Database techniques for semantics-rich semi-structured Web data

Julien Leblay

École Doctorale d'Informatique de Paris-Sud

Laboratoire de Recherche en Informatique

THÈSE DE DOCTORAT

présentée par : **Julien LEBLAY**

soutenue le : **27 septembre 2013**

pour obtenir le grade de : **Docteur de l'Université Paris-Sud**

Discipline / Spécialité : **Informatique / Bases de données**

Techniques d'optimisation pour des données semi-structurées du Web sémantique

THÈSE DIRIGÉE PAR

M. GOASDOUÉ François	*Univ. Paris-Sud*
Mme. MANOLESCU Ioana	*Inria Saclay*

RAPPORTEURS

M. AMANN Bernd	*Univ. Pierre et Marie Curie*
M. CERI Stefano	*Politecnico di Milano*

EXAMINATEURS

M. GROSS-AMBLARD David	*Univ. de Rennes 1*
Mme. FROIDEVAUX Christine	*Univ. Paris-Sud*

Résumé

RDF et SPARQL se sont imposés comme modèle de données et langage de requêtes standard pour décrire et interroger les données sur la Toile. D'importantes quantités de données RDF sont désormais disponibles, sous forme de jeux de données ou de méta-données pour des documents semi-structurés, en particulier XML. La coexistence et l'interdépendance grandissantes entre RDF et XML rendent de plus en plus pressant le besoin de représenter et interroger ces données conjointement. Bien que de nombreux travaux couvrent la production et la publication, manuelles ou automatiques, d'annotations pour données semi-structurées, peu de recherches ont été consacrées à l'*exploitation* de telles données.

Cette thèse pose les bases de la gestion de données hybrides XML-RDF. Nous présentons XR, un modèle de données accommodant l'aspect structurel d'XML et la sémantique de RDF. Le modèle est suffisamment général pour représenter des données indépendantes ou *interconnectées*, pour lesquelles chaque nœud XML est potentiellement une ressource RDF. Nous introduisons le langage XRQ, qui combine les principales caractéristiques des langages XQuery et SPARQL. Le langage permet d'interroger la structure des documents ainsi que la sémantique de leurs annotations, mais aussi de *produire* des données semi-structurées annotées.

Nous introduisons le problème de composition de requêtes dans le langage XRQ et étudions de manière exhaustive les techniques d'évaluation de requêtes possibles. Nous avons développé la plateforme XRP, implantant les algorithmes d'évaluation de requêtes dont nous comparons les performances expérimentalement. Nous présentons une application reposant sur cette plateforme pour l'annotation automatique et manuelle de pages trouvées sur la Toile. Enfin, nous présentons une technique pour l'inférence RDFS dans les systèmes de gestion de données RDF (et par extension XR).

Mots clés : Web sémantique, XML, RDF, Linked Data, modèles de données, langages de requêtes, composition de requêtes, réponse aux requêtes, optimisation de requêtes

Abstract

Since the beginning of the Semantic Web, RDF and SPARQL have become the standard data model and query language 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 documents, typically XML. The ability to apply RDF annotations over XML data emphasizes the need to represent and query data and metadata simultaneously. While significant efforts have been invested into producing and publishing annotations manually or automatically, little attention has been devoted to *exploiting* such data.

This thesis aims at setting database foundations for the management of hybrid XML-RDF data. We present a data model capturing the structural aspects of XML data and the semantics of RDF. 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. We also introduce the XRQ query language that combines features of both XQuery and SPARQL. XRQ not only allows querying the structure of documents and the semantics of their annotations, but also producing annotated semi-structured data on-the-fly.

We introduce the problem of query composition in XRQ, and exhaustively study query evaluation techniques for XR data to demonstrate the feasibility of this data management setting. We have developed an XR platform on top of well-known data management systems for XML and RDF. The platform features several query processing algorithms, whose performance is experimentally compared. We present an application built on top of the XR platform. The application provides manual and automatic annotation tools, and an interface to query annotated Web page and publicly available XML and RDF datasets concurrently. As a generalization of RDF and SPARQL, XR and XRQ enables RDFS-type of query answering. In this respect, we present a technique to support RDFS-entailments in RDF (and by extension XR) data management systems.

Keywords: Semantic Web, XML, RDF, Linked Data, data models, query languages, query composition, query answering, query optimization

Résumé de la thèse en français

Introduction

Depuis plus d'une décennie, le format XML [www08c] (*eXtensible Markup Language*) s'est imposé comme format de prédilection pour publier des données sur la Toile. L'écrasante majorité des pages Web modernes sont des fichiers XHTML, une des nombreuses incarnations du format XML. Il est aussi couramment utilisé pour l'échange de données sur les services Web (SOAP, XML-RPC), les flux de nouvelles (RSS, Atom), etc. Son succès est en partie dû à sa simplicité et à sa lisibilité. Un document XML peut être vu conceptuellement comme un arbre dont les nœuds sont étiquetés, ordonnés et d'arité non-bornée. Il existe plusieurs langages pour interroger les données XML, par parmi lesquels XPath et XQuery, qui sont recommandés par le W3C depuis 2007 [www10].

Développé à la même période qu'XML, le modèle de données RDF [www04b] (*Resource Description Framework*) est resté confidentiel à ses débuts. Il diffère d'XML en plusieurs points. Tout d'abord, une instance de données RDF est un graphe orienté, aux arêtes étiquetées et non-ordonnées, et dont les nœuds sont étiquetés différemment selon leurs types. Un nœud d'un graphe RDF correspondant généralement à une *ressource*, c'est-à-dire n'importe quel concept ou entité du monde réel. Une ressource peut être identifiée par une URI [1] (*Universal Resource Identifier*) ou être *anonyme*, auquel cas on parle de *nœud blanc* (en anglais *blank node*). Un nœud peut également être un littéral. Dans ce cas, il est étiqueté par une valeur, constante, éventuellement accompagnée d'un type ou d'une langue (pour les chaînes de caractères en langue naturelle). On peut également voir un jeu de données RDF comme un ensemble de *faits* de la forme (*sujet*, *prédicat*, *objet*).

Le modèle RDF est le substrat du *Web Sémantique*, un concept visant à permettre de publier sur la Toile, des données manipulables de façon non-ambigüe par des machines. Sa sémantique est définie en terme de règles d'inférences, permettant de dériver des nouveaux faits à partir de faits existants. Par exemple, les faits "Shakespeare est un écrivain" et "un écrivain est un homme" permettent d'inférer le fait "Shakespeare est un homme". Le modèle a connu un essor phénoménal depuis 2006 avec l'avénement du mouvement *Linked Open Data* [www06b], guidé par un ensemble de recommandations pour publier, lier et découvrir des données RDF de façon ouverte. Un grand nombre d'entreprises et

1. http ://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/

d'institutions mettent aujourd'hui à disposition du public d'importants jeux de données au format RDF [wwwh].

Peu de travaux ont, jusqu'à présent, tenté de réconcilier les modèles de données XML et RDF, à l'exception de quelques études visant à convertir les données d'un modèle vers l'autre. Les modèles RDF et XML étant particulièrement bien adaptés à des applications distinctes, nous postulons que de nouvelles solutions peuvent émerger pour des problèmes concrets, en particulier dans les domaines récents du *data journalisme* et de la *vérification de faits en ligne* (en anglais *online fact checking*), si l'on tire parti des spécificités de chacun de ces modèles, à savoir gérer et interroger conjointement la structure et la sémantique des documents sur le Web.

Cette thèse vise à rendre ce type d'application possible, en jetant les bases d'outils combinant des données XML et RDF en une instance commune. Nous proposons un modèle de données et son langage de requête, et présentons une plate-forme mettant en œuvre ces idées. Celle-ci nous permet d'étudier de manière exhaustive les stratégies d'évaluation et d'optimisation de requêtes rendues possibles lorsque des données XML et RDF sont mises en présence. Les principales contributions de cette thèse sont les suivantes :

- **un modèle de données pour documents XML annotés.** Alors que la plupart des travaux à ce jour se restreignent à représenter des données du modèle RDF au format XML ou *vice versa*, le modèle XR fournit le moyen de représenter des données XML et RDF *interconnectées*, c'est-à-dire des données pour lesquelles chaque nœud XML est potentiellement une ressource RDF ;
- **un langage de requête clos en terme de composition.** Nous introduisons un langage de requêtes permettant d'appliquer des contraintes *structurelle* et *sémantiques* sur des données. Le langage est présenté sous deux formes, (i) une forme simple permettant de retourner des ensembles de tuples, (ii) une forme étendue, close en terme de composition. En d'autres termes, le résultat d'une requête XR est une instance de données XR et peut à son tour être soumise à l'évaluation de requêtes ;
- **une étude exhaustive des stratégies d'évaluation et d'optimisation de requêtes.** Nous étudions en détail les techniques possibles pour évaluer efficacement des requêtes sur des données XR, en tenant compte, notamment des contraintes techniques imposées par les systèmes de gestion de données existants ;
- **une plateforme de gestion de données.** Cette séparation claire entre données XML et RDF dédouane l'administrateur de toute conversion, et permet de stocker les données dans des sous-systèmes distincts. Nous avons développé un système de stockage et d'interrogation de données pouvant être tirer partie de la plupart de systèmes de gestion de données XML et RDF existant ;
- **une étude expérimentale.** Nous présentons une série d'expériences comparant les performances de nos algorithmes d'évaluation et d'optimisation pour un large éventail de requêtes et pour des volumes de données jamais atteints à ce jour pour des données hybrides XML et RDF ;
- **un outil pour le *data journalisme*.** Le modèle de données XR fournit des bases solides pour un ensemble d'applications qui ont récemment fait leur apparition sur

la Toile, comme le data-journalisme. Pour démontrer l'utilité du modèle et de son langage pour ce type d'applications, nous avons développé FactMinder, un outil d'aide à la vérification de faits sur la Toile. L'outil se présente sous la forme d'une extension pour navigateur Web et repose sur la plateforme développée dans le cadre de nos travaux.

– **une approche alternative pour la réponse aux requêtes.** Comme évoqué précédemment, la sémantique d'RDF impose de tenir compte des faits implicites dans les réponses aux requêtes. Notre modèle de données étant un sur-ensemble du modèle RDF, il en préserve la sémantique. Toutefois, les techniques existantes pour la réponse aux requêtes RDF présentent des inconvénients. Nous introduisons une technique basée sur l'utilisation d'index bitmaps, qui minimise les problèmes inhérents à ces techniques ;

Cette thèse est structurée de la façon suivante. Dans un premier temps, nous dressons au chapitre 2 un état de l'art des standards XML, RDF et des travaux à leur jonction. Le chapitre 3 présente de façon formelle le modèle de données, le langage et en détail les propriétés. Le chapitre 4 étudie les techniques d'évaluation et d'optimisation de requêtes pour ce modèle, présente la plateforme de gestion de données, ainsi qu'une série d'expériences validant nos techniques. Le chapitre 5 présente notre outil d'aide à la vérification sur la Toile. Le chapitre 6 s'attarde sur la réponse aux requêtes RDFS. Enfin, le chapitre 7 clôt cette thèse en évoquant les directions vers lesquels ces travaux pourront mener.

État de l'art & motivation

Les recherches en gestion de données annotées s'articulent autour de deux axes : d'une part la conception d'outils pour l'annotation de données structurées, d'autre part l'étude de combinaisons des modèles XML et RDF.

Outils d'annotation pour les données du web. Dès les débuts du modèle RDF, plusieurs solutions ont été proposées pour faciliter l'annotation des pages web, que ces annotations soient créées manuellement par des utilisateurs [Yee02, HS02] ou de façon automatique et semi-automatique [VVMD+02, DEG+03]. à cet égard, [RH05] dressent un tour d'horizon des systèmes d'annotations mis au point à ce jour. Ces travaux sont principalement orientés vers le stockage et l'interrogation des documents annotés et ne tiennent pas compte des requêtes portant à la fois sur leurs structures et leurs sémantiques.

Concernant l'intégration entre annotations et documents, plusieurs recommandations ont récemment été faites pour la publication d'annotations RDF *au sein* de documents XHTML. Ces solutions, incluant microformat [2], eRDF [3] et RDFa [4] du W3C, n'apportent

2. http ://microformats.org/
3. http ://research.talis.com/2005/erdf/wiki/Main/RdfInHtml
4. http ://www.w3.org/TR/xhtml-rdfa-primer/

pas de solution au problème d'interrogation de données. En outre, elles supposent que l'utilisateur qui désire annoter un document ait accès en écriture à ce document, ce qui réduit considérablement leur utilité. Le modèle que nous proposons permet de créer et gérer des annotations sémantiques sans impact sur la structure ou le contenu du document.

Une seconde perspective sur ce thème couvre l'interconnexion entre les modèles de données XML et RDF.

Modèles de données hybrides. Dans ce contexte, la solution la plus courante, déjà mentionnée, consiste à convertir les données du modèle RDF au format XML pour en faire l'interrogation uniquement au moyen du langage XQuery, ou effectuer la transformation inverse pour mener l'interrogation à travers le langage SPARQL [RGN+01, PSS02, DFG+07]. Il a aussi été imaginé d'inclure des fonctionnalités d'un langage comme composants externes d'un autre [CKKC+09], par exemple une fonction d'extraction XPath accessible depuis un environnement SPARQL. Certains travaux proposent de modéliser plusieurs langages en un seul en partant d'un cadre plus général, comme les systèmes à base de règles [FBB05]. Enfin, des langages hybrides ont été conçus pour extraire des documents XML les annotations qui peuvent s'y trouver [AKKP08, BDK+11], comme décrit dans la recommandation GRDDL du W3C [5].

Bien que ces solutions permettent d'interroger conjointement des données XML et RDF, elles reposent généralement sur un principe de pré-traitement au cours duquel les données ou les requêtes doivent être converties vers l'un ou l'autre des modèles. Cette alternative comporte toutefois des inconvénients : (a) la réécriture peut aboutir à des requêtes complexes et difficilement optimisables, (b) la conversion des données est généralement coûteuse, qu'elle soit effectuée juste avant l'exécution des requêtes ou en amont sur l'ensemble des données, et (c) aucune des approches précédentes ne considère les nœuds XML comme des ressources RDF en tant que telles.

C'est pour pallier ces manques que nous avons orienté notre travail sur une solution qui respecte les formats dans lesquels les données sont initialement créées.

Exemple d'utilisation. Illustrons notre propos par une situation dans laquelle les annotations jouent un rôle central, par l'exemple suivant. Lors d'une campagne électorale, :Robert publie sur son site des transcriptions de ses discours, dans lesquelles il partage ses opinions sur la situation en :Turquie et au :Japon, citant notamment le :TauxDeChomageMensuel pour :Juillet2012 comme étant de 8% dans ces pays. En utilisant des jeux de données ouverts, tels que `http://data.gouv.fr`, il devient possible de vérifier semi-automatiquement les chiffres cités. De plus, en archivant les discours du candidat, il pourra déterminer, par exemple, "les dates de ses plus anciens et plus récents discours mentionnant un pays d'Asie" ou encore, si le calendrier officiel du candidat est pris en compte, "pour chaque pays où le candidat a effectué une visite, les citations

5. http ://www.w3.org/TR/grddl/

subséquentes dans lesquelles ce pays est évoqué". Si ce type de requêtes est trop ambigüe pour obtenir des résultats exploitables à partir de techniques basées sur le traitement des langues naturelles, elles peuvent être exprimées formellement dans un langage de requête tel que SPARQL, sous la forme d'une requête ou d'une composition de requêtes.

Gestion de données semi-structurées annotées

Le modèle de données XR

Le modèle de données XR est conçu pour représenter des documents annotés. Un des objectifs étant de préserver les propriétés des modèles de données standard XML et RDF, une instance de données XR comprend deux sous-instances : (a) une sous-instance XML, composée d'arbres XML, (b) et une sous-instance RDF, composée de triplets RDF. L'association entre les deux est assurée par l'attribution d'une URI unique à chaque nœud d'arbre XML. Ces URI peuvent ainsi apparaître dans des triplets RDF.

Formellement, considérons l'ensemble $\mathcal{U}$ des URI, et $\mathcal{L}$ l'ensemble des littéraux [www04b], i.e., pour simplifier l'ensemble des chaînes de caractères. $\mathcal{N}$ est l'ensemble des noms d'éléments et d'attributs XML possibles, auquel on ajoute le nom vide ϵ. Enfin, $\mathcal{B}$ est l'ensemble des *nœuds blancs*, i.e., des littéraux ou URI indéfinis comme nous le détaillerons par la suite.

Définition (Arbre XML)**.** *Un* arbre XML *est un arbre,* $T = (N, E)$*, fini, étiqueté, non ordonné, d'arité non bornée, où* N *sont les nœuds et* E *les arêtes, et à chaque nœud* $n \in N$ *sont associés une étiquette* $\lambda(n) \in \mathcal{N}$ *et un type* $\tau(n) \in \{document, attribute, element, text\}$.
Un nœud de type element *ne peut avoir deux attributs fils ayant le même nom.*
Un nœud de type attribute *est nécessairement le fils d'un nœud de type* element*, il comporte une valeur appartenant à l'ensemble des littéraux* $\mathcal{L}$ *et n'a pas d'enfant.*
Un nœud de type text *n'a pas d'enfant.*
Enfin, un arbre XML a au plus un nœud de type document*. Le nœud* document *est la racine de l'arbre, a exactement un fils et a pour étiquette* ϵ*.*

Définition (Instance XML)**.** *Une* instance XML I_X *est un ensemble fini d'arbres XML.*

Nous supposons maintenant l'existence d'une fonction $f_{uri} : N \to \mathcal{U}$ attribuant une URI unique à *chaque nœud* de l'instance XML. L'URI d'un nœud *document* est l'URI du document lui-même.

La fonction d'attribution des URI est centrale pour connecter les sous-instances XML et RDF. En effet, il est possible de faire référence dans la sous-instance RDF aux identifiants attribués aux nœuds de la sous-instance XML. Dans la section 4, nous présentons plusieurs approches possibles pour implémenter une telle fonction en pratique. En bref,

il suffit de tenir compte d'une fonction retournant une nouvelle valeur pour toute entrée présentée pour la première fois, et retournant cette même valeur à tout appel suivant pour cette entrée.

La sous-instance RDF est définie comme un ensemble de triplets, pouvant faire notamment référence aux URI des nœuds XML.

Définition (Instance RDF)**.** *Une* instance RDF I_R *est un ensemble de triplets de la forme* (s, p, o)*, où* $s \in (\mathcal{U} \cup \mathcal{B})$*,* $p \in \mathcal{U}$*, et* $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$*.*

Formellement, pour une instance RDF I_R donnée, sa sémantique est l'instance RDF I_R^∞, ou *clôture* de I_R, contenant I_R ainsi que tous les triplets pouvant être inférés à partir d'I_R et des règles d'inférences de la norme RDF. L'inférence RDF est cruciale pour l'évaluation de requêtes, puisque celle-ci doit tenir compte des triplets implicites de l'instance pour être complète. Nous abordons sur ce sujet en section 3.2.2 et au chapitre 6.

Nous pouvons maintenant définir une instance XR comme suit :

Définition (Instance XR)**.** *Une* instance XR *est un couple* (I_X, I_R)*, où* I_X *et* I_R *sont une sous-instance XML et une sous-instance RDF respectivement, construites sur le même ensemble d'URI.*

Les instances XML et RDF étant formées sur le même ensemble d'URI $\mathcal{U}$, les triplets RDF peuvent être utilisés pour annoter n'importe quel nœud XML. L'exemple suivant illustre cette interconnexion entre les deux sous-instances du système.

Le langage de requête XRQ

Les utilisateurs d'une instance XR doivent pouvoir interroger les données sur leur structure (décrite dans l'instance XML) ainsi que sur leur sémantique (décrite dans l'instance RDF). C'est l'objectif d'XRQ, un langage permettant d'accéder aux données selon ces deux perspectives. Dans la section 3.2, nous établissons tout d'abord la syntaxe du langage, avant d'en détailler la sémantique dans la section 3.2.2. Enfin, les sections 3.2.3 et 3.2.4 présentent une extension du langage qui fournit les moyens de construire des résultats complexes, produisant une instance XR et faisant ainsi de XRQ un langage clos.

À l'image du modèle de données, une requête XRQ est composée de deux parties : un ensemble de *motifs d'arbres* permettant de filtrer les données XML, et un ensemble de *motifs de triplets* portant sur les données RDF. Nous définissons ces deux types de motifs ci-après. Notons tout de suite que des variables utilisées dans des motifs d'arbres peuvent aussi être utilisées dans des motifs de triplets, ce qui rend possible l'interrogation dépendante de la structure et des annotations sémantiques.

Définition (Motif d'arbre)**.** *Un motif d'arbre est un arbre fini, ordonné, d'arité non bornée et* $\mathcal{N}$*-étiqueté, comportant deux types d'arêtes : des arêtes* enfant *et des arêtes* descendant*. À chaque nœud de l'arbre peuvent être adjointes au plus une variable* uri*, une variable*

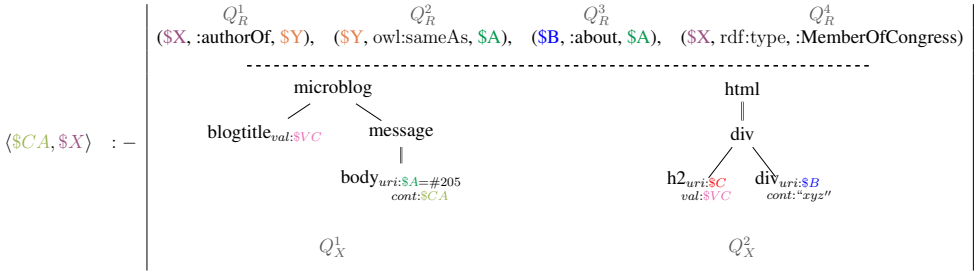


FIGURE 1 – Exemple de requête XRQ

val *et une variable* cont. *Un nœud peut être également étiqueté par un prédicat d'égalité de la forme* [t=c] *où* $c \in \mathcal{L}$*, et* t *est une* $type$ *parmi* $\{uri, val, cont\}$.

Un motif d'arbre est semblable au motif d'arbre décrit dans la littérature [AYCLS01] à une différence près, le fait de pouvoir adjoindre des variables *typées* aux nœuds. Ces variables ont deux objectifs : (i) indiquer les jointures entre motifs d'arbres (ou de triplets) (ii) indiquer quels éléments de la requête feront partie du résultat (à la manière des requêtes conjonctives). Le type désigne quelle information du nœud XML sera affectée à la variable. Lorsqu'un nœud n_t d'un motif d'arbre est mis en correspondance avec un nœud n_d d'un arbre XML, la variable adjointe à n_t recevra la valeur suivante en fonction de son type : pour une variable *uri*, l'URI du nœud n_d ; pour une variable *val*, la concaténation des valeurs de tous les descendants de n_d si n_d est un élément, ou la valeur de n_d si c'est un attribut ; pour une variable *cont*, l'arbre enraciné à n_d sous forme sérialisée.

Définition (Motif de triplet)**.** *Un motif de triplet est un triplet de la forme* (s, p, o)*, où* s, p *sont des URI ou des variables, et* o *est une URI, un littéral ou une variable.*

En combinant motifs d'arbres et motifs de triplets et en leur ajoutant un ensemble de variables projetées (variables de tête), on obtient une requête XRQ.

Définition (Requête XRQ)**.** *Une requête XRQ est composée d'une* tête *et d'un* corps. *Le corps est un ensemble de motifs d'arbres et de motifs de triplets, construits sur le même ensemble de variables. La tête est formée d'une liste de variables apparaissant aussi dans le corps.*

Les jointures sont formulées en utilisant une même variable à plusieurs reprises dans la requête. Trois types de jointures sont donc possibles : entre motifs d'arbres, entre motifs de triplets et entre motifs d'arbres et de triplets. C'est ce dernier type de jointure qui offre au langage XRQ sa capacité à marier les données XML et RDF.

Le langage XRQ étendu. Une requête XRQ retourne un ensemble de tuples, bien qu'elle prenne en entrée une instance XR. Idéalement, il devrait être possible de produire une instance XR comme résultat d'une requête. Nous nous proposons d'étendre le

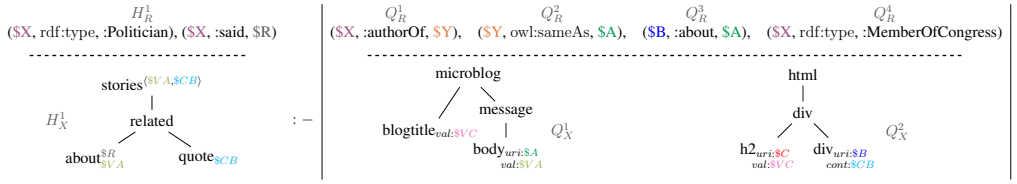


FIGURE 2 – Exemple de requête XRQ étendue

langage XRQ pour atteindre ce but en adjoignant au langage un constructeur permettant de créer de nouveaux arbres et triplets. La suite de cette section donne la définition et la sémantique de cette extension.

Définition (Requête XRQ étendue)**.**
Une requête XRQ étendue, dénotée $Q = (H_X, H_R, Q_X, Q_R, Sk)$*, consiste en un corps* (Q_X, Q_R)*, semblable à celui d'une requête XRQ simple, une tête de la forme* (H_X, H_R)*, où* H_X *est un ensemble de* patrons d'arbres XML *et* H_R *est un ensemble de triplets, et une application bijective* $Sk : H_X \to \mathcal{S}$*, où S est un ensemble infini de fonctions de Skolem.*

Soit V_Q *l'ensemble des variables apparaissant dans le corps de la requête,* V_H *l'ensemble de variables apparaissant uniquement dans la tête, et* $V = V_Q \cup V_H$*. Pour chaque arbre* $t_x \in H_X$*, chaque nœud* $n_x \in t_x$ *peut être annoté de trois façons : (i) une* étiquette d'affectation *est une variable* $v \in V_H$ *to type* uri*, (ii) une* étiquette de valeur *est une variable* $v \in V_Q$ *ou une constante, et ne peut être appliqué qu'à une feuille, (iii) une* étiquette de groupe *est une liste ordonnée de constantes ou de variables de* V_Q*, telle qu'elle ne contienne aucune variable présente dans une étiquette de groupe d'un ancêtre du nœud courant.*

Les triplets $t_R \in S_R$ *peuvent contenir des variables de* V *dans n'importe quelle position (s, p ou o).*

La sémantique du langage dans sa forme simple et étendue est détaillée dans la section 3.2.2.

Composition. Le language étendue permet notamment la composition de requêtes, c'est-à-dire l'évaluation d'une requête à partir du résultat d'une autre requête (ou vue). Nous présentons un algorithme qui construit, pour une requête q et une vue v passées en paramètres, une nouvelle requête q' telle que pour toute instance $\mathcal{I}$, $q'(\mathcal{I})$ est correcte vis-à-vis de $(q \circ v)(\mathcal{I})$.

La plate-forme XRP

Le modèle XR étant compatible avec les standards XML et RDF, on peut envisager de stocker des données XR dans une plateforme *native*, ou s'appuyer sur des logiciels de gestion de données XML et RDF existants, à la manière d'un intégrateur de données. Dans

ce cas, le système de gestion de données XML doit être *augmenté* de manière à permettre de la gestion d'URI de nœuds. Il est parfois possible de modifier directement le système pour que celui-ci puisse interpréter ces URIs. Mais, lorsque le système est entièrement fermé, il devient nécessaire d'avoir recours à une structure externe, par exemple un index, pour stocker la correspondance entre URI et nœuds XML.

En tenant de tous ces cas de figures, nous avons établi une hiérarchie de techniques utilisables pour évaluer des requêtes XR sur des systèmes existants. Celle-ci compte notamment des stratégies de passage d'information horizontale, évaluant les motifs de triplets (resp. d'arbre) en priorité, avant de transmettre et lier les variables des motifs d'arbres (resp. triplets) restants pour les rendre plus sélectifs. Ces techniques ouvrent la voie à certaines optimisations. Il devient par exemple possible d'éliminer certains candidats avant de les transmettre, s'il est évident qu'ils rendront les prochaines requêtes insatisfiables.

Pour mettre en pratique cette famille d'algorithmes d'évaluation, nous avons développé la plateforme XR, un moteur de stockage et d'évaluation de requêtes complets, pouvant reposer sur n'importe quel système de gestion de données XML ou RDF existant. La plateforme comporte un optimiseur pour choisir, lors de l'évaluation de requêtes, parmi différentes stratégies d'évaluations. Le moteur possède un ensemble d'opérateurs physiques ; il peut notamment déléguer certaines sous-parties d'une requête aux moteurs d'évaluations sous-jacents et compléter l'évaluation de lui même.

Le modèle XR, le langage XRQ et la plateforme XRP ont fait l'objet d'articles dans une conférence nationale [GKK+11b], un workshop international [GKK+11a] et des revues nationale [GKK+12] et internationale [GKK+13b].

