). To this end, it sets up a set of specialized feeders, one from each source of content (e.g., one for crawling open Web content, others as subscriptions to specific Twitter hashtags etc.), and archives the XML results brought by these feeders in a database. The documents are then parsed, analyzed, and compared with ACME's RDF knowledge database containing brands, models, clients, sales, information about ongoing marketing campaigns, etc. The warehoused XML content is thus connected to the objects and contents of the knowledge base, and can be subsequently exploited by asking, e.g., for "the authors and affiliation (if any) of all blog posts from July 2012, mentioning ACME :Prod1 products (regardless of their model)". This query involves reasoning through an RDF Schema to understand that :Prod1v1 and :Prod1v2 are all versions of ACME's :Prod1, querying the XML warehouse for blog posts mentioning :Prod1, :Prod1v1 or :Prod1v2, and returning the desired blog author affiliation. Observe that if the authors' affiliations (e.g., organizations they work for) are also recognized in the RDF database, one may refine the query result by further exploring their links in this database, finding, for instance, in which country each organization is located or how many employees it has.

**Scenario 3: patient records**

Another use case for annotated documents is in the area of electronic patient records (EPR). French hospitals seeking more interoperability among their respective patient files (partly paper-based, and partly electronic), set up systems where paper-based records are scanned and then subjected to text recognition. Subsequently, they apply natural language processing techniques on these electronic files, annotate them with entities (diseases, symptoms etc.) recognized from a domain ontology, and index them accordingly. Physicians can then more easily find "admission dates of female patients with heart problems" or "the list of drugs targeting eating disorders that have been administered to patients diagnosed with diabetes". These queries typically touch upon data that may exist in different models.

[3] http://wikileak.org
[4] http://twitter.com
[5] http://maps.google.com
[6] http://guardian.co.uk/data
[7] http://www.factcheck.org
[8] http://www.politifact.org
[9] http://storyful.com
[10] http://on.ted.com/MarkhamNolan
[11] Here and subsequently in this paper, we make the convention that strings starting with : are URIs. Formally, URIs consist of two parts: a namespace, and a local name [4], separated by the : symbol. A URI without a specified namespace is of the form :LocalName, and is interpreted to refer to a default namespace.

This work aims at enabling such scenarios, by proposing a unified model allowing the combination of XML data with RDF data into a single instance. We have designed the model and the corresponding query language and implemented a system for storing and querying instances of the proposed model. Moreover, we showcase optimizations that are possible when XML and RDF are combined in the same instance.

A separate class of applications that calls for a combination of XML and RDF, e.g., [6], focuses on extracting data and semantics (which can be encoded in RDF) from structured XML documents. Such applications typically adhere to the following workflow: the text (and possibly the structure) of an XML document are analyzed with the help of natural language processing tools, named entities are recognized and extracted and phrase patterns are matched to convert the information in an XML document to RDF facts. For instance, the text "Einstein was born in Ulm" would be translated into an RDF statement of the form (:AlbertEinstein, :birthPlace, :Ulm).

This work makes the following contributions:

**Data Model for Annotated XML Documents.** Our data model naturally allows the representation of XML data, RDF data and the union thereof, but more importantly it also allows for instances where XML and RDF are interconnected (e.g., where an RDF triple refers to an XML node).

**Query Language.** To allow existing users of XML and RDF platforms to easily transition to our combined platform, we designed a query language that not only allows querying inter-connected XML and RDF instances, but does so by staying close to the standard query languages employed for each of the data models in isolation.

**Implementation & Optimizations.** As a first cut, we implemented a system for annotated XML documents by leveraging existing XML and RDF engines. However, as we will explain, there are multiple ways in which a query over a combined XML and RDF instance could be decomposed into separate queries that are shipped to the XML and RDF engine. In this work, we explore this space of possible query evaluation strategies and present optimizations to speed up query processing.

**Experimental Results.** Our experiments highlight classes of query evaluation strategies that are very inefficient and some that provide better performance and scale linearly on datasets of an overall size of 17 GB, intelligently exploiting pre-existing XML and RDF engines. We study the impact of our proposed optimizations and identify the classes of problems where they have the biggest impact. It is worth noting that among similar works focusing on the combined querying of XML and RDF, very few provide experimental results, and those that do [7] present query evaluation strategies that do not scale beyond 100 MB. Thus, our experiments validate the interest of our techniques for large-scale querying of annotated documents.

The paper is structured as follows. Sections 2 and 3 describe the data model and query language, respectively. We discuss query evaluation strategies in Section 4, outline our implemented platform in Section 5, and present the experimental results in Section 6. Section 7 describes related work and Section 8 concludes the paper.

## 2 The XR Data Model

To represent annotated documents, we introduce the *XR data model*. In keeping with the widely accepted standards for representing semi-structured data (i.e., XML) and semantic relationships (i.e., RDF), an instance of the XR data model comprises two sub-instances: an XML sub-instance, consisting of a set of XML trees, and an RDF sub-instance, consisting of a set of RDF triples. The connection between the two sub-instances is achieved by assigning to each XML node a unique Uniform Resource Identifier (URI), which can then be referred to from an RDF triple, as we will explain below.

Next, we formally define XR sub-instances. We rely on a set $\mathcal{U}$ of URIs as defined in [4], and a subset $\mathcal{I} \subseteq \mathcal{U}$ of *document URIs* acting as document identifiers. We denote by $\mathcal{L}$ the set of literals [8] (which for simplicity can be seen as the set of all strings). $\mathcal{N}$ is the set of possible XML element and attribute names, to which we add the empty name $\epsilon$. Finally, $\mathcal{B}$ is a set of blank nodes (accounting for unknown literals or URIs, as we will explain later on). An XML tree is defined as usual:

**Definition 1 (XML Tree)** An *XML tree* is a finite, unranked, ordered, labeled tree $T = (N, E)$ with nodes $N$ and edges $E$, where each node $n \in N$ is assigned a label $\lambda(n) \in \mathcal{N}$ and a type $\tau(n) \in \{document, attribute, element, text\}$. An attribute node must be the child of an element node, it has a value belonging to $\mathcal{L}$ and it does not have any children. A text node can only appear as a leaf. Finally, an XML tree can have at most one document node. The document node can only appear as the root of the tree, has exactly one child and has the empty name $\epsilon$.

Most frequently, we are concerned with trees that are also documents, i.e., those rooted in document nodes. However, we may also consider trees rooted at simple XML elements, for instance, when XML trees are passed from the output of one query to the input of another, without being permanently stored within a document. A set of XML trees forms an XML instance:

**Definition 2 (XML Instance)** An *XML instance* $I_X$ is a finite set of XML trees.

We assume available a function assigning a unique URI to each node in an XML instance. Notably, the URIs assigned to document nodes correspond to the aforementioned *document URIs*. The URI assignment function is crucial for interconnecting the XML and RDF sub-instances, since the URIs assigned to the nodes allow the RDF sub-instance to refer to nodes of the XML sub-instance. While discussing our system implementation in Section 5, we present such a URI assignment function that can be used in practice. However, for the purpose of the definitions, it suffices to consider any URI assignment function acting like a Skolem function, i.e., returning a new ("fresh") value every time it is called for the first time with a given input, and consistently returning that value to any subsequent call with the same input.

The RDF sub-instance is defined as a set of triples, which can among others refer to the URIs of XML nodes:

**Definition 3 (RDF Instance)** 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 $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$.

Following the common nomenclature, the components of a triple $(s, p, o)$ are referred to from left to right as its *subject*, *property* and *object*, respectively.

As defined above, the subject or the object of the triple can be bound to a so-called *blank node*. Blank nodes are used in RDF [2] to denote *unknown URIs or literals*, similarly to *labelled nulls* in the database literature [9]. For instance, one can use a blank node $b_1$ in the triple $(b_1, country, "France")$ to state that the *country* of $b_1$ is *France*, without using a concrete URI. Blank nodes can be repeated in an RDF instance, thus allowing multiple triples to refer to the same unknown URI or literal. For example, a second triple $(b_1, city, "Paris")$ could specify that the *city* of the same $b_1$ is *Paris*. Finally, multiple blank nodes can co-exist in a data set, thus allowing the representation of several unknown URIs or literals. For example, one may also state that the *country* of some other unknown URI $b_2$ is *Japan*, while its *population* is an unspecified literal $b_3$.

Furthermore, RDF does not only model explicit triples, but also implicit (a.k.a. *entailed*) triples. The latter can be derived from the former based on a set of *entailment rules*. More details on this process, known as *RDF entailment*, can be found in [10]. For the purposes of our discussion though, it suffices to be aware of the following: Given an RDF instance $I_R$, its semantics is the RDF instance $I_R^\infty$, called the *saturation* (or *closure*) of $I_R$, consisting of $I_R$ plus all the implicit triples derived from $I_R$ through RDF entailment. RDF entailment is central to RDF query answering, and thus to XR (as discussed in Section 3.2), since we need to take into account the implicit answers in order to guarantee the completeness of query answers. The interconnection between XML and RDF opens the way to cross-model inference, by allowing one to query *intensional* XML data, derived by combining extensional XR data with entailment rules. We believe this is a novel perspective on XML data management that deserves to be further explored in future works.

We can now define an XR instance as follows:

**Definition 4 (XR Instance)** 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.

It is important to note that the XML and the RDF sub-instances are defined over the same set $\mathcal{U}$ of URIs, thus allowing RDF triples to annotate nodes of XML trees. The following example illustrates such an interconnected XR instance.

**Example.** Figure 1 shows a sample XR instance corresponding to a political news scenario, which we will use hereafter as our running example. The RDF sub-instance is shown on the top part of the figure, while the XML sub-instance is shown at the bottom. The instance consists of three XML trees linked through RDF annotations. The first XML tree includes a post on a blog concerning a campaigning politician named :Charlie. The second XML tree is :Charlie's micro-blogging site, whereas the third is an article in an online newspaper. XML node URIs are shown as subscripts next to each node. The dashed edges in the XML tree denote some levels of XML hierarchy omitted for simplicity.

URIs are used to allow the RDF triples to annotate the XML trees. For instance, the first two triples, coming from a social site, specify that :Alice worked with :Bob in the past and that :Bob follows :Charlie's micro-blog. The next three triples state that :Charlie posted an entry on his blog at 12pm on Sept. 5, 2012. Note that a blank node (denoted _:x) is used here as a means of gathering facts around a single concept; we follow the usual convention of denoting blank nodes by _:-prefixed names.

The triple (_:x, owl:sameAs, #205) states that the blank node _:x and the XML node #205 of the blog stream are the same (the owl:sameAs property is frequently used for encoding such statements in RDF [11]). The RDF sub-instance further states that :Alice posted the blog entry found on the node #106 of the leftmost document, and that :Bob is the author of the entry #305 on the newspaper page. The two following triples specify that :Alice's blog post (#106) refers to :Bob's article for further information, using the :about property. Similarly, :Bob's article links to :Charlie's post, as one source of his report. The RDF instance also states that :Charlie's attendance of Congress sessions is rather low.

Finally, the triples in gray do not appear explicitly in the instance. They can be inferred from an RDF Schema (RDFS) characterizing this application (the RDFS is not shown in the figure), and stating that:

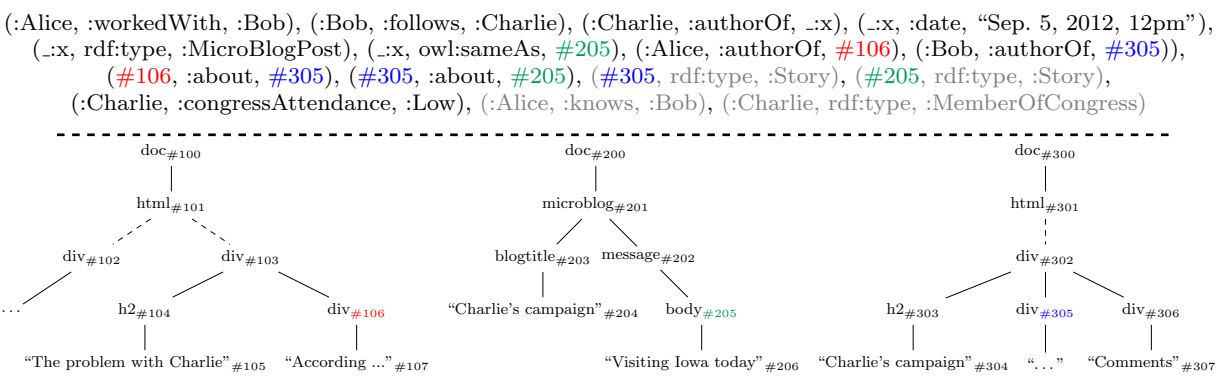


Fig. 1: XR instance representing annotated documents.

$(i)$ if a resource $A$ is *about* another resource $B$, then $B$ is a *story*, $(ii)$ if a person $A$ *worked with* a person $B$, then necessarily $A$ *knows* $B$, and $(iii)$ someone whose :congressAttendance property is defined is a member of the Congress.

