UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD (ED 427)

Laboratoire de Recherche en Informatique (LRI)

***DISCIPLINE** INFORMATIQUE*

THÈSE DE DOCTORAT

presentée en vue d'obtention du titre de docteur

par **Asterios KATSIFODIMOS**

Scalable View-based Techniques for Web Data : Algorithms and Systems

Directeur de thèse :	Ioana Manolescu	Inria Saclay and Université de Paris-Sud
Composition du jury :		
Rapporteurs :	Yanlei Diao	University of Massachusetts Amherst, U.S.A.
	Philippe Rigaux	Conservatoire National des Arts et Métiers
Examinateurs :	Alain Denise	Université de Paris-Sud
	Patrick Valduriez	Inria Sophia Antipolis
	Vasilis Vassalos	Athens University of Economics and Business

Résumé

"Techniques efficaces basées sur des vues matérialisées pour la gestion des données du Web: algorithmes et systèmes"

Asterios Katsifodimos

Le langage XML, proposé par le W3C, est aujourd'hui utilisé comme un modèle de données pour le stockage et l'interrogation de grands volumes de données dans les systèmes de bases de données. En dépit d'importants travaux de recherche et le développement de systèmes efficace, le traitement de grands volumes de données XML pose encore des problèmes des performance dus à la complexité et hétérogénéité des données ainsi qu'à la complexité des langages courants d'interrogation XML.

Les vues matérialisées sont employées depuis des décennies dans les bases de données afin de raccourcir les temps de traitement des requêtes. Elles peuvent être considérées les résultats de requêtes pré-calculées, que l'on réutilise afin d'éviter de recalculer (complètement ou partiellement) une nouvelle requête. Les vues matérialisées ont fait l'objet de nombreuses recherches, en particulier dans le contexte des entrepôts des données relationnelles.

Cette thèse étudie l'applicabilité de techniques de vues matérialisées pour optimiser les performances des systèmes de gestion de données Web, et en particulier XML, dans des environnements distribués. Dans cette thèse, nos apportons trois contributions.

D'abord, nous considérons le problème de la sélection des meilleures vues à matérialiser dans un espace de stockage donné, afin d'améliorer la performance d'une charge de travail des requêtes. Nous sommes les premiers à considérer un sous-langage de XQuery enrichi avec la possibilité de sélectionner des nœuds multiples et à de multiples niveaux de granularités. La difficulté dans ce contexte vient de la puissance expressive et des caractéristiques du langage des requêtes et des vues, et de la taille de l'espace de recherche de vues que l'on pourrait matérialiser. Alors que le problème général a une complexité prohibitive, nous proposons et étudions un algorithme heuristique et démontrer ses performances supérieures par rapport à l'état de l'art.

Deuxièmement, nous considérons la gestion de grands corpus XML dans des réseaux pair à pair, basées sur des tables de hachage distribuées. Nous considérons la plateforme ViP2P dans laquelle des vues XML distribuées sont matérialisées à partir des données publiées dans le réseau, puis exploitées pour répondre efficacement aux requêtes émises par un pair du réseau. Nous y avons apporté d'importantes optimisations orientées sur le passage à l'échelle, et nous avons caractérisé la performance du système par une série d'expériences déployées dans un réseau à grande échelle. Ces expériences dépassent de plusieurs ordres de grandeur les systèmes similaires en termes de volumes de données et de débit de dissémination des données. Cette étude est à ce jour la plus complète concernant

une plateforme de gestion de contenus XML déployée entièrement et testée à une échelle réelle.

Enfin, nous présentons une nouvelle approche de dissémination de données dans un système d'abonnements, en présence de contraintes sur les ressources CPU et réseau disponibles; cette approche est mise en oeuvre dans le cadre de notre plateforme Delta. Le passage à l'échelle est obtenu en déchargeant le fournisseur de données de l'effort de répondre à une partie des abonnements. Pour cela, nous tirons profit de techniques de réécriture de requêtes à l'aide de vues afin de diffuser les données de ces abonnements, à partir d'autres abonnements. Notre contribution principale est un nouvel algorithme qui organise les vues dans un réseau de dissémination d'information multi-niveaux ; ce réseau est calculé à l'aide d'outils techniques de programmation linéaire afin de passer à l'échelle pour de grands nombres de vues, respecter les contraintes de capacité du système, et minimiser les délais de propagation des information. L'efficacité et la performance de notre algorithme est confirmée par notre évaluation expérimentale, qui inclut l'étude d'un déploiement réel dans un réseau WAN.

Mots Clefs: XML, données du web, vues matérialisées, optimisation des requêtes, selection des vues, systèmes d'abonnements, gestion des données.

Abstract

"Scalable View-based Techniques for Web Data: Algorithms and Systems"

Asterios Katsifodimos

XML was recommended by W3C in 1998 as a markup language to be used by device- and system-independent methods of representing information. XML is nowadays used as a data model for storing and querying large volumes of data in database systems. In spite of significant research and systems development, many performance problems are raised by processing very large amounts of XML data.

Materialized views have long been used in databases to speed up queries. Materialized views can be seen as precomputed query results that can be re-used to evaluate (part of) another query, and have been a topic of intensive research, in particular in the context of relational data warehousing.

This thesis investigates the applicability of materialized views techniques to optimize the performance of Web data management tools, in particular in distributed settings, considering XML data and queries. We make three contributions.

We first consider the problem of choosing the best views to materialize within a given space budget in order to improve the performance of a query workload. Our work is the first to address the view selection problem for a rich subset of XQuery. The challenges we face stem from the expressive power and features of both the query and view languages and from the size of the search space of candidate views to materialize. While the general problem has prohibitive complexity, we propose and study a heuristic algorithm and demonstrate its superior performance compared to the state of the art.

Second, we consider the management of large XML corpora in peer-to-peer networks, based on distributed hash tables (or DHTs, in short). We consider a platform leveraging distributed materialized XML views, defined by arbitrary XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. This thesis has contributed important scalability oriented optimizations, as well as a comprehensive set of experiments deployed in a country-wide WAN. These experiments outgrow by orders of magnitude similar competitor systems in terms of data volumes and data dissemination throughput. Thus, they are the most advanced in understanding the performance behavior of DHT-based XML content management in real settings.

Finally, we present a novel approach for scalable content-based publish/subscribe (pub/sub, in short) in the presence of constraints on the available computational resources of data publishers. We achieve scalability by off-loading subscriptions from the publisher, and leveraging view-based query rewriting to feed these subscriptions from the data accumulated in others. Our main contribution is a novel algorithm for organizing subscriptions in a multi-level dissemination network in order to serve large numbers of subscriptions, respect capacity con-

straints, and minimize latency. The efficiency and effectiveness of our algorithm are confirmed through extensive experiments and a large deployment in a WAN.

Keywords: XML, Web data, materialized views, query optimization, view selection, publish/subscribe, data management.

Acknowledgments

I offer my sincerest gratitude to my advisor, Ioana Manolescu, who has helped me throughout my thesis with her trust, experience, knowledge and patience (lots of it). Ioana has been more than an advisor for the last 4 years. She has been a supportive friend, a great discussion partner and a caring mentor. She has never refused to help me, even if this required her working inside a hammam! Ioana has stayed up with me until the last minute of each and every one of our deadlines. She showed me that work should be done the hard way and that there are no shortcuts to get around it. This thesis would not be possible without her insistence, help and guidance.

I would like to thank Yanlei Diao and Philippe Rigaux that have thoroughly read my thesis and provided great feedback. Also my thesis committee, Alain Denise, Patrick Valduriez and Vasilis Vassalos for doing me the honor of being present in my defense. Vasilis has also given me great guidance and help in the second year of my thesis, with his fruitful comments and discussions. Finally, I would like to thank Cédric Bentz for his suggestions in Linear Programming modeling and Yannis Manoussakis for his guidance on the NP-Hardness of one of our problems.

I am also very grateful to my previous advisor Prof. Marios Dikaiakos from my home university, the University of Cyprus. He is the one that first got me into research, encouraged me to start a PhD and inspired me as much as possible. In addition, I would like to thank Prof. George Pallis. He was the one to guide me in my very first steps in research and the one to forward to me that DBWorld email with which Ioana was announcing a PhD scholarship in her team.

I was very lucky to be surrounded by a cheerful group of colleagues. The Oak team (formerly Gemo and Leo) has always been a pool of great people. Danai Symeonidou, Julien Leblay, Stamatis Zampetakis, Katerina Tzompanaki, Alexandra Roatis and Andrés Aranda Andújar have been a great companion in our everyday life in the lab. I am grateful to Spyros Zoupanos and Alin Tilea for their work on the ViP2P platform - almost all parts of this thesis are based on it.

I need lots of space to thank Konstantinos Karanasos, a true friend, travel buddy, fellow dining philosopher and work colleague; I will try to keep it short. In our long, late night discussions, we have covered every aspect of human existence. From Konstantinos I learned two very important things about science and life: that a convincing example does not count as a proof and that sometimes aubergines have their place in a velouté (but they still badly affect its texture). Apart from a friend, Konstantinos has also given me lots of advise during my thesis, he listened to me patiently and put my thoughts in order. The last part of this thesis started from one of his ideas, and finished with his invaluable help.

I would also like to thank Jesús Camacho-Rodríguez. Our time together (either in front of a beer pitcher or in the lab) included lots of fun, interesting discussions. Our legendary road trips with him and Konstantinos were long, really long, but so much fun to do! His quality as a person and his hard working attitude really

inspire me. I hope he reads this paragraph as he still owes me a (whole) pata negra jamon.

I owe a lot of my passion for photography to Yannis Katsis, a very good friend and gastronomical buddy. My first two years in Paris would never be the same without him. I wish we could meet in the same city in the future for another walk and some dinners.

Federico Ulliana has always stimulated the most interesting of discussions, whether this involved people, science or politics. After work pints could never be more interesting. I would also like to thank him for explaining some obscure complexity numbers to me.

I had lots of fun and great culinary nights with Penny Karanasou, Zoi Kaoudi and Despoina Trivela. We happened to get closer in the last year of my PhD and they gave me a lot of support and careless moments. I will miss them a lot!

Isabelle van Sloten has a very big merit on this thesis. Always believing in me, she has been my source of strength, confidence and energy. She never accepted my excuses for procrastinating and she always insisted that I sleep early and get up early the next day and work (I'm still struggling to follow her advice). Writing a thesis is a long, lonely journey; having Isabelle with me made it a pleasant walk.

Finally I thank my friends Asterios, Miltos, Pavlos and Giannis; we spent lots of time discussing and (loudly) arguing about every little thing on earth. They have always set the bar high and showed me that no difficulty is a reason to fail.

I dedicate this thesis to my parents. I owe a lot to my father, the oldest of my best friends. He defined the way I see the world today. Fixing my motorbike together with him triggered my passion for experimentation and systems engineering. My mother gave me trust, love, care and strong principles. She never complained when it took me long to call her. Finally I would like to thank my grandfather for showing me the simple side of life and for being positive about everything. It is an honor to have him in my thesis defense, after a long trip from Greece, in his late 80s.

Contents

List of Algorithms

Chapter 1

Introduction

The Web is increasingly acknowledged as the single richest and most diverse source of data, whether unstructured, that is, organized in Web pages, or structured, with rigid (tabular) or heterogeneous (tree-like) structure. Web data is not only searched, but also parsed to extract phrase structure, queried to search for precise answers, mined for knowledge, and processed at large scale by harnessing large-scale parallel processing platforms [AMR+12]. Vast amounts of data is produced every day such as governmental data [1], maps [2] and global economy reports [3] to name a few. Storing, organizing, and sustaining long-term infrastructures as well as building systems for significant amounts of data are very challenging tasks. Thus, the urgency for new tools and systems that will enable scalable distributed data management, has never been bigger.

1.1 Context: Web Data Management

In the last decade XML has established itself as standard data model for the exchange of data and organizations increasingly employ XML within their information systems. XML is becoming more than a document markup language or a data exchange format: applications are built with XML as their core data model taking advantage of XML's flexibility to describe schema-less, semistructured data. Following this trend, commercial database systems, traditionally supporting the relational model, now broadly support XML (e.g., IBM DB2 [BCH+06], Microsoft SQL Server [PCS+04], Oracle [LM09], etc.).

