HAL
open science

Gestion des données efficace en pair-à-pair

Spyros Zoupanos

To cite this version:

Spyros Zoupanos. Gestion des données efficace en pair-à-pair. Databases [cs.DB]. Université Paris Sud - Paris XI, 2009. English. NNT : . tel-00695402

HAL Id: tel-00695402

https://theses.hal.science/tel-00695402

Submitted on 8 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris-Sud 11
Ecole Doctorale d'Informatique de Paris-Sud

THÈSE

Présentée en vue d'obtention du grade de

DOCTEUR EN SCIENCES

Discipline : Informatique

par
Spyros Zoupanos

Sujet de la thèse :

Efficient peer-to-peer data management

(Gestion des données efficace en pair-à-pair)

Directeurs de thèse : **Serge ABITEBOUL et Ioana MANOLESCU**

Soutenue le 9 décembre 2009 devant le jury composé de :

Rapporteurs :	M.	Bernd AMANN	LIP6 - Univ. de Pierre et Marie Curie
	M.	Vasilis VASSALOS	Athens Univ. of Economics and Business
Examinateurs :	Mme.	Geneviève JOMIER	Univ. Paris-Dauphine
	M.	Patrick VALDURIEZ	INRIA Sophia Antipolis - Méditerranée et Univ. de Nantes
Directeurs de thèse :	M.	Serge ABITEBOUL	INRIA Saclay–Île-de-France et LRI, Université de Paris Sud
	Mme.	Ioana MANOLESCU	INRIA Saclay–Île-de-France et LRI, Université de Paris Sud

Acknowledgements

There were many people that stood by me during these three years and that I would like to thank. First of all, I would like to thank my parents who were always next to me, doing their best to support me in my studies and my life in general. Their help and advices are very important. I would also like to thank my girlfriend Ana who also supported me and was next to me during all the difficult moments of this thesis.

I would like to thank my supervisor, Ioana Manolescu, for her help, guidance and support. She was always next to me whenever I needed her. She was next to me even during difficult times for her, like during her pregnancy. I believe that part of this work would not have existed if she had not been so passionate with our common work. I would like to thank Serge Abiteboul. I may have cooperated more with Ioana during this thesis but he was always there whenever I wanted his help, like in the beginning of my thesis, and he was always making me feel secure with his presence.

I would like to thank my professors in Athens and especially Yannis Ioanidis and Alexis Delis. They were the first that made me love databases and distributed system. Moreover, I would also like to thank them for their help and guidance when I was searching a PhD position.

I also appreciated my cooperation with the engineers Evaldas Taroza, Mohamed Ouazara and Alin Tilea. The last deserves many thanks for his patience and help in the development of ViP2P. I would like to mention that I enjoyed being in the same group with Nada Abdallah, Andrei Arion, Vincent Armant, Pierre Bouhris, François Calvier, Bogdan Cautis, Philippe Chatalic, Philippe Dague, Hélène Gagliardi, Alban Galland, Anca Ghitescu, François Goasdoué, Fayçal Hamdi, Konstantinos Karanasos, Asterios Katsifodimos, Yannis Katsis, Evgeny Kharlamov, Wael Khemiri, Julien Leblay, Yingmin Li, Bogdan Marinoiu, Cedric du Mouza, Yassine Mrabet, Huu-Nghia Nguyen, Nobal Niraula, Marilena Oita, Nathalie Pernelle, Radu Pop, Nicoleta Preda, Chantal Reynaud, Philippe Rigaux, Jesus Camacho Rodriguez, Laurent Romary, Brigitte Safar, Fatiha Saïs, Pierre Senellart, Laurent Simon, Cristina Sirangelo, Mohamadou Thiam, Gabriel Vassile, Michalis Vazirgiannis, Véronique Ventos, Ravi Vijay and Lina Ye. I would

also like to thank the secretaries of the group (Marie Domingues and Celine Halter) for doing their best to resolve our bureaucratic problems.

Résumé

Le développement de l'internet a conduit à une grande augmentation de l'information disponible pour les utilisateurs. Ces utilisateurs veulent exprimer leur besoins de manière simple, par l'intermédiaire des requêtes, et ils veulent que ces requêtes soient évaluées sans se soucier où les données sont placées ou comment les requêtes sont évaluées. Le travail qui est présenté dans cette thèse contribue à l'objectif de la gestion du contenu du Web de manière déclarative et efficace et il est composé de deux parties. Dans le premier partie, nous présentons OptimAX, un optimiseur pour la langage Active XML qui est capable de reécrire un document Active XML donné dans un autre document équivalent dont l'évaluation sera plus efficace. OptimAX contribue à résoudre le problème d'optimisation des requêtes distribuées dans le cadre d'Active XML et nous présentons deux études de cas. Dans le deuxième partie, nous proposons une solution au problème de l'optimisation d'un point de vue différent. Nous optimisons des requêtes en utilisant un ensemble des requêtes pré-calculées (vues matérialisées). Nous avons développé une plateforme pair-à-pair, qui s'appelle ViP2P (views in peer-to-peer) qui permet aux utilisateurs de publier des documents XML et de spécifier des vues sur ces documents en utilisant une langage de motifs d'arbres. Quand un utilisateur pose une requête, le système essaiera de trouver des vues qui peuvent être combinées pour construire une réécriture équivalente à la requête. Nous avons fait des expérimentations en utilisant des ordinateurs des différents laboratoires en France et nous avons montré que notre plateforme passe à l'échelle jusqu'à plusieurs GB de données.

Abstract

Internet has led to a fundamental increase of information that is available to its users over the latest years. The users want to express their needs by simple means, such as queries and they want their queries to be evaluated without caring where the data are placed or how the queries are optimized. The work presented in this thesis contributes to the goal of declarative and efficient management of Web content in distributed settings and it is divided into two main chapters. In the first chapter we study OptimAX, an optimizer for the Active XML language which is able to rewrite a given Active XML document to an equivalent document which would, very likely, have smaller execution cost. With OptimAX we focus on the problem of distributed query optimization in the Active XML setting and we present two interesting case studies inspired by the R&D projects in which our group has been involved. In the second chapter, we propose solutions to the optimization problem from a different perspective. We optimize queries using a set of precomputed queries (materialized views). We have developed a peer-to-peer platform, called ViP2P (views in peer-to-peer) that gives to the users the opportunity to publish their XML documents and to specify views over these documents using a tree pattern language. Whenever a user asks a query, the system will try to find views that can be combined in order to find a rewriting equivalent to the asked query. We have carried WAN experiments that show the scalability of the ViP2P platform.

Contents

1 Introduction **1**

2 OptimAX **5**
2.1 The AXML language 6
2.1.1 Documents and services 7
2.1.2 Active XML data 9
2.1.3 Extension: built-in AXML services and replication . . 11
2.2 AXML activation and optimization problems 13
2.2.1 AXML activation 13
2.2.2 AXML optimization 15
2.3 an optimizer for AXML 18
2.3.1 Optimization rules 19
2.3.2 Implementation issues 24
2.3.3 Search strategies and heuristics 29
2.4 Case studies 30
2.4.1 Distributed software development in EDOS 31
2.4.2 Warehousing Web data in WebContent 43
2.5 Experimental analysis 50
2.6 Software architecture design 54
2.6.1 Inside OptimAX 54
2.6.2 Integrating OptimAX with the AXML peer 60
2.7 Related works 62
2.8 Conclusion 63

3 ViP2P - Views in peer-to-peer **65**
3.1 Patterns 67
3.2 Algebraic rewritings using patterns 69
3.2.1 Algebra 69
3.2.2 Problem statement 70
3.2.3 Complexity 72
3.3 Rewriting-based query answering 75
3.3.1 Building a rewriting out of a set of views 75
3.3.2 Rewriting algorithms 78

3.3.3 Evaluating a rewriting 80
3.4 P2P view management 80
3.4.1 View materialization 81
3.4.2 Identifying views for rewriting 81
3.5 Performance evaluation 83
3.5.1 System implementation and configuration 83
3.5.2 Setup for view building and query processing 84
3.5.3 View building 84
3.5.4 Query evaluation 84
3.5.5 View indexing and lookup strategies 87
3.5.6 Query rewriting 89
3.5.7 Conclusions of the experiments 91
3.6 Software architecture design 91
3.7 Related works 99
3.8 Conclusion 100

4 Conclusion 101

Chapter 1

Introduction

Internet has led to a fundamental increase of information that is available to its users over the latest years. Moreover, the number of Internet users is big and has a tendency to increase even more. This mean of communication has been used by simple users such as bloggers which are willing to share their opinion with the Internet community, by readers which would like to be informed on some of their areas of interest, but also by professional users such as companies which would like to cooperate or use the Web as an area to perform their target group research. All these different classes of Web users share the need to access and propagate the right information. The blogger needs to make his opinion available to a specific target group, the reader is always searching for interesting articles from his favorite newspapers, the companies are constructing their strategy based on information collected by searching the Web. The plethora and diversity of the available information makes their task more difficult than it should be. They should be able to express their need by simple means, such as queries and they should not worry about how the information will be obtained. Moreover, they want their task to be executed fast enough but, in the same time, not to worry about optimization issues. The work presented in this thesis contributes to this goal: declarative, efficient management of Web content in distributed settings.

One way of combining information from different sources and performing computations on them is given by Active XML [23]. Active XML (AXML, for short), is a declarative framework that harnesses Web services for distributed data management. The architecture used in AXML is peer-to-peer, an architecture widely used on the Web. The model is based on AXML documents. These are XML documents that may contain embedded calls to Web services and to AXML web services. The latter are Web services capable of exchanging AXML documents.

An AXML peer is a repository of AXML documents, acting both as a client by invoking the embedded service calls, and as a server by providing

AXML services. The latter services are generally defined as queries or updates over the persistent AXML documents. The approach gracefully combines stored information with data defined in an intensional manner, as well as dynamic information. This framework can be used to satisfy the needs of the users that were mentioned before. For example, a user who is interested in finding the news that his friend has chosen from a specific newspaper, needs to contact his friend's AXML peer to get the newspaper's name and article ids, and then join these article ids with the ids of the articles that the newspaper web service will provide. This kind of computations are very common in the AXML context and are easily expressed withing an AXML document.

Needless to say, such computations may become very expensive because of various parameters that can affect a document's evaluation. Such parameters are the complexity of queries and the possible use of indices, the computational power of the peers and their bandwidth, the proximity of the peers and others. In this context, an optimizer has been developed which, given an AXML document, statistic information and a proposed strategy, is going to rewrite the given document into an equivalent document that will have, very likely, lower execution cost. Chapter 2 presents the AXML language and the extensions that we bring to it to make it more amenable to optimization. We formalize the AXML optimization problem, and we present OptimAX, the optimizer which we have built to solve it. We furthermore present two interesting case studies, inspired from R&D projects in which the Gemo/IASI group has been involved. The first case study (EDOS) shows how OptimAX can be used in an unstructured peer-to-peer network, in a distributed open-source software development architecture. The second case study (WebContent) shows how OptimAX is used in an already deployed data management project. Last but not least is the presentation of experimental results.

Apart from the optimization problem in the AXML's context, we are interested in query optimization using precomputed queries (materialized views). We have developed a peer-to-peer platform, called ViP2P (views in peer-to-peer), that gives to the users the opportunity to publish their (XML) documents and to specify views over these documents using a tree pattern language. Views can be queries frequently asked by the users, or more generally, specify a data need that the user has and which must be met based on the document present in the network, past, present and future. ViP2P peers cooperate in an transparent way to the user, to fill in the views with the needed data. Whenever a user asks a query, the system will silently try to find new views that can be combined in order to find a rewriting equivalent to the asked query. The goal of the platform is to provide fast answers to the asked queries by using the already precomputed results of materialized views. It should be noted that ViP2P is very useful in cases where the users want to create a large repository of XML documents. ViP2P

was also inspired by the WebContent project mentioned above. WebContent focuses on giving to the participating companies efficient tools for gathering, enriching and exploiting structured documents from the Web or produced by the companies. ViP2P can give the opportunity to the participating peers to access a structured content warehouse where they can efficiently query documents used for market analysis or Web intelligence gathering. Chapter 3 presents our platform, shows our solution to the query rewriting problem, presents algorithms for finding the needed views for the query rewriting and for view materialization, shows how our platform scales and compares ViP2P with related work.

In Chapter 4, we conclude by presenting remaining open issues that need our attention and need further work.

Chapter 2

OptimAX - An Active XML optimization framework

This Chapter considers the problem of *efficient execution of distributed Web services*. Our solution is based on a composition language, namely ActiveXML (or AXML in short) [23], which in our setting can be seen as equivalent to a subset of BPEL. An ActiveXML document is an XML document specifying which services to call, how to build their input messages, and how the calls should be ordered. The contribution of this Chapter is an *AXML optimizer* called OptimAX, which given an AXML document, applies *equivalence-preserving rewriting* that transforms it into a different document, producing the same results, but possibly very different in shape and in the set of services it invokes. Thus, the execution of the rewritten document is likely to both be faster and consume less CPU resources than that of the original document.

Following the service-oriented architecture of AXML platform, we have implemented OptimAX as a Web service which, when invoked with an AXML document, returns a rewritten document. This step allows to benefit from the kind of performance-enhancing techniques typically applied in distributed databases [55], but in a new setting: loosely coupled (vs. tightly controlled servers), generic (vs. tailored to specific indices and execution techniques), extensible to any service (vs. limited to the "inside" of the database server box). Another important difference is that AXML (and OptimAX) support continuous (streaming) services, such as e.g. RSS feeds. XML streams are at the core of many modern Web applications, e.g. for keeping a portal's content up to date, or for implementing continuous business interactions in a workflow-style setting.

OptimAX has first been the focus of a demonstration in the French national database conference [8] (informal proceedings) and then in the ICDE conference [9]. The full approach been published in [11] and also presented at [10]. A specific application of OptimAX in the context of the WebContent

R&D project was demonstrated at the VLDB 2008 conference [1], and will be discussed in detail in a dedicated Section of this Chapter.

This Chapter is organized as follows. Section 2.1 describes the AXML language and the extensions we bring to it to enable optimization. Section 2.2 formalizes the AXML optimization problem, and Section 2.3 describes our optimizer. Section 2.4 presents two case studies where OptimAX is used. Section 2.5 presents experimental results. Section 2.6 analyzes implementation design decisions. In Section 2.7, based on our problem analysis (Section 2.2), we classify and compare this work with previous related AXML works and with the state of the art. Finally in Section 2.8 we conclude.

Writing conventions Throughout this thesis we will use a set of typographic conventions which we list here:

- XML documents are depicted in sans serif font, e.g. ⟨document⟩ content ⟨/document⟩
- XML queries expressed in XPath or XQuery are shown in typewriter font, e.g. `/document//paragraph`

2.1 The AXML language

To introduce the AXML language, we use the following alphabets: a set $\mathcal{P}$ of *peer names*, a set $\mathcal{D}$ of *document names*, a set $\mathcal{S}$ of *service names*, a set $\mathcal{N}$ of *node identifiers* (for the XML tree nodes), and a set $\mathcal{L}$ of *labels* (for the XML tree tags). All peer names are distinct, thus they also serve as peer identifiers (or IDs in short). Document and service names are unique inside each peer, and they also serve as document/service *address*. The triple (peer name, document name, node identifier), suffices to identify a node, therefore we term it *node ID*, or node address. We may omit the document or peer when it is obvious from the context. Elements in the sets $\mathcal{P}, \mathcal{D}, \mathcal{S}, \mathcal{N}, \mathcal{L}$ are respectively denoted p, d, s, n and l, possibly with adornments such as subscripts or primes. Trees are denoted by the letter t, possibly with adornments. For sets, we use capital letters. By convention, we prefix node IDs with $\sharp$.

Intuitively, a peer represents a context of computation; we make no assumption about how the peers are logically connected, i.e. whether the peer network is structured or not.

Section 2.1.1 makes an introduction to XML documents and services. Section 2.1.2 makes an introduction to AXML and to activation order of service calls. Finally Section 2.1.3 presents our extensions to the AXML language.

2.1.1 Documents and services

We view an XML tree as a pair of $E \subseteq \mathcal{N} \times \mathcal{N}$, and a labeling function λ from the nodes in E to $\mathcal{L}$. Using the standard XML syntax, a sample XML tree t is:

```
⟨person id="#5"⟩
        ⟨email⟩jdoe@ms.com⟨/email⟩
        ⟨first⟩john⟨/first⟩
        ⟨last⟩doe⟨/last⟩
⟨/person⟩
```

In this example, the node identifier #5 is depicted as an attribute. An XML document is a pair (t, d) where t is an XML tree and $d \in \mathcal{D}$; we may refer to it by d. A given document d, resp. service s, on a peer p is denoted $d@p$, resp. $s@p$.

Deterministic services We consider *deterministic* services, returning the same answer when invoked with the same parameters. In a Web environment, we may allow "relatively slow" variations in call results, and consider the answers at time t and $t + \epsilon$, for some small ϵ, to be equally acceptable. In its simplest form, a service can be seen as a function with XML inputs and outputs, in the style of the request-response operation [56].

Continuous services Within the class of deterministic services, without loss of generality, we consider *continuous services* that work on streams of trees and start processing their input incrementally, *before* it has been fully received. A particular class of continuous service have no inputs and emit a stream of XML trees. This corresponds to an XML subscription, in the style of RSS. Observe that a non-continuous service can be seen as a particular case of continuous service, whose answer stream always includes only one tree.

Let s be a service with n inputs. When the service is running, it expects to receive a stream of XML trees for each input. Any stream finishes with a special token denoted *eof*, that no tree may follow. Trees can arrive in all inputs in parallel. When a tree is received in one input, the service may perform an internal computation and/or may output zero or more trees.

Distributive services Among deterministic, continuous query services, of particular interest to us are *distributive* services, characterized as follows. Assume the service has n XML input streams. The service is said distributive if, for each $1 \leq i \leq n$, and for any finite streams $T_1, \ldots, T_i', T_i'', \ldots, T_n$, the following holds:

$$s(T_1, \ldots, (T_i' + T_i''), \ldots, T_n) = s(T_1, \ldots, T_i', \ldots, T_n) + s(T_1, \ldots, T_i'', \ldots, T_n)$$

where + stands for stream concatenation.

An important class of distributive services comprises those defined by *parameterized* XPath queries of the form `$in//PE` where `$in` is a variable to be dynamically bound at evaluation time, to a list of nodes (typically designated as *the dynamic context of the query*). A very similar class of distributive queries consists of XQuery queries having some variables in the outermost `for` clause iterate over such a context list.

Example Consider a query service qs_1 defined by the following query q_1:

```
for $x in $in1, $y in $in2
where $x/b=$y/b
return ⟨z⟩{x/a}⟨/z⟩
```

The query is distributive over both its inputs, materialized by the lists `$in1` and `$in2`. A possible sequence of inputs and outputs for this service is rendered in the following table. The values in the leftmost column represent discrete time moments; the other columns depict possible inputs and outputs of the service, occurring at the respective moments in time.

time	$in1	$in2	result
1	⟨x⟩⟨a⟩0⟨/a⟩⟨b⟩1⟨/b⟩⟨/x⟩	⟨y⟩⟨b⟩0⟨/b⟩⟨/y⟩	
2		⟨y⟩⟨b⟩1⟨/b⟩⟨/y⟩	⟨z⟩⟨a⟩0⟨/a⟩⟨/z⟩
3		⟨y⟩⟨b⟩2⟨/b⟩⟨/y⟩	
4	⟨x⟩⟨a⟩3⟨/a⟩⟨b⟩0⟨/b⟩⟨/x⟩		⟨z⟩⟨a⟩3⟨/a⟩⟨/z⟩

The service qs_1 defined by the query q_1 is a distributive service, with two parameters (or inputs). When the service is invoked with two actual XML streams xs_1 and xs_2 as inputs, the query will be evaluated by binding successively `$in1` to each tree in the stream xs_1, and `$in2` to each tree in the stream xs_2.

Generic query service To simplify the usage of services defined by declarative, continuous, distributive queries, we consider a *generic query service*, denoted *gqs*, and defined as follows. Its first parameter is a string, more precisely, a query expressed in distributive XPath or XQuery. Let n be the number of unbound variables (thus, input streams) in this query. The generic query service accepts n more parameters, which are treated as XML streams, and evaluates the query specified by the first parameter, over the remaining parameters.

Example Let s be the string containing the query q_1 of the previous example. As before, let xs_1 and xs_2 be two XML streams. A call to the sevice $gqs(s, xs_1, xs_2)$ has exactly the same effects as the call to $qs_1(xs_1, xs_2)$, where qs_1 is the query service of the previous example, implemented by the specific query q_1.

2.1.2 Active XML data

An *AXML document* is an XML document where some nodes labeled with the label *sc* (standing for *service call*) are given particular semantics. Specifically, an *sc* node has:

- Two children, labeled *peer* and *service*, specify a peer name $p_1 \in \mathcal{P}$ and a service name $s_1 \in \mathcal{S}$, where $s_1@p_1$ identifies an existing Web service.
- A set of children labeled *param* specify the parameters.

Let $d_0@p_0$ be an AXML document containing a service call to a service $s_1@p_1$ as above. When the call is *activated*, the following sequence of steps takes place:

1. p_0 sends a copy of the *param*-labeled children of the *sc* node, to peer p_1, asking it to evaluate s_1 on these parameters;
2. p_1 evaluates s_1 on this input;
3. a copy of the result is inserted as a sibling of the *sc* node.

When a continuous service call is activated, step 1 above takes place just once, while steps 2 and 3, together, occur repeatedly starting from that moment. The response trees successively sent by p_1 accumulate as siblings of the *sc* node.

Observe that *sc* nodes may appear as children of other *sc* nodes. Moreover, the results of an activated service call may contain other service calls.

AXML supports several mechanisms for deciding when to activate a service call. One may *explicitly* request each call activation. For instance, consider a service call $sc_2(sc_1)$, i.e. sc_1 is a parameter of sc_2. The user may choose to activate just sc_2, in this case the sc_1 element *as such* is used as a parameter for sc_2. The call sc_1 may be activated in the future.

Another more frequent case occurs when users want to activate *all the necessary service calls* to bring the document to a certain state. For instance, before sending a document d to a partner that does not understand AXML, we need to activate all the calls in d, the calls which may be received in their results, and so on. Or, it may be necessary to bring d to a given (A)XML type by selectively activating some calls only; algorithms to find these calls are given in [14].

In this work, we define a simple, yet flexible approach for deciding when to activate calls. This approach is based on a set of *default activation order*

(dao_1)	A call sc_1 which is a parameter of sc_2 is activated before sc_2.
(eao_1)	A call sc_1 having an afterActivated attribute whose value is the ID of another call sc_2 is activated after sc_2 has been activated.
(eao_2)	A call sc_1 having an afterTerminated attribute whose value is the ID of a call sc_2 is activated after sc_2 has terminated its execution.
(dao_2)	A call to the service $send@p$ is activated before all the service calls comprised in its parameters have been activated.
(dao_3)	A call to the service $receive@p$ is activated when the first message from the corresponding $send@p'$ call reaches p.
(noa_1)	Let sc be a call to $send@p$, in some document $d@p$. After activating sc, the descendant calls of sc are never activated (at p).
(dao_4)	A call to *newnode* is activated before all its descendant calls.
(noa_2)	After a call to *newnode* has been activated, its descendant calls are never activated at the original peer.

Table 2.1: Activation order rules.

constraints, which apply by default, and on some *explicit activation order* constraints, which can be manipulated by the user.

The first default activation order rule is dao_1 in Table 2.1. The reason for this rule is that a majority of the services available today require plain XML inputs and return plain XML outputs. Activating the inner call first is more likely to lead to call sc_2 with XML input. Rule dao_1 cannot influence the activation order of two calls when none is an ancestor of the other. To capture such constraints, we enable users to specify that a given service call should be activated only after another call's activation and more precisely, after receiving the first answer of that service. Moreover, for continuous services, we may wish to distinguish between activating service call sc_1 after sc_2 has been *activated* (but has not finished executing), and activating sc_1 after sc_2 has been activated and has finished, i.e. it has sent its eof. Syntactically, such constraints are expressed using two attributes afterActivated and afterTerminated, whose interpretation is provided by the rules eao_1 and eao_2 in Table 2.1.

Sample activation order Figure 2.1 depicts a simple AXML document at peer p_0, and Figure 2.2 shows a possible timeline of the activations of its calls. The period between the activation and termination of each service is shown by a horizontal bar.

```
⟨doc⟩
    ⟨sc service="f@p1" id="#1" afterTerminated="#4"⟩
        ⟨par⟩⟨sc service="h@p2" id="#2"/⟩⟨/par⟩
        ⟨par⟩⟨sc service="k@p3" id="#3"/⟩⟨/par⟩
    ⟨/sc⟩
    ⟨sc service="g@p4" id="#4"⟩
        ⟨par⟩⟨sc service="j@p2" id="#5" afterActivated="#4"/⟩⟨/par⟩
    ⟨/sc⟩
⟨/doc⟩
```

Figure 2.1: Sample AXML document.

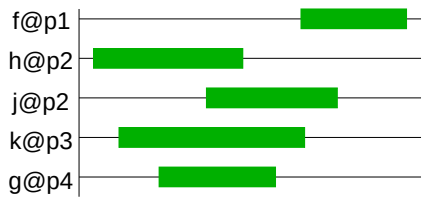


Figure 2.2: Timeline for service call activation.

2.1.3 Extension: built-in AXML services and replication

To the basic AXML model above, we add a small set of predefined services, which we assume available on all peers.

Send and receive are two services used to send (streams of) XML data from one place to another. The *send* service has two parameters. The *what* parameter represents the data to be sent from one site to another. This may be plain XML, some service calls or references to service calls. The *where* parameter is a node ID. The *receive* service has one $from$ parameter which is a node ID. The following integrity constraint applies: for each call to a *send* service, there is exactly one call to a *receive* service, such that the value of the *where* child of the *send* call is the ID of the *receive* call, and the value of the $from$ child of the *receive* call is the ID of the *send* call.

One of the main applications of *send* and *receive* concerns the sending of data streams, as Figure 2.3 illustrates. Consider for now only the XML content shown in upright part of the table. The document $doc_1@p1$ contains a call to the local service $send@p_1$, with a call to $f@p_3$ as parameter. The destination address is the node identified by $(p_2, doc_2, \#3)$, which is the call to *receive*. Once the call to $f@p_3$ is activated, it returns $fres$ elements shown in italic font in $doc_1.xml$ in Figure 2.3; activating the calls to *send*

doc1 @p1	⟨doc⟩ ⟨sc service="**send@p1**" id="**1**"⟩ ⟨what⟩ ⟨sc service="f@p3" id="2"/⟩ ⟨*fres*⟩*1*⟨/*fres*⟩ ⟨*fres*⟩*2*⟨/*fres*⟩ ⟨/what⟩ ⟨where⟩**p2.doc2.#3**⟨/where⟩ ⟨/sc⟩ ⟨/doc⟩
doc2 @p2	⟨doc⟩ ⟨sc service="**receive@p2**" id="**3**"⟩ ⟨from⟩**p1.doc1.#1**⟨/from⟩ ⟨/sc⟩ ⟨*fres*⟩*1*⟨/*fres*⟩ ⟨/doc⟩

Figure 2.3: Sample activation of calls to *send* and *receive*.

and *receive* transmits these elements into the document $doc_2@p_2$. In the Figure, the last element has not yet arrived in $doc_2.xml$.

A call to $send@p$ or $receive@p$ can only be activated when the call is in a document at peer p. This is a syntactic simplification only; we will show that it is possible for a peer p to trigger the sending of some data from another peer p'.

The introduction of the *send* and *receive* services requires new activation order rules, namely dao_2, dao_3 and noa_1 in Table 2.1. Rule dao_2 specifies that by default, *send* distributes the computation (not its result). Rule dao_3 shows that *receive* calls are not activated individually but only as a consequence of receiving a message. Finally, the *no* activation rule noa_1 states that if a service call *sc* is sent from p to p' by a *send*, *sc* is not be evaluated at p.

The newnode service installs new AXML trees on a peer. It has a single *what* parameter, which is an AXML tree. Activating the call to $newnode@p(t)$ creates a new document at peer p, whose associated data tree is t. The service returns the identifier of the new document's root. Observe that *newnode* is quite powerful, since it enables the distribution of data and computations among peers.

The activation order rules dao_4 and noa_2 apply to calls to *newnode*. Rule dao_4 favors distribution, i.e. it causes AXML code to be sent before being activated. The no-activation rule noa_2 is similar to noa_1.

Activation order, putting it all together: Together, rules dao_1, dao_2, dao_3 and dao_4 provide the default evaluation order for service calls appearing in AXML documents. These rules cannot cause cyclic dependencies. Explicit order constraints (eao_1, eao_2) override the default rules, and may introduce cycles. Documents with cyclic constraints are invalid and we do not consider them further.

Activation schedule Let d be a document and $sc_1, sc_2, \ldots, sc_k$ be the service calls from d. An activation schedule (or schedule, in short) for S is a list of pairs $[(sc_1, \tau_1), (sc_2, \tau_2), \ldots, (sc_k, \tau_k)]$ such that for $1 \leq i \leq k$, τ_i is a moment in time, sc_1 is activated at the moment τ_1, sc_2 is activated at the moment τ_2 etc. The schedule is said *valid* iff it respects: (i) all the eao_1 and eao_2 constraints of d; and (ii) as many of the $dao_1 - dao_4$ constraints as possible without violating the (eao_1) and (eao_2) constraints.

Observe that valid schedules largely allow parallel activation of continuous services (the only limitation being the explicit use of afterTerminated). The activation order example of subsection 2.1.2 illustrates a valid schedule. We focus on valid ones from now on.

Replication We assume that some AXML documents may be *replicas* (or copies) of each other, and similarly services may be replicated. A most important example for us is a *query service*. Given the string of the query, any peer equipped with a query processor can provide this query as a service, and all such services are equivalent. Observe that different copies of the same document may evolve independently with time, however, they will eventually reach the same state.