## 3 The XRQ Query Language

XRQ allows querying an XR instance w.r.t. both its structure (described in the XML sub-instance) and its semantic annotations (modelled in the RDF sub-instance). We introduce XRQ's constructs in Section 3.1 and then we give its semantics in Section 3.2.

### 3.1 XRQ syntax

XRQ allows querying an XR instance based on commonly used primitives: XML tree pattern queries, introduced, e.g., in [12], and the Basic Graph Pattern queries (or BGPs, in short) for RDF [13]. Tree patterns express structural constraints on the expected trees in the XML sub-instance, while BGPs (a fragment of SPARQL) allow constraining the expected triples of the RDF sub-instance.

**Definition 5 (Tree Pattern)** 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 *[val=c]* for $c \in \mathcal{L}$, denoting a selection on the *val* variable, i.e., it must be bound to $c$.

A tree pattern may also have at most one 'special' document node. This node can only appear as the root of the tree, has exactly one child, and has a *uri* variable constrained by an equality predicate of the form *[uri=u]* for $u \in \mathcal{I}$, denoting that the tree pattern must be evaluated against the XML document of URI $u$.

Such variable-annotated patterns have been previously used, e.g., in [14,15] to represent XML queries and/or materialized views. The variables attached to nodes serve three purposes: $(i)$ to denote data items that are returned by the query (in the style of distinguished variables in conjunctive queries), $(ii)$ to express selections on the document to query or on node values, and $(iii)$ to express joins between tree (or triple) patterns. The variable type specifies the exact information item from an XML node, to which the variable will be bound. When a node $n_t$ of a tree pattern is matched against a node $n_d$ of an XML tree, the variables attached to the node $n_t$ will be bound as follows, according to the variable's type. First, a *uri* variable is bound to the URI of $n_d$. If $n_d$ is a document or element node, a *val* variable is bound to the concatenation of all text descendants of $n_d$; if $n_d$ is an attribute node, a *val* variable is bound to the attribute value; if $n_d$ is a text node, a *val* variable is bound to $n_d$'s text value. Finally, a *cont* variable is bound to the serialization of the subtree rooted at $n_d$. The semantics of *val* variables are copied from the XPath (and XQuery) specification. Indeed, an XPath snippet of the form `$x=''Paris''`, where `$x` is bound to some XML element, is interpreted as: check if the concatenation of all text descendants of that element equals "Paris". We represent such predicates by annotating a tree pattern node with [*val*="Paris"]. Similarly, a comparison of the form `where $x=$y` is interpreted as: the value of `$x` (as we defined it above) is equal to the value of `$y`. Our queries also allow expressing such comparisons, as we will explain later on.

**Example.** The bottom part of Figure 2 shows two tree patterns for our running example. As usual, single (double) edges correspond to parent-child (ancestor-descendant, resp.) relationships. For instance, the tree pattern on the left looks for a *message* node with a descendant *body* node. For each match of the pattern against the tree, $\$A$ will be bound to the URI of the matched *body* node, while $\$CA$ will be bound to the serialization of the node itself and its entire subtree.

A Basic Graph Pattern query is a conjunction of triple patterns.

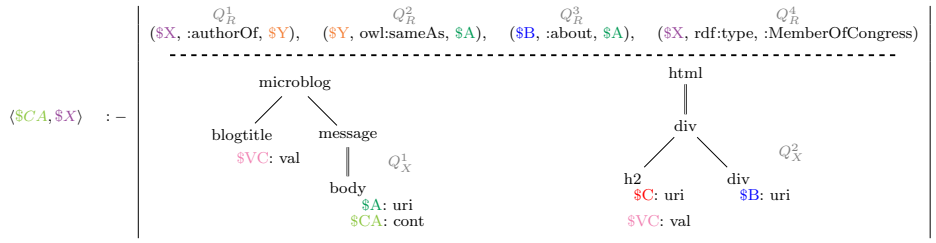


Fig. 2: Sample XRQ query

**Definition 6 (Triple Pattern)** 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.

**Example.** The top part of Figure 2 depicts four triple patterns. For instance, the left-most triple pattern finds all pairs of resources connected via the property :authorOf.

By combining tree and triple patterns and endowing them with a set of projected (head) variables, we obtain an XRQ query:

**Definition 7 (XRQ Query)** An XRQ query consists of a *head* and a *body*. The body is a set $Q_X$ of tree and a set $Q_R$ of 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 $Q = (h, Q_X, Q_R)$.

Note that by using variables in multiple places within the query, one can express joins. In general, three types of joins are possible: $(i)$ between tree patterns; $(ii)$ between triple patterns; $(iii)$ between tree patterns and triple patterns. In particular, the latter type of joins allow correlating structural and semantic constraints within queries. The following example illustrates the expressivity of XRQ.

**Example.** Figure 2 shows an XRQ query, whose body (shown on the right) comprises four triple patterns (shown on the top) and two tree patterns (shown at the bottom). It asks for all authors of some resource (first triple pattern) that is known to be the same (second triple pattern) as the body of a message from the micro-blog stream (first tree pattern). In turn, the query filters `html` pages containing a `div` node, with a header (`h2` node) equal to the title of the micro-blog's stream, and retrieves the `div` node containing the article body (second tree pattern). The selected micro-blog posts must be referred by the article (third triple pattern) and their authors must be congress members.

To sum up, the query returns the member of the congress who authored micro-blog posts referred by articles of the same title, as well as the posts contents. Note the use of variables for expressing joins. Three types of joins are illustrated in Figure 2: between two tree patterns (through variable $\$VC$), between two triple patterns (through variables $\$A$, $\$X$ and $\$Y$) and between a tree pattern and a triple pattern (through variables $\$A$ and $\$B$).

### 3.2 XRQ semantics

We now define the semantics of XRQ. To this end, we first define the notion of *matches* and *variable bindings* for each of its components (i.e., tree patterns and triple patterns).

A match of a tree pattern against an XML instance is defined as usual through tree embeddings [12]:

**Definition 8 (Match of a tree pattern against an XML instance)** Let $Q$ be a tree pattern and $I_X$ an XML instance. A *match* of $Q$ against $I_X$ is a mapping $\phi$ 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 $n$, 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, $\phi$ satisfies the equality predicates as follows: $(i)$ if $n$ is a document node constrained with the predicate *[uri=u]*, then $\phi(n)$ is the document node of the XML document whose URI is $u$ and $(ii)$ if $n$ is any node constrained with the predicate *[val=c]*, then the value of $\phi(n)$ equals to $c$.

A match of a tree pattern $Q$ against an XML instance $I_X$ defines the mapping of nodes of $Q$ to nodes of $I_X$. However, recall that a tree pattern, apart from nodes, contains also variables for expressing selections on values or joins, which have to be bound to objects. This mapping of such variables to objects, referred to as *variable binding* is formally defined below:

**Definition 9 (Variable binding of a tree pattern against an XML instance)** Let $\phi$ 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 $\phi$ 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 $\phi(n)$.

As explained above, a variable binding $f$ of a tree pattern $Q$ against $I_X$ is associated with a match $\phi$ of $Q$ against $I_X$. For simplicity however, in the following we will assume the existence of a match and refer to $f$ simply as a variable binding of $Q$ against $I_X$.

Similarly, we also define matches and variable bindings for triple patterns:

**Definition 10 (Match of a triple pattern against an RDF instance)** Let $Q$ be a triple pattern $(s, p, o)$, $I_R$ an RDF instance and $I_R^\infty$ 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 $\phi(ul) = ul$ ($\phi$ maps any URI or literal only to itself).

It is important to note that in accordance with the RDF semantics as specified by the W3C, a triple pattern is matched not against an RDF instance $I_R$, but against the saturation of $I_R$, denoted $I_R^\infty$. As defined in Section 2, $I_R^\infty$ contains in addition to the explicit triples of $I_R$, a set of implicit triples.

We recall the notion of *restriction of a function to a subset of its domain*. Let $f$ be a function over a set $A$. The restriction of $f$ to a subdomain $A' \subseteq A$, denoted by $f|_{A'}$, is a function $f'$ over $A'$, s.t. $f'(x) = f(x), \forall x \in A'$. Based on this, we can define the variable binding of a triple pattern as follows:

**Definition 11 (Variable binding of a triple pattern against an RDF instance)** Let $\phi$ 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 $\phi$ is the function $\phi|_V$, where $V$ is the set of variables in $Q$.

We now provide the semantics of an XRQ query:

**Definition 12 (XRQ Semantics)** Let $Q$ be an XRQ query, $V$ 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 $V_P$, where $V_P \subseteq V$, $f|_{V_P}$ 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.

The definition combines in the intuitive fashion the notion of variable bindings in the RDF and XML sub-instances. When a variable is shared by a tree pattern and a triple pattern, the XRQ semantics ensures that it is bound to the same value (URI or literal) within the XML trees in $I_X$ and the RDF triples in $I_R$.

**Example.** Applying the XRQ query of Figure 2 to the XR instance of Figure 1 yields the result: ($CA= ⟨body⟩Visiting Iowa today⟨/body⟩, $X= Charlie).

Figure 3 shows the match found for each tree/triple pattern and the variable binding for the entire XRQ query.

**All joins allowed.** We stress that XRQ queries may feature all the types of joins one may encounter within a conjunctive RDF query or within an XML query, in addition to the aforementioned joins across the RDF and XML sub-instances (by sharing variables within tree and triple patterns of an XR query). It is worth noticing that join variables may be used in places having disjoint types. For instance, a variable may appear in the subject of a triple pattern (denoting a URI value) and as the *val* of a tree pattern's node (denoting a literal). Rather than considering type mismatches as errors in queries, we adopt the permissive approach of converting all variable bindings to literals and comparing their string representations.

**Cartesian products.** XRQ enables users to specify queries comprising Cartesian products. The latter occurs when some tree (or triple) pattern(s) do not share any variable with some other tree (or triple) pattern(s). At the same time, even when an XR query does not feature such Cartesian products, the subquery consisting only of its XML (or RDF) patterns may have Cartesian products. For instance, consider a query $Q$ consisting of two XML tree patterns $t_x$ and $t_y$ and a triple pattern $p_{x,y}$, such that a variable $\$X$ is shared by $t_x$ and $p_{x,y}$, a variable $\$Y$ is shared by $p_{x,y}$ and $t_y$, while $t_x$ and $t_y$ share no variable. In this case, the restriction of $Q$ to its XML sub-expression is $t_x \times t_y$. This aspect requires some extra care when evaluating XRQ queries, as we will discuss in the next section.

Finally, for the clarity of the query evaluation discussion in the next section, we also define the result of a set of tree patterns (resp., triple patterns) in isolation over an XML (resp., RDF) instance. Let $Q_X$ be a set of tree patterns and $I_X$ an XML instance. Then the result $Q_X$ over $I_X$, denoted $Q_X(I_X)$, intuitively corresponds to evaluating the set $Q_X$ of tree patterns against the XML instance $I_X$ and returning tuples of bindings for all variables appearing in $Q_X$. Formally, $Q_X(I_X)$ equals to $Q'(I')$, where $Q' = (h_X, Q_X, \emptyset)$ is an XRQ query that contains in its body only the set $Q_X$ of tree patterns and in its head $h_X$ all variables appearing in $Q_X$ and $I' = (I_X, \emptyset)$ is an XR instance having $I_X$ as its XML sub-instance and the empty instance as its RDF sub-instance. The result $Q_R(I_R)$ of a set of triple patterns $Q_R$ over an RDF instance $I_R$ can be defined in a similar way.

## 4 XRQ Query Evaluation

This section discusses evaluation strategies for XRQ queries. Since there are by now many platforms for

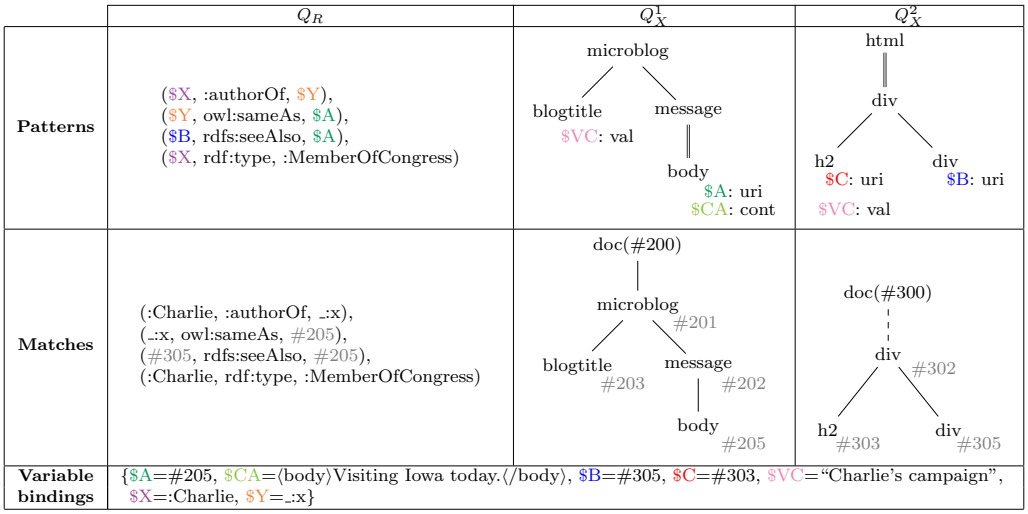


Fig. 3: Pattern matches and variable bindings of the query of Figure 2 on the XR instance of Figure 1.

handling XML and RDF separately, we aimed, whenever possible, to reuse the functionalities developed by such platforms and develop our XRQ processor as a layer on top. In the following, Section 4.1 introduces some preliminary notions which will help us present various query evaluation strategies. The remainder of the section presents the set of strategies of this study.