These systems and many other software tools developed for working with XML (content management systems, Web services, etc.) have made vast improvements in terms of performance and scalability since the early days of XML. However, as the data volumes keep increasing and the complexity of the data and queries follows similar trends, performance-oriented optimizations are still in great need.

1. `http://www.data.gov`, `http://www.data.gouv.fr`
2. `http://planet.openstreetmap.org`
3. `http://datacatalog.worldbank.org`

The interest in XML processing tools and primitives is also due to their applicability to other document-oriented, semistructured data formats, such as JSON (Javascript Object Notation) [JSO].

1.2 Motivation

Materialized views have long been used in database systems in order to expedite database queries [Hal01]. Materialized views can be seen as precomputed query results that can be re-used to evaluate (part of) another query, and have been a topic of broad research in the database community, in particular in the context of relational data warehousing [GM99a]. In this thesis, we investigate the applicability of techniques based on *materialized views* to optimize the performance of Web data management systems. More specifically, we consider XML data and queries in distributed settings.

In the context of XML data management, distributed systems are of significant interest for two reasons. First, as organizations interact more and more, sharing and consuming one another's information, it is often the case that (XML) data is produced independently by several distributed sources. Second, a distributed system can accommodate data volumes going far beyond the capacity of a single machine or cluster.

The work presented in this thesis aims to show that materialized views over XML data can be successfully used, in particular within distributed systems, to enable efficient sharing and querying of large volumes of Web data.

1.3 Thesis Outline

Aiming at the efficient view-based Web data management, this thesis considers two main problems: the view selection problem for XML query workloads and the distributed view-based XML data management. Below we provide an overview of the organization of this thesis.

Chapter 2 provides the necessary background to follow the rest of the thesis, including notably XML data management and problems related to view-based data management. As we will see in the chapters that follow, this thesis depends on view-based query rewriting, view size estimations, algebraic rewriting plans and a cost-based optimizer. To this end, we use an existing rewriting algorithm [MKVZ11, Kar12] while we have implemented our own view size estimations and a simple cost-based optimizer. All these details can be found in Chapter 2.

Chapter 3 considers the problem of choosing the best views to materialize within a given space budget in order to improve the performance of a query workload. The contributions of this chapter are the following:

— The work presented in this chapter has been the first to formalize and address the view selection problem for queries and views expressed in a rich

subset of XQuery, namely tree patterns with value joins.
— We analyze the space of potential candidate views and present several effective candidate pruning criteria.
— While the general problem has prohibitive complexity, we propose and study a heuristic algorithm and experimentally demonstrate its superiority compared to the state of the art.

The work described in this chapter has been published in [KMV12], while an earlier version had been briefly outlined in [CRKMR10].

Chapter 4 explores the case of distributed XML materialized views and presents ViP2P (standing for *Vi*ews in *P*eer-to-*P*eer), a distributed platform for sharing XML documents based on a structured P2P network infrastructure (DHT). At the core of ViP2P stand distributed materialized XML views, defined by arbitrary XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. Views in ViP2P can be proposed automatically (by the algorithm of Chapter 3) or manually, by the peers themselves. The contributions of Chapter 4 can be summarized as follows:

— We present a complete architecture for query evaluation, both in continuous (subscription) and in snapshot mode. This architecture enables the efficient dissemination of answers to tree pattern queries (expressed in an XQuery dialect) to peers that are interested in them, regardless of the relative order in time between the data and the subscription publication.
— We have fully implemented our architecture, on top of the FreePastry [Fre] P2P infrastructure and present a comprehensive set of experiments performed in a WAN, demonstrating ViP2P's superiority over the state of the art.
— The ViP2P platform scales to several hundreds of peers and hundreds of GBs of XML data, both unattained in previous works.

This chapter is an extension of the work published in [KKMZ12] and closely follows a technical report [KKMZ11].

Chapter 5 presents Delta, a novel approach for scalable content-based publish/subscribe in the presence of constraints on the available computational resources of the data publisher. Unlike ViP2P (Chapter 4) where views/subscriptions are filled directly by the publishers of documents, in Delta scalability is achieved by offloading some subscriptions from the publisher, and leveraging view-based query rewriting to feed these subscriptions from the data accumulated in others. Our main contributions are:

— We are the first that consider the design and implementation of a publish/subscribe platform that is based on materialized views.
— We present a novel algorithm for organizing views in a multi-level dissemination network, exploiting view-based rewriting and powerful integer linear programming capabilities to scale to many views, respect capacity constraints, and minimize latency.
— We provide a full implementation of our architecture and we present exten-

sive experiments validating the efficiency and effectiveness of our algorithm that are confirmed through experiments and a large deployment in a WAN.

The results of this chapter are part of an article submitted for publication on May 1st, 2013 and which is currently being reviewed.

Chapter 6 provides a summary of the thesis and discusses directions for future work.

Focused comparisons to the state of the art To facilitate the reading of this thesis, descriptions of pointed areas of related work are delegated to the chapters to which they most naturally relate. Thus, Chapter 3 presents the state of the art concerning materialized view selection. Chapter 4 discusses P2P management platforms and finally, Chapter 5 provides more details on publish/subscribe systems.

Other Scientific Activity During the Thesis In parallel with the works described in this thesis, we have continued and finalized a collaboration that started prior to this PhD thesis with Prof. Marios Dikaiakos and Prof. George Pallis from University of Cyprus (2009-2011). The work focused on Minersoft, a fully functional search engine for software resources installed in infrastructures like computing Grids and Clouds. Minersoft visits remote infrastructures, crawls their file systems, indexes content and allows keyword-based queries to its users. The results of this work have been published in [PKD10, DKP12].

Finally, we have collaborated with Dr. Jean-Daniel Fekete from Inria Saclay, and Prof. Cécile Germain-Renaud from Université Paris-Sud (2009-2012). We have implemented LogDice, a data visualization tool based on GraphDice [BCD+10]. LogDice aims to achieves interactive, visual exploration of the spatio-temporal structure of data access and data sharing in e-Science social networks. We have performed log file analysis, developed social network graph aggregation algorithms and designed a relational database schema for social network graph queries. Early results of this work have been presented in [KFCGR10].

Chapter 2

XML Databases, Views and Rewritings

XML [W3C08] was recommended by W3C in 1998 as a markup language to be used by device- and system-independent methods of representing information. Since then, it has gone far beyond the original intentions of W3C and is nowadays used as a data model for data exchange supporting various applications. XML databases have been a popular subject of research by the database community.

Materialized views, one of the oldest topics in database research, have been proposed as a means to expedite processing of XML queries over XML databases. Various research issues arose when materialized views met XML databases. Two of the most important of them are: the problem of storing and retrieving XML materialized views and the problem of optimizing XML queries using views.

In this chapter, we first discuss the XML data model (Section 2.1). We then move to presenting the main problems that are related to materialized views as well as the current state of the art in answering XML queries using views (Section 2.2). Finally, in Section 2.3, we show how one can estimate the cost of query plans that are evaluated over materialized views and how those plans can be optimized for lower evaluation costs.

2.1 The XML Data Model

XML is the de facto standard for the representation of structured information on the Web. It stands for eXtensible Markup Language and is a semistructured data model that was designed to be generic, platform-independent and self-descriptive and serves as a uniform data exchange format between applications in the WWW.

In this section, we first overview the XML data model (Section 2.1.1) and present the standard XML query languages (Section 2.1.2). Then, we discuss existing approaches for storing and retrieving XML data (Section 2.1.3) and finally, we present the query language that is studied in this thesis (Section 2.1.4).

2.1.1 Data Model

XML is organized in documents. In each XML document there is exactly one root and each node (other than the root) has exactly one parent. Each node in an XML tree has a label and can have multiple children. Each XML node can be an element, a text node or an attribute. Each child of an element node may contain a list of (sub-)elements, text nodes and/or attributes. Note that attribute nodes do not have children. Moreover, the order in which the children of a node appear in an XML document matters.

XML Structuring and Typing An XML document is *well-formed* if it respects certain syntactic rules. However, those rules do not define anything specific about the structure of the document (e.g., what children nodes an element is allowed to have). For reasons of compatibility (e.g., two applications that try to communicate and should agree on a common vocabulary), it is necessary to define all the element and attribute names that may be used as well as what values an attribute may take, which elements can (or must) occur within other elements, etc.

There are two ways of defining the structure and enforcing the typing of XML documents: the Document Type Definition (DTD) [W3C04] an older and more restricted way, and XML Schema [XML], which offers extended possibilities, mainly for the definition of data types and value constraints. An XML document is *valid* against a schema, if it respects the constraints that are specified in the given schema.

2.1.2 Standard XML Query Languages

Since the appearance of the XML data model, various XML query languages have been proposed for navigating and querying XML documents. XPath [W3C07a] and XQuery [W3C07b], both specified by the W3C, are the two most widely used and studied XML query languages. Currently, both languages rely on a common data model, namely the XML Query Data Model (XDM) [W3C07c]. What is more, XPath is a fragment of XQuery. A brief description of the two languages is provided below.

XPath is a domain specific language (DSL) and is used for extracting bits of an XML document using path expressions. XPath is popular for its simplicity and conciseness. It uses several axes in its path expressions, in order to facilitate the navigation in XML documents (e.g., child, descendant, ancestor and sibling axes). The current version (XPath 2.0) extended the data model of the previous version, adding features such as intersection and complementation operators, as well as iteration capabilities. [BK08, tCM07] provide formal results (complexity, relation to first-order logic etc.) of the XPath versions 1.0 and 2.0.

XQuery syntactically contains XPath as part of its syntax. XQuery was designed for use with XML documents, however, it is *functional* and allows the use of user defined functions (UDFs). It is thus a Turing-complete language and can be con-

sidered a general-purpose language. XQuery overcomes many of the limitations of XPath at the expense of introducing complexity: its rich semantics significantly increase the complexity of query evaluation and optimization. XQuery provides a feature called FLWOR expressions. The FLWOR acronym stands for *for*, *let*, *where*, *order by* and *return*. All FLWOR expressions start with a for or a let expression and end with a return expression. FLOWR expressions enable XQuery to:

— perform iterations (for)
— define variables (let)
— order results (order by)
— impose constraints and perform joins (where) and finally
— construct custom formatted data (return).

The expressiveness and complexity for various fragments of XQuery is studied in [BK09a].

2.1.3 XML Data Management Systems

An intuitive way of querying XML data is based on the idea that the entire XML document is loaded into main memory in the form of a tree and then XML queries are evaluated over this tree. Although such a main-memory implementation may seem straightforward, it is feasible for querying only small XML documents, and is inefficient for querying large XML data repositories.

Motivated by this, in the recent years, significant effort has focused in the development of high-performance XML database systems, that either use existing relational database systems (RDBMS, in short) or they implement "native" algorithms and systems specifically designed for XML and tree structured data.

RDBMS-based Approaches Many works have focused on employing RDBMS for the storage of XML documents. These works proposed techniques to map the semistructured data of XML documents to relational tables and then translate XML queries into relational ones and hand the optimization of such queries to the underlying relational query optimizer.

For instance, in [FK99], the key idea was to construct a table that stores the two end points (source, target) of each edge in the XML tree along with the type (element or attribute) and the value of the source node. Query evaluation for "/" queries was done by simple joins on the edge endpoints.

At the same time, [STZ+99], used knowledge from DTDs in order to perform the mapping of XML to tables, whereas in [DFS99] the mapping was done given information on the expected data and queries. Several other approaches were also proposed later, e.g., [FM00, BGvK+06].

Commercial RDBMS also provide support for XML, such as IBM DB2 [BCH+06], Microsoft SQL Server [PCS+04] and Oracle [LM09]. Moreover, SQL/XML was proposed in 2003 is an extension to the SQL specification, which defines the use of XML in conjunction with SQL. The XML data type was introduced, as well as several routines, functions, and XML-to-SQL data type mappings to support manipulation and storage of XML in a SQL database.

Native XML Approaches Along with relational approaches, native XML systems have also been developed, such as the Lore [IHW01] and the Tukwila [MW99] systems. Nowadays, eXist [eXi], BaseX [BSX] and Saxon [Sax] are three of the most widely used open source XML databases.

To overcome the limitations of early RDBMS-based approaches, novel ID schemes that capture information about the position of each node in the XML document (such as the start and end point of the node, its depth in the tree, etc.) were proposed [TVB+02, LLCC05]. Such IDs can be used to speed up query evaluation for "//" queries (e.g., efficiently determine whether a node is the parent/ancestor of another) and have been used by XML-specific join algorithms [ZND+01, AKJP+02, BKS02].