Analyse et vérification de faits sur la Toile

L'Internet a remodelé le journalisme de nombreuses manières, en ouvrant les vannes d'une dissémination d'information à grande échelle. Les professionnels de l'information se sont soudain vus entrer en compétition avec de nouveaux acteurs, pour la plupart amateurs (activistes, blogueurs ou simples citoyens engagés) qui se sont imposés comme sources d'informations alternatives aux médias établis. La force de ce mouvement vient pour part du nombre des parties en présence, lui permettant, collectivement, d'accumuler, traiter et publier une quantité d'information beaucoup plus importante qu'un journal ou une agence de presse ne pourrait espérer manipuler. Toutefois, quantité n'est pas synonyme de qualité et ce phénomène a donné naissance à une nouvelle catégorie de journalistes. Les *data-journalistes* et *vérificateurs de faits*, parfois aussi qualifiés de *désintoxicateurs*[6], sont spécialisés dans l'analyse de faits publiés sur la Toile. Si la vérification des sources fait partie d'intégrante du métier de journaliste, la nouveauté vient surtout des méthodes et des moyens dont ceux-ci disposent aujourd'hui pour mener à bien leur tâche.

6. `http://www.liberation.fr/desintox`

Ces outils sont les services en ligne (tels que Twitter [7] ou Google Maps [8]), mais surtout les jeux de données publiques mis à disposition par des gouvernements [9] [10], organismes non-gouvernementaux [11] et entreprises privées [12]. Ces métiers n'est restent pas moins largement manuels [13]. L'écrasante majorité des informations publiées sur la Toile étant aux formats XML ou RDF, nous postulons qu'XRP représente une plateforme de choix pour assister ce type d'utilisateurs dans leurs tâches.

Nous présentons le logiciel FactMinder, un outil d'aide à l'analyse et la vérification de faits sur la Toile. L'application est une extension pour navigateur connectée à une plateforme XRP. Lors de l'activation de l'extension, l'écran du navigateur se scinde en deux parties verticales. Lorsqu'un utilisateur accède à une page Web (partie gauche), celle-ci est transmise à un système d'extraction d'entités nommées (dans notre cas, OpenCalais [wwwi]). Ces entités, identifiant personnes, lieux, institutions, dates et citations, sont intégrées à la page même. La partie droite du navigateur renferme un tableau de bord, permettant de visualiser des informations liées à celles de la page. Le tableau de bord est entièrement composé de *vues XRQ*, ou XIP (*XR Information Panel*), à savoir des requêtes XR nommées. Ces vues sont évaluées sur la plateforme XR, sur laquelle sont stockées des données concernant les pages précédemment visitées, mais aussi un ensemble de jeux de données ouvertes ou collectées sur la Toile. Les vues sont rafraîchies lorsque certains événements-utilisateurs se produisent, par exemple, lorsqu'il sélectionne un item à l'écran. Des dépendances peuvent être imposées entre les vues, de manière à ce que celles-ci ne soient évaluées qu'à la condition que des informations supplémentaires soient fournies.

Ce logiciel a été présenté dans une démonstration lors de la conférence SIGMOD 2013 [GKK+13a].

Réponse aux requêtes RDF à l'aide d'indexes bitmaps

Répondre aux requêtes SPARQL suppose de tenir compte de règles d'inférences comme celles présentées en introduction. Les méthodes généralement utilisées à cet effet sont appelées *chaînage avant* et *chaînage arrière*. La première consiste à dériver exhaustivement tous les faits possibles à partir de l'extension des données et des règles considérées. Ceci permet d'obtenir la *clôture* des données, que l'on peut ensuite matérialiser avec les données initiales. Dans ce cas, le problème de réponse aux requêtes est réduit à celui de l'évaluation de requêtes. L'inconvénient majeur de la méthode réside dans le fait qu'elle

7. `http://twitter.com`
8. `http://maps.google.com`
9. `http://data.gov`
10. `http://data.gouv.fr`
11. `http://data.worldbank.org`
12. `http://google.com/publicdata/home`
13. `http://on.ted.com/MarkhamNolan`

nécessite un espace de stockage potentiellement important. De plus, lorsque les données sont dynamiques, leur consistance doit être maintenue. La seconde méthode n'affecte pas les données, mais consiste à réécrire une requête conjonctive donnée en une union de requêtes conjonctives dont le résultat, pour une instance $\mathcal{I}$, sera équivalent à celui de la requête initiale, si elle était évaluée sur la clôture de $\mathcal{I}$. L'union de requêtes créée est exponentielle dans la taille de la requête initiale, et de ce fait, difficile à optimiser en pratique.

Dans cette thèse, nous proposons une méthode de stockage des données et d'évaluation de requêtes permettant des minimiser les problèmes liées à ces deux approches. En observant la clôture RDFS d'un jeu de données, on remarque que le processus d'inférence produit des résultats prévisibles. Ainsi, une ressource appartenant à une classe donnée appartient aussi à toutes ses super-classes. De même, deux ressources liées par une propriété sont aussi liées par toutes ses super-propriétés. L'idée principale est de raisonner en terme d'appartenance à des ensembles. Par exemple, détecter le plus rapidement possible si une classe donnée appartient à l'ensemble des super-classes d'une ressource donnée. Les opérations ensemblistes sont couramment implantées en terme d'opérations bit-à-bit. Il suffit pour cela de fixer un ordre pour les éléments de l'ensemble considéré. Chacun de ses sous-ensembles peut être représenté par un mot dont les bits sont placés à un pour chaque index d'éléments appartenant au sous-ensemble. Hormis la rapidité évidente des opérations ensemblistes sur des données binaires (par exemple l'intersection est obtenue par un ET logique, l'union par un OU), de nombreuses techniques de compression existent, y compris des méthodes permettant d'effectuer ses opérations sur les données sans le décompresser.

Nous présentons un modèle de stockage de données RDF dans lequel l'ensemble des classes auxquelles appartiennent une ressource sont stockées sous forme de mots binaires compressés. Grâce à cette méthode, matérialiser la clôture des données, obtenue par chaînage avant, n'entraîne qu'une faible augmentation de la taille de stockage requise. De plus, il reste possible de ne pas saturer les données, et d'avoir recours à une technique proche de celle du chaînage arrière. Celle-ci ressemble de près au concept d'index sémantique proposée dans des travaux précédents [MAYU05, RMC11]. Toutefois, dans notre cas, les indexes sont des indexes bitmaps, qui présentent divers avantages sur les techniques existantes. Tout d'abord, ces indexes, efficacement compressés requièrent peu d'espace et peuvent notamment être stockés en mémoire vive. Ensuite, ils sont plus faciles à maintenir en cas de mise à jour de la hiérarchie de classes et propriétés.

Nous présentons notre approche en détail, en expliquant notamment comment un système de gestion de données classique peut tirer partie de ce modèle de stockage, et nous présentons des résultats expérimentaux préliminaires qui en démontrant l'intérêt.

Cette approche a fait l'objet d'un article publié dans un workshop international [Leb12].

Conclusion

La nécessité de gérer efficacement des informations structurées et annotées sémantiquement se fait de plus en plus pressante. Dans cette étude, nous jetons les bases d'un modèle de données et d'un langage de requêtes pour représenter et interroger des documents d'après leurs structures et la sémantique de leurs annotations. Nous étudions en profondeur les techniques d'évaluation et d'optimisation de requêtes possibles tirant partie d'application existante. Nous avons développé une plateforme complète pour le stockage et l'interrogation de données XR. Nous avons mis en place un ensemble d'expériences pour démontrer l'efficacité et la pertinence de notre approche. Nous présentons FactMinder, un logiciel pour l'aide à la vérification de faits en ligne. Enfin, nous introduisons une technique alternative pour le réponse aux requêtes RDF.

Nous comptons prolonger ce travail selon les axes suivants : (i) étendre nos travaux sur la composition, en introduisant des optimisations (élagage et minimisation) dans l'algorithme de composition et en considérant le problème dans un contexte à plusieurs vues, (ii) introduit un modèle de coût pour automatiser l'optimisation de requêtes, (iii) étendre notre langage de requêtes, notamment en considérant des prédicats plus riches (négations, inégalités), des opérations d'agrégation, et enfin (iv) étudier les interactions sociales entre utilisateurs dans un contexte où les annotations peuvent être partagées et échangées.

Contents

Chapter 1

Introduction

When the World Wide Web emerged in the early nineties, few people could have guessed how fast it would come to be both ubiquitous and indispensable. In three decades, it has become an invaluable part of our private and public lives. The so-called *Big Data* era was spawned by the combined improvements of data collection and storage capacities (e.g., sensors, portable devices), of means of publications (e.g., wikis, social networks) and by the growing amount of people who have access to the Internet. Finding ways to reach out for relevant information in the plethora of data is the center of the data management community.

1.1 From Web 1.0 to Web 3.0

Originally, most documents published on the Internet complied to the markup language HTML. What was retrospectively named the Web 1.0 was mainly a large set of unstructured documents interconnected through hyperlinks. Seldom Internet users had the technical tools and knowledge to publish simple documents on the Web, let alone structured data.

In 1998, the W3C published the XML recommendation [www08c], a format for publishing data in a more principled manner than HTML. XML quickly became the substrate of the Web 2.0, the Web of wikis and social networks, where *exchanging* data is within reach of anyone. Its best-known incarnation, XHTML, is the format of virtually every modern Web pages. Under other forms, it is also prevalent in data exchange, such as Web services (SOAP), remote procedure calls (XML-RPC), news feeds (RSS/Atom), and so on. The Web 2.0 was still human-centric, and a large part of the information contained in Web pages remained opaque to computer programs.

In 2004, the W3C published the current RDF recommendation [www04b] as a means to enable a more machine-processable "Semantic Web". Briefly, RDF allows representing knowledge as a graph, where nodes are *resources* or *literal values* and edges are *predicates*

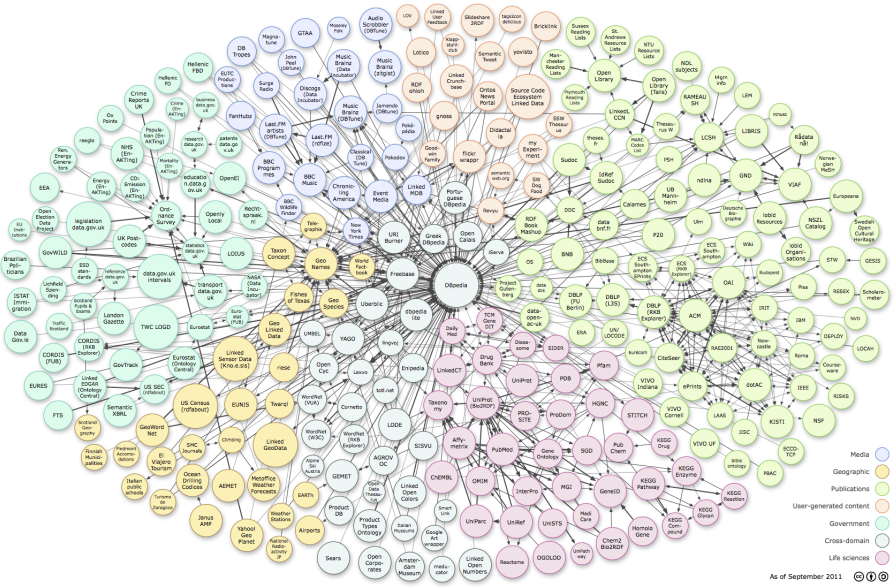


Figure 1.1: The Linked Open Data cloud as of September 2011

indicating relationships between resources or assigning values to resources. Each pair of nodes linked by an edge, can be seen as a statement (a.k.a. fact) of the form "Subject Predicate Object", the basic information unit of RDF. An essential feature to RDF is its semantics. In short, some RDF statements can be used to described relationships among concepts, to minimize the amount of data that is explicitly declared. For instance, it is possible to derive the fact "Shakespeare is a person", from two other facts: "A writer is a person" and "Shakespeare is a writer".

RDF started to take off from 2006 thanks to the Linked Data principles [www06b], a set of simple rules proposed by Tim Berners-Lee, to encourage the publication, consumption and discovery of data on the Web. Since then, a growing number of institutions, governments and companies have adopted these practices and made data available to users online. Data sets covering knowledge in a wide variety of domains (science, culture heritage, public services, etc.) have already been published. They also generally link to one another. The diagram [1] depicted on Figure 1.1 shows a subset of publicly available RDF data sets listed by the nonprofit organization Open Knowledge Foundation [wwwh] as of 2011. It estimated then that 31 billion statements were publicly available.

1.2 Motivation: Structure versus semantics

In some respect, XML and RDF could both be considered "success stories", yet, they are still the center of attention of the Web data management community. Data on the Web

1. `http://lod-cloud.net/state/`

is inherently distributed, inconsistent and incomplete, and many problems pertaining to XML and RDF are still, in isolation, keeping researchers busy. XML and RDF have some fundamental differences, which push people to use them with distinct purposes in mind. XML relies an unranked, node-labeled, ordered tree model. It is commonly used to give *structure* to data, possibly constrained by some schema (DTD or XML Schema). RDF is an unordered, node- and edge-labeled graph data model. It is particularly convenient for representing *knowledge* or data that does not conform to a particular schema. Little has been done in the way of considering the management of *combined* XML and RDF data. This, however, would have the following advantages:

- The amount of data available in either formats is so large that converting data into a single format is not feasible in practice. Because of the differences previously mentioned, translating XML data to RDF (or the opposite) generally leads to overly complex data instances and queries.
- By considering XML and RDF data *together*, some properties specific to one model could carry over to the other. For instance, the reasoning capabilities offered by RDF could be extended to XML data management.
- The concurrent management of XML and RDF would also bring about interesting opportunities for query optimization, which deserve to be thoroughly explored.
- Beyond the traditional usages of XML and RDF, such as *data exchange* and *knowledge representation*, there are emerging applications, such as *fact checking* and *data journalism*, which require solid database foundations. We believe that looking at both models through a common lens would support such applications in an elegant way.

Below we present three scenarios that further highlight the interest of combining XML with RDF data.

Scenario 1: semi-automated fact-checker. 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%". 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 feeds, 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 feeds 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's 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.

This work aims at enabling such requirements, by proposing a unified model allowing the combination of XML data with RDF data into a single instance.

1.3 Contributions and outline

This thesis addresses the management of mixed XML and RDF data and query answering for RDF data. Since these two aspects are orthogonal, the solutions introduced for the latter can in practice easily be applied to the former. The remaining chapters and their respective contributions are described below:

Chapter 2 introduces XML, RDF and the approaches that have been proposed so far to efficiently store and query data in these two models. At the end of this chapter, we present prior works that considered such a setting in the past and in various contexts.

Chapter 3 formally defines a data model and query language for managing data at the junction of XML and RDF.

- We introduce XR, a model for representing interconnected XML and RDF data.
- We present XRQ, a query language, which comes in two version: a core language returning tuples and an extended language producing XR data.
- We detail the syntaxes and semantics of the core and extended versions of the language.
- We provide an algorithm for composing XRQ queries over views.

Chapter 4 presents the XR platform, and explores the space of query evaluation and optimization strategies.

- We study the problems of evaluating and optimizing XRQ queries under a wide range of storage systems.
- We present XRP, a full-fledged query evaluation engine that can run on the top of virtually any existing XML and RDF data management system.
- We present an experimentation campaign for all the evaluation strategies and optimizations previously introduced.

Chapters 3 and 4 were the main topic of four papers [GKK+11b, GKK+11a, GKK+12, GKK+13b] published respectively in a national conference, an international workshop, a national and an international journals.

Chapter 5 describes FactMinder, a W3C-standards compliant, rich browser interface aimed at data journalism and online fact checkers. The system, implemented as a browser extension, brings together the techniques presented in this thesis. This work was presented as a demonstration [GKK+13a] in an international conference.

Chapter 6 focuses on efficient *RDF query answering*, i.e., query evaluation under RDFS entailments, with an approach based on bitmap indexes. In particular:

- We introduce a storage model for RDF that minimizes space requirements over classic approaches, especially in cases when saturation is used. The model can conveniently be used on top of existing storage systems.
- We detail how query evaluation can be done under this model.
- We present a novel index that compiles the semantics of an RDF graph into a compressed structure, removing the need for saturation and show how queries are evaluated in the presence of such indexes.
- We provide experimental evidence that the storage model and the index will be valuable instruments in the toolkit of RDF management system implementors.

This work was the topic of a short paper [Leb12] published in an international workshop.

Chapter 7 closes this thesis with a summary and a discussion on the directions in which these topics could be further explored.

Chapter 2

Background and state-of-the-art

This chapter describes XML and RDF in isolation and previous works that have studied the interaction thereof.

More precisely, in Sections 2.1 and 2.2, the data models and query languages associated with XML and RDF are respectively detailed. We present well-known complexity results and a number of storage models that have been proposed for both formats. In Section 2.3, we explore prior works that have considered the combined management of XML and RDF data.

2.1 XML data management

The W3C introduced the XML recommendation in 1998 [www98], which has reach its fifth edition in 2008 [www08c]. It was originally derived from SGML [1], an ISO-standard markup language, and as such, one of its primary goals was to enforce a clean separation between data and the way it should be processed.

Although it is best known for publishing documents on the Web, it is also commonly used in data exchange, for example as a serialization format for other data models, including RDF. Arguably, its success is partly due to its simplicity and flexibility. XML files are, to a certain extent, easy to write and read for a human. Optionally, one can attach a so-called DTD (Document Type Definition) or a Schema to an XML file to limit its vocabularies, constrain its structure or enforce typing.

In this section, we introduce the XML data model, as well as the query languages associated with it. We provide some formal results pertaining to query evaluation.

1. ISO-8879

2.1.1 Data model and query languages

Multiple data models and query languages based on XML have been proposed such as XML-QL [ADMFDF+98], XQL [IKKN99] relying on nested query syntaxes, or some more user-friendly graphical languages, such as XML-GL [CCD+99], XGL [FFG02] and XQBE [BCC05].

Yet, the XQuery and XPath data model [www10], a W3C recommendation, is the *de facto* standard for managing XML data. The notions of *tree* and *sequence* are at the core of the model. The *nodes* of a tree can be one of seven types: document, element, attribute, text, namespace, processing instruction, and comment. Without loss of generality, in this thesis, we focus on the first four types. An XML document is a node-labeled, unranked, ordered tree, rooted at a *document* node. *Elements* can have children of any types other than *document*. *Attributes* may feature at most a single *text* node as a child. *Text* nodes are necessarily leaves. Figure 2.1 depicts a sample XML document in its serialized form (top) and as a conceptual tree (bottom). The document represents information about the book "Principles of Database" by Jeffrey D. Ullman of Stanford University, published in 1980.

As its name suggests, *XPath* is a language for selecting nodes in XML documents based on their paths. The initial version of the language, XPath 1.0, allows navigating along *axes* such as *child*, *descendant*, *previous* and *next siblings*. It also allows some boolean predicates on nodes along a path. As an example, the query $//author/name$ evaluated on the document of Figure 2.1, where "/" and "//" represent child and descendant axes respectively, returns the node labeled "name". The combined complexity, i.e., when both the input query and data vary, for XPath 1.0 is in PTIME [BK08]. The current XPath 2.0 extends its predecessor with set operators and iteration capabilities.

Although XPath can be used as stand-alone language, it is also a component of the more powerful *XQuery* language. XQuery queries are nicknamed FLOWR expressions, an acronym gathering initials of the language's most important key words. Among them, the *for* and *let* key words let one iterate over sequences and assign variables. The *where* clauses allows one to impose constraints on those variables. The *result* defines how the result of the query should be constructed. Queries can be nested, and functions (both built-in and user-defined) can be used within the language. XQuery is a Turing-complete functional language. The expressivity and complexity of various of its fragments are studied in [BK09]. In this thesis, we also use the Unified Data Model [MPV09], an extension of XQuery that allows sets, bags and lists of tuples.

2.1.2 Storing XML data

When a new data model emerges, researchers are first tempted to adapt it to the well-established relational databases. This was also true for XML, where a host of works have explored the ways to split XML documents to store them in relational tables [STZ+99,

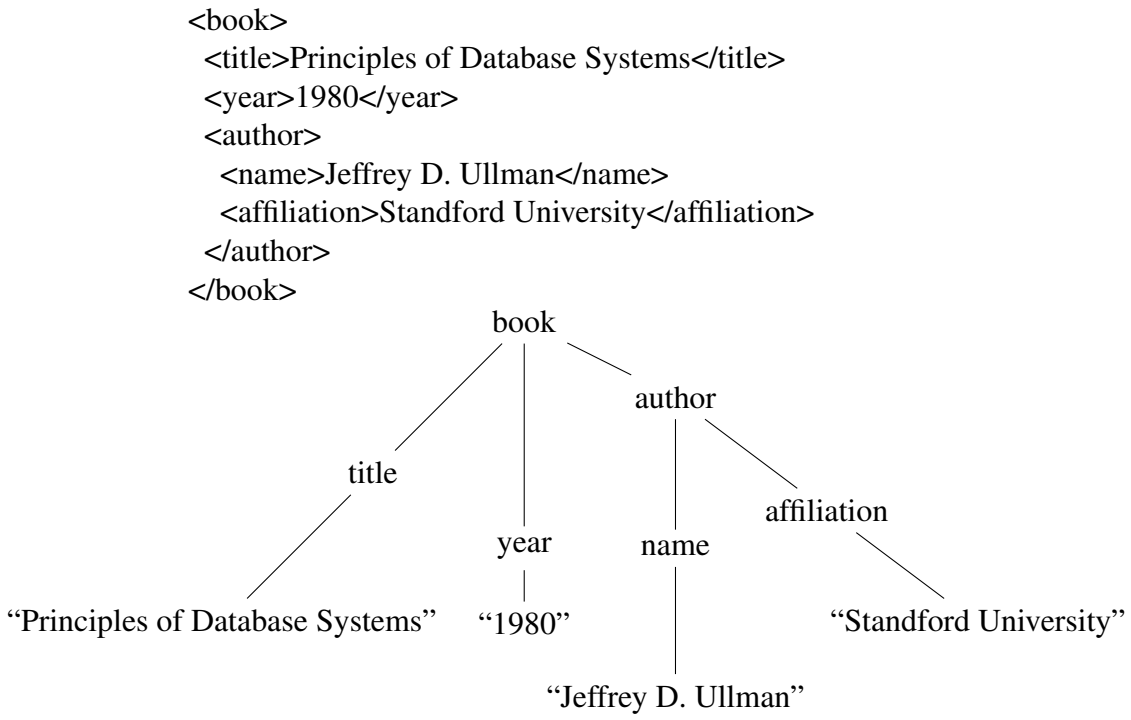


Figure 2.1: Example of an XML document, serialized (top) and conceptual (bottom)

DFS99, YASU01, TVB+02]. These approaches use either DTDs, query workloads or simply XML data as input information to shred the data into sets of tuples. Queries are then translated into SQL, to be optimized and evaluated as in a conventional relational setting. Also originally a relational database, the column-store MonetDB was modified to support XQuery evaluations [BGvK+06]. Along the same line, most commercial RDBMS systems, such as IBM DB2 [BCH+06], Microsoft SQL Server [PCS+04] and Oracle [LM09b] have implemented extensions to support XML, XPath and XQuery.

Another set of XML-relational systems, that could be called XML mediators, includes SilkRoute [FTS00], Xperanto [CKS+00] and XTables [FKS+02]. In those systems, data that originally resides in relational tables is published as XML *views*, allowing to query these tables with a XML query language such as XQuery. As with previously cited works, queries are ultimately rewritten into SQL queries.

Native XML storage systems have also been proposed, among which Natix [FHK+03] is one of the first. BaseX [wwwa] and eXist [Mei03] are open source systems that have been extensively used in research.

Because XML data is pervasive and distributed in nature, distributed XML management techniques have also been introduced, such as peer-to-peer systems [BC06, AMP+08, MKK08, KKMZ12] and more recently Cloud-based systems [KCS11, CRCM12]. These works explore problems such as data placement, distributed indexing, and cost amortization.

Storing node metadata. A number of XPath/XQuery operators allow to check certain relationships between nodes, for instance, whether a node a *follows* b or is an ancestor thereof. Such relationships are also useful when XML documents are shred into smaller pieces, e.g., in a materialized view-based system such as ViP2P [KKMZ12]), and must be combined back through *structural joins* at query time. A common practice, first advocated in [YASU01], is to label node with prefix, postfix and depth positions obtained by a single depth-first traversal of the document. This technique however is not sufficient when one need to list the children of a node in order. A more fine-grained approach, Extended Dewey IDs [LLCC05], labels each node with the positions of each of its ancestors among siblings. A variant named Dynamic Dewey ID [XLWB09] makes the labeling amenable to updates.

Those techniques are also useful to determine node identity, i.e., whether two nodes are the same regardless of their serialization. This will be further discussed in Chapter 4.

2.2 RDF data management

The Resource Description Framework (RDF) [www04b] is a W3C recommendation for representing and interchanging data on the Web. In recent years, notably with the advent of the Linked Data movement, RDF has imposed itself as a prominent format, with thousands of data sets publicly available and linking with one another [wwwh].

An RDF data instance can be seen a directed, node- and edge-labeled graph. It is not required to comply with a schema. However, some edges of an RDF graph may belong to a predefined vocabulary, allowing to *derive* additional knowledge from the graph and a set of simple rules.

In section, we formally define the RDF data model and the conjunctive fragment of SPARQL, the principle language for querying RDF. Next, we present various ways in which RDF can be stored. Finally, we provide some details on SPARQL query evaluation and query answering under RDFS entailment.

2.2.1 The data model

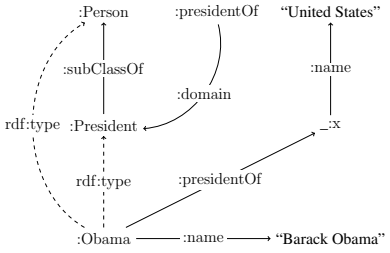
Let $\mathcal{U}$ be the set of URIs as defined in [www01], and $\mathcal{L}$ the set of literals. Literals are terminal values used to assign properties to resources. Although literals can be been given a language or a type, for simplicity, we can see $\mathcal{L}$ as the set of all strings. Finally, $\mathcal{B}$ is the set of blank nodes (accounting for unknown resources).

Definition 2.2.1 (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* $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$*.*

The components of a triple $(s,\ p,\ o)$ are usually referred to as *subject*, *property* (or *predicate*) and *object*, respectively.

{(:Obama, :presidentOf, _:x)
(:Obama, :name, "*Barack Obama*")
(_:x, :name, "*United States*")
(:presidentOf, rdfs:domain, :President)
(:President, rdfs:subClassOf, :Person)
(:Obama, rdf:type, :President)
(:Obama, rdf:type, :Person)}

(a)



(b)

Figure 2.2: Example of an RDF data set, as a set of triples (a) and as a graph (b)

RDF is also commonly represented a graph, as shown in Figure 2.2, where the set of triples in (a) is represented and as a graph in (b). Note that here and subsequently in this work, we make the convention that strings starting with : are URIs. Formally, URIs consist of two parts: a namespace, and a local name, separated by the : symbol. A URI without a specified namespace is of the form :LocalName, and is interpreted to refer to a default namespace. We also follow the usual convention of denoting blank nodes by _:-prefixed names.

As defined above, the subject or the object of the triple can be bound to a so-called *blank node*, i.e., *unknown URIs or literals*, similarly to *labeled nulls* in the database literature [AHV95]. For instance, in Figure 2.2, the blank node _:x in the triple (_:x, :name, "*United States*") to state that the *name* of the resource _:x is "United States", without using a concrete URI.

2.2.1.1 RDF Semantics

The W3C's RDF recommendation comes with a *vocabulary*, RDF Schema, i.e., a predefined set of URIs whose semantics is fully described in [www04d]. Among these, rdf:type, used as a property, assigns a type (or class) to a resource. This is a central mechanism in RDF that sets rdf:type statements (or *class assertions*) apart from other statements (*property assertions*). Other notable URIs include rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain and rdfs:range.

The semantics of RDF is defined through sets of entailments rules to *derive* implicit data from an RDF graph. In this work, we focus of RDFS-entailments, whose rules are listed in Table 2.1. These rules enforce hierarchical constraints among classes and properties as well as typing constraints on the domain or range of a property.

As an example, the triple represented in gray in Figure 2.2 (a) (corresponding to dashed edges in Figure 2.2 (b)) are not explicitly stored as part of the data. The fact that ":Obama is a president" is *inferred* from the fact ":Obama is :presidentOf of something", and that the "*domain* of the property :presidentOf is :President". Moreover, since ":President is a sub-class of :Person", it can further be inferred that ":Obama is a

Semantic relationship	RDF notation	FOL notation
Class inclusion	$(c_1, \text{rdfs:subClassOf}, c_2)$	$\forall X(c_1(X) \Rightarrow c_2(X))$
Property inclusion	$(p_1, \text{rdfs:subPropertyOf}, p_2)$	$\forall X \forall Y(p_1(X,Y) \Rightarrow p_2(X,Y))$
Domain typing of a property	$(p, \text{rdfs:domain}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(X))$
Range typing of a property	$(p, \text{rdfs:range}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(Y))$

Table 2.1: Semantic relationships expressible in an RDF Schema

:Person".