### 4.1 Preliminaries

We introduce a set of useful notions before presenting concrete query evaluation algorithms.

**XDM** stands for an XML data management platform, i.e., any XML data management system supporting tree pattern queries. Such queries can be expressed in XQuery, thus any XQuery engine falls into this category. We denote by $\mathsf{XEval}(Q, I)$ a function provided by the XDM, which returns the result of the XML query $Q$, consisting of a set of tree patterns possibly connected through joins, over the XML instance $I$.

**RDM** stands for an arbitrary RDF data management platform, i.e., any RDF data management system supporting at least (unions of) Basic Graph Pattern queries of SPARQL. Similarly, we denote by $\mathsf{REval}(Q, I)$ a function provided by the RDM, which computes the result of the RDF query $Q$ (that is, a set of triple patterns) over the RDF instance $I$.

**XURI** denotes URIs [4] of XML nodes. A deterministic method assigning an XURI to every node from a given document is termed a *labelling scheme*.

***$Q_X$* and *$Q_R$*** are the XML and RDF sub-queries, respectively, of a given XR query $Q$. Let $|Q_X|$ be the number of tree patterns in $Q_X$ and $|Q_R|$ the number of triple patterns in $Q_R$. We will denote the XML tree patterns in $Q$ by $Q_X^1$, $Q_X^2$, …, $Q_X^{|Q_X|}$ and, similarly, the triple patterns of $Q$ by $Q_R^1$, $Q_R^2$, …, $Q_R^{|Q_R|}$.

***$I_X$* and *$I_R$*** are the XML and RDF sub-instances, respectively, of an XR instance $I$.

**XURI hypotheses.** To facilitate the integration of any XML or RDF data management system in our XR platform, we should interface with the XDM/RDM at the level of standardized data declaration and data manipulation languages, such as XQuery and SPARQL, avoiding more specific assumptions regarding their implementation. One crucial issue that is specific to XR, however, is the support for XURIs within the XDM. While URIs are explicit in RDF data, in the XML data model [16], the closest notion to XURIs is that of node identity, which by default is implicit[12]. Most XDMs [18,19] (including recent ones [20]) use internal node IDs, which can easily be mapped to XURIs as soon as one gains access to the system internals. For the purpose of evaluating XR queries, we identify two important properties that an XDM may have (or, alternatively, hypotheses which may or may not hold about $\mathsf{XEval}$):

**XURI-out**: the outputs of $\mathsf{XEval}$ include the XURI of each XML node participating in this result.

**XURI-in**: given an XURI as input, $\mathsf{XEval}$ is capable of recognizing the (unique) XML node having this XURI. In other words, $\mathsf{XEval}$ can perform selections on XURI values, thus $\mathsf{XEval}$ understands the special semantics of XURIs.

These hypotheses are independent, i.e., an XDM may adhere to one, the other, none or both. Concrete ways of implementing them will be discussed in Section 5. The algorithms we present next have specific requirements in terms of XDM hypotheses, as we explain in each case.

[12] The W3C's xml:id recommendation [17] makes node identity explicit as an `xml:id` attribute, however, this has not been widely adopted. We explore the xml:id idea as one option in our implementation (see Section 5).

**What to delegate?** The XRQ processor delegates sub-queries for evaluation to the underlying XML, respectively, RDF engines. As explained in Section 3.2, if we decide, e.g., to send $Q_X$ as such to the XDM, this may introduce Cartesian products whose evaluation may be very inefficient.

An alternative consists in sending to the XDM the connected components of $Q_X$, if one considers $Q_X$ as an undirected graph where $(i)$ each tree pattern is a node; $(ii)$ there is an edge between two nodes if the corresponding tree patterns share some variable(s), in the spirit of the classical Query Graph Model [21]. Each connected component thus obtained is an XML query without Cartesian products, and is independently sent to XEval. Clearly, the symmetric discussion holds regarding $Q_R$.

Going one step further, one could question the distribution of join operations between XEval, REval and the XR platform itself. Intuitively, the native XDM engine should be able to best optimize the computation of tree pattern queries, that is, if $Q_X$ is of the form $tx_1 \bowtie_{\$X} tx_2$, we could send $Q_X$ as such to XEval. However, it turns out that XML queries with numerous value joins are still challenging for current XML query processors, as was initially noted in [22]. Therefore, it may be more efficient to send $tx_1$ and $tx_2$ to XEval, and join the results outside the XDM, within the XR platform.

To mitigate such issues, we adopt the following approach. Whenever $Q_X$ (respectively, $Q_R$) must be delegated to XEval (respectively, REval), a specific optimizer is invoked to determine which fragments of these queries to actually delegate; the remaining joins are handled in the XR platform. This decomposition is achieved based on $(i)$ heuristics (e.g., never push unnecessary Cartesian products), $(ii)$ query cardinality estimations, and $(iii)$ some empirical calibration tests to gauge how the XDM (respectively, RDM) performance compares with XR's own execution engine.

In the sequel, to simplify the presentation, we will just write $\mathsf{XEval}(\cdots)$, respectively $\mathsf{REval}(\cdots)$, to denote: find out the best way to decompose the respective query between the XDM (resp. RDM) and XR, and execute it according to that decomposition of work.

### 4.2 Independent executions

The simplest approach for evaluating an XRQ query consists in evaluating independently $Q_X$ and $Q_R$, and then evaluating any remaining joins (on XURIs or values) outside the XML and RDF engines. We denote this approach XML||RDF, for "independent evaluation of $Q_X$ and $Q_R$". To enable the join on XURIs outside the XDM, this approach requires hypothesis **XURI-out**. Moreover, to the extent that XEval and REval can run in parallel, this method has a good potential for parallelization. Algorithm 1 outlines the XML||RDF strategy.

**Algorithm 1:** XML||RDF

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings
1 $T_X \leftarrow \mathsf{XEval}(Q_X, I_X); T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
2 $T_{XR} \leftarrow \pi_h(T_X \bowtie T_R)$

**Algorithm 2:** XML→RDF

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings
1 $T_X \leftarrow \mathsf{XEval}(Q_X, I_X)$
2 $UCQ \leftarrow \emptyset$
3 **foreach** *tuple* $t_X \in T_X$ **do**
4 $\quad UCQ \leftarrow UCQ \cup PushJoins(t_X, Q_R)$
5 $T_{XR} \leftarrow \pi_h(\mathsf{REval}(UCQ, I_R))$

**Example.** Recall the query in Figure 2, and assume we send the whole $Q_X$ and $Q_R$, respectively, for independent evaluation. $\mathsf{XEval}(Q_X, I_X)$ produces two tuples of bindings:

($\$A = \#205, \$B = \#305, \$C = \#303,$
$\$CA = \langle$body$\rangle$Visiting Iowa today$\langle$/body$\rangle$,
$\$VC =$ "Charlie's campaign"),
($\$A = \#205, \$B = \#306, \$C = \#303,$
$\$CA = \langle$body$\rangle$Visiting Iowa today$\langle$/body$\rangle$,
$\$VC =$ "Charlie's campaign")

Moreover, $\mathsf{REval}(Q_R, I_R)$ returns the following tuple:

($\$X =$:Charlie, $\$Y = \_:x, \$A = \#205, \$B = \#305$)

Combining the two binding tuple sets through a natural join on $\$A, \$B$ and projecting on the head attributes of the query results in the single tuple:

($\$CA = \langle$body$\rangle$Visiting Iowa today$\langle$/body$\rangle$,
$\$X =$:Charlie)

### 4.3 Bind XML, then RDF

The second approach consists in evaluating tree patterns first and, assuming **XURI-out**, pushing the resulting variable bindings into $Q_R$ which is then handed to the RDM.

Algorithm 2, named XML→RDF, details the process. First, $Q_X$ is evaluated, then for each resulting tuple of variable bindings, the $Q_R$ variables on which $Q_R$ and $Q_X$ join are bound to the respective values (XURIs and literals). This substitution is achieved by the function *PushJoins*. If there are several tuples in the result of $Q_X$, this substitution transforms $Q_R$ into a union of conjunctive queries ($UCQ$ in the algorithm), one for each tuple retrieved by $Q_X$.

**Example.** Pushing the result of $\mathsf{XEval}(Q_X, I_X)$ into $Q_R$ results in the following union:

$Q_R(\$X, \$Y,$"Visiting Iowa today") :-
($\$X$, :authorOf, $\$Y$),
($\$Y$, owl:sameAs, #205),
(#305, rdfs:seeAlso, #205),
($\$X$, rdf:type, :MemberOfCongress) ∪
$Q_R(\$X, \$Y,$"Visiting Iowa today") :-
($\$X$, :authorOf, $\$Y$),
($\$Y$, owl:sameAs, #205),
(#306, rdfs:seeAlso, #205),
($\$X$, rdf:type, :MemberOfCongress)

whose evaluation is then delegated to the RDM.[13]

Note that the SPARQL 1.1 recommendation [23] introduced the BIND and VALUES operators to pass *inline* one or more sets of bindings to a SPARQL query. The union of conjunctive queries described above can easily be rewritten using this new syntax. However, the way such queries are evaluated and optimized remains platform-dependent.

### 4.4 Bind RDF, then XML

The main idea of this approach is to evaluate $Q_R$ first and inject the bindings thus obtained into XEval. When considering concrete algorithms for implementing this approach, two independent choices can be made, leading to a total of four possible algorithms. We explain these choices first and then present the resulting four algorithms.

**Does XURI-in hold?** Observe that the bindings returned by $Q_R$ may include XURIs. To exploit these bindings in XEval we need the **XURI-in** assumption, that is, the engine must be capable of retrieving an element having a specific XURI; this is generally not possible with an off-the-shelf XDM, since the implicit XML node IDs are not visible in the XML data and thus are not accessible to the XML queries.

When **XURI-in** does not hold, we may still exploit XURI bindings brought by $Q_R$ as follows.

We term **dereferencing** the process of obtaining from a node XURI, the URI of its XML document, as well as the (unique) linear parent-child XPath expression (possibly with positional predicates) from the root of the document, down to the node itself. For instance, dereferencing the XURI #305 leads to the document URI "doc200.xml" and the linear XPath `/microblog/message[12]/body[1]`. Dereferencing is easily supported if XURIs are implemented using some Dewey-style XML node identifiers, of which [24] is a recent representative. Alternatively, an XURI-to-XPath index can be materialized to support dereferencing through a look-up by the XURI.

[13] As can be seen in the example, in practice *PushJoins* also extends the projection list of $Q_R$ to include the bindings for the variables of $Q_X$ that exist in $Q$'s head but do not exist in $Q_R$ (e.g., the binding for variable $\$CA$ in this example). However, to keep the presentation simple, this detail is omitted from the algorithm's pseudocode.

**Algorithm 3:** RDF⇒XML-URI

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

```
1 T_R ← REval(Q_R, I_R)
2 T_XR ← ∅
3 foreach t_R ∈ T_R do
4     q ← PushJoins(t_R, Q_X)
5     T_XR ← T_XR ∪ π_h(XEval(q, I_X))
```

When dereferencing is available, the RDF-then-XML approach can be implemented by:

1. evaluating $Q_R$;
2. dereferencing any resulting XURIs to linear parent-child XPaths (XURIs correspond to the bindings of the variables in $Q_R$ that also appear as *uri* variables in $Q_X$);
3. composing these XPaths with $Q_X$ and sending the result to XEval.

**One or several XML queries?** A second dimension of choice concerns the way in which we handle *multiple* tuples of bindings returned by the RDM. We could send several XML queries to the XDM, one for each tuple of bindings (this approach can be seen as a union of multiple queries); or, we could gather all these tuples in a collection (i.e., use the union of these tuples) and issue a single query to the XDM, involving this collection.

The difference between these options basically boils down to the relative order between a union and a join. One would expect the XDM to transparently pick the best evaluation order, regardless of the query syntax used. In practice, however, we experienced significant differences in performance, with the single XML query solution being much more efficient.