More specifically, a variation of the traditional merge join algorithm was first proposed in [ZND+01]. It was then improved by the tree-merge and stack-tree structural join algorithms [AKJP+02]. The improvement was based on the idea that the XML data can be stored in inverted lists for each tag in the XML document. Each inverted list stores the positions (Start, End, Level) in the form of structural IDs, of all elements with the same tag name, sorted by the element's start position. That way, the structural join can be performed through a single pass over its inputs.

Both [ZND+01, AKJP+02] need to apply a structural join for each edge (relationship between tags) of the XML query. To solve this problem and further optimize query evaluation, the holistic twig join algorithm [BKS02] builds the result of a query in a single pass over all the input relationships in parallel, eliminating the need for storing and sorting intermediate results.

2.1.4 XML Query Dialect and Tree Pattern Formalism

The query language, the rewriting algorithm and the algebraic plans considered in this thesis, have been extensively studied in a paper [MKVZ11] as well as in a recent thesis [Kar12]. We recall these fundamental notions here, as background and to make this thesis self-contained. The algorithms described in subsequent chapters build the contributions of this thesis upon these notions.

In this section, we first describe the XQuery dialect we consider throughout this thesis in Section 2.1.4.1 and then, in Section 2.1.4.2 we present a joined tree pattern formalism, conveniently representing queries.

2.1.4.1 XQuery Dialect

Let $\mathcal{L}$ be a set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language [MS04]. In this thesis, we consider views and queries expressed in the XQuery dialect described in Figure 2.1. In the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the bindings of a previously-

1	q :=	for $absVar$ (, ($absVar$|$relVar$))* (where $pred$ (and $pred$)*)? return ret	
2	$absVar$:=	x_i in doc(uri) p	
3	$relVar$:=	x_i in x_j p x_i	// x_j *introduced before*
4	$pred$:=	string(x_i) = (string(x_j) | c)	
5	ret :=	$\langle l \rangle$ $elem$* $\langle /l \rangle$	
6	$elem$:=	$\langle l_i \rangle$\{ (x_k | id(x_k) | string(x_k)) \}$\langle /l_i \rangle$	

Figure 2.1: Grammar for views and queries.

introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i, either with the string value of another variable x_j, or with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause:

— x_k denotes the full subtree rooted at the binding of x_k;
— string(x_k) is the string value of the binding;
— id(x_k) denotes the ID of the node to which x_k is bound.

There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of n's text descendants [w3c07d].

Therefore, string(x_i) is very likely smaller than x_i's content, but it holds less information. Second, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones.

We assume structural IDs, the most prominent of which are proposed in [TVB+02, LLCC05]. In a nutshell, structural IDs serve for comparing the IDs of two XML nodes and determining whether one is a parent (or ancestor) of the other. Our XQuery dialect distinguishes structural IDs, value and contents, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

For illustration, Figure 2.2 shows a query q in our XQuery dialect, as well as two views v_1 and v_2. The parent custom function returns true if and only if its inputs are node IDs, such that the first identifies the parent of the second. Moreover, as usual in XQuery, the variable bindings that appear in the where clauses imply the string values of these bindings (e.g. \$e='ACM' is implicitly converted to string(\$e)='ACM').

q	for $p in doc("confs")//confs//SIGMOD/paper, $y1 in $p/year,
	$a in $p//author[email], $c1 in $a/affiliation//country,
	$b in doc("books")//book, $y2 in $b/year, $e in $b/editor,
	$t in $b//title, $c2 in $b//country
	where $e='ACM' and $y1=$y2 and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{string($t)}⟨/tval⟩ ⟨/res⟩

v_1	for $p in doc("confs")//confs//paper, $a in $p/affiliation return ⟨v1⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨acont⟩{$a}⟨/acont⟩ ⟨/v1⟩

v_2	for $b in doc("books")//book, $c in $b//country, $e in $b/editor,
	$t in $b/title, $y1 in $b/year, $p in doc("confs")//SIGMOD/paper,
	$y2 in $p/year, $a in $p//author[email]
	where $e='ACM' and $y1=$y2
	return ⟨v2⟩ ⟨cval⟩{string($c)}⟨/cval⟩ ⟨tval⟩{string($t)}⟨/tval⟩
	⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨/v2⟩

r	for $v1 in doc("v1.xml")//v1, $p1 in $v1/pid, $af1 in $v1/aid,
	$c1 in $v1//acont//country, $v2 in doc("v2.xml")//v2,
	$c2 in $v2/cval, $t2 in $v2/tval, $p2 in $v2/pid, $a2 in $v2/aid
	where $p1=$p2 and parent($a2,$af1) and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{$v2}⟨/tval⟩ ⟨/res⟩

Figure 2.2: XQuery query, views, and rewriting.

2.1.4.2 Joined Tree Patterns

We use a dialect of joined tree patterns to represent views and queries. Formally, a tree pattern is a tree whose nodes carry labels from $\mathcal{L}$ and may be annotated with zero or more among: ID, val and $cont$ (corresponding to the ID, string value and full subtree seen in the previous section). A pattern node may also be annotated with a value equality predicate of the form $[= c]$ where c is some constant. The pattern edges are either simple for parent-child or double for ancestor-descendant relationships.

A joined tree pattern is a set of tree patterns, connected through value joins, which are denoted by dashed edges. For illustration, Figure 2.3 depicts the (joined) tree pattern representations of the query and views shown in XQuery syntax in Figure 2.2. In short, the semantics of an annotated tree pattern against a database is a list of tuples storing the ID, val and $cont$ from the tuples of database nodes in which the tree pattern embeds. The tuple order follows the order of the embedding target nodes in the database. The detailed semantics feature some duplicate elimination and projection operators (from the algebra we will detail next), in order to be as close to the W3C's XPath 2.0 semantics as possible. The only remaining difference is that tree patterns return tuples, whereas standard XPath/XQuery semantics uses node lists. Algebraic operators for translating between the two are by now well understood [MPV09]. The semantics of a joined tree pattern is the join of the semantics of its component tree patterns.

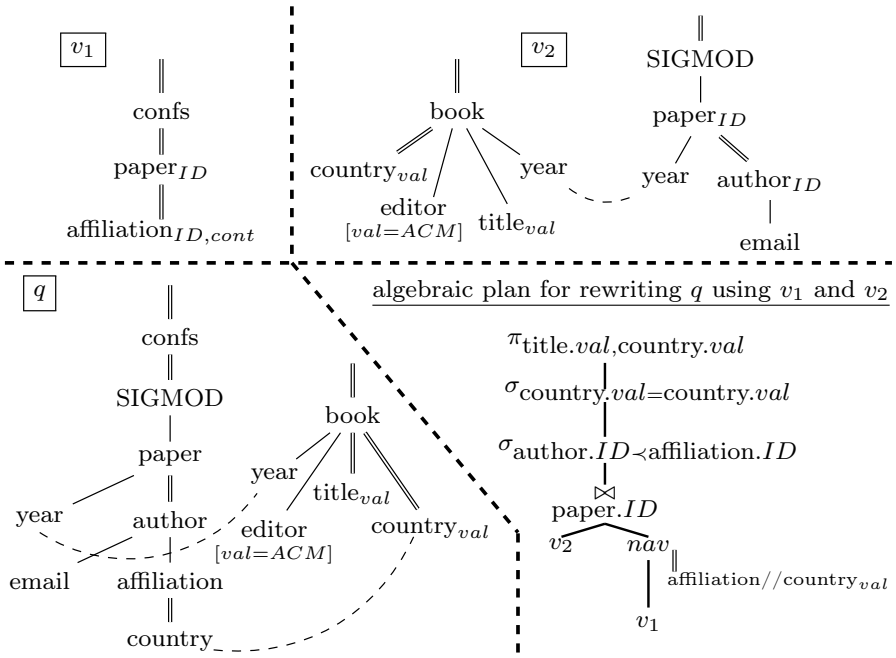


Figure 2.3: Pattern query and views, and algebraic rewriting.

Translating from the XQuery dialect presented earlier to the joined tree patterns is quite straightforward. The only part of the XQuery syntax *not* reflected in the joined tree patterns is the names of the elements created by the return clause. However, these names are not needed when rewriting queries based on views. Once a rewriting has been found, the query execution engine creates new elements out of the returned tuples of XML elements, values and/or identifiers, using the names specified by the original query, as explained in [SSB+00]. From now on, for readability, we will only use the joined tree pattern query representations of views and queries.

2.2 XML Materialized Views

A materialized view is a query defined over a database whose results (called the view extent) are computed and stored in the database. A materialized view is, thus, like a cached copy of the data that can be accessed quickly. Indexes can be built on materialized views, making query evaluation over materialized views typically faster than accessing the base data. This has made materialized views a prevalent tool for *query optimization*. However, whether the use of materialized views will result in a better or worse query evaluation time, depends on the query and the statistical properties of the database. It is up to the query optimizer to decide whether the evaluation will be done over materialized views or over the base data.

In order to be able to answer a query using a set of views, one has to *rewrite* the query into an equivalent rewriting expression. Rewriting expressions, or plans, typically imply an algebraic formalism describing their computations and can be further optimized to achieve lower evaluation costs. Optimizers typically require a cost model, able to quantify the resources needed for the execution of plans.

In what follows, we outline the basic notions and methods of using materialized views in Section 2.2.1. We then move to XML specific methods of answering XML queries using views. In Section 2.2.2 we present an algebraic formalism that describes the rewriting plans that we use in this thesis, and then we present the view-based query rewriting algorithm that we use. Finally, in Section 2.3 we present our cost estimation techniques and our algebraic plan optimizer.

2.2.1 Materialized Views Concepts and Core Problems

Materialized views have long been used in *data warehousing* as a means of data replication and abstraction. Data from multiple databases are merged into one global schema comprising materialized views. Such data replication has two main advantages:

1. data moves away from the online transactional database, making online analytical processing (OLAP) more performant over lock-free materialized views
2. data moves closer to the users (or applications) that use them.

Answering Queries Using Views Given a query q and a set of views V, *answering queries using views* (AQUV, in short) deals with the problem of how can q be answered using the views in V.

Answering queries using views has been extensively studied in the relational context [CKPS95, DPT99, GL01, DGL00]. A survey on answering queries using views is given in [Hal01]. This thesis depends on an algorithm for answering XML queries using views that is presented in the next section (Section 2.2.2).

The View Selection Problem Given a query workload Q, *view selection* is the problem of choosing a view set V to be materialized in order to answer the queries in a workload Q. Depending on the context, the view set V is selected, such that the query evaluation time (using the selected views), the view storage space and/or the view maintenance cost is minimized.

View selection has been extensively studied, especially in the context of data warehouses for SPJ queries [TS97] and OLAP queries [Gup97, HRU96, GM05]. Several formal results concerning the view selection problems are provided in [CHS02]. More recently the view selection problem was addressed in the context of RDF warehouses [GKLM10, GKLM12].

View selection has been addressed in the past for XPath [MS05, TYT+09]. This thesis extends the state of the art and addresses the problem of view selection for an XQuery dialect (Chapter 3).

2.2.2 XML View-based Rewriting and Logical Algebraic Plans

A rewriting is an XQuery query expressed in the same dialect as our views and queries, but formulated against XML documents corresponding to materialized views. For instance, the rewriting XQuery expression r in Figure 2.2 is an equivalent rewriting of the query q using the views v_1 and v_2 in the same Figure.

Logical Algebraic Plans An alternative, more convenient way to view rewritings is under the form of *logical algebraic plans*, or simply plans. Before presenting our plans, we introduce some useful logical operators. We denote by $\prec$ the parent comparison operator, which returns true if its left-hand argument is the ID of the parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the ancestor comparison operator. Observe that $\prec$ and $\prec\!\!\prec$ are only abstract operators here (we do not make any assumption on how they are evaluated, neither what kind of structural IDs are used).