Formally, we consider an abstract peer, called *any*, and use $d@any$ to refer to any of the replicas of d, and similarly for services. We assume that each peer is able to identify one of the concrete resources corresponding to $d@any$ or $s@any$. For instance, an approach based on semantic service matching is provided in [25].

2.2 AXML activation and optimization problems

Having introduced AXML, we now chart several interesting problems which arise in this setting and we show how they relate to each other in Section 2.2.1. We pinpoint the specific optimization problem addressed in this thesis in Section 2.2.2.

2.2.1 AXML activation

Given an AXML document d, we denote by $SC(d)$ the set of service calls in the document. Observe that $SC(d)$ may grow with time, as results

(including *sc* elements) are added to the document. In principle, $SC(d)$ may grow to be infinite, e.g. consider a call to a service $f@p$ that returns exactly one call to $f@p$. We consider the practical setting when the size of $SC(d)$ is bounded.

Cost of an activation Let $d@p_0$ be an AXML document and $sc \in SC(d)$ be a call to $f@p$. The cost of activating sc is defined as:

$$c(sc) = \alpha \times c_f + \beta \times (s_p/bw_{p_0 \to p} + s_f/bw_{p \to p_0})$$

where: α and β are some numerical weights; c_f is the cost associated to the computation of f at the peer p; $bw_{p \to p_0}$ and $bw_{p_0 \to p}$, respectively, are the bandwidths from p to p_0, respectively from p_0 to p; s_p is the size of the parameters of the calls to $f@p$; s_f is the size of the results produced by the calls to $f@p$. Observe that this cost model is concentrated on total work and not in response time.

We focus on activations with a finite cost, which requires that c_f, s_p and s_f be finite; the latter implies that f returns a finite number of answers. (A simple extensions to infinite streams would consider the cost per tree in the stream.) If p is *any*, then $c(sc)$ is set to an upper bound constant max.

We define the *cost of an activation schedule* as the sum of the activation cost of all the calls in the schedule. While a schedule describes very precisely a given AXML computation, we would like to consider activation costs independently of the particular moment when each call is activated. To that effect, we introduce the following definition.

Equivalent schedules Let d be an AXML document and T_1, T_2 be two schedules over two sets of services $S_1, S_2 \subseteq SC(d)$. We say the schedules are equivalent, denoted $T_1 \equiv T_2$, iff applying T_1, resp. T_2 on the document leads to documents that are equal.

Note that S_1 and S_2 may or may not coincide, as shown in the following example.

Empty-result call Let T_1 a schedule over $S_1 \subseteq SC(d)$. Assume that for some $sc \in S_1$, it is known that activating sc does not bring results other than eof. Let $S_2 = S_1 \setminus \{sc\}$ and T_2 the restriction of T_1 to S_2, then T_1 and T_2 are equivalent (modulo eof, which we ignore by a mild abuse of terminology).

Valid schedule equivalence Let d be an AXML document and $S \subseteq SC(d)$ be a set of service calls from d. All valid schedules for S are equivalent and have the same cost.

Intuitively, valid schedules are equivalent due to the distributive, deterministic services which, called with the same parameters, produce the same results, even if some streams are created at different moments and progress at different rates in different schedules. They have the same cost because our cost model focuses on the total work, which does not change with time.

Set activation Let d be an AXML document and $S \subseteq SC(d)$. We term set activation of S on d the execution of any valid schedule for S. The cost of the activation is the cost of any valid schedule for S.

We term *one-stage activation* of d the set activation of all calls in $SC(d)$. If a call in $SC(d)$ returns another call sc', the latter is not activated in this stage.

We now consider the process of activating all calls in an AXML document until the document reaches a stationary state. Under the assumptions made here (the number of service calls in d is bounded, and services return finite streams), the fixed point state is finite, which entails that after a while, no new calls are returned by running service calls.

Full schedule Let d be an AXML document and SC_0 be the initial set of service calls in d. Let SC_1 be the service calls returned by the set activation of SC_0, and similarly, for $i = 2, \ldots, k$, let SC_{i+1} be the set of service calls returned by the set activation of SC_i. (We chose k so that $SC_k \neq \emptyset$ and $SC_{k+1} = \emptyset$). A full schedule for d is a schedule for all the calls in $\bigcup_{0 \leq i \leq k} SC_i$, such that:

- The restriction of the schedule to SC_i, for any $0 \leq i \leq k$, is valid.
- Whenever a call sc_i returned a call sc_j, sc_i appears before sc_j in the schedule.

Observe that in a full schedule, calls need not appear in the order in which they appeared in d: a call from SC_0 may be activated after a call from SC_5.

As before, all full schedules of d are equivalent and have the same cost. We define the *full activation* of d as the execution of any full schedule of d. If all services return plain XML data, full and one-stage activation coincide.

In practice, in one-stage *full activation*, our engine makes a best effort attempt to trigger activations as soon as the schedule permits. This is likely to favor parallelism but only applies after the total work oriented optimization.

2.2.2 AXML optimization

We now consider the problem of *AXML optimization*, focusing first on a restricted version of the problem, which we call one-stage optimization.

One-stage optimization Let d be an AXML document and $S \subseteq SC(d)$ a subset of the calls in d. The process of one-stage optimization for d consist of finding a document d' such that:

- one-stage activation of d and d' produce identical documents (up to terminated service calls and their subtrees);
- the cost of the set activation of d' is smaller than, or equal to that of d.

Observe that optimization, as defined above, is a static process, which does not involve call activations. We say that optimization is *exhaustive* if it produces a document d' with the minimum cost among all documents equivalent to d.

Let us now consider the integration of optimization in a full evaluation process, where we have to activate the calls in d, then the possible calls in their results etc. The choice of when and how often to invoke the optimizer impacts the rewritings it may find, thus the full activation cost. The main reason is that the optimizer decides to rewrite the document based on the service calls it contains *at optimization time*, and the latter change as activation proceeds. To characterize the goal of optimization, we define:

Document equivalence Let $d@p$, $d'@p$ be two documents at the same peer. We say d and d' are equivalent, denoted $d \equiv d'$, if the result of full activation of d and d' coincide. This means that after full activation, the documents should be identical apart from terminated service calls and their subtrees. We do not compare these nodes during the equivalence test because:

- the test focuses on the produced data after a full activation and terminated service calls can not produce any more data;
- the children of the service calls are their parameters. Two seemingly different service calls with different parameters may produce the same results.

This notion of equivalence characterizes documents that are *eventually* equal after their full activation. The documents may go through different states during the process, may call services from different peers etc. From the perspective of the user requiring the full activation result of $d@p$, the result of $d'@p$ is the same. From the system perspective, given a document d, optimization can be seen as repeatedly replacing d with d', where $d \equiv d'$, and finally retaining, from the total set of explored documents, the one having the minimum cost.

Full optimization Let d be an AXML document and $\mathcal{R}$ a set of rules. The full optimization problem for d consists of finding a sequence of steps chosen among:

- pick a service call currently in d which can be activated according to the ordering constraints of d, and activate it

– replace d with another document d' such as $d \equiv d'$

until all services calls in d have been activated, such that the total activation cost (including the past and possibly future service call activations) plus the total cost of optimization is the smallest among all possible such sequences of steps. Without loss of generality, we can consider that each document replacement step mentioned above can be performed in a time interval bounded by some constant value c_o.

Observe that the full optimization problem is more complex than just inserting optimization steps on some places into a given full schedule, because optimization may remove or add service calls, leading to re-scheduling.

Two cases can be distinguished:

1. If all service calls return plain XML results, then invoking the optimizer only once, prior to any activation, solves the full optimization problem.
2. If service calls are allowed to return any AXML trees with service calls, the problem is undecidable. The intuition for this is the following. Optimizing too rarely may lead to poor activation decisions, which could have been avoided if we had chosen to invoke the optimizer more often. Optimizing too often, on the other hand, may also be suboptimal. For instance, the optimizer may rewrite a subtree t of d into t', instead of waiting for some more activations which may have produced, say, a subtree t'' of d, such that considering t and t'' together enables a big cost-saving rewriting, which cannot be applied based on t' and t''. Thus, one can exhibit a document when optimizing *before each call activation*, even assuming $c_o = 0$, is suboptimal. Moreover, in reality, $c_o \neq 0$, thus very frequent optimization is impractical.

At this point, we consider concrete means of moving from one document d to an equivalent document d' as mentioned above. Our practical solution for moving from one document to another is to use a specific set of already know rules.

Template A template *templ* is an XML tree pattern. Its nodes share the same label set $\mathcal{L}$ with the Active XML documents, are connected with parent-child edges and form a tree-like structure. Some of the nodes are characterised as *variables* and there can be preconditions on this variables that should be satisfied.

Template matching Given a template *templ* and a document d, we call *template matching* a mapping $h : templ \rightarrow d$ from the nodes of *templ* to the nodes of d such that:

– h preserves node labels: $\forall u$, u and $h(u)$ have the same label or u is a variable and its preconditions are satisfied;

- h preserves structural relationships: $\forall u : u$ father of v, then $h(u)$ is father of $h(v)$ and $\forall u : u$ ancestor of v, $h(u)$ is ancestor of $h(v)$.

Document transformation Let:

- $templ_1$, $templ_2$ be two templates;
- $cond_1$ be the precondition set on the variables of $templ_1$ and that these preconditions are satisfied;
- a relation 1-to-many that maps each of the variables of $templ_1$ to the variables of $templ_2$;
- d a document and h a mapping such that $templ \rightarrow d$;
- r be the root of $templ_1$.

We call document transformation of d to d' the following actions:

- deletion of the subtree rooted by $h(r)$ from d;
- addition of the $templ_2$ as child of $h(r)$;
- copy of the variable contents from $templ_1$ to $templ_2$ based on the 1-to-many variable mapping.

Rule-based optimization A rule R is a tuple of the form $(a_R, cond_R, a'_R)$ such that:

- a_R, a'_R are XML tree templates, as defined above;
- $cond_R$ is a precondition to be satisfied by the a_R subtree and further preconditions which are rule specific.

Let d be an XML document, n be a node in d and R be a rule. Assuming that n matches a, and that precondition $cond_R(n)$ holds, applying R to d yields a document d' in which the subtree rooted in n has been replaced by the subtree obtained by applying the a' template on n.

In the sequel we focus on *one-stage optimization* whose details will be described in the following Sections. In the full evaluation setting, we recommend invoking the optimizer once on the initial set of services SC_0, then on SC_1, then on SC_2 and so forth, a heuristic which we find a reasonable compromise.

2.3 OptimAX: a parametric rule-based optimizer for AXML

A question that rises is how, from a set of rules, we lead to the complete search space. The solution is not as simple as applying the right set of rules at the right order but also avoiding problems that may occur during the search space exploration. In this Section we present the optimization rules that are used by our optimizer in Section 2.3.1. We then describe the various problems that we encountered while implementing the optimizer and how they were addressed in Section 2.3.2. In Section 2.3.3 we discuss possible search strategies and heuristics to be used during optimization.

Clearly, many equivalence-preserving rules can be considered and based on any given set of rules R, multiple optimization avenues are open. In the

```
⟨a⟩
  ⟨sc service="f@p1" ⟩α⟨/sc⟩
⟨/a⟩
⇓
⟨a⟩
  ⟨sc service="receive@p0" id="1"⟩
    ⟨from⟩#2⟨/from⟩⟨/sc⟩
  ⟨/sc⟩
  ⟨sc service="newnode@p2"⟩
    ⟨sc service="send@p2" id="2"⟩
      ⟨what⟩
        ⟨sc service="f@p1"⟩α⟨/sc⟩
      ⟨/what⟩
      ⟨where⟩#1⟨/where⟩
    ⟨/sc⟩
  ⟨/sc⟩
⟨/a⟩
```

Figure 2.4: Delegation rule.

sequel of this Section, we will present a set of general and helpful rules, including some which we used in applications encountered in two R&D projects to which we participated. The complete search space is the set of distinct documents obtained by repeatedly applying a set of rewriting rules which we describe next. They all preserve AXML equivalence as defined in Section 2.2.2.

Search space for a given rule set Given a set of rules R and a document d, the search space determined by R is the set of all documents d' obtained by applying on d a finite sequence of rewriting rules from R.

In the sequel, we will consider the topic of building an optimizer capable of exploring the complete search space corresponding to any given rule set R, as well as some heuristics adapted to a specific set of rules, which enable obtaining good cost reductions without requiring the exploration of the full search space.

2.3.1 Optimization rules

Delegation The rule is depicted in Figure 2.4. In this Figure, and in the following similar ones, the upper box shows a template XML document before applying the rule, while the lower box represents the same document on which the rule has been applied.

```
⟨a⟩
  α
  ⟨b id="1"⟩β⟨/b⟩
  ⟨c id="2"⟩γ⟨/c⟩
  δ
⟨/a⟩
```

⇓

```
⟨a⟩
  α
  ⟨b id="1"⟩β⟨/b⟩
  ⟨sc service="send@p0" id="3"⟩
    ⟨what⟩#1⟨/what⟩
    ⟨where⟩#4⟨/where⟩
  ⟨/sc⟩
  ⟨c id="2"⟩
    ⟨sc service="receive@p0" id="4"⟩
      ⟨from⟩#3⟨/from⟩
    ⟨/sc⟩
  ⟨/c⟩
  δ
⟨/a⟩
```

Figure 2.5: Factorization rule.

The delegation rule distributes computations across sites. The rule introduces three new service calls to *send*, *receive* and *newnode*. The effect is to install at p_2 an AXML document (via *newnode*), such that the call to $f@p_1$ will be performed from that document, i.e. from p_2, not from p_0. As soon as f results start accumulating at p_2, the *send* call will transmit them to the *receive* call at the original peer p_0, bringing thus the results in the original document.

The delegation rule may reduce costs by cutting down data transfers. For instance, assume that $p_1 = p_2$ and the call to f had as parameter a call to $g@p_1$. In this case, activating the upper document would transit the results of $g@p_1$ from p_1 to p_0 and then back from p_0 to p_1. The lower plan eliminates these needless transfers.

Factorization The *factorization* rule is depicted in Figure 2.5. In this Figure, β and γ are two sets of AXML trees, such that each tree in β is equivalent (as defined in Section 2.2.2) to some tree in γ and vice-versa. This rule eliminates redundant computations. Detecting the equivalence of two AXML trees is, in the general, a hard problem. We adopt a simple, conservative estimation of this test, which will be described in Section 2.3.2. Our method is

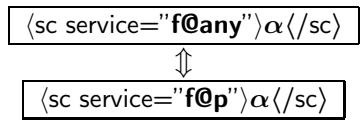


Figure 2.6: Instantiation/Generalization rule.

conservative, in the sense that if two forests β and γ are recognized as equivalent, then this is indeed the case. However, our method is not complete, in the sense that it may fail to detect equivalence in some cases.

The factorization rule replaces γ with a pair of calls to *send* and *receive*, which copy β as children of c, effectively in replacement of γ. If γ contained service calls, the rewritten document reduces the actual activated calls and (if the services were on remote peers) also reduces inter-peer transfers. The rewritten document requires a local copy of data from the b to the c element, but such transfers are likely less costly (and our cost formula ignores them).

Instantiation/Generalization This rule is depicted in Figure 2.6. Let p be one of the peers providing f. This rule turns an abstract service call to $f@any$ into a concrete call to $f@p$. Moreover, it can change a concrete call to $f@p$ to an abstract service call to $f@any$, if it knows that f is an abstract service.

Recall from Section 2.2.1 that OptimAX' cost model assigns maximum cost to the activation of calls at *any*. As a consequence, applying the instantiation rule always reduces cost. Moreover, when several peers provide f, different cost reductions can be obtained. Documents with calls to services $@any$ give more options to the optimizer. When services are queries (Section 2.1.1), plans produced by instantiation resemble distributed strategies in mediator systems [55].

Generalization is needed when there is a concrete call $f@p_{old}$ of an abstract service f which is costly. In this case, to minimize execution cost, a generalization ($f@any$) and a instantiation ($f@p_{new}$), selecting the right peer, are needed.

Query composition/decomposition The *query composition/decomposition* rule is shown in Figure 2.7. The rule focuses on calls to services implemented by XML queries, such as q, q_1 and q_2 in the Figure. These queries are such that $q \equiv q_1(q_2)$, i.e. for any XML input t, $q(t) = q_1(q_2(t))$. The upper document calls the query q, while the lower document nests the call to q_2 as a parameter of the call to q_1. In this rule, p can also be *any*. Query composition (going from the lower part to the upper part in Figure 2.7) reduces costs by cutting the overhead of one activation.

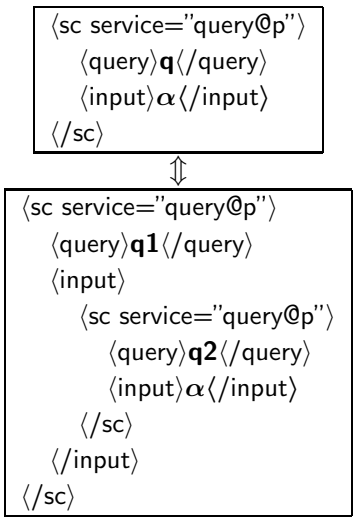


Figure 2.7: Query (de)composition rule.

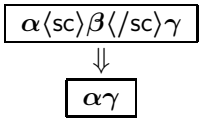


Figure 2.8: Useless call elimination rule.

Query decomposition (going from the upper part to the lower part) may reduce costs if the sub-queries q_1, q_2 can be handled very efficiently by some processor unable to handle the full query q. For example, q_1 may be an XPath query which may be answered using an index [5], and q_2 is an XML construction query applying on the results of q_1.

More generally, the query decomposition rule applies for any queries $q, q_1, \ldots, q_n$ such that $q \equiv q_1(q_2, \ldots, q_k)$. The rule *de facto* integrates XML query optimization as a sub-problem of the larger AXML optimization problem.

Useless call elimination The useless call elimination rule is shown in Figure 2.8. This rule eliminates calls with anticipated empty results (recall the example for the empty result call in Section 2.2.1).

Flattening rule This last rule highlights an interesting interplay between XML queries and the AXML formalism. The rule is illustrated in Figure 2.9. In the document at the top of the Figure appears a call to the generic query service (*gqs*) that we have introduced in Section 2.1. The *gqs* must evaluate a query of the form $q_1(\mathsf{document}(q_2))$; What is important here is that the data

```
⟨sc⟩ gsq(q1(document(q2)) ⟨/sc⟩
                ⇓
⟨container⟩
            ⟨sc id="#1"⟩gsq(q2)⟨/sc⟩
⟨/container⟩
⟨sc buildGSC afterActivated="#1" ⟩
     ⟨param⟩ ../container/* ⟨/param⟩
     ⟨param⟩ q1 ⟨/param⟩
⟨/sc⟩
```

Figure 2.9: Flattening rule.

source on which the query applies is not known, but must be determined by evaluating a sub-query q_2, before one can actually evaluate q_1.

The purpose of the rule is to decompose the query expression $q_1(\mathsf{document}(q_2))$, using AXML. To that purpose, two calls are introduced in the document at the bottom of Figure 2.9:

- The first call has the id #1 and it adds in the document the list of document URIs resulting from the evaluation of q_2. The rule encapsulates these URIs as children of a new ⟨container⟩ element.
- The second call invokes the built-in *buildGSC* service, which *b*uilds *g*eneric *s*ervice *c*alls. The service is continuous, and it has two parameters. The first parameter must be instantiated to a stream of document URIs. The second parameter must be instantiated into a stream of queries. For each given document URIs and query string, *buildGSC* returns a service call element, which will be added as a sibling of the call to *buildGSC*, in usual AXML fashion.
  The call to *buildGSC* must be activated after the document URIs start arriving, thus, after the service identified by #1.

The flattening rule moves the data dependency of q_1 on q_2, from the level of the XQuery expression $q_1(\mathsf{document}(q_2))$, to the level of AXML. The interest of the rule is that first, q_2 can be optimized in itself using the other AXML/XQuery optimization rules, and second, the remainder query part, namely q_1, is brought to a simpler form of *gqs* calls, which also lend themselves to optimization.

For readability, we introduced the *buildGSC* service when presenting the flattening rule. However, *buildGSC* is nothing but a particular usage of the generic query service *gqs*. More precisely, a call of the form $buildGSC(\$in_1, \$in_2)$ is in fact realized as a call of the form $gqs(q_{gsc}, \$in_1, \$in_2)$, where $q_{gsc}(\$in_1, \$in_2)$ is the following XQuery query:

```
for $x in $in1, $y in $in2
return ⟨sc gqs@any⟩ ⟨document⟩{$x}⟨/document⟩ ⟨query⟩{$y}⟨/query⟩ ⟨/sc⟩
```

Since XML queries build XML results, and XML results may include service calls, it is possible using AXML to write code that will modify itself – a rather powerful language property.

Finally, observe that the query ../container/* appearing in a parameter in the lower part of Figure 2.9 can be evaluated by another call to *gqs*, over the document itself (to find the URIs returned by q_2). We omit the details.

Estimating the rule-based optimization search space Let d be a document such that all calls in $SC(d)$ refer to specific peers (not *any*) and assume delegation is the only rule. Let P be a set of peers known to the optimizer. Each call can be delegated to each $p \in P$, leading to a search space of size $|SC(d)|^{|P|}$. On the contrary, assume now that all calls in $SC(d)$ refer to *any* and enable instantiation: the size grows to $|SC(d)|^{2|P|}$. The impact of the other rules described above is more difficult to quantify since it depends heavily on the document. In any case, exhaustive search may be quite costly. Other optimization techniques, like divide and conquer [45] are not optimal in the presence of factorization.

2.3.2 Implementation issues

OptimAX is implemented in Java and integrated with the recent v.2 of the AXML engine [23]. This version relies on an XQuery-compliant database (eXist [59]) to store and update AXML documents, enabling it to scale up; the previous AXML engine was limited by its in-memory, DOM-based document management. Axis2 [22], which is an implementation by Apache Software Foundation of the SOAP protocol [57], is used for Web service messaging. The activation order constraints described in Section 2.1.2, and continuous query services as described in Section 2.1.1 were specified and implemented in AXML v2 as part of the optimizer integration effort. We discuss here some notable engineering issues.

Detecting tree equivalence We consider the limited case when equivalence of two AXML tree is detected by checking two-way containment mappings. The condition is sufficient but is not necessary.

This check is performed at optimization time between two unevaluated trees, t_1 and t_2. A containment mapping h from a t_1 to t_2 is a mapping from the nodes of t_1 to the nodes of t_2 such that:

- h preserves node types: $\forall n$, n and $h(n)$ describe the same data or service call node;
- h preserves structural relationships: whenever n is a child of m in t_1, $h(n)$ is a child of $h(m)$ in t_2;

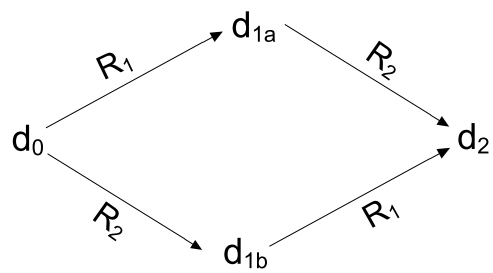


Figure 2.10: Confluence example.

– h preserves activation order constraints: for every activation constraint *constr* that exists at t_1 between service sc_1 and sc_2, there exists a $constr'$ of the same type between $h(sc_1)$ and $h(sc_2)$ at t_2.

To make the above comparison efficient, each node carries a hash code describing itself and its descendants. Whenever we want to compare two subtrees, firstly we compare their hash codes and only if they are equal, we proceed to the above detailed check.

Detecting loops The first and the most common problem that appears are loops during the search space exploration. The loops are due to the presence of bi-directional rules, such as instantiation/generalization and query composition/decomposition. For example, consider a document d and assume that the decomposition rule can be applied to service call sc of d which results to its decomposition to sc_1 and sc_2. Continuing the optimization process with the resulting document d', we apply the composition rule trying to compose the service calls sc_1 and sc_2 to one service. As we can imagine the resulting service call will be the original sc and the new document d'' will be equal to d.

The solution is to keep track of the documents that we have visited, in order to know when we visit a document for a second time.

Confluence Another problem that appears very often is the problem of confluence, illustrated by the sketch in Figure 2.10. When various optimization rules commute, we may reach the same document using different optimization paths. For example, in Figure 2.10 we reach document d_2 from d_0 by following the upper path and applying first R_1 and then R_2. Thereafter, we may reach it a second time by following the lower path and applying first R_2 and then R_1. When visiting for first time document d_2, we will continue its optimization, if needed. However, in the following visits we should detect that this document has been already visited and processed and we should not proceed to any further processing.

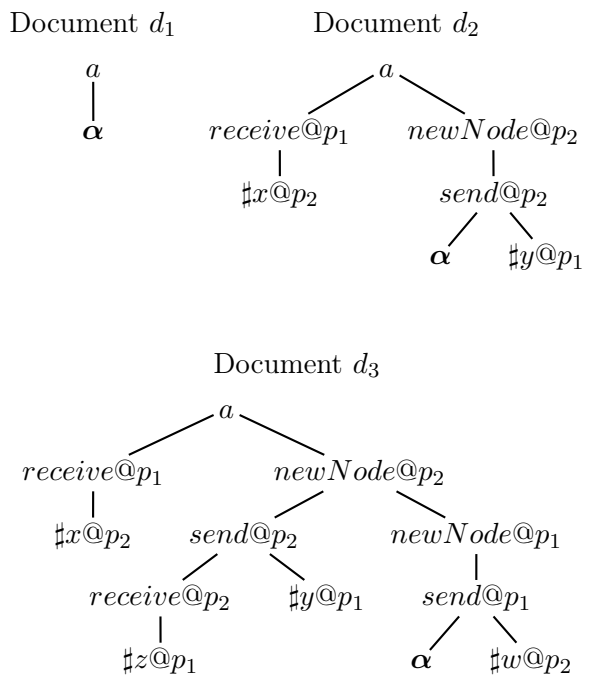


Figure 2.11: "Ping-pong" example introduced by repeated delegation.

The solution to this problem is the same that we use to solve the problem of loops: we keep track of all the documents that we have visited.

Delegation loops (the "ping-pong" problem) A more subtle problem arises when we try to delegate a subtree α to a peer from which that subtree has already been delegated. The example of Figure 2.11 shows the exact problem.

A document d_1 resides at peer p_1. We decide to delegate the subtree α to peer p_2, which results in the document d_2. In the sequel, we delegate α back to p_1. The result of this action is the document d_3. By comparing d_1 and d_3, we see that in both documents the α subtree resides at peer p_1. However, in the case of d_3 the subtree α has to pass from peer p_2 to be installed to peer p_1. Moreover, the results of any service calls in α which are activated during the evaluation of d_3, will need to arrive to p_1, then will be sent to p_2 and then sent back to p_1 (Figure 2.12).

This kind of delegations are obviously needless and may continue endlessly. OptimAX discovers them as soon as they appear, by examining to which peer each subtree of a document has been delegated, and does not develop plans that have this kind of loops.

In the case where we do not stop the development of these plans, the search space becomes infinite even for simple problems with one service

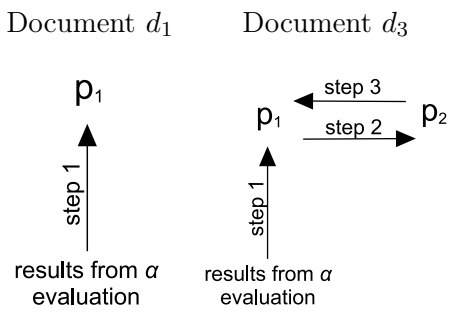


Figure 2.12: Evaluation result arrival to the documents d_1 and d_3 of Figure 2.11.

call and two participating peers. The plans that contain these kind of loops, will be classed as bad plans by OptimAX' cost estimation function (which is in the lines of the cost function described at Section 2.2.1) and will never be proposed as good plans. However, this entails that we stop the optimization after some repetitions since the search space will be infinite.

Preserving document validity during delegation When applying delegation some precautions have to be taken in order not to produce an invalid document. The two special cases where the delegation can not be performed are shown in Figure 2.13. In this Figure and in the sequel, dashed lines represent references. They may represent

- the *where* part of a *send* or a *receive*, which points to the respective *receive* and *send* (Figure 2.14);
- the *what* part of a *send*, in the case that it is a reference to a service call (Figure 2.13).

Considering the document at left in the Figure, we want to delegate the grey subtree rooted at sc_1 to a different peer. However, a *send* node in this subtree expects to read the results of the evaluation of the service call sc_2 located outside the subtree. If we delegate the grey subtree to another peer, the results of sc_2 will be produced at that peer, and therefore will be unavailable to the *send* call which expects to ship them. To avoid this inconsistency, we disallow such delegations. Note that this type of *send*s, which copy evaluation results from one place of the document to another, can be introduced by the factorization rule. Thus, such a problem may appear very easily by applying just one optimization rule.

The document at right in Figure 2.13 illustrates the opposite scenario. Consider again the grey subtree rooted at sc_1. It contains a service call sc_2, which is pointed by the what child of a *send* outside the subtree. If we delegate the grey subtree to a distant peer, the *send* becomes isolated from

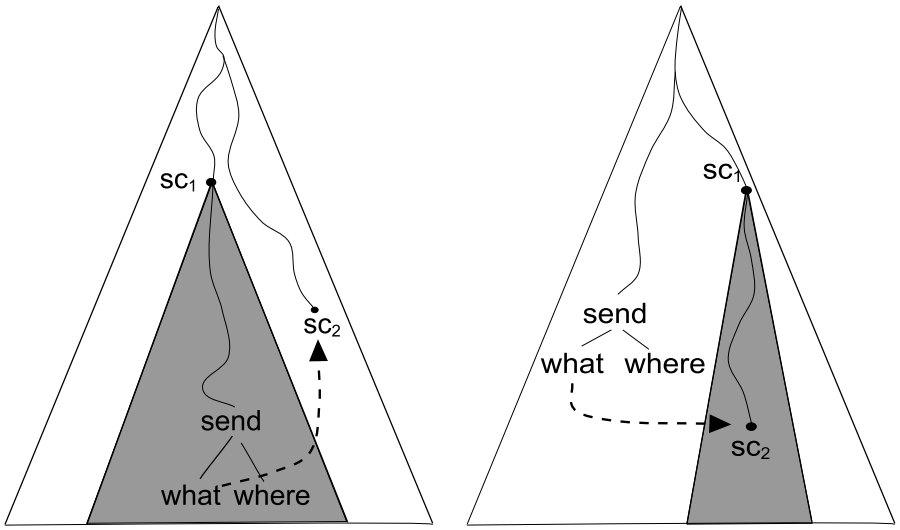


Figure 2.13: Problematic delegation cases.

its stream of data to be sent, and cannot function.