**Algorithms.** Based on the above analysis, we have devised four concrete algorithms:

- Algorithm RDF⇒XML-URI assumes **XURI-in** (i.e., pushes XURIs into the XDM) and sends **one XML query per tuple of bindings** from $Q_R$;
- Algorithm RDF⇒XML-XPath uses **dereferencing** (i.e., pushes linear XPaths into the XDM) and sends **one XML query per tuple of bindings** from $Q_R$;
- Algorithm RDF→XML-URI assumes **XURI-in** and sends **a single query** to the XDM;
- Algorithm RDF→XML-XPath uses **dereferencing** and sends **a single query** sent to the XDM.

Algorithm 3 details the RDF⇒XML-URI procedure. Here, the function *PushJoins* propagates to $Q_X$ values (XURIs and literals) from the tuples of bindings resulting from $Q_R$.

**Example (RDF⇒XML-URI).** Recall the XR query from Figure 2, where for simplicity we only consider the first

**Algorithm 4:** RDF⇒XML-XPath

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

$T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
$T_{XR} \leftarrow \emptyset$
**foreach** *tuple* $t_R \in T_R$ **do**
  $t'_R \leftarrow Deref(t_R)$
  $q \leftarrow PushJoins(t'_R, Q_X)$
  $T_{XR} \leftarrow T_{XR} \cup \pi_h(\mathsf{XEval}(q, I_X))$

**Algorithm 5:** RDF→XML-URI

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

$T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
$UCQ \leftarrow \emptyset$
**foreach** *tuple* $t_R \in T_R$ **do**
  $UCQ \leftarrow UCQ \cup PushJoins(t_R, Q_X)$
$T_{XR} \leftarrow \pi_h(\mathsf{XEval}(UCQ, I_X))$

**Algorithm 6:** RDF→XML-XPath

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

$T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
$UCQ \leftarrow \emptyset$
**foreach** *tuple* $t_R \in T_R$ **do**
  $t'_R \leftarrow Deref(t_R)$
  $UCQ \leftarrow UCQ \cup PushJoins(t'_R, Q_X)$
$T_{XR} \leftarrow \pi_h(\mathsf{XEval}(UCQ, I_X))$

XML tree pattern $Q^1_X$, and the full $Q_R$. An XQuery serialization of $Q^1_X$ is:

```
for $x1 in collection("XMLDB")//microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message,
    $x4 in $x3//body
return ($x2/text(), $x4)
```

Suppose that the evaluation of $Q_R(I_R)$ has led to the tuple of bindings with $A=#205, and assume **XURI-in** holds. Then, Algorithm RDF⇒XML-URI pushes this XURI into $Q^1_X$, which turns into:

```
for $x1 in collection("XMLDB")//microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message,
    $x4 in $x3//body
where XURI($x4)="#205"
return ($x2/text(), $x4)
```

where the function `XURI($x4)` is assumed to return the XURI of the node to which `$x4` is bound.

Algorithm 4 outlines RDF⇒XML-XPath. Here, the function *PushJoins* is slightly modified w.r.t. Algorithm 3: it adds `where` clause conditions to $Q_X$, stating that every node labeled with a URI variable in $Q_X$ and participating in a join between $Q_X$ and $Q_R$, should be on the path obtained by dereferencing the respective URI retrieved by $Q_R$. Dereferencing is achieved in Algorithm 4 by the *Deref* function.

**Example (RDF⇒XML-XPath).** Continuing on the last example above, assume now that **XURI-in** does not hold, and that dereferencing #205 has led to the document URI doc200.xml and the XPath `/microblog/message[12]/body[1]`. Algorithm RDF⇒XML-XPath injects this XPath into $Q^1_X$ transforming it into:

```
for $x1 in collection("XMLDB")//microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message,
    $x4 in $x3//body
where $x4 is doc("doc200.xml")/microblog/
             message[12]/body[1]
return ($x2/text(), $x4)
```

where we used the XQuery predicate `is` to ensure that `$x4` element is the one having the XURI #205. Clearly, the query could have been written in a more compact manner as:

```
for $x1 in doc("doc200.xml")/microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message[12],
    $x4 in $x3/body[1]
return ($x2/text(), $x4)
```

We leave the task of recognizing this equivalence to the XDM. Algorithms for simplifying such "intersection" queries (in our example, node `$x4` is reached by two different paths) can be found in [25,26].

Algorithm 5 spells out RDF→XML-URI, which assumes **XURI-in** and sends a single XML query to the XDM.

**Example (RDF→XML-URI).** Based on the previous example, assume **XURI-in**, and that $Q_R$ returns two tuples with $A=#205 and $A=#405. In this case, RDF→XML-URI sends the single XQuery:

```
let $XURIList:=("#205", "#405")
for $x1 in collection("XMLDB")//microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message,
    $x4 in $x3//body
where XURI($x4)=$XURIList
return ($x2/text(), $x4)
```

in which the existential XQuery semantics of the list comparison in the `where` clause, ensures that the URI of `$x4` belongs to the `$URIList`.

Our example assumed that $Q_R$ returns bindings for just one URI variable (namely $A). Along the same lines, at the cost of more complex XQuery syntax (which we omit), this single-XQuery approach generalizes to the case where $Q_R$ returns tuples of bindings for several URI variables.

Finally, Algorithm 6 describes RDF→XML-XPath, which uses dereferencing and issues a single XQuery.

**Example (RDF→XML-XPath).** Consider **XURI-in** does not hold, and that $Q_R$ returns the two tuples with

**Algorithm 7:** RDF→XML-Data

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

1 $T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
2 $I'_X \leftarrow Materialize(T_R, I_X)$
3 $Q'_X \leftarrow TripleToTreePatterns(Q)$
4 $T_{XR} \leftarrow \mathsf{XEval}(Q'_X, I'_X)$

$A=#205 and $A=#405, dereferenced into `/microblog/message[12]/body[1]` and `/microblog/message[22]/body[1]`, respectively. In this case, Algorithm RDF→XML-XPath issues the query:

```
let $NodeList:=(/microblog/message[12]/body[1],
                /microblog/message[22]/body[1])
for $x1 in collection("XMLDB")//microblog,
    $x2 in $x1/blogtitle,
    $x3 in $x1/message,
    $x4 in $x3//body
where XURI($x4)=$NodeList
return ($x2/text(), $x4)
```

### 4.5 Materialize RDF, then query XML

Other approaches to query joined XML and RDF data involve *materializing* data retrieved from one sub-instance into a temporary container of the other sub-instance. In short, these approaches push bindings into the data itself, rather than pushing them into the query. Although the materialization step may entail I/O costs, the advantage is that the query sent to the target sub-instance does not contain any union and can be kept small compared with those of the approaches previously described.

We first turn to the case where $Q_R$ is evaluated first. Algorithm 7 details how this join is executed. After extracting tuples that result from answering $Q_R$ over $I_R$ (line 1), the *Materialize* function stores these bindings into $I_X$, creating a new sub-instance containing the actual data *and* the newly added tuples (line 2). This new sub-instance, called $I'_X$, is temporary and ceases to exist at the end of the algorithm's execution. Then, a new query $Q'_X$ is built (function *TripleToTreePatterns*) by turning all triple patterns in $Q$ to tree patterns (line 3). The last instruction of the algorithm (line 4) retrieves the final result simply by evaluating $Q'_X$ over $I'_X$. There are potentially many ways to materialize the additional tuples in the $I'_X$, and converting triple patterns to tree patterns directly depends on the representation used. The representation we chose is presented in the next example.

**Example (RDF→XML-Data)** From our running example, suppose the bindings returned by $Q_R$ are:

```
($X =:Charlie, $Y = :x, $A = #205, $B = #305)
($X =:Charlie, $Y = :x, $A = #205, $B = #306)
```

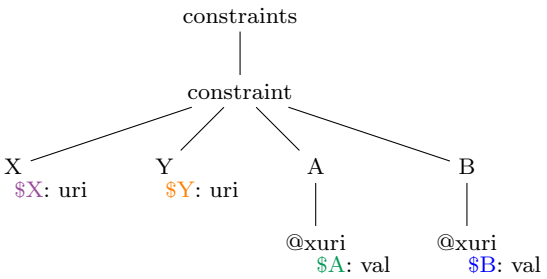


Fig. 4: Additional tree pattern added to $Q_X$

**Algorithm 8:** XML→RDF-Data

**Input** : an XR instance $I = (I_X, I_R)$,
an XRQ query $Q = (h, Q_X, Q_R)$
**Output**: $T_{XR} = Q(I)$, a set of tuples of bindings

1 $T_X \leftarrow \mathsf{XEval}(Q_X, I_X)$
2 $I'_R \leftarrow Materialize(T_X, I_R)$
3 $Q'_R \leftarrow TreeToTriplePatterns(Q)$
4 $T_{XR} \leftarrow \mathsf{REval}(Q'_R, I'_R)$

These bindings are stored in the XDM as a new document such as:

```
<constraints>
<constraint>
<X>:Charlie</X><Y>:x</Y>
<A xuri="#205" /><B xuri="#305"/>
</constraint>
<constraint>
<X>:Charlie</X><Y>:x</Y>
<A xuri="#205" /><B xuri="#306"/>
</constraint>
</constraints>
```

$Q'_X$ is obtained by removing all triple patterns from $Q$ and adding the new tree pattern depicted in Figure 4. Observe that, once extracted from the RDM, XURIs cannot be stored strictly as XML node URIs anymore. If we did so, the XDM would contain distinct XML nodes with identical URIs, which goes against our data model. To work around this, we store XURIs as the value of a reserved attribute. This explains why URI variables are typed as VAL variables in the newly added tree pattern.

### 4.6 Materialize XML, then query RDF

Our last algorithm is the converse of the one presented above. In this case, $Q_X$ is evaluated first. The tuples thus obtained are stored in the RDM, then a single query made of triple patterns only is answered from the newly created RDM sub-instance. Algorithm 8 details the process.

**Example (XML→RDF-Data)** Assuming the evaluation of $Q_X$ over $I_X$ returns the following bindings,

```
($CA = <body>Visiting ..., $A = #205)
```

we store them in the RDM sub-instance as a set of triples, representing a specific tuple of bindings:

```
(urn:1, urn:val_CA, "<body>Visiting ...")
(urn:1, urn:uri_A, #205)
...
```

where *urn*:1, *urn*:*val_CA* and *urn*:*uri_A* are URIs disjoint from those of the RDF instance. The URIs and literals stored in object positions are the values bound to these variables.

The function *TreeToTriplePatterns* in Algorithm 8 returns a query $Q'_R$ made of the triple patterns of $Q$ to which we add the following ones:

```
($binding, urn:val_CA, $CA)
($binding, urn:uri_A, $X)
...
```

These patterns feature variables from the query $CA and $A, in object positions, forming a join with the original triple patterns of $Q$. The variable $binding in subject position joins the additional triple patterns together ensuring that bindings from the same original tuple will be considered together.

### 4.7 Pruning optimizations for RDF-then-XML

We now describe an optimization that can be applied to the strategies binding first $Q_R$ and then $Q_X$. For those algorithms that use dereferencing (that is, RDF→XML-XPath and RDF⇒XML-XPath), one may limit the amount of work sent to the XDM by pruning some of the tuples $t_R$ as follows:

1. For each tree pattern of $Q_X$ and tuple of bindings $t_R \in Q_R(I_R)$, if $t_R$ contains multiple variables bound (in $Q_X$) to nodes of the tree pattern, check the document URIs obtained after dereferencing these variables' values from $t_R$. If two such URIs are not identical, discard $t_R$. The reason is that all XML nodes matching that $Q_X$ tree pattern must belong to the same document. Therefore, $Q_R$ result tuples that attempt to bind them in different documents cannot lead to valid matches.
2. Consider a variable $\$X$, which appears in $Q_X$ as an XURI variable, and bound by $Q_R$ to a URI which is subsequently dereferenced into an XPath expression $xp$. Assume that the path on which $X appears in $Q_X$ is incompatible with $xp$, that is: for any XML sub-instance $\mathcal{D}_X$, we have $xp(\mathcal{D}_X) \cap \pi_{\$X}(Q_X(\mathcal{D}_X)) = \emptyset$. Algorithms for statically detecting such query independence are provided, e.g., in [27].