Given two RDF graph G and H, and a graph H' obtained by adding to H all triples that can be derived through RDFS-entailments, the problem of determining whether a graph G *RDFS-entails* H, i.e., G is a sub-graph of H', is NP-complete in the general case, and polynomial in the absence blank nodes [tH05]. The problem remains however tractable if blank nodes respect simple structures [PPWW08].

2.2.2 Querying & storing RDF

2.2.2.1 SPARQL

Queries on RDF data are usually expressed in SPARQL [www08b], a graph matching language with strong ties to relational query languages. In fact, it has been shown that a large fragment of SPARQL can be translated to SQL [CLF09] and datalog [Pol07].

In this thesis, we only consider the conjunctive fragment of SPARQL, a.k.a. *Basic Graph Pattern* (BGP) queries. A BGP is a set of *triple patterns*, i.e., triples in which variables can appear in subject, property or object positions. Such queries can be expressed as conjunctive queries over a single relation $t(s,\ p,\ o)$. For instance, the following query finds all resources typed as :Person, which have some relationship with a resource whose :name is "United States":

$$q(X) \text{ :- } t(X,\ \text{rdf:type},\ \text{:Person}), t(X,\ Y,\ Z), t(Z,\ \text{:name},\ \text{“}United\ States''})$$

The most recent SPARQL 1.1 recommendation [www13a] introduces the notion of entailment regimes [www13b], allowing one to choose among different semantics under which a query should be evaluated. In this context, the RDFS entailment regime corresponds to the semantics earlier described. Under the RDFS entailment regime, this query must return :Obama when evaluated over the data set of Figure 2.2.

2.2.2.2 Triple stores

Because RDF data can either be viewed as a graph or as a set of triples, various classes of storage systems have been proposed to store RDF data.

Relational approaches. Historically, relational approaches have been the first ones introduced, as they allow to inherit a host of features available in legacy RDBM systems, such as optimization and transaction management.

Among these, Jena [WSK+03] introduced so-called *property tables*, where each record corresponds to a distinct subject while each attribute of a table corresponds to a distinct property. Depending on the data distribution, such tables can be sparse. This can be mitigated with proper clustering and partitioning of the tables [LM09a]. Vertically partitioned tables, a.k.a. binary tables, are a particular case of property tables, where each table is made of two columns (subject, object) and stores all the triples of a given property [AMMH07, SGK+08].

Triple tables are yet another relational approach to store RDF. Sesame [BKVH02], Hexastore [WKB08] and RDF-3x [NW08] are proponent members of this class. As evaluating (even small) SPARQL queries on such tables leads to a large number of self-join on a potential large relation, the latter two systems resort to multi-way indexing of the three columns to maximize opportunities for merge joins. Index compression, cost-based join-ordering, sideways information passing [NW09] and fine-grained cardinality estimation [NM11] are among the ingredients used to minimize space consumption and maximize query evaluation performances.

As data sets grow in size, data compression has become in an important challenge when storing RDF. A simple yet effective compression technique introduces with RDF-3x is called *dictionary encoded*, whereby all distinct URIs and literals of a data instance are stored in a dictionary table and assigned a unique key. The triple table in turn only stores keys to the dictionary. This approach has been pushed further in the HDT system [FMPG+13] where, in addition to dictionary encoding, authors propose various compression schemes for the triple table itself.

Other approaches. As a graph data model, RDF can naturally be managed within graph data management systems. These systems are generally based on Key-Value stores, where keys are nodes and values are adjacency lists. This is the solution adopted in gStore [ZMC+11]. Neo4J [wwwg], a Java-based graph database, has built-in support for RDF. Another notable class of storage systems are known as RDF cubes, introduced by [MPK07] and later explored by Atre et al. [ACZH10] and De Virgilio [Vir12]. Conceptually, in an RDF cube, dictionary keys of each triple $(s,\ p,\ o)$ are the coordinates of a single point in a 3-dimensional discreet space. Each point in that space is a bit set to one if the triple exists, zero otherwise. An RDF instance can thus been seen as a sparse, and therefore highly compressible, bit tensor. At query evaluation time, bitwise operations allow to quickly prune intermediate results to improve memory usage.

The growing size of available RDF data sets has pushed many to study distributed storage schemes. Among them, Kaoudi et al. [KMK08] provides procedures to store RDF and perform RDFS query answering over DHTs. Urbani et al. [UKOvH09, UvHSB11] explored RDFS reasoning techniques over a Map-Reduce architecture. Huang et al. [HAR11]

use graph partitioning to distribute subparts of a large RDF dataset on nodes of a Map-Reduce cluster, so as to maximize the chances of joining data within individual nodes, and thus minimize data transfer.

Commercial systems are primarily based on relational back ends, such as Oracle [CDES05], IBM DB2 [wwwb] and Virtuoso [wwwk], with AllegroGraph [www04a], a graph database-backed system, as a notable exception.

2.2.2.3 Reasoning support in RDF data management systems

There are a variety of techniques to answer SPARQL queries under RDFS entailments, among which *forward chaining* and *backward chaining* are the most common ones. Briefly, the forward chaining approach consists in applying RDFS entailment rules onto the data to derive new facts, until it reaches a fixpoint. The statements thus derived are materialized along with the data. Query answering then amounts to query evaluation. With backward chaining, the data remains unchanged, however queries are rewritten using the entailment rules *backward*. This results in a union of conjunctive queries whose evaluation returns a complete answer w.r.t. the rules. These techniques are further detailed in Section 6.1.

In practice, rewritings obtained through backward chaining grow exponentially in the size of the input query, and are therefore generally hard to optimize. This is why most research and commercial systems resort to forward chaining despite its obvious space and maintenance overheads. Other techniques are in fact variations or optimizations of forward and backward chaining. In chapter 6, we present a novel approach to mitigate the pitfalls encountered with forward and backward chaining.

2.3 At the junction of RDF and XML

One 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] (OAC) show 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 these data in their 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 (respectively XML).

There are three major contexts in which research have sought to bring XML and RDF together. The first involves XML data integration using ontologies, the second covers

2. `http://openannotation.org`

attempts to achieve interoperability between the XML and RDF data models, while the third considers using RDF for annotating structured data.

2.3.1 XML data integration using ontologies

Although the idea of using ontologies to abstract the schema of distinct data sources is not new, one of the first attempts to use ontologies for XML integration is OntoBroker [DEFS99, ES01]. The system uses mapping rules from ontological concepts to DTD-style snippets, to automatically extract facts about XML documents from their structures. This allows later querying those documents on the concept derived from their structure, rather than just the structure itself. In [AFS+01, ABFS02a, ABFS02b], authors focus on a Local-As-View approach, where a global schema is expressed using domain ontology and views are rules from concepts or relationships to XPath expressions. These rules are used to rewrite queries on the global ontology to queries on the local XML documents, taking into account the presence of relations between rules and the distribution of sources.

Although, for historical reasons, these works rely on a general ontological formalism, RDF could trivially be used in the mapping definitions. They provide a means to convey semantics to XML data, but ultimately ontologies are only use at a *terminological* level, thus data instances are XML. In this works, we aim at bringing together *RDF and XML data instances*. Moreover, as we introduce the concept of XRQ views (Section 3.3), we will see that these can also be used in data integration scenarios, with the advantage that mappings can be directly expressed using our query language.

2.3.2 Interoperability between XML and RDF

RDF has several serializations, the most popular of which has long been 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 multiple 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 [AM08], an observation confirmed also in [KZ10]. Thus, one can consider the XML serialization of RDF as helpful for data sharing but not for human consumption, nor for query processing.

Some works consider employing the language of one model to query the other. These approaches imply two levels of translations: (i) converting the queries themselves, (ii) converting the data ahead of query evaluation and/or the data from the query results. These conversions bring either models to the level of the more complex one, which provides sufficient generality, but loses the performance benefits attained by current XQuery and SPARQL processors on many types of queries. This has been attempted to query RDF using XQuery or XSLT [RGN+01, PSS02]. The most recent papers in this line of work

are, to our knowledge [BGTC09] and [GGL+08], where performance evaluations show that in addition the data conversion costs, queries run two to four times slower (with Saxon and QuizX respectively) than the native SPARQL query engine Jena. The transformation of XML into RDF, so that it can be queried with SPARQL, is studied in [DFG+07].

The RDF model provides a means for typing literals in object position of a triple. Among those, the XMLLiteral type is a valid one. In [CKKC+09], authors have implemented XPath/XPointer function into SPARQL, to access sub-part of such literals. In this context, the XML content is belongs to the RDF instance, and therefore disconnected from any XML document. Moreover, the RDF model, forbidding literals in subject position, would prevent to connect XML nodes with a single triple. In [DFG+09], authors show how to translate such XPath expression into SPARQL sub-queries. Their experimental evaluation shows that the translation is detrimental to query evaluation.

Other works aimed at, as described in W3C's GRDDL recommendation [www08a], transforming XML data to RDF and *vice versa*. In the literature, these are known as *lifting* and *lowering*, respectively. XSPARQL [AKKP08], a hybrid query language based on the W3C recommendations for SPARQL and XQuery/XPath, is a proponent member of this family. Its syntax is essentially an extension of XQuery allowing interleaved SPARQL calls. Among other extensions, the language introduces `for` loops with multiple variables, e.g., to iterate over the results of a SPARQL query. The semantics of the language is defined in terms of rewritings from XSPARQL queries to XQuery and SPARQL. Any information passing between the two models is done through custom XQuery functions. Queries take XML and RDF files as input and, in turn, output XML *or* RDF. The authors detail an execution engine in-depth in [BDK+11]. Incoming queries are rewritten as XQuery, executed on off-the-shelve systems (Saxon and Qexo in this case). User-defined SPARQL functions access a Jena database. Because of its strong ties with XQuery, XSPARQL queries are by nature, made of nested loops. Authors presents some optimizations, also defined in terms of query rewritings, which attempt at unnesting queries as much as possible. Experimental results are presented, showing query evaluation times on RDF-translated of the XMark data. The largest dataset used in these experiments has 100M of data. Their results indicate that unnesting reduces the evaluation time by one to three orders of magnitude. Queries are however optimized primarily at a syntactic level. In this work, we will show it is also possible to optimize queries at the level of the physical plan. We found that unnesting is not *always* the best solution in practice, and that a closer look must be taken at the data and query structures, to make the best optimization choice.

Finally, a rule-based query language, Xcerpt [FBB05] allows to the management of XML or RDF through a common syntax.

2.3.3 Document annotations

Since the emergence of RDF, a set of tools was proposed to exploit the model and enable users to attach semantic annotations to Web pages. The representation of annotations

on XML documents has inspired projects focusing on a data model perspective such as Annotea [KK01], which served as a basis for the OAC initiative [HSSVdS11]. The latter model, currently a W3C community draft, is solely based on RDF. An annotation is a set of triples with a *body*, representing the content of the annotation along with metadata, and one or more *targets* that can be virtually of any type. In particular, targets can be defined by *selectors*, a set of triples that define the way to access the object (or one of its part) being annotated. Such selectors can be used to point to XML content, in which case, it is expressed using XPath or XPointer.

From an end-user perspective, tools have been proposed to annotate Web pages manually [HS02, Yee02] or in a semiautomatic fashion [DEG+03, VVMD+02]. A comprehensive overview of annotation systems can be found in [RH05]. However, these works focus solely on the problem of representing, storing or querying RDF annotations, and they do not consider the possibility to query simultaneously 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 documents created by employees on projects that may be similar to each other. In the French R&D project WebContent [AAC+08], authors have worked on building tools for warehousing semantically annotated pages gathered from the Web. 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" or "EU-manufactured aircraft".

A more recent addition to the family of frameworks for representing and querying documents with annotations is the QUASAR system [CBK12]. The data model relies on documents structured around three levels of granularity, paragraphs, sequence of sentences and tokens. RDF annotations can be scoped over *snippets*, i.e., contiguous sequences of blocks of any types. The query language is inspired from SQL, with additional syntactic features to allow for simple RDFS type of reasoning.

The problem of publishing RDF annotations *within* XML documents has been tackled by recent technology standards applied in the XHTML context: microformat [wwwf], eRDF [www06a] and W3C's RDFa [www04c] 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, only those having the right to modify the page, which is restrictive, can use such models. Moreover, the models do not lend themselves to the situation when users wish to keep their annotations private (or share only specific annotations with specific users).

2.3.4 Summary

XML and RDF are distinct data models, yet it was clear from RDF's inception that both models should complement one another. Data is now widely available in both models and converting all data from one to the other is not a pragmatic option. To leverage the benefits of XML and RDF, we need a model that works on existing data, while allowing to represent and query interconnected XML and RDF data. Such a model should also preserve all reasoning capabilities that RDF provides. It should be amenable to query composition, such that the result of a query be seen a new data instance. Ideally, the model should allow taking advantage of all existing query optimization techniques that have been proposed for XML and RDF.

In the next chapter, we present XR and its query language XRQ, that bring all these requirements together. After defining the model formally, we present the syntax and semantics of its query language and provide an algorithm to compose an XRQ query over a view.

Chapter 3

The XR data model and query language for semantics-rich documents

In this chapter, we present XR and XRQ, a data model and query language for Web data management. XR combines the advantage of the XML and RDF frameworks, without compromising any of the features defined in their respective recommendations. The contributions gathered in this chapter are the following:

Data model (Section 3.1). 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 may refer to an XML node).

Query language (Section 3.2). 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.

Composition (Section 3.3). The query language is amenable to query/view composition, by which a query is evaluated over the result of a view, rather than over an instance. We present an algorithm to compose an XRQ query over a view and prove its soundness.

3.1 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 [www01], and a subset $\mathcal{I} \subseteq \mathcal{U}$ of *document URIs* acting as document identifiers. We denote by $\mathcal{L}$ the set of literals (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 ϵ. 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 3.1.1 (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* $\tau(n) \in \{document, attribute, element, text\}$.

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 3.1.2 (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 4.2, we present URI assignment functions 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 3.1.3 (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* $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$.

Furthermore, RDF does not only model explicit triples, but also implicit (a.k.a. *entailed*) triples. These can be derived from the former based on a set of *entailment rules* as described in Section 2.2. For the purposes of our discussion though, we define the following notion: Given an RDF instance I_R, its semantics is the RDF instance I_R^∞, 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 3.2.2), since we need to take into account the implicit answers in order to guarantee the completeness of query answers.

We can now define an XR instance as follows:

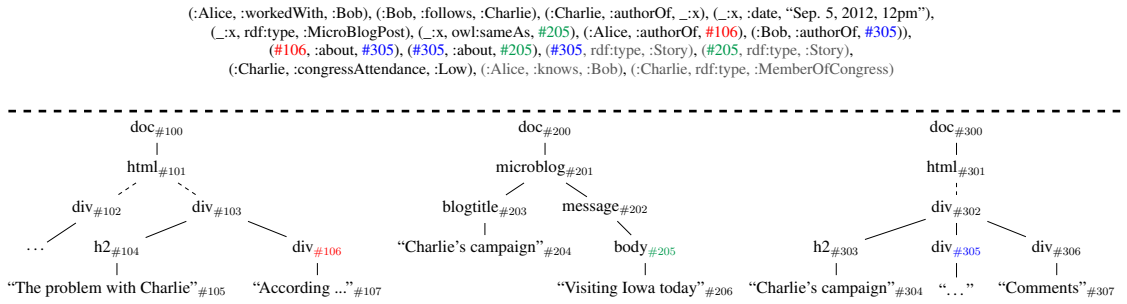


Figure 3.1: XR instance representing annotated documents

Definition 3.1.4 (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 3.1 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.

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 [wwwj]). 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 low.

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

3.2 The XRQ query language

XRQ allows querying an XR instance w.r.t. its structure (described in the XML sub-instance) and its semantic annotations (modeled in the RDF sub-instance). We first introduce the syntax (Section 3.2.1) and semantics (Section 3.2.2) of a core language, returning tuples of bindings. Next, we present an extended version of the language, which allows the creation of XR data. Both the syntax (Section 3.2.3) and semantics (Section 3.2.3) are described. The two languages essentially differ in the way results are returned and, as such, the semantics of the extended language includes that of the core language.

The interest of separating the languages into two level is two-fold:
- when introducing the composition problem (Section 3.3), looking at the core semantics of extended queries helps get better insight at the composition problem and algorithm,
- the cost of constructing results in the extended language plays a minor role in the overall query evaluation problem. Thus, for simplicity, we focus on the core language in our query evaluation study (Chapter 4.1),

3.2.1 Core XRQ syntax

XRQ allows querying an XR instance based on commonly used primitives: XML tree pattern queries, introduced, e.g., in [AYCLS01], and the BGP queries for RDF [www08b]. Tree patterns express structural constraints on the expected trees in the XML sub-instance, while BGPs allow constraining the expected triples of the RDF sub-instance.

Definition 3.2.1 (Tree Pattern)**.** *A **tree pattern** is a finite, unordered, unranked, $\mathcal{N}$-labeled 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. Variable labels are of the form $t : v$, where t is the variable's type, and v its name. We may also attach to a node one equality predicate of the form $t : v = c$, where $c \in \mathcal{L}$, denoting a selection on the variable v. The variable name may be omitted if it is not used anywhere else in the query, leading to a label of the form $t = c$.*

A 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 [BÖB+04, ABM05]

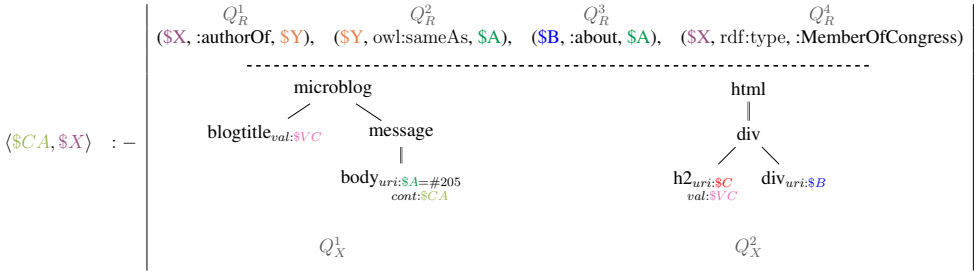


Figure 3.2: Sample XRQ query

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 3.2 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. Moreover, a selection on variable $A, which in this case must match the URI #205.

Definition 3.2.2 (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.*

A BGP is a conjunction of triple patterns.

Example. The top part of Figure 3.2 depicts four triple patterns. For instance, the leftmost 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 3.2.3 (Core 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 a tuple of variables appearing also in the body or constants. 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 allows correlating structural and semantic constraints within queries. The following example illustrates the expressivity of XRQ.

Example. Figure 3.2 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 3.2: 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 $\$B$).

3.2.2 Core 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 [AYCLS01]:

Definition 3.2.4 (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 ϕ from the nodes of Q to the nodes of I_X that preserves (i) node labels, i.e., for every node $n \in Q$, $\phi(n) \in I_X$ has the same label as n, and (ii) structural relationships, that is: if n_1 is a $/$-child of n_2 in Q, then $\phi(n_1)$ is a child of $\phi(n_2)$, while if n_1 is a $//$-child of n_2, then $\phi(n_1)$ must be a descendant of $\phi(n_2)$.*

Moreover, ϕ satisfies 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 3.2.5 (Variable binding of a tree pattern against an XML instance)**.** *Let ϕ be a match of a tree pattern Q against an XML instance I_X and V the set of variables in Q. Let $v \in V$ be a variable associated with a node n. Then the* variable binding *f of Q against I_X corresponding to ϕ is a function over V such that: (i) if v is a* uri *variable, then $f(v)$ is the URI of $\phi(n)$ in I_X, (ii) if v is a* val *variable, then $f(v)$ is the value of $\phi(n) \in I_X$, and (iii) if v is a* cont *variable, then $f(v)$ is the serialization of the subtree of I_X rooted at $\phi(n)$.*

As explained above, a variable binding f of a tree pattern Q against I_X is associated with a match ϕ 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 3.2.6 (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^∞ the saturation of I_R. A* match *of Q against I_R is a mapping from $\{s, p, o\}$ to the components of a single triple $t_\phi = (s_\phi, p_\phi, o_\phi) \in I_R^\infty$, such that $\phi(s) = s_\phi$, $\phi(p) = p_\phi$ and $\phi(o) = o_\phi$, and for any URI or literal ul appearing in s, p or o, we have $\phi(ul) = ul$ (ϕ maps any URI or literal only to itself).*

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^∞. As defined in Section 3.1, I_R^∞ 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 3.2.7 (Variable binding of a triple pattern against an RDF instance)**.** *Let ϕ be a match of a triple pattern Q against an RDF instance I_R. Then the* variable binding *of Q against I_R corresponding to ϕ is the function $\phi|_V$, where V is the set of variables in Q.*

We now provide the semantics of an XRQ query:

Definition 3.2.8 (Core 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 {⟨⟩} 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 3.2 to the XR instance of Figure 3.1 yields the result: ($CA=⟨body⟩Visiting Iowa today⟨/body⟩, $X= Charlie).

Figure 3.3 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 sub-query 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 chapter.

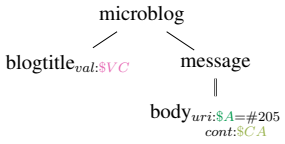
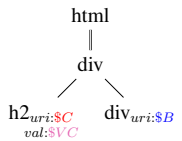
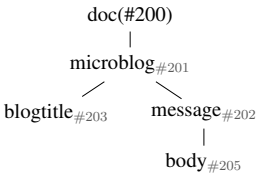
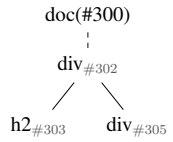
	Q_R	Q_X^1	Q_X^2
Patterns	($X, :authorOf, $Y), ($Y, owl:sameAs, $A), ($B, rdfs:seeAlso, $A), ($X, rdf:type, :MemberOfCongress)		
Matches	(:Charlie, :authorOf, _:x), (_:x, owl:sameAs, #205), (#305, rdfs:seeAlso, #205), (:Charlie, rdf:type, :MemberOfCongress)		
Variable bindings	{$A=#205, $CA=⟨body⟩Visiting Iowa today.⟨/body⟩, $B=#305, $C=#303, $VC="Charlie's campaign", $X=:Charlie, $Y=_:x}		

Figure 3.3: Pattern matches and variable bindings of the query of Figure 3.2 on the XR instance of Figure 3.1

3.2.3 Extended XRQ syntax

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

The head of an extended query not only allows the generation of trees and triples in the output but also allows triples that annotate fresh nodes. The syntax and the semantics of the extended language are presented below.

Definition 3.2.9 (Tree Constructor)**.** *A **tree constructor** is a finite, ordered, unranked, $\mathcal{N}$-labeled tree child edges only. We may attach to each node at most one* assignment label *consisting of a single variable and one* grouping label *consisting of a tuple of variables and constants. This tuple may not contain any variable already present in the grouping label of an ancestor. When omitted, the empty tuple is assumed. We may also attach to each leaf node a* value label *consisting of a single constant or variable.*

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

Example. The trees at the bottom left hand side of Figures 3.4 and 3.5 are tree constructors. The constructors are identical except in the way they are labeled. In Figure 3.4, the

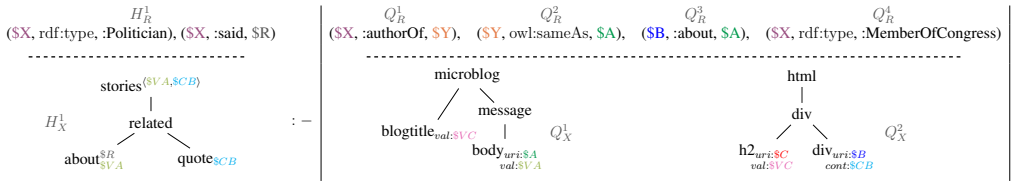


Figure 3.4: Sample XRQ$_{ext}$ query a grouping label on the top node

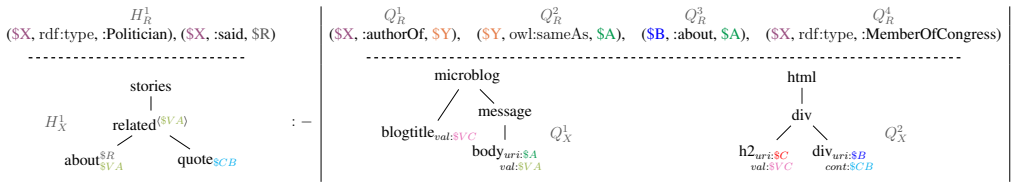


Figure 3.5: Sample XRQ$_{ext}$ query grouping labels on intermediate nodes

top *stories* node has a grouping label featuring variables $\$VA$ and $\$CB$, the *about* node at the bottom left is labeled with an assignment label $\$R$ and a value label $\$VA$, while the *quote* node at the bottom right has the value label $\$CB$. In contrast, the top *stories* node in Figure 3.5 does not have any label, but its child *related* has the grouping label featuring variable $\$VA$. The "about" and "quote" nodes are labeled similarly to Figure 3.4.

Definition 3.2.10 (XRQ$_{ext}$query head)**.** *An XRQ$_{ext}$ is a tuple (H_X, H_R, h), where H_X is a set of tree constructors, H_R is a set of triple patterns, and h is a tuple of variables or constants in $\{\mathcal{U} \cup \mathcal{L}\}$.*

For each tree constructor $t_X \in H_X$, and for each node $n_X \in t_X$:
– if n_X has a assignment label *v, then v is a variable of type* URI *s.t. $v \notin h$,*
– if n_X has a grouping label *of the form $(l_1, \ldots, l_k)$, then $l_i \in \{\mathcal{L} \cup \mathcal{U} \cup h\}$,*
– if n_X has a value label *l, then $l \in \{\mathcal{L} \cup \mathcal{U} \cup h\}$.*
Finally, each triple pattern $t_R \in H_R$ complies to Definition 3.2.2 with the restriction that variables may only belong to h or be assignment labels appearing in H_X.

Example. The left hand side of Figures 3.4 and 3.5 are both query heads made of two triple patterns and a single tree constructor, with $h = (\$X, \$VA, \$CB)$ and the assignment label $R. The head triple patterns feature $R and the variable $X appearing in h.

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

Definition 3.2.11 (XRQ$_{ext}$ Syntax)**.** *An XRQ$_{ext}$ query, denoted $Q = (H_X, H_R, h, Q_X, Q_R, \text{SK})$, consists of a head (H_X, H_R, h), a core query (h, Q_X, Q_R) and a bijective mapping* SK *: $H_X \to \mathcal{S}$, assigning a distinct Skolem function to each node of H_X.*

Example. Figures 3.4 and 3.5 depicted XRQ$_{ext}$ queries whose body is similar to that of the query in Figure 3.2. The two queries only differ in their grouping labels. Figure 3.4 has a grouping label $\langle \$VA, \$CB \rangle$ on the top node and none on the "related" node, while Figure 3.5 has the grouping label $\langle \$VA \rangle$ on the "related" node and none on the top node.

3.2.4 Extended XRQ semantics

We now formalize the semantics of the extended language. In the following definition, a variable binding f of Q against an XR instance I is defined as for XRQ queries in Definition 3.2.8.

Definition 3.2.12 (XRQ$_{ext}$ semantics)**.** *The result of an XRQ$_{ext}$ query $Q = (H_X, H_R, h, Q_X, Q_R, \text{SK})$ over $I = (I_X, I_R)$, is an XR instance (I'_X, I'_R).*

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

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

Example. Figures 3.6 and 3.7 show the results of the queries depicted in Figures 3.4 and 3.5 respectively, with the following bindings (we omitted variables that do not appear in the head):

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

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

In Figure 3.4, the root node has grouping label $\langle \$VA, \$CB \rangle$. Its result is obtained as follows. Let us consider the first binding f_1. After copying the root node, its URI is assigned through the function call $sk_1((f_1(\$VA), f_1(\$CB)))$, where sk_1 is the Skolem function assigned to the root node in H_X. This returns URI $\#sk_1$:1. The URI of the node label "related" will be obtained through the call $sk_2((f_1(\$VA), f_1(\$CB)))$, which returns $\#sk_2$:1. Since the node has no grouping label, the empty list is assumed. The input of

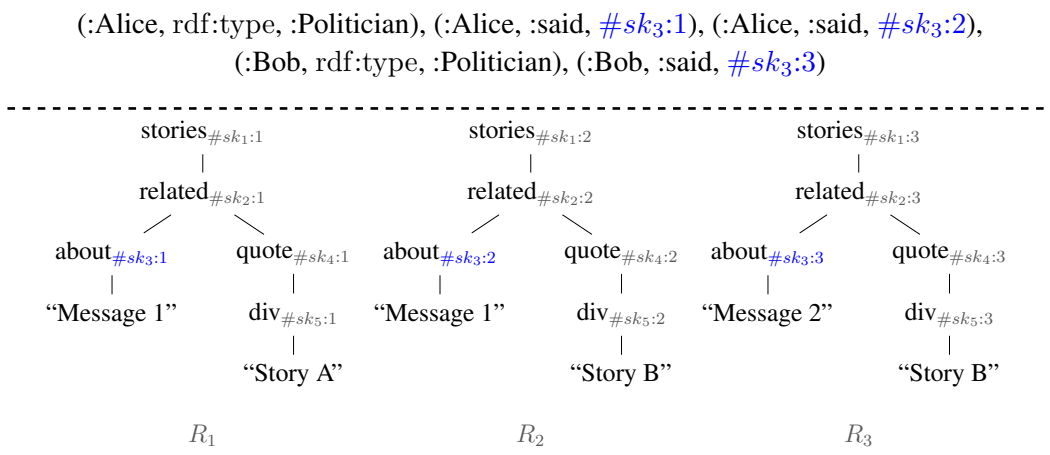


Figure 3.6: Result of an XRQ$_{ext}$ query without grouping nodes

sk_2 is a concatenation of the variable bindings declared in the grouping label of all the node's ancestors down to the current one. Since the root node's grouping label already contained all possible variables, the input tuple of sk_2 will be the same as the input tuple of sk_1. This applies to all nodes in H_X, to produced R_1, the left-most tree on Figure 3.6. The second and third trees R_2 and R_3 are built in a similar manner. Since, the variable bindings of f_1, f_2 and f_3 are all different, the sets of URIs for nodes of each tree are disjoint. However, the binding f_4 is identical to f_2, thus Skolem function calls for each node of the tree associated with f_4 return the same URIs as in R_2, eventually merging the two trees into a single one. The URIs of the "about" nodes, $\#sk_3$:1, $\#sk_3$:2, $\#sk_3$:3 and $\#sk_3$:2, are bound to \$R and added to bindings f_1, f_2, f_3 and f_4 respectively. These are necessary to produce the triples shown at the top of Figure 3.6.