Factorization constraints In the case of factorization, the consistency check is more complex than in the case of delegation. When applying a factorization rule, we replace a subtree *sub* with a *send* and a *receive* which are going to copy from another part of the document, the data that *sub* was going to produce. The deletion of *sub* entails two important computations:

1. finding the set of nodes that should be deleted;
2. verifying that these nodes are not data providers for service calls that remain in the document after factorization.

Figure 2.14 depicts a document just before applying the factorization rule. We are planning to replace sc_2 with a *send* and a *receive* that are going to copy the results of sc_1 where sc_2 is placed. We start by identifying the nodes that should be deleted from that document (*nodesToBeDeleted* set). These are the nodes of the subtree rooted at sc_2, and all the nodes that provide data to this subtree. The latter means that after adding to *nodesToBeDeleted* the nodes under the sc_2, we should check the *nodesToBeDeleted* for any *receive*s, like the $receive_2$ in Figure 2.14. For each *receive* found, there is a *send* somewhere in the document that will provide it with data, such as $send_2$ in the Figure. Thus, the subtrees rooted at these *send*s should be added to *nodesToBeDeleted*, and so on recursively. The computation ends when all the *send*s of the *receive*s found in *nodesToBeDeleted* have been visited.

The second step is the verification of the document consistency. We have identified the nodes that provide data to sc_2 and we are ready to delete them.

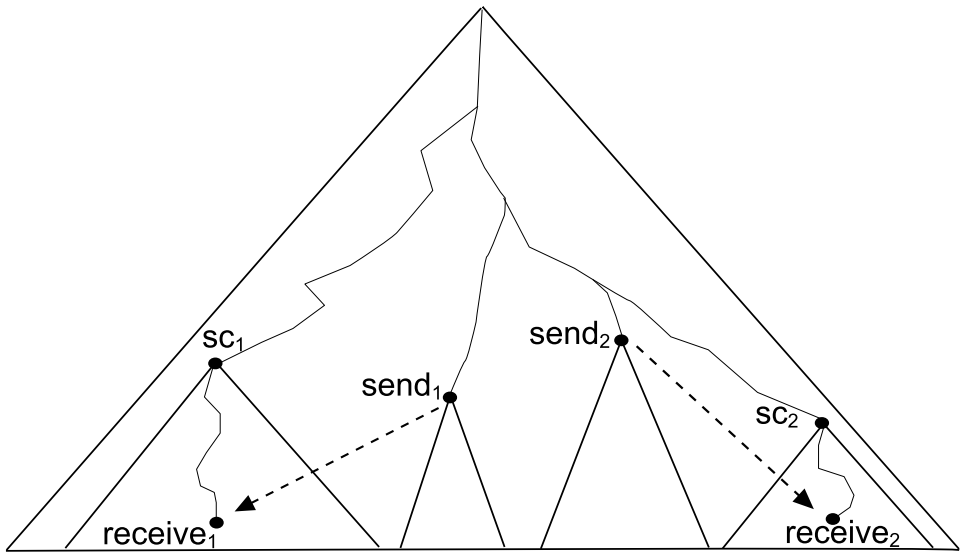


Figure 2.14: Factorization example.

However, there is a possibility that these nodes provide data to another part of the document. For example, consider the node $send_1$ inside the *nodesToBeDeleted* set. We can only delete such a node after checking that its corresponding *receive* node is also among *nodesToBeDeleted*. If this is not the case, factorization cannot be applied.

2.3.3 Search strategies and heuristics

The applications we consider for AXML have very varied profiles. One class of applications focuses on subscriptions [12], where factorization is crucial (to avoid duplicate data transfers) and query composition may also apply (to efficiently filter out subscriptions). Other applications consider distributed data management workflows [46], where instantiation and delegation are central. As another example, our current WebContent project [60] focuses on XQuery processing in a structured P2P network, based on a distributed XML index; the main rule is query decomposition isolating the largest subquery the index may handle. The total time budget given to the optimizer also varies with the application.

To accommodate such a variety of settings, we have devised a simple XML dialect for specifying the optimizer's search strategy. Each strategy is a sequence of *steps*. Each step applies a given *search algorithm* (which can be: depth-first or breadth-first, and possibly attempt to rewrite the cheapest plan first), using a given *rule set*, and with an *upper bound* on the number of plans developed.

```
⟨strategy⟩
    ⟨step⟩
        ⟨algo⟩BF⟨/algo⟩⟨algo⟩CD⟨/algo⟩
        ⟨repetition⟩
            ⟨atom⟩DELG⟨/atom⟩
            ⟨atom⟩FACT⟨/atom⟩
            ⟨card⟩100⟨/card⟩
        ⟨/repetition⟩
    ⟨/step⟩
    ⟨step⟩
        ⟨algo⟩DF⟨/algo⟩⟨algo⟩CD⟨/algo⟩
        ⟨repetition⟩
            ⟨atom⟩DELG⟨/atom⟩
            ⟨atom⟩FACT⟨/atom⟩
            ⟨card⟩20⟨/card⟩
        ⟨/repetition⟩
    ⟨/step⟩
⟨/strategy⟩
```

Figure 2.15: Sample strategy file.

For instance Figure 2.15 shows an OptimAX strategy file consisting of two steps. The first step may develop 100 plans in a breadth-first, cost-driven manner, then the second step may apply depth-first, cost-driven search producing 20 more plans, then the best plan found so far is chosen. In both of the steps, only the delegation and the factorization rewriting rules are used.

If no specific strategy is provided, the default strategy runs a single step, using the depth-first, cost-driven strategy, and develops 100 plans using all rules. In our experience, this simple strategy lead to useful rewritings.

2.4 Case studies

In this Section we discuss two case studies involving distributed optimization in a peer-to-peer setting which can be handled by OptimAX. The scenario in Section 2.4.1 involved peer interacting in a unstructured peer-to-peer network in a read- and write-intensive application. Section 2.4.2 considers a different situation, when large data volumes are handled in a structured peer-to-peer network in which two structured XML indices coexist.

2.4.1 Distributed software development in EDOS

The following distributed data management scenario is closely inspired from a real application encountered in the *EDOS* (efficient distribution of open source software) EU project, concerning the automatic management of the Mandriva (formerly known as Mandrake) Linux distribution. The scenario concerns the collaborative development and distribution of software packages. Several developers, distributed all over the world, write updates for a set of software packages. Each developer works at his own location, and pushes his updates to one or several servers geographically close to him. Each server hosts some, but not all, of the distribution's packages. In this context, a typical user query is: *Whenever there is a new update on the Emacs package, I want to receive the update and the name of its developer.*

In this section, we first analyze the needs of such applications in Section 2.4.1.1. We then show how to handle them using AXML and OptimAX in Section 2.4.1.2.

2.4.1.1 Application analysis

Let us first analyze some characteristics and requirements of this application. First, in a *distributed* setting, this data need must be answered *efficiently*, regardless of where the data comes from, and, if possible, regardless of *how the user was able to express this need using some query language*. Second, the query has an inherent *continuous*, *incremental* nature: the user should not receive every day information on all Emacs updates ever written, but only receive new updates as they arise. Finally, the application is *highly dynamic*:

- The set of packages under development changes over time;
- The overall developer community changes over time;
- The set of packages a given developer contributes to changes over time, as well as the peers where his updates are published.

Clearly, the scope of changes in the system is very wide. It is thus an application requirement that once a query has been defined, its execution proceeds seamlessly and efficiently independently of the particular state of the system.

Other natural requirements for such applications include *flexibility, genericity and ease of deployment*. Writing custom application code specific to each such data need is cumbersome and does not scale. *Conceptual uniformity* of the models employed to capture the application is also desirable. Modern content management software is quite complex, making applications very difficult to optimize. In particular, data management operations (filtering, restructuring, reformatting, data transfers, subscription etc.) are increasingly delegated to software tools other than a database management server. As a consequence, a database optimizer's power is confined to a

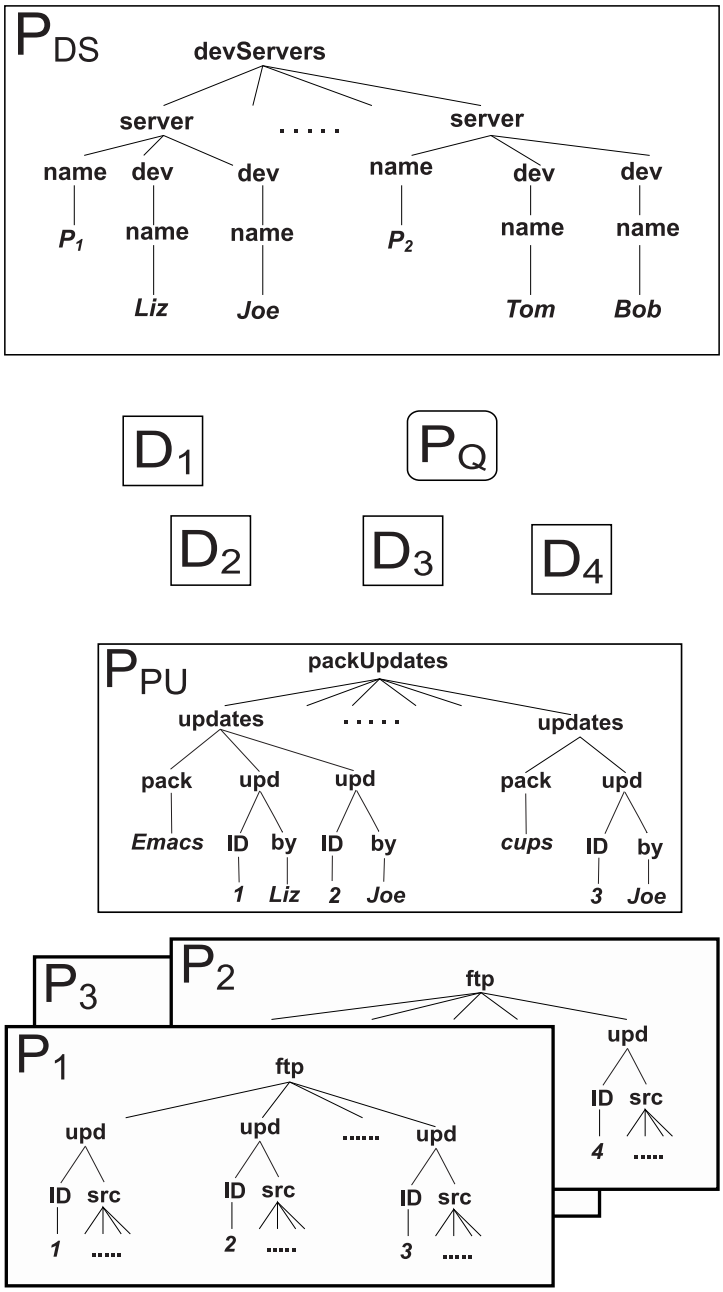


Figure 2.16: Motivating example: distributed software management.

smaller and smaller part of the global computation. In contrast, deploying an application based on a few paradigms (ideally one!) allows *reasoning* over the application globally, and opens avenues for optimization.

```
for    $m in document(“P_DS/devServers.xml”)//server,
       $u1 in document(“P_PU/packUpdates.xml”)//updates,
       $u2 in document(concat($m/name,“/ftp.xml”)//upd
where  $u1/pack=“Emacs” and $u1/ID=$u2/ID and
       $m/dev/name=$u1/by
return {$u2/src}, {$u1/by}
```

Figure 2.17: EDOS application example expressed in XQuery.

To explain the technical issues raised by such applications, we layout the data placement on various peers in Figure 2.16. In this Figure, trees stand for XML documents, node labels in regular font denote element or attribute names, while labels in italic denote values (leaves). On peer P_{DS}, the document `devServers.xml` lists the peers on which each developer posts his contributions. On peer P_{PU}, the document `packUpdates.xml` associates developers with the updates they produced for each package. On peers $P_1, P_2, \ldots, P_k$, software updates are available for download. These peers play the role of servers with respect to the possible users that may download code from them. Information on a given software package is scattered in the system: one peer, say P_i, holds the update IDs and code, the P_{PU} peer associates packages with their updates, while the peer P_{DS} indicates the server where each developer contributes his work. The query peer is denoted P_Q, while developers work on peers D_1, D_2 etc.

We may attempt to solve our problem by writing an XQuery query such as the one shown in Figure 2.17. In this query, `$m` iterates over the code-hosting servers to which some developer may upload code. The list of these servers can be obtained from the peer P_{DS}. The query variable `$u1` iterates over all the package update entries, listed at the peer P_{PU}. Recall that P_{PU} only stores the ID and contributor of each update, but does not specify where each update can be downloaded from.

The variable `$u2` iterates over all the updates found at *any* code hosting server. Indeed, the URIs of the documents in which `$u2` must be matched are dynamically computed from the bindings of the variable `$m`. The first condition in the `where` clause restricts the query to the updates of Emacs packages. The remaining conditions ensure that `$u1` and `$u2` reference the same code update, and that the author of `$u1` is the same as the contributor listed at P_{DS}.

Clearly, the query in Figure 2.17 refers to distributed data and must be evaluated in a distributed fashion. Several alternatives can be considered for its evaluation.

– One may ship the document `devServers.xml` from P_{DS} to the query peer P_Q, the document `packUpdates.xml` from P_{PU} to P_Q, and finally all the documents `ftp.xml` found at the server peers P_1, P_2, P_3 etc. Then,

the query could be evaluated locally at P_Q. This approach entails important network transfers, since all data in the network is sent at P_Q.

- An improvement can be made by exploiting the selection on Emacs code packages. Indeed, one could transfer from P_{PU} to P_Q, instead of `packUpdates.xml`, only the result of the sub-query:
  `document(‘‘packUpdates.xml’’)/updates[pack=’’Emacs’’]`
  However, all `ftp.xml` documents from the code servers still have to be transferred.
- To avoid transferring all `ftp.xml` documents, one may push to all the code server peers P_i a query of the form `document(‘‘ftp.xml’’)//upd[ID=$ID]`, where `$ID` stands for the package IDs resulting from the Emacs package query evaluated at P_{PU}. This would still require contacting all code serve peers, which is not satisfactory considering how many user queries similar to our example may exist in the system.
- With the help of a distributed index, one may avoid systematically querying all code server peers. Indeed, one may index the content of all the `ftp.xml` documents e.g. in a DHT-based XML index such as KadoP [5, 6]. Using the index, one may efficiently locate the `ftp.xml` document(s) containing updates with the ID equal to `$ID`.

Observe that a lookup in a global distributed index (such as KadoP or close competitors [27, 33]) cannot answer the query. This is because each document is indexed separately, whereas in the EDOS application, different documents from peers initially unknown must be joined in order to find results. At best, one may use a global XML index twice: once to obtain the IDs of Emacs packages, and a second time to locate the desired packages in the network consisting of the $P_1, P_2, \ldots, P_k$ code server peers.

In the sequel, we consider the possible optimizations that may be brought to the EDOS application, if we chose to model it using AXML and optimize it via OptimAX rewritings.

2.4.1.2 Optimizing the EDOS application

To enable OptimAX to handle the application, we must first express it under the form of an AXML document, namely $d_1@P_Q$ shown in Figure 2.18. In this document, a call to the generic query service is made, requiring the evaluation of application query (initially shown in Figure 2.17).

Writing conventions For simplicity, in the sequel, in AXML documents we will use names of queries with some location peer, e.g. $q@p$ or $q@any$, to denote a call to the generic query service of the form $gqs(q)@p$, respectively, $gqs@any$. Moreover, in the Figures of this section, whenever an AXML document is represented at the top, including calls to the generic query service parameterized by various queries, the definitions of those queries

$d_1@P_Q$ $\boxed{q@any}$

q

```
for    $m in document(“P_DS/devServers.xml”)//server,
       $u1 in document(“P_PU/packUpdates.xml”)//updates,
       $u2 in document(concat($m/name,“/ftp.xml”)//upd
where  $u1/pack=“Emacs” and $u1/ID=$u2/ID and
       $m/dev/name=$u1/by
return {$u2/src}, {$u1/by}
```

Figure 2.18: Original AXML document d_1 which resides in peer P_Q.

appear underneath the AXML document, for reference.

Coming back to our presentation of the example in Figure 2.18, observe that in $d_1@P_Q$, the peer that must evaluate this query is set to *any*, denoting that no assignment of work has been done yet.

The first optimization step which OptimAX may apply is to decompose q into smaller sub-queries. These are:

- $subQ_1$, which is a join between the data of P_{DS} and P_{PU}, and will provide the ids of the useful updates, the names of their authors and the peers where these authors publish their updates;
- $subQ_2$, which is a selection of the right updates based on the output data of $subQ_1$.

Such query decomposition is based on standard XQuery syntactic analysis. OptimAX does not implement such functionalities itself, and instead may rely on one or several static XQuery analysis tools for this purpose. A particular example of such an XQuery analyzer, which has been actually integrated with OptimAX, is Tree Graph Views [54]. Other similar query decomposition methods are based on tree pattern extraction from XQuery queries [20, 44]. Figure 2.19 shows $subQ_1$, $subQ_2$ and the new document d_2 that is produced by this decomposition.

Optimizing subQ$_1$ Similarly, OptimAX relies on an XQuery analyzer to further decompose $subQ_1$ into the simple queries $subQ_{1a}$, $subQ_{1b}$ and $subQ_{1c}$ shown in Figure 2.20.

The query $subQ_{1a}$ retrieves the `server` data from peer P_{DS}, while $subQ_{1b}$ retrieves only the IDs and the developer's names for the Emacs updates and $subQ_{1c}$ joins the output of these two services on the developer name. Figure 2.19 depicts the document d_3 that was the result of this decomposition.

$d_2@P_Q$

$subQ_2@any$

$subQ_1@any$

$subQ_1$

```
for    $m in document(‘‘P_DS/devServers.xml’’)/server,
       $u1 in document(‘‘P_PU/packUpdates.xml’’)/updates
where  $m/dev/name=$u1/by and $u1/pack=‘‘Emacs’’
return <result>
                <server>{$m/name}</server>
                <ID>{$u1/ID}</ID>
                <by>{$u1/by}</by>
       </result>
```

$subQ_2$

```
for    $in in inputService,
       $u2 in
       document(concat($in/result/server,‘‘ftp.xml’’))//upd
where  $in/result/ID=$u2/ID
return {$u2/src}, {$in/result/by}
```

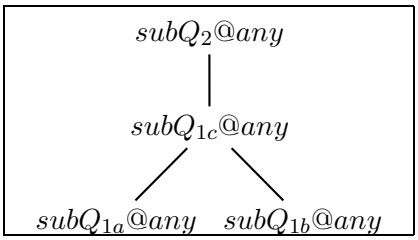
Figure 2.19: Decomposing query q of d_1 into $subQ_1$ and $subQ_2$.

Since $subQ_{1a}$ scans the data of server P_{DS}, its execution peer is set to P_{DS}. Similarly, the execution peer of $subQ_{1b}$ is set to P_{PU}. The next decision that has to be taken is the execution peer of the join $subQ_{1c}$. OptimAX should choose one peer that has the power to perform the join and has a good network connection, at least, with the peers that execute the input queries in order to perform fast the operation. Let's assume that peer P_{DS} is chosen based on various statistics collected in the past and d_3 is rewritten to d_4, shown in Figure 2.21.

Note that the above step is not enough to minimize the execution costs of the subtree rooted in $subQ_{1c}$. The evaluation steps that we should follow when evaluating the subtree rooted in $subQ_{1c}$ of d_4 are the following:

1. P_Q will contact P_{DS} and it will ask the evaluation of $subQ_{1a}$,
2. P_Q will contact P_{PU} and it will ask the evaluation of $subQ_{1b}$,
3. P_{DS} will evaluate $subQ_{1a}$ and will send the execution results to P_Q,
4. P_{PU} will evaluate $subQ_{1b}$ and will send the execution results to P_Q,
5. P_Q will send the execution results of $subQ_{1a}$ and $subQ_{1b}$ to P_{DS} and it will ask for the evaluation of the $subQ_{1c}$,
6. P_{DS} will evaluate $subQ_{1c}$ and will send the execution results to P_Q.

$d_3@P_Q$



$subQ_{1a}$

```
for    $m in document(“P_DS/devServers.xml”)/server,
return <result><server>{$m}</server></result>
```

$subQ_{1b}$

```
for    $u1 in document(“P_PU/packUpdates.xml”)/updates
where $u1/pack=“Emacs”
return <result><ID>{$u1/ID}</ID><by>{$u1/by}</by></result>
```

$subQ_{1c}$

```
for    $m in inputService subQ_1a,
       $u1 in inputService subQ_1b
where  $m/server/dev/name=$u1/by
return <result>
               <server>{$m/server}</server>
               <ID>{$u1/ID}</ID>
               <by>{$u1/by}</by>
       </result>
```

$subQ_2$

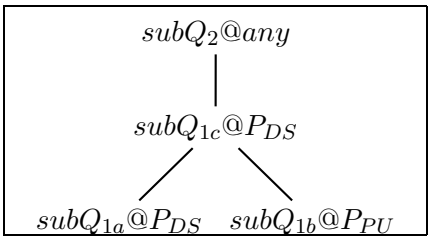
```
for    $in in inputService,
       $u2 in
       document(concat($in/result/server,“ftp.xml”))//upd
where  $in/result/ID=$u2/ID
return {$u2/src}, {$in/result/by}
```

Figure 2.20: Decomposing query $subQ_1$ of d_2 into $subQ_{1a}$, $subQ_{1b}$ and $subQ_{1c}$.

The network transfers of the above execution are depicted in the left part of Figure 2.23. Even if we have chosen P_{PU} to perform the join $subQ_{1c}$, we see that there is a lot of (unnecessary) network traffic because P_Q still coordinates the evaluation.

Our goal is to ask from P_{DS} to evaluate the subtree rooted in $subQ_{1c}$ and to return to P_Q the final result, which, in other words, means that we want to delegate that subtree to P_{DS}. This delegation is depicted in Figure 2.24. Figure 2.22 will help us understand what happens when we

$d_4@P_Q$



$subQ_{1a}$

```
for    $m in document(‘‘P_DS/devServers.xml’’)/server,
return <result><server>{$m}</server></result>
```

$subQ_{1b}$

```
for    $u1 in document(‘‘P_PU/packUpdates.xml’’)/updates
where $u1/pack=‘‘Emacs’’
return <result><ID>{$u1/ID}</ID><by>{$u1/by}</by></result>
```

$subQ_{1c}$

```
for    $m in inputService subQ_1a,
       $u1 in inputService subQ_1b
where  $m/server/dev/name=$u1/by
return <result>
                <server>{$m/server}</server>
                <ID>{$u1/ID}</ID>
                <by>{$u1/by}</by>
       </result>
```

$subQ_2$

```
for    $in in inputService,
       $u2 in
       document(concat($in/result/server,‘‘ftp.xml’’))//upd
where  $in/result/ID=$u2/ID
return {$u2/src}, {$in/result/by}
```

Figure 2.21: Assigning execution peers to $subQ_{1a}$, $subQ_{1b}$ and $subQ_{1c}$ (queries unchanged from Figure 2.20).

evaluate document d_5. The first line of that Figure depicts d_5 at the end of its evaluation, with the service calls that have been activated, shown with bold fonts. As we see, the subtree rooted in $send@P_{DS}$ will not be evaluated at peer P_Q, where d_5 resides, but at peer P_{DS}, where it is going to be installed by the newNode service. The new document which is generated and evaluated at peer P_{DS}, is shown at the second line of Figure 2.22. The right part of Figure 2.23 represents the network transfers that occur during the evaluation of the delegation.

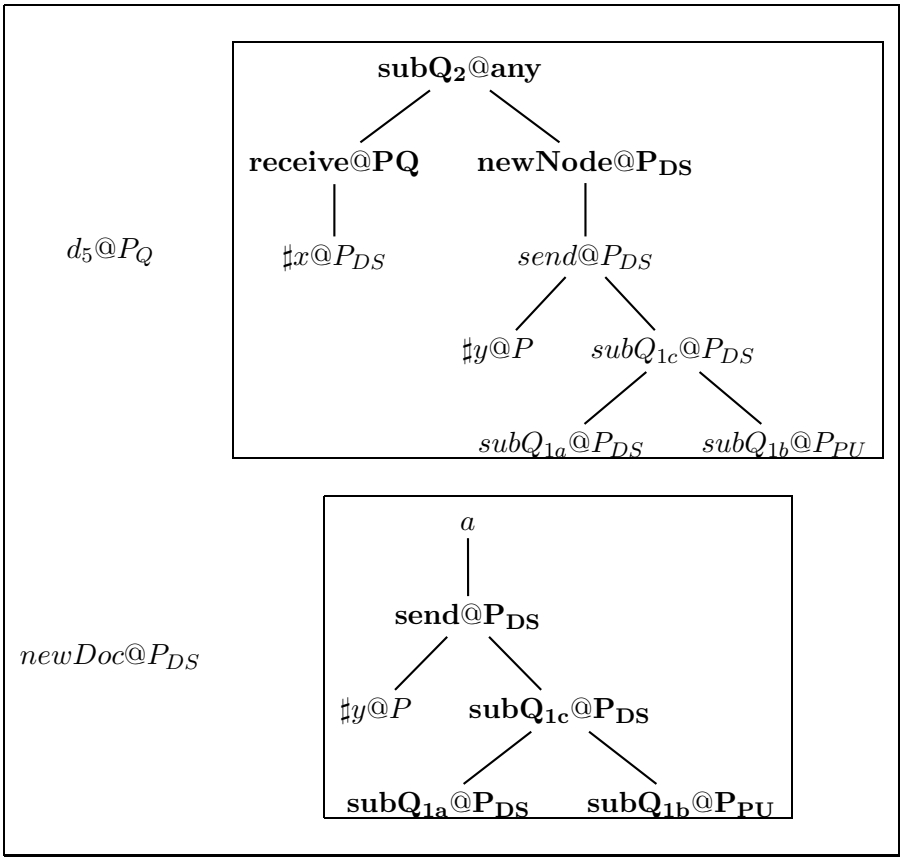


Figure 2.22: Evaluation analysis of $d_4@P_Q$.

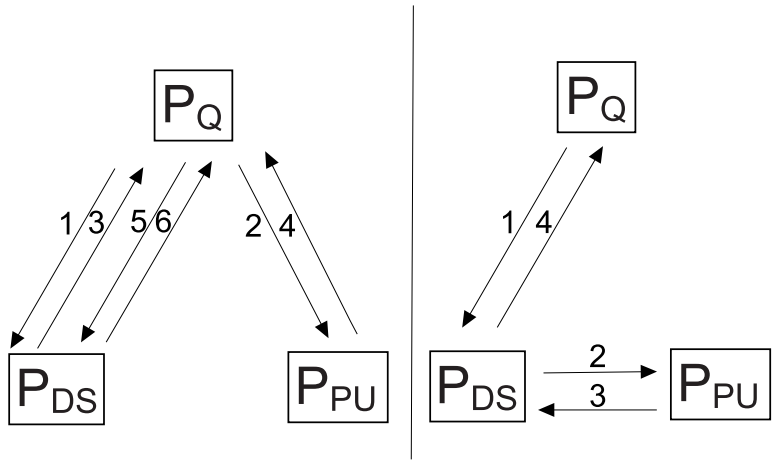


Figure 2.23: Ordering of the messages exchanged in the EDOS application.

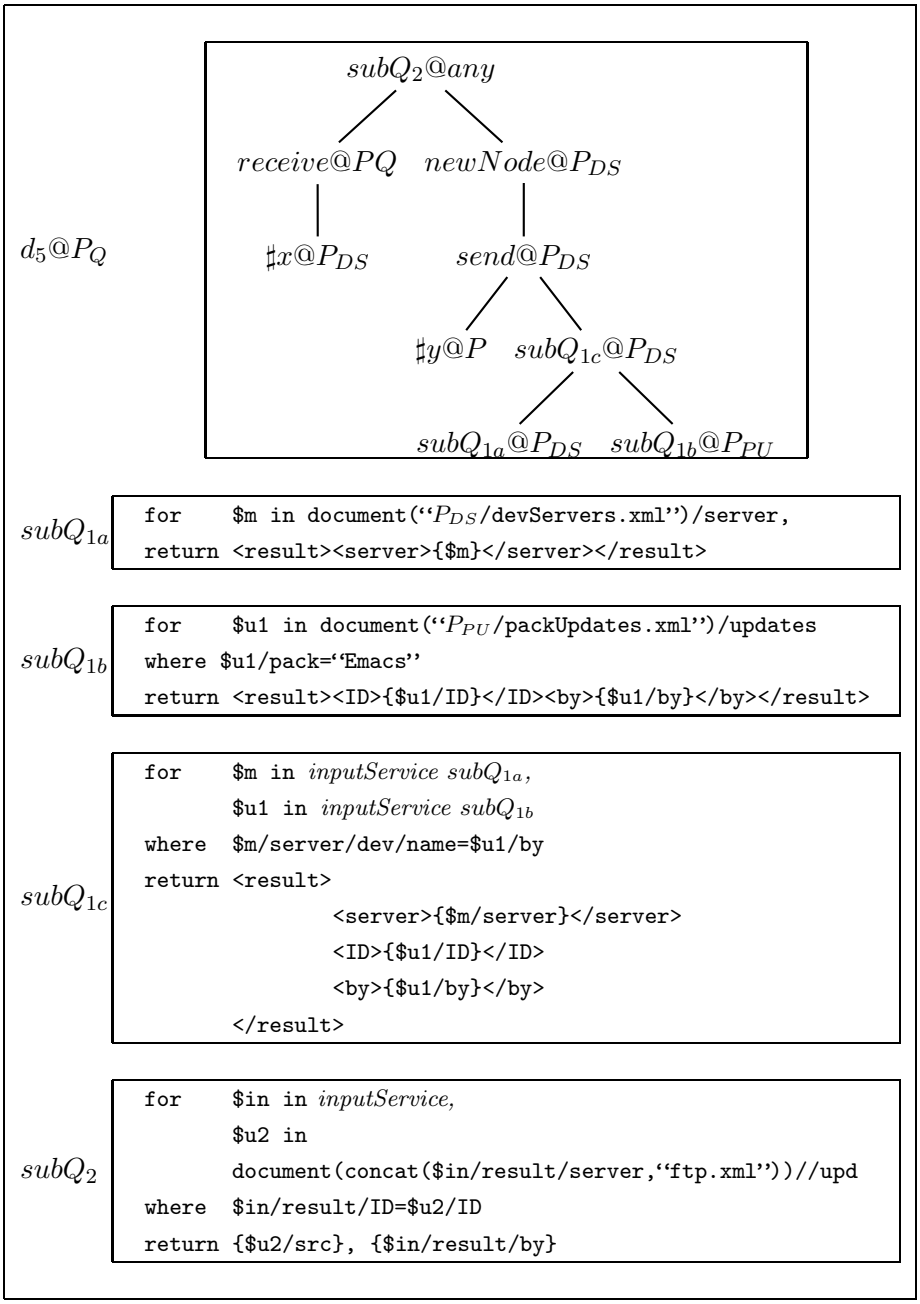


Figure 2.24: Delegating the subquery rooted in $subQ_{1c}$ of d_3 to peer P_{DS}.