Algorithm 9 (RDF⇒XML-XPath-Pr) illustrates how to extend RDF⇒XML-XPath to account for these two pruning criteria. Each tuple $t_R$ of bindings returned by $Q_R$ is checked for validity, according to the two criteria provided above. First, the XURIs belonging to $t_R$ are dereferenced into a new tuple $t'_R$ (line 5). Then,

**Algorithm 9:** RDF⇒XML-XPath-Pr

```
   Input  : an XR instance I = (I_X, I_R),
            an XRQ query Q = (h, Q_X, Q_R)
   Output: T_XR = Q(I), a set of tuples of bindings
1  T_XR ← ∅
2  T_R ← REval(Q_R, I_R)
3  foreach tuple t_R ∈ T_R do
4      valid:=true
5      t'_R ← Deref(t_R)
6      foreach tree pattern tx_i of Q_X do
7          Let $V_i^1, $V_i^2, ..., $V_i^{k_i} be the XURI
           variables of tx_i which are bound in t_R to the
           XURIs v_i^1, v_i^2, ..., v_i^{k_i}, respectively
8          Assume dereferencing returns the document
           URI d_i^1 and the linear positional XPath xp_i^1
           for v_i^1, and similarly (d_i^2, xp_i^2) for v_i^2, ...,
           (d_i^{k_i}, xp_i^{k_i}) for v_i^{k_i} in t_R
9          // Compare document URIs:
10         if d_i^1 = d_i^2 = ... = d_i^{k_i} then
11             // Check compatibility between the
               linear XPaths and paths of the respective
               variables in Q_X:
12             foreach $V_i^j, 1 ≤ j ≤ k_i do
13                 if xp_i^j is incompatible with the path on
                   which $V_i^j appears in tx_i then
14                     valid:=false;
15     if valid then
16         q ← PushJoins(t'_R, Q_X)
17         T_XR ← T_XR ∪ π_h(XEval(q, I_X))
```

the document URIs corresponding to XURI variables bound to the same tree pattern are checked for equality, at line 10; then, path compatibility is checked between the linear XPath of each variable, at line 13. Only for valid tuples of bindings, that is, those that pass successfully both pruning criteria, do we push the joins into XEval as in the previous algorithms (lines 16-17).

**Example (RDF⇒XML-XPath-Pr).** Consider an XR query consisting of: $Q_R$ as in Figure 2, and the tree pattern $Q_X^2$ of the same figure. Assume for the purpose of the example, that $Q_R$ returns a tuple of bindings $t_R$ with $B=#405 and $C=#303. Moreover, assume that dereferencing returns:

- `doc(#400)/html[1]/body[1]/div[1]` for #405;
- `doc(#300)/html[1]/div[5]/div[3]` for #303.

Since the two nodes belong to distinct documents, $t_R$ is not used to solicit XEval.

As an illustration of the second pruning rule, assume that $Q_R$ returns a tuple with $B=#305 and $C=#405. In $Q_X^2$, the variable $\$C$ is on the path `html//div/div`. This path indicates that the parent of the node to which $C is bound is labeled `div`, whereas the XPath resulting from dereferencing #405 indicates that the parent should be labeled `body`. Thus, we have detected an incompatibility between the two, and $t_R$ is discarded.

In a similar way, RDF→XML-URI and RDF→XML-Data could be extended with the same flavor of pruning,

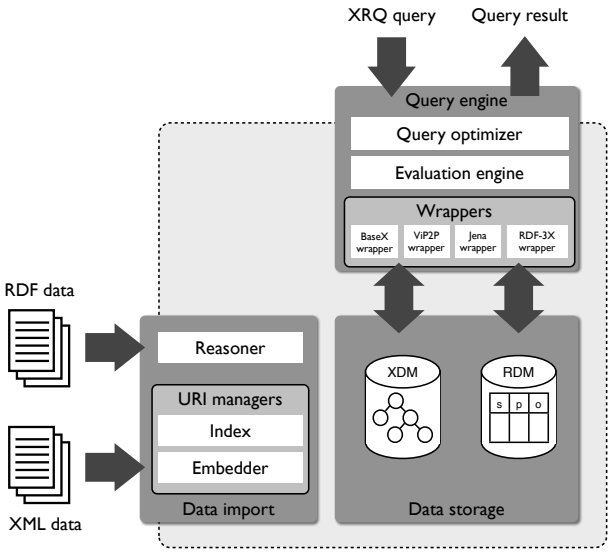


Fig. 6: Architecture of the XR platform.

leading to the respective variants RDF→XML-XPath-Pr and RDF→XML-Data-Pr (omitted for brevity).

When both **XURI-in** *and* **dereferencing** are supported, one may apply the same pruning technique as presented in Algorithm 9, and push XURIs directly into the XML sub-queries rather than the dereferenced nodes (lines 16 and 17). This variant comes in two flavours, RDF→XML-URI-Pr and its tuple-at-a-time counterpart RDF⇒XML-URI-Pr.

**Putting it all together.** Figure 5 systematizes the XRQ evaluation algorithms discussed so far.

## 5 The XR Platform

We implemented the XR platform in Java 1.6 (16.000 lines); Figure 6 depicts its architecture. The XR platform builds on pre-existing data management systems: one for XML (XDM) and one for RDF (RDM). Such systems are integrated within through wrappers that allow delegating them the evaluation of XML, respectively, RDF sub-queries of XR queries. Since XRQ corresponds to well-established conjunctive subsets of XQuery and SPARQL, most existing XDM and RDM may be plugged in our platform.

### 5.1 Existing wrappers

As RDF query engines, we have experimented with RDF-3X [28], established as a very efficient RDF query processor; we used the version 0.3.7. We also implemented a wrapper for Jena 2.6.4, a widely used open source suite. Our experiments with Jena have shown that it does not scale beyond a few million triples, thus our experiments focus on RDF-3X.

Concerning the XML query engine, our experiments use the BaseX platform (`http://basex.org`), version 7.3. BaseX is a quite recent XML store which we found to be competitive w.r.t. QizX and MonetDB, in recent tests that we ran comparing them on the XMark [29] and XPathMark [30] benchmarks. *We used BaseX "off-the-shelf"*, and interacted with it through its XPath- and XQuery-compliant query interface. Unless otherwise specified, thus, BaseX is our XDM. It *does not* satisfy **XURI-in** nor **XURI-out**.

Given the importance of XURIs in the XR model, we also wanted to test the case when we have access to the XDM's internals, and in particular to its internal node IDs, exposed as XURIs. For that purpose, we used the XML query engine of the ViP2P project [31] (see also `http://vip2p.saclay.inria.fr`), which we had developed in the group. ViP2P supports the XML tree pattern dialect introduced in Section 3.

The ViP2P XML engine is based on SAX, and evaluates tree patterns by traversing the complete document, computing and returning node XURIs dynamically as required by the query. Thus, ViP2P satisfies **XURI-out**.

ViP2P also satisfies **XURI-in**, but not very efficiently: to find the XML element having a given XURI, it traverses the complete corresponding document from the beginning and stops upon encountering the respective element. To get more efficient support from ViP2P, we exploited its built-in materialized view-based rewriting framework [32], and considered the optimistic case in which *when processing a query* $Q = (h, Q_X, Q_R)$*, each tree pattern in* $Q_X$ *is available as a materialized view.* This is obviously not always guaranteed; therefore, our experiments with ViP2P are aimed as a "lower bound" of sorts, for the case when $(i)$ we do have access to the XDM internals and $(ii)$ we are able to tune the store to a specific workload[14].

### 5.2 XR's own query engine

To combine partial query results, the XR platform provides its own execution engine, comprising *selections*, *projections*, *hash joins* etc. It also includes a generic *fetch* operator which, depending on the context, performs the function of REval and XEval introduced in Section 4. The platform is currently single-site, but to exploit the parallelization opportunities provided by nowadays' multicores, in our implementation, all the *fetch* operators of an execution plan are launched simultaneously when the plan execution begins (as opposed to letting the implicit iterator-based scheduling [33] of our operators trigger them). Our tests have shown that such parallel, eager *fetch* execution significantly speeds up the query evaluation. This

[14] One could further speed up ViP2P by $(i)$ indexing its views on the XURI attributes that are passed as bindings from the RDF query and/or $(ii)$ pushing value joins among $Q_X$ tree patterns within the materialized views etc. We did not pursue these alternatives, as they are rather orthogonal to the main purpose of this paper.

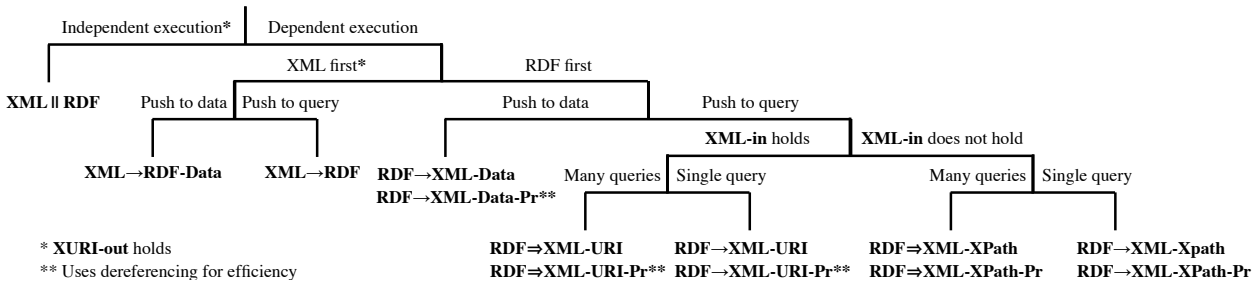


Fig. 5: Taxonomy of the proposed XRQ query evaluation algorithms.

is because the *fetch* operators ship potentially complex sub-queries to the underlying XDM and RDM, thus their evaluation is a significant part of the overall processing time.

### 5.3 URI management

For URI management (**XURI-in**, **XURI-out** and **Deref**), we resorted to the following techniques.

*When using BaseX*, we store *within the XML instance*, the XURIs of only those XML nodes which are referred to by the RDF sub-instance. Specifically, let $d$ be an XML document and $n \in d$ a node, and dURI:lnID be the XURI of $n$, where dURI is the URI of $d$ and lnID is the local identifier of $n$ within $d$. If dURI:lnID appears within the RDF data instance, then within $d$, we add a special attribute to $n$, of the form `id=''lnID''`, which the run-time re-assembles with dURI into $n$'s full XURI. The module inserting such IDs is the *embedder* in Figure 6.

The advantages of this approach are: (*i*) both **XURI-in** and **XURI-out** can be supported through trivial XQuery rewritings, and (*ii*) some underlying systems can be tuned to index these attributes and therefore improve the performance of (XR-specific) joins between the XML and RDF data. One may also consider leveraging directly the internal ID representation schemes specific to most XDMs, as we did in a previous version of this work [34].

*For BaseX and ViP2P*, to implement the **Deref** function, we store in a dedicated index (materialized in the XR platform but outside the XML data management platform), for each node URI, the parent-child XPath query (with positional predicates) leading from the document root to the respective node. For instance, this index associates to $\texttt{doc}_1\texttt{:node}_{15}$ the corresponding node XPath, e.g., `/a/b[1]/c[2]/d[1]`. Clearly, once stored, these XURI/XPath pairs can be indexed in one or two ways (e.g., in persistent hash tables provided by the BerkeleyDB library [35]) so as to perform the dereferencing in constant time. In our platform, we indexed the XPaths with the XURIs as look-up keys. This approach for implementing **Deref** is non-intrusive and can be applied on the top of any existing system.

The XR plan generator takes as input an XR query and a given query evaluation strategy among those described in Section 4, and produces an execution plan implementing the respective strategy for that query. As explained in Section 4.2, one needs to decide how to group the XML sub-queries sent to XEval, i.e., whether to delegate value joins among XML tree patterns to the underlying database or not. To determine this, the XR platform includes a calibration module which sends to the XML database a set of fixed queries whose performance it then compares with the case when value joins among XML tree patterns are run in the XR platform and these tree patterns are run independently on the XML database.

Finally, the XR platform includes an XR data generation module we devised, which we further detail when presenting our experimental evaluation, in the next section.

## 6 Experimental Evaluation

This section presents the findings of our experimental study. Section 6.1 describes the experimental settings we used to test our algorithms. Section 6.2 provides an extensive comparison of all our XR query evaluation algorithms on a small XR data instance, illustrating their performance and allowing us to discard the most inefficient ones. Section 6.3 focuses on the more efficient ones, and studies their scalability with respect to the size of the data instance. In Section 6.4, we compare these algorithms based on two quite different XDMs, then we conclude.

### 6.1 Experimental settings

*Datasets* We have used a set of synthetic XR data sets, generated in two stages as follows.

First, we use the XMark [29] XML document generator to produce a set of XML documents.

Second, we generate a set of RDF triples, some of whose subject and object values are URIs of nodes from the previously generated documents. Specifically, 1/2 of the subjects are URIs of XML nodes, while the others are synthetic URIs, picked from a fixed pool using a uniform distribution; 1/3 of the objects are

| Dataset sizes | #RDF edges (millions) | #XML edges (millions) |
|---|---|---|
| $\mathcal{D}^1_{1/3}$ | 0.5 | 1.6 |
| $\mathcal{D}^1_1$ | 1.5 | 1.6 |
| $\mathcal{D}^1_3$ | 5 | 1.6 |
| $\mathcal{D}^{10}_{1/3}$ | 5 | 16 |
| $\mathcal{D}^{10}_1$ | 15 | 16 |
| $\mathcal{D}^{10}_3$ | 50 | 16 |
| $\mathcal{D}^{100}_{1/3}$ | 50 | 167 |

Table 1: XR datasets used in the experiments

XML node URIs, 1/3 are picked from the fixed pool of subject URIs, while the last 1/3 are taken from a distinct (disjoint) URI set. The values of properties in the RDF data are picked from a set of 1,185 distinct properties present in the DBPedia database [5], using a Zipf distribution.