Next, we explain Figure 3.7, which represents the result of the query depicted on Figure 3.5. Since no grouping label is defined on the root node, the empty list is assumed, then $sk_1(())$ is called to assign a URI to the first node as well as every subsequent copies of the root node. Consequently, all the top nodes in the results will be merged into a single one. Next, the "related" nodes are produced. Their corresponding grouping label includes \$VA, therefore the input of sk_2 is $(f_i(\$VA))$, for each finding f_i, with $1 \leq i \leq 4$. Calls to Skolem function for this node with bindings f_2 and f_4 will return the same URIs, since the same value is bound to \$VA in those bindings. However, for binding f_3 a fresh node will be generated. The input of the Skolem function for every child of the "related" nodes will be the same, as none of them specifies any additional variable. The node label "quote" however, features the variable \$CB as a value label, therefore the input of the Skolem function for this node will be $(f_i(\$VA), f_i(\$CB))$, i.e., the concatenation of the input of its ancestors and \$CB, resulting in the grouping shown in the Figure.

The construction of XML nodes affects the construction RDF triples. In Figure 3.6, the three distinct "related" nodes led to the creation of three corresponding triples. In Figure 3.7, only two "related" nodes were ultimately generated. The variable \$R was

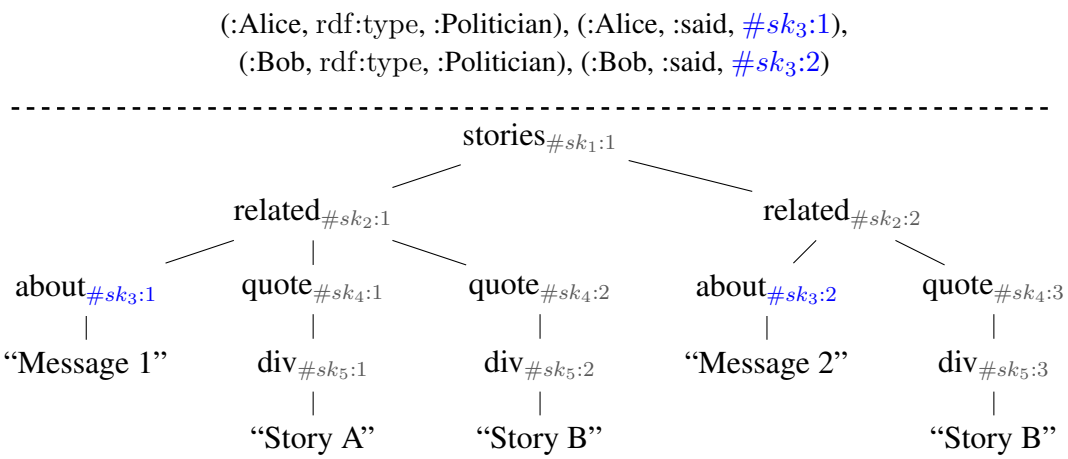


Figure 3.7: Result of an XRQ$_{ext}$ query with grouping nodes

bound to the same URI $sk_3{:}1$ in the three bindings f_1, f_2 and f_4, leading to three identical triples $(:\text{Alice}, :\text{said}, \#sk_3 : 1)$. Due to the set semantics of RDF, these coincide into a single one in the RDF sub-instance.

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

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

Definition 3.2.13 (Unions of XRQ$_{ext}$)**.** *An XRQ$^{\cup}_{ext}$ query is a set of XRQ$_{ext}$ queries. For any data instance $\mathcal{I}$, the result of an XRQ$^{\cup}_{ext}$ query is the union of the results of its sub-queries over $\mathcal{I}$.*

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

The semantics of XRQ$^{\cup}_{ext}$ follows that of XRQ$_{ext}$. Note that although the node-to-Skolem function mapping SK of Definition 3.2.12 is bijective, nodes from tree constructors of distinct sub-queries in a union may map to the same Skolem function. As a consequence, if data nodes yielded by distinct sub-queries have the same URI, they will coincide in the result of the union. The interest of unions of XR queries will become clear when we tackle the problem of composition in the next section.

3.3 XRQ view-query composition

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

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

3.3.1 Motivations

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

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

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

Data integration. There exists a large body of XML and RDF data on the Web that cover related topics or contain complementary information. For instance, every year, the OECD publishes very fine-grained economic and social data on a wide range of countries in XML. This data does not, however, incorporate the historical background of these countries or information about the current political situation. A RDF dataset such as DBpedia does, however, contain that type of information. But since these data sets are

stored in pure XML and RDF models, querying both of them concurrently requires insight about the data on either side. For example, to join the data on the country, one would need to know which OECD and DBpedia fields to use. An elegant way to simplify that type of queries would be to integrate the two datasets into an XR-ready form, where each OECD yearly record is annotated with country-specific data coming from DBpedia. We give a concrete example of that kind of data integration at the end of this section.

3.3.2 Problem statement

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

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

3.3.3 Preliminaries

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

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

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

Definition 3.3.1 (Body normalization)**.** *Let $Q = (H_X, H_R, Q_X, Q_R)$ be an XR query, where $n_1, \ldots, n_n$ are XML nodes appearing in Q_X and $(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)$ are triples patterns appearing in Q_R.*

$$\frac{n_i \in Q_X}{Node(x, t_i)}, \tag{3.1}$$

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

$$\frac{n_i \in Q_X, \textit{ s.t. } n_i \textit{ has a selection label of the form } uri = c}{x = c} \tag{3.2}$$

$$\frac{n_i \in Q_X, \textit{ s.t. } n_i \textit{ has a Val variable label of the form } val : v}{Val(x, v)} \tag{3.3}$$

$$\frac{n_i \in Q_X, \textit{ s.t. } n_i \textit{ has a selection label of the form } val = c}{Val(x, y), y = c} \tag{3.4}$$

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

$$\frac{n_i, n_j \in Q_X, \textit{ such that } n_i \textit{ is a single-edge parent of } n_j}{Edge(x, y)} \tag{3.5}$$

$$\frac{n_i, n_j \in Q_X, \textit{ such that } n_i \textit{ is a double-edge parent of } n_j}{Path(x, y)} \tag{3.6}$$

where x (resp. y) matches the first attribute of the $Node$ predicate produced by rule 3.1 for the node n_i (resp. n_j).

$$\frac{(s_i, p_i, o_i) \in Q_R}{Triple(s'_i, p_i, o'_i)} \tag{3.7}$$

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

The restrictions on the rules impose that rule 3.1 be applied exhaustively before any other rule.

Intuitively, normalizing the body of an XR query produces a $Triple$ atom for each triple pattern in the query body, a $Node$ atom for each XML node, an $Edge$ atom for each parent-child edge and a $Path$ for each ancestor-descendant edge. The presence of Val variables and selections on tree patterns gives rise to Val and $=$ predicates. The arguments of the $Edge$ and $Path$ predicates, as well as the first argument of $Node$ and Var admit a variable that represents a node's URI. Since, all XML nodes of an XR instance have distinct URIs, this allows our set of atoms to represent the relationships among the nodes of tree patterns. The second arguments of the $Node$ and Val act as selections on a node's tag and string value respectively.

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

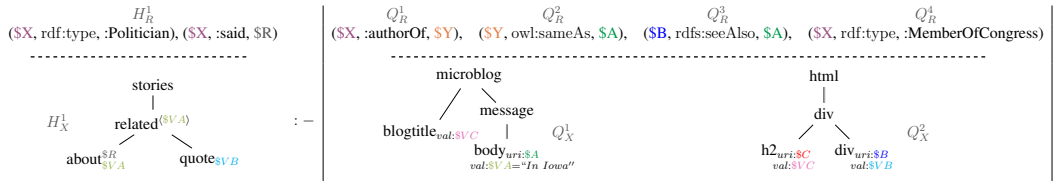


Figure 3.8: Variant of the query depicted on Figure 3.5

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

Table 3.1: Normalization for the body of the query described in Figure 3.8

Example. As an example, consider the query depicted on Figure 3.8, a variant of the one depicted on 3.5, where a selection has been added on the *body* node of the rightmost tree pattern. Table 3.1 lists the atoms obtained by normalizing its body. Fresh variables are all of the form $\$v_i$. Constants and variables that appeared in the input query are left intact and are all present in the normalization.

Definition 3.3.2 (Head normalization)**.** *Let $Q = (H_X, H_R, Q_X, Q_R)$ be an XR query, where $n_1, \ldots, n_n$ are XML nodes appearing in H_X and $(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)$ are triples patterns appearing in H_R.*

$$\frac{n_i \in H_X}{Node(x, t_i)}, \tag{3.8}$$

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

$$\frac{n_i \in H_X, \textit{ s.t. } n_i}{x = sk_i} \tag{3.9}$$

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

$$\frac{n_i \in H_X, \textit{ s.t. } n_i \textit{ has a value label } y}{Val(x, y)} \tag{3.10}$$

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

Table 3.2: Normalization for the head of the query depicted on Figure 3.8

$$\frac{n_i, n_j \in H_X, \textit{ such that } n_i \textit{ is a single-edge parent of } n_j}{Edge(x, y)} \tag{3.11}$$

$$\frac{n_i, n_j \in H_X, \textit{ such that } n_i \textit{ is an ancestor of } n_j}{Path(x, y)} \tag{3.12}$$

where x (resp. y) matches the first attribute of the $Node$ predicate produced by rule 3.8 for the node n_i (resp. n_j).

$$\frac{(s_i, p_i, o_i) \in H_R}{Triple(s_i, p_i, o_i)} \tag{3.13}$$

where s_i, p_i and o_i are variables or constants.

As for Procedure 3.3.1, rule 3.8 must be applied exhaustively before all the others.

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

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

Example. In Table 3.2, we present the normalization of the query depicted on Figure 3.8. Equality constraints have already been propagated. Notice how the selection applied in the body to variable $VA propagates to the body and head atoms. Likewise, although variable $\$R$ initially appears in the first stages of the normalization, it was ultimately replaced by the constant $f_{about}(\text{“}In\ Iowa'')$, due to the equality constraint $\$R = f_{about}(\text{“}In\ Iowa'')$ (generated by Rule 3.4).

We now extend the concept of normalization to XR instances.

Definition 3.3.3 (Instance normalization)**.** *We call $\overline{\mathcal{I}}$, the normalization of an instance $\mathcal{I}$, a set of atoms whose parameters are constants or blank nodes, obtained by applying the following rules onto $\mathcal{I}$:*

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

Table 3.3: Number of atoms produced by normalizing of an XR query q into a FOL formula $\overline{q}$

$$\frac{(s, p, o) \in \mathcal{I}}{Triple(s', p, o')}, \tag{3.14}$$

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

$$\frac{n \in \mathcal{I}}{Node(u_n, t_n)}, \tag{3.15}$$

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

$$\frac{n, m \in \mathcal{I}\text{, s.t. } n \text{ and } m \text{ are parent and child nodes of an edge}}{Edge(u_n, u_m)}, \tag{3.16}$$

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

$$\frac{n, m \in \mathcal{I}\text{, s.t. } n \text{ and } m \text{ are ancestor / descendant node pair of any path}}{Path(u_n, u_m)}, \tag{3.17}$$

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

$$\frac{n \in \mathcal{I}\text{, s.t. } n_i \text{ has a text child}}{Val(u_n, c)}, \tag{3.18}$$

where c is the value of the text node.

Complexity of the normalization. The size of a normalized query is dominated by the number of *Path* predicates generated by rule 3.12 making the normalization result complexity quadratic in the size of the original query. Let $q = (H_X, H_R, Q_X, Q_R)$ be an XRQ$_{ext}$ query and $\overline{q}$ its normalization. We note $|Q_R|$ and $|H_R|$ the number of triple patterns in the head and body of q respectively, $|Q_X|$ and $|H_X|$ the number of XML nodes in q. The number of atoms produced by the normalization is given in Table 3.3.

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

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

3.3.4 Composition algorithm

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

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

Algorithm 1 has two phases:

1. During the matching phase (lines 4-10), we attempt to unify each atom p_i in the body of the normalized query $\overline{q}$, with each head atom from the normalized view $\overline{v}$. For each atom p_i, we also keep a copy of $\overline{v}$, whose variables have been replaced with fresh ones. Whenever a valid substitution is found, it is added to a set of valid substitutions Σ_i associated with p_i. At the end of the matching phase, n sets of substitutions have been created, one for each atom from the normalized query body.

2. During the building phase (lines 12-18), we iterate over the Cartesian product of these sets, and we form the union of the substitutions in each entry. We call this union a *compound substitution*, noted $\sigma_\cup$. Each compound substitution covers all the atoms in the body of q. A compound substitution may be inconsistent, in which case it is discarded. Consistency can be checked applying simple rules. For instance, if a single variable maps to two distinct constants, the substitution is inconsistent.

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

Algorithm 1: XRCOMP

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

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

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

3.3.4.1 Properties

We now discuss the properties of Algorithm 1.

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

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

Given a query q, an instance $\mathcal{I}$, a match μ of q over $\mathcal{I}$ and its associated binding β of q variables into $\mathcal{I}$, there exists a corresponding homomorphism $\overline{\mu} : body(\overline{q}) \rightarrow \overline{\mathcal{I}}$, and a

substitution $\overline{\beta}$ mappings variables of $\overline{q}$ to constants of $\overline{\mathcal{I}}$, such that $\forall a \in body(q), a^{\overline{\beta}} = \overline{\mu}(a)$.

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

Let q be a query, v a view without fresh blank nodes in the head and q' the union of XR queries obtained by Algorithm 1 for q and v.

Consider the set of queries $q^1_{core}, \ldots, q^n_{core}$, s.t. the body of each q^i_{core} is a single atom from the body of q_{core}, while the head of q^i_{core} features all variables of $body(q^i_{core})$. For any instance $\mathcal{I}$, we have:

$$q_{core}(\mathcal{I}) = \pi_{head(q_{core})}(q^1_{core}(\mathcal{I}) \bowtie \ldots \bowtie q^n_{core}(\mathcal{I})) \tag{3.19}$$

In particular, (3.19) holds for any instance $\mathcal{I}' = v(\mathcal{I})$, that is, for any image of an XR instance $\mathcal{I}$ through v. Replacing $\mathcal{I}$ with $v(\mathcal{I})$, we obtain:

$$q_{core}(v(\mathcal{I})) = \pi_{head(q_{core})}(q^1_{core}(v(\mathcal{I})) \bowtie \ldots \bowtie q^n_{core}(v(\mathcal{I}))) \tag{3.20}$$

which, by the definition of the view-query composition $\circ$, is equivalent to:

$$(q_{core} \circ v)(\mathcal{I}) = \pi_{head(q_{core})}((q^1_{core} \circ v)(\mathcal{I}) \bowtie \ldots \bowtie (q^n_{core} \circ v)(\mathcal{I})) \tag{3.21}$$

Let us now characterize the result of each sub-query $(q^i_{core} \circ v)(\mathcal{I})$. Let p_i be the atom in $body(q^i_{core})$; p_i may be unifiable with multiple atoms in $head(v)$; let P_i be the set of those $head(v)$ atoms, and for each $p_j \in P_i$, θ^i_j is the MGU of p_i and p_j.

Then, it follows that:

$$(q^i_{core} \circ v)(\mathcal{I}) = \bigcup_{p_j \in P_i} (p_j \text{ :- } body(v))^{\theta^i_j}(\mathcal{I}) \tag{3.22}$$

In the above, the equality follows from the fact that q^i_{core} may only produce results from those atoms in the head of v into which the single atom in $body(q^i_{core})$ embeds.

Consider a single conjunctive XR query q'_σ from the union q', produced in the build phase of Algorithm 1 (lines 12-18) from an entry $\sigma \in \{\Sigma_1, \ldots, \Sigma_n\}$. As specified in the Algorithm, q'_σ is of the form:

$$head(q)^{\sigma_\cup} \text{ :- } (body(v_1))^{\sigma_\cup}, \ldots, (body(v_n))^{\sigma_\cup} \tag{3.23}$$

For every $1 \leq i \leq n$, the sub-query $body(v_i)^{\sigma_\cup}$ in q'_σ corresponds to a match identified by Algorithm 1 during its matching phase (lines 4-10) from an atom $p_i \in body(q)$, into an atom from $head(v)$. Observe that p_i is precisely the atom in the body q^i_{core}.

In the context of q'_σ, p_i matches a single atom p_j of $head(v)$, and σ^i_j is the MGU of p_i and p_j. Note that θ^i_j from (3.22) is also defined as the MGU of p_i and p_j, thus it is equivalent to σ^i_j. Therefore,

$$(p_i \text{ :- } body(v_i))^{\sigma^i_j}(\mathcal{I}) \subseteq (q^i_{core} \circ v)(\mathcal{I}) \tag{3.24}$$

And since σ^i_j is included in $\sigma_\cup$:

$$(p_i \text{ :- } body(v_i))^{\sigma_\cup}(\mathcal{I}) \subseteq (q^i_{core} \circ v)(\mathcal{I}) \tag{3.25}$$

Observe that although variables in $v_1, \ldots, v_n$ are disjoint, during the matching phase, variables of each predicate in $body(q)$ are matched against variables of a distinct view among $v_1, \ldots, v_n$. Therefore, joins that exist among any predicates pair p_i, p_j of $body(q)$ will be preserved in $\sigma_\cup$, and v_i will necessarily join with v_j. Therefore,

$$\begin{aligned}&(p_1 \text{ :- } body(v_1))^{\sigma_\cup}(\mathcal{I}) \bowtie \ldots \bowtie (p_n \text{ :- } body(v_n))^{\sigma_\cup}(\mathcal{I}) \\ &\quad = (p_1, \ldots, p_n \text{ :- } body(v_1), \ldots, body(v_n))^{\sigma_\cup}(\mathcal{I})\end{aligned} \tag{3.26}$$

If we extend claim (3.25) to all the sub-queries of q'_σ, we have:

$$\begin{aligned}&(p_1 \text{ :- } body(v_1))^{\sigma_\cup}(\mathcal{I}) \bowtie \ldots \bowtie (p_n \text{ :- } body(v_n))^{\sigma_\cup}(\mathcal{I}) \\ &\quad \subseteq (q^1_{core} \circ v^1)(\mathcal{I}) \bowtie \ldots \bowtie (q^n_{core} \circ v^n)(\mathcal{I})\end{aligned} \tag{3.27}$$

From (3.21) and (3.26), this reduces to:

$$\pi_{head(q_{core})}(p_1, \ldots, p_n \text{ :- } body(v_1), \ldots, body(v_n))^{\sigma_\cup})(\mathcal{I}) \subseteq (q \circ v)(\mathcal{I}) \tag{3.28}$$

Hence, the set of bindings upon which q'_σ constructs a result is included in the set of bindings given by $q_{core} \circ v$.

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

$$q'_\sigma(\mathcal{I}) \subseteq (q \circ v)(\mathcal{I}) \tag{3.29}$$

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

$$q'(\mathcal{I}) \subseteq (q \circ v)(\mathcal{I}) \tag{3.30}$$

□

Blank nodes. There are two reasons why blank nodes are not allowed in the head of views in the composition algorithm. Firstly, blank nodes (which in our semantics are defined as existential variables) may appear among the parameters of the Skolem functions of XML node constructors. Secondly, by definition, fresh blank nodes in the head of a view do not appear in its body. Fresh head blank nodes may take part in the matching process and yield some substitutions, yet these substitutions will have no effect on the final composed query, because it is formed by combining copies on the view's body, resulting in incorrect compositions.

Complexity of Algorithm 1. As we saw in Section 3.3.3, the complexity on the normalization steps at lines 1 and 2 are respectively quadratic and linear in the size of q. It directly follows that the nested loops of the matching phase (lines 4-10) run it cubic time. The **mgu** function (line 6) operates in constant time since we only attempt to unify pairs of atoms of arity 2 or 3. In the worst case, the size of the Cartesian product explored in the building phase (lines 12-18) is exponential in the size of the q, i.e., $O(|\overline{v}|^{|\overline{q}|})$. The consistency check performed at line 14 can be done in linear time in the size of q. Similarly, the substitutions involved in the creation of the UCQ (lines 15-17) can be performed in linear time. It follows that the time complexity of the building phase is overall exponential q and v. Every entry in the Cartesian product of substitutions may yield a sub-query in the union q', each of which features a body of $|\overline{q}| \times body(v)$, therefore the size of q' is bounded by $O(|\overline{v}|^{|\overline{q}|} \times (size(head(q)) + |\overline{q}| \times size(body(v))))$.

3.3.5 Composition examples

Lifting and lowering examples. We now provide some examples of XRQ view, queries, along with their compositions, in the context of social networks. Query Q_1, depicted on Figure 3.9, on the one hand, evaluates over an XML document featuring message feeds between users. It retrieves the URIs and names of users who have sent messages to their friends, along with the URIs of these messages, of their recipients and the content value of the messages sent. It outputs roughly the same information in RDF, with additional triples stating that the sender knows the recipients and vice versa.

($\$user$, :name, $\$name$)
($\$user$, :knows, $\$friend$)
($\$friend$, :knows, $\$user$)
($\$user$, :said, $\$msg$)
($\$msg$, rdf:type, :Message) :-
($\$msg$, :content, $\$body$)
($\$msg$, :recipient, $\$friend$)

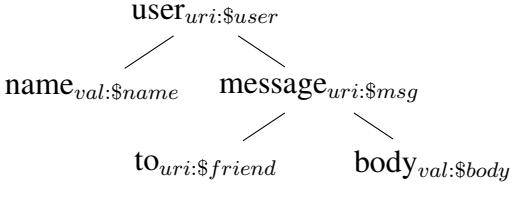
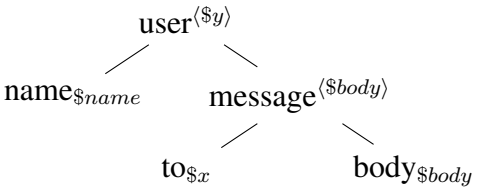


Figure 3.9: Query Q_1, *lifting* social network information from XML to RDF

Query Q_2, depicted on Figure 3.10, on the other hand, evaluates over an RDF sub-instance. It retrieves users who have sent an identical message to pairs of people that know each other. In turn, it builds an XML tree in which the top node groups each distinct sender, and append their name as a child node. Then, under each sender, it groups messages sent to multiple people, appends each recipient under a distinct child labeled "to" and finally appends the message itself (once).



$$
\text{:-} \;\Big|\; \begin{array}{l} (\$x,\ \text{:knows},\ \$z) \\ (\$y,\ \text{:name},\ \$name) \\ (\$y,\ \text{:said},\ \$msg1) \\ (\$msg1,\ \text{:content},\ \$body) \\ (\$msg1,\ \text{:recipient},\ \$x) \\ (\$y,\ \text{:said},\ \$msg2) \\ (\$msg2,\ \text{:content},\ \$body) \\ (\$msg2,\ \text{:recipient},\ \$z) \end{array} \;\Big|
$$

Figure 3.10: Query Q_2, *lowering* social network information from RDF to XML

These types of queries are sometimes qualified as lowering and lifting queries, as they respectively lower data from RDF to XML or lift it from XML to RDF. These two queries happen to be composable on one another. Composing Q_2 over Q_1 produces a union of two queries, whose respective bodies feature eight self-joins over copies of Q_1 (as there are eight atoms in the normalized body of Q_2, thus eight variable-renamed copies of Q_1 are used in the matching phase). These query bodies can be minimized. For readability, Figure 3.11 shows the composition of Q_2 over Q_1, where each sub-query of the union has been minimized and features only to three self-joins.

Figure 3.12 depicts the composition of Q_1 over Q_2. In this case, at the end of the matching phase, only one entry of the Cartesian products of substitutions yields a consistent substitutions after been unioned. This explains why the composed query is not a union. The head of the composed query features multiple calls to Skolem functions. These are the calls that would be used to assign URIs to XML node in the result of Q_1, e.g., if the view had to be materialized.

XR integration. The next scenario shows how one can query interconnected XR, in the absence of such data, for example, on legacy XML and RDF data. Figure 3.13 shows a view that combines data from an RDF data instance such as DBpedia and XML data instance such as some OECD report, gathering data about countries productivity and GDP per year. The query uses the country code to join data from both sub-instances, and constructs an XML tree, that is similar to the one filtered, up to XML node URIs. However, the RDF sub-instance produced by the query features new RDF triples that directly link countries to the XML nodes containing informations collected by the OECD.

Query Q_4 on Figure 3.14 attempts to find the productivity and GDP of France in 2010,

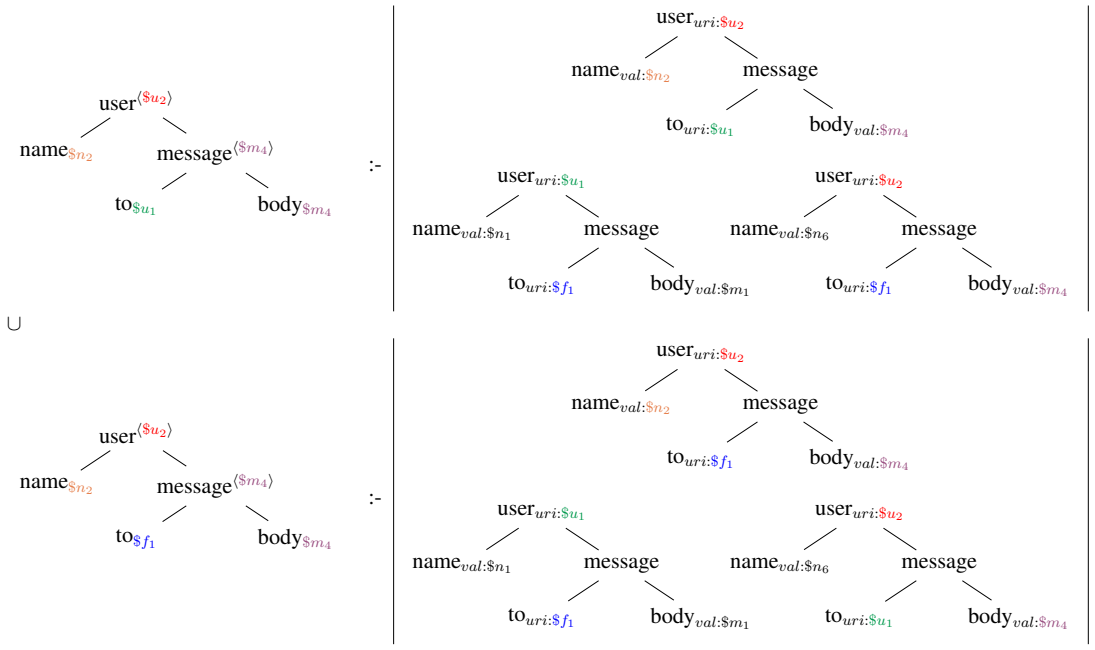


Figure 3.11: Composition of Q_2 over Q_1

$$
\begin{array}{l|l}
(f_{user}(\$y),\ \text{:name},\ \$name) & (\$x,\ \text{:knows},\ \$z) \\
(f_{user}(\$y),\ \text{:knows},\ f_{to}(\$y, \$body, \$x)) & (\$y,\ \text{:name},\ \$name) \\
(f_{to}(\$y, \$body, \$x),\ \text{:knows},\ f_{user}(\$y)) & (\$y,\ \text{:said},\ \$msg1) \\
(f_{user}(\$y),\ \text{:said},\ f_{msg}(\$y, \$body)) & (\$msg1,\ \text{:content},\ \$body) \\
(f_{msg}(\$y, \$body),\ \text{rdf:type},\ \text{:Message}) \quad \text{:-} & (\$msg1,\ \text{:recipient},\ \$x) \\
(f_{msg}(\$y, \$body),\ \text{:content},\ \$body) & (\$y,\ \text{:said},\ \$msg2) \\
(f_{msg}(\$y, \$body),\ \text{:recipient},\ f_{to}(\$y, \$body, \$x)) & (\$msg2,\ \text{:content},\ \$body) \\
 & (\$msg2,\ \text{:recipient},\ \$z)
\end{array}
$$

Figure 3.12: Composition of Q_1 over Q_2

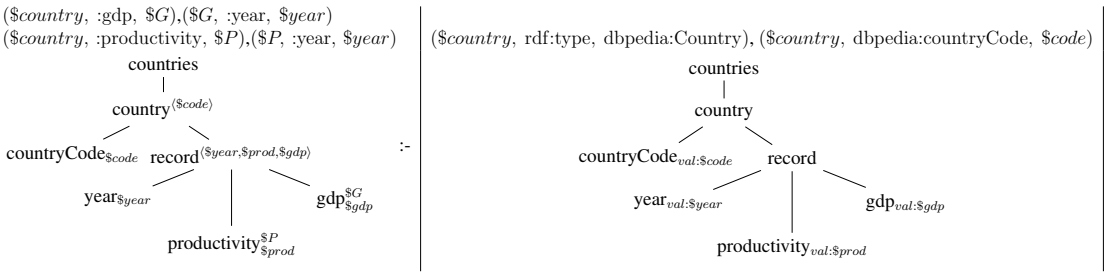


Figure 3.13: Query Q_3, integrating legacy XML and RDF into fresh XR

referring directly to its DBpedia URI.