Optimizing subQ$_2$ After optimizing the sub-query $subQ_1$, we are going to proceed to the optimization of $subQ_2$. We assign P_Q as its execution peer and we get document d_6 shown in Figure 2.25. We proceed by analyzing the

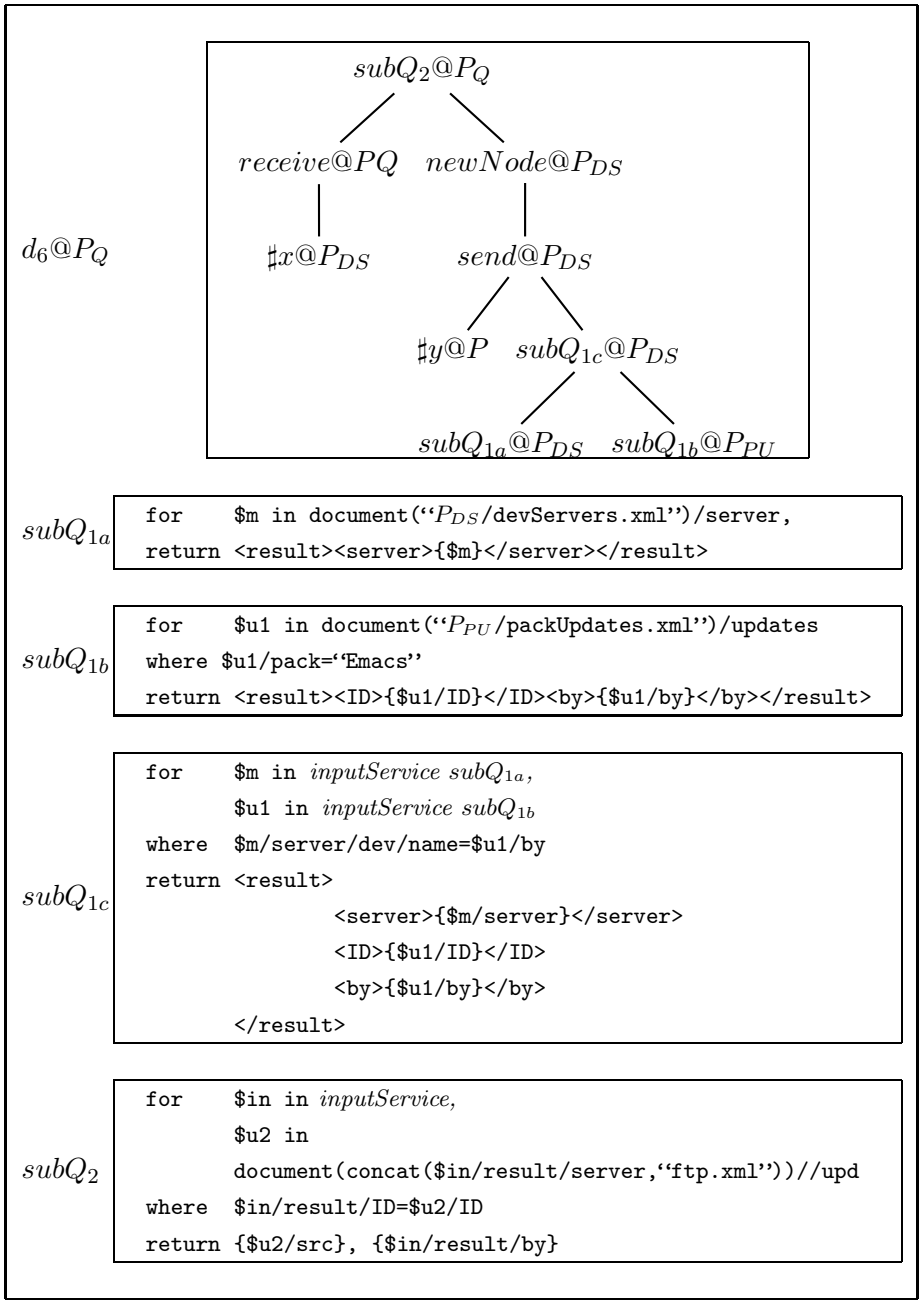


Figure 2.25: Changing the evaluation peer of $subQ_2$.

evaluation steps of $subQ_2@P_Q$. The hosting peer of $subQ_2$ is P_Q and is going to evaluate it locally. $subQ_2$ will receive data from $subQ_{1c}$ and based on these data, it is going to call every of the servers $P_1, P_2, \ldots, P_k$ indicated by $subQ_{1c}$ in order to receive updates. After receiving *all* the updates from each of the

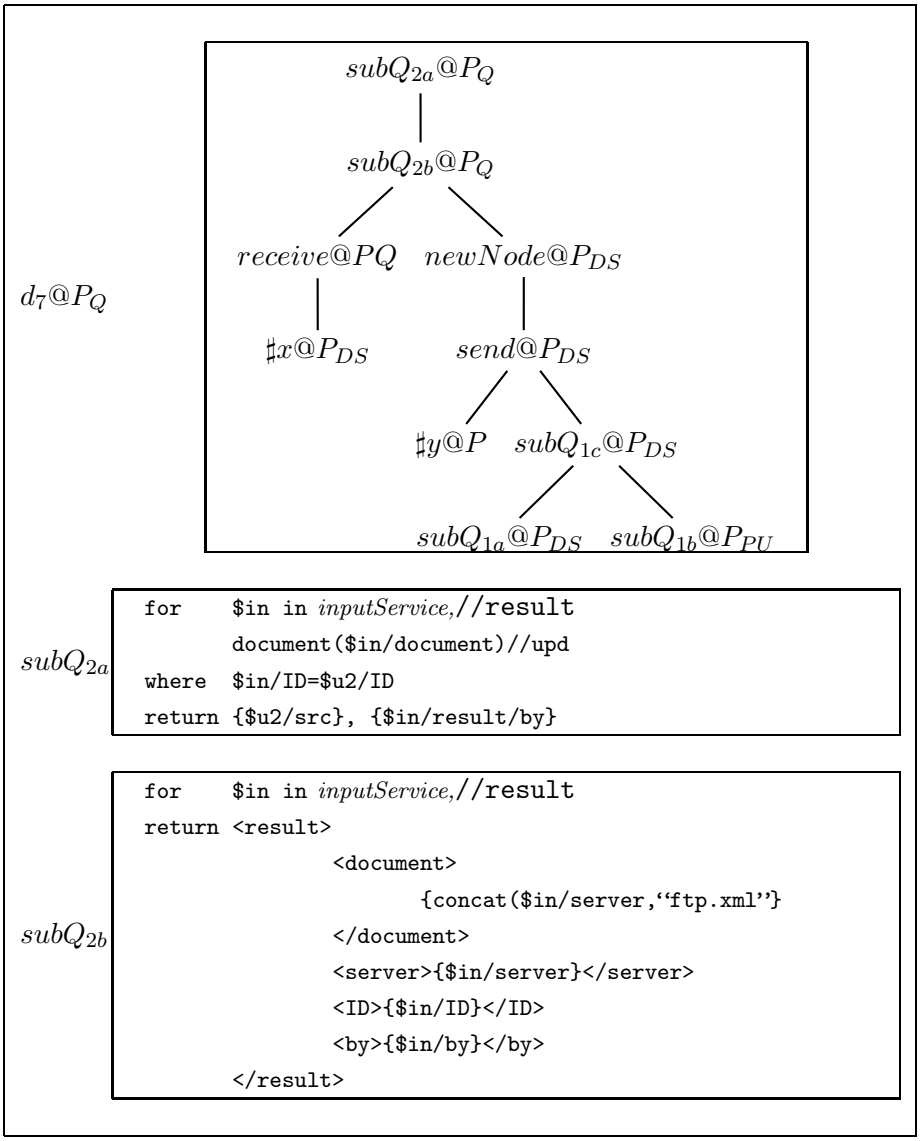


Figure 2.26: Changing the evaluation peer of $subQ_2$.

servers where the developers of Emacs updates publish their updates, it is going to filter them based on their ID in order to see which of them are the needed ones. The evaluation scenario described here is better than the $q@P_Q$ which retrieved updates from all the available peers, even from peers that did not contain such updates. Nevertheless further optimization is needed in order to filter the non-Emacs updates early enough, at $P_1, P_2, \ldots, P_k$.

This can be achieved by decomposing $subQ_2$ to $subQ_{2a}$ and $subQ_{2b}$ and by using the flattening rewriting rule. Figure 2.26 shows the new document

that results from the decomposition of $subQ_2$ (sub-queries $subQ_{1a}$, $subQ_{1b}$ and $subQ_{1c}$ are omitted due to lack of space in the image). Applying the flattening rewriting rule, will result a new document d_8 at which $subQ_{2b}$ will be the call in the *container* of the flattening rule and $subQ_{2a}$ will be the call that will be copied during evaluation as many times as the results of $subQ_{2b}$. This rewriting will allow us to call only the servers that have the Emacs updates and filter out the non-Emacs updates at these servers and not at the query server P_Q.

Until this point we have not discussed about *continuous* and *incremental* requirements identified in this application. However, the final produced query can be evaluated in a *continuous* fashion which means that the services $subQ_{1a}$ and $subQ_{1b}$ will continuously monitor the documents `devServers.xml` and `packUpdates.xml` for new information. Whenever a developer publishes a new update to peer P_*, the server P_{PU} will be updated with the new information. As soon as it is updated, new data will be provided to sub-query $subQ_{1b}$ which will filter out the non-Emacs related data. Then data will be passed to *buildGSC* which is going to create a new service call $subQ_{2a}$ for each new update. These service calls are going to fetch the correct updates and the needed information to peer P_Q.

2.4.2 Warehousing Web data in WebContent

This Section describes the usage that was made of the OptimAX AXML optimizer within the WebContent R&D project. First, Section 2.4.2.1 outlines the purpose of the project, its technical choices, and in particular its distributed peer-to-peer architecture. Section 2.4.2.2 then goes into the details of a WebContent peer, and presents its storage, indexing, and query processing functionalities.

Finally, Section 2.4.2.3 describes a particular technical point we solved in the WebContent platform thanks to the presence of OptimAX: integrating two different DHT-based XML indices and being able to use them both when processing a single query.

2.4.2.1 WebContent overview

WebContent is a research & development project funded by ANR, the French National Research Agency from 2006 to 2009. The project includes 4 INRIA groups (INRIA-Mostrare, INRIA-GEMO, INRIA-InSitu and INRIA-EXMO Project EXMO), several universities (e.g. Paris 6 and University of Versailles) and several industrial partners. Among them, the most important ones are CEA-LIST, EADS DS - SDRT/IPCC Team, Exalead (web search company) and Soredab (a consortium of food industry companies).

The goal of the project is information discovery, information understanding and information construction in all possible forms it can appear. In par-

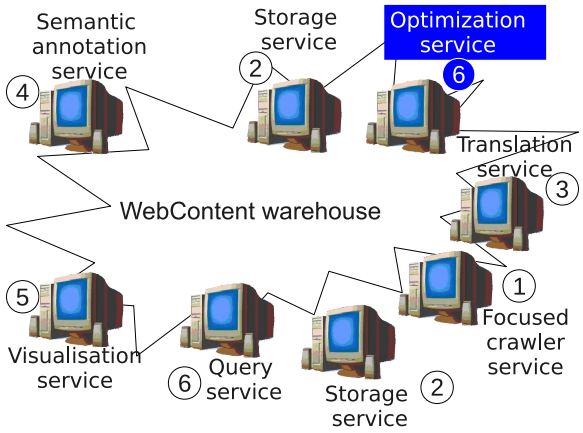


Figure 2.27: Service call oriented view of the WebContent project.

ticular we concentrate on the usage of XML & HTML documents and web services. We provide semantic integration utilities and intelligent search engines. The WebContent is a flexible and generic platform for content management and integrates Semantic Web technologies in order to show their effectiveness on real applications with strong economic or societal stakes. The first group of applications on the platform revolve around:

- economic watch in aeronautics
- strategic intelligence
- microbiological and chemical food risk
- watch on seismic events

The project is directly influenced by the results obtained from the older e.dot previous R&D project, focused on the construction of Web data warehouses. The Thus, XML syntax is widely used inside the platform for the meta data representation, ontologies are used to describe the application semantics, and web services are used to develop distributed applications. WebContent goes one step further and it incorporates more semantics following the W3C standard, interfaces, peer-to-peer architectures and multilingual support. More information on WebContent is available from the project Web site [60].

The WebContent architecture is centered around WebContent documents, conforming to a relatively loose schema, which captures generic document structure (title, sections, paragraphs etc.) as well as RDF annotations attached to document fragments and serialized in XML. The project follows a *a document warehouse approach*, allowing the storage, enrichment and exploitation of the documents by means of *WebServices*. Two architectures have been envisioned:

- a tightly coupled architecture, where services and documents are integrated by means of specifying precisely the URIs of each service and document to be used in a processing chain. Communication takes place via ESB [30].
- a loosely coupled architecture, decentralized setting aiming at processing large volume of data while preserving location transparence that is: users may specify *what* data must be processed, without the need to specify *where* the data is, and the system is then responsible for locating the respective data sources. In this case computers are connected via the Internet and communication takes place view Web services exchanging SOAP [57] messages.

In both cases, there can be several instances of each service, in particular, storage services are provided by multiple machines; and, services can be called from inside or outside the federation of sites implementing the warehouse. In particular, the GEMO group has been involved in devising the loosely coupled architecture. The following technical choices have been made:

- *D*istributed *H*ash *T*ables (or DHTs, in short [31]), for indexing the documents and query processing. DHTs are a class of decentralized distributed systems which provide a lookup service similar to hash table. Data items are inserted in a DHT and found by specifying a unique *key* for that data. The underlying algorithm determines which node is responsible for storing the data associated with any given key. Therefore each node maintains information (e.g. IP address) of a small number of nodes in the system, which are called neighbors, forming an *overlay network* and rooting messages to the overlay in order to store and retrieve keys. The DHT technology makes the process of indexing and locating documents easy and transparent to the user.
- Active XML, for coordinating the processing and integrating services developed within the platform. AXML was chosen since it allows assembling services from any providers, whether standard Web servers or AXML peers. As we have seen earlier in this Chapter, AXML documents endowed with proper data dependencies or activation order constraints can be used to sequence and combine calls to several services in a way that resembles a simple Web service workflow model.

A WebContent application deployed in the P2P architecture is outlined in Figure 2.27. The Figure depicts a WebContent warehouse consisting of documents gathered for the purpose of a Web market survey application for the EADS partner. More specifically, it shows:

1. a *focused crawler* service which returns Web documents related to specific domains. In our case it may return aircraft sales by Airbus and Boeing (for a continuous, on-line market survey) and food risk information, for a consortium of food companies seeking to organize and

structure information related to different food problems (contaminations, allergens etc.). The crawler service returns XML documents with information-rich headers (crawling date, origin site etc.)

2. a *storage* service which can be invoked to make the crawled document persistent in the WebContent warehouse
3. multiple *translation* services which are used to translate to and from English, French, Chinese etc
4. *semantic annotation* services which are invoked to analyze the text of the crawled pages and extract, e.g., specific aircraft brands, names of edible plants or bacteria that taint food etc. The annotations are added as a semantic header to the XML documents, under the form of XML-ized RDF snippets, and the modified documents are put back in the store
5. *visualization* and *query* services can be used at this point to exploit the corpus, either via advanced user interfaces (e.g. "fish-eye lens" view on documents) or by querying it, using a subset of XQuery (with full-text search) or SPARQL [50].

In this distributed setting, two issues arise:
- locating useful resources;
- efficiently implementing DHT-based versions of the Web services which are more easily provided in a centralized setting.

In the sequel of the section, we focus on the implementation of the distributed query service.

2.4.2.2 Structure of a Web Content peer

The architecture of a WebContent peer and the P2P communication among the peers is shown in Figure 2.28. We list the main modules and functionality, and then briefly discuss each of them.

Each WebContent peer provides the following functionalities:
- resource storage
- DHT-based resource indexing and lookup primitives
- low-level (execution engine) processing primitives
- high level (query interface/optimizer) data access operations. Of these, the execution engine primitives use the resource storage and the DHT indexing level, whereas the high-level data access operations only on the lower levels.

Resource storage is implemented by a local XML database to each peer. Our current implementation uses eXist [59]. We have considered the alternative of using MonetDB [41] which we rejected due to missing support for some particular syntax constrains brought by the WebContent format. Each eXist database provides query and update functionalities on the documents

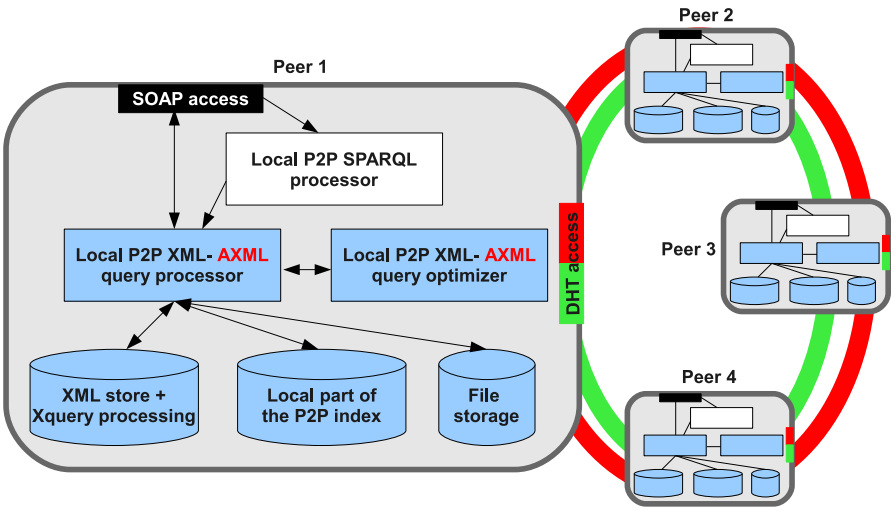


Figure 2.28: Internal architecture of a WebContent peer.

it stores.

DHT index/lookup is implemented by specialized modules which were developed by the project participants. More concretely, we have used two such platforms:

- **KadoP** [5] is an XML data management system which indexes data and answers tree pattern queries over this data.
- **Pathfinder** XML data management system [32] it is able to process tree pattern and range queries. A modified hash function permits to the system to evaluate efficiently this type of queries.

The **execution engine layer** includes first, the distributed tree pattern query processor of KadoP [5], and second, the Web service invocation infrastructure underlying the AXML platform.

The **high level data access layer** is implemented by an XQuery interface, the AXML engine, OptimAX and TGV [54]. The latter is a XQuery algebraic analyzer which is able to decompose complex XQuery queries to simpler ones that can be processed by the the DHTs and the AXML engine. TGV is called by OptimAX when the query composition/decomposition rule is applied.

2.4.2.3 Processing distributed XQueries over two DHTs

Each of the two DHTs used in the platform has its own interfaces and capabilities. We start by outlining them both, and then we explain how distributed query processing was implemented based on them.

KadoP indexing The first indexing model considered is provided by the KadoP system [5]. In this indexing model all element & attribute names and words (ignoring stop words) that appear in XML files are used as *index keys*. *Index values* are structural identifiers of the form $docID, start, end$ specifying all the locations where each individual key exist in the system. KadoP's tree pattern language consists of trees where each node is labeled with a XML node or word and edges stand for parent-child or ancestor-descendant relationships. When a tree pattern query is received by the system a Holistic Twig Join is performed on the lists identifiers associated to the query node labels. A drawback of the KadoP system is that it can not process queries with inequalities e.g. $//article[year > 2005]$.

PathFinder indexing: The second indexing model considered comes from the PathFinder platform [32] which places index entries in a way that reduces the number of peers contacted (hops) when answering a query. The index keys used are *linear parent-child rooted paths* encountered in the network's XML documents. The *index values* are sorted $docID$ list of identifiers of documents which contain a given path. By *tuning the hash function*, PathFinder ensures that index entries corresponding to *keys* that are close in lexicographic order, are placed on peers that are close to each other (in the sense of proximity in the DHT structure). The biggest advantage of PathFinder is that it can answer inequality queries such as $//article[year > 2005]$.

Query processing: Assume that peer p receives a query q_x expressed in a XQuery dialect. Such queries want results from all the documents published in the WebContent warehouse. Of course there is a naive, semantic driven implementation which would exhibit very poor performance. In our context a query is modeled as an ActiveXML document which contains a call to an abstract service called $WebContentQuery(q)$. OptimAX is aware of this service and it has a rewriting rule that replaces that call with:

$$gqs@p(recomposeQ,\ tpq@any(t1),\ tpq@any(t2),\ ...,\ tpq@any(tn))$$

where $t1, t2$ etc are conjunctive tree pattern queries, tpq is a tree pattern

query processing service, which we will detail in the sequel, and *recomposeQ* is a XQuery query such that, for any set of XML documents indexed in the WebContent warehouse

$$q \equiv recomposeQ(t_1, t_2, \ldots, t_n)$$

In the above, *gqs* stands for the generic query service used in AXML, and which we had introduced in Section 2.1.1. Recall that this service is available on all peers; given an XQuery and some locally available XML inputs, it evaluates the query and returns the XML answer and it is most often used in order to join XML streams that arrive at a specific peer. The interest of decomposing the *gqs* is that the *tpq* service is available on every WebContent peer and it is efficiently implemented by the two available DHT instances on every peer (see Figure 2.28). The notation *tpq@any* means that any concrete endpoint of the *tpq* service can be used. All of them will provide the same answer but with different response times because data transfers during the index look-ups may change when *tpq*s are executed at different peers. It is the task of OptimAX to make the right choice in order to minimize the total query execution time by consulting cardinality statistics maintained in the network or locally cached copies which may reside on the peer.

Putting it all together Given an XQuery query x_q, the TGV algebraic analyzer extracts the recomposition query and a set of tree patterns, which are then analyzed. Depending on their syntax, OptimAX will dispatch them to the KadoP, respectively to the PathFinder index. As an example, consider the query:

```
for $x in //report[//'A320'],
    $y in /airbuslib/report[year>2005]
where $x/author=$y/author
return <res>{$x/title, $y/title}</res>
```

The pattern `//report[//'A320']` will be handled by KadoP, while `/airbuslib/report[year>2005]` will be handled by PathFinder. The recomposition query is:

```
for $i in $res1, $j in $res2
where $i/author=$j/author
return <res>{$i/title, $j/title}</res>
```

Patterns featuring both `//` and inequalities are currently handled to KadoP ignoring the value predicates, which are applied in a post-processing step.

2.5 Experimental analysis

In this Section, we study the efficiency of plan finding using OptimAX. The AXML activation cost reductions due to delegation and instantiation have been shown to reach several orders of magnitude [45, 46]. Clearly, other rules can improve that factor.

Experiments described in this Section have run on a computer with Intel Xeon CPU 5120 @ 1.86GHz, 3GB of Ram and Mandriva Linux 10.

We consider three sets of synthetic parametric documents, consisting exclusively of service calls (except the root). Document *deep-n.xml* consists of n service calls organized in a linear tree of fanout 1. Document *flat-n.xml* consists of a root node and n service call children. Finally, document *tree-n.xml* is an arbitrary tree of n service call nodes where the maximum fanout is $f_{max} = 6$. Each document is characterized by n_d, the number of *distinct* services called. Services assigned randomly (uniform distribution over a set of n_d) to each tree node. Each optimization problem is further characterized by n_p, the number of peers which to which computations may be delegated. In these experiments we consider delegation and factorization.

The graphs in Figure 2.29 and 2.30 depict the time for exhaustive optimization, resp. to obtain a plan with the best cost (as determined by of the exhaustive process).

In the top graph of Figure 2.29, we study the documents *deep-n.xml*, maximizing opportunities for delegation due to the deep nesting of calls. Total optimization time grows exponentially with n, which agrees with the lower bound for the search space size (Section 2.3.1). The bottom graph considers the document *flat-20.xml* while varying n_d; here, factorization applies more and more as n_d lowers.

In Figure 2.30, we use the documents *tree-n.xml*, varying n, with $n_p = 2$ (upper) and $n_p = 3$ (lower). We set n_d to n, resp. $2n$. Since the services are chosen independently for each node, for both values of n_d, some calls are likely to refer to the same service, and some factorization occurs (more for $n_d = n$). The graphs show that the total optimization time grows exponentially with n, and generally grows as n_d decreases, since this often means more factorization opportunities. Moreover, the total time grows from a few seconds, to a few hundred seconds as n_p moves from 2 to 3, also as predicted by the lower bound on the search space size from Section 2.3.1. For applications running for a long time, it may make sense to spend some minutes optimizing, but for others, exhaustive optimization is prohibitive.

The good news shared by all graphs in Figure 2.29 and 2.30 is that *the time to the best solution is very moderate*, of the order of 0.1-0.5 seconds. This is due to our depth-first then cost-based search strategy we apply (to rewrite a plan, we consider the most rewritten ones, and among them, we pick the cheapest one).

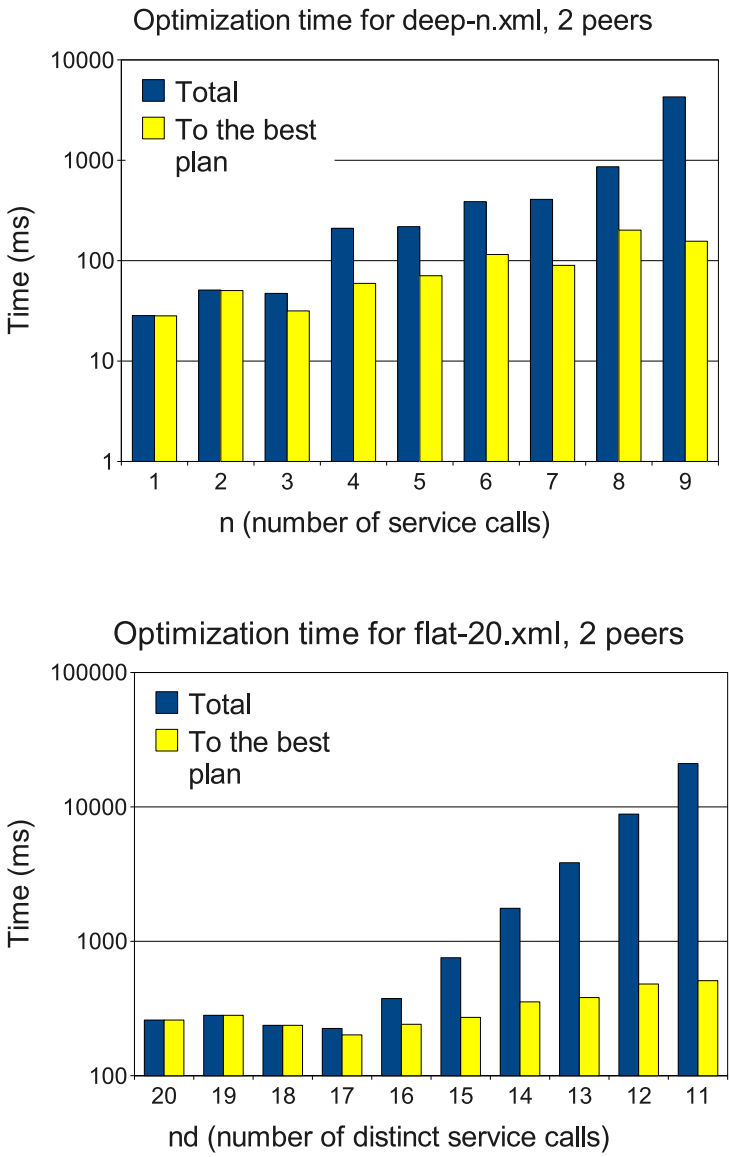


Figure 2.29: Optimization time for *deep* and *flat* documents.