This data generation approach aims at resembling actual settings where some RDF triples annotate the XML nodes with properties from a given vocabulary, some triples connect the nodes to each other, and finally some other triples are not related to the document nodes (but may still join with those that are).

We moreover controlled:

- The size factor of the XMark XML generator, denoted $i$. We experimented with size factors of 1, 10 and 100, which respectively lead to XML datasets of 100MB, 1GB and 10 GB.
- The splitting of the XML content across documents. This parameter matters, because each XML tree pattern can only match within a single document; moreover, XML query processors often perform better on smaller documents. Thus, we generated the XML data: all in a single file; split in $n$ files where $n$ is the XMark input size factor (thus, each file is of about 1MB); finally, split in XML files of approximately 1000 nodes each. Unless specified otherwise, in this paper, we report on this last option, which enabled us to best compare our algorithms. Results with other XML segmentation sizes are provided on our online experimental site [36].
- The ratio between the number of XML nodes and the number of RDF triples in the instance, denoted $j$. We chose size ratios of 1/3, 1 and 3. This parameter was introduced in order to control the amount of connections between the XML and RDF parts of the data set.

We denote by $\mathcal{D}^i_j$ the dataset obtained by setting the XMark input size to $i$ and the RDF-to-XML ratio to $j$. For instance, $\mathcal{D}^{10}_{1/3}$ is a dataset generated with size factor 10 (approximately 1GB and 16M XML nodes), and 1 RDF triple for 3 XML nodes, i.e., approximately 5M triples in this case. The size of the datasets w.r.t. the input size factor are reported in Table 1.

*Workloads* We hand-crafted four workloads of eight queries each. Queries are ordered by increasing complexity, from one tree pattern joined with one triple pattern, to three tree patterns joined with two triple patterns. On average, a tree pattern has 4.7 nodes. Each query features joins: between the triple patterns, between the tree patterns, and between triple and tree patterns, on node URIs. Query $Q_7$ features a Cartesian product in $Q_X$, whereas the query as a whole is Cartesian product-free. Finally, $Q_8$ features a Cartesian product among $Q_R$ triples, although the query is overall connected through shared variables among $Q_X$ and $Q_R$.

All workloads share the tree and triple patterns of the first workload $W_1$. To gauge the impact of the selectivity of each sub-query, we have added selections in the other workloads as follows. In the workload $W_2$, selections have been added to the RDF triple patterns only. In the workload $W_3$, selections have been placed on XML tree patterns only, while workload $W_4$ features the selections of both $W_2$ and $W_3$, on the XML and RDF patterns.

*Encoding URIs for BaseX and consequences for querying* As explained in Section 5, BaseX satisfies neither **XURI-in** nor **XURI-out**, and to be able to test all our algorithms on BaseX, we added `xml:id` attributes to only those XML nodes whose XURIs appear in the RDF sub-instance. With this encoding of XURIs in the data, BaseX can be considered as satisfying both **XURI-in** and **XURI-out**.

It turns out that this simple encoding improves the performance of $Q_X$ evaluation, even for simple strategies such as `XML||RDF`. The reason is that whenever **XURI-out** is assumed, the XQuery syntax of $Q_X$ involves the `xml:id` attribute. This attribute is present only in those nodes which appear as subjects or objects within the RDF sub-instance. Thus, $Q_X$ filters out of the XML instance the XML nodes whose URIs do not appear in the RDF instance.

### 6.2 Comparison of all strategies

Our first set of experiments compares all the strategies described in Section 4, on the dataset $\mathcal{D}^1_1$ and on all workloads. In this experiment, we sent to the RDM the connected components of $Q_R$ one by one, whereas to the XDM we sent only isolated tree patterns, and performed all the remaining joins using our own operators, at the level of the XR engine and outside the XDM. Our calibration tests indicated that these choices allowed us to maximize the performance of the RDM, respectively, XDM. Figure 7 presents the running time (limited to our timeout of five minutes) for workloads $W_1$ to $W_4$ in this setting.

A first remark is that the workload $W_1$, with less selections in $Q_X$ and $Q_R$, is the hardest, that is, for each strategy and query $Q_i$, the strategy's running

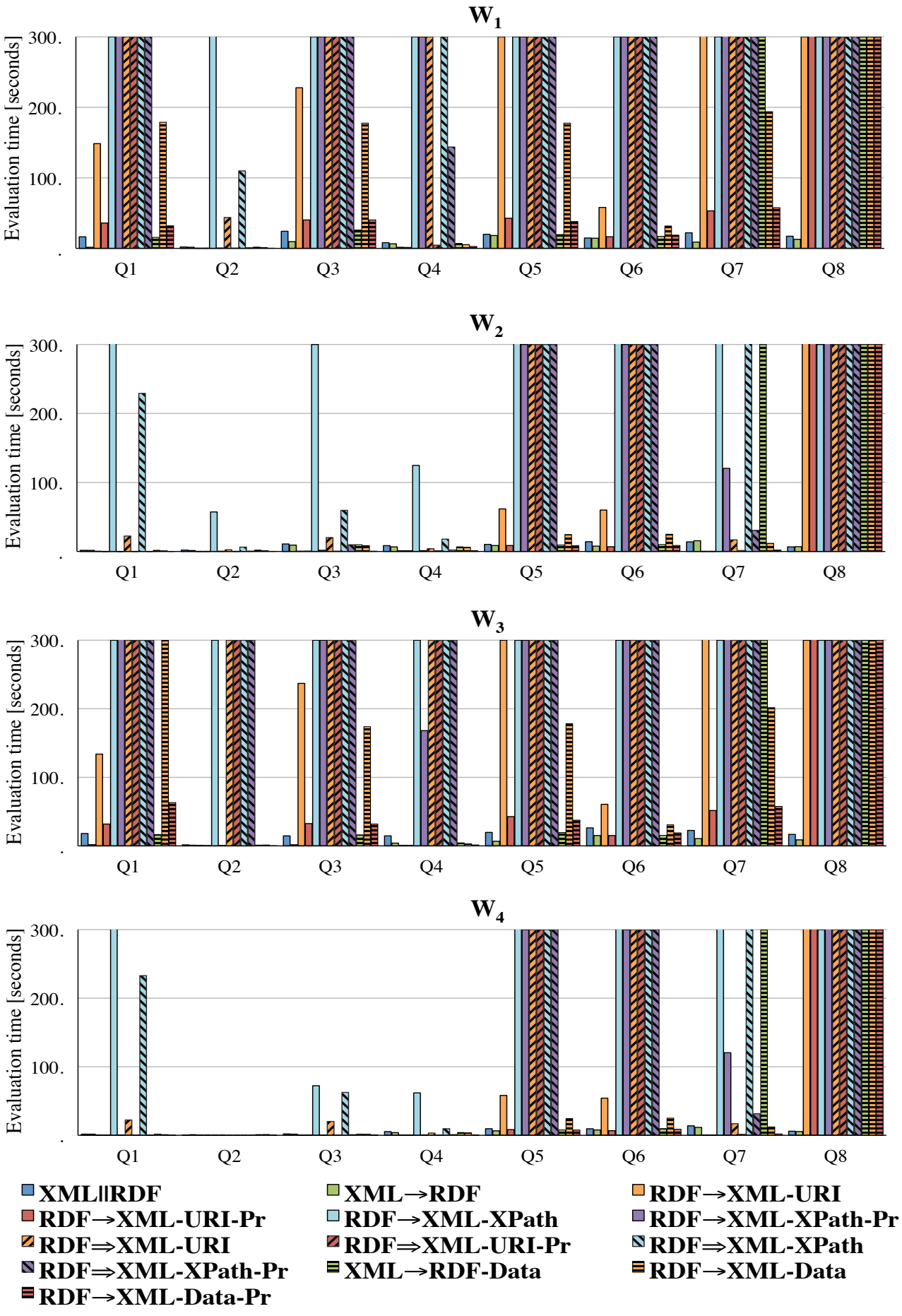


Fig. 7: XR query evaluation strategies compared on workloads $W_1 - W_4$ and dataset $\mathcal{D}_1^1$.

time is longest on the $Q_i$ from $W_1$. Similarly, $W_4$, featuring selections both in the XML and RDF sub-queries, is the easiest. The workloads $W_2$ and $W_3$, having selections only in the RDF, respectively, the XML part, are in-between; the "harder" queries ($Q_5$ to $Q_8$) are poorly handled in both workloads, while the "simpler" queries ($Q_1$ to $Q_4$) are evaluated more efficiently in their $W_2$ versions than in their $W_3$ counterparts. This is because a selection has a very significant impact on the amount of data manipulated by $Q_R$, turning, for instance, a triple of the form ($x, $y, $z) which matches the whole RDF sub-instance, into one of the form ($x, :p1, $z) matching only a few triples. In contrast, a selection added to $Q_X$ may turn, e.g., `/site//person` into `/site//person[age=''20'']`, still a sizeable reduction in the result size, but not as dramatic as in the case of RDF.

Our second remark concerns the tuple-at-a-time strategies from the RDF-to-XML family, those whose names include RDF⇒XML (and which are shown in oblique dashed bars in the Figure). Overall, these strategies perform very poorly, for all but a few selective queries in $W_2$ and $W_4$. Among the worst are RDF⇒XML-URI (Algorithm 5) and RDF⇒XML-XPath (Algorithm 6), running out of time for all but seven (respectively, two) queries. The tuple-at-a-time RDF⇒XML algorithms are slow because of their numerous calls to the XML engine. Moreover, RDF⇒XML-URI is better than RDF⇒XML-XPath. This is because RDF⇒XML-URI assumes **XURI-in** and thus performs the join between the RDF bindings and the XML database, on the `xml:id` attribute. RDF⇒XML-XPath requires evaluating numerous linear XPath expressions, which slows down executions significantly. Finally, tuple-at-a-time strategies with pruning, having names of the form RDF⇒XML*Pr, bring only marginal performance improvements. Based on these experiments and many similar others, we decided to discard the tuple-at-a-time RDF-to-XML strategies from further tests.

A third remark is that among the remaining strategies, pruning does help. For instance, RDF→XML-XPath-Pr performs in many cases better than RDF→XML-XPath; the latter is overall not competitive, thus we will omit it from further tests. Similarly RDF→XML-URI-Pr is often better than RDF→XML-URI.

Based on this analysis, in the following, we only consider the strategies showing acceptable performance in Figure 7, namely: XML||RDF, XML→RDF, RDF→XML-URI, RDF→XML-URI-Pr, RDF→XML-XPath-Pr, XML→RDF-Data, RDF→XML-Data and RDF→XML-Data-Pr.

### 6.3 Scalability

In this second batch of experiments, we focus on the scalability of the competitive strategies when the size of the XR data instance grows. For clarity, we needed an aggregate measure to characterize the cumulated size of the XML and RDF sub-instances. We chose the *total number of edges* in the data instance, that is: the number of XML nodes (we can view each of them as being at the lower end of an edge in the respective tree) plus the number of RDF triples (each triple can be seen as an edge between its subject and object). We used datasets of various sizes, ranging from $D^1_{1/3}$ to $D^{100}_{1/3}$ (the exact cardinality characteristics of these datasets are listed in Table 1). For instance, for $D^{100}_{1/3}$, 217 M edges correspond to a total of 17 GB of data (11 GB of XML and 6 GB of RDF). We ran the queries of workload $W_4$, since its selections both in the XML and RDF sub-queries made it closest to real-world scenarios.

Figure 8 shows the variation of the evaluation times when the dataset (measured in edges) increase. Notice the logarithmic scale on both axes. As in the previous experiments, we used a time-out of 5 minutes and did not plot the runs interrupted at the time-out.

For the less complex queries $Q_1 - Q_4$, all strategies scale up to the largest data size and roughly linearly. The algorithms from the RDF→XML family, namely RDF→XML-URI, RDF→XML-URI-Pr and RDF→XML-XPath-Pr perform best for the most selective queries ($Q_1$ to $Q_4$). The advantage of the pruning-based strategies against the plain RDF→XML-URI fades out at large data scales, since the time spent comparing XURIs (or XPaths) offsets the benefit of pruning the binding tuples sent to the XDM. Strategies XML||RDF and XML→RDF exhibit similar behaviour and also scale roughly linearly. While the conceptual difference between independent and dependent execution is important, in practice the difference may be smoothed out by the fact that for both XML||RDF and XML→RDF, when encoding XURIs as XML attributes, the XQuery corresponding to $Q_X$ operates quite some filtering on the XML sub-instance, even in the absence of passed XURIs (as we have explained in Section 6.1).