We consider an algebra on tuple collections (as described in Section 2.1.4.1) whose main operators are:

1. **Scan** of all tuples from a view v, denoted $scan(v)$ (or simply v for brevity, whenever possible);
2. **Cartesian Product**, denoted $\times$;
3. **Selection**, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are tuple attributes, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$;
4. **Projection**, denoted π_{cols}, where $cols$ is the attributes list that will be projected;
5. **Navigation**, denoted $nav_{a,np}$. nav is a unary algebraic operator, parameterized by one of its input columns' name a, and a tree pattern np. The name a must correspond to a $cont$ attribute in the input of nav. Let t be a tuple in the input of nav, and $np(t.a)$ be the result of evaluating the pattern np on the XML fragment stored in $t.a$. Then, $nav_{a,np}$ outputs the tuples $\{t \underset{a}{\bowtie} np(t.a)\}$.

Figure 2.4 illustrates the functioning of nav on a sample input operator op. The parameters to this nav are $book_{cont}$ (the name of the column containing $\langle$book$\rangle$ elements), and the tree pattern $//author_{cont}$. The first tuple output by nav is obtained by augmenting the corresponding input tuple with a $author_{cont}$ attribute containing the single author-labeled child of the element found in its $book_{cont}$ attribute. The second and third nav output tuples are similarly obtained from the last tuple produced by op. Observe that the second tuple in op's output has been eliminated by the nav since it had no $\langle$author$\rangle$ element in its $book_{cont}$ attribute.

The algebra also includes the join operator, defined as usual, sort and duplicate elimination. For illustration, in the bottom right of Figure 2.3, we depict the algebraic representation of the rewriting r as it was shown earlier in XQuery syntax at the bottom of Figure 2.2.

op

$book_{ID}$	$book_{cont}$
$book_{ID,1}$	⟨book⟩ ⟨author⟩author1⟨/author⟩ ⟨/book⟩
$book_{ID,2}$	⟨book/⟩
$book_{ID,3}$	⟨book⟩ ⟨author⟩author2⟨/author⟩ ⟨author⟩author3⟨/author⟩ ⟨/book⟩

$\Rightarrow$

$nav_{book_{cont},\ //author_{cont}}(op)$

$book_{ID}$	$book_{cont}$	$author_{cont}$
$book_{ID,1}$	⟨book⟩ ⟨author⟩author1⟨/author⟩ ⟨/book⟩	⟨author⟩author1⟨/author⟩
$book_{ID,3}$	⟨book⟩ ⟨author⟩author2⟨/author⟩ ⟨author⟩author3⟨/author⟩ ⟨/book⟩	⟨author⟩author2⟨/author⟩
$book_{ID,3}$	⟨book⟩ ⟨author⟩author2⟨/author⟩ ⟨author⟩author3⟨/author⟩ ⟨/book⟩	⟨author⟩author3⟨/author⟩

Figure 2.4: Sample input and output to a logical nav operator.

2.2.3 XML View-Based Rewriting Algorithm

The rewriting algorithm we build upon [MKVZ11, Kar12] is computationally expensive: it extends the simpler XPath 1.0 case when the views as well as the query have a single return node, for which the problem is coNP-hard [CDO08]. As we will show in Chapter 3, frequent calls to the rewriting algorithm are very expensive and should be avoided. Finally, the rewriting algorithm we build upon generates *complete* and *minimal* rewritings.

Definition 2.2.1 (Complete Rewriting)**.** *A rewriting r of a query q is called complete when r can be processed completely based on materialized views, i.e. without access to the base data.*

Alternatively, one could also consider partial rewritings, i.e., evaluate part of the query based on the views and part of the query using the base documents. In a partial rewritings setting, one would need to join the results from the views, with those directly extracted from the document, to obtain the query answer. For our tree pattern query language, any decomposition of a query in, say, two sub-queries amounts to some split of the corresponding tree pattern. Joining the partial results requires the base data store to use the same class of node identifiers as the views, which may not always be possible.

What is more, if one traverses the base data tree to match part of the query, it is often as or more efficient to match the whole query tree pattern during this traversal, as to retrieve just partial query results and to further join with a view. While we consider that focusing on complete rewritings only does not generally lead to loss of performance, a more thorough investigation of partial rewritings is the topic of future work.

Definition 2.2.2 (Minimal Rewriting)**.** *A rewriting r of a query q based on a set of views V is minimal if no other rewriting of q uses a proper subset of V.*

In other words, one cannot obtain an equivalent rewriting of q using only *a strict subset of the view occurrences appearing in* r. For instance, the rewriting r in

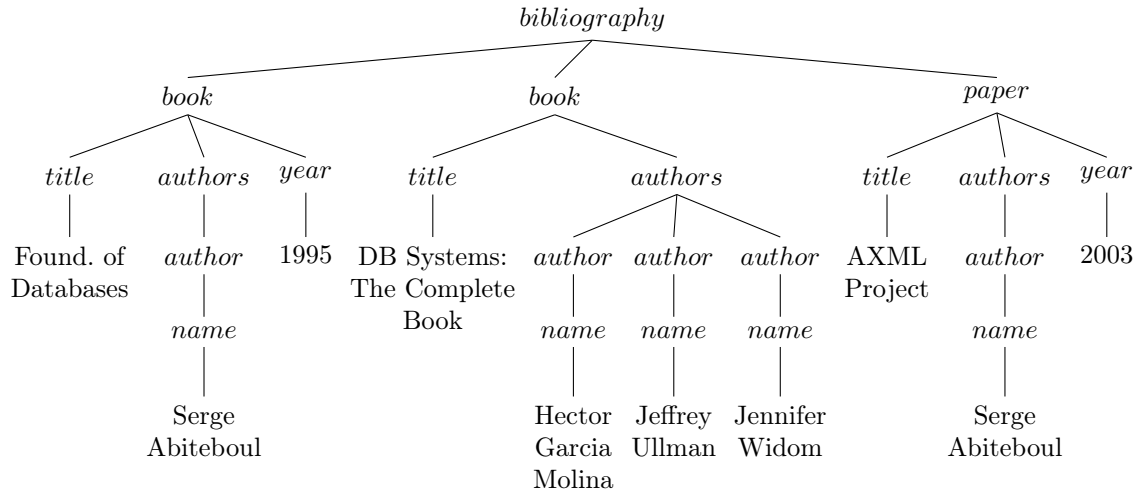


Figure 2.5: Sample XML Document.

Figure 2.2, of the form $\sigma(v_1 \bowtie_{paper.ID} v_2)$ is minimal. In contrast, a rewriting r' of the form $\pi(\sigma(v_1 \bowtie_{paper.ID} v_1 \bowtie_{paper.ID} v_2))$, using the v_1 view twice, in a self-join on paper.ID, is not minimal. Considering only minimal rewriting allows keeping view storage space *and* rewriting evaluation costs low. Indeed, with our assumptions on $cost^{\epsilon}$, a non-minimal rewriting is likely to incur a higher evaluation cost than the non-minimal one – if only for scanning the extra view occurrences.

2.3 Rewriting Cost Estimation and Optimization

In this section we present methods for estimating the cost of rewriting plans and then optimizing them for expediting query evaluation over materialized views. More specifically, in Section 2.3.1 we show how we estimate materialized view sizes, while in Section 2.3.2 we show details about our cost estimations. Finally, in Section 2.3.3 we show how algebraic plans can be optimized for low cost execution.

2.3.1 View Size Estimation

Query optimizers and database tuning algorithms decide on the most efficient plan and database design, based on estimations of the size of query results. Similarly, on our view selection algorithms Chapter 3, knowing the size of the views that are candidates for materialization is very important. The size of a given view is characterized by two metrics:

1. the view's *cardinality* (number of tuples), and
2. its *space occupancy* (in bytes).

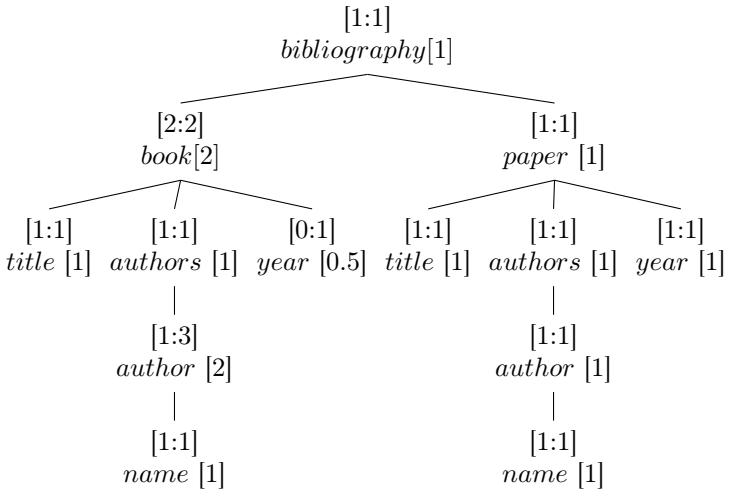


Figure 2.6: XSum summary for the document of Figure 2.5.

Both the cardinality and the space occupancy of a materialized view depend on the view itself but also on the database (of documents) over which the view is defined. Size estimations for graph and document databases have been proposed in the past, the most important of which being [PGI04, QLO03, GW97]. The general idea behind these works is simple: for each document, one can build a document *summary*, a data structure that encapsulates a set of statistics about:

— the paths found in the document;

— statistics of values inside XML elements, value distribution histograms etc.

Such a summary of a document d, denoted by $sum(d)$, can be used to estimate the size of a view without having to actually materialize it.

View Cardinality Estimation To summarize graph structures, Dataguides [GW97] were introduced in the context of semistructured OEM (Object Exchange Model) graphs. An implementation of Dataguides had been previously developed within our group [ABMP08], called XSum, and we enhanced it with a selectivity estimation module used for our query processing purposes, which we describe below. Figure 2.6 depicts a sample document summary extracted by XSum out of the document depicted in Figure 2.5.

XSum summaries store (among others) the following:

— **Average Number of Children** For every path p/l where p is a path and l is a label, XSum stores the average number of children labeled l found under path p. In Figure 2.6, the average number of children is placed next to the respective l node. For instance, Figure 2.6 we see that there is (in average) one $bibliography$ element in the document, 0.5 $year$ elements under $book$ elements and 2 $author$ elements under each $/bibliography/book/authors$.

— **Edge Cardinalities** For every path p/l where p is a path and l is a label, XSum stores the minimum and maximum number of children labeled l found under path p. In Figure 2.6, edge cardinalities are placed over the respective l node. For instance, Figure 2.6 we see that there

is exactly one $bibliography$ element in the document ([1:1]), 0 to 1 $year$ ([0:1]) elements under $book$ elements and 1 to 3 $author$ elements under $/bibliography/book/authors$ ([1:3]).

— **Value/Size Statistics** For each distinct path, XSum summaries also store:
  — the average subtree size (in bytes);
  — the average text value size (in bytes) [1] and;
  — the number of distinct text values.

Finally note that, for our estimations, we assume that values in a given XML path are *uniformly distributed*, and that values of different paths are *independently distributed*.

Tree Pattern Cardinality Estimation In order to estimate the cardinality of a tree pattern view v, given a document d, we first use the XSum summary of d, denoted by $XSum(d)$ to find all the paths of d that match v. We then use those paths to generate a set of *unfolded* tree patterns, where all //-edges are replaced by the parent-child paths from the document summary, that match the // path.

For example, consider the view that stores all the authors of the bibliography document depicted in Figure 2.5. The view is defined by the pattern $//author_{val}$. Given the summary of Figure 2.6, unfolding the view $//author_{val}$, yields the following two unfolded tree patterns:

1. $/bibliography/book/authors/author_{val}$
2. $/bibliography/paper/authors/author_{val}$.

We now move to explaining how the unfolded tree patterns of a given pattern p can be used to estimate the cardinality of p.

From the above example, it is easy to see that the sum of estimated cardinalities of the unfolded tree patterns gives an estimation of the actual cardinality of the original tree pattern.
For instance, consider the pattern:

$$/bibliography/book/authors/author_{val}.$$

There is in average 1 $bibliography$ element in the document, and 2 $book$ elements under $bibliography$. Thus, the cardinality of the path $/bibliography/book$ would be $1*2 = 2$. Similarly, there is 1 $authors$ element for each $/bibliography/book$. Thus, the cardinality of $/bibliography/book/authors$ would be $1 * 2 * 1 = 2$. Finally there are in average 2 $author$ elements under each $/bibliography/book/authors$. Thus, the total cardinality of the pattern p is estimated to be $1 * 2 * 1 * 2 = 4$. Similarly, the estimated cardinality of $/bibliography/paper/authors/author_{val}$ is 1. Thus, the total cardinality of $//author_{val}$ is estimated to be $4 + 1 = 5$.