Our next experiment demonstrates the interest of non-exhaustive strategies. For the two graphs in Figure 2.31 at the top, our strategy ran a depth-first, greedy search limited at 200 delegations and/or factorizations. For small n values, this strategy is complete, and the running time (a few ms) is almost identical to that of the full exploration. For larger n values, limiting the search to 200 rewritings marginally increases the cost of the best

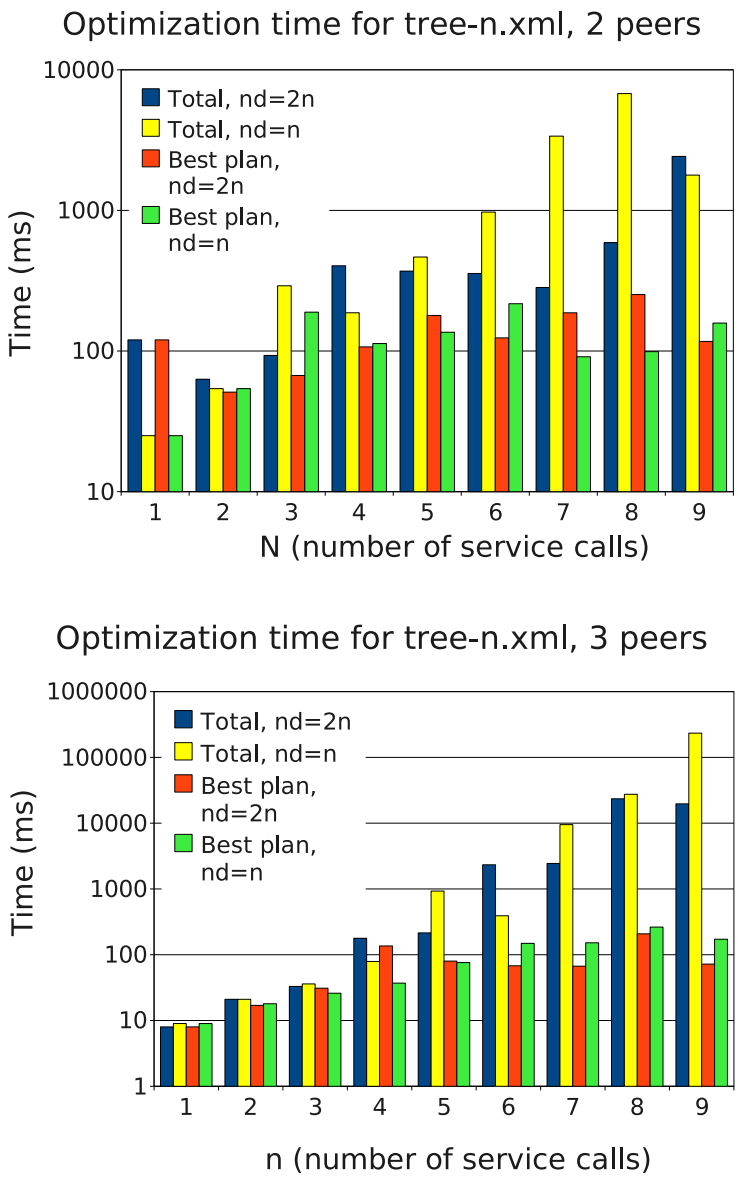


Figure 2.30: Optimization time for *tree* documents.

solution, but decreases the search time by an order of magnitude! At the bottom graphs of Figure 2.31, we used the same plans as at the bottom graph of Figure 2.30, with the following strategy: explore 55 factorizations, *then* 55 delegations. This strategy finds plans within a factor of 2 of the optimum, but may decrease running time by 3 orders of magnitude!

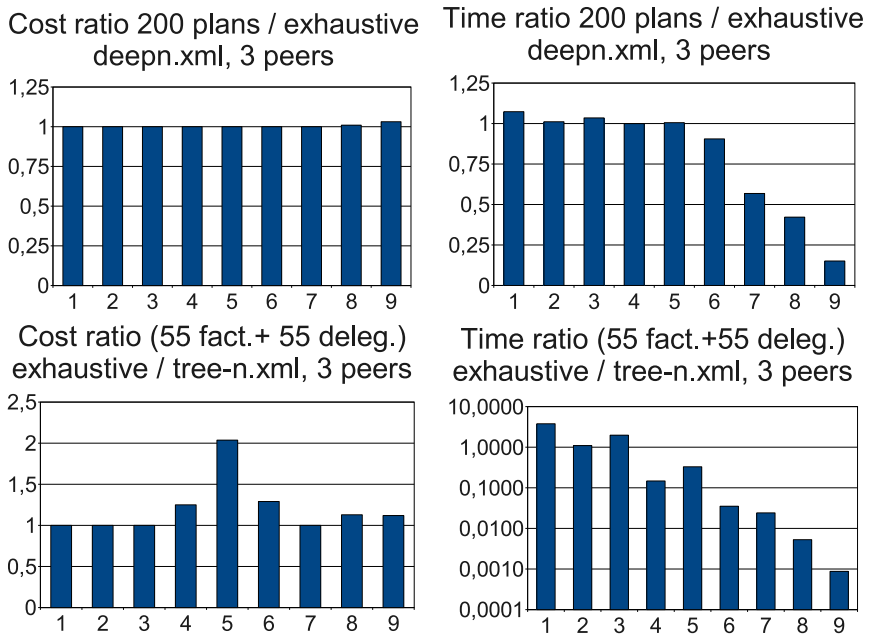


Figure 2.31: Time and cost trade-offs for *deep-n.xml* with non-exhaustive strategies.

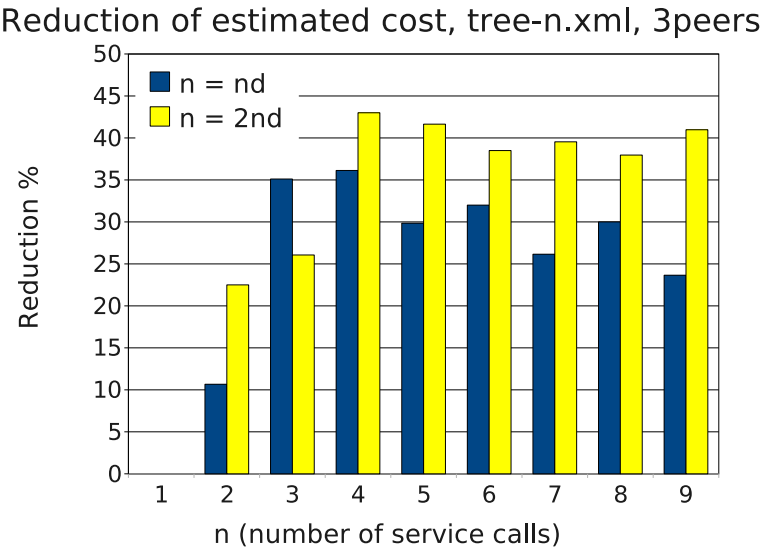


Figure 2.32: Estimated cost reduction for *tree* documents.

In Figure 2.32, we use the documents *tree-n.xml*, varying n, with $n_p = 2$ and $n_p = 3$. In this experiment, we use a depth-first cost driven strategy and we limit the search space size to 3000 explored documents. We measure the cost of a document before and after the optimization and we calculate the cost reduction that we have achieved. As we may observe, the average cost reduction is between 30% and 35%, which is really encouraging.

In conclusion, that while the AXML search space is huge, efficient exhaustive strategies typically find an optimal plan fast. For larger problems, non-exhaustive "smart" strategies critically cut optimization time, while producing near-optimal plans. This demonstrates the practical applicability of our optimizer.

2.6 Software architecture design

OptimAX is a project that has more than 16.000 lines of Java code. In this Section, we discuss the design of the code, which had to reach several goals. The first major goal was extensibility and facility to modify the set of rules. The second goal was smooth integration within the AXML platform. In particular, it has been our goal that OptimAX may work on any implementation of AXML, and that AXML may work with or without OptimAX, since in some very simple AXML applications, optimization is not needed since either all operations are inexpensive or performance is not the main goal. Finally, and crucially, the optimizer itself had to be quite efficient in order to be worth the effort. Observe that while the first goals plead for a clean separation between the optimizer and the AXML platform, the latter is typically achieved by embedding the optimizer tight within the execution engine of a given data management system, with the known drawbacks (difficulty of evolution, as well as over-adaptation to a given set of rules and execution engine).

In this Section, we present the design that has allowed us to conciliate these seemingly conflicting goals. Section 2.6.1 outlines OptimAX' class structure, and the relationships among these classes. We focus on two major class families, the AXML document representation related classes and the rewriting rule related classes. In Section 2.6.2, we address the integration of OptimAX with the AXML peer, and the communication between them.

2.6.1 Inside OptimAX

This section briefly explains the internal code organization of OptimAX. We first consider the internal representation of an AXML document within OptimAX. A proper design at this level was crucial to ensure good performance. We then outline the AXML rewriting rules that have been implemented and experimented with.

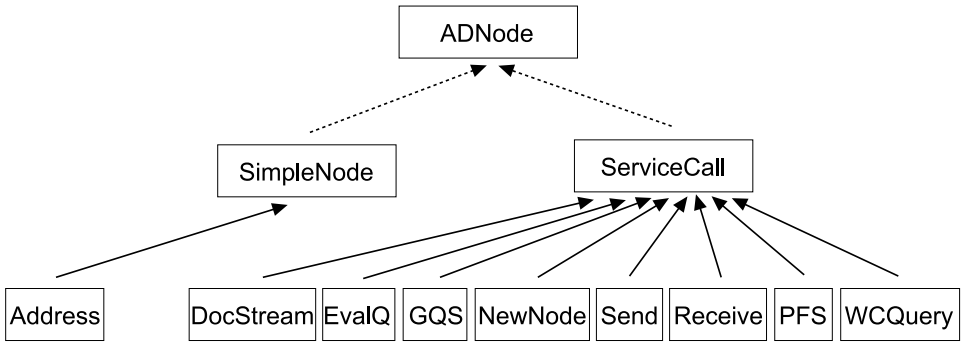


Figure 2.33: UML diagram of the ADNode interface and related classes.

From AXML document to ADocument OptimAX uses AXML documents to communicate with the AXML engine. This data representation is useful for many reasons, such as user readability of the data, but it is not ideal for the optimization process. Plain data, which will not be used during optimization, should not be visible to the optimizer and moreover should not be loaded into memory. The latter is really important since the exploration of the optimization search space, the document rewritings and the resulting documents are done and kept into memory until the end of the optimization procedure. Apart from the document size, there are also other reasons why we have decided to follow a different document representation. More particularly, we need a data representation providing efficient support for:

- applying rewriting rules
- estimating the execution cost of every generated document
- checking document equivalence.

For these reasons, we have decided to adopt an internal representation of an AXML document, implemented by a class called ADocument. Every AXML document can be converted to an ADocument and converted back to the initial AXML document without any data loss. The ADocument retains the needed information for the optimization process and for the rest of the data, keeps XPath pointers to the original AXML document.

Continuing the discussion of ADocument implementation, the UML diagram of Figure 2.33 shows the interfaces and classes that compose an ADocument. Before analyzing each of the classes and their relationship, we should mention that there is no 1-to-1 correspondence between AXML elements and ADocument nodes. This means that, very likely, more than one AXML element is represented by one ADocument node.

ANode is the interface of an ADocument node. It contains all the common functionalities that the ADocument nodes share. This interface is

implemented by two major classes: the SimpleNode class and the ServiceCall class.

Every service call, which may appear in an AXML document, can be represented by an object of the **ServiceCall** class. However, there are some service calls that should be handled differently. There are eight classes which extend the ServiceCall class, and represent the eight specific types of service calls that OptimAX can identify. These classes are:

- **DocStream** objects represent the calls to an AXML service which read a (AXML) documents and return their contents as a stream. The result transmission frequency and the result number per transmission is configurable.
- **EvalQ** objects represent the calls to the KadoP DHT service. This service was discussed in Section 2.4.2.2.
- **GQS** objects represent the calls to the generic query service provided by the AXML engine. This service can evaluate distributive XQueries as described in Section 2.1.1 over continuous input streams, using a fixed size window.
- **NewNode**, **Send** and **Receive** ojects represent calls to the respective *newnode*, *send* and *receive* AXML services, introduced in Section 2.1.3.
- **PFS** objects represent calls to the PathFinder [32] DHT service, previously discussed in Section 2.4.2.2.
- **WCQuery** objects represent $WebContentQuery(q)$ abstract service calls. We say the calls are abstract, since no direct implementation is provided for this service; instead, OptimAX always rewrites such calls into a composition of calls to the *gqs*, EvalQ and PFS services. An example showing how this service is handled by OptimAX has been shown in Section 2.4.2.3.

SimpleNode objects represent all the AXML document nodes that are not represented by a ServiceCall object. The SimpleNode class is extended by the **Address** class. Objects of the last class are used to represent AXML document nodes that reference service calls. Such references are introduced during optimization and are children (parameters) of the *send*, *receive* and *newnode* service calls (Section 2.1.3).

Rules and their details Rule classes are implemented in the lines of ANode classes. The **Rule** interface describes the one and only method, called *applyRule*. All rule classes must implement this method. The six current implementations of this interface are:

- **Composition** and **TGVDecomposer** implement the composition and decomposition rule;
- **Factorization**, **Delegation** and **Instantiation** implement the respective synonymous rules;

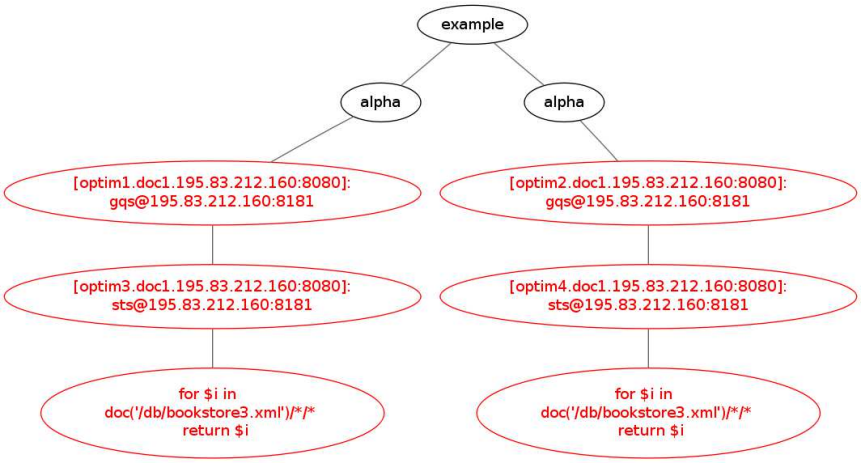


Figure 2.34: Sample AXML document (before optimization) in the GUI.

– **XQuery2EQS** is the specific rule activated when a call to the WebContent query service is identified. As previously explained, such calls are compiled using the two available DHTs of the WebContent peer-to-peer platform.

OptimAX GUI elements Several graphical user interface elements were developed for the OptimAX demo [11] and were helpful in visualizing and making sense of the whole process.

First, a visualization of ADocuments has been realized and provides a quick view of the essential elements in an AXML document. Figure 2.34 exemplifies the display associated to a simple document used in the OptimAX demo. Since two subtrees perform identical computations in this document, a factorization rule applies, and the resulting document is shown in Figure 2.35. This document looks more complex, due to the *send* and *receive* calls introduced by factorization, but its evaluation is likely to be more efficient.

Second, the search space as a whole can be visualized under the form of a state graph. Figure 2.36 exemplifies the small search space that corresponds to the document shown in Figure 2.34. Each numbered node corresponds to one AXML document. The graph has a source, which is the initial document, and several sinks, which are documents to which no further rewriting rule can be applied (without reaching a state that has already been explored). Green-colored transitions represent a reduction in the estimated AXML evaluation cost. Red-color transitions represent an increase in cost. The doubly circled node represents the selected (best) optimization alternative.

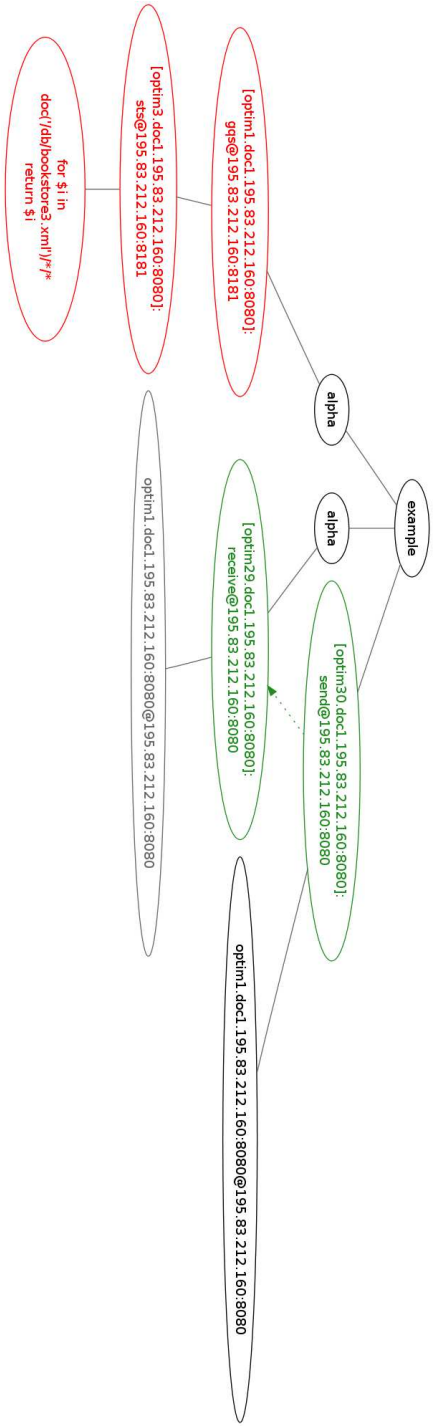


Figure 2.35: The AXML document of Figure 2.34 after factorization.

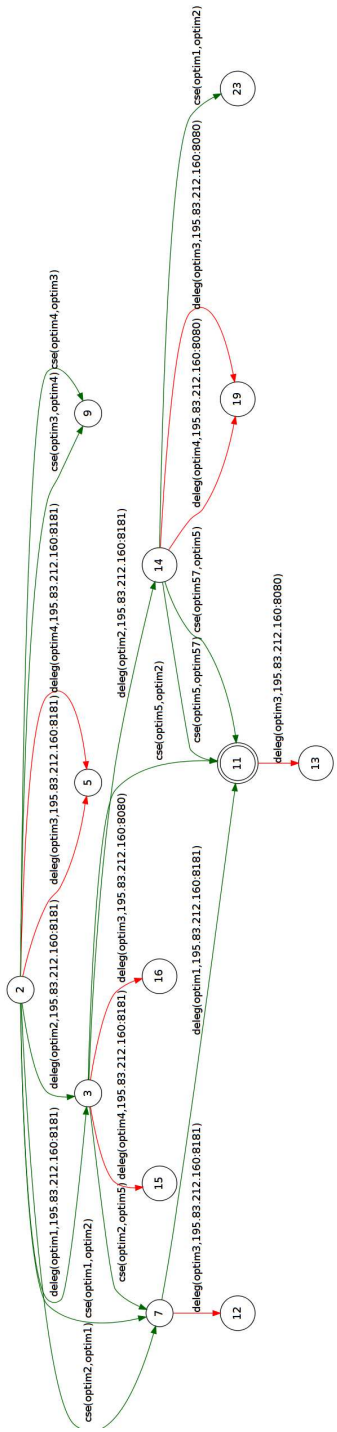


Figure 2.36: The optimization search space for the document of Figure 2.34.

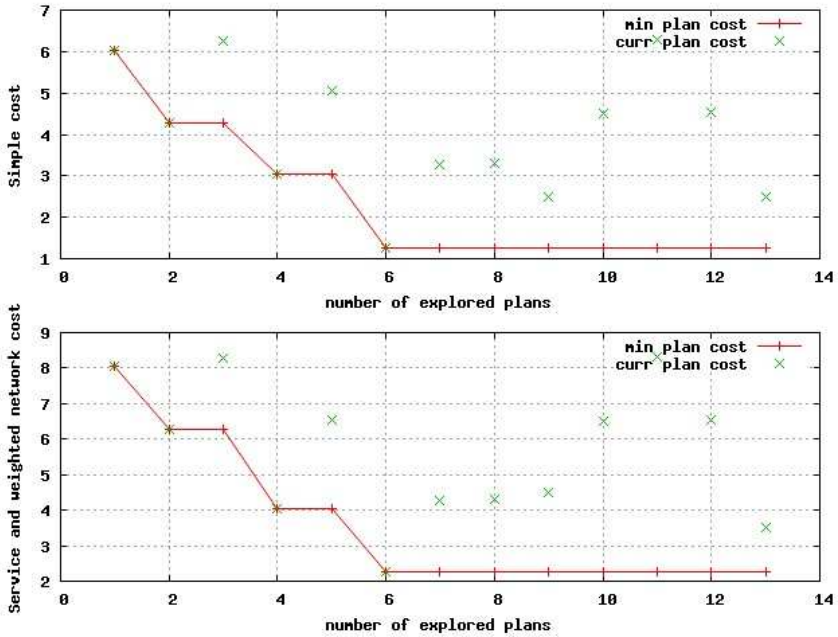


Figure 2.37: Cost reduction as estimated by OptimAX during the optimization of the AXML document of Figure 2.34.

Finally, OptimAX produces graphs showing the variation of the estimated cost of the best alternative found, during the optimization. Figure 2.37 illustrates this. On the x axis, we represent the number of explored rewritings, which naturally increases with the optimization time. The curve in the Figure traces the smallest estimated cost identified so far. The scattered points above the curve show the cost of other states which are explored as the search goes on, but which do not lead to a reduction in the smallest estimated cost.

2.6.2 Integrating OptimAX with the AXML peer

OptimAX is implemented as a service of the second version of the AXML peer [23]. Figure 2.38 represents the interactions between the peer and the optimizer.

The AXML peer is equipped with an XML/XQuery DBMS and it is responsible of storing and managing the (A)XML documents of the peer. When a user requests the evaluation of a document d, the peer will call OptimAX to trigger optimization. The peer provides to OptimAX the document d, the needed statistics and the optimization strategy that should be used. All this information is stored as XML data in the DBMS of the peer. Based on the given data, OptimAX will explore some part of the complete

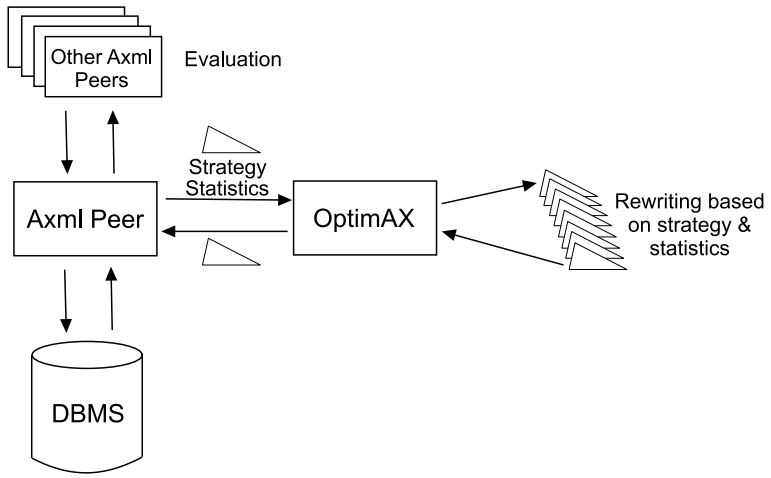


Figure 2.38: Interactions between the AXML peer and OptimAX.

search space, and it will return the document d' with the smallest *estimated evaluation cost*. This document will be evaluated by the AXML peer with the help of other AXML peers, if needed. The evaluation results will arrive to the peer that started the evaluation and will be appended to d'.

Throughout the process, the user is not aware of, and should not be aware of, the existence of d', since he only requested the evaluation of d. Therefore, the AXML peer should update d with the results that arrive at d'. This is ensured by a mapping between the service call node IDs of d and d'. Whenever a result arrives at d', the AXML peer will check in the map if there is a corresponding service call at d. If it is found, then the result is also added as a result of that service call at d.

We make here two interesting observations.

- There may be service calls in d' that do not map to service calls of d. This is very likely because some rewriting rules produce more service calls from a given service call (such as the decomposition rule).
- There may be service calls in d that may never receive results as a result of the evaluation of the optimized document. This may happen for various reasons. For example, a family of service calls in d may have been replaced by one service in d' (composition rule). Another example is when the evaluation of a specific subtree of d is assigned to another peer (delegation rule).

What is ensured throughout optimization, is that the top level service calls of d, the service calls that do not have other service calls as ancestors, will always receive results, if there are any.

2.7 Related works

The starting point of this work is the AXML language [3], which we extended with the *send*, *receive* and *newnode* services and with a flexible yet simple way to control call activation order (Section 2.1). This brings important benefits to users, which may combine continuous and non-continuous services at will. It is also crucial for the efficiency of optimized plans. Consider e.g. the plan $send@p_1(f@p_x, p_2.d_1.\#1)$. Depending on whether p_x is p_1, p_2 or another peer, we may wish to activate *send* first (thus, push the computation to p_x) or f first (thus, call f from p_1). Our previous algebraic proposal [7] was unable to express both - a shortcoming we detected while implementing OptimAX. We defined valid schedules and formalized the optimization problem accordingly.

A language for AXML replication and some XPath execution strategies were introduced in [4], which does not address optimization. The full optimization problem is solved in [2] in the particular case when the only rule is useless call elimination (Section 2.3). Delegation and instantiation have first been proposed in [46] in isolation and in an ad-hoc way which could not be generalized. In [7] we proposed an abstract algebra (with the shortcomings mentioned above) but did not discuss how it can be mapped into an implementable language. Finding which calls to activate to bring a document to a given type is shown to be sometimes undecidable in [14, 42] which do not consider optimization. Continuous services are used in [12] to specify monitoring programs, but algebraic optimization is not considered. XCraft [45] is an optimizer for non-continuous AXML. It uses a work-flow model for AXML documents, which are split in pieces of fixed size optimized independently. In contrast, OptimAX, based on tree rewritings, is intimately connected with the AXML model, which allows it to include many more rules, e.g. factorization, query composition/decomposition etc., giving it more generality. Moreover, we explore many strategies, and we show that a greedy-based depth-first can quickly identify efficient plans, more reliably than a fixed-size divide and conquer approach.

Web service orchestration in work-flow style, e.g. via BPEL4WS, is a very active area [16]. While AXML with ordering constraints has some work-flow flavor, it trades many BPEL aspects (complex processes, exception handling etc.) for a data-centric character that enables specific data-oriented efficient optimizations. More generally, one can use either AXML or BPEL4WS to specify relatively simple work-flows; we have experimented with a simple translation tool from one to the other. OptimAX functions in the realm of AXML documents, which we found more convenient to handle via rewriting than process specifications.

The work presented in this Chapter follows previous works on distributed query processing [37, 55], and in particular in the context of mediator systems [35].

2.8 Conclusion

In this Chapter, we have described the AXML language and the extensions that we bring to it. These extensions permit us to formalize the optimization problem and to propose OptimAX, our approach in solving it. In this Chapter, we have also seen how OptimAX can be used in real life - demanding applications. We have thoroughly examined two case studies.

In the first case study, we have examined how OptimAX could have solved optimization problems encountered in the EDOS EU project. The proposed solution may be theoretical but it shows the optimization opportunities that OptimAX offers.

The second case study shows how our optimizer was used into the WebContent project. Several research laboratories, universities and industrial partners have collaborated in order to develop this platform. OptimAX participation to this project, part of which was demonstrated at the VLDB 2008 conference [1], shows OptimAX' importance and capability to perform in a wide variety of environments.

We have also experimented using OptimAX and have seen how it decreases the estimated execution cost. At the end of this Chapter we have classified and compared our work with previous related AXML works and with the state of the art.

Chapter 3

ViP2P - Views in peer-to-peer

We now move to describe a second main contribution of this thesis, namely, the ViP2P (standing for *Vi*ews in *P*eer to *P*eer) platform. We have devised this platform inspired by the class of distributed data management applications that we encountered in the WebContent project, outlined in Section 2.4.2.

The common characteristic of the applications we target is the need for efficient techniques for sharing large volumes of XML content in structured P2P networks organized on DHTs. As we have previously explained, in WebContent, two XML content indexing platforms are integrated: KadoP [5], and PathFinder [32]. The former provides full-text indexing of XML structure and keywords, while the latter is able to handle linear path queries including inequalities on particular data values. Such XML indices enabled the processing of relatively complex XML queries on the overall set of documents published in the network. However, the indexing model is fixed by each of the individual indexing platforms, and cannot be adapted to the particular needs of an application. For instance, should a specific application frequently ask a tree pattern query involving n nodes, assuming it can be processed via KadoP, n posting lists will always require joining in order to compute the result, since KadoP does not allow the definition of custom indices (or materialized views) suited to a specific application. (The same holds for PathFinder or for similar DHT-based XML content management platforms, such as [26, 33] etc.)

In ViP2P, we set out to build a flexible, generic system for DHT-based XML content management. ViP2P can be seen as a tool for *redistributing restructured data where it is needed*. Any ViP2P site (or peer) may establish some materialized views, which reflect data published anywhere in the network, that the peer is interested in. A more likely scenario is that several peers which are physically close (e.g. machines in the same company site) share the burden of storing some views which may be interesting to all of them. All view definitions are then indexed in the DHT, so that any peer

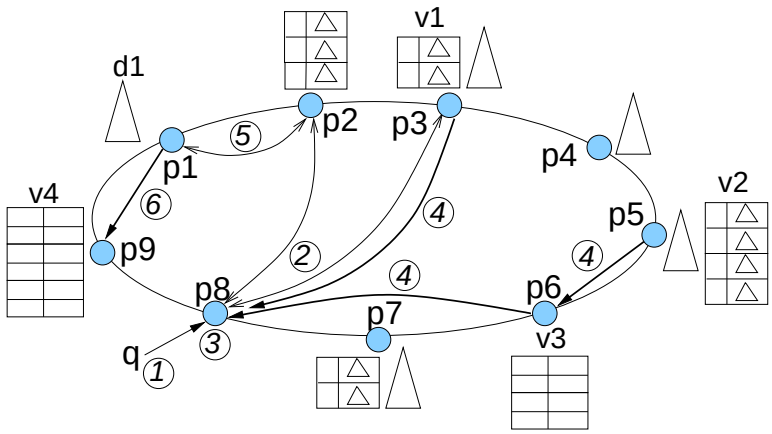


Figure 3.1: Architecture overview.

may learn about them. A query posed on any peer is re-written using the existing views. In this work, we focus on the problem of finding equivalent query rewritings based on the views in the DHT, as well as on building and advertising the views.