For the more complex queries $Q_5 - Q_8$, Figure 8 shows that RDF→XML-XPath-Pr takes longer than the time limit in most cases. This is because in this strategy, dereferencing entails many individual XPath expressions packed into the single XQuery sent to the XDM, which fails to process them. The other strategies fare better; remember that the curves end before the first point that would cross the time limit. Interestingly, XML||RDF behaves well up to the largest data size on $Q_8$, the query with a Cartesian product within $Q_R$, thanks to the optimization consisting of sending to the RDM connected queries only. As an example, on the smallest data instance, $Q_8$ is evaluated by joining the result of one triple pattern (approximately 150 triples) with the XML tree pattern results (approx. 14.000 tuples), and then with the result of the second triple pattern (200.000 triples), leading to a result of 1 triple. This demonstrates the interest of carefully choosing the queries to be delegated to the XDM, respectively, RDM, as discussed in Section 4.1.

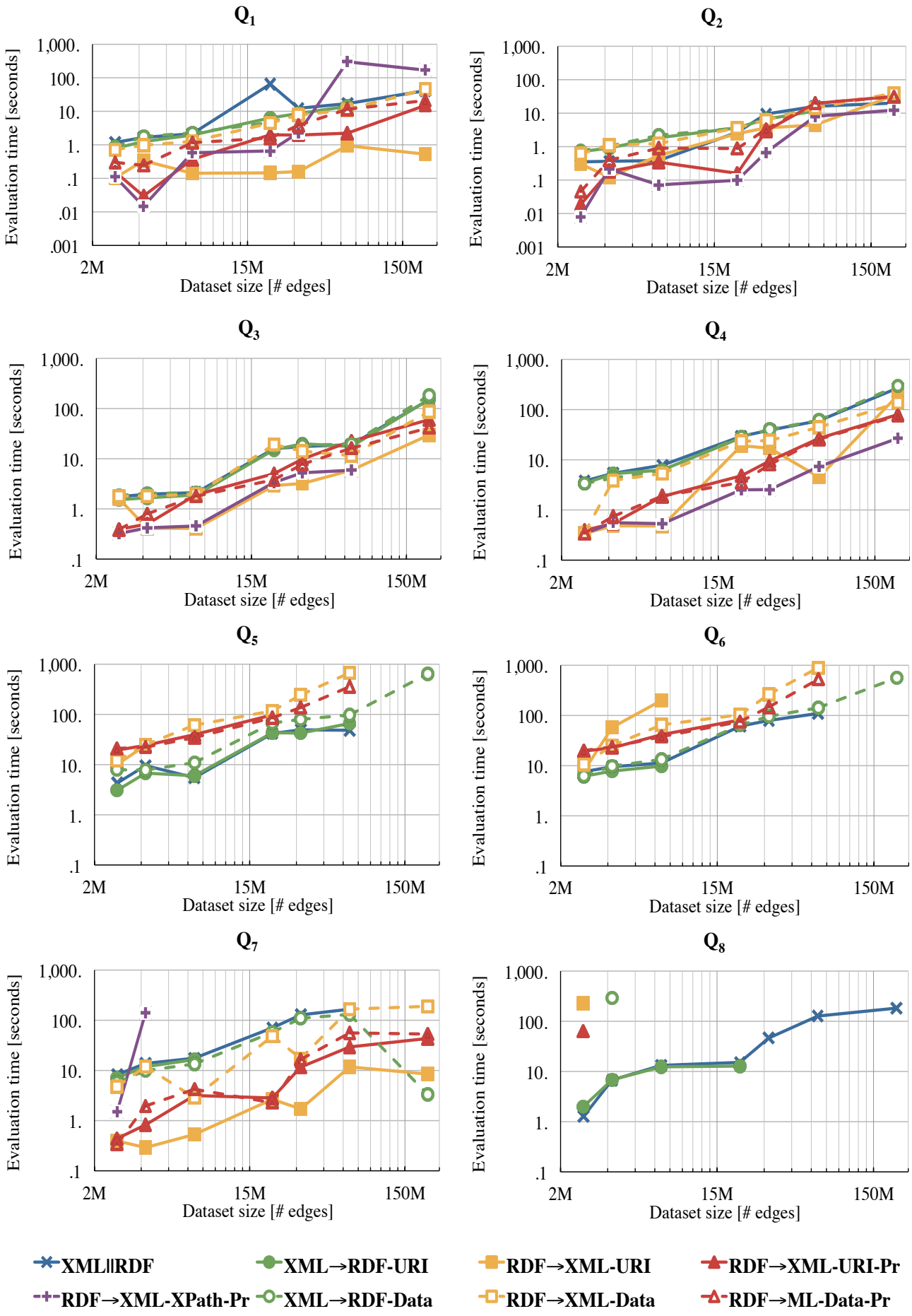


Fig. 8: Evaluation times for $W_4$ with datasets of increasing sizes.

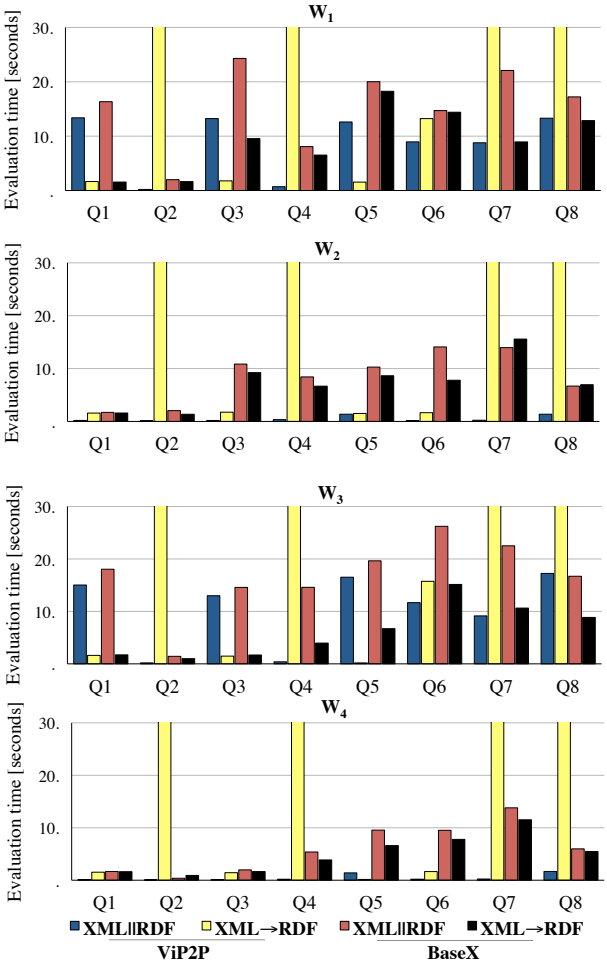


Fig. 9: Evaluation times for workloads $W_1$ to $W_4$ on dataset $\mathcal{D}_1^1$ using ViP2P and, respectively, BaseX.

Each strategy involving data materialization presents a similar trend with its non-materializing counterpart, but with slightly worse performance. For instance, RDF→XML-Data is generally one order of magnitude slower than RDF→XML-URI, while RDF→XML-Data-Pr tightly follows the performance of RDF→XML-URI-Pr. This is due to the materialization cost, which involves disk I/O. The main advantage of those strategies, however, lies in their robustness. As selectivity decreases, strategies that pass information at the query level do not scale, while materialization pays off. Note that curves do not climb monotonously due to the fact that each dataset was generated independently. Therefore larger datasets do not necessarily include smaller ones. This is particularly obvious in $Q_7$ with strategy XML→RDF-Data where response time suddenly declines for the largest dataset. In this case, not only no materialization takes place, but RDF-3X statically detects that the final query returns an empty result.

### 6.4 Experiments using VIP2P

The last experiments we present compare two quite different XDMs: on one hand BaseX off-the-shelf, and on the other hand our own ViP2P engine, both of which were detailed in Section 5.1. We recall that unlike BaseX, ViP2P natively supports **XURI-out**, simplifying the implementation of the XML||RDF and XML→RDF strategies. Moreover, ViP2P is able to exploit materialized views, expressed as joins over tree patterns, to efficiently rewrite queries [32].

To see if the benefits of such view-based techniques transfer to XR query evaluation, prior to running an XR query $Q$, we materialized *each tree pattern in* $Q_X$ *as a separate view*. This admittedly puts ViP2P at an advantage compared to engines which do not support XML materialized views; indeed, the latter are not as frequently provided as is the case for XML indexes. Therefore, our motivation for including VIP2P with this configuration in our tests, was to illustrate the performance than can be achieved using an appropriately set up XDM; view-based rewriting techniques, e.g. [15,32], are likely to be gradually included in popular XML databases as they mature.

Figure 9 depicts the running times of strategies XML||RDF and XML→RDF on the workload $W_4$, when the XDM is ViP2P and BaseX respectively (the BaseX times are from Figure 7, re-plotted here as a reference). Overall, ViP2P performs better than BaseX for both strategies, in particular more than an order of magnitude faster for $Q_6$ and $Q_7$. For the other queries, the times differ by less than one order of magnitude, and overall the trends are similar - "hard" queries for a strategy and system tend also to be comparatively hard for the other system using the same strategy. This gives some support to the idea that our XRQ evaluation strategies are not tied to the particulars of one engine and can accommodate different underlying systems.

In Figure 9, we stopped execution at 5 minutes. All runs ended much faster, except for XML→RDF on ViP2P, on the queries $Q_2$, $Q_4$, $Q_7$ and $Q_8$. We investigated this and found a surprising explanation. In these cases, XML→RDF sends to RDF-3X the XURIs retrieved by ViP2P. *Because ViP2P assigns XURIs to all nodes (whether or not these XURIs appear in the RDF data)*, some of the XURIs ViP2P sends to RDF-3X *are not present in the RDF database.* For reasons not yet clarified, RDF-3X is extremely slow on queries where a variable must belong to a given set of URIs, *if some of these URIs are not in its RDF database.* The difference w.r.t. the same query but using only URIs from the RDF database is a factor of more than a hundred. We have isolated a small example exhibiting this problem and contacted the system authors; when the problem is clarified or solved, we will update the corresponding graphs on our online experiment site [36]. Except for these cases, RDF-3X was overall fast and accurate in our tests, thus we kept it as the RDM of choice for our experiments.

Interestingly, when XML→RDF times-out on ViP2P, XML→RDF on BaseX runs typically fast! This is because, as explained in Section 6.1, *the XURIs sent by*

*BaseX to the RDM are only those of nodes referred to by the RDF sub-instance.* Therefore, the unexpected behaviour of RDF-3X is not triggered[15].

### 6.5 Experiments conclusion

Our experiments allow us to establish the following observations. First, naïve tuple-at-a-time strategies for passing XURIs from the RDM to the XDM are prohibitively slow, even when applying pruning optimizations; similar strategies which pass a single query to the XDM perform much better. Second, XML||RDF and XML→RDF are clearly the best on small data instances (Figure 7), and are robust (especially XML||RDF) up to very large data instances (Figure 8). Thus, if the XDM supports **XURI-out**, one can safely choose the XML||RDF or XML→RDF strategies. This supports the idea that deploying XR based on an XDM whose internal node IDs can be exposed as XURIs, leads to simple yet efficient and robust XRQ evaluation strategies.

For queries and data instances of moderate size, however, the pruning-based strategies RDF→XML-URI and RDF→XML-XPath-Pr can be faster by one order of magnitude than XML||RDF and XML→RDF; RDF→XML-URI requires **XML-in**, whereas RDF→XML-XPath-Pr does not. The advantages of RDF→XML-XPath-Pr are erased if many XURIs are passed from the RDM to the XDM, e.g., in $Q_5 - Q_8$ in Figure 8, since the evaluation of numerous linear XPath expressions (to check whether the nodes from the XML and RDF sub-instances coincide) incurs high costs. Strategies involving materialization, although generally slower than their information-passing counterparts, tend to scale well beyond them.

Finally, we have shown that improvements to the performance of the underlying XDM, in particular by means of storage tuning using VIP2P as the XDM, translate into respective gains for the overall XR query performance. This, as well as our XR platform design which communicates with existing systems through wrappers, and our design of algorithms depending on the hypotheses and capabilities of the underlying XDM, give us confidence that the XR model can be efficiently deployed in a variety of settings.

## 7 Related Work

Two major lines of work are closely related to this paper. The first shares our motivation of annotating structured data, while the second is related to achieving interoperability between the XML and RDF data models.

[15] This interaction between XURI encoding and RDF-3X performance can be reasonably seen as an "implementation accident"; we only explain it for completeness.

### 7.1 Standards and tools for annotated documents

Since the emergence of RDF, a set of tools was proposed to exploit the RDF model and enable users to attach semantic annotations to Web pages. The representation of annotations on XML documents has inspired many projects focusing on a data model perspective [37,38], or an end-user perspective, with tools to annotate web pages manually [39,40] or in a semi-automatic fashion [41,42]. A comprehensive overview of annotation systems can be found in [43]. However, these works focus solely on the problem of storing and querying RDF annotations, and they do not consider the possibility to query simultaneously the structured documents and the annotations on top of them.