Definition 2.3.1 (Estimated Tree Pattern Cardinality)**.** *We define the estimated cardinality of a tree pattern p, as the sum of the estimated cardinalities of its unfolded tree patterns.*

1. The average subtree and text value size correspond to what the $cont$ and val annotations return respectively in our query language.

Algorithm 1: Tree Pattern Cardinality Estimation

```
   Input  : Tree Pattern pattern, XSum summary sum
   Output: Estimated Cardinality of pattern
1  Algorithm TPCardinality(pattern, sum)
2      card ← 1
3      foreach unfolded tree pattern p ∈ sum.unfold(pattern) do
4          card ← card + cardinality(p, p.root, sum)
5      return card
1  Procedure cardinality(tree pattern p, node n, XSum summary sum)
2      card ← 1
3      foreach edge e ∈ outEdges(n) do
4          if ∃ node n_r ∈ p descendant of n such that n_r is return node then
5              card ← card × sum.avgChildren(e.src, e.dst)
6          else
7              //this is an existential branch. card reduced by a constant C
               card ← card/C
8          if n annotated with equality predicate then
9              card ← card × 1/sum.numOfDistinctTextValues(n)
10         card ← card × cardinality(p, e.dst, sum)
11     return ⌈card⌉
```

Algorithm 1 describes the cardinality estimation algorithm exemplified above. First, the given tree pattern view is unfolded. The result of the unfolding is a set of unfolded tree patterns. Then, for every unfolded tree pattern uP, the algorithm estimates uP's cardinality with a call to the recursive `cardinality` procedure and adds uP's cardinality to the total cardinality $card$.

The main functionality of our estimations is implemented by the `cardinality` procedure of Algorithm 1. It goes as follows: given the root n of an unfolded tree pattern, the `cardinality` procedure iterates through the outgoing edges of n. Given an edge e [2], `cardinality` retrieves the average $e.dst$ children of $e.src$ from the summary sum.

The algorithm then checks whether there exists a descendant of n, $n_r \in p$, that returns an ID, val or $cont$. If such an n_r exists, the current cardinality is multiplied by avg. If not, the cardinality will be reduced by a constant C as the branch of p, rooted at n is existential; existential branches can only reduce the cardinality of a view. Finally if n is annotated with an equality predicate, the cardinality will also be reduced: $card$ is multiplied by the inverse of the number of distinct values that the summary stores for that path.

The final step (line 10) is to go further down in the unfolded tree pattern and calculate the cardinality of the subtree of p, rooted at $e.dst$. Returning from the recursion, the cardinality of the subtree will be multiplied with the current

2. The source of e, $e.src$ is n itself and the destination of e, $e.dst$ is one of the children of n.

cardinality $card$.

Joined Pattern Cardinality Estimation A joined tree pattern can be alternatively expressed as a projection, over a selection over the cartesian product of all its tree patterns [MKVZ11]. More specifically, the general form of a joined tree pattern jp is: $\pi(\sigma_{pred}(t_1 \times t_2 \times, \ldots, \times t_k))$ where $\{t_i\}_{1 \leq i \leq k}$ are the tree patterns in jp and $pred$ is the predicate that applies all equality selections that enforce the value joins found in jp.

The traditional approach to estimate the cardinality of equality selection (thus, also equi-join) operators in relational databases is the one taken in System R [SAC+79] where only 10% of the input tuples survive a selection [3]. The same heuristic is used by recent cardinality estimations [TGMS08] proposed for XQuery joins.

We apply this heuristic also in our case. Given the joined pattern jp, shown above, we estimate the cardinality its cardinality as follows:

1. we estimate the cardinality of each tree pattern in jp;
2. we calculate the product of tree pattern cardinalities (thus, calculating the cardinality of $t_1 \times t_2 \times \ldots$ shown above) and finally;
3. we reduce the product of cardinalities by 10% for each of the value joins in jp.

Note that, unlike tree patterns, joined tree patterns might join results coming from multiple documents. Thus, in order to estimate the cardinality of a joined tree pattern, one has to calculate the cardinality of all tree patterns of a joined pattern *over all the documents* of the database.

Formally, based on the above, we estimate the cardinality for a joined pattern jp, given a set of documents D as follows:

$$cardinality(jp, D) = 0.1^{joins(jp)} \times \prod_{i=1}^{k} (\sum_{\forall d \in D} \texttt{TPCardinality}(t_i, XSum(d))) \quad (2.1)$$

where $joins(jp)$ is the number of value joins in the joined tree pattern jp and $\{t_i\}_{1 \leq i \leq k}$ are the tree patterns of jp.

View Space Occupancy Estimating the space occupancy of a joined pattern is straightforward: given a joined pattern jp, we first find the average tuple size of the tuples returned by jp and multiply it by the estimated cardinality of jp. The final number is an estimation of the total space occupancy of jp.

For instance, consider a joined pattern jp that returns tuples consisting of two columns: a $cont$ column whose average size is 100 bytes and a val column whose average size is 20 bytes. It is easy to see that the average tuple size of jp is 120 bytes. If the cardinality of jp is 200, the total estimated space occupancy of jp is 120*200=24.000 bytes.

3. Alternatively, one could change our estimations such that more recent heuristics are used; for instance [RG00].

Average Tuple Size Recall that for a given a document d, the summary of d, $XSum(d)$, stores the average subtree size and average text value for all paths in d. Given a tree pattern tp, and a return node $n_r \in tp$ one can match tp against the summary $XSum(d)$ and then retrieve the average size of n_r over all matching paths.

The average tuple size of a tree pattern tp given a document d, is given by the sum of the average sizes of tp's return nodes. We denote the average tuple size of tp given a document d by $avgTupleSize(tp, d)$.

Taking this one step further, one can calculate the average tuple size of a tree pattern tp expressed over a *set of documents* D, simply by calculating the average over all $avgTupleSize(tp, d), \forall d \in D$. Formally:

$$avgTuple(tp, D) = \frac{\sum_{\forall d \in D} avgTupleSize(tp, d)}{|D|} \tag{2.2}$$

Finally, the tuple signature (i.e., which columns and of which type ($ID, val, cont$) the tuple contains) of a joined tree pattern jp is the concatenation of the the tuple signatures of all tree patterns $tp \in jp$. Thus, in order to estimate the average tuple size of a joined tree pattern, it is enough to calculate the average tuple size of its contained tree patterns and then sum them.

2.3.2 Algebraic Plan Cost Estimation

There might be several rewritings for a query q using a set of views V; for example, consider the query q: $/a/b_{cont}/c_{val}$ and the views v_1: $/a_{id}$ and v_2: $//b_{id,cont}/c_{val}$. Query q can be rewritten by performing a structural join of views v_1 and v_2 on the IDs of a and b and finally:

— projecting b_{cont} and c_{val} of v_2 or;

— projecting b_{cont} and navigating into b_{cont} of v_2 to extract and project c_{val}.

Observe that these two rewritings use exactly the same views but their evaluation costs may considerably differ. The evaluation costs depend on the data and physical operators used during the rewriting plan execution.

Definition 2.3.2 (Evaluation cost of a rewriting plan)**.** *We define the evaluation cost of a rewriting plan r as a function $c : r \to \mathbb{R}^k$ where k is the number of distinct resources that are involved in the evaluation of r, typically I/O, CPU etc.*

Note that each result of c is a vector stating the consumption along each cost dimension (I/O, CPU etc.).

Distributed Evaluation Costs We now turn to a distributed scenario where, without loss of generality, each view is located in a different network site. In this case, each network site can have different I/O, CPU, incoming/outgoing bandwidth costs etc.

Let N be the set of network sites on which query evaluation can be distributed. In our previous example, the data of v_1 can be shipped to the network site holding

v_2 and execute there the rest of the plan. Alternatively, the data can be shipped to the network site holding v_1 and evaluate the rest of the plan there. Observe that, depending on where the plan is evaluated, the costs might differ.

Returning to the general case, let P_r be the set of all distributed physical plans for a given rewriting plan r running on the sites N. In this case, the cost function c defined earlier, should output the cost along all cost dimensions (I/O, CPU, incoming/outgoing bandwidth etc.) for each of the N sites on which r is running.

For this distributed scenario, the cost function c is defined as follows:

Definition 2.3.3 (Evaluation cost of a distributed rewriting plan)**.** *We define the evaluation cost of a distributed rewriting plan r as a function $c : P_r \rightarrow \mathbb{R}^{|N|\times k}$, assigning to each plan $p \in P_r$, the estimated costs, along different cost dimensions in different network sites, entailed by the evaluation of p.*

Observe that each result of c is now a matrix stating the consumption along each cost dimension and at each site.

Comparing Costs To enable comparing costs, we rely on a single *cost aggregator* which combines the utilization cost of all the different resource components of the sites involved in the execution of a plan, and returns a single (real) number. The aggregator may for instance sum up all the cost components, possibly assigning them various weights depending on the metric and/or the site involved.

Definition 2.3.4 (Cost aggregator)**.** *The cost aggregator α, is defined as a function $\alpha : \mathbb{R}^{|N|\times k} \rightarrow \mathbb{R}$, that combines the utilization costs of all different resource components into a single real number.*

In the sequel, for a given plan $p \in P_r$, we will simply write $cost(p)$ to denote the scalar aggregation $\alpha(c(p))$ of p's multidimensional costs.

Concerning $cost$, we make a few assumptions that are well-supported by the existing XML processing literature. First, each physical operator has a positive cost. Second, the cost of each physical operator (such as view scan, hash join, holistic twig join etc.) is monotonous in the size of each of its inputs, that is: if we fix all but one inputs to the operator and *add* tuples to the last input, the cost of evaluating the operator increases. Summing this up over a fixed physical plan, the more data is added in the views on which the plan is computed, the higher the physical plan evaluation cost.

2.3.3 Plan Optimization

We assume available an *algebraic cost-based optimizer* that is pipelined at the output of the query rewriting algorithm: for each distributed algebraic rewriting plan r of a query q found by the rewriting algorithm, the optimizer applies logical and physical plan transformations, looking for the most efficient way to evaluate r, such that the $cost(r)$ function returns the *minimum physical evaluation cost* among all equivalent rewritings of q. Several XML cost-based optimization techniques have been proposed in the past [CC10, PZIÖ06, WPJ03].

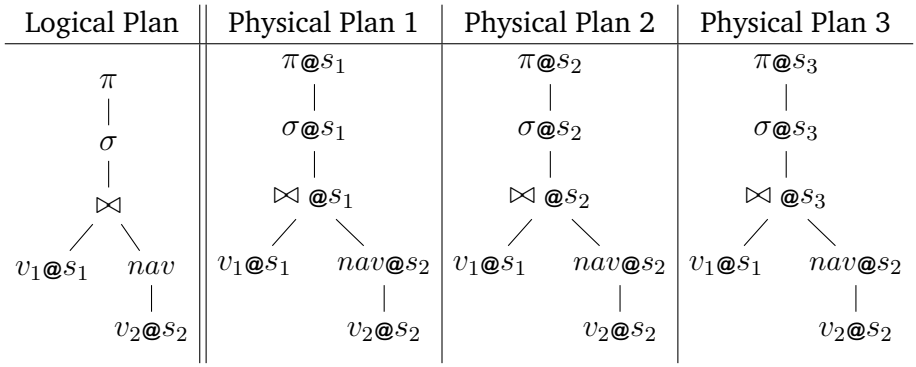


Figure 2.7: Logical plan and two physical operator placement configurations.

To build distributed physical plans out of logical plans formulated in the algebra previously described, our optimizer first applies optimizations on the logical level and then proceeds in the translation of the logical plan to a physical one.

Logical Plan Optimization At the logical level, all selections and projections are simply pushed as low as possible in the logical algebraic plan. One could also perform join reordering in the logical level, though, our optimizer is not yet capable of doing so. This is admittedly a limitation of our current optimizer implementation. The rest of the optimizations take place while transforming the logical plan into a physical one.

Physical Plan Optimization The optimizer transforms a *logical plan* into a *physical plan* through a recursive traversal, which generates plans from the bottom up. At any given point, the optimizer has to make two decisions:

1. what physical operator will be used as an "implementation" of the logical one, i.e. a logical join operator can be implemented by a Nested Loop or Merge or Hash join etc. and;
2. where the operator is going to be placed, i.e. a binary join operator can be placed on either of the two sites that feed it.

More specifically, the logical to physical transformation follows the principles of [ML86] and goes as follows:

1. The first physical operators produced, are view scans. These are leaf physical operators, each of which is placed at the site of the respective view.
2. Selections, projections and navigations are always translated into their corresponding physical operators and are placed at the same site as their immediate child operators. In other words, a selection, projection or navigation is evaluated at the site where its input data originate from, thus avoiding unnecessary data transfers.
3. Now consider joins of plans, of the form $o = o' \bowtie o''$.
   The first decision to make is the choice of the physical join algorithm which must be used to implement the logical join $\bowtie$:

— If the join condition is on a conjunction of equality predicates, an in-memory hash join is used. In case the inputs or outputs of the join are estimated to exceed the amount of main memory, an external join is used instead.
— If the join condition is a single structural ID comparison (checking if some node is a parent or ancestor of the other), the optimizer checks if the inputs are properly ordered for this join, adds Sort operators on the inputs as needed, and then develops *two* physical join plans: one using the StackTreeAncestor physical join operator [AKJP+02] and another one using the StackTreeDescendant operator.
— In all other cases, a Nested Loop physical join operator is used.

The second decision to make concerns the placement of the physical join(s) thus obtained. The optimizer generates all possible combinations of:
— all the physical join algorithms selected to implement $\bowtie$, as explained above and;
— the site where the plan can run (i.e. the sites where o' and o'' are evaluated).

The procedure above may generate a very large number of physical plans. To avoid this very lengthy exploration, a budget k (number of allowed plans) is fixed before the optimization starts and is not exceeded during optimization.

Figure 2.7 depicts a logical plan where the view v_1 is stored at the network site s_1 and v_2 is stored at s_2. Suppose that the query is posed in network site s_3. On the right of the logical plan we depict three of its physical plan translations.

In the current implementation, the optimizer always places a navigation operator on the same site as the site of its child operator. This is based on the heuristic that oftentimes navigation reduces the number of tuples, since some tuples may lack matches for the navigation paths - and such tuples are pruned away by the navigation operator. In practice, this heuristic has held in most of the cases we experimented with, although it can obviously be contradicted by other examples.

On the other hand, the join operation can be evaluated in one of the two sites s_1, s_2 that store the views (Physical Plan 1 and 2) or, on site s_3 where the query is posed (Physical Plan 3).

After the physical plans have been enumerated, it is up to the optimizer to estimate the cost of those plans and choose the one with the least cost. In summary, our optimizer going from a logical plan to an executable physical plan, relies on a set of heuristics, which aim at reducing data transfers between sites and picking efficient join operators whenever possible.

2.4 Summary

In this chapter we presented XML, a popular data model for representing and sharing data on the Web, along with the main approaches that have been proposed by the scientific community for the efficient storage and retrieval of XML data. Then we discussed the management of data based on materialized views and how XML queries can be answered from views. We then showed how the cost of rewriting plans can be estimated and how rewriting plans can be optimized for lower evaluation costs.

Chapter 3

Materialized View Selection for XQuery

In this chapter, we consider the setting of XML databases where data is stored in XML files and query processing is done over materialized views defined over the XML files. Based on a query workload and a given storage space budget, we address the problem of selecting which set of materialized views to select in order to minimize query evaluation costs. We start by identifying the family of views that can be candidates for materialization and we propose techniques to reduce it. We then devise two selection algorithms. The first is inspired by previous works on the same problem that consider only simple, one-view query rewritings while the second, is applicable to environments where multiple views can be used to answer a query. The performance and efficiency of our algorithms is demonstrated through a series of experiments that include algorithms from the state of the art.

The work presented in this chapter has led to a demonstration of a software prototype [CRKMR10] and this chapter closely follows a published paper [KMV12].

3.1 Motivation and Outline

The efficient processing of XML queries raises many challenges, due to the complex and heterogeneous XML structure, and on to the complexity of the W3C XQuery language. XQuery is Turing-complete, thus many performance-enhancing works focused on speeding up the processing of a central language subset, typically consisting of tree patterns. Performance enhancing techniques include efficient tree pattern evaluation algorithms [BK09b, GKM09, KRML05], new physical operators such as the Holistic twig join [BKS02] and its improved variants, query simplification and minimization [AYCLS02], algebraic optimization etc. To speed up XML data access, previous research has focused on building efficient stores, exploiting XML node identifiers encapsulating useful structural information, as well as building XML summaries and indices, with DataGuides [GW97] and $D(K)$ indices [QLO03] being among the best-known proposals.

Materialized views have improved performance by orders of magnitude in relational databases [ACN00, GM99b, HRU96], and they raised interest also in the context of XML databases [AMR+98]. The problem of rewriting an XML query using one view has been extensively studied e.g., in [MS05, XO05, YLH03], and using several views, e.g., in [CDO08, CC10, MKVZ11, TYÖ+08]. A dual problem to view-based rewriting is the automated selection of materialized views to improve the performance of a given XQuery workload. Well-studied for relational [ACN00, GM99b, MCB11] and RDF [GKLM12] databases, it has also attracted attention for XML queries and views [EAZZ09, MS05, TYT+09, YLH03].

In this chapter, we consider the problem of selecting a set of views to be materialized in order to minimize the processing costs associated to a given query workload Q. We consider queries and views expressed in a large subset of XQuery, consisting of conjunctive tree patterns (using the child and descendant axis and existential branches) that return data from several nodes and are connected with value joins. Following [ABMP07, BOB+04, CDO08, MKVZ11, TYÖ+08], our views (and queries) are also allowed to store XML node identifiers, which enable interesting view joins and potentially more efficient rewritings.

We picked this language since it is among the most expressive for which multiple-views equivalent query rewriting algorithms are known. Specifically, we rely on the rewriting algorithm of [MKVZ11] which, given a set of views $V = \{v_1, v_2, \ldots, v_n\}$ materialized over a database D and a query q, returns the equivalent rewritings of q using the views in V. Each such rewriting is *complete*, in the sense that the database is no longer needed in order to evaluate the queries. A rewriting is expressed in a tuple-based XML algebra, to be detailed further on.

Assuming that each query $q_i \in Q$ is associated a weight $w_i \geq 0$ (for instance, reflecting the query frequency), the view selection problem we consider is: find the set of materialized views V_{best} such that the weighted sum of the costs for processing all workload queries through rewritings based on V_{best} views, is the smallest that can be attained among any other view set.

In this chapter, we make the following contributions:

— We are the first to formalize the problem of materialized view selection for the expressive tree pattern query with value joins dialect we consider. We show that the space of potential candidate views makes complete exploration unfeasible and present several effective candidate pruning criteria.

— We leverage an existing query rewriting algorithm [MKVZ11] to propose two view selection algorithms: a benefit-oriented greedy algorithm named UDG, reminiscent of previous algorithms [MS05, TYT+09], and a state search-based algorithm named ROA, exploring many view set transformations and including a randomized component.

— We compare UDG and ROA with their closest competitors from the literature [MS05, TYT+09]. Our experiments show that ROA scales well beyond the algorithms of [TYT+09] and our UDG. While ROA is slower than the algorithm of [MS05], we show that it consistently recommends view sets leading to lower processing costs. This is because ROA considers many-

views rewritings, and exploits the full spectrum of rewriting possibilities our query and view language enable.

The remainder of this chapter is organized as follows. Section 3.2 gives the problem definition. Section 3.3 discusses candidate view sets. Section 3.4 presents our view selection algorithms, while Section 3.5 details the closest competitors we compare with. Section 3.6 describes our experiments. We discuss other related works in Section 3.7 and then we conclude.

3.2 Problem Statement

A view set V brings cost savings to a given query workload Q since queries don't have to be evaluated from the base data. We define the benefit of evaluating a given workload Q in the presence of a set of materialized views V as follows:

Definition 3.2.1 (Benefit of a view set)**.** *The benefit b of a view set V is the difference between the cost of executing the queries of a workload Q over the base data, denoted by $cost(q|_\emptyset)$, and the cost of executing the same queries over the set of materialized views V, denoted by $cost(q|_V)$. Formally:*

$$b(V,Q) = \sum_{q \in Q} (w_i \times (cost(q|_\emptyset) - cost(q|_V)))$$

where w_i is the weight of query q_i in Q.

Adding or removing a view v to a given set of views V can increase or decrease the benefit of V. In addition, for a given set of views V and two views v_1, v_2, adding v_1 *alone* or v_2 *alone* to the existing view set V may not increase its cost savings, if $V \cup \{v_1\}$ and $V \cup \{v_2\}$ do not enable any new interesting rewritings, while adding v_1 and v_2 *simultaneously* to V would increase it. This is the case when a more efficient rewriting can be found based on $V \cup \{v_1, v_2\}$.

Our problem statement can be formalized as follows:

Definition 3.2.2 (Problem Definition)**.** *Given a space budget S, find the view set V_{best} such that $size(V_{best}) \leq S$ and $b(V_{best}, Q) \geq b(V, Q)$ for all view sets V fitting in S.*

Note that by setting S to be infinitely high, one can get a variant of our problem where the goal of the optimization is to *find the set of views V_{best} with the largest benefit $b(V, Q)$ among all other view sets*.

3.3 Candidate View Sets

In this section, we first explain which views can be considered candidates for materialization (Section 3.3.1). Section 3.3.2 presents a set of techniques for pruning candidate views and finally, Section 3.3.3 presents different candidate view sets for our problem.

3.3.1 Candidate Views for a Workload

Definition 3.3.1 (Candidate view)**.** *A view v is a candidate view for a query q* iff *there exists an equivalent rewriting of q using v (and possibly other views).*

We denote by $\mathcal{CS}_0(q)$ the set all candidate views for the query q, and by $\mathcal{CS}_0(Q)$ the candidates for all queries in a workload Q. We start by considering candidates for tree pattern queries.

3.3.1.1 Candidate Views for a Tree Pattern Query

It has been shown [MKVZ11, TYÖ+08] that a tree pattern view v may participate in an equivalent rewriting of a tree pattern query q only if there exists a tree embedding $\phi : v \rightarrow q$. This embedding must preserve node labels, i.e., for any $n \in v$, $label(n) = label(\phi(n))$. The embedding must also respect structural relationships between nodes:

— for any node $n \in v$ and m a $/$-child of n, $\phi(m)$ must be a $/$-child of $\phi(n)$;
— for any node $n \in v$ and m a $//$-child of n, $\phi(m)$ must be a descendant of $\phi(n)$.

Finally, ϕ must not contradict value predicates from the query, i.e.: for any node $n \in v$, such that $m = \phi(n) \in q$, if m is annotated with a predicate of the form $[val = c_1]$ for some constant c_1, then n must not be annotated with a predicate of the form $[val = c_2]$ for some constant $c_2 \neq c_1$.

We now turn to the task of enumerating $\mathcal{CS}_0$ candidates. We denote by *non-annotated* tree patterns those patterns whose nodes carry an attribute or element name, but no annotation of the form val, $cont$, ID, or $[val = c]$. One can enumerate all candidate views for a workload Q by:

— enumerating all *non-annotated* tree patterns that can be embedded in some query $q \in Q$ and
— creating from each non-annotated tree pattern thus obtained, all possible tree patterns that differ in their ID, val and $cont$ annotations.

For example, consider the workload Q_1 consisting of the single query q_1: $/a/b/c_{val}$. Sample non-annotated tree patterns which can be embedded in q_1 are: $/a$, $//a$, $//b$, $//c$, $/a/b$, $/a//b$, $\ldots$, $/a/b/c$, $//a/b/c$ etc. In turn, from the non-annotated tree pattern $/a$, one can derive e.g., the annotated patterns $/a_{ID}$, $/a_{ID,val}$, $/a_{ID,val,cont}$, $/a_{cont}$ etc.

Estimating the Size of $\mathcal{CS}_0(q)$ We denote the number of nodes of q by $|q|$ and start by counting the non-annotated tree patterns which can be extracted out of q. For a given $k < |q|$, there are $\binom{|q|}{k}$ subsets of q nodes, and in the worst case, each subset determines a sub-pattern of q, having k edges (counting also the edge above the sub-pattern root, e.g., in Figure 2.3, the $//$ edge above the $confs$ node). Each such edge could be annotated $/$ or $//$. Thus, the number of non-annotated tree patterns of k nodes that can be constructed from a query q is, in the worst case, $\binom{|q|}{k} \times 2^k$.