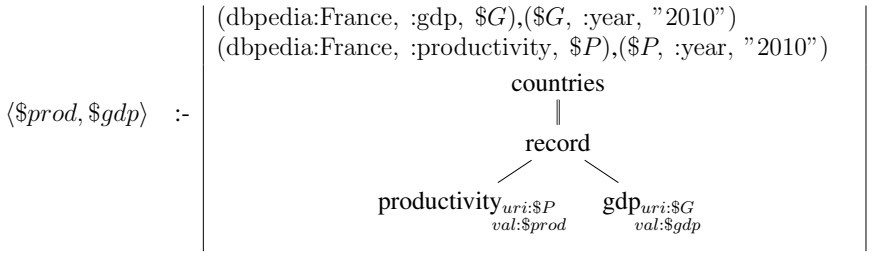


Figure 3.14: Query Q_4, retrieving information of a virtually integrated XR instance

Composing Query Q_4 over Query Q_3 yields the query depicted on Figure 3.15.

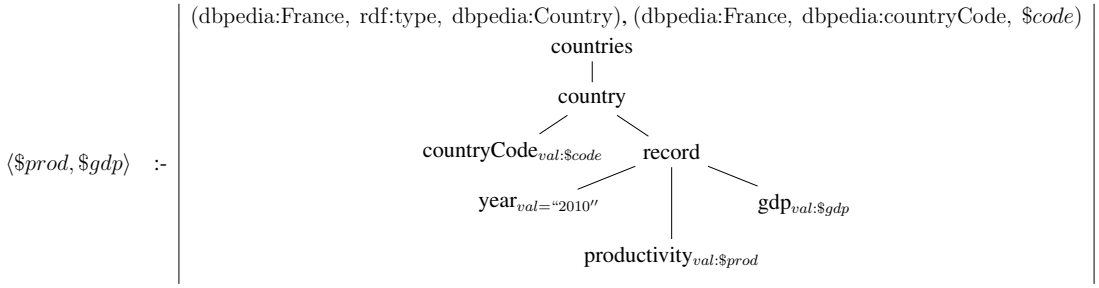


Figure 3.15: Composition of Q_4 over Q_3

3.3.6 Discussion

In the relational data management literature [AHV95], composition is defined as follows: given a *program* P with a final rule S, i.e., a set of rules whose heads define intensional predicates, and whose bodies are made of extensional and intensional predicates, a composition is a query q, s.t. for any database I, $q(I) = [P(I)]S$. For non-recursive programs, the problem is quite straightforward and amounts to expanding the body of the final rule S, i.e., replacing intensional predicates with the body of the corresponding rule, until it contains nothing but extensional predicates.

In XR, both the head and body of queries are made of triple and tree patterns or constructs, preventing the mere expansions that relations allow. In fact, our composition algorithms have strong connections with the problem of answering queries using views, where solutions require matching atoms of the query body with that of the body of views. Next, we discuss similarities between our approach and three well-known algorithms to find maximally contained rewriting using materialized views.

The inverse rule algorithm [DG97] guarantees to find, for a given a (possibly recursive) datalog program P and a set of views V, a maximally contained program P' whose rules body solely relies on V. The central idea of the algorithm is to turn V into a set of rules V', where for each view $v \in V$ and each atom $a \in v$, there is a rule in V' featuring a as head and the head of v as body. Skolem functions are used to replace non-distinguished

variables revealed in the head of such rules. P' is obtained by removing from P all rules using extensional predicates that do not appear in any view of V, and adding all rules of V' whose head match with a body atom of P. The final step of the algorithm consists in removing rules that rely on Skolem functions.

The bucket algorithm [LRO96] and its direct offspring, the MiniCon algorithm [PH01], also share resemblance with ours. The first phase of these algorithms is essentially the same as the matching phase, except that in those cases, body atoms of the query are matched with body atoms of the views. A bucket of matching view atoms, similarly to our substitutions sets Σ_i, is created for each query atom. Then, the algorithms diverge in the way these buckets are combined into conjunctive queries that *cover* the whole query. The bucket algorithm considers the whole Cartesian product of the buckets, while the MiniCon algorithm performs additional checks to reduce the number of combinations to explore. One of these optimizations consists in discarding candidates whose body may match some subgoal of the query, but whose head does not features the variables that would allow to join its results with that of views matching other subgoals of the query. Although our algorithm iterates on the Cartesian product of buckets, as in MiniCon, many entries can quickly be discarded by checking the consistency of $\sigma_\cup$.

This work also shares strong connections with an algorithm proposed by Le et al. [LDK+11], to rewrite queries on SPARQL views. In particular, the algorithm also comprises two phases, corresponding to our matching and building phases. An external algorithm is proposed to prune unsatisfiable sub-queries in the resulting union.

3.4 Conclusion and perspectives

In this chapter, we introduced the XR data model, for representing interconnected XML and RDF data and its query language XRQ. We detailed the syntax and semantics for the core language, returning tuples of variable bindings, and the extended language yielding XR data. The extended language enables composition for which we provide an early algorithm. The algorithm takes a query and a view as inputs and returns a query, which we proved correct w.r.t. the inputs.

The composition algorithm, along with tests, was implemented by Prachi Jain, who pursued an internship in our team. The algorithm is exponential in the size of the input query and would therefore require optimizations to be scalable in practice. We have already started to consider two types of optimizations: (i) many sub-queries in the resulting union can quickly be deemed unsatisfiable (e.g., when variable labels of a tree pattern violate its structure), (ii) the body of each sub-queries is made of variants of the view body, it should be possible in practice to minimize them, i.e., finding equivalent but tighter sub-queries, simply by looking at the body's variable labels. Finally, we believe the composition algorithm can be extended rather easily to the problem of finding a composed query in the presence of multiple views. These will be the focus of our attention in

the immediate future.

Next, we focus on query evaluation and optimization in XRP, a fully implemented storage and evaluation platform for XR.

Chapter 4

The XR platform

This chapter focuses on the practical aspects of XRQ query evaluation.

Evaluation & optimizations (Section 4.1). We explore this space of possible query evaluation strategies and present optimizations to speed up query processing.

Implementation (Section 4.2). 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.

Experimental results (Section 4.3). 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, few provide experimental results, and those that do [BDK+11] 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.

4.1 Query evaluation

This section discusses evaluation strategies for XRQ queries. Since there are by now many platforms for 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 4.1.1 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. Note that the type of result construction that XRQ_{ext} allows can be achieved linearly in the size of queries heads. Hence, for simplicity, in this section as well as the following one, we only consider core XRQ queries.

4.1.1 Preliminaries

For the clarity of the discussion, we 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.

We now 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 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 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 [www01] of XML nodes. A deterministic method assigning an XURI to every node from a given document is termed a *labeling 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 [www10], the closest notion to XURIs is that of node identity, which by default is implicit [1]. Most XDMs [TVB+02, Rys05] (including recent ones [CBC+12]) 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 XEval):

XURI-out: the outputs of XEval include the XURI of each XML node participating in this result.

XURI-in: given an XURI as input, XEval is capable of recognizing the (unique) XML node having this XURI. In other words, XEval can perform selections on XURI values, thus 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 4.2. The algorithms we present next have specific requirements in terms of XDM hypotheses, as we explain in each case.

What to delegate? The XRQ processor delegates sub-queries for evaluation to the underlying XML, respectively, RDF engines. As explained in Section 3.2.2, 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 [HFLP89]. 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 [AM08]. 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 in-

1. The W3C's xml:id recommendation [www05] 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 4.2).

voked to determine which fragments of these queries to 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.

Our evaluation strategies include a simple approach where Q_X and Q_R can be parallelized and a set of *information-passing* strategies, where bindings travel at query evaluation time from Q_X to Q_R or *vice versa*. Strategies of this family can be further broken down into three groups: (i) bindings are passed one-by-one to the target query, requiring no particular rewriting, (ii) bindings are sent in a single pass, requiring the target query to be rewritten into a union, (iii) bindings are materialized in the target sub-instance, in which case, the rewritten target query does not feature any union and is thus more amenable to query optimization.

4.1.2 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 $\mathsf{XML||RDF}$, 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 2 outlines the $\mathsf{XML||RDF}$ strategy.

Algorithm 2: $\mathsf{XML||RDF}$

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 $T_{XR} \leftarrow \pi_h(T_X \bowtie T_R)$

Example. Recall the query in Figure 3.2, 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:

Algorithm 3: 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

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

$(\$A = \#205, \$B = \#305, \$C = \#303,$
$\$CA = \langle$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:

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

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)

4.1.3 Information-passing algorithms

4.1.3.1 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 3, 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 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) $\cup$
$Q_R(\$X, \$Y,$"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. [2]

Note that the SPARQL 1.1 recommendation [www13a] 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.

4.1.3.2 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

2. 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.

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 [XLWB09] is a recent representative. Alternatively, an XURI-to-XPath index can be materialized to support dereferencing through a look-up by the XURI.

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 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 4 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 3.2, where for simplicity we only consider the first XML tree pattern Q_X^1, and the full Q_R. An XQuery serialization of Q_X^1 is:

Algorithm 4: 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

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

```
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_X^1, 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 5 outlines RDF⇒XML-XPath. Here, the function *PushJoins* is slightly modified w.r.t. Algorithm 4: 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 5 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_X^1 transforming it into:

Algorithm 5: 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**
  $t'_R \leftarrow \mathit{Deref}(t_R)$
  $q \leftarrow PushJoins(t'_R, Q_X)$
  $T_{XR} \leftarrow T_{XR} \cup \pi_h(\mathsf{XEval}(q, I_X))$

```
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 [CDO08, Kar12].

Algorithm 6 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,
```

Algorithm 6: 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 7: 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**
  $t'_R \leftarrow Deref(t_R)$
  $UCQ \leftarrow UCQ \cup PushJoins(t'_R, Q_X)$
$T_{XR} \leftarrow \pi_h(\mathsf{XEval}(UCQ, I_X))$

```
    $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 7 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 $A=#205 and $A=#405, dereferenced into `/microblog/message[12]/body[1]` and `/microblog/ message[22]/body[1]`, respectively. In this case, Algorithm

Algorithm 8: 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 $T_{XR} \leftarrow \mathsf{XEval}(Q'_X, I'_X)$

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)
```

4.1.4 Materialization-based algorithms

4.1.4.1 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 8 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.

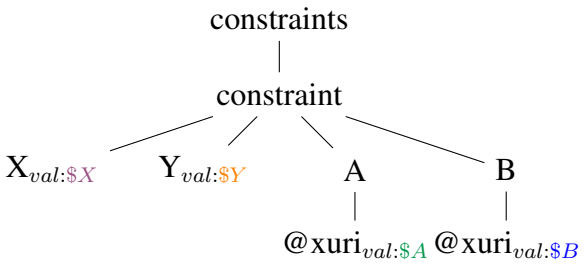


Figure 4.1: Additional tree pattern added to Q_X

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)
```

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 4.1. 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. URI variables are typed as VAL variables in the newly added tree pattern and automatically cast to URI variables at evaluation time. Another way to proceed would be to introduce an `xuriref` attribute whose semantics would be inspired from `ID/IDREF` attributes. An element endowed with an `uriref` attribute of value $\#x$ would act as a reference to the resource of URI $\#x$. In such a case, the variable of the tree pattern of Figure 4.1 would not need their type to be modified.

Algorithm 9: 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 $T_{XR} \leftarrow \mathsf{REval}(Q'_R, I'_R)$

4.1.4.2 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 9 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 9 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.

4.1.5 Pruning optimizations

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 [Hid03].

Algorithm 10 (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, 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 3.2, and the tree pattern Q^2_X 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^2_X, 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 kind of pruning, leading to the respective variants RDF→XML-XPath-Pr and

Algorithm 10: 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 $T_{XR} \leftarrow \emptyset$
2 $T_R \leftarrow \mathsf{REval}(Q_R, I_R)$
3 **foreach** *tuple* $t_R \in T_R$ **do**
4 valid:=true
5 $t'_R \leftarrow$ *Deref*(t_R)
6 **foreach** *tree pattern* tx_i *of* Q_X **do**
7 Let $\$V_i^1, \$V_i^2, \ldots, \$V_i^{k_i}$ be the XURI variables of tx_i which are bound in t_R to the XURIs $v_i^1, v_i^2, \ldots, v_i^{k_i}$, 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** $d_i^1 = d_i^2 = \ldots = d_i^{k_i}$ **then**
11 // Check compatibility between the linear XPaths and paths of the respective variables in Q_X:
12 **foreach** $\$V_i^j$, $1 \leq j \leq k_i$ **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 $q \leftarrow PushJoins(t'_R, Q_X)$
17 $T_{XR} \leftarrow T_{XR} \cup \pi_h(\mathsf{XEval}(q, I_X))$

RDF→XML-Data-Pr.

When both **XURI-in** *and* **dereferencing** are supported, one may apply the same pruning technique as presented in Algorithm 10, and push XURIs directly into the XML subqueries rather than the dereferenced nodes (lines 16 and 17). This variant comes in two flavors, RDF→XML-URI-Pr and its tuple-at-a-time counterpart RDF⇒XML-URI-Pr.

Figure 4.2 systematizes the XRQ evaluation algorithms discussed so far.

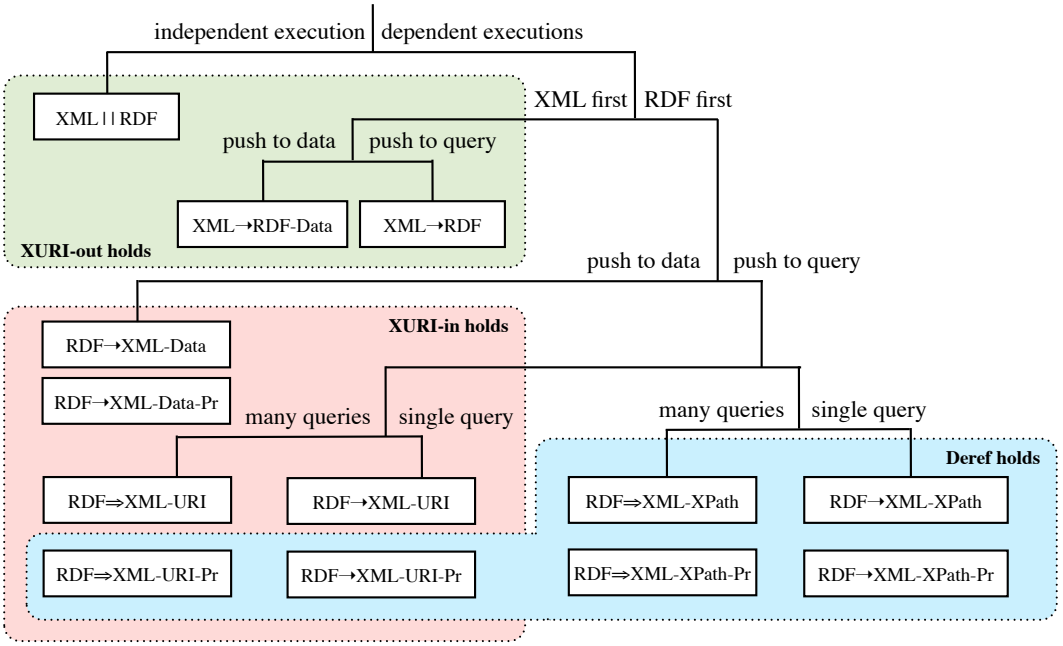


Figure 4.2: Taxonomy of the proposed XRQ query evaluation algorithms

4.2 Implementation

We implemented the XR platform in Java 1.6 (16.000 lines); Figure 4.3 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.

4.2.1 Existing wrappers

As RDF query engines, we have experimented with RDF-3X [NW10], 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 recent XML store, which we found to be competitive w.r.t. QizX and MonetDB in tests that we ran comparing them on the XMark [SWK+02] and XPathMark [Fra05] 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**.

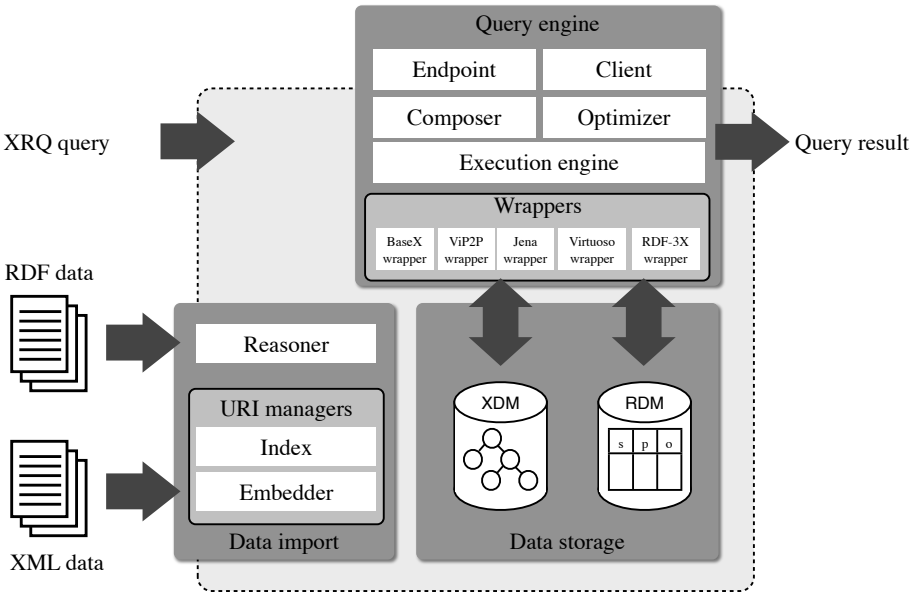


Figure 4.3: Architecture of the XR platform

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 [KKMZ12] (see also `http://vip2p.saclay.inria.fr`), which we had developed in the group. ViP2P supports the XML tree pattern dialect introduced in Section 3.2.

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 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 [MKVZ11], 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 [3].

Wrappers for Jena and Virtuoso have been implemented later on to be used in the

3. 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 thesis.

application described in Chapter 5. The main rationale for resorting to Virtuoso in this case, was the relatively good support the system provides for updates compared with RDF-3x.

4.2.2 XR 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 4.1. 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 [Gra90] of our operators trigger them). Our tests have shown that such parallel, eager *fetch* execution significantly speeds up the query evaluation. This is because the *fetch* operators ship potentially complex subqueries to the underlying XDM and RDM, thus their evaluation is a significant part of the overall processing time.

4.2.3 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 reassembles with dURI into n's full XURI. The module inserting such IDs is the *embedder* in Figure 4.3.

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 [GKK+11a].

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]`. 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 [BDB]) 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 4.1, and produces an execution plan implementing the respective strategy for that query. As explained in Section 4.1.2, 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 that 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.

4.2.4 Reasoner

In its first version, the XR platform reasoner exclusively relied on forward chaining to take account of RDFS-entailed triples. Recall that in this context, query answering amount to query evaluation. Thus, the techniques and algorithms described in Section 4.1 can directly be applied onto a *saturated* XR instance. Likewise, all experimental results presented in the next section cover query evaluation only. We have studied other techniques for query answering, based on bitmap indexes (detailed in Chapter 6) and materialized views [GKLM11b], will be integrated to the platform in the future.

4.2.5 Endpoint

The platform features an endpoint, i.e., a Web interface to the query engine. In the spirit of a SPARQL endpoint, the Web application allows a user to interact with the query engine through a form, where she is invited to type in a query in a text area, specify the name of the instance on which the query will be evaluated, and an output serialization format. The application can also be accessed in a RESTful manner, in which case all these parameters are passed through an HTTP request, and the results are written directly to HTTP response. The endpoint was not use directly in the experiments described in Section 4.3, but the application presented in Chapter 5 heavily relies on it.

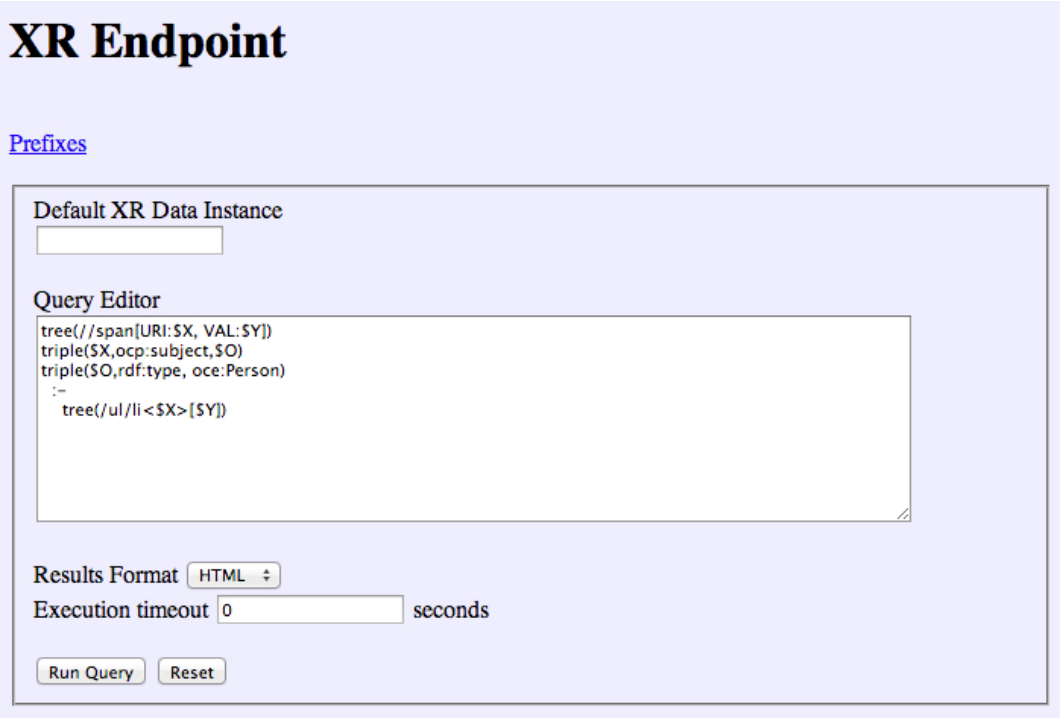


Figure 4.4: Screenshot of the XR endpoint

4.2.6 Composer

Finally, a query-view composition module, implementing Algorithm 1 (Section 3.3) was last added to the platform.

4.3 Experimental evaluation

This section presents the findings of our experimental study. Section 4.3.1 describes the experimental settings we used to test our algorithms. Section 4.3.2 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 4.3.3 focuses on the more efficient ones, and studies their scalability with respect to the size of the data instance. In Section 4.3.4, we compare these algorithms based on two quite different XDMs, then we conclude.

4.3.1 Experimental setting

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

First, we used the XMark [SWK+02] XML document generator to produce a set of XML documents. Second, we generated 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 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 [wwwc], 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 chapter, 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 [wwwd].
- 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 $\mathcal{D}_j^i$ the dataset obtained by setting the XMark input size to i and the RDF-to-XML ratio to j. For instance, $\mathcal{D}_{1/3}^{10}$ 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 factors are reported in Table 4.1.

Workloads. We handcrafted 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.

We briefly explain what each query does in W_1:

- Q_1 filters on the type, quantity and price of featured items that appeared in closed

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 4.1: XR datasets used in the experiments

auctions, where the auction nodes are annotated.

- Q_2 filters on the type and seller of items with a quantity of one appearing in open auctions, where the auction nodes are annotated with two distinct properties.
- Q_3 filters on the ID, location and category of items (first tree pattern), and the names of people in Germany who have an interest in the same category (second tree pattern). Moreover, the item nodes have to be annotated.
- Q_4 finds the type and bidder's name of items in open auctions (first tree pattern), and the age, name and email address of people in the United-States selling these items (second tree pattern). The items must be annotated with a least one triple.
- Q_5 has a three-way join between tree patterns. It retrieves the people who have been both sellers and bidders, who bid on a fix date (10/14/2000) and whose bidder's node is annotated.
- Q_6 has a chain-join between three patterns, finding names and locations of people watching open "regular" auctions, where the items being sold are located in Africa. In addition, the person's node must have an annotated chain of length two.
- Q_7 filters interests one hand, and watched auctions on the other hand, while these nodes have to be linked with a chain annotated of length two. The query features a Cartesian product in Q_X, but is connected when considered as a whole.
- Q_8 on the contrary features a Cartesian product among Q_R triples. The two triple patterns are non-selective, but they must annotated the item and category nodes of items being sold.

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 4.2, BaseX satisfies neither **XURI-in** nor **XURI-out**, and to be able to test all our

	Q_X	Q_R
W_1	LOW	LOW
W_2	LOW	HIGH
W_3	HIGH	LOW
W_4	HIGH	HIGH

Table 4.2: Workload relative selectivities

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.

Methodology. To cover all possible configurations, we reused and adapted a methodology and tool set developed for a prior project [GKLM11b]. Our evaluation program outputs logs at regular time intervals or on specific events (e.g., test failure). Each log record contains information about the test being performed, e.g., the configuration used, the start time or whether a timeout was reached. It also features data about the platform on which it runs, like the amount of memory used. Finally, the logs contain context-specific strings that can be used to filter the proper subset of records that is relevant to a given aspect of the experiments. For instance, lines containing *aggregate* results, such as the average run time of a given query over n executions, are prefixed with "CLUSTERED", while lines containing atomic information are prefixed with "SIMPLE".

Tests were run on distinct machines, with identical software and hardware, in parallel to minimize the overall time requirement. Logs from each machine produced in this fashion can be easily merged for later treatment. After the tests completion, we collected the samples and analyzed them with a set of scripts. The scripts are simple loops, in which each test configurations were considered. We used each configuration to filter the corresponding records from the logs, produced a visual chart with gnuplot [4], and finally outputted the results into HTML/SVG pages [wwwd]. This procedure allowed producing a large set of visualizations, to understand the overall behavior of our algorithms. In the sequel, we present some selected results that we believe are the most relevant to our discussion.