This Chapter makes the following original contributions. (1) We describe the VIP2P architecture for managing and exploiting materialized XML views based on a DHT. (2) We consider the problem of tree pattern query rewriting problem based on multiple views. The sets of rewritings identified by our and similar works [19, 29, 51] partially overlap; we prove an interesting bound on the maximal rewriting size, making it polynomial in the number of views, and we study several corresponding rewriting algorithms. (3) We study several strategies for indexing materialized view definitions on a DHT, and compare their usefulness. (4) We demonstrate via experiments in a fully implemented platform the scalability of our platform.

An early version of the work presented in this chapter has been informally presented in an international workshop (not in the proceedings) [40]. The complexity analysis and experiments presented in this chapter have been obtained subsequently, and accepted for presentation at the informal French database conference BDA 2009 [39], and are part of an ongoing submission. A continuation of this work (not described in this thesis) is a demonstration to appear in ICDE 2010 [36].

The architecture of ViP2P is depicted in Figure 3.1. Network peers labeled p_1 to p_9 store documents, shown as triangles, and/or views, shown as tables. Such tables attributes may be of type *xml* (whose values are serialized XML subtrees), in the style of the native XML data type in SQL/XML 2003. Such

attributes are shown as triangles inside tuples. We designate a document d or view v at peer p by the notation $d@p$, respectively, $v@p$.

Each view is defined by a tree pattern, and this pattern (not the view extent) is indexed in the DHT. Query processing can be traced following the numbered arrows in the Figure. Assume query q is asked at peer p_8 (step 1). Then, p_8 will perform a DHT look-up to find which view definitions may be useful to rewrite the query. For instance, p_2 and p_3 may return to p_8 relevant view definitions (step 2). Peer p_8 will then run a view-based query rewriting algorithm, trying to reformulate q based on these definitions (step 3). A query rewriting is a logical algebraic plan based on some views, in our example, $v_1@p_3$, $v_2@p_5$, and $v_3@p_6$. After picking a rewriting, p_8 transforms it into a distributed physical plan, which runs in a distributed fashion (step 4, thick arrows denote data transfer). In our example, v_2 is sent to p_6 which joins it with v_3 and sends the result to p_8. At p_8 it joins with v_1 which is sent from p_3.

Each ViP2P view v is complete, i.e. it includes $v(d)$ for any document d in the network (modulo some update propagation time). To obtain such views, whenever a new document, say $d_1@p_1$ in Figure 3.1, is published, the publishing peer performs another type of lookup (step 5) to determine (possibly a superset of) the view definitions to which the new document may contribute tuples. In the Figure 3.1, such definitions are returned by p_2, and p_1 finds out that d_1 contributes some tuples to the view $v_4@p_9$. The tuples are sent to p_9 (step 6), which adds them to the view extent.

The remainder of this Chapter is organized as follows. Our pattern language is discussed in Section 3.1. Section 3.2 presents the rewriting problem and its complexity, and Section 3.3 studies several rewriting algorithms. How views are materialized, indexed in the P2P network, and looked up is discussed in Section 3.4. We present our experiments in Section 3.5, discuss code organization and its main modules in Section 3.6, compare our work with the state of the art in Section 3.7, and then conclude.

3.1 Patterns

We will rely on a tree pattern dialect $\mathcal{P}$, defined as follows.

1. Pattern nodes can correspond to *XML internal nodes* (elements or attributes), or to *leaves* (words in text occurring inside XML elements, or in attribute values). For presentation purposes, we do not distinguish between elements and attributes. We extend the XPath descendant axis to consider that words are children of their closest element or attribute ancestors. Observe that we allow a simple word to make up a pattern node, which corresponds to the importance that keyword searches play in our context. Each internal pattern node carries a la-

bel from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.

2. Pattern edges correspond to parent-child or ancestor-descendant relationships between nodes.
3. Each pattern node may be annotated with some *stored attributes*, describing some information items that the pattern stores out of each XML node matching the pattern node. The *cont* annotation indicates that the full (serialized) image each matching XML tree node is stored. The *id* annotation indicates that a node identifier, which uniquely identifies the node (and the document it belongs to). Moreover, we assume *structural* IDs, i.e. such that one may decide, by comparing the identifiers of two nodes n_1 and n_2, whether n_1 is an ancestor/parent of n_2 or not. Many variants of structural identifiers exist, e.g., [15, 38, 53], some of which provide further information, e.g. allow identifying the least common ancestor of two nodes etc. *For the purpose of this work, we only require that parent-child and ancestor-descendant relationships can be determined from the node IDs.* Finally, the *val* labels stands for the node's text value, obtained by concatenating all its text descendants in document order.
4. Each node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$, restricting the XML nodes which match the pattern node, to those satisfying the predicate.

Notations and syntax simplification We say a pattern node *has* an *id*, respectively *val*, *cont*, or value predicate, if the node is decorated with such an index.

In this thesis, we only consider ancestor-descendant edges between tree pattern nodes. Extending this to also support parent-child edges is left for future work.

We introduce a simple text syntax for patterns. We denote nodes by their A_l or A_w label. The possible *id*, *val* and *cont* labels, and predicates over *val*, are shown as indices to the node. For instance, $a_{id\,cont}$ is a pattern storing the structural IDs and the content of all a elements. We use parenthesis to show the nesting of children inside parents, and commas to separate the children of the same pattern node among themselves. For instance, $a(b(c_{id}))$ stores the IDs of elements found on some path matching $//a//b//c$. The pattern $a_{[val=5]}(\underline{b}, c_{id})$ stores the identifiers of all c elements having an a ancestor of value 5, and whose serialized XML subtree contains the word $\underline{b}$.

Pattern semantics Let p be a pattern and d be an XML document. As customary, an embedding $\phi : p \rightarrow d$ of p in d is a function associating d nodes to p nodes, preserving node labels and ancestor-descendant relationships [17]. The result of evaluating p on d, denoted $p(d)$, is the list of tuples obtained

by lining together in a tuple, all IDs and/or values and/or serialized content, for each embedding of p in d. Assuming a total order over the nodes of p (top-down, left-to-right traversal), the tuple order in $p(d)$ is dictated by the lexicographic order over the d nodes which are targets of the embeddings. For a document set $\mathcal{D}$, the semantics of p over $\mathcal{D}$ is defined as the concatenation (in the order of the document IDs) of all $p(d)$, $d \in \mathcal{D}$.

We use $a.id$ (respectively, $a.val$, $a.cont$) to denote the corresponding attribute in $p(\mathcal{D})$.

We say two patterns p_1, p_2 are *equivalent*, denoted $p_1 \equiv p_2$, if for any database $\mathcal{D}$, $p_1(\mathcal{D}) = p_2(\mathcal{D})$. We establish containment and equivalence of $\mathcal{P}$ patterns in time polynomial in the size of the patterns [17].

3.2 Algebraic rewritings using patterns

Given a query $q \in \mathcal{P}$ and a set $\mathcal{V}$ of views, we are interested in the rewritings of q, based on $\mathcal{V}$. As explained in Section 3.1, the semantics of both queries and views are *relations*, therefore, we investigate rewritings which combine views by means of a relational algebra, specified in Section 3.2.1. Based on this, Section 3.2.2 formally states the rewriting problem, while Section 3.2.3 show that its complexity is polynomial in the number of views.

3.2.1 Algebra

The algebra we consider consists of the following operators:

1. $scan(v)$ (or v, in short), where $v \in \mathcal{V}$ is a view.
2. The n-ary cartesian product operator $\times$, projection (denoted π_{cols}), duplicate elimination (denoted π^o), and sort (denoted s_{cols}).
3. Selection, denoted σ_{pred}. Here, $pred$ is a conjunction of predicates of the form $i \odot \underline{c}$ or $i \odot j$, where i, j are attribute names, $\underline{c} \in A_w$, and $\odot \in \{=, \prec\}$ is a binary operator. We use $\prec$ to designate the "is ancestor of" predicate. Thus, assuming the attributes named i and j contain IDs, $\sigma_{i \prec j}(op)$ returns those op tuples where the identifier in attribute i corresponds to an ancestor of the node whose identifier is in attribute j.
   Note that the presence of $\times$ and σ allows, in particular, ID equality joins $\bowtie_=$, as well as structural joins [15], denoted $\bowtie_\prec$.
4. A navigation operator, designated $nav_{i,np}$, which takes as input one algebraic expression. The attribute i in the input must correspond to a $cont$ attribute, and np is a pattern whose nodes must not have ids. Let t be an input tuple to the nav, and $np(t.i)$ denote the result of evaluating the pattern np on the XML fragment stored in $t.i$ (as defined in Section 3.1). Then, $nav_{i,np}$ will output the tuples $t \times np(t.i)$, i.e., obtained by successively appending to t each of the tuples in $np(t.i)$.

If $np(t.i)$ is empty, $nav_{i,np}$ acts like a selection, erasing t. The reason why np nodes must not have ids is that it is generally not possible to determine the ID of a node, from an XML fragment (not the whole document) to which the node belongs.

As an example, let v be the view $a_{id\ cont}$. The expression

$$e = nav_{a.cont,b(c_{cont},d_{cont})}(v)$$

returns tuples with 4 attributes: a identifiers, a contents, and contents of c and d descendants of a, having a common b ancestor below a.

For convenience, we extend the notation to allow several patterns to be applied on the same *cont* attribute by a single *nav* operator. Thus, $nav_{i,np_1,np_2}(op) = nav_{i,np_2}(nav_{i,np_1}(op))$.

3.2.2 Problem statement

Equivalent rewritings Let q be a $\mathcal{P}$ query, and $e(v_1, v_2, \ldots, v_k)$ be an algebraic expression over the views in $\mathcal{V}$. We say $e(\mathcal{V})$ is an equivalent rewriting of q if and only if, for any database $\mathcal{D}$, $e(v_1(\mathcal{D}), v_2(\mathcal{D}), \ldots, v_k(\mathcal{D})) = q(\mathcal{D})$.

As an example, the expression e from the last example above is an equivalent rewriting for the query $q = a_{id,cont}(b(c_{cont}, d_{cont}))$.

Problem statement (first attempt) We may at this point specify our problem as: given q and $\mathcal{V}$, find all equivalent rewritings of q using the views $\mathcal{V}$. Here and in the sequel, we assume the views and the query have been minimized as in [17] (a difference to be made for our patterns with attributes is: no node having *id*, *cont* or *val* can be removed by minimization).

However, this problem definition leads to an artificially large space of solutions, since two algebraic expressions may differ in their view join orders, selection and projection positions etc., all the while corresponding to the same rewriting. For instance, let $q = a_{id}(b, c, d)$ and $v_b = a_{id}(b)$, $v_c = a_{id}(c)$, $v_d = a_{id}(d)$. Twelve syntactically different join expressions over v_b, v_c and v_d are equivalent rewritings of q. We are not interested in exploring these alternatives during rewriting, as this exploration pertains to the subsequent algebraic optimization step. To that effect, we introduce the notion of *canonical algebraic expressions*. An algebraic expression e is said to be canonical if it has one of the following forms:

- form 1: $scan(v)$ or $nav(scan(v))$
- form 2: $\times(\alpha_1, \ldots, \alpha_k)$, where each α_i is of form 1
- form 3: $\sigma_{pred}(\beta)$, where β is of form 1 or 2
- form 4: $s_{cols}(\gamma)$, where γ is of form 1, 2 or 3
- form 5: $\pi_{cols}(\delta)$, where δ is of form 1, 2, 3 or 4
- form 6: $\pi^0(\epsilon)$, where ϵ is of form 1, 2, 3, 4 or 5.

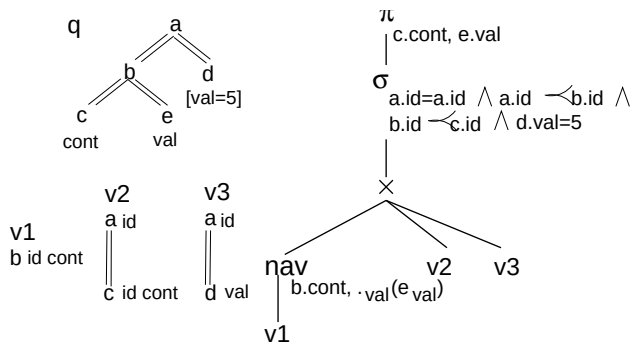


Figure 3.2: Sample query, views, and rewriting.

Intuitively, the operators in canonical expressions are (1) consolidated - there will be at most one of each of the following operators: cartesian product, selection (possibly on a conjunction of predicates), sort, projection, and duplicate elimination and (2) applied in a specific order (*scan*, then *nav*, then $\times$, σ, s, π and π^0 respectively).

Any algebraic expression can be brought to a canonical form. We say e is a *canonical rewriting* of q if e is a rewriting of q, and e is a canonical expression.

Minimal rewritings Certain canonical rewritings exhibit some redundancy, as illustrated by the query $q = a_{id}$ and identical views $v = v' = a_{id}$. Then, $e_1 = v$ and $e_2 = v'$ are canonical rewritings, but so is $e_3 = \pi_{id}(\sigma_{id=id}(v \times v'))$. Intuitively, we are interested in finding e_1 and e_2, but not e_3. More formally, let e be an algebraic expression. We say e is *minimal* if and only if all the expressions obtained by removing a view from e are not equivalent to e. Several minimal canonical rewritings can be obtained from a non-minimal one, as shown in the last example above.

View pruning If v appears in a rewriting of q, then there exists an embedding $\phi : v \rightarrow q$, such that:

1. ϕ preserves node names
2. if n is a parent of m in v, $\phi(n)$ is an ancestor of $\phi(m)$
3. if m has a value predicate $[val = c_1]$ in q and $\phi(n) = m$, for some v node m, then m must not have a value predicate $[val = c_2]$, if $c_1 \neq c_2$.

A similar observation has been made in [51] (excluding item 3 above). This observation allows pruning down the set of views $\mathcal{V}$ to a subset $\mathcal{U}$ of views which can be embedded in q, while being guaranteed not to lose any rewritings by doing so.

View expansion An important refinement of our problem is needed. From the *cont* attribute of a view node, one may extract the value of this node's *val* attribute (e.g. by the XPath query $./text()$), as well as information about its descendants, since they appear in the *cont* XML subtree. For instance, in Figure 3.2, one can expand v_1 by navigating within *b.cont* to find its text value, and the text values of all its *e* descendants. This is represented by the algebraic plan $nav_{b.cont, \cdot_{val}(e_{val})}(v_1)$, where by a slight abuse of syntax, we denoted by $\cdot_{val}$ a pattern node matching only the root of the XML tree on which it is evaluated, and having a *val* attribute. The result can be seen as an expanded view $v_1' = b_{id,val,cont}(e_{val})$. The algebraic expression at right in Figure 3.2 builds on this plan, and is a canonical minimal rewriting for the query.

The need for partial expansions Observe that an expanded view does not necessarily add under the view node having *cont*, all the forest rooted at the corresponding query node. In the above example, expansion added an *e* node but not a *c* node. Indeed, had we added the *c* node too, it would have been impossible to rewrite *q*. Let us see why. Assume expansion transforms v_1 into $v_1'' = b_{id,val,cont}(c_{cont}, e_{val})$. We may join v_1'' with v_2 enforcing that *a.id* is an ancestor of *b.id* and *b.id* is an ancestor of *c.id* (the latter from v_2). The resulting expression has two *c* nodes, descendants of *b*: the one from v_1'' has *cont*, while the other has *id* and *cont*. As a result, this expression contains a cartesian product of the $//b//c$ nodes with themselves, which was not required by the query. We cannot unify the two *c* nodes, as the one from the expanded view v_1'' does not have *id*. Thus, it is impossible to rewrite *q* based on v_1''; v_1' is necessary. This phenomenon is due to the fact that unlike XPath views used e.g. in [51], our views may store data from more than one nodes.

We consider the views in $\mathcal{U}$ have all been expanded into a set $\mathcal{W}$, and reason on $\mathcal{W}$ from now on.

Problem statement (final) Our problem can be now stated as follows: given a query *q* and a set of views $\mathcal{V}$, find all minimal canonical rewritings of *q* using views from the set $\mathcal{W}$, obtained by pruning, and then expanding, $\mathcal{V}$.

3.2.3 Complexity

Several aspects impact rewriting complexity.

View pruning is performed by evaluating each view on the query, considered as a data tree; if the result is non-empty, an embedding has been found, and the view is kept. Thus, the cost of obtaining the set $\mathcal{U}$ from $\mathcal{V}$ is $\Theta(|q| \times \Sigma_{v \in \mathcal{V}} |v|)$.

All views which survive pruning have at most as many nodes as the query (recall also that they are minimized from the start). Thus, for any $v \in \mathcal{U}$, $|v| < |q|$.

Expansion impact We consider the size of the set $\mathcal{W}$ obtained by expanding $\mathcal{U}$ views.

Let v be a view in $\mathcal{U}$, where a single node m has a *cont*. Assume an embedding ϕ maps m to the query node n. In principle, we should generate $2^{|n|} - 1$ copies of v (where $|n|$ is the size of the query tree rooted at n), each of which copies as a new child of m, a subtree of the n-rooted query tree. If a node in this subtree has an *id*, the *id* will be erased in the copy, since as said in Section 3.2.1, IDs cannot be found inside *cont* attributes.

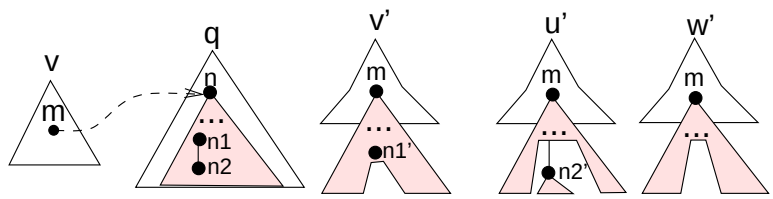


Figure 3.3: View expansion.

Three observations allow to reduce this set (see Figure 3.3):

1. Let n_1 be a descendant of n, and n_2 be a child of n_1. Let v' be an expansion of v, in which n_1 is copied as n'_1, whereas n_2 is not copied. To build a rewriting based on v', we would need some other view v_x covering node n_2, and a predicate of the form $n'_1.id \prec n_2.id$ enforcing the appropriate relationship between the two nodes. However, n'_1 lacks an ID, thus this predicate cannot be checked, and v' is useless. Thus, we only develop the expanded views such that: if a descendant n_1 of n is copied, so are its children, but also their children etc. - thus, the full subtree rooted at n_1 must be copied.
2. Again, let n_1 be a descendant of n, and n_2 a child of n_1. This time, we consider an expanded view u' where n_2 is copied as n'_2, and n_1 is not copied. To build a rewriting based on u', we would need another view covering n_1, and a predicate of the form $n_1.id \prec n'_2.id$ enforcing the appropriate relationship among the two nodes. Again, this predicate cannot be checked since n'_2 lacks an ID. Therefore, if we copy n_2, all its ancestors up to n must be copied.
3. In the case when m does not have an *id*, let n_1 be a descendant of the query node n, and w' be an expanded view such that n_1 is not copied in w. By a similar argument as above, a rewriting based on w' needs another view covering node n_1, and a predicate over $n_1.id$ ensuring

that it is a descendant of m's copy in w'. Since the latter node does not have an ID in v nor w', no rewriting can use w'.

Observations 1. and 2. entail that we only need to enumerate the subsets of n children, and for each subset, build an expanded pattern which fully copies the subtrees rooted in those children. This reduces the number of generated expanded views from $2^{|n|}$ to $2^{f(n)}$ (where $f(n)$ is the fan-out of n), which is often smaller. Observation 3. further reduces it to 1 in the particular case where m does not have an ID.

More generally, let $f(q)$ be the maximum fan-out of a *cont* node in q, $f(q) \leq |q|$. Let v_i be a view whose nodes $m_i^1, m_i^2, \ldots, m_i^{k_i}$ have *cont*, $k_i \leq |v_i| \leq |q|$, and ϕ_i an embedding from v_i to q such that $\phi_i(m_i^j) = n_j$, $1 \leq j \leq k_i$. The expansion of v_i produces $\Pi_{j=1,\ldots,k_i}(2^{f(n_j)})$ views, which is bounded by $\Pi_{j=1,\ldots,k_i} 2^{f(q)} \leq 2^{|q| \times f(q)}$.

Thus, $|\mathcal{W}| \leq |\mathcal{V}| \times 2^{|q| \times f(q)} \leq |\mathcal{V}| \times 2^{|q|^2}$.

Rewritings and covers Let e be a canonical rewriting based on some a subset V of $\mathcal{W}$. Clearly, the set of q nodes can be seen as covered by the union of the node sets of the view involved in e. Thus, from a rewriting, one can extract a *query cover* based on the view nodes.

Not any query cover leads to a rewriting. For instance, consider the views $v_1 = a_{id}(b)$ and $v_2 = c_{id}$, and the query $q_1 = a_{id}(b(c))$. In this case, the query requires the b node to be an ancestor of the c node, but since v_1 does not store identifiers for b, we are unable to enforce this constraint.

A cover may use a view several times, and in distinct positions. Consider, for instance, $q_2 = a(a_{id})$, and the views $v_3 = v_4 = a_{id}$; q_2 may be covered (and rewritten) by using: v_4 twice, or v_3 twice, or v_3 in the ancestor role and v_4 in the descendant role, or the opposite.

More generally, *to each set of pairs of the form (view from $\mathcal{W}$, embedding from the view to the query), where distinct pairs may use the same view with different embeddings, corresponds at most one rewriting.* We will describe an algorithm which builds this rewriting when possible in Section 3.3; the algorithm runs in quadratic time in the combined size of the views.

How many different embeddings exist from a view to the query? If all query nodes have different labels, then this also holds for each view, and at most one embedding exists. In general, let $\nu(q)$ be the maximum number of times a given node label appears in the query. Then, the view can be embedded in at most $\nu(q)^{|v|} \leq \nu(q)^{|q|}$ ways.

Rewriting size bound As we will show in Section 3.3, the maximum number of views involved in a minimal canonical rewriting is equal to the number of query nodes.

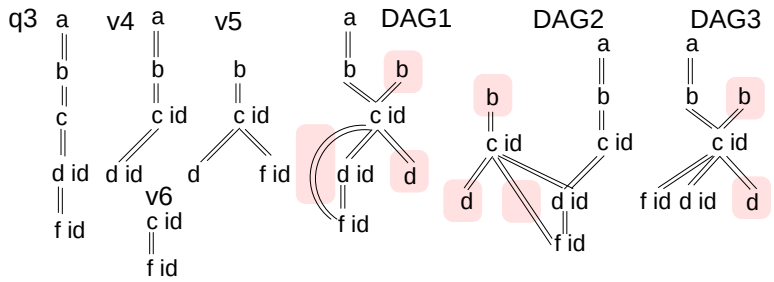


Figure 3.4: Sample query, views, and DAGs.

Putting it all together, the worst-case complexity for enumerating all minimal canonical rewritings is of the form $|q| \times (\Sigma_{v \in \mathcal{V}} |v|) + (\Sigma_{k=1,\ldots,|q|} C^k_{|\mathcal{W}| \times \nu(q)^{|q|}}) \times (\Sigma_{v \in \mathcal{W}} |v|)^2$. The first term accounts for pruning. The second is the cost of enumerating all subsets of (view from $\mathcal{W}$, embedding) pairs, of size at most $|q|$ (by a known formula, this is in $O((|\mathcal{W}| \times \nu(q)^{|q|})^{|q|}/|q|!)$), multiplied by the quadratic cost of building a rewriting out of such a set. More simply, the sum of the combinations can be put as $O(|\mathcal{W}|^{|q|})$ which translates to $O(|\mathcal{V}|^{|q|})$. The last factor $(\Sigma_{v \in \mathcal{W}} |v|)^2$ is less than $(|\mathcal{W}| \times |q|)^2$, thus in $O(|\mathcal{V}|)^2$. Thus, the total cost is in $O(|\mathcal{V}|)^{|q|+2}$. This is significantly less than the $O(2^{|\mathcal{V}|})$ which can be attained with a naïve set-cover approach.

3.3 Rewriting-based query answering

In this Section, we describe how queries can be answered in our architecture, based on materialized tree pattern views. Section 3.3.1 discusses if and how an algebraic rewriting can be built out of a set of views. Section 3.3.2 presents algorithms for enumerating all minimal canonical rewritings. Section 3.3.3 outlines the optimization and execution of our rewriting plans.

3.3.1 Building a rewriting out of a set of views

As we have seen, rewritings can be obtained out of some, but not all, covers of the query using the views. We present an algorithm (Algorithm 1) which, given the query, a subset of views in $\mathcal{W}$, and embeddings from each view into the query, builds a particular algebraic expression over all the views, or fails. In particular, if the views constitute a node cover, and if an expression is returned, it will be a canonical rewriting of q, using all the views. This rewriting is not guaranteed to be minimal; we will address minimality further on.

Algorithm 1 starts by building the cartesian product of the views. Then, it adds a selection on two kinds of predicates (lines 2-9). First, all the view

Algorithm 1: Views-to-rewriting (possibly partial)

Input : query q, views $v_1, \ldots, v_k \in \mathcal{W}$,
embeddings $\phi_i : v_i \to q$, $1 \leq i \leq k$
Output: partial rewriting of q using $v_1, \ldots, v_k$, if one exists

1 $e \leftarrow \times(v_1, v_2, \ldots, v_k)$
2 **foreach** *query node* n **do**
3 $\quad S(n) \leftarrow \{n_i \in v_i$ such that $\phi_i(n_i) = n$ and n_i has an $id\}$
4 $pred \leftarrow$ true
5 **foreach** *query node* n **do**
6 $\quad pred \leftarrow pred \wedge \bigwedge_{n_i, n_j \in S(n)} (n_i.id = n_j.id)$
7 $\quad$ **foreach** m *child of* n **do**
8 $\quad\quad pred \leftarrow pred \wedge \bigwedge_{n_i \in S(n), m_j \in S(m)} (n_i.id \prec m_j.id)$
9 $e \leftarrow \sigma_{pred}(e)$;
10 $d \leftarrow DAG(e)$; $d \leftarrow minimize(d)$
11 **if** *d is not a tree, or d cannot be embedded in q* **then**
12 $\quad$ fail
13 attemptFinalize(e, q)
14 return e

nodes having an *id*, and embedded in the same query node, must be equal. Second, structural relationships between query nodes should be enforced over the corresponding view nodes. As an example, consider the query q_3 and the views v_4 and v_5 in Figure 3.4. They lead to the expression $e_1 = \sigma_{c.id=c.id \wedge c.id \prec d.id \wedge d.id \prec f.id}(v_4 \times v_5)$, where in the second predicate, c is from v_5 and d is from v_4. Observe that no predicate connects the b nodes, since they do not have *id*s.

At line 10 in Algorithm 1, we examine the resulting expression by means of a DAG representation, built as follows. Take all the v_i trees, and fuse all view nodes mapped to a same query node, into one node. (The result is guaranteed to be a DAG, since the embeddings ϕ_i map the v_i trees to q.) We then minimize the DAG, by removing all redundant nodes, and redundant edges. A node is redundant if it has no *id*, *val* or *cont* attribute, and is not the only one mapped to its corresponding query node. An edge is redundant if there exists a path in the DAG connecting the same nodes. In the example of Figure 3.4, DAG_1 is the DAG obtained from e_1; the nodes and edge on gray background are eliminated by minimization.

The function **attemptFinalize** attempts to build a q rewriting out of e. It first tests if an embedding from q to d can be found; if yes, this also implies that the views form a query cover. If so, it will attempt to add the value selection predicates from q that e does not have, a sort, a projection, and/or a duplicate elimination on top of e. The sort addition is

a bit complex, since the order in which e produces results depends on the physical operators implementing it, and these operators are not known at this point. Thus, we check (a) if the existing view orders allow producing e outputs in the right order for q by *some* possible physical implementation of e, or (b) if not, if e projects sufficiently many IDs to enable sorting e's output as desired. If some of the desired operations cannot be added, attemptFinalize fails. In the example in Figure 3.4, the canonical rewriting found is $\pi_{d.id,f.id}(e_1)$.

Correctness Algorithm 1 is correct, i.e. if attemptFinalize produces an output and flags it as a complete rewriting (line 13), that is indeed canonical rewriting of q. This is guaranteed by the fact that e is equivalent to d (and the minimized d). If the minimized d is isomorphic to q, and the necessary selection, projection and sort operations could be applied on its equivalent expression e, then the result is an equivalent rewriting of q.

Completeness If a canonical rewriting based on a set of views and embeddings exists, Algorithm 1 produces it. This is because of the aggressive application of node predicates (lines 5-8), enforcing as many of the query-derived relationships between nodes as possible. Intuitively, omitting one of the predicates may lead to an undesired cartesian product in e. For instance, in Figure 3.4, if we omit the predicate $c.id = c.id$ from e_1, we obtain the DAG_2 from the same Figure, which, after minimization, is not a tree. If we omit the predicate $d.id \prec f.id$, we obtain the DAG_3 in Figure 3.4, which, after minimization, is a tree, but attemptFinalize cannot turn it into a rewriting, since f is not a descendant of d.

Complexity Algorithm 1 runs in quadratic time in the combined size of the views; the most expensive operation is the DAG minimization.

Bound on minimal rewriting size We now prove that a minimal canonical rewriting of q uses at most $|q|$ views.

We start by proving the following lemma: for any query node n, there must exist at least a view node m_i of a view v_i used in the rewriting, such that $\phi_i(m_i) = n$ and m_i has *at least as many attributes* as n. We distinguish possible cases by the number of attributes that n has:

- No attributes: the fact that at least one view v_i *must* have a node m_i mapping to n satisfies our claim.
- One attribute: the rewriting must provide it, so at least one of the m_i nodes mapped to n must have it.
- Two attributes: consider first the case when one attribute is an *id*. Either m_i is the only view node mapped to n (in which case m_i must also provide the other attribute), or there are several view nodes mapped to n. Among these, some may have no attributes at all, but those which

do have attributes *must all* have *id*s, to enable the corresponding view join[1]. Among these joined views, having m_j nodes with *id*s such that $\phi_j(m_j) = n$, one at least must also provide the other attribute of n. Now consider the case when n has *val* and *cont*. Similarly, either m_i is the only view node mapped to n, thus it provides both; or, the rewriting joins several views by the equality of their nodes mapped to n. Among these nodes, some must have *cont*, and those who do, also have *val* due to expansion (Section 3.2.2).