We now consider building candidate views out of an non-annotated tree pattern t_{ul}. Let n be a node of t_{ul} such that there is an embedding from $\phi_{ul} : t_{ul} \rightarrow q$

for some q in the workload. To obtain an annotated tree pattern t (candidate view) out of t_{ul}, we need to decide on the annotations of each node $n' \in t$ corresponding to $n \in t_{ul}$. We can annotate n' with any of the four subsets of the attribute set $\{ID, cont\}$, to indicate whether t stores an ID and/or the full serialized XML image of the node. With respect to the val attribute, two cases occur: (i) if the query node $\phi_{ul}(n)$ is annotated with a predicate of the form $[val = c]$, one may label n' with either val, $[val = c]$ or no val label; (ii) if $\phi(n)$ has no such predicate, we can annotate n' with val, or omit the val label, but we cannot annotate with $[val = c]$ for any constant c, since this would prevent the existence of an embedding $\phi : t \to q$ and thus prevent t from being a candidate view for q. Thus, in the worst case, there are 3 val annotation possibilities for n', which, multiplied by the 4 possibilities of $ID, cont$ annotation, lead to 12 possible node annotations for n'. Assuming the size of t_{ul} (and t) is k, the node annotation possibilities alone lead to 12^k possible t trees out of a given t_{ul}.

Based on this, out of a query q, the number of candidate views of size k is: $\binom{|q|}{k} \times 2^k \times 12^k$, where the 2^k factor is due to the edge labeling possibilities and the 12^k factor is due to node annotations. It follows that $|\mathcal{CS}_0(q)|$ is:

$$\sum_{k=1}^{|q|} \binom{|q|}{k} \times 2^k \times 12^k = \sum_{k=0}^{|q|} \binom{|q|}{k} \times 24^k - 1 = 25^{|q|} - 1$$

We end the discussion of candidate views for tree pattern queries with an interesting remark. Given a workload Q and view set V, we say a view $v \in V$ is *useful* if v is used in the best rewriting of some query $q \in Q$ using V. The set of useful candidate views for a query is guaranteed to be quite small: it turns out that a minimal rewriting of a tree pattern query q uses no more than $2 \times |q|$ views [MKVZ11]. However, we do not know which are the useful views before rewriting all the queries; moreover, our aim is to select views that are *globally* best for the whole workload. Thus, one cannot use this known small bound to prune out candidates.

3.3.1.2 Candidate Views for a Query with Value Joins

We now turn to the case of a tree pattern query q with value joins. One can show that a view v may participate in an equivalent rewriting of q only if there exists a set of tree embeddings $\phi_1 : t_1^v \to t_1^q$, $\phi_2 : t_2^v \to t_2^q$ etc. embedding each view tree pattern to some query tree pattern, and satisfying the following condition. For each value join in v of the form $n_i^v.val = n_j^v.val$, where n_i^v, n_j^v are nodes in the view tree patterns t_i^v, respectively t_j^v, the query must feature a value join edge between the nodes $\phi_i(n_i^v)$ and $\phi_j(n_j^v)$.

For example, consider the view v_2, with a value join between the year nodes of its tree patterns, and the query q in Figure 2.3. Let ϕ_l the embedding from v_2's left subtree into the right subtree of q, and ϕ_r the embedding from v_2's right subtree into the left subtree of q. Observe that ϕ_r and ϕ_l map the year nodes of v_2 into the two year nodes of the query, thus the condition for v_2 to participate in some

rewriting of q is satisfied. The intuition is that a view with "more join predicates" than the query cannot be used to rewrite it, following the similar property of relational containment mappings.

More generally, let q be a query q consisting of k tree patterns t_i^q, $1 \leq i \leq k$, and assume q has m value joins. We enumerate the candidate views as follows. (i) Build the candidate view sets $\mathcal{CS}_0(t_i^q)$ for $1 \leq i \leq k$; (ii) For each subset $\{i_1, i_2, \ldots, i_n\}$ of $\{1, 2, \ldots, k\}$, and each set of candidate tree patterns $t_1 \in \mathcal{CS}_0(t_{i_1}^q)$, $t_2 \in \mathcal{CS}_0(t_{i_2}^q)$ etc., create the candidate view $t_1 \times t_2 \times \ldots \times t_n$. Then, let JE be the set of query value join edges, such that embeddings from $t_1, t_2, \ldots, t_n$ into the query reach both ends of the value join edge. For each subset of $J \subseteq JE$, we generate a distinct candidate view by pushing on top of $t_1 \times t_2 \times \ldots t_n$ the join conditions of J. Overall, the number of candidate views for q is bound by $2^m \times |\mathcal{CS}_0(t_1^q)| \times |\mathcal{CS}_0(t_2^q)| \times \ldots \times |\mathcal{CS}_0(t_k^q)|$.

For example, in Figure 2.3, to obtain candidate views for q, one can first chose $\{i_1 = 1\}$ and generate the set $\mathcal{CS}_0^1$ of all candidate tree patterns for the left subtree of q; then, chose $\{i_1 = 2\}$ and generate the set $\mathcal{CS}_0^2$ of all candidate tree patterns for q's right subtree; finally, choosing $\{i_1 = 1, i_2 = 2\}$ leads to enumerating all combinations of the form $\{t_1, t_2 \,|\, t_1 \in \mathcal{CS}_0^1, t_2 \in \mathcal{CS}_0^2\}$ and, for each such t_1 and t_2: (a) add the view $t_1 \times t_2$ to the candidate set of q; (b) if t_1 and t_2 both have a year node, also add the view $t_1 \bowtie_{year.val} t_2$ to the candidate set.

The number of candidate views for all queries in a workload may be prohibitively high. Of course, the more commonality the queries exhibit, the more common candidates they may have, but this still leaves a large number of candidate views.

3.3.2 Pruning Candidate Views

We now describe several methods for pruning candidate views. We start by introducing two important notions. Let Q be a workload and V_1, V_2 be two candidate view sets.

Definition 3.3.2 (Rewriting power preservation)**.** *If, for every query $q \in Q$ and rewriting r of q using views in V_1, there exists a rewriting r' of q using views from V_2, we say that replacing V_1 with V_2* preserves rewriting power.

Rewriting power preservation ensures that V_2 enables to rewrite at least the queries V_1 did. However, it says nothing about the cost of the rewritings using V_2. The following notion is more restrictive:

Definition 3.3.3 (Rewriting cost preservation)**.** *If (i) replacing V_1 with V_2 preserves rewriting power and (ii) for any query $q \in Q$ and rewriting r of Q using V_1, there exists a rewriting r' of q using the views V_2 such that $cost(r') \leq cost(r)$, we say that replacing V_1 with V_2* preserves rewriting costs.

We now describe several candidate view pruning techniques. Based on a set of candidate views V, our first techniques each focus on a tree pattern query. Then we discuss pruning methods targeting queries with value joins.

NoCart Eliminating views with cartesian products, i.e., those having a pattern unconnected (by a value join) to any other tree pattern, can significantly reduce the number of candidates. For instance, for a query of 2 tree patterns t_1, t_2 combined by a value join, avoiding cartesian products leads to an upper bound reduction from $2 \times |\mathcal{CS}(t_1)| \times |\mathcal{CS}(t_2)|$ to $|\mathcal{CS}(t_1)| + |\mathcal{CS}(t_2)|$. However, in the case that the query features a cartesian product, and that the space budget allows it, it may be a good idea to materialize it. We define the pruning method NoCart as pruning away candidate views with a cartesian product such that in the query, the corresponding tree patterns are connected by a value join.

Formally, let cv be a candidate view for a query q. If cv comprises a cartesian product of two tree patterns, t_1^v, t_2^v, then there must exist two tree patterns t_1^q, t_2^q in the query, such that t_1^v embeds into t_1^q, t_2^v embeds into t_2^q, and t_1^v, t_2^v are not connected by a join edge in the query (that is, the cartesian product of t_1^q and t_2^q is a sub-expression of the query). NoCart preserves rewriting power and cost.

TrimAxis We have mentioned earlier in this section that by labeling candidate view edges either / or //, one gets 2^k possible edge labelings for a k-node view. Our TrimAxis pruning method eliminates some of the options brought by the edge labeling possibilities. For example, consider a workload consisting of the query q of Figure 3.1 and a view set including the candidate views cv_5 and cv_6 in the same Figure. In this case, TrimAxis will remove cv_5, since it has an ancestor-descendant edge // mapped only to a parent-child query edge. TrimAxis will preserve cv_6, identical to cv_5 except for the label of the edge between a and c.

Formally, we define this pruning as follows: let v_1, v_2 be two views in V, identical except for one edge: e_1 in v_1 is of the form n_1^1/n_1^2, and e_2 in v_2 is of the form $n_2^1//n_2^2$, n_1^1 and n_2^1 have the same label, while n_1^2 and n_2^2 have the same label. Assume that for every $q \in Q$ and every embedding $\phi_2 : v_2 \rightarrow q$, there is an embedding $\phi_1 : v_1 \rightarrow q$ such that $\phi_1(n_1^1) = \phi_2(n_2^1)$, $\phi_1(n_2^1) = \phi_2(n_2^2)$ and ϕ_1, ϕ_2 coincide on all the other nodes of v_1 and v_2. Then, TrimAxis transforms V into the view set $V' = V \setminus \{v_2\}$, in other words it removes v_2.

It can be shown that TrimAxis preserves rewriting power, because for every rewriting r based on v_2 one can build a rewriting r' based on v_1 computing the same results. TrimAxis also preserves rewriting costs, since v_1 stores at most as much data as v_2.

One may wonder whether TrimAxis should not also work the other way around, that is, prune views with / edges if they only match // edges in workload queries. However, such views views, because they do not embed into the query. For instance, if the query is $//a//b_{cont}$, the view $//a/b_{cont}$ cannot be used to rewrite it, thus it is not a candidate view.

TrimVal Let $v \in V$ be a view and n be a view node. Assume that the set of all embeddings of v into workload queries is $\{\phi_1 : v \rightarrow q_1, \phi_2 : v \rightarrow q_2, \ldots, \phi_k : v \rightarrow q_k\}$. Assume that for any $1 \leq i \leq k$, the query node $\phi_i(n)$ is neither annotated val nor with a predicate of the form $[val = c]$ neither takes part in any value-join. The Pruning method TrimVal replaces v in V with a copy v', in which the copy of n is

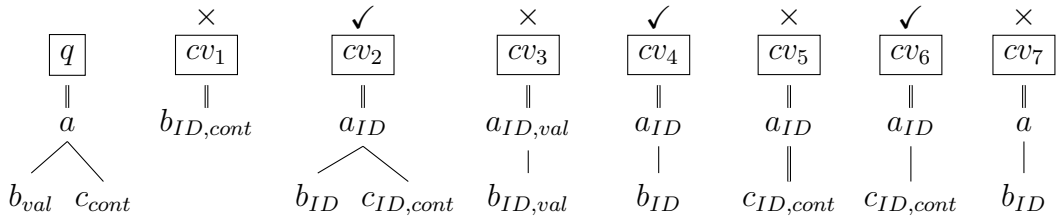


Figure 3.1: Sample query and some of its candidate views.

not annotated val.

For instance, in Figure 3.1, TRIMVAL replaces the view cv_3 with a copy of cv_3 whose a node is not annotated with val. The replacement takes place since the a query node q is not annotated val, nor $[val = c]$, and does not take part in a value join.

TRIMVAL preserves rewriting power, since it removes only val annotations that are useless in any rewriting. It also preserves rewriting cost, since eliminating val reduces view space occupancy without breaking any useful rewritings.

TRIMCONT It seems natural to remove unused $cont$ annotations just like val ones. One must take into account, however, that $cont$ attributes can be used by rewriting in a way that val does not support: as explained in Section 2.2.2, one may navigate by applying an XPath expression within a $cont$ attribute, to extract a subset of the data stored in that node (recall the example in Figure 2.4). Thus, before removing a $cont$, one must ensure this does not prevent some interesting navigation.

Formally, let $n \in v$ be a node in a candidate view v and let $\{\phi_1 : v \to q_1, \phi_2 : v \to q_2, \ldots, \phi_k : v \to q_k\}$ be the set of all embeddings of v into Q queries. If for any $1 \leq i \leq k$, the query node $\phi_i(n)$ is (i) not annotated $cont$ and (ii) a leaf in q_i, pruning method TRIMCONT replaces v with a copy v', in which the copy of n is not annotated $cont$. For instance, in Figure 3.1, TRIMCONT removes cv_1 and replaces it with a copy thereof, where the b node is not annotated $cont$. It is easy to show that TRIMCONT preserves rewriting power and cost as it removes only useless annotations.