4. `http://gnuplot.info`

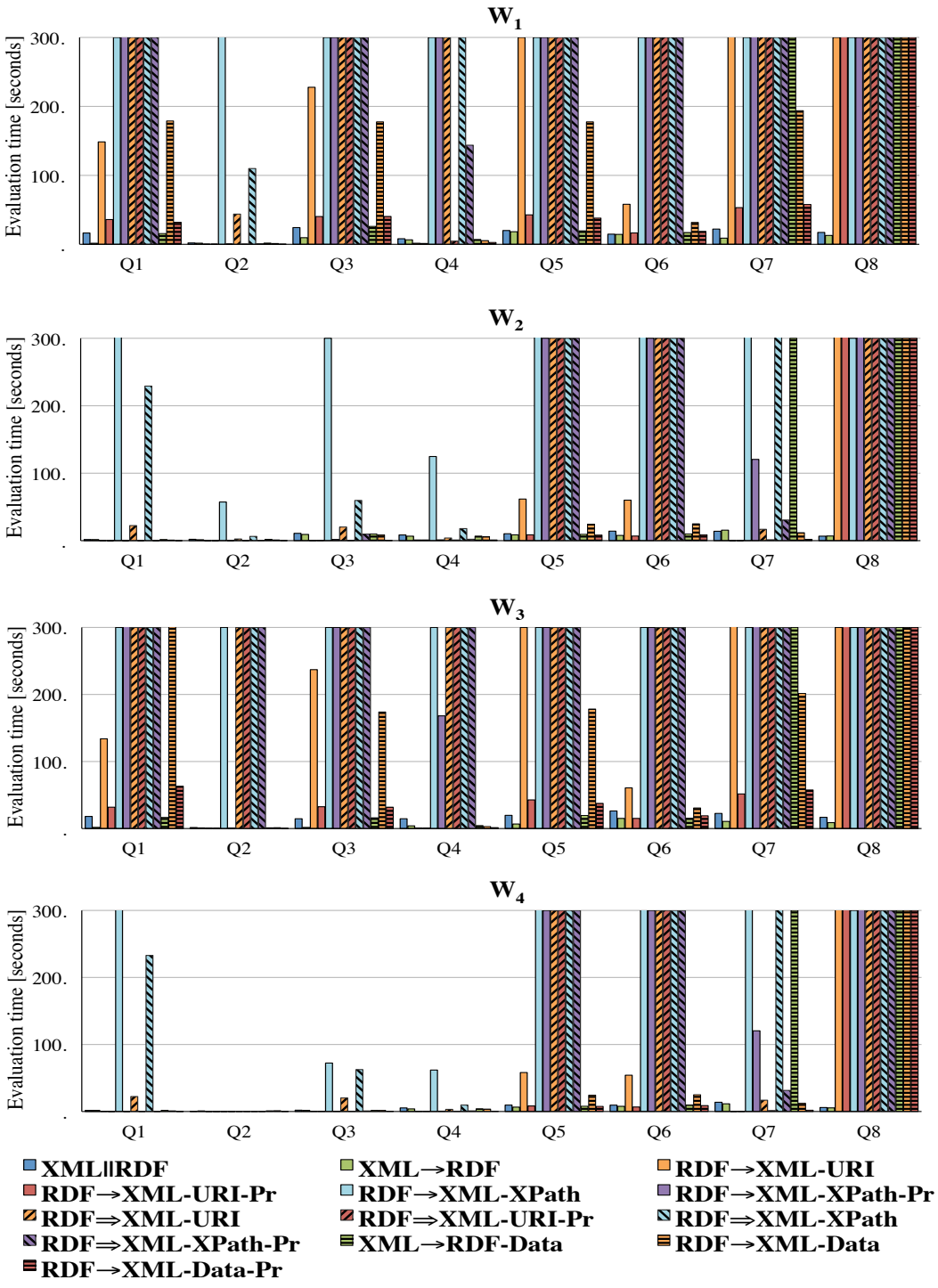


Figure 4.5: XR query evaluation strategies compared on workloads $W_1 - W_4$ and dataset $\mathcal{D}_1^1$

4.3.2 Comparison of all strategies

Our first set of experiments compares all the strategies described in Section 4.1, 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 4.5 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 time is longest on the Q_i from W_1. Similarly, W_4, featuring selections both in the XML and RDF subqueries, 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 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 sizable 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 poorly, for all but a few selective queries in W_2 and W_4. Among the worst are RDF⇒XML-URI (Algorithm 6) and RDF⇒XML-XPath (Algorithm 7), 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.

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 these experiments and similar others, we decided to exclude RDF→XML-XPathand the tuple-at-a-time RDF-to-XML strategies from further tests, and we only consider the strategies showing acceptable performance in Figure 4.5, 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.

4.3.3 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 varying sizes, ranging from $D^1_{1/3}$ to $D^{100}_{1/3}$ (the exact cardinality characteristics of these datasets are listed in Table 4.1). 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 4.6 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 behavior 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 some filtering on the XML sub-instance, even in the absence of passed XURIs (as we have explained in Section 4.3.1).

For the more complex queries $Q_5 - Q_8$, Figure 4.6 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. 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

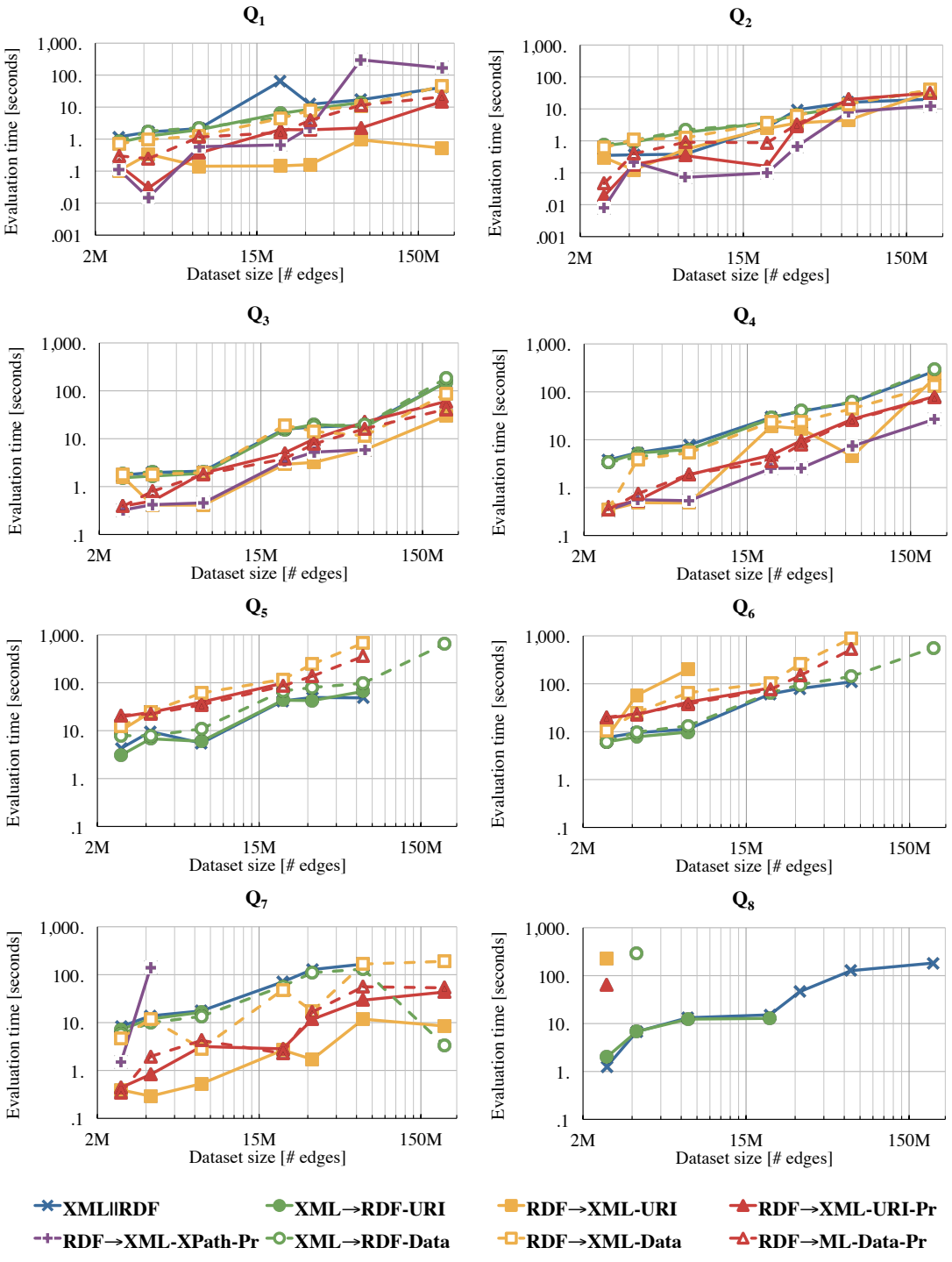


Figure 4.6: Evaluation times for W_4 with datasets of increasing sizes

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 4.1.1.

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 also RDF-3X statically detects that the final query returns an empty result.

4.3.4 Experiments using VIP2P

The last experiments we present compare two 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 4.2.1. 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 [MKVZ11].

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 that 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. [BÖB+04, MKVZ11], are likely to be gradually included in popular XML databases as they mature.

Figure 4.7 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 4.5, 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 4.7, 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

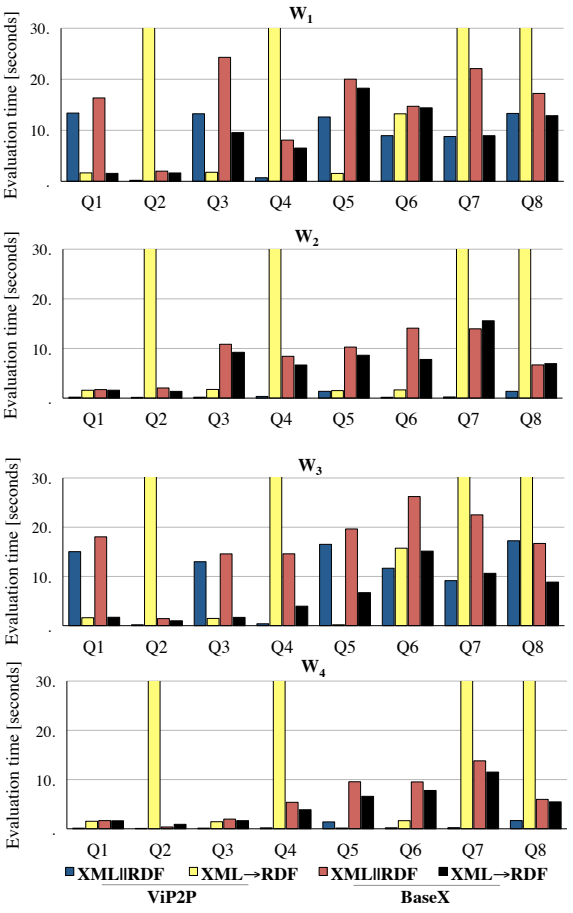


Figure 4.7: Evaluation times for workloads W_1 to W_4 on dataset $\mathcal{D}_1^1$ using ViP2P and, respectively, BaseX

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 [wwwd]. 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.

When XML→RDF times-out on ViP2P, XML→RDF on BaseX runs typically fast! This is because, as explained in Section 4.3.1, *the XURIs sent by BaseX to the RDM are only those of nodes referred to by the RDF sub-instance*. Therefore, the unexpected behavior

of RDF-3X is not triggered [5].

4.3.5 Experiments wrap-up

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 the best on small data instances (Figure 4.5), and are robust (especially XML||RDF) up to very large data instances (Figure 4.6). 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 4.6, 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.

4.4 Conclusion

In this chapter, we discussed query evaluation strategies available when XML and RDF are considered together. We presented the XR platform, a XR storage and query evaluation platform, implementing the full range of algorithms previously introduced. Finally, we presented an in-depth evaluation campaign for this family of algorithms. So far, we have purposely kept the problem on RDF query answering under RDFS-entailments out of the discussion. Chapter 6 specifically addresses this problem. The approach we present applies to RDF in general. Our results, however, directly carry over to XR.

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

Chapter 5

Fact checking and analyzing the Web with XR

We introduce FactMinder, an application leveraging the XR platform to empower data journalist and online fact checkers. The application is implemented as a browser plugin, i.e., a special panel in the navigator featuring focused content, enabling users to better understand, analyze and verify the claims made on pages they browse online. The focused content area is mainly a *hierarchical layout* of XRQ views, which we nicknamed XR Information Panel (or *XIPs*). XIPs can cross information from the pages browsed by the users and any other openly available XML or RDF datasets. Users can also manually annotated pages, e.g., to point to arguments supporting or disproving a claim, and publish these annotations online for others to see.

In the next section, we detail our motivation for building FactMinder. Then, we present its architecture (Section 5.2) before describing the user-facing modules (Section 5.3).

5.1 Motivation

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 [1]) and use online

1. `http://wikileak.org`

services (such as Twitter [2] or Google Maps [3]) to integrate and corroborate facts found online. Journalists have become data publishers themselves as witnessed in sites such as The Guardian [4], FactCheck [5], and Politifact [6]. However, as skillful as these professionals may be, their work is still very manual as demonstrated by Storyful [7] founder in a recent presentation [8] and, as of today, they lack powerful tools for analyzing, consuming and producing data.

The main intent behind FactMinder is to provide to such professionals with a first-hand tool to help them *search* for information and *crosscheck* with openly available data set. As they move forward into their investigation, users can publish the results of their analysis for others to see, and reuse in the own work.

5.2 Architecture

FactMinder follows a client-server architecture, detailed in Figure 5.1. In this Figure, boxes depict the FactMinder modules. Solid ovals represent the tasks performed by the application automatically, whereas dashed ovals are the tasks performed by users inside and outside the system.

The FactMinder client is made of three components, an *information extractor*, a *rich browser* and a *dashboard*, illustrated by the screenshot in Figure 5.2, each playing a different role in the fact checking process. When a document is opened within the client, the information extraction module (such as OpenCalais [wwwi]) automatically finds the topics, entities and relationships it contains. Documents are opened in the rich browser, where the user can manually add or edit annotations. The dashboard is made of views over the XR database with the background information, each rendering a specific aspect of the information at hand. These views are easily customizable; the user iteratively refines them through the GUI when performing an analysis task. To pursue her investigation, the user switches back and forth between the rich browser and the dashboard. Any insight she gets from the interface may lead her to add some detail to the document as a new annotation, or to refine a view and get a better understanding of the data. FactMinder uses an XR server to integrate XML and RDF from the Web, and store the annotations created both automatically and manually. The server runs on top of BaseX 7.3 [wwwa] for managing XML data, and the open-source edition of Virtuoso 6.1.6 [wwwk] for the RDF data. Bold arrows in Figure 5.1 denote data flows between each component of the system. Content created during the investigation is stored in the database for future use.

2. `http://twitter.com`
3. `http://maps.google.com`
4. `http://guardian.co.uk/data`
5. `http://www.factcheck.org`
6. `http://www.politifact.org`
7. `http://storyful.com`
8. `http://on.ted.com/MarkhamNolan`

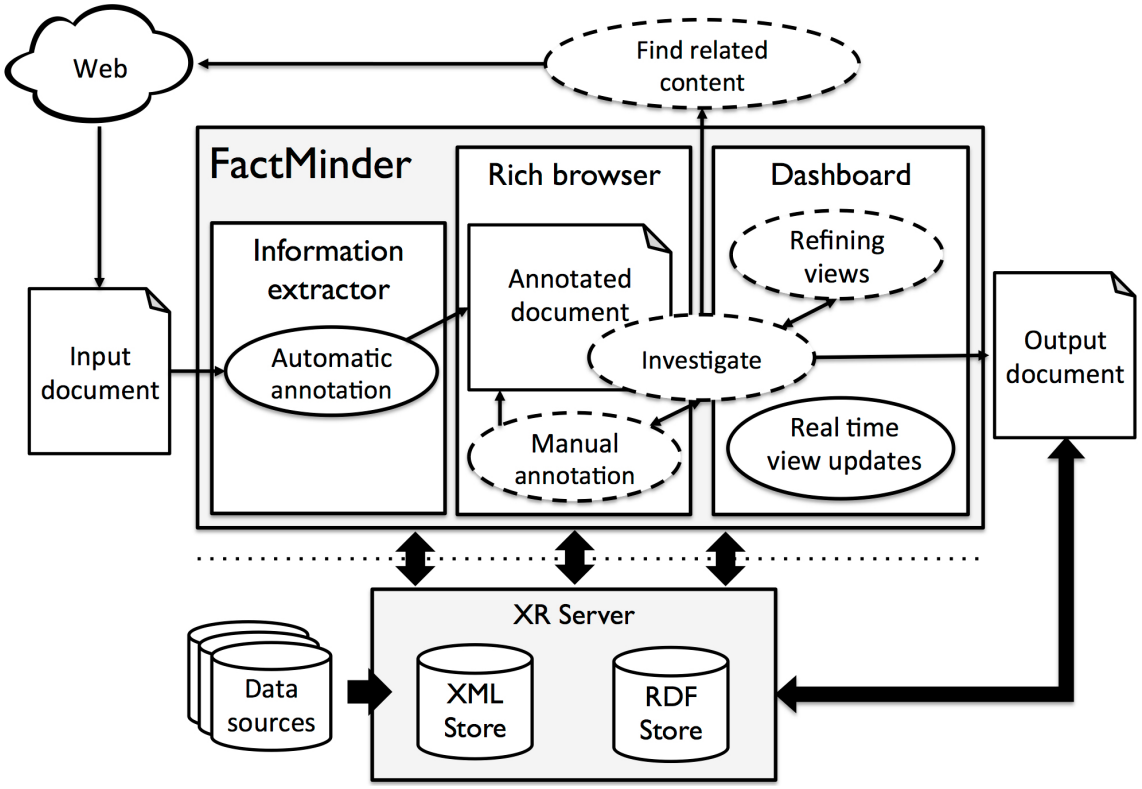


Figure 5.1: FactMinder architecture

The analysis may lead the user to find related contents, run additional documents through the same analytical process, etc.

Next, we detail how users interact with the application.

5.3 User Interface

The screenshot in Figure 5.2 exemplifies the main UI components assuming a scenario where a prospective PhD student want to verify facts about the OAK research team before applying for a position.

The rich browser. In this area, users can open documents by entering their location, either on the local machine or on the Web. Unlike conventional browsers, it provides a rich set of annotation tools. First, the annotations produced by the information extractor upon opening the document are accessible to the user by hovering over the text. Second, the user can add her own annotations in a faceted editor by selecting some content, specifying comments and new knowledge, and enriching or correcting existing annotations. Com-

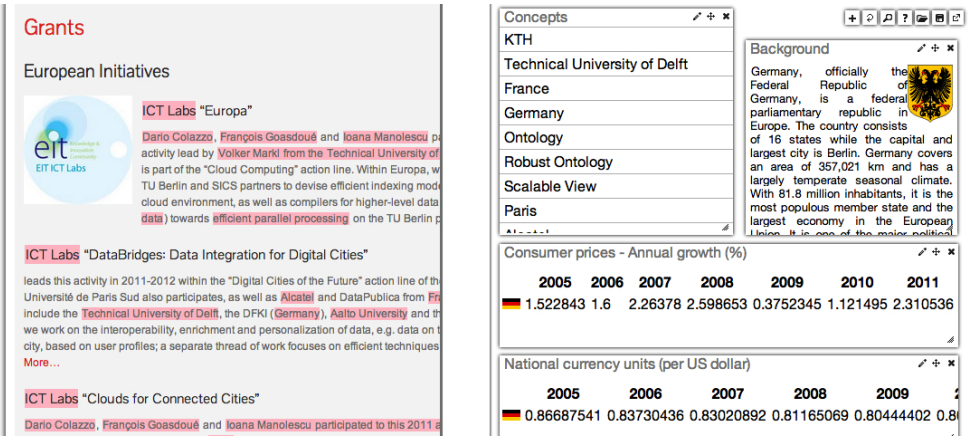


Figure 5.2: Main FactMinder window

ments contain, e.g., a text body, creation date, author and category, such as "FalseClaim", "uri_1 confirms this", etc.

The dashboard. The dashboard area is composed of XR info panels (XIPs, for short). These panels assist the user in understanding the content she is working on; each panel is dedicated to one aspect of the information under scrutiny. The information content of an XIP is gathered through an *XRQ parameterized view* over the data in the browsing panel and/or the background information (the preexisting XR database). Semantic connections may exist among XIPs used simultaneously (much in the way the content in part of a Web page changes according to the user interaction with the rest of the page). For instance, the default XIP, shown in Figure 5.2, comprises a "Concepts" XIP, at the center top of the figure, featuring all the unique concepts appearing in the currently selected document.

XIPs are highly customizable, and users can add, delete, and rearrange them at any time. Users define new XIPs by opening a "new panel" editor (detailed below), providing the XIP name and the associated XR query.

Dependent XIPs. By default, a XIP is refreshed automatically when the user opens or selects a new document. Moreover, the re-computation of an XIP can be governed by the user interaction with another XIP; in this case, we call the former *dependent*, and the latter *parent*. For instance, in Figure 5.2, the "Background", "Consumer prices" and "National currency" XIPs depend on the "Concepts" XIP. Selecting an item in the list, here "Germany", causes the XIPs to be displayed, presenting information gathered by crossing data from the page itself with XML data from public OECD datasets and RDF data from DBpedia.

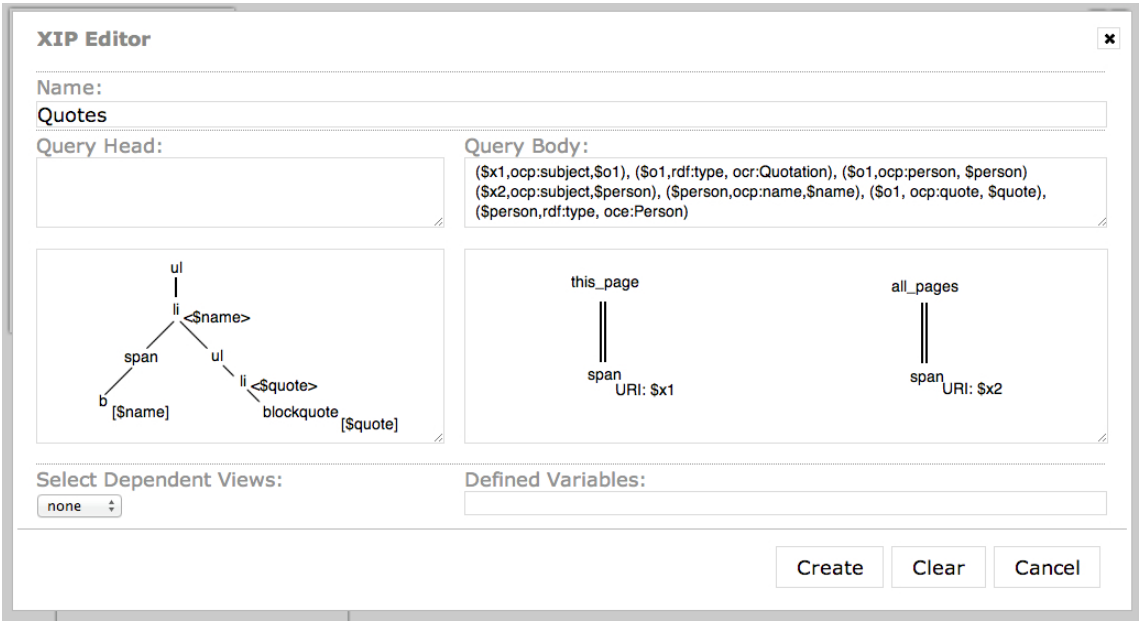


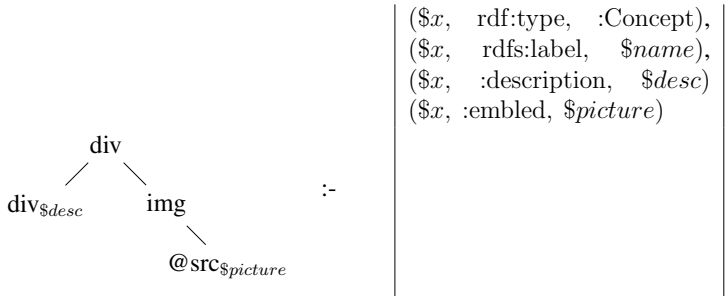
Figure 5.3: View editor

Editing XIPs. To create or edit an XIP, the user relies on a graphical editor, such as the one illustrated in Figure 5.3. An XIP editor opens up, for building XR queries and viewing them in a graphical form. To create dependent XIPs, the user simply needs to designate an XIP among the list of existing ones at the bottom of the editor. The user is then presented with the list of the variables returned by the parent XIP that can be reused to establish connections among dependent views.

Semantic template XIPs. One can easily write concept-specific XIPs. For instance, the default facts for a :Country may include the historical or economics background, whereas for a :Person, the relevant facts may be the age, nationality and profession. XR's support for RDFS semantics (e.g., subclass relationships) allows one to adapt queries to the context. Suppose that XIP "Concept" of Figure 5.2 has the following query definition:

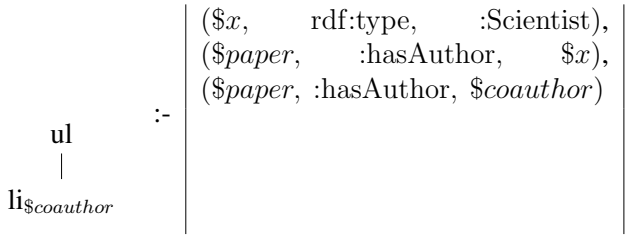
$$\begin{array}{c} \text{ul} \\ | \\ \text{li}_{\$label} \end{array} \quad :- \quad \left| \begin{array}{c} (\$x,\ \text{rdf:type},\ \text{:Concept}),\ (\$x,\ \text{rdfs:label},\ \$label) \\ \text{html} \\ \| \\ \text{span}_{uri:\$x} \end{array} \right|$$

Next, one can define an XIP called "Background" depending on "Concept", displaying focused information about the item currently selected in "Concept" and referring to any :Concept:



Notice that variable $\$x$ appears in the queries of both XIPs. By selecting an item from "Concept", the value bound to $\$x$ in that item is passed to the dependent query before executing it. The dependent query only yields a result if the resource bound to $\$x$ has type :Concept.

Further, assume that the user wants to show specific information if the selected item is a :Scientist. He can customize the template by creating a second XIP depending on "Concept", for instance to include information about scientists. The new XIP's query could look like this:



When the selected item in the parent XIP refers to a :Scientist, both dependent XIPs will be refreshed, but only those returning a result will eventually be displayed. This is illustrated in Figure 5.4. After selecting "François Goasdoué" from the list of concepts, his co-authors and publications are displayed. In this case, the information is gathered from DBLP, a dataset of Computer Science publications. However, there is currently no information on the selected item in DBpedia, thus the "Background" panel is not displayed as it yields no result.

5.4 Conclusion and perspectives

Growing computing power and storage capabilities are reshaping the profession of journalists. The problems of finding, aggregating, visualizing and validating facts online are making their ways into the Computer Science research community, as witnessed by

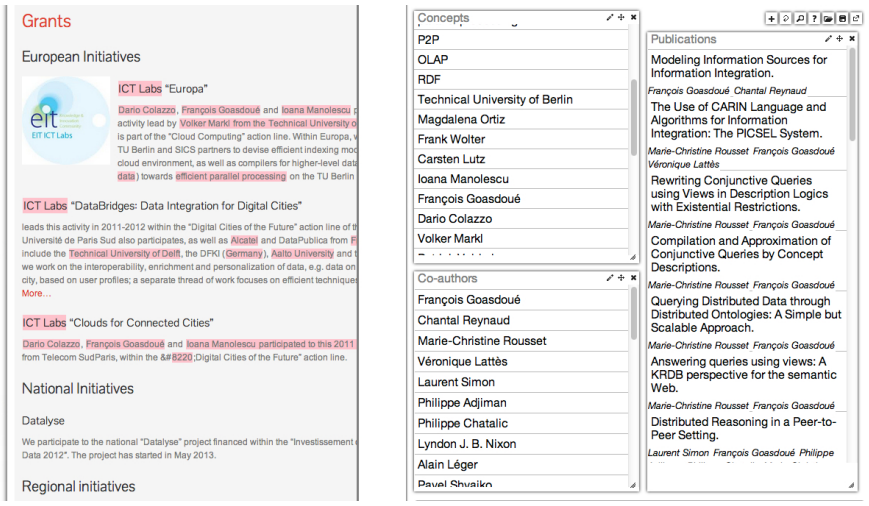


Figure 5.4: FactMinder showing scientist-specific XIPs

recent publications [CHT11, CLYY11]. We believe that XR would be a powerful tool for future research in the field.

To support this claim, we designed FactMinder, a rich browser application relying on the XR platform, with the aims of breaking the barrier that stands between online content, which is highly heterogeneous, incomplete and inconsistent in nature, and trusted XML and RDF data sets.

An important part of the vision behind FactMinder is that annotations produced by users should be shared among them. As users may express different opinions on similar topics, confronting their views could bring valuable contributions to the area of sentiment analysis and crowd-sourcing. A forthcoming PhD thesis will specifically focus on the study of social interactions in XR.

Acknowlegements. FactMinder's user interface was implemented by Stamatis Zampetakis, co-authors and PhD candidate of the OAK team. The background data used by FactMinder was collected and curated by Andrés Aranda Andújar, engineer since 2011 in our group.

Chapter 6

Answering SPARQL queries with bitmap indexes

In this chapter, we introduce a novel approach for SPARQL query answering under RDFS entailments. It essentially consists in a storage model and query evaluation strategies that can be used in a wide range of existing engines, while mitigating some of the pitfalls associated of the traditional techniques of forward and backward chainings.

The key idea is to store as *sets* all classes a resource belongs to, and all properties that exist between a pair of RDF nodes. These sets are stored in bitmap indexes. Single *synthetic* facts that represent all classes a resource belongs to (resp. all properties between two nodes) are stored in a relational table. When a query is evaluated, the bitmap indexes are consulted to modify an execution plan such that it remains optimizable with existing algorithms.

Section 6.1 introduces the methods that have been proposed to date for answering queries under RDFS entailments. We detail our general approach in Section 6.2 under the assumption that all implicit facts of a data instance have already been derived. In Section 6.3, we discuss how additional indexes can be used to avoid such preprocessing of the data. Section 6.5 shows how our technique affects storage space and query evaluation time for commonly used data sets. Section 6.6 covers the related works. We conclude this chapter by discussing future works in Section 6.7.

6.1 SPARQL query answering under RDFS entailments

Several approaches have been proposed for RDF query answering, some of them directly imported from deductive database research.