– Finally, if n has *id*, *val* and *cont*, either m_i is the only view node mapped to n and it has these attributes, or several views are joined on *id*s of nodes mapped to n. The first one to have *cont*, also has *val*, due to expansion.

Let $v_1, \ldots, v_k$ be an ordering over the views in the rewriting. Based on the lemma, we define the *contribution* of v_i, denoted $C(v_i)$, as the set of query nodes n, such that (a) a node of v_i is mapped to n and has at least all the attributes of n, and (b) no view appearing *before* i in the rewriting satisfies this, i.e., for all $j < i$, either v_j does not have a node mapped to n, or that node does not have all the attributes of n.

It is easy to see that for two distinct views v_i, v_j, the sets $C(v_i)$ and $C(v_j)$ are disjoint.

A view v_i such that $C(v_i) = \emptyset$ is redundant. This is because any node relationship, or attribute, which v_i brings to the rewriting, can also be found using the previous views. Thus, for v_i not to be redundant, $|C(v_i)|$ must be at least 1.

Finally, the union of the $C(v_i)$ sets, for all views v_i, is the set of the query nodes. Thus, at most $|q|$ views participate to a minimal rewriting. □

For example, consider the rewriting of q_3 based on v_4 and v_5 discussed above. In this case, $C(v_4) = \{a, b, c, d\}$, and $C(v_5) = \{f\}$. This rewriting is minimal. Now assume that we also add the view v_6 from the Figure to the rewriting, before v_4 and v_5. Then, $C(v_6) = \{c, f\}$, $C(v_4) = \{a, b, d\}$, $C(v_5) = \emptyset$ and v_5 is redundant.

DAG vs. rewriting minimality Algorithm 1 minimizes the DAG d, not the rewriting e. Using the C sets, one can extract from a non-minimal rewriting e *some* minimal one, in time polynomial in $|q|$.

3.3.2 Rewriting algorithms

The first end-to-end rewriting algorithm we consider is called **Subset-Enumeration**, or **SE** in short (Algorithm 2). It iterates over all $\mathcal{W}$ subsets

1. Otherwise, undesirable cartesian products (recall Partial Rewritings from Section 3.2.2) or, equivalently, non-tree DAGs (such as DAG_2 in Figure 3.4) may occur, and prevent rewriting.

Algorithm 2: Subset-enum

Input : query q, view set $\mathcal{V}$
Output: all minimal canonical rewritings of q based on $\mathcal{V}$

```
1 U ← prune(V, q); W ← ∪_{v∈V} expand(v)
2 R ← ∅
3 foreach qc = {v_1, v_2, ..., v_k} subset of W, |qc| ≤ |q| do
4     foreach tuple φ_1, φ_2, ... φ_k of embeddings from v_1, v_2, ..., v_k
      into q do
5         e ← views-to-rewriting(qc, φ_1, φ_2, ... φ_k) (use Algorithm 1)
6         if e is an equivalent rewriting then
7             add e to R
8 remove from R non-minimal rewritings
9 return R
```

of size at most $|q|$, all embedding combinations from the views into q, and accumulates rewritings in the set R. A rewriting r is non-minimal if another rewriting $r' \in R$ uses a subset of r's views.

Algorithm SE does not specify a subset enumeration order; thus, in the worst case, all rewritings are enumerated before a minimal one is returned. A simple improvement is **Increasing-Subset-Enumeration**, or **ISE**, which builds $\mathcal{W}$ subsets from the smallest to the largest. Using a proper trie structure for R, one can efficiently check if a subset of the views used in a rewriting has already lead to another rewriting, and if so, discard the larger one.

Algorithm ISE repeats a lot of work. For example, let v_7 be the view b_{val} and v_8 be the view $b_{c_{id}}$. They cannot be joined on b, and any $\mathcal{W}$ subset including them both will not lead to a rewriting. However, Algorithm ISE will try such subsets. Similarly, if v_9 and v_{10} can be joined, then this partial result could be stored to be re-used in several larger rewritings.

Based in this intuition, we devise a bottom-up, **Dynamic Programming Rewriting** algorithm (or **DPR**, in short). It attempts to build larger and larger partial rewritings, by combining smaller ones. The initial set of rewritings is made of the pairs of ($\mathcal{W}$ view, embedding in the query). Then, DPR combines an existing rewriting, and a rewriting made of only one view, akin to building left-deep plans during optimization. However, unlike an optimizer, DPR only explores one ordering per sets of views, exactly to avoid doing the optimizer's work. To combine two partial rewritings, namely e_1 over the set of views V_1 and set of embeddings Φ_1, and e_2 similarly based on V_2 and Φ_2, DPR invokes Algorithm 1 on $V_1 \cup V_2$ and $\Phi_1 \cup \Phi_2$. Coming back to the above examples, DPR will observe that v_7 and v_8 cannot be combined, and not attempt a rewriting combination if $\{v_7, v_8\}$ is a subset of $V_1 \cup V_2$. The partial rewriting joining v_9 and v_{10}, returned by Algorithm 1,

will be used to build larger rewritings using one extra view. This give DPR a significant reduction of work over ISE.

Algorithm DPR will identify a rewriting of k views only after having tried all rewritings using up to $k-1$ views, which may take too long.

To alleviate this, we propose the **Depth-First Rewriting** algorithm (or **DFR**, in short). Like DPR, it is bottom-up, and it builds only minimal, left-deep rewritings. However, instead of exploring all combinations of increasingly many views, DFR is a greedy algorithm. At any moment, it picks the partial rewriting *covering the most query nodes* found so far, and joins it with a 1-view partial rewriting. This leads DFR to frequently finding a first rewriting very fast. In exchange, when DFR tries a set V of views, its subsets may have not been previously explored. For instance, it may explore $\{v_7, v_8\}$ after $\{v_7, v_8, v_{12}\}$ and after $\{v_7, v_8, v_{13}\}$, thus discover the incompatibility of v_7 with v_8 several times.

ISE, **DPR** and **DFR** are correct and complete; they produce the minimal rewritings of q given $\mathcal{V}$. ISE and DPR produce the rewritings having the fewest number of views possible, before the others. ISE and to a lesser extent DFR may repeat some work. ISE and DPR produce rewritings towards the end of the search, whereas DFR may produce some very early on.

3.3.3 Evaluating a rewriting

A logical rewriting plan must be optimized by standard algebraic transformations, e.g. transforming the $\sigma(\times)$ into a join tree, pushing σ and π etc., and then transformed into a physical plan. In ViP2P, this plan is typically distributed over the peers in the DHT. The execution engine includes standard implementations for *scan*, σ, π, hash joins, binary structural joins [15] and a holistic twig structural join [28]. In the view definition index, we annotate the view tree pattern with its cardinality (known at the view peer), allowing the optimizer to decide about join orders. The optimizer applies heuristics to reduce, first, inter-site transfers, and second, the number of sort operations.

3.4 P2P view management

We have so far explained how to exploit views for query rewriting. We now consider how views are materialized (Section 3.4.1), and identified in order to rewrite a query (Section 3.4.2) in the DHT network. Both operations require some *view definition indexing* in the DHT. We stress that we do not index view extent (tuples), but only the pattern defining the views.

We start by introducing a useful term: if d is a document and v is a view such that $v(d) \neq \emptyset$, we say d *affects* v.

3.4.1 View materialization

Assume peer p decides to establish a view v. Then, when a peer p_d publishes a document d affecting v, p_d needs to find out that v exists. To that effect, view definitions are *indexed for document-driven lookup* as follows. For any label (node name or word) appearing in the definition of the views $v_1, v_2, \ldots, v_k$, the DHT will contain a pair where the key is the label, and the value is the set of view URLs $v_1, v_2, \ldots, v_k$.

When a peer p_d publishes a document d, p_d performs a lookup with all d labels (node names or words) to find a superset S_a of the views that d might affect. Then, p_d evaluates $v(d)$ for each $v \in S_a$. We implemented this step based on a SAX traversal, with time complexity in $\Theta(|d| \times |v|)$. In practice, large fragments of d are typically not interesting for a given view v, thus computing $v(d)$ tends to spend some time traversing useless parts of d. To share this cost, we group view definitions in batches of some size n (we set $n = 10$) and evaluate all the views of a batch in a single d traversal. Thus, d fragments useless to all the views in a batch, are parsed only once per batch.

Finally, p_d sends, for each view v, the tuple set $v(d)$ (if it is not empty) to the peer p_v publishing v. Recall from Section 3.1 that element IDs include document URIs, which may get rather lengthy. To speed up transfers, tuples are encoded so that the URI of d is sent only once for the tuple set $v(d)$.

We have so far considered that v is published before the documents which affect it. The opposite may also happen, i.e. when v is published, a document d affecting v may already exist, and $v(d)$ needs to be added to v's extent. To that effect, we require the publisher p_d of a document d to periodically look up the set of views potentially affected by d, and send $v(d)$ to those views as described above. Thus, v will be up to date (reflecting all network documents that affect it) after the periodical check and subsequent actions have been performed by all document publishing peers.

We end the Section by considering view maintenance in the face of document deletion or change. When documents are deleted from the system, a similar view lookup is performed, and the peers holding the views are notified to remove the respective data. We model document changes as deletions followed by insertions.

3.4.2 Identifying views for rewriting

A second form of view definition indexing is performed in order to find views that may be helpful for rewriting a given query. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy*. For each such strategy, a *view lookup* method is needed, in order to identify, given a query q, (a superset of) the views which could be used to rewrite q. Many strategies can be devised. We present four that we have implemented, together with the space complexity of the view

indexing strategy, and the number of lookups required by the view lookup method. We also briefly show that these strategies are *complete*, i.e. they retrieve at least all the views that could be embedded in q and, thus, lead to q rewritings.

Label indexing (LI): index v by each v node label (either some element or attribute name, or word). The number of (key, value) pairs thus obtained is in $O(|v|)$.

View lookup for LI: look up by all node labels of q. The number of lookups is $\Theta(|q|)$.

LI completeness is quite straightforward (details omitted).
The LI strategy coincides with the view definition indexing for document-driven lookup (described in the previous Section). An interesting variant can furthermore be devised:

Return label indexing (RLI): we index v by the labels of all v nodes which project some attributes (at most $|v|$).

View lookup for RLI, interestingly, is the same as for LI. The labels on which LI indexes v, and RLI doesn't, are those of v nodes without attributes. On such nodes, no join can be applied on v (due to the lack of *id*), and no navigation (due to the lack of *cont*). Moreover, such nodes obviously do not provide attributes corresponding to those returned by the query. Therefore, one does not need to advertise v based on their labels.

The drawback of *LI* and *RLI* is their lack of precision. For instance, a view $a_{id}(c_{id})$ will be retrieved for all queries involving the terms a, although it is useless for all queries not containing c. A more precise strategy is the following.

Leaf path indexing (LPI): let $LP(v)$ be the set of all the distinct root-to-leaf label paths of v. *In this context, a path is just a sequence of the node names, it does not include the edges.* Index v using each element of $LP(v)$ as key. The number of (key, value) pairs thus obtained is in $\Theta(|LP(v)|)$.

View lookup for LPI: let $LP(q)$ be the set of all the distinct root-to-leaf label paths of q. Let $SP(q)$ be the set of all non-empty sub-paths of some path from $LP(q)$, i.e., each path from $SP(q)$ is obtained by erasing some labels from a path in $LP(q)$. Use each element in $SP(q)$ as lookup key.

As an example, let $v = a_{id}(b_{id}, c_{id})$, then v will be indexed by the keys $a.b$ and $a.c$. Let q be the query $a(f(b_{id}, c_{id}))$. With LPI, the view lookups will be on a, $a.f$, $a.b$, $a.c$, f, $f.b$, $f.c$, b, and c. Thus, v will (correctly) be

identified as potentially useful to rewrite q. Indeed, if a view $v' = f_{id}$ exists, then $q = \sigma_{a \prec f \wedge f \prec b \wedge f \prec c}(v \times v')$.

Let $h(q)$ be the height of q and $l(q)$ be the number of leaves in q. The number of lookups is bound by $\Sigma_{p \in LP(q)} 2^{|p|} \leq l(q) \times 2^{h(q)}$.

LPI completeness: observe that if a view v can be embedded in the query q, then $LP(v) \subseteq SP(q)$.

The last strategy we consider is:

Return Path Indexing (RPI): let $RP(v)$ be the set of all rooted paths in v which end in a node that returns some attribute. Index v using each element of $LP(v)$ as key. The number of (key,value) pairs is also in $\Theta(|RP(v)|)$.

View lookup for RPI coincides exactly with the lookup for LPI. RPI completeness is shown similarly to RLI.

3.5 Performance evaluation

In this Section, we present a set of experiments we made to estimate the performance of various aspects of our architecture. Section 3.5.1 briefly describes our platform. Section 3.5.2 presents the experimental setup for the next two Sections: Section 3.5.3 considers view materialization, while Section 3.5.4 studies query processing. Section 3.5.5 studies view indexing and lookup techniques, whereas Section 3.5.6 focuses on query rewriting on one peer. Section 3.5.7 concludes our study.

3.5.1 System implementation and configuration

We have fully implemented the platform described so far, using Java 6. Berkeley DB (version 3.3.75, available from http://www.oracle.com) and FreePastry (version 2.1, available from http://freepastry.org) are used for storing view data and indexing view definitions respectively. For the implementation of the *nav* operator, patterns are translated to XQueries, and executed by the Saxon XQuery processor (version Saxon-B 9.1, available from http://saxon.sourceforge.net). The *nav* operator is always placed on the same peer as its input, thus it is evaluated locally.

We have made some optimizations to speed up inter-peer data transfers. More precisely, when sending a stream of tuples, potentially including many document URIs in node IDs, we encode the URIs on the fly in compact integers, and send the dictionary with the tuples, so that they can be decompressed on the other side.

In our experiments, unless otherwise specified, we have deployed **1000 ViP2P peers on 250 machines** on the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other fund-

ing bodies (see https://www.grid5000.fr). The machines are distributed across 9 big French academic centers. They have between 2 GB and 4 GB of memory; most of them are multi-cores. All run 64 bits Debian Linux 2.6.18. We have installed 4 ViP2P peers on each machine. *Due to Grid5000 restrictions, we could not reserve the same sets of machines for all experiments.* We ran most experiments three times and averaged the times; the difference between 2 runs was up to 20% of the values, but the general tendencies were stable.

3.5.2 Setup for view building and query processing

The peers publish a total of 2000 XMark benchmark documents [47] of equal size; the total size of the network documents varies across successive runs, from 400 MB to 3.2 GB. The peers also publish 500 views of up to 7 nodes. 70 views are affected by all documents; the others use XMark node names but have no results on XMark documents. The documents and views are split uniformly over the network. The views are indexed using LI. All 500 views are retrieved for all 2000 documents by the document-driven lookup method described in Section 3.4.1.

Once views are indexed, a designated *coordinating peer* sends to all others a *start* signal. Then, in parallel, the peers look up views, extract data, send and receive tuples, and store them in their local BerkeleyDB databases. After all its tuples are stored, each peer sends a *done* signal to the coordinating peer. Of course, this synchronization is just for the experiment, and is not needed otherwise.

3.5.3 View building

Figure 3.5 shows the total time needed to evaluate 500 views on the 2000 documents. Extraction takes place at the documents' sites. The times are summed up for all the peers; in reality, extraction takes place in parallel. As expected from the description in Section 3.4.1, extraction time grows linearly with the total document size.

Figure 3.6 shows the time measured at the coordinating peer, between its *start* signal and the last of the 1000 *end* signals. It can be seen as the time to load the network with our documents and views, at the fastest possible pace. The time grows linearly in the data size, as was to be hoped.

Data transfers for view materialization increased linearly in the size of the documents. For the 3.2 GB of published data, we transferred 468 MB of data for view materialization, after URI compression.

3.5.4 Query evaluation

Once the views are loaded, we ask the query:

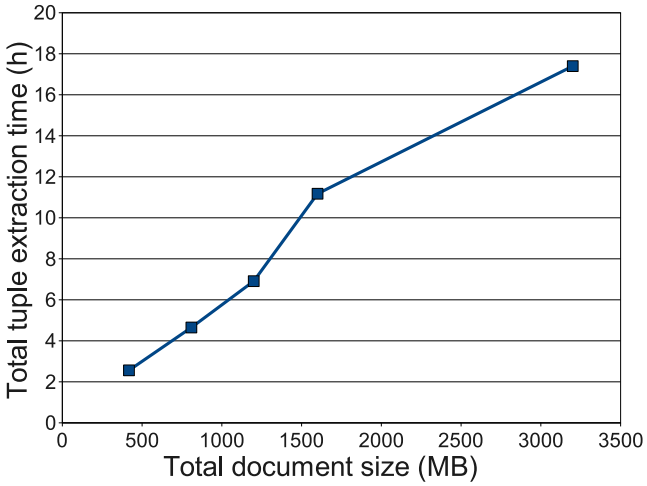


Figure 3.5: Total tuple extraction time.

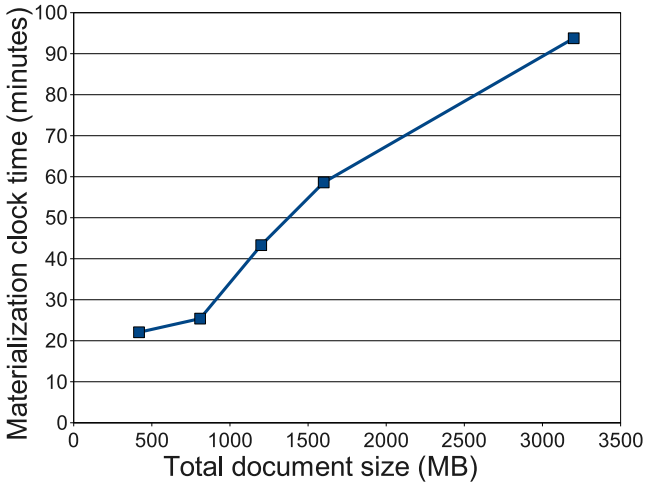


Figure 3.6: Observed view materialization latency.

$$site_{id}(regions_{id}(africa_{id}(item_{id})), catgraph_{id}(edge_{id}))$$

Query rewriting and optimization at the query peer take, respectively, 30 ms and 100 ms. The smallest rewriting uses two views on two machines, different from one where the query is asked.

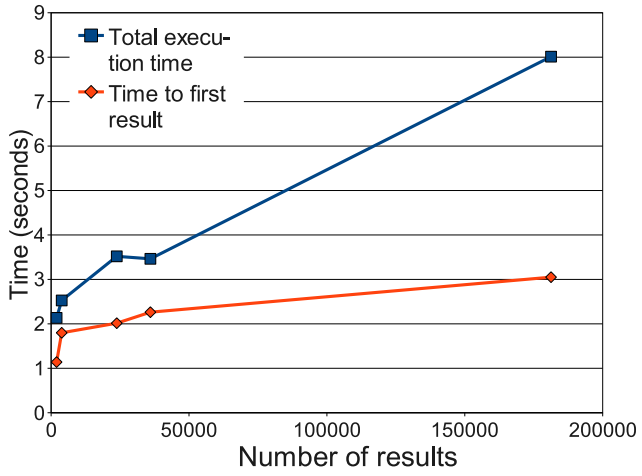


Figure 3.7: Execution time for increasing data size.

Query execution Figure 3.7 shows that as expected, query execution time scales up with the size of the data set.

Data transfers for query processing also grew linearly with the total document size, up to 12.57 MB for processing our query on the 3.2 GB document set.

The benefits of VIP2P views can be appreciated on the following simple example. We use a data set of 750 XMark [47] documents having the total size of 20 MB. We use three different view sets to rewrite the query $site(item(description_{cont}))$:

- $\mathcal{V}_1$ contains the view $site_{cont}$. This corresponds to storing the full documents in one single view; we use it to have a glimpse of the interest of *document-level granularity* indices. Indeed, a system such as [33] would identify all the corresponding documents and then evaluate the query on the fly on those documents. We proceed quite in the same way, by our rewriting $nav_{site_{cont},item(description_{cont})}(v_1)$.
- $\mathcal{V}_2$ contains three views: $site_{id}$, $item_{id}$ and $description_{id,cont}$. This corresponds to the node-granularity indexing used in [5], but unlike [5], we also time the transfer of the XML results to the query peer.
- $\mathcal{V}_3$ contains one view which is exactly q.

This experiment was made with 2 peers in a 10 GB LAN, to minimize data transfer impact. The view lookup and rewriting times are negligible; the execution times are: 8.8 seconds for $\mathcal{V}_1$; 2.1 seconds for $\mathcal{V}_2$; and 1 second for $\mathcal{V}_3$. As expected, having a view exactly matching the query is best. This

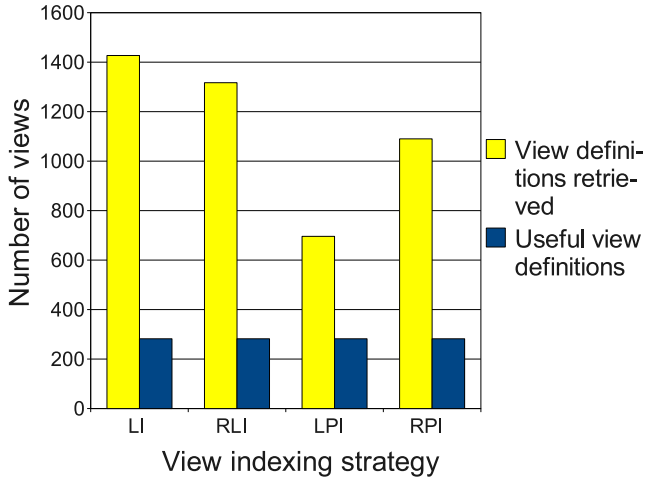


Figure 3.8: View definition retrieval.

exemplifies the query speed-up that can be obtained using views, if we pay the cost of building them.

3.5.5 View indexing and lookup strategies

In this Section, we compare the view indexing and lookup strategies LI, RLI, LPI and RPI described in Section 3.4. We consider a synthetic query q of 30 nodes labeled $a_1, \ldots, a_{30}$. Each node of q has between 0 and 2 children, and q's height is 5. From q, we create three variants:

- q' has the same labels as q, but totally disagrees with q on the structure (whenever a_i is an ancestor of a_j in q, this does not hold in q')
- q'' coincides with q for half of the query (one child of the root), while the other half conserves the corresponding q labels but totally changes structure (as q' does)
- q''' has the same structure as q, half of it has the same labels $a_1, \ldots, a_{15}$, while the other half uses a different set of tags $b_1, \ldots, b_{15}$ (instead of $a_{16}, \ldots, a_{30}$).

From each of q, q', q'' and q''' we automatically generate 360 views of 2 to 5 nodes, for a total of 1440 views. The views can all be embedded into the respective queries, i.e. those generated from q can be embedded in q, those generated from q' can be embedded in q' and so on. We, thus, obtain a mix of views ressembling the query to various degrees. To this randomly-generated view set, we added 3 hand-picked views to ensure that one query rewriting exists.

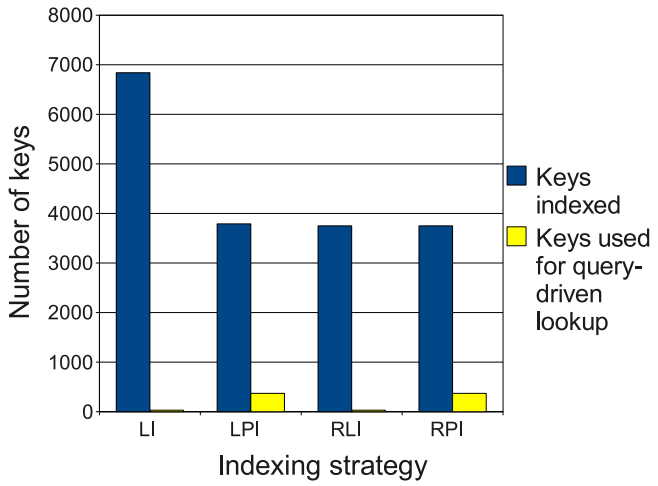


Figure 3.9: Index entries and lookups generated for the views.

We have indexed the resulting 1443 views in our network, following the LI, RLI, LPI and RPI strategies described in Section 3.4. We then performed the four corresponding lookups.

Figure 3.8 shows how many views have been retrieved for each strategy, compared with the number of useful views (those that are found to be embeddable in q, in our example, those generated from q, and possibly some generated from q'' and q'''). We see that the path indexing-lookup strategies (LPI and RPI) are more precise than label based ones (LI and RLI). Moreover, LPI is the most precise. This is because LPI uses longer paths as keys, thus, it describes views more precisely, eliminating some false positives.

Figure 3.9 presents the number of (key, value) pairs added to the index by each view indexing strategy, and the number of lookups needed by each strategy for the query we considered. As expected, LI leads to most index pairs. With respect to query-driven lookup, LI and RLI lead to 30 lookups, much less than LPI and RPI lead to 370 lookups.

Figure 3.10 shows the time to obtain the initial set of views. The Figure distinguishes the time to perform in parallel all the lookup calls on Pastry, and the time to test if each view is useful by embedding it into q. The Figure shows that the simple LI strategy is the best. Indeed, even though Pastry lookups are asynchronous, issuing many lookups from the same peer comes with a penalty, thus, LPI and RPI, which needed 370 lookups, are significantly slower. LI makes up for its low precision by requiring few lookups.

We mention that rewriting the query based on the relevant views (282 in this example) takes around 6 seconds, whereas finding the first solution takes around 0.5 seconds. Comparing this with the times in Figure 3.10,

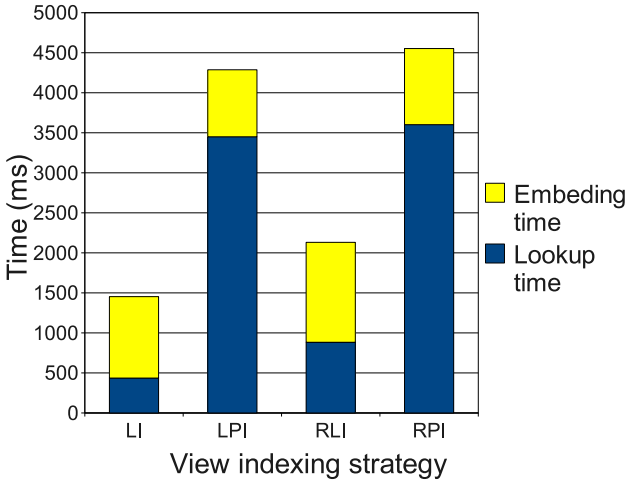


Figure 3.10: Identifying useful views.

one notices that view definition look-up is quite short, which validates the feasibility of retrieving view definitions at query rewrite time. One may also consider *locally caching* view definitions, to completely avoid look-ups. The view pruning time could further be reduced as we explain in Section 3.7.

3.5.6 Query rewriting

We use queries of 5, 9, 13 and 17 nodes, respectively. Each query is a balanced binary tree where all internal nodes have two children. All nodes have different labels; the root has *id*, the other nodes have no attributes. This experiment ran on a MacBookPro on the Darwin 9.6.0 kernel, and having a 2.5 GHz Intel Core 2 Duo processor.

First, for each query, we make a set of $|q|$ 1-node views, one per query label, each having an *id*. This is a very hard case for our bottom-up algorithms, as almost any subset of views can be joined. The expected complexity here is $O(|q|^{|q|+2})$.

Second, we devise for each query another set of approx. $|q|/2 + 1$ views. One of these views copies the top 2 levels of the query nodes; the remainder ones are small subtrees of 1-3 nodes, made of the lowest levels in the query trees. In these sets of views, only about half of the nodes have *id*s (we took care that rewritings still exist). The complexity her is in $O((|q|/2)^{|q|+2})$. Reducing *id* presence also reduces the join opportunities and thus, simplifies the problem. In both cases, all the views can be embedded in the query.

Figure 3.11 shows, for the first family of view sets (top) and the second family (bottom) the total time, and the time to the first rewriting, taken by ISE, DPR and DFR. The missing points are times longer than 2 minutes.

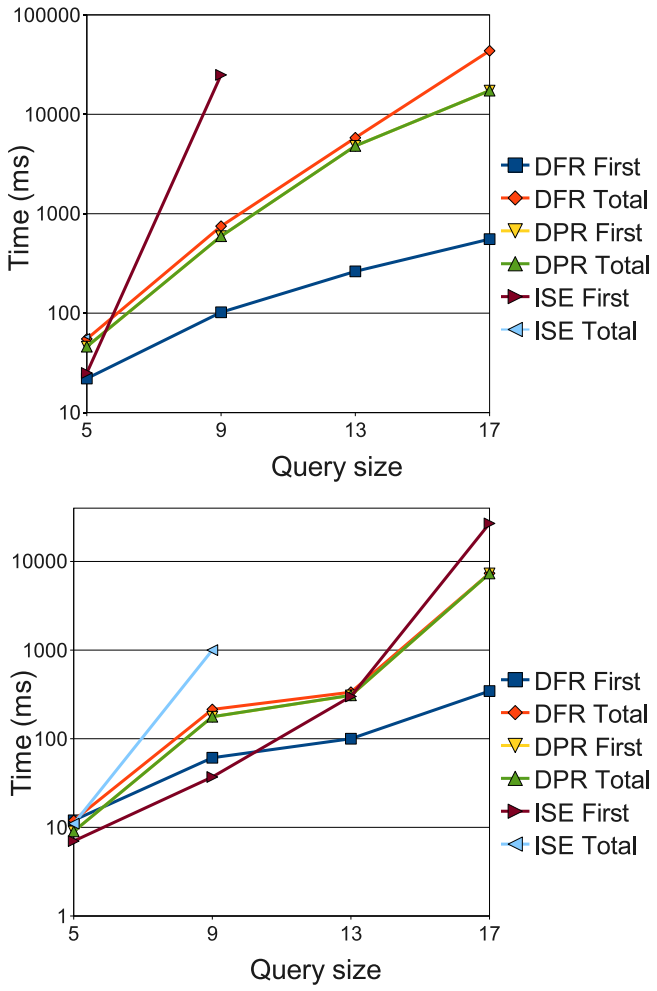


Figure 3.11: Query rewriting times.