Many applications require smart warehousing of structured (or simple text) documents, notably on intranets, where one tries to make the most out of the various documents created by employees on projects which may be similar to each other. In the French R&D project WebContent [44], we have worked on building tools for warehousing semantically annotated pages gathered from the Web. In WebContent, Web crawlers gathered pages on specific topics, e.g., specialized press reviews of aircraft for the Airbus project partner; such pages were then cleaned of unwanted banners etc., a natural language analysis was run and specific entities (such as e.g., "Airbus A320") were localized in the text. Accordingly, the documents were annotated with this named entity, allowing to connect them to specific concepts in the ontology, such as "passenger airplane", "EU-manufactured aircraft" etc. The XR model extends and generalizes the WebContent data model by allowing XML nodes to be referenced in RDF in all places where a URI can appear as opposed to only subjects, as was the case in WebContent. The unified language of XR is also novel and specific to this work. It provides a flexible framework for capturing such semantic annotations at a fine granularity and processing complex queries on top of them.

The problem of publishing RDF annotations *within* XML documents has been tackled by recent technology standards applying in the XHTML context: microformat [45], eRDF [46] and W3C's RDFa [47] standard. The goal of these works is to provide specific syntax enabling the publisher (author) of a Web page to embed some semantic annotations in the page itself. However, such models can only be used by those having the right to modify the page, which is quite restrictive. Moreover, the model does not lend itself to the situation when one user wishes to keep her annotations of a given document private (or share only specific annotations with specific users).

### 7.2 Interoperability between XML and RDF

RDF is a model rather than a language. As such, it has several serializations, the most popular of which is actually based on XML. However, any particular way of encoding triples into trees must somehow arbitrarily pick or create root elements without a clear RDF meaning, while a central RDF feature, namely, joins on URIs appearing in several triples, is encoded by sharing XML attribute values. Processing an RDF query on such XML-encoded data leads to XML queries with numerous value joins, whose evaluation is still challenging for current XML query processors [22], an observation confirmed also in our previous work [48]. Thus, one can consider the XML serialization of RDF as helpful for data sharing but not for human consumption, nor for query processing.

In the same vein, there have been several proposed languages which allow, as described in W3C's GRDDL recommendation [49], the transformation of XML data to RDF and vice versa [50,7]. In the literature, these are known as *lifting* and *lowering*, respectively. Some of this works consider employing the query language of one model to query the other (e.g., using XQuery to query RDF) [51,52] or building hybrid languages that embed constructs of a query language for one model (e.g., XPath) into a query language for the other model (e.g., SPARQL) [53].

To this family also belongs XSPARQL [7], which allows uniform querying of XML and RDF interleaving the XQuery and SPARQL syntaxes. XSPARQL queries may be translated either completely into XQuery, or partially to XQuery with custom function calls to a SPARQL engine. From this perspective, the XSPARQL execution engine compares directly with our current XR engine, since they both delegate processing to existing underlying engines. The evaluation of XSPARQL, based on XMark queries, is quite comparable with ours, although they do not consider our URI-based connections between the two subinstances. Interestingly, they obtain much better performance when translating XSPARQL to finely tuned XQueries, whereas most queries do not work for 100 MB of data (the size of our smallest data set!) if they are partially translated into SPARQL. By taking some of the joins outside the XDM and RDM and intelligently delegating sub-queries, in XR we were able to scale two orders of magnitudes beyond the XSPARQL engine. Finally, an interesting work [54] presents a data model framework rich enough to capture side by side XML and RDF, however, they do not share the particularity of XR consisting of considering XML nodes as resources and injecting them into the world of RDF statements.

The transformation of XML into RDF so that both can be queried with SPARQL is studied in [53,55,56]. This conversion brings both models to the level of the more complex (RDF), which provides sufficient generality, but loses the performance benefits attained by current XQuery processors on many types of queries (and in particular on XPath 1.0 on which many of them perform quite well). Moreover, this XML-to-RDF conversion does not envision treating XML nodes as resources, either.

We have previously outlined the core XR ideas in a short paper [34] as well as in a longer article presented in an informal setting (no proceedings) [57]. These works have introduced the data model and query language; the evaluation algorithms at the core of the current submission are new. Among the related works referenced above, XR also stands out by having been implemented in a full platform, and scaling two orders of magnitude beyond comparable systems [53].

## 8 Conclusion and Perspectives

Structured text, e.g., Web contents, electronic books or enterprise documents, is frequently encoded in XML, and is often valuable in this structured, linear form, which comprises not only facts (or data), but also a linear discourse building ideas from paragraphs and metaphors from words; the original text also serves as reference and lends its authority, e.g., as a proof or a citable source. Contemporary means of exploiting and enriching electronic structured text require the ability to interconnect it with existing data- and knowledge-bases, and to do so in a manner as automatic as possible. A database of documents enriched this way allows not only to better exploit the text, but also to better illustrate and connect the resources and concepts of the database through the documents.

While many works have focused on devising automatic and semi-automatic text annotation tools, drawing on Natural Language Processing capabilities, we have considered the problem of modelling and efficiently querying such corpora of interconnected documents, facts and concepts. Our first goal was to re-use whenever possible, thus we devised the XR data model that naturally extends the W3C's existing XML and RDF model, connecting them on the core idea that any XML node may have a URI, which in turn may appear in the RDF database in any place where a URI is allowed to be. (This may be easily extended to allow annotations at even finer granularity, e.g., a word appearing in a text node.) We have accordingly proposed a core XR query language, combining the conjunctive cores of XML and RDF standard query languages, i.e., triples and tree patterns possibly connected through various flavors of joins. We have then investigated efficient light ways of processing XRQ queries, relying on existing XML, respectively, RDF storage and query engines. It turns out that the central connection made in the XR model on XML node URIs requires some care, given that XML node identity is implicit in the XML model and not necessarily

explicit. We identified the core hypothesis which the XDM may or may not satisfy, and accordingly devised and implemented thirteen XR query evaluation algorithms (Figure 5), some of which exploit some simple optimizations.

We have built an XR platform which interfaces with various XML, respectively, RDF systems by means of wrappers, and experimented with a variety of systems including Jena, RDF-3X, MonetDB/XQuery, QizX, BaseX, and our in-house ViP2P XML query processor. We present the results obtained with the most stable and efficient platforms, which we found to be RDF-3X, BaseX, and ViP2P (the latter hand-tuned for performance). Our experiments demonstrate that there are wide performance differences between various strategies, and that the most efficient (XML||RDF and XML→RDF) scale up well on databases of a total (XML+RDF) size of up to 17 GB (210 millions edges); however, in specific cases (moderate-size databases and simple queries) other strategies, and in particular RDF→XML-XPath-Pr may be much faster.

Based on these observations, our next task is to devise a global XR optimizer capable of automatically selecting the most appropriate strategy for a given XR instance and XR query. As ingredients to this optimizer, we plan to plug the query cardinality estimation components we have previously built and used in our prior works for conjunctive RDF queries [58] and conjunctive tree pattern queries [59].

In a recent work [60], we integrated the XR platform into a rich web browser interface, to enable scenarios such as those presented in the Introduction. We are currently working on an extension of the XR query language to enable it to *return XR instances* (as opposed to tuples of bindings as presented in this paper), continuing our first attempt in this direction [57]. With this language, closed under composition, we envision various new research directions, such as view composition, view-based query answering, as well as problems related to data exchange rules. An XR data instance, combined with a set of rules, would provide an elegant framework for XML-RDF data exchange, and permit querying intensional XML data, which is a little-studied problem.

We believe that in today's annotated, commented, shared, and fact-checked Web, annotated documents will be increasingly adopted. The purpose of this work was to set up a database foundation for expressively and efficiently exploiting such interconnected databases of structured documents, facts, and knowledge.

## References


1. Extensible Markup Language (XML) 1.0 (fifth edition). http://www.w3.org/TR/xml/, 2008.
2. RDF. http://www.w3.org/RDF/, 2004.
3. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004.
4. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/, 2001.
5. DBpedia 3.7. http://wiki.dbpedia.org/Downloads37.
6. F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from Wikipedia and WordNet. *J. Web Sem.*, 6(3), 2008.
7. S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping Between RDF and XML with XSPARQL. Technical report, DERI, 2011.
8. RDF concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004.
9. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
10. RDF Semantics. http://www.w3.org/TR/rdf-mt/, 2004.
11. OWL 2 web ontology language document overview. http://www.w3.org/TR/owl2-overview/.
12. S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD*, 2001.
13. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008.
14. A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.
15. A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.
16. XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/xpath-datamodel/, 2010.
17. xml:id. http://www.w3.org/TR/xml-id, 2005.
18. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, New York, NY, USA, 2002. ACM.
19. M. Rys. XML and relational database management systems: inside Microsoft SQL Server. In *SIGMOD*, pages 958–962, New York, NY, USA, 2005. ACM.
20. L. Chen, P. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a wide sparse table. In *ICDE*, pages 630 –641, April 2012.
21. L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *SIGMOD*, 1989.
22. L. Afanasiev and M. Marx. An analysis of XQuery benchmarks. *Inf. Syst.*, 33(2):155–181, 2008.
23. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2012.
24. L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: from Dewey to a fully dynamic XML labeling scheme. In *SIGMOD*, 2009.
25. B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.
26. K. Karanasos. *View-based techniques for the efficient management of Web Data*. PhD thesis, U. Paris Sud, 2012.
27. J. Hidders. Satisfiability of XPath Expressions. In *DBPL*, pages 21–36, 2003.
28. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
29. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.
30. M. Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *XSym*, 2005.
31. K. Karanasos, A. Katsifodimos, I. Manolescu, and S. Zoupanos. ViP2P: Efficient XML management in DHT networks. In *ICWE*, 2012.

32. I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.
33. G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, 1990.
34. 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 M. Brambilla, F. Casati, and S. Ceri, editors, *VLDS*, Seattle, United States, 2011.
35. Oracle Berkeley DB Java Edition. http://oracle.com/technetwork/database/berkeleydb/.
36. Online experiment site. http://tripleo.saclay.inria.fr/xr/experiments.
37. J. Kahan, M.-R. Koivunen, E. Prud'hommeaux, and R. R. Swick. Annotea: an open RDF infrastructure for shared Web annotations. *Computer Networks*, 39(5):589–608, 2002.
38. B. Haslhofer, R. Simon, R. Sanderson, and H. Van de Sompel. The Open Annotation Collaboration (OAC) Model. In *Multimedia on the Web (MMWeb), 2011 Workshop on*, pages 5 –9, sept. 2011.
39. S. Handschuh and S. Staab. Authoring and annotation of Web pages in CREAM. In *WWW*, 2002.
40. K.-P. Yee. CritLink: Advanced Hyperlinks Enable Public Annotation on the Web. In *Computer Supported Cooperative Work (CSCW)*, 2002.
41. S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. SemTag and seeker: bootstrapping the Semantic Web via automated semantic annotation. In *WWW*, 2003.
42. M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. MnM: Ontology Driven Semi-automatic and Automatic Support for Semantic Markup. In *EKAW*, 2002.
43. L. Reeve and H. Han. Survey of semantic annotation platforms. In *ACM SAC*, 2005.
44. S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos. WebContent: Efficient P2P Warehousing of Web Data (demonstration). *PVLDB*, 2008.
45. Microformats. http://microformats.org/.
46. RDF in HTML. http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml, 2006.
47. RDFa Primer. http://www.w3.org/TR/xhtml-rdfa-primer/, 2004.
48. K. Karanasos and S. Zoupanos. Viewing a world of annotations through AnnoVIP (demonstration). In *ICDE*, 2010.
49. GRDDL. http://www.w3.org/TR/grddl/, 2008.
50. W. Akhtar, J. Kopecký, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 432–447. Springer, 2008.
51. P. Patel-Schneider and J. Siméon. The Yin/Yang web: XML syntax and RDF semantics. In *WWW*, 2002.
52. J. Robie, L. M. Garshol, S. Newcomb, M. Biezunski, M. Fuchs, L. Miller, D. Brickley, V. Christophides, and G. Karvounarakis. The syntactic web. *Markup Lang.*, September 2001.
53. O. Corby, L. Kefi Khelif, H. Cherfi, F. Gandon, and K. Khelif. Querying the Semantic Web of Data using SPARQL, RDF and XML. Research Report RR-6847, INRIA, 2009.
54. T. Furche, F. Bry, and O. Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML in Web Querying. In *Principles and Practice of Semantic Web Reasoning*. Springer Berlin / Heidelberg, 2005.
55. M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler, and S. Zugal. Translating XPath queries into SPARQL queries. In *OTM*, 2007.
56. M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler, and S. Zugal. Bringing the XML and Semantic Web Worlds Closer: Transforming XML into RDF and Embedding XPath into SPARQL. In *Enterprise Information Systems*. Springer Berlin Heidelberg, 2009.
57. 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 *BDA (Informal proceedings)*, Rabat, Morocco, 2011.
58. F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View Selection in Semantic Web Databases. *PVLDB*, 5(2), Oct. 2011.
59. A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.
60. F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Fact-checking and analysing the Web (demonstration). In *SIGMOD*, New York, NY, USA, 2013.