Forward chaining. The first, and arguably the most widely used technique in commercial systems, consists in *materializing* the RDFS-*closure* of the data. By applying the

RDFS entailments rules *forward*, new RDF triples are generated and added to the original data set. Each newly created triple may in turn trigger entailment rules. Thus, the process completes when the RDF instance reaches a fixpoint, i.e., no new fact can be derived.

The main advantages of this technique are that (i) once the closure is materialized, query answering can take place using conventional evaluation and optimization techniques, (ii) the materialization is performed *off-line* and thus does not interfere with the query evaluation process. However, materializing the closure requires additional storage space. As we will see, the space overhead is polynomial in the size of the schema. Moreover, in the presence of updates, the closure may become inconsistent in which case it needs to be properly updated. The problem of maintaining the closure of an RDF instance is still actively studied [SB05, BKO$^+$11, GHV11, GMR13]. In the worst case, i.e., when the schema is updated rather than the instance, the closure may have to be entirely recomputed.

Backward chaining. At the other end of the spectrum, it is possible to leave the data untouched and reason *at query time*. This is done by applying the entailment rules *backward* onto a conjunctive query to form a *reformulated* query, i.e., a union of conjunctive queries whose result will include all derivable data. Although this obviously saves from the space and maintenance requirements of forward chaining, the query reformulations grow exponentially in the size of the input query, making them hard to optimize and evaluation in practice.

Depending on whether the data or schema changes over time, it may be advantageous to choose one technique rather than the other. A recent work [GMR13] explores this kind of trade-offs and proposes an efficient technique to maintain the RDFS-closure incrementally, when updates occur.

Other approaches. Other techniques, such as magic sets [BMSU85] have been applied to RDF [KMK08]. They lie in-between forward and backward chaining, by attempting to derive a minimized set of intermediary facts that are relevant to the answer of an input query.

Recent works [MAYU05, RMC11, UvHSB11] have shown there are other viable options between these forward and backward chaining for answering SPARQL queries. In this chapter, we present a storage model that can be used in conjunction with a forward chaining approach or with a variant of semantic indexes introduced in [RMC11]. A semantic index is a table encoding the hierarchies in an ontology using a DAG labeling scheme. At query evaluation time, the index is used to rewrite each atom to a SQL range query retrieving all the classes (resp. properties) that are subsumed by a given class (resp. property). They present the advantage being amenable to any relational database system. In previous works however, certain types of atoms, such as triple patterns where a variable stands in place of a class or a property, cannot be handled easily.

In this work, we redefine semantic indexes as bitmap indexes and explain how to create

an execution plan that rely on them, e.g., in a context where using forward chaining is not an option. Our method is a variant of the scheme proposed in [CPST03] but dismissed for efficiency reasons. We mitigate this issue by storing bitmap indexes in structures separate from the triple table itself.

6.2 Overview

6.2.1 Data storage

A described in Section 2.2.2.2, a dictionary-encoded triple table is a relational table T with three attributes s, p and o (for $subject$, $property$ and $object$) containing a record for each fact of the data instance. RDF facts fall into two categories: (i) *class assertion* of the form $(:\mathrm{Alice},\ \mathrm{rdf{:}type},\ :\mathrm{Person})$, which assign a type to a resource, (ii) *property assertions* of the form $(:\mathrm{Alice},\ :\mathrm{knows},\ :\mathrm{Bob})$, which define a relationship between two resources, or between a resource and a value. In practice, the triple table can be partitioned along these two categories to reduce the time required to retrieve triples of either type, but for simplicity, we will only refer to T hereafter.

Facts are atomic in nature, therefore, a resource "$:Alice$" belonging to n classes will be stored as n records. Similarly, if resources "$:Alice$" and "$:Bob$" are linked with m properties, there will be m records to represent those facts. This type of redundancy is common in real-world datasets and the process of materializing the closure of the data typically makes it worse. However, it is possible to mitigate this by storing a single *synthetic* fact for all classes a resource belongs to (resp. for all property between two nodes). Let $\mathcal{D}$ be an RDF data instance containing extensional *and* intensional facts w.r.t. RDFS entailment rules.

Definition 6.2.1 (Concise class)**.** *Let C be an ordered set of all classes in $\mathcal{D}$, and x a resource in $\mathcal{D}$. $C[i]$ denotes the class at position i in C. The* concise class *of x is a bitmap where every bit at position i is set to 1 if $(x,\ \mathrm{rdf{:}type},\ C[i]) \in \mathcal{D}$ and to 0 otherwise.*

Definition 6.2.2 (Concise property)**.** *Let P be an ordered set of all properties in $\mathcal{D}$, x a resource in $\mathcal{D}$ and y a resource or a literal in $\mathcal{D}$. $P[i]$ denotes the property at position i in P. The* concise property *of the pair $(x,\ y)$ is a bitmap where every bit at position i is set to 1 if $(x,\ P[i],\ y) \in \mathcal{D}$ and to 0 otherwise.*

Note that the orders of C and P are arbitrary and fixed over time. We compute concise classes for all resources in $\mathcal{D}$ and concise properties for all pairs of nodes in $\mathcal{D}$, and assign a unique ID number to each distinct concise class and property. We define a special concise property representing the set {rdf:type} to which we assign the ID 0. We store concise classes (resp. concise properties) and their IDs in a bitmap index, called CLIDX (resp. PRIDX). Henceforth, we will write CLIDX[i] (resp. PRIDX[i]) to denote the concise class (resp. property) obtained by looking up CLIDX (resp. PRIDX) for the ID i.

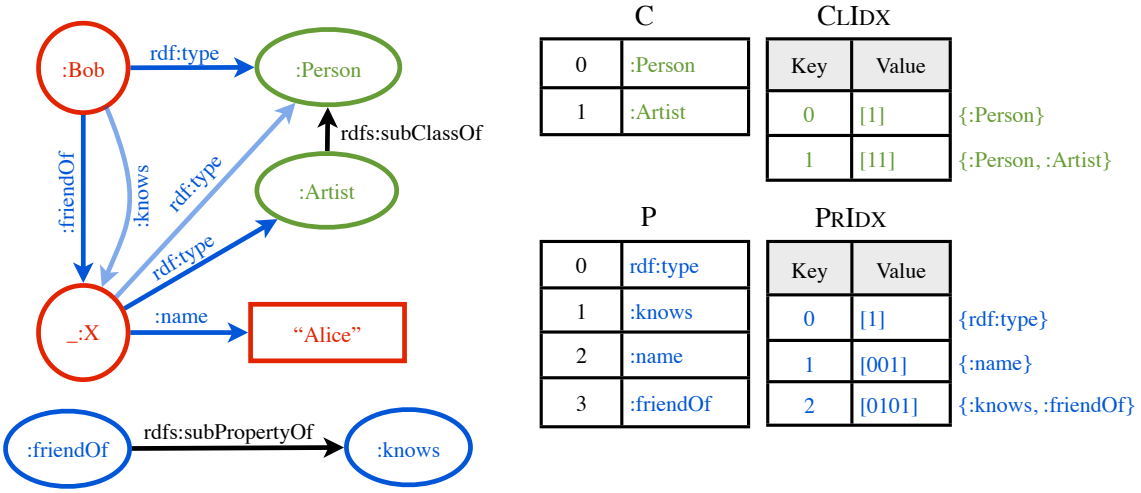


Figure 6.1: An RDF graph with the corresponding C, P, CLIDXand PRIDX.

Example. Figure 6.1 depicts a small RDF graph along with the corresponding C and P tables, and CLIDX and PRIDX indexes. The graph contains the following knowledge: Bob is a person, he is a friend of (and therefore knows) someone named Alice, who is an artist (and therefore a person too). Classes are colored in green, properties in blue and the remaining nodes in red. RDFS-related edges are left in black, and derived edges are in light color. There are two distinct classes (:Person and :Artist), and four distinct properties (rdf:type, :knows, :name and :friendOf). The sets on the right hand side of CLIDX and PRIDX are those encoded in the bit vectors of the same line. Note that only the subsets of classes and properties that appear in the graph are stored in CLIDX and PRIDX.

Finally, we create a dictionary-encoded triple table T' containing for each resource r with a non-empty concise class c, the triple $(Key(r),\ 0,\ Id(c))$, where $Key(r)$ is the key of r in the dictionary and $Id(c)$ is the ID of c in CLIDX. Similarly, for each pair of nodes (x, y) such that there exists a non-empty concise property p between x and y, we store the triple $(x,\ Id(p),\ y)$, where $Id(p)$ is the ID of p in PRIDX. We call such records *concise facts*.

Example. Figure 6.2 shows on the left hand side a raw triple table storing the knowledge of the graph depicted in Figure 6.1. This table contains six records. On the right hand side, we show the concise triple table T', containing four records, and its dictionary. T' contains keys to either the dictionary, CLIDX or PRIDX. In fact, all keys of the p column refer to PRIDX, and if the value of p for a given record is 0, then its object key refers to CLIDX, the dictionary otherwise.

Advantages of using bitmaps. Each concise class (resp. concise property) represents a subset of C (resp. P). Bitmaps are commonly used in databases for representing subsets as they can be efficiently compressed and set operations translate to simple bitwise

Raw triple table

S	P	O
:Bob	rdf:type	:Person
:Bob	:friendOf	_:X
:Bob	:knows	_:X
_:X	rdf:type	:Person
_:X	rdf:type	:Artist
_:X	:name	"Alice"

T'

S	P	O
0	0	0
0	2	1
1	0	1
1	1	2

Dictionary

Key	Value
0	:Bob
1	_:X
2	"Alice"

Figure 6.2: Raw triple table representation of the graph depicted in Figure 6.1 (left) and its final encoding (right).

operations. Many compression schemes are now available. In this work, we focused on schemes that allow for bitwise operations without decompression [CDP10, DP10]. The best compression ratios are usually obtained on sparse or dense bitmaps, as long chains of zeros or ones can be drastically summarized. The sizes of C and P are generally small compared with the size of the data instance. Moreover, concise classes and properties represent small subsets of C and P, making for sparse bitmaps that easily fit in memory once compressed, even when one considers the closure of the data instance. Since trailing zeros are not stored at all, compression rates can be further improved by ordering C and P by decreasing frequencies of classes and properties in the instance.

6.2.2 Query answering

The following query, which will be our running example, retrieves the names of artists, by matching the first triple pattern with all statements with a :name property, the second triple pattern with all statements typing the subject as an :Artist, and joining them by their subjects:

```
SELECT ?x ?y ?z
WHERE { ?x :name ?y.
        ?y ?w ?z.
        ?x rdf:type :Artist.
}
```

In a relational setting, a typical execution plan for a conjunctive query is made of selection, projection, join and scans operators, respectively $Select$, $Project$, $Join$ and $Scan$. Our example query would require three scan accesses to the same relation T, as depicted in Figure 6.3, where the numbers in the projection and join operator are column indexes to their respective children.

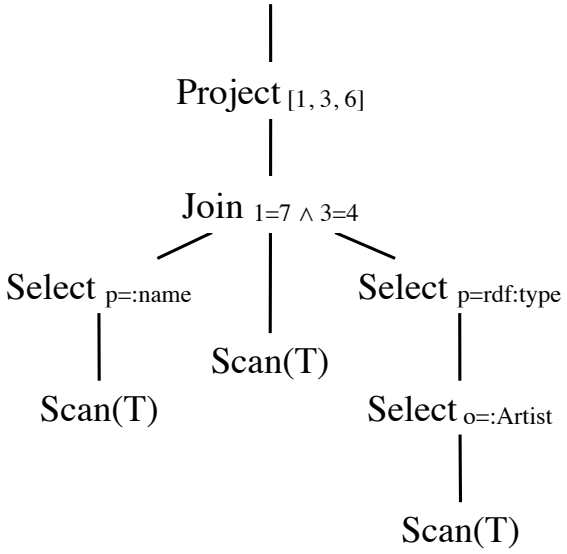


Figure 6.3: An execution plan for our running example

We now define a new scan operator, $Scan'$, that traverses the triple table and reconstructs the actual facts from the concise classes and properties read. For each record t of the form $(s,\ p,\ o)$ of T', $Scan'(T')$ performs the following:

1. If $p = 0$, i.e., t is a class assertion, then for each class $k \in \textsc{ClIdx}[o]$, the triple $(s,\ \text{rdf:type},\ k)$ is returned.
2. If $p \neq 0$, then for each property $k \in \textsc{PrIdx}[p]$, the triple $(s,\ k,\ o)$ is returned.

Observe that executing $Scan'(T')$ produces the same result as the $Scan(T)$, assuming T contains no duplicate record.

Next, we explain how to produce a plan to execute over the concise triple table T', rather than the triple table T. Let q be a conjunctive SPARQL query. For each triple pattern r in q:

1. If r is of the form $(?x,\ ?y,\ ?z)$, add the operator $Scan'(T')$.
2. If r is of the form $(?x,\ \text{rdf:type},\ ?z)$, add the operator $Scan'(T')$ as a child of a selection $Select_{p=0}$.
3. If r is of the form $(?x,\ \text{rdf:type},\ k)$, add the operator $Scan(T')$ as a child of a selection $Select_{p=0 \wedge o \in S}$, where S contains IDs of concise classes to which the class k belongs, more formally $S = \{Id(i) \mid \textsc{ClIdx}[i] \cap \{k\} \neq \emptyset\}$.
4. If r is of the form $(?x,\ k,\ ?z)$, where $k \neq \text{rdf:type}$, add the operator $Scan(T')$ as a child of a selection $Select_{p \in S}$, where S is the set of IDs of concise properties to which the property k belongs, formally $S = \{Id(i) \mid \textsc{PrIdx}[i] \cap \{k\} \neq \emptyset\}$.

Remaining selections, projections and joins are handled as usual. The construction of S is achieved by a single traversal of CLIDX or PRIDX. The intersection operation translates to a bitwise AND between compressed bitmaps. Overall, these sets are inexpensive to build and they can easily be cached.

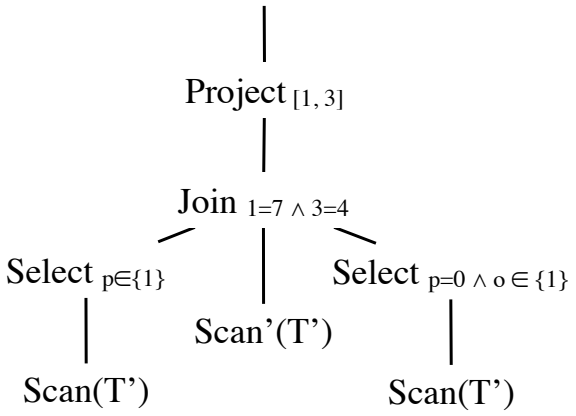


Figure 6.4: An execution plan relying on T'

Example. Figure 6.4 shows a plan for our running query relying exclusively on T'.

6.3 Semantic index-based approach

We now turn the cases when one cannot materialize the closure as part of the data, e.g., due to space constraints or if the data is subject to frequent updates. The process is reminiscent of semantic indexes [RMC11], however, in our setting, these are redefined as bitmap indexes.

6.3.1 Data storage

The procedure to build the concise triple table, CLIDX and PRIDX is the same as the one described in Section 6.2 except that $\mathcal{D}$ now only contains extensional facts. The semantic indexes are built by computing the *terminological closure*, i.e., the closure on the facts that belong to the schema. More precisely, we compute for each class, the sets of its sub-classes, its super-classes, the properties of which the class is in the domain and the properties of which it is in the range. The four sets associated with each class are encoded into bitmaps using the procedure described in Section 6.2.

We store these sets in four new indexes, named SUBCL, SUPCL, DOMCL and RNGCL. We proceed similarly for each property to obtain the sets of its sub-properties, super-properties, domains and ranges, which we store in indexes SUBPR, SUPPR, DOMPR and

RNGPR. We use the same notation as for CLIDX and PRIDX to refer to the sets contained in these indexes. For example, SUBCL[c] is the set of sub-classes of c.

6.3.2 Query answering

First, we define two new scan operators $Scan''$ and $Scan_c$, where c is a constant passed as parameter. For each record t of the form $(s,\ p,\ o)$ of T', $Scan''(T')$ performs the following:

1. If $p = 0$, i.e., t is a class assertion, then for each $k \in S$ where $S = \bigcup_{i \in \text{CLIDX}[o]} \text{SUPCL}[i]$, the triple $(s,\ \text{rdf:type},\ k)$ is returned. S is the union of super-classes of the classes encoded in o.
2. If $p \neq 0$, then
   - for each $k \in S_1$, where $S_1 = \bigcup_{i \in \text{PRIDX[p]}} \text{SUPPR}[i]$, the triple $(s,\ k,\ o)$ is returned,
   - for each $k \in S_2$, where $S_2 = \bigcup_{i \in \text{PRIDX[p]}} \text{DOMPR}[i]$, the triple $(s,\ \text{rdf:type},\ k)$ is returned,
   - for each $k \in S_3$, where $S_3 = \bigcup_{i \in \text{PRIDX[p]}} \text{RNGPR}[i]$, the triple $(o,\ \text{rdf:type},\ k)$ is returned.

The operator $Scan_c$ behaves essentially like $Scan$ with the following exceptions. For each triple t in T' of the form $(s,\ p,\ o)$:
- if $p = 0 \wedge o \in \text{SUBCL}[c]$, it returns the triple $(s,\ \text{rdf:type},\ c)$,
- if $p \in \text{DOMCL}[c]$, it returns the triple $(s,\ \text{rdf:type},\ c)$,
- if $p \in \text{RNGCL}[c]$, it returns the triple $(o,\ \text{rdf:type},\ c)$,
- otherwise, t is ignored.

Both $Scan''$ and $Scan_c$ may produce duplicates. These can be avoided using simple caching techniques.

Next, as we did in Section 6.2.2, we explain how to use these operators in the presence of semantic indexes. Let q be a conjunctive SPARQL query. For each triple pattern r in q:

1. If r is of the form $(?x,\ ?y,\ ?z)$, add the operator $Scan''(T')$.
2. If r is of the form $(?x,\ \text{rdf:type},\ ?z)$, add the operator $Scan''(T')$ as a child of a selection $Select_{p=0}$
3. If r is of the form $(?x,\ \text{rdf:type},\ k)$, add the operator $Scan_k(T')$.
4. If r is of the form $(?x,\ k,\ ?z)$, where $k \neq \text{rdf:type}$, add the operator $Scan(T')$ as a child of a selection $Select_{p \in S}$, where S contains the IDs of concise properties to which a sub-property of k belongs, i.e., $S = \{Id(i) \mid \text{PRIDX}[i] \cap \text{SUBPR}[k] \neq \emptyset\}$.

Remaining selections, projections and joins are handled as usual.

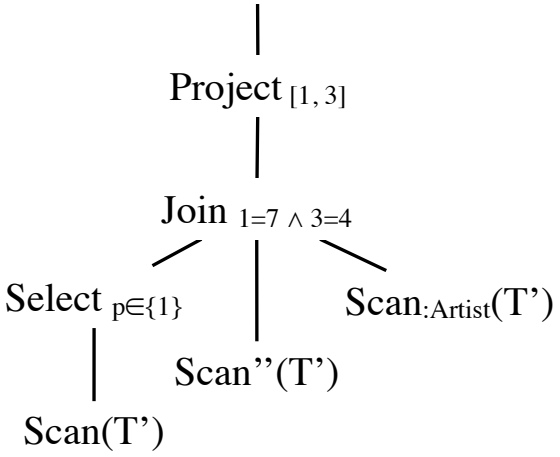


Figure 6.5: An execution plan relying on T' in the presence of semantic indexes

Index	Max. length	Max. width
CLIDX	$2^{\|C\|}$	$\|C\|$
PRIDX	$2^{\|P\|}$	$\|P\|$
SUBCL	$\|C\|$	$\|C\|$
SUPCL	$\|C\|$	$\|C\|$
SUBPR	$\|P\|$	$\|P\|$
SUPPR	$\|P\|$	$\|P\|$
DOMCL	$\|C\|$	$\|P\|$
RNGCL	$\|C\|$	$\|P\|$
DOMPR	$\|P\|$	$\|C\|$
RNGPR	$\|P\|$	$\|C\|$

Table 6.1: Upper bounds for indexes sizes

Example. Figure 6.5 shows a plan for our running query relying exclusively on T' in the presence of semantic indexes.

6.4 Discussion

Time & space complexity of indexes construction. The upper bounds for the uncompressed indexes sizes can be easily determined. They are listed in Table 6.1.

For all indexes other than CLIDX and PRIDX the length is in fact fixed, and given by the cardinalities of C and P. The upper bound for the length of CLIDX and PRIDX is large, however in practice, it tends to be close the $|C|$ and $|P|$ respectively. The reason is that resources generally belong to only a few classes at the same time. Hierarchies are graphs, generally acyclic, and often simply trees. When a resource belongs to multiple classes

that lie on a common branch of the hierarchy, only one records needs to be stored. When the data is saturated, the bitmap representing the classes of a resource also contains all their ancestors. Otherwise, if no derivable class is actually stored, only the most specific classes of a resource are represented. Thus, the length of CLIDX is therefore identical in both cases [1] (saturated or not). The same remark applies to PRIDX. A recent empirical study [DKSU11] shows that $|C|$ and $|P|$ are often orders of magnitude smaller than data instances themselves.

Compressing the indexes. In our setting, all bitmaps indexes are compressed, and in most case, we can expect a good reduction of the total space required for storing them. Since our operators rely heavily on set operations, we are particularly interested at compression schemes that are amenable to bitwise operations without decompression. In our implementation, we used Concise [CDP10] a word-aligned compression technique that falls into this category. This compression scheme achieves its best performances for sparse or dense bitmaps, but both bit density and distribution are important factors. In the worst case, i.e., when a bitmap is partially filled and bits are evenly distributed, it requires $|C|/31$ 4-bytes words of space in CLIDX (resp. $|P|/31$ in PRIDX).

There are different reasons why this limit is unlikely to be reached. First, as we already mentioned, resources tends to belong to few classes at a time, and any pair of resources is in relation through few properties. Thus, one can expect bitmaps in both CLIDX and PRIDX to be generally sparse. Hierarchies also tend to be broad rather than deep. For indexes encoding the terminological closure, each bitmap contains either the ancestors or descendant of a given class or property in the hierarchies. As a result, super-class and super-property bitmaps are likely to be sparse, while sub-class and sub-property bitmaps density is undetermined and can range from very sparse to very dense.

However, since the order C and P is fixed, but arbitrary, one can influence compression ratios simply by reordering them. For instance, by ordering classes in decreasing order of frequency in the data instance, bitmaps in CLIDX will tend to be denser towards the most significant bit with long sequences of trailing zeros that can easily be ignored. This simple observation can also be applied to PRIDX.

6.5 Implementation and experiments

We implemented the storage model and the plan operators in Java 1.6.0_29 (64bits). All our tests were performed on a single machine with 8 Intel Xeon CPUs running at 2.13GHz with 4096 KB of cache each. We allocated 2GB of RAM to the virtual machine.

1. This is not true anymore if the data is only partially saturated however

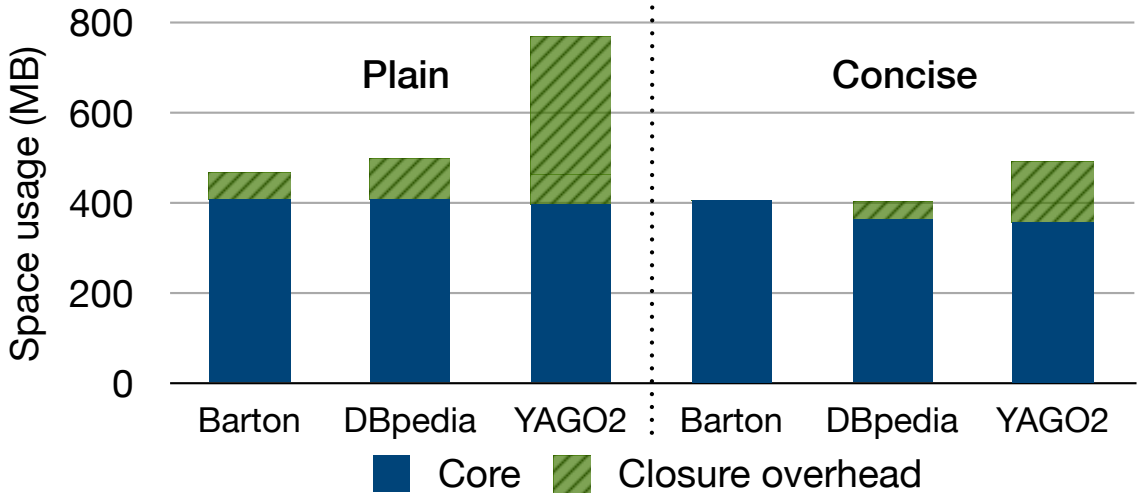


Figure 6.6: Space usage (in MB) of the triple table for 3 datasets with plain facts (left) and concise facts (right)

6.5.1 Space requirement for tables and indexes

We first compare the space required to store the triple table with three widely used datasets: Barton, DBpedia and YAGO2. We chose these datasets for two main reasons. First, they come with very different schemas. The terminological closure of the schema used with Barton counts 201 statements, while the schemas of DBpedia and YAGO2 contain 7,745 and 3,983,638 statements respectively. Second, after cleaning the data and computing the *core*, i.e., the set of facts that cannot be derived through entailments [GHMP11], each dataset featured approximately 33M facts. This allows comparing how materializing the closure affects space usage. Figure 6.6 shows the amount of space (in MB) occupied by the plain triple table (left), against the concise triple table along with CLIDX and PRIDX (right)[2]. The blue (dark solid) bars represent the core data, while the overhead incurred by the materialization is shown in (light hatched) green. The size of the schema has a clear impact on storage space with a conventional approach. For instance, YAGO2 almost doubles in size as a result of materialization, jumping from 33M to 64M triples. On the other hand, the concise approach reduces this overhead to nearly 33% for YAGO2. The core also occupies less space with a concise approach because groups of facts are summarized into single ones. The space occupied by the closure of DBpedia is thus comparable to that occupied by the core dataset in a conventional triple store.

2. The dictionary tables are not included on the figure.

	Schema size	Space (KB)	Time (sec.)
Barton	201	17	0.07
Wordnet	524	11	9.56
Uniprot	1,283	39	0.13
Dbpedia	7,745	329	0.54
Sumo	52,909	1,505	7.11
Galen	217,724	2,565	2.18
Gene Onto.	532,118	4,465	6.91
YAGO 2	3,983,638	44,868	72.82

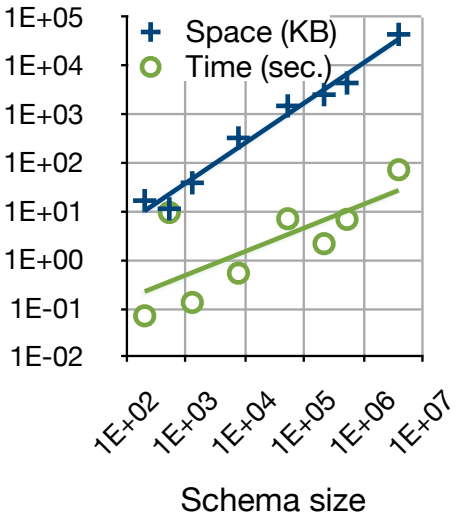


Figure 6.7: Time (in seconds) and space (in KB) required to build and store indexes for ontologies of varying sizes in memory

6.5.2 Semantic indexes construction

Next, we take a look at the space and time required to build the bitmap semantic indexes. For this, we used a set of freely available ontologies of varying sizes. The table in Figure 6.7 shows their sizes (first column), as the number of triples in the terminological closure. The second and third columns detail the total space occupied by the compressed indexes in memory (in KB) and the time to compute them (in seconds), respectively. The graph on the right displays the same figures on a scatter plot. Notice the log scale on both axes. The data shows that the time and space requirements grow sub-linearly in the size of the schema and remain reasonable for nowadays systems. YAGO2, the largest schema, loads in 72 seconds and fits in 45MB of memory only.