3.3.3 Sets of Candidate Views

Many different candidate view sets can be used for solving our view selection problem. We present four different view sets below.

View Set $\mathcal{CS}_0$ As described in Section 3.3.1, given a query workload Q, the view set $\mathcal{CS}_0(Q)$ is the set of views that can be exhaustively enumerated by (i) enumerating all *non-annotated* tree patterns that borrow their tags from $q \in Q$; (ii) creating from each non-annotated tree pattern thus obtained, all possible tree patterns that differ in their ID, val, $cont$ and $[val = c]$ annotations and; (iii) by generating all possible joins and cartesian products between the generated tree patterns. The

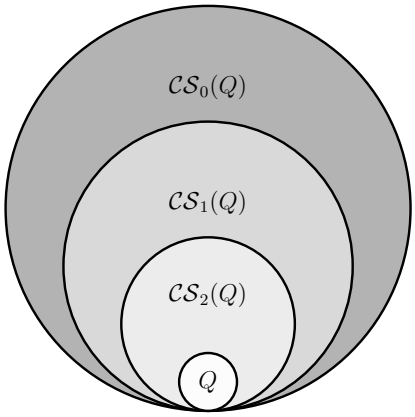


Figure 3.2: Venn diagram of the 4 view sets for a workload Q.

resulting $\mathcal{CS}_0(Q)$ set contains views that are syntactically correct according to our view language and are *candidate views* for a query $q \in Q$.

View Set $\mathcal{CS}_1$ For a query Q, we denote by $\mathcal{CS}_1(Q)$ the set obtained from $\mathcal{CS}_0(Q)$ by applying TRIMAXIS, TRIMVAL, TRIMCONT and NOCART exhaustively, until none of them applies any more. For example, in Figure 3.1, we mark the views that are included in $\mathcal{CS}_1(Q)$ with "✓" and the ones pruned out with "×".

Since the size of $\mathcal{CS}_1(q)$ may still be quite high for large queries, we identify a smaller candidate set:

View Set $\mathcal{CS}_2$ We denote by $\mathcal{CS}_2(q)$ the candidate view set obtained by pruning out from $\mathcal{CS}_1(q)$ all candidate views that contain non-linear tree patterns. Thus, the candidate view set is a view set containing candidates with linear tree patterns. For a workload Q, we set $\mathcal{CS}_2(Q) = \cup_{q \in Q} \mathcal{CS}_2(q)$. For instance, in Figure 3.1, the candidate view cv_2 is part of $\mathcal{CS}_1(q)$, but not part of $\mathcal{CS}_2(q)$.

Figure 3.2 depicts a Venn diagram of candidate view sets for a query workload Q.

3.4 View Selection Algorithms

In this section we describe algorithms for solving the view selection problem. Section 3.4.1 presents a simple exhaustive algorithm. Section 3.4.2 presents an algorithm inspired from the classic Knapsack problem.

3.4.1 Exhaustive Search

A simple algorithm for finding the best candidate view set is:

1. enumerate all candidate views in $\mathcal{CS}_0(Q)$
2. generate the powerset $\mathcal{P}(\mathcal{CS}_0(Q))$
3. compute the benefit of each set $V \in \mathcal{P}(\mathcal{CS}_0(Q))$ and

4. choose the set $V_{best} \in \mathcal{P}(\mathcal{CS}_0(Q))$ having the maximum benefit among those who fit the space budget S.

Clearly, this algorithm computes the solution to our view search problem. However, due to the large size of $\mathcal{CS}_0(Q)$ and even more of its powerset, it is unfeasible for meaningful workloads Q.

3.4.2 Knapsack-style View Selection

Our view selection problem is closely related to the 0-1 knapsack problem. The 0-1 knapsack problem considers a set of k items having the *space occupancy* s_1, s_2, ..., s_k and the *benefits* b_1, b_2, ..., b_k and tries to fill a space budget S with items so as to maximize the sum of the benefits of the selected items. However, there is a fundamental difference: in our case, since we support multiple-view rewritings, the benefit of selecting one view depends on the presence of other views among those already recommended for materialization. Considering that at some point during an algorithm we have selected the view set V, what we need to identify next is the candidate view (among those not already in V) that would lead to the greatest benefit *together with the views in* V.

We adapt knapsack-style view selection to our setting as follows.

Definition 3.4.1 (View utility)**.** *For a given workload Q and set of materialized views V, the utility of a view v is the ratio between the benefit brought by materializing v next to V, and the space occupancy of v:* $u(v) = b(V \cup \{v\}, Q)/size(v)$.

Evaluating $b(V \cup \{v\}, Q)$ requires rewriting each query $q \in Q$ using $V \cup \{v\}$, which for large Q and V sets may be extremely expensive. One can reduce this effort by rewriting only *some* queries $q \in Q$, namely those having a tree pattern t^q into which some tree pattern t^v of v embeds. (It is easy to show that the best rewritings of the other Q queries are not affected by the addition of v).

Utility-driven Greedy Algorithm (UDG) Our first utility-driven view selection algorithm, based on a given candidate view set V_c, goes as follows:

1. initialize the recommended view set V to $\emptyset$;
2. compute the utility of each view $v \in V_c$;
3. sort the candidate views in descending order of their utility;
4. add the candidate with the highest utility to V, if it fits the space budget, and remove it from V_c;
5. repeat steps (2)-(4) until no view $v \in V_c$ has a strictly positive utility, or the space budget S has been attained.

Observe that at step (2), the algorithm updates the utilities after each addition to V. This is because the utility of a view v_1 with respect to a view set V can either increase or decrease when a view v_2 is added to V. The utility of v_1 may increase, if v_1 and v_2 together enable a very efficient rewriting; the utility of v_1 may decrease

if v_2 is a competitor to v_1, i.e., v_2 enables a lower-cost rewriting than one enabled by v_1. The repeated recomputation of utility values brings quite some overhead, especially since it requires calling an expensive query rewriting algorithm, such as [MKVZ11, TYÖ+08]. An important remark allows to reduce this overhead: when considering whether or not to add a view v_1 to V, we only need to find out the new rewritings (if any) that v_1 enables. In turn, v_1 can only lead to new rewritings for those queries $q \in Q$ such that there is an embedding $\phi : v \rightarrow q$ (as explained in Section 3.3.1). This observation significantly reduces the benefit recomputation costs, especially for large workloads.

Algorithm UDG may miss the optimal solution. For instance, assume that query $q \in Q$ can be rewritten very efficiently based on v_1 and v_2, but neither v_1 nor v_2 suffice to rewrite q. In this case, the benefits of v_1 alone and v_2 alone may be small, leading UDG to chose neither v_1 nor v_2 for materialization. This prevents UDG from realizing how interesting it would have been to add *both*.

3.4.3 State Search-based View Selection

One can model our view selection problem as a state search problem. Every *state* consists of a set of materialized views, and a benefit. The *initial state* corresponds to a seed view set V_0, having some benefit $b(V_0)$. By adding, modifying, or removing a view from the initial state, one can obtain another state, characterized by the view set V, having the benefit $b(V)$, which may be higher or lower than that of V_0. Recall that the benefit is based on the cost of the *best* rewritings that V supports. Thus, we represent view selection as an optimal-state search problem in a directed graph, where nodes are states, and edges are transitions from one state to another.

How should one pick the initial state, i.e., the seed set of views? We have decided to start with $V_0 = Q$, that is, the workload queries themselves. If Q needs more space than S, the initial state is not a solution. To solve this problem, we will introduce space-reducing transitions, allowing us to reach acceptable states.

In the following, Section 3.4.3.1 describes a set of view set transformations, while in Section 3.4.3.2 we present a search algorithm based on these transformations.

3.4.3.1 State Transformations

The transformations we use determine the space of states that we can reach, and how fast we find interesting states. Moreover, those we present below, when applied on subset of $\mathcal{CS}_1$, can always yield another $\mathcal{CS}_1$ subset. Thus, there is a close relationship between the transformations and the pruning criteria presented in Section 3.3.2, which will be formalized at the end of this Section.

We start by describing a set of state transformations which can be shown not to lose rewriting power (recall Definition 3.3.2). Such transformations are most

interesting for us since our goal is to maximize benefit, and thus to rewrite as many queries as possible.

BREAK Splitting a view in several sub-views may enable the identification of common sub-expressions across views. Transformation BREAK has three variants:

— *Structural break*: BREAK picks a view $v \in V$ and a v edge connecting the node n_1 to its child node n_2. It replaces the view set V with $V \setminus \{v\} \cup \{v_1, v_2\}$, where: v_2 is a copy of the v subtree rooted at n_2, and v_1 is copy of v from which n_2's subtree is removed. BREAK also adds ID annotations to the copy of n_1 in v_1 and to the copy of n_2 in v_2. This ensures that any rewriting using v is still possible by replacing v with: $v_1 \bowtie_{n_1 \prec n_2} v_2$ if n_2 is a $/$-child of n_1, respectively, $v_1 \bowtie_{n_1 \prec\prec n_2} v_2$ if n_2 is a $//$ child of n_1.

— *Value-join break*: BREAK picks a view $v \in V$ and a j value-join edge connecting two nodes n_1, $n_2 \in v$. BREAK removes the j edge and adds val annotations to the nodes it used to connect. This may lead either to two distinct views $\{v_1, v_2\}$ (if the removed edge was the only one connecting them), or to a single view having one less value join.

— *Cartesian product break*: BREAK picks a view $v \in V$ and a cartesian product of two sub-views of v, $\{v_1, v_2\}$, such that $v \equiv v_1 \times v_2$. BREAK replaces v with $\{v_1, v_2\}$.

BREAK preserves rewriting power, since the broken join can always be reinforced, using either the ID, val attributes it introduced, or a cartesian product.

JOIN Opposite to BREAK, this transformation adds to the view set V the join of two views $v_1, v_2 \in V$. This transformation may reduce costs by pushing a join from the query into some view. For example, consider the views v_1: $//a_{ID}$ and v_2: $//b_{ID,Val}$ and the query q_1 $//a//b_{Val}$. To evaluate q_1, one has to scan both views, and join them as follows: $v_1 \bowtie_{a_{id} \prec\prec b_{id}} v_2$. Transformation JOIN adds to V the new view $v_{1,2} = //a_{ID}//b_{ID,Val}$, which is exactly the result of the join. Similarly, JOIN joins two views with a value-join or a cartesian product. Note that JOIN joins two views only if the resulting view respects the TRIMAXIS and NOCART pruning techniques of Section 3.3.2.

GENERALIZE GENERALIZE tries to identify commonality between workload queries by generalizing/relaxing a candidate view. Relaxing a candidate view may increase its space occupancy but it makes it more reusable. GENERALIZE has two variants:

— *$Cont$ generalization*: GENERALIZE picks a view $v \in V$ and a non-leaf node $n \in v$, and replaces v by a view v' in which the child subtrees of n have been erased and n has been annotated $cont$. For instance, if v is $//a[//b]//c[//d_{cont}]/e_{val}$, GENERALIZE may replace it with $//a[//b]//c_{cont}$ if the c node is chosen, or $//a_{cont}$ if the a node is chosen.

— *Val generalization*: GENERALIZE picks a view $v \in V$ and a node $n \in v$ which is annotated with a equality predicate of the form $n_{[val=c]}$ and replaces v

with a view v' in which:

1. node n is no longer annotated with an equality predicate and
2. node n is annotated val.

For instance, if v is $//a//b_{[val=3]}$, GENERALIZE replaces it with $//a//b_{val}$.

GENERALIZE applies to a view only if the resulting view respects the TRIMCONT and TRIMVAL pruning techniques. GENERALIZE preserves rewriting power. However, it can either increase or decrease the storage space, and thus rewriting cost.

ADAPT A candidate view may turn out to be more general than any of the queries into which the view embeds. In this case, transformation ADAPT adds a query-adapted view, typically smaller, which may also reduce query cost by removing the need for processing the view to adapt it to the query. Two variants of adaptation exist:

— Pick a view $v \in V$, a // edge e of v and an embedding $\phi : v \to q$ for some $