The highest times are 43 seconds for DFR-Total (at the top) and 27 seconds for ISE-First (at the bottom). Recall that the complexity of the problems we study is in $O(|\mathcal{V}|^{15})$. Figure 3.11 shows, first, that ISE does not scale; the total time is very large even for $|q| = 9$ in the upper graph. Second, as expected, for DPR the total time and the time to the first solutions almost coincide. Third, DFR reaches a first solution much faster than the others; we checked and these first rewritings were also of the minimal size (although this cannot be guaranteed in general). Fourth, DFR total time is indeed longer than DPR's, due to the fact that DFR may repeat some work since it does not explore subsets the increasing order of their sizes.

Finally, we consider again the 17-nodes query and the 9 views used in lower part of Figure 3.11. We add a view of 1 node, with the label of the query root, and having *cont*. The query root has 2 children, thus, as explained in Section 3.2.2, expansion transforms this view into 4 views (and not 2^{17}). Thus, rewriting proceeds with 13 views. Now, the smallest rewriting uses just the fully expanded view; DFR, DPR and ISE all find it in less than 100 ms. The total times are respectively 5.8 seconds, 4.3 seconds, and more than 2 minutes.

3.5.7 Conclusions of the experiments

Our experiments show that the VIP2P approach for view materialization and query processing scales up linearly in the data size, on a network of 1000 peers. With respect to the rewriting problem, when queries are complex and/or there are many views, DFR tuned to stop after the first rewriting gives reasonable performance. Rewriting time is also strongly correlated to the number of *id*s in the views, since they enable joins.

View indexing and lookup are relatively fast, which validates the feasibility of exploiting views distributed over the peer network. Among the view indexing strategies we compared, LI cuts the most interesting compromise between precision, number of entries in the DHT index, and number of lookups needed for a given query to rewrite.

On one simple example, we have demonstrated the potential for performance improvement provided by VIP2P views, over DHT indexes either at document granularity level, or at node level. This demonstrates that there exists a large in-between space, where views closely suited to application needs can provide significant performance benefits.

3.6 Software architecture design

ViP2P is a big project that contains 44 packages and sub-packages, 294 classes and around 60.000 lines of code. Part of the project comes from a previous project called ULoad [19]. An overview of the components of an AXML peer is shown in Figure 3.12. We proceed by presenting the various modules of ViP2P, their functionalities and how they interact among them.

ViP2P Core This module is responsible to initialize and coordinate all the major modules of the system. It initializes:

- the Pastry code and creates a Pastry peer
- the document arrival module, the query arrival module and the view arrival module. These three modules are responsible of monitor specific directories in the peer's file system, in order to identify newly arrived documents, queries and views

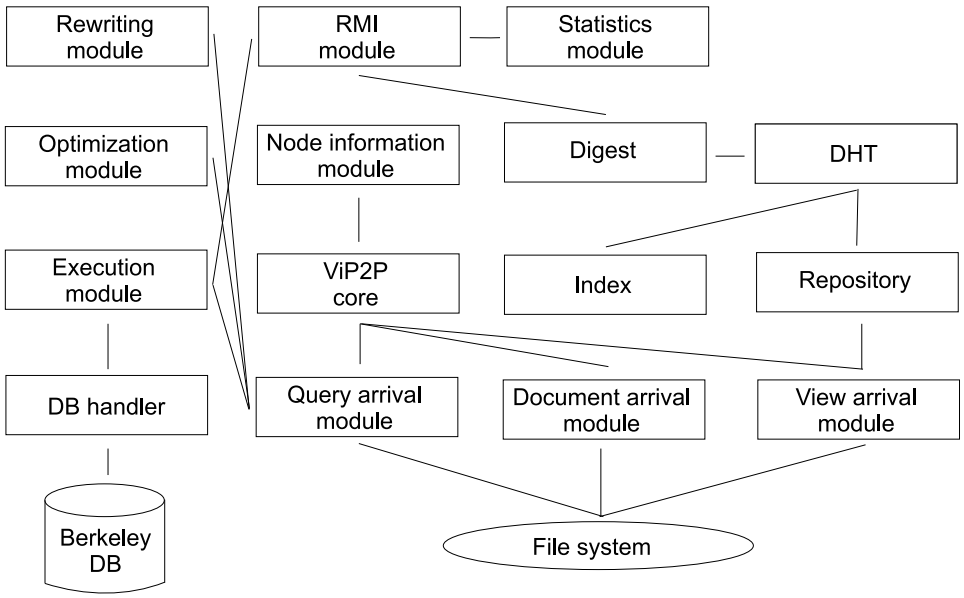


Figure 3.12: An overview of the components of the ViP2P peer and the major interactions among them.

– other subsystems like the RMI module which is needed for the inter-peer communication.

Document arrival module This module will search repeatedly every few minutes for newly arrived documents. As soon as a new document is placed in the document directory of the peer, the peer starts to publish the document. The first step is to look up in the DHT for views that might be interested in this document, i.e., those to which the document is likely to contribute data. This lookup is done with the use of the digest and depending on the digest configuration chosen, with the use of the DHT.

Digest The digest retrieves, when a document arrives, the definitions of a superset of the views that are interested in that document. Two different implementations of this module have been realized and can be alternatively used:

DHT digest In this implementation we make use of the DHT to index the view definitions and to look them up. By using the DHT we achieve load distribution and load balancing when using the digest. Moreover, there is no centralized point of control which makes our system less vulnerable. This digest faithfully implements the view indexing and look up mechanisms explained in Section 3.4.1.

Centralized digest This digest is hosted at a single, designated super-peer, which centralizes the information and then broadcasts it to all

	l_1	$\dots$	l_m
v_1	0	$\dots$	0
$\vdots$	$\vdots$		$\vdots$
v_n	0	$\dots$	0

Figure 3.13: Centralized digest.

the peers. This digest was not implemented first as a way to bootstrap the system. Obviously, it introduces a bottleneck and a single point of failure. However, it behaves well and better that the DHT digest in small networks of 100 to 200 peers with less that 2000 views.

No digest This is the simplest implementation of the digest interface. In this case, as the title reveals, there is no digest structure. When a document arrives, a super-peer is contacted which provides the definitions of all the views of the system. This implementation has even more disadvantages than the centralized digest, since it does not filter the view definitions that it returns. This was developed for testing purposes and is now obsolete.

The centralized digest can be seen as a $n \times m$ array, where n stands for the number of the available views, and m stands for the number of different labels of all the views of the system. Figure 3.13 shows an example of the centralized digest. Every column represents a label and every row represents a view. The element $e_{i,j}$ of the digest is 1 if the view v_i has a label l_j, otherwise $e_{i,j} = 0$. Hence, if the set of all the labels that appear in all available views is L, then each row r_i of the digest shows which labels of the set L appear in the view v_i. When a document arrives, we traverse the document to see which of the labels of L appear in it. Based on this traversal we construct a row r_d and then for each of the rows r_i of the digest we perform the check $r_d \wedge r_i = r_i$. If the check is satisfied, it means that all the view labels of the view v_i appear in the specific document. Closing, we should mention that its size is not very big because the digest data are bits. Additionally the check $r_d \wedge r_i = r_i$ can be performed easily and fast because it is a logical and calculation between two bite arrays.

View arrival module This module is responsible for handling newly arrived views. It continuously monitors the view directory and when a new view is added by a user, it informs the *digest*, the *repository* and the *index*.

Repository The purpose of this module is to store temporarily, or permanently, newly arrived view descriptions. These descriptions are indexed in the DHT using an *interval timestamp*, corresponding to the moment when

the view was added to the system. The interval timestamps are a feature implemented by the underlying DHT (in our case, Pastry). The timestamp is computed as:

$$ts = currentTime - (currentTime) mod (interval)$$

which means that time is divided into intervals, and all views published in a given interval are associated (in the repository) to the timestamp of that interval. Figure 3.14 illustrates this concept. In this Figure, v_1 belongs to the interval $(t_{i+1}, t_{i+2}]$, v_2 belongs to the interval $(t_{i+2}, t_{i+3}]$ and v_3 belongs to the interval $(t_{i+3}, t_{i+4}]$.

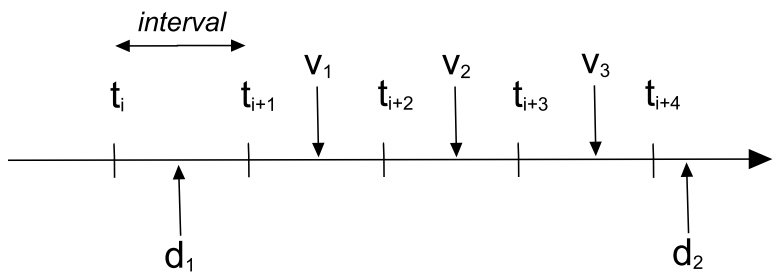


Figure 3.14: Update and lookup of the repository.

The document arrival module of a peer is responsible for checking the repository for new view descriptions that might be interested in that peer's documents. Thus, it checks repeatedly every *interval* moments of time for new views. Whenever a new view definition is found, the document arrival module of the peer checks, for each of the documents, if we have already processed that view using the digest module. If not, it checks if the view might be interested in any of the documents. This filtering is done by comparing the view labels with the document labels.

Going back to Figure 3.14, we see that document d_1 arrives during the $(t_i, t_{i+1}]$ time interval, and it will need the help of the repository to discover views v_1, v_2 and v_3 which arrive later. d_2 will discover views v_1 and v_2 using the digest and it will discover v_2 and v_3 using the repository. Note that v_2 will be returned and by the digest and by the repository and this is the reason why we should keep track of the views that each document has discovered.

Index The index module is responsible to index and retrieve view descriptions based on the strategies that are presented in Section 3.4.2. The goal of the index module is to make globally available, at query time, all the view descriptions of the ViP2P peers. This is achieved using the DHT module with which all the ViP2P peers are equipped.

As we have already seen, the *digest*, the *repository*, and the *index* make use of common modules, such as the DHT, and may index and look-up view definitions in a similar way. However, they have a fundamental difference. The *digest* is used by the documents to discover views that have arrived before their arrival to the system, the *repository* is used by the documents to be informed for views that have arrived after their arrival to the system and the *index* is used by the *queries* to be informed for views that may be useful to their rewriting.

Query arrival module This module of ViP2P monitors the directory where the query files are placed. Whenever a new query is detected, this module starts the procedure for replying it by:

- making a look-up using the *index* module in order to find descriptions of interesting views that can be used for the rewriting of the query
- contacting the *rewriting* module with the query and the view definitions found and asking for a rewriting (which is a logical plan)
- contacting the *optimization* module asking the conversion of the logical plan to a physical plan
- executing the physical plan via the *execution engine*

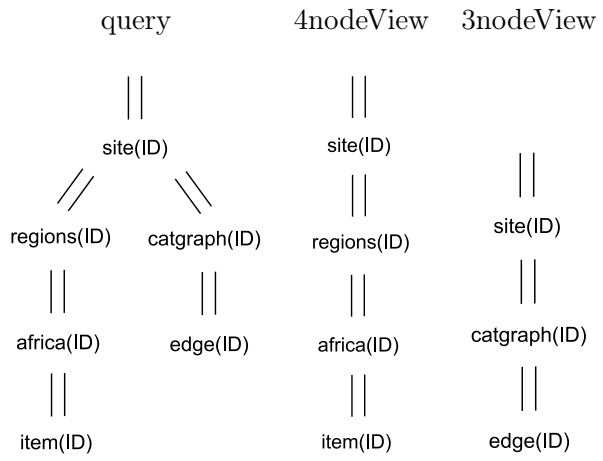


Figure 3.15: A tree-pattern query and views which can be used for its rewriting.

Rewriting module The functionality of this module has been thoroughly explained in Section 3.3. It uses a given query and a set of available view descriptions in order to produce a logical plan using the ViP2P's available logical operators: *cartesianproduct*, *join*, *navigate*, *scan*, *projection*, *selection*, *sort*, *structural join* and *structural semi − join*. Figure 3.15

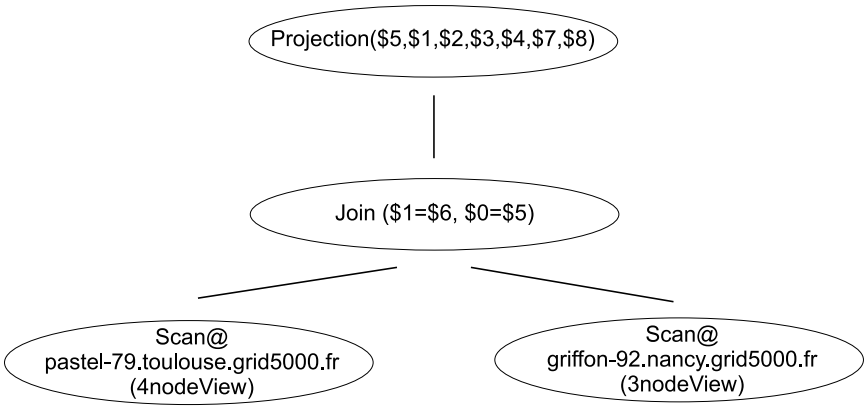


Figure 3.16: A possible output of the rewriting module using the query and the views of Figure 3.15 as input.

shows an example of a tree-pattern query with views that can be used for its rewriting. Figure 3.16 shows a rewriting of the query using the views of Figure 3.15. A rewriting is a logical plan that can be given to the *optimization module* for conversion to a physical plan. We observe that the logical plan is indicating how the views $4nodeView$ and $3nodeView$ should be combined and it is not interested in how the data will arrive to the query peer or where the operators will be executed.

DHT module This module is nothing more than a wrapper to the underlying DHT used. We have created a wrapper for the DHT methods that we use in order to enable possibly switching to any DHT in the future. However, using some DHT-specific modules (such as the global time/interval service provided by Pastry) does raise some obstacles to this goal.

Optimization module The purpose of this module is to transform a logical plan (Figure 3.16) to a physical plan (Figure 3.17). This is achieved by selecting the right physical operators, introducing sorting operators whenever the chosen physical operators ask for sorted inputs and inserting *Send* and *Receive* operators to transfer data between peers. The available physical operators are: *Scan*, *HashJoin*, *Nested Loops Join*, *Holistic Twig Join*, *Structural Ancestor Join*, *Structural Descendant Join*, *Saxon Navigation*, *Selection*, *Projection*, *Send*, *Receive*, *Memory Sort* and *Berkeley DB Sort*.

Execution engine This module provides implementations of all the physical operators of ViP2P. These are a subset on those developed for the ULoad

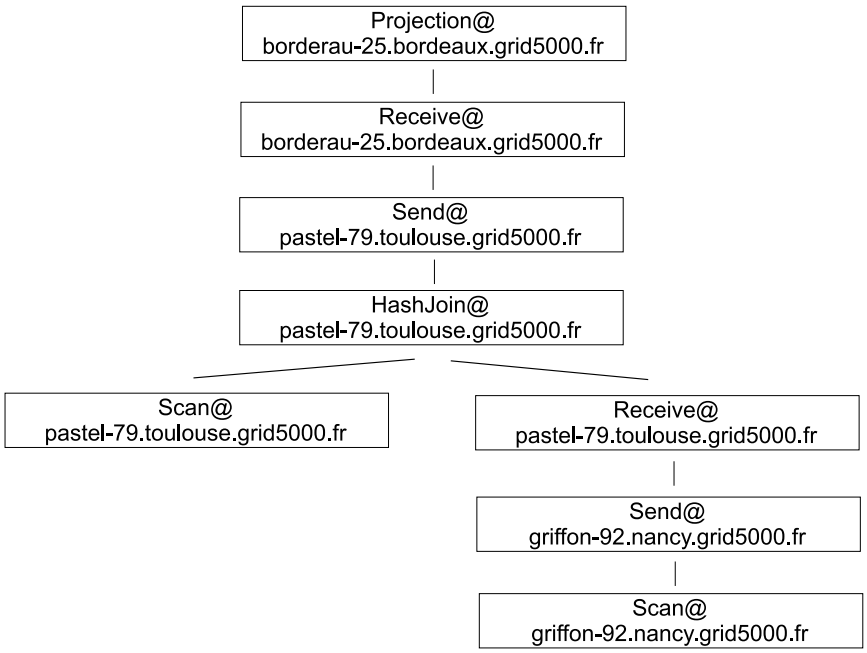


Figure 3.17: A possible result of the transformation of the logical plan of Figure 3.16 to a physical plan.

system [21], which were based on the nested relational model; in ViP2P, for the time being, we consider flat tuples only (indeed, the pattern language of ViP2P is the conjunctive, unnested restriction of the XAM language presented in[18]). In keeping with the standard iterator-based execution model [34], all operators support an interface based on the methods *open*, $hasNext$, *next* and *close*.

When all the operators are placed on the right peers and they are open, we will call the $hasNext()$ method of the root operator, which is $Projection@borderau-25.bordeaux.grid5000.fr$ in our example, in order to find out if there are any results. The $Projection$ will look at its temporary memory if there are any precalculated results. Since it is the first time that $hasNext()$ is being called, the temporary memory will be empty and the operator will ask tuples from its child (*Receive*) in order to apply the needed projection. The *Receive* is going to perform the same operation as the $Projection$ did. It will look at its temporary memory and if there are no tuples, it is going ask from *Send* new tuples. This operation is recursive and in the end, the leaf operators will be asked to scan their database files in order to feed the upper operator with tuples. After the end of these recursive calls, the $hasNext()$ method of the $Projection$ will reply true or false depending on the existence of results. In the case that $hasNext()$

replies true, we can call the $next()$ method of that operator in order to get the first result. For the second result, we will have to call the $hasNext()$ before calling the $next()$ method. This procedure is repeated since $hasNext()$ replies false, reassuring us that there are no more tuples for this physical plan.

Node information module This module stores all the information which is common and shared by different classes. These can be the IP address of the current ViP2P peer, the IP address of the bootstrap ViP2P peer, the communication ports of these peers, the directories where the views, the queries and the documents of the peer are located etc.

DB handler The task of the *DB handler* is to properly handle the Berkeley DB databases for which a peer is responsible. These databases are accessed when the peer wants to access a view's data for reading or writing. The handler keeps pointers to the databases that have been recently accessed and keeps these databases open for a specific time limit. This ensures that the databases are not open and closed for consecutive accesses which hurt the peer's performance.

RMI module Most of the inter-peer communication is being performed using the Java RMI. RMI module gives us the opportunity to call methods of remote ViP2P peers in order to interchange data. These data can be from tuples that are needed during physical plan execution to statistics that may need to be gathered.

Statistics module This module is responsible for monitoring various functionalities of the peer like view materialization, physical plan execution, query rewriting, view definition indexing & look-up etc. It can also communicate with the statistic modules of different peers in order to calculate global statistics of all the ViP2P network.

Acknowledgements ViP2P software development and its experimental evaluation have benefited from the help of Alin Ţilea, engineer since september 2008 in our group. The distributed digest was implemented also with the help of Julien Leblay, also an engineer from June to September 2009, working on ViP2P.

3.7 Related works

Our work is related to view-based XML query rewriting using, and to distributed XML data management.

Tree pattern query rewriting Rewriting an XPath query based on an XPath view has been studied in [24, 58]. More recent works have considered rewriting XPath queries using multiple views. View intersection is used to build rewritings in [29], and the DAGs we use in Section 3.3.1 recall their study, since ID equality join is akin to intersection. Our rewriting problem is complicated by the fact that our views have multiple attributes at various places in the view. Thus, we need joins, and we need to take into account how many times a tuple is multiplied by each extra join (as in the discussion around expansion and Figure 3.2). Also, we assume structural *id*s, which enable e.g. rewriting $a(b_{cont})$ out of a_{id} and $b_{id,cont}$, which [29] does not handle. The recent work of [51] takes structural *id*s a step further. They use XPath views (including wildcard nodes labeled $*$) where the return node always has *cont and* a powerful structural *id*, encapsulating the *id*s and labels of all its ancestors, up to the root. Thus, unlike us and [29], they may rewrite $a(b_{cont})$ using $b_{id,cont}$, simply by checking the $b.id$ for an a-labeled ancestor. We chose not to adopt such *id*s since they are rather lengthy, and their encoding relies on an NFA [38]. In our context, querying many documents, each of which would need an NFA, would significantly increase node *id* size, and thus, potentially data transfers. Rewriting is reduced in [51] to finding covers of the query leaves. Our rewritings need to cover the whole query, but we have proved in Section 3.3.1 a $|q|$ bound on the rewriting size, and polynomial complexity for the rewriting. In contrast, in [51] the rewriting size bound is $|\mathcal{V}|$ and the complexity is exponential in the number of query leaves. View embedding in the query is very expensive in the presence of $*$, thus [51] prunes views by building a view automaton at a cost of $\Sigma_{v\in\mathcal{V}}(|v|)$, and then running q through the automaton. We could also apply this; it would reduce our pruning cost (e.g., the embedding time in Figure 3.10) by a factor of $|q| - 1$.

Rewriting rich patterns with multiple attributes is studied in [19], under Dataguide constraints which strongly impact the algorithm, and without considering distribution. XQuery rewriting based on XQuery views is studied in [43], which establishes polynomial complexity for the XPath case.

From XQuery to tree patterns More generally, tree pattern views with multiple attributes allow answering more queries than XPath views (the presence and properties of node IDs also impacts the queries which may be answered, as shown above). For instance, $article(abstract_{id,cont}, author_{id,val})$ allows answering both $article(abstract_{cont})$ and $article(author_{val})$ (use π and duplicate elimination on some *id*s). Rich tree patterns, including optional and nested edges, come very close to capturing an XQuery dialect of nested

FLWR (for-where-return) expressions [20]. In particular, the mandatory part of a nested FLWR query is found in the for-where clauses of the outermost block, and is captured by a conjunctive pattern, as considered in this work.

[61] describes efficient XQuery evaluation techniques, for queries over documents whose URIs are known (without using views or a DHT). The benefits of such techniques are orthogonal - and could be cumulated with - those of using pre-computed view results, as we advocate in this work.

Distributed XML processing Closest to our work are techniques for indexing and querying XML in DHT networks [33, 26, 49, 5]. Each of these works uses a specific single XML indexing strategy, whereas we propose more flexible views, which can be better tailored to the query needs. View definitions are indexed on a DHT in [48], but they consider RDF data and rewritings based on only one view.

3.8 Conclusion

The efficient management of large XML corpora in structured P2P networks requires the ability to deploy data access support structures, which can be tuned to closely fit application needs. We have presented the VIP2P approach for building and maintaining structured materialized views, and processing peer queries based on the existing views in the DHT network. Using DHT views adds the cost of a view definition lookup, but pre-computed views can strongly reduce query evaluation times. We have characterized the complexity of rewriting conjunctive tree pattern queries with attributes, using materialized views, and we have compared several algorithms for view-based query rewriting; DFR seems to be the most useful. We studied several view indexing strategies and associated complete view lookup methods. The LPI method seems best, due to its low cost both in DHT messages involved in indexing and lookup, and to its good precision.

Chapter 4

Conclusion

In this thesis, we have worked on the problem of efficient data management which is becoming more and more demanding now-days. Our contribution to this research area is an optimizer for the AXML platform and a platform that allows us to exploit materialized views deployed in the DHT network independently by the peers, to answer an interesting dialect of tree pattern queries.

In Chapter 2, we have thoroughly presented our proposal to the AXML document optimization problem and we have seen how OptimAX can be used in various demanding real life applications. Many avenues for future work exist. In the current AXML platform, OptimAX is called only once before each document's complete evaluation. It is interesting to see how repeated optimization and evaluation steps can be combined. Moreover, it would have been useful if OptimAX would cooperate with a peer-to-peer monitoring system, such as the one presented in [13]. With accurate and up-to-date statistics, OptimAX' estimations will be closer to the real evaluation costs.

In Chapter 3, we have presented a full-fledged platform allowing first, to manage tree pattern XML subscriptions in a dynamic DHT network, and second, to rewrite an interesting dialect of tree pattern queries using materialized views. We have analyzed the view materialization, the query rewriting and view lookup procedures and we have shown the scalability of our platform by doing experiments on a network of computers placed in various research centers in France. Many avenues for future work exist. To efficiently handle very large views, we could employ horizontal view fragmentation, which would parallelize query execution, as was done for the DHT index in [5]. Collaborative view recommendation is a next step; algorithms for the centralized case start to appear [52]. Also, we are currently extending the view pattern language presented here with value joins, to handle queries over XML documents with RDF annotations. A demonstration of AnnoVIP, a follow-up on ViP2P focusing on the scalable management of XML docu-

ments with RDF annotations in DHT platforms, will be presented at ICDE 2010 [36].

To conclude, this thesis has been an exciting experience and discovery of areas which were unknown to me in the past. I saw how a theoretical work such as [7] is formalized and extended. Furthermore I saw how an optimizer is built using these principles [11]. This thesis has given me the opportunity to see how our work can be used in large R&D projects such as WebContent, on which many laboratories and companies have worked. Last but not least, a very interesting experience has been the development of the ViP2P platform. It has been the first time that I participated from the beginning in the development of a fully fledged peer-to-peer platform which in its current state has around 60,000 lines of code. A fascinating experience were the scalability experiments of the ViP2P platform. Testing a peer-to -peer platform in a nation-wide scale, using 250 different computers (1000 ViP2P peers) is challenging. In addition, it shows problems that can not be encountered during smaller tests, carried on small - local clusters.

Bibliography

[1] 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. In *VLDB (demo)*, 2008.

[2] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *SIGMOD*, 2004.

[3] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *VLDB (demo)*, 2002.

[4] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.

[5] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[6] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and Querying Peer-to-Peer Warehouses of XML Resources. In *ICDE (demo)*, 2005.

[7] S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *EDBT*, 2006.

[8] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: optimizing distributed continuous queries (demo). Journées Francophones en Bases de Données Avancées (BDA), 2007.

[9] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: Efficient Support for Data-Intensive Mash-Ups (demo). In *ICDE*, 2008.

[10] S. Abiteboul, I. Manolescu, and S. Zoupanos. Optimax: optimisation d'applications distribuées activexml. Journées Francophones en Bases de Données Avancées (BDA), 2008.

[11] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: Optimizing distributed ActiveXML applications. In *ICWE*, 2008.

[12] S. Abiteboul and B. Marinoiu. Distributed monitoring of peer-to-peer systems. In *WIDM*, 2007.

[13] S. Abiteboul, B. Marinoiu, and P. Bourhis. Distributed monitoring of peer-to-peer systems. In *ICDE (demo)*, 2008.

[14] S. Abiteboul, T. Milo, and O. Benjelloun. Regular rewriting of active XML and unambiguity. In *PODS*, 2005.

[15] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, 2002.

[16] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[17] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[18] A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[19] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured Materialized Views for XML Queries. In *VLDB*, 2007.

[20] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-Based Identification of Tree Patterns in XQuery. In *FQAS*, 2006.

[21] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the right storage for your XML application. In *VLDB*, pages 1330–1333, 2005.

[22] Apache Axis2. http://ws.apache.org/axis2/.

[23] ActiveXML home page. Available at http://www.activexml.net.

[24] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *VLDB*, 2004.

[25] S. Benbernou, X. He, and M. Said-Hacid. Implicit Service Calls in ActiveXML Through OWL-S. In *ICSOC*, 2005.

[26] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.

[27] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, 2004.

[28] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[29] B. Cautis, A. Deutsch, and N. Onose. XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency. In *WebDB*, 2008.

[30] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[31] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, 2003.

[32] F. Dragan, G. Gardarin, and L. Yeh. PathFinder: Indexing And Querying XML Data in a P2P System. In *WTAS*, 2006.

[33] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[34] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[35] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[36] K. Karanasos and S. Zoupanos. Viewing a world of annotations through AnnoVIP. In *ICDE (demo)*, 2010. To appear.

[37] D. Kossmann. The State of the art in distributed query processing. *ACM Comput. Surv.*, 32(4), 2000.

[38] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *VLDB*, 2005.

[39] I. Manolescu and S. Zoupanos. Vues matérialisées xml pour les entrepôts de données pair-à-pair. Journées Francophones en Bases de Données Avancées (BDA), 2009.

[40] I. Manolescu and S. Zoupanos. XML materialized views in P2P. DataX workshop (not in the proceedings), 2009.

[41] Monetdb: open source database system. monetdb.cwi.nl.

[42] A. Muscholl, T. Schwentick, and L. Segoufin. Active Context-Free Games. In *STACS*, 2004.

[43] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[44] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, 2004.

[45] G. Ruberg and M. Mattoso. XCraft: Boosting the Performance of Active XML Materialization. In *EDBT*, 2008.

[46] N. Ruberg, G. Ruberg, and I. Manolescu. Towards Cost-based Optimizations for Data-Intensive Web Service Computations. In *SBBD*, 2004.

[47] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

[48] L. Sidirourgos, G. Kokkinidis, T. Dalamagas, V. Christophides, and T. K. Sellis. Indexing views to route queries in a PDMS. *Distributed and Parallel Databases*, 23(1), 2008.

[49] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *CoopIS*, 2005.

[50] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[51] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *ICDE*, 2008.

[52] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz. Materialized View Selection in XML Databases. In *DASFAA*, 2009.

[53] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[54] N. Travers, T.-T. Dang-Ngoc, and T. Liu. TGV: A Tree Graph View for Modeling Untyped XQuery. In *DASFAA*, 2007.

[55] P. Valduriez and T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[56] W3C. WSDL: Web Services Definition Language 1.1.

[57] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007.

[58] W. Xu and M. Ozsoyoglu. Rewriting XPath Queries Using Materialized Views. In *VLDB*, 2005.

[59] eXist: Open Source Native XML Database. exist.sourceforge.net.

[60] WebContent, the Semantic Web platform (RNTL project). www.webcontent.fr.

[61] Y. Zhang, N. Tang, and P. A. Boncz. Efficient Distribution of Full-Fledged XQuery. In *ICDE*, 2009.