6.5.3 Impact on query evaluation

To assess how query evaluation performs in this model, we ran four queries on the three datasets mentioned before. Each query comprised one of the triple patterns described in Sections 6.2.2 and 6.3.2. Q_1 and Q_2 are of the form $(?x,\ ?y,\ ?z)$ and $(?x,\ \text{rdf:type},\ ?z)$ respectively. Q_3 corresponds to $(?x,\ \text{rdf:type},\ k)$, and Q_4 to $(?x,\ k,\ ?y)$, where k is the class (resp. property) that belongs to the most concise classes (resp. properties), i.e., the most adverse cases for evaluation. Figure 6.8 reports evaluation times (in seconds) averaged over 5 runs. Plain refers to the evaluation on a conventional triple table, Concise-1 refers to the concise triple table with materialized closure, Concise-2 to the semantic index approach. Tables were partitioned across typing and relationship statements. For

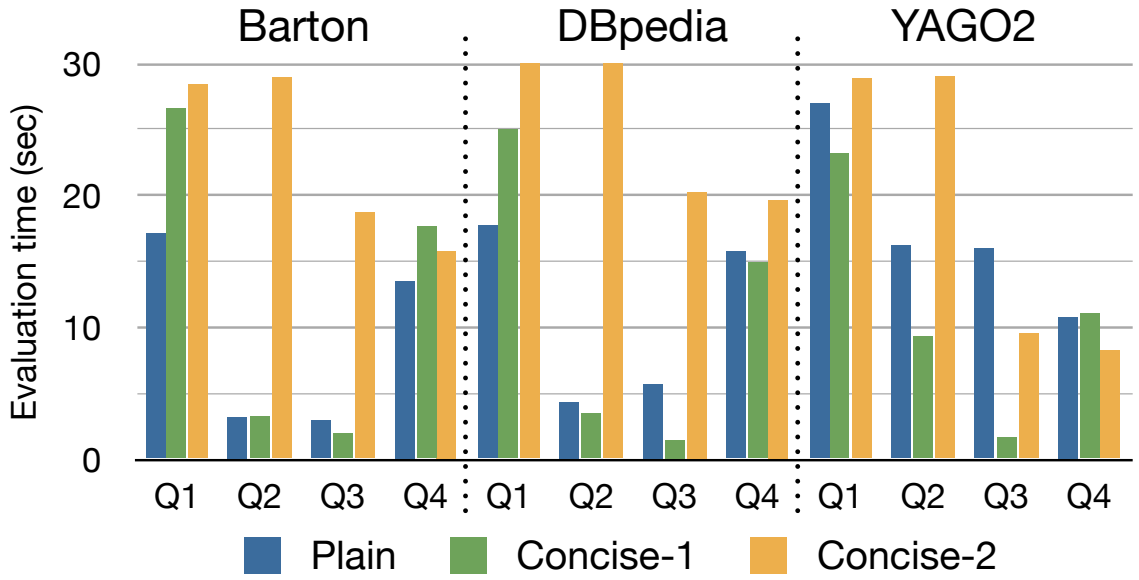


Figure 6.8: Query run times (in seconds) for 4 simple queries

most queries and datasets, Concise-1 performs better than a conventional approach, as the size of the scanned tables dominates the processing cost of the new operators overall. With Concise-2, Q_2 and Q_3 are penalized as they need to scan both partitions to collect all possible results. However, this is amortized for Q_3 with YAGO2 as the plain triple table becomes prohibitively expensive to scan. We plan to explore optimizations in the presence of B+tree indexes and with parallelization in future work.

6.6 Related works

In [CPST03], the idea using bit vectors as a labeling schema for RDFS subsumptions checking is evoked, along with two others, a path-based and a Dewey labeling schemes. The paper compares tradeoffs between the two latter schemes, for various hierarchies and query types but dismisses the bitmap-based approach, considering it requires O(n) for subsumption checking. In this work, we mitigate the problem as the bitmap index is not placed on the triple table itself, but in external structures (CLIDX and PRIDX) containing only the subsets of classes and properties that appear in the data. Although the space upper bounds of these tables are large, we observed that in pratice they are much smaller than the triple table. The triple table stored index keys pointing to those structures, and can therefore be indexed as any integer column. To the best of our knowledge, the use of bitmap index for RDFS query answering had not yet been studied in detail.

In [UvHSB11], Urbani et al. observed that by computing the *terminological closure*, one can reduce the backward-chaining phase at query evaluation time. *Semantic indexes*,

presented in [RMC11], go a step further by storing the terminological closure in indexes and the assertion facts in relational tables. The indexes map each class (resp. property) to pairs of integers obtained through a traversal of the schema hierarchies. Rewritings of all possible triple patterns to SQL range queries are cached, by-passing the need for backward chaining. This method is the closest to ours. The idea of path-based indexing of an RDF terminology was introduced in [MAYU05].

Bitmaps have been used in other storage models. In [ACZH10], they are used to store an RDF dataset as a cube and process joins with the objective of avoiding large intermediate results. Zou at al. [ZMC+11] used bitmaps in gStore to model an in-memory graph database. Fernández et al. [FMPG+13] rely on bitmaps to compress RDF for data exchange over networks. In [Erl], Orri Erling explains how the commercial system Virtuoso makes uses of bitmap indexes to speed up triples loading time, and reduce space usage. The Virtuoso documentation also claims to support reasoning through backward chaining, by means of special operators that include derivable data at execution time. A technique that can be compared with the query answering approach presented in 6.3.2.

A series of works investigate ways to mitigate the pitfalls associated with these techniques in a distributed setting, such as Peer-to-Peer [KMK08] or Map-Reduce [UKOvH09, UKM+10, UvHSB11]. In a previous work, we explore an approach based materialized view selection [GKLM10a, GKLM11a, GKLM10b, GKLM11b]. This was also one of the main topics of a separate thesis [Kar12], which the interested reader is invited to check for further details.

6.7 Conclusion and outlook

In this section, we introduced a storage model that (i) reduces the storage space required for an RDF data instance, (ii) suffers little space overhead when the closure is materialized in the instance, (iii) improves query evaluation time in the general case, (iv) can be used in conjunction with semantic indexes when forward-chaining must be avoided. Although we presented the model as a variant of a triple table setting, it could also be applied to other storage types such as property tables [wwwe] or RDF cubes [ACZH10].

Our next focus will be on query optimization, e.g., in the presence of indexes, as described in [NW08]. The method will be extended to support other entailment regimes such as the OWL 2 EL Profile and queries on the terminology itself. We think our approach would be particularly well suited to answer queries on RDF data streams and for analytical query processing.

Chapter 7

Conclusion

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 with RDF annotations allows not only exploiting the text better, but also illustrating and connecting the resources and concepts of the database.

7.1 Summary

While many works have focused on devising automatic and semiautomatic text annotation tools, drawing on Natural Language Processing capabilities, we have considered the problem of modeling and efficiently querying such corpora of interconnected documents, facts and concepts. Our first goal was to reuse 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 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 (Fig-

ure 4.2), 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 strategies, and that the most efficient (XML||RDF and XML→RDF) scale up well. 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 [GKLM11b] and conjunctive tree pattern queries [KMV12]. We introduced new ways to answer queries under RDFS-entailments. The techniques, introduced for RDF, naturally extend to the XR model.

We integrated the XR platform into a rich web browser interface, to enable online fact checking and data journalism scenarios. We believe that annotated documents will be increasingly adopted. The purpose of this work was to set database foundations for expressively and efficiently exploiting such interconnected databases of structured documents, facts and knowledge.

7.2 Ongoing and future works

View composition. The most recent part of this work covers the extended language and query-view composition problem (Sections 3.2.3, 3.2.4 and 3.3). As pointed in Section 3.4, we are currently looking at improving the composition algorithm using some pruning and query minimization techniques. To be more useful in practice, the composition algorithm will be extended to support composition w.r.t. multiple views. These questions are still under scrutiny or we will be the subject of our attention in the immediate future.

Cost-based optimization. Our experiments showed that, since the pruning algorithm are costly, it is only worthwhile in cases when a large portion of the total result can be skipped. A cost-model would help make systematic decisions on which strategies to use. One should in practice be able to get some relevant statistics from the underlying XDM and RDM themselves. Beside the conventional ingredients of a cost model, such as selectivity estimation, one could pre-compute a *summary* of annotated XML nodes (e.g., a probabilistic tree) to gauge whether pruning is likely to improve performances. Given a path in a tree pattern ending with a URI-variable annotated node, if the likelihood of that path in the probabilistic tree is low, then intuitively pruning should be amortized. In a

context of Web data, other cost components such as network bandwidth and the reliability of distant sources would be worth taking into account.

In our evaluation study, we put aside the problem of RDFS-entailments, considering the RDFS closure was materialized as part of the RDF instance. We would like to extend the work described in Chapter 6, integrate to the physical operators devised for this approach to the XR platform, and in doing so, incorporate entailed data as part of selectivity estimation. Future evaluation campaigns would include a comparison on the trade-offs between a pure forward-chaining approach, and our bitmap-index based storage approaches.

OWL 2 & XR Rules. We have defined the semantics of an RDF sub-instances in terms of RDFS-entailment rules. OWL 2 (Web Ontology Language) introduces the notion of profiles, corresponding to fragments of the Description Logics with interesting properties. Extending the semantics of XR to OWL 2 will also enable the study in XR of classic Description Logics problems, such as consistency checking.

The problem of view composition also has applications in a deductive database perspective, where XRQ views are seen as rules. We could also delve further into data integration as described in Section 2.3.1. However, in our case, mapping rules between global and locale schemata could be expressed directly in XRQ.

FactMinder 2.0 & Social XR. Scenarios like the ones described in Section 1 and Chapter 5 are inherently human tasks, and may never be entirely automated. The current XRQ language only allow unions of conjunctive queries. Adding a richer set of operations to the language, such as *negation*, *inequality predicates*, *outer joins*, *aggregate function*, would be an obvious direction to take.

As mentioned in Section 5.4, the study of social interactions in XR and FactMinder is a natural evolution of the project. Letting users of social networks annotates content on the Web is strong trend, as acknowledged by the popularity of Facebook [1] "Like" and Google+ [2] "+1" functions, or more recent online services such as Medium [3], where communities of twitter users can annotate fine-grained pieces of online blogs. Beyond crowd-sourcing and sentiment analysis, such a setting would also open the door to areas that are still actively studied in the relational literature, such as *transactions*, *privacy* or *provenance*. This will be the topic of a separate PhD thesis.

1. `facebook.com`
2. `plus.google.com`
3. `medium.com`

Bibliography

[AAC+08] 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). *Proceedings of the VLDB Endowment*, 2008. 17

[ABFS02a] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Ontology-based integration of XML web resources. In *Proceedings of the Internation Conference on the Semantic Web*, 2002. 15

[ABFS02b] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Querying xml sources using an ontology-based mediator. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 429–448. Springer Berlin Heidelberg, 2002. 15

[ABM05] Andrei Arion, Véronique Benzaken, and Ioana Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005. 22

[ACZH10] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the International World Wide Web Conference*, 2010. 13, 100

[ADMFDF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql/, 1998. 8

[AFS+01] Bernd Amann, Irini Fundulaki, Michel Scholl, Catriel Beeri, and Anne marie Vercoustre. Mapping XML Fragments to Community Web Ontologies. In *WebDB*, 2001. 15

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 11, 45

[AKKP08] Waseem Akhtar, Jacek Kopecký, Thomas Krennwallner, and Axel Polleres. XSPARQL: Traveling between the XML and RDF Worlds

- and Avoiding the XSLT Pilgrimage. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *Proceedings of the European Semantic Web Conference*, volume 5021 of *ESWC'08*, pages 432–447, Berlin, Heidelberg, 2008. Springer-Verlag. x, 16

[AM08] Loredana Afanasiev and Maarten Marx. An analysis of XQuery benchmarks. *Inf. Syst.*, 33(2):155–181, 2008. 15, 51

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the International Conference on Very Large Databases*, 2007. 13

[AMP+08] Serge Abiteboul, Ioana Manolescu, Neoklis Polyzotis, Nicoleta Preda, and Chong Sun. XML processing in DHT networks. In *Proceedings of the International Conference on Data Engineering*, pages 606–615. IEEE, 2008. 9

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 497–508, New York, NY, USA, 2001. ACM. xiii, 22, 24

[BC06] Angela Bonifati and Alfredo Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data & Knowledge Engineering*, 59(2):247–269, 2006. 9

[BCC05] Daniele Braga, Alessandro Campi, and Stefano Ceri. XQBE (XQuery By Example): A visual interface to the standard XML query language. *ACM Transactions on Database Systems*, 30:398–443, June 2005. 8

[BCH+06] K. Beyer, R. Cochrane, M. Hvizdos, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, M. Nicola, F. Ozcan, H. Pirahesh, N. Seemann, A. Singh, T. Truong, R. C. Van Der Linden, B. Vickery, C. Zhang, and G. Zhang. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006. 9

[BDB] Oracle Berkeley DB Java Edition. http://oracle.com/technetwork/database/berkeleydb/. 67

[BDK+11] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. Mapping Between RDF and XML with XSPARQL. Technical report, DERI, April 2011. x, 16, 49

[BGTC09] Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki, and Stavros Christodoulakis. Querying xml data with sparql. In *Proceedings of the 20th International Conference on Database and Expert Systems Applications*, DEXA, pages 372–381, Berlin, Heidelberg, 2009. Springer-Verlag. 16

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, SIGMOD '06, pages 479–490, New York, NY, USA, 2006. ACM. 9

[BK08] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008. 8

[BK09] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Transactions on Database Systems*, 34(4), 2009. 8

[BKO+11] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011. 88

[BKVH02] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the Internation Conference on the Semantic Web*, pages 54–68. Springer, 2002. 13

[BMSU85] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 1–15. ACM, 1985. 88

[BÖB+04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proceedings of the International Conference on Very Large Databases*, 2004. 22, 76

[CBC+12] L.J. Chen, P.A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J.F. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a wide sparse table. In *Proceedings of the International Conference on Data Engineering*, pages 630 –641, April 2012. 51

[CBK12] Luying Chen, Michael Benedikt, and Evgeny Kharlamov. QUASAR: querying annotation, structure, and reasoning. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *Proceedings of the International Conference on Extending Database Technologies*, pages 618–621. ACM, 2012. 17

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. In *Proceedings of the International World Wide Web Conference*, pages 1171–1187, New York, NY, USA, 1999. Elsevier North-Holland, Inc. 8

[CDES05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceed-*

ings of the International Conference on Very Large Databases, pages 1216–1227. VLDB Endowment, 2005. 14

[CDO08] Bogdan Cautis, Alin Deutsch, and Nicola Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008. 57

[CDP10] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. *Information Processing Letter*, 2010. 91, 96

[CHT11] Sarah Cohen, James T. Hamilton, and Fred Turner. Computational journalism. *Commun. ACM*, 54(10):66–71, October 2011. 85

[CKKC+09] Olivier Corby, Leila Kefi Khelif, Hacène Cherfi, Fabien Gandon, and Khaled Khelif. Querying the Semantic Web of Data using SPARQL, RDF and XML. Research Report RR-6847, INRIA, 2009. x, 16

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *Proceedings of the International Conference on Very Large Databases*, pages 646–648, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. 9

[CLF09] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10), 2009. 12

[CLYY11] Sarah Cohen, Chengkai Li, Jun Yang, and Cong Yu. Computational journalism: A call to arms to database researchers. In *CIDR*, pages 148–151. www.cidrdb.org, 2011. 85

[CPST03] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On labeling schemes for the semantic web. In *Proceedings of the International World Wide Web Conference*, WWW '03, pages 544–555, New York, NY, USA, 2003. ACM. 89, 99

[CRCM12] J. Camacho-Rodriguez, D. Colazzo, and I. Manolescu. Building Large XML Stores in the Amazon Cloud. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 151–158, 2012. 9

[DEFS99] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In Robert Meersman, Zahir Tari, and Scott Stevens, editors, *Database Semantics*, volume 11 of *IFIP, The International Federation for Information Processing*, pages 351–369. Springer US, 1999. 15

[DEG+03] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin, and Jason Y. Zien. Semtag and seeker: bootstrapping the semantic web via automated semantic annotation. In *Proceedings of*

the International World Wide Web Conference, WWW '03, pages 178–186, New York, NY, USA, 2003. ACM. ix, 17

[DFG+07] Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker. Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler, and Stefan Zugal. Translating XPath queries into SPARQL queries. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems - Volume Part I*, OTM'07, pages 9–10, Berlin, Heidelberg, 2007. Springer-Verlag. x, 16

[DFG+09] Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler, and Stefan 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. 16

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, SIGMOD '99, pages 431–442, New York, NY, USA, 1999. ACM. 9

[DG97] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In Alberto O. Mendelzon and Z. Meral Özsoyoglu, editors, *Proceedings of the ACM SIGMOD Conference on the Principles of Database Systems*, pages 109–116. ACM Press, 1997. 45

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, SIGMOD '11, pages 145–156, New York, NY, USA, 2011. ACM. 96

[DP10] François Deliège and Torben Bach Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the International Conference on Extending Database Technologies*, 2010. 91

[Erl] Orri Erling. Advances in Virtuoso RDF triple storage (bitmap indexing). At http://virtuoso.openlinksw.com/. 100

[ES01] Michael Erdmann and Rudi Studer. How to structure and access XML documents with ontologies. *Data & Knowledge Engineering*, 36(3):317 – 335, 2001. 15

[FBB05] Tim Furche, François Bry, and Oliver 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. x, 16

[FFG02] S. Flesca, E. Furfaro, and S. Greco. XGL: a graphical query language for XML. In *IDEAS*, pages 86 – 95, 2002. 8

[FHK+03] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Natix: A technology overview. In AkmalB. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, editors, *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*, pages 12–33. Springer Berlin Heidelberg, 2003. 9

[FKS+02] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 41(4):616–641, October 2002. 9

[FMPG+13] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 2013. 13, 100

[Fra05] Massimo Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *XSym*, 2005. 64

[FTS00] Mary Fernández, Wang-Chiew Tan, and Dan Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1-6):723–745, June 2000. 9

[GGL+08] Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller, and Christoph Reinke. Embedding SPARQL into XQuery/XSLT. In *ACM SAC*, 2008. 16

[GHMP11] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011. 97

[GHV11] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. RDFS Update: From Theory to Practice. In Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter Leenheer, and Jeff Pan, editors, *The Semanic Web: Research and Applications*, volume 6644 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2011. 88

[GKK+11a] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents. In Marco Brambilla, Fabio Casati, and Stefano Ceri, editors, *VLDS*, volume 880 of *CEUR Workshop Proceedings*, pages 30–33. CEUR-WS.org, 2011. xv, 5, 66

[GKK+11b] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing Triples

on Trees: an XML-RDF Hybrid Model for Annotated Documents. In *Bases de Données Avancées*, Rabat, Morocco, 2011. xv, 5

[GKK+12] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Des triplets sur des arbres. Un modèle hybride XML-RDF pour documents annotés. *Ingénierie des Systèmes d'Information*, 17(5):87–111, 2012. xv, 5

[GKK+13a] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Fact-checking and analysing the Web (demonstration). In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, New York, NY, USA, 2013. xvi, 5

[GKK+13b] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing triples on trees: an XML-RDF hybrid model for annotated documents. *The Very Large Databases Journal*, pages 1–25, 2013. xv, 5

[GKLM10a] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In Jimmy Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *Proceedings of the ACM Conference on Information and Knowledge Management*, pages 1947–1948. ACM, 2010. 100

[GKLM10b] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. Materialized View-Based Processing of RDF Queries. In *Bases de Données Avancées*, 2010. 100

[GKLM11a] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In *Bases de Données Avancées*, 2011. 100

[GKLM11b] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View Selection in Semantic Web Databases. *Proceedings of the VLDB Endowment*, 5(2), October 2011. 67, 71, 100, 102

[GMR13] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Efficient query answering against dynamic RDF databases. In Giovanna Guerrini and Norman W. Paton, editors, *Proceedings of the International Conference on Extending Database Technologies*, pages 299–310. ACM, 2013. 88

[Gra90] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1990. 66

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011. 13

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1989. 51

[Hid03] Jan Hidders. Satisfiability of XPath Expressions. In *DBPL*, pages 21–36, 2003. 62

[HS02] Siegfried Handschuh and Steffen Staab. Authoring and annotation of web pages in CREAM. In *Proceedings of the International World Wide Web Conference*, WWW '02, pages 462–473, New York, NY, USA, 2002. ACM. ix, 17

[HSSVdS11] 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. 17

[IKKN99] H. Ishikawa, K. Kubota, Y. Kanemasa, and Y. Noguchi. The design of a query language for XML data. In *DEXA*, pages 919–922, 1999. 8

[Kar12] Konstantinos Karanasos. *Techniques fondées sur des vues matérialisées pour la gestion efficace des données du web*. Thesis, Université Paris Sud - Paris XI, June 2012. 57, 100

[KCS11] Shahan Khatchadourian, Mariano P Consens, and Jérôme Siméon. Having a ChuQL at XML on the Cloud. In *Alberto Mendelzon International Workshop*, 2011. 9

[KK01] José Kahan and Marja-Riitta Koivunen. Annotea: an open RDF infrastructure for shared Web annotations. In *Proceedings of the International World Wide Web Conference*, pages 623–632, 2001. 17

[KKMZ12] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. ViP2P: Efficient XML management in DHT networks. In *ICWE*, 2012. 9, 10, 65

[KMK08] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *Proceedings of the Internation Conference on the Semantic Web*, volume 5318 of *Lecture Notes in Computer Science*, pages 499–516. Springer, 2008. 13, 88, 100

[KMV12] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized view selection for XQuery workloads. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2012. 102

[KZ10] Konstantinos Karanasos and Spyros Zoupanos. Viewing a world of annotations through AnnoVIP (demonstration). In *Proceedings of the International Conference on Data Engineering*, 2010. 15

[LDK+11] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on SPARQL views. In *Proceedings of the International World Wide Web Conference*, pages 655–664, New York, NY, USA, 2011. ACM. 46

[Leb12] Julien Leblay. SPARQL query answering with bitmap indexes. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *SWIM*, page 9. ACM, 2012. xvii, 5

[LLCC05] Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In *Proceedings of the International Conference on Very Large Databases*, 2005. 10

[LM09a] J.J. Levandoski and M.F. Mokbel. RDF Data-Centric Storage. In *Web Services, 2009. ICWS 2009.*, pages 911–918, 2009. 13

[LM09b] Zhen Hua Liu and Ravi Murthy. A Decade of XML Data Management: An Industrial Experience Report from Oracle. In *Proceedings of the International Conference on Data Engineering*, ICDE '09, pages 1351–1362, Washington, DC, USA, 2009. IEEE Computer Society. 9

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the International Conference on Very Large Databases*, pages 251–262. Morgan Kaufmann, 1996. 46

[MAYU05] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A Path-based Relational RDF Database. In Hugh E. Williams and Gillian Dobbie, editors, *ADC*, volume 39 of *CRPIT*, pages 95–103. Australian Computer Society, 2005. xvii, 88, 100

[Mei03] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, Lecture Notes in Computer Science. Springer, 2003. 9

[MKK08] Iris Miliaraki, Zoi Kaoudi, and Manolis Koubarakis. XML data dissemination using automata on top of structured overlay networks. In *Proceedings of the International World Wide Web Conference*, pages 865–874. ACM, 2008. 9

[MKVZ11] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery rewriting using multiple views. In *Proceedings of the International Conference on Data Engineering*, 2011. 65, 76

[MPK07] Akiyoshi Matono, Said Mirza Pahlevi, and Isao Kojima. RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores. In *Databases, Information Systems, and Peer-to-Peer Computing*, pages 323–330. Springer, 2007. 13

[MPV09] Ioana Manolescu, Yannis Papakonstantinou, and Vasilis Vassalos. XML Tuple Algebra. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3640–3646. Springer US, 2009. 8

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the International Conference on Data Engineering*, 2011. 13

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1), 2008. 13, 100

[NW09] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2009. 13

[NW10] Thomas Neumann and Gerhard Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proceedings of the VLDB Endowment*, 2010. 64

[PCS+04] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML data stored in a relational database. In *Proceedings of the International Conference on Very Large Databases*, VLDB '04, pages 1146–1157. VLDB Endowment, 2004. 9

[PH01] Rachel Pottinger and Alon Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *The Very Large Databases Journal*, 10(2-3):182–198, 2001. 46

[Pol07] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the International World Wide Web Conference*, pages 787–796. ACM, 2007. 12

[PPWW08] Reinhard Pichler, Axel Polleres, Fang Wei, and Stefan Woltran. dRDF: Entailment for Domain-Restricted RDF. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin Heidelberg, 2008. 12

[PSS02] Peter Patel-Schneider and Jérôme Siméon. The Yin/Yang web: XML syntax and RDF semantics. In *Proceedings of the International World Wide Web Conference*, WWW '02, pages 443–453, New York, NY, USA, 2002. ACM. x, 15

[RGN+01] Jonathan Robie, Lars Marius Garshol, Steve Newcomb, Michel Biezunski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassillis Christophides, and Gregorius Karvounarakis. The syntactic web. *Markup Lang.*, September 2001. x, 15

[RH05] Lawrence Reeve and Hyoil Han. Survey of semantic annotation platforms. In *ACM SAC*, 2005. ix, 17

[RMC11] Mariano Rodríguez-Muro and Diego Calvanese. Semantic Index: Scalable Query Answering without Forward Chaining or Exponential Rewritings. In *Proceedings of the Internation Conference on the Semantic Web*, 2011. xvii, 88, 93, 100

[Rys05] Michael Rys. XML and relational database management systems: inside Microsoft SQL Server. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 958–962, New York, NY, USA, 2005. ACM. 51

[SB05] Heiner Stuckenschmidt and Jeen Broekstra. Time-Space Trade-Offs in Scaling up RDF Schema Reasoning. In Mike Dean, Yuanbo Guo, Woochun Jun, Roland Kaschek, Shonali Krishnaswamy, Zhengxiang Pan, and Quan Z. Sheng, editors, *WISE Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 172–181. Springer, 2005. 88

[SGK+08] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2), 2008. 13

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Databases*, VLDB '99, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. 9

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Databases*, pages 974–985, 2002. 64, 69

[tH05] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, 2005. 12

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 204–215, New York, NY, USA, 2002. ACM. 9, 51

[UKM+10] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *Proceedings of the European Semantic Web*

Conference, volume 6088 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2010. 100

[UKOvH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable Distributed Reasoning Using MapReduce. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *Proceedings of the Internation Conference on the Semantic Web*, volume 5823 of *Lecture Notes in Computer Science*, pages 634–649. Springer, 2009. 13, 100

[UvHSB11] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *Proceedings of the Internation Conference on the Semantic Web*. Springer, 2011. 13, 88, 99, 100

[Vir12] Roberto De Virgilio. A linear algebra technique for (de)centralized processing of SPARQL queries. In Paolo Atzeni, David W. Cheung, and Sudha Ram, editors, *ER*, volume 7532 of *Lecture Notes in Computer Science*, pages 463–476. Springer, 2012. 13

[VVMD+02] Maria Vargas-Vera, Enrico Motta, John Domingue, Mattia Lanzoni, Arthur Stutt, and Fabio Ciravegna. MnM: Ontology Driven Semi-automatic and Automatic Support for Semantic Markup. In *EKAW*, 2002. ix, 17

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 2008. 13

[WSK+03] Kevin Wilkinson, Craig Sayers, Harumi Kuno, Dave Reynolds, et al. Efficient RDF storage and retrieval in jena2. In *Proceedings of SWDB*, volume 3, pages 7–8, 2003. 13

[wwwa] BaseX. http://basex.org. 9, 80

[wwwb] RDF application development for IBM data servers. http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/index.jsp?topic=14

[wwwc] DBpedia 3.7. http://wiki.dbpedia.org/Downloads37. 69

[wwwd] Online experiment site. http://tripleo.saclay.inria.fr/xr/experiments. 69, 71, 77

[wwwe] Jena semantic web framework. Available at jena.sourceforge.net/. 100

[wwwf] Microformats. http://microformats.org/. 17

[wwwg] Neo4jo. http://neo4j.org. 13

[wwwh] Open knowledge foundation. http://ofkn.org. viii, 2, 10

[wwwi] OpenCalais. http://opencalais.com/. xvi, 80

[wwwj] OWL 2 web ontology language document overview. http://www.w3.org/TR/owl2-overview/. 21

[wwwk] Virtuoso Open Source Edition 6.1.6. http://virtuoso.openlinksw.com. 14, 80

[www98] Extensible Markup Language (XML) 1.0 (first edition). http://www.w3.org/TR/1998/REC-xml-19980210, 1998. 7

[www01] URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/, 2001. 10, 20, 50

[www04a] Allegro graph. http://www.franz.com/agraph/allegrograph/, 2004. 14

[www04b] RDF concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004. vii, xi, 1, 10

[www04c] RDFa Primer. http://www.w3.org/TR/xhtml-rdfa-primer/, 2004. 17

[www04d] RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004. 11

[www05] xml:id. http://www.w3.org/TR/xml-id, 2005. 51

[www06a] RDF in HTML. http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml, 2006. 17

[www06b] Linked Data - Design Issues. http://www.w3.org/DesignIssues/LinkedData.html, 2006. vii, 2

[www08a] GRDDL. http://www.w3.org/TR/grddl/, 2008. 16

[www08b] SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008. 12, 22

[www08c] Extensible Markup Language (XML) 1.0 (fifth edition). http://www.w3.org/TR/xml/, 2008. vii, 1, 7

[www10] XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/xpath-datamodel/, 2010. vii, 8, 51

[www13a] SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/, 2013. 12, 54

[www13b] SPARQL 1.1 Entailment Regimes. http://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/, 2013. 12

[XLWB09] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. DDE: from Dewey to a fully dynamic XML labeling scheme. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2009. 10, 55

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Techn.*, 1(1):110–141, 2001. 9, 10

[Yee02] Ka-Ping Yee. CritLink: Advanced Hyperlinks Enable Public Annotation on the Web. In *Computer Supported Cooperative Work (CSCW)*, 2002. ix, 17

[ZMC+11] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, 2011. 13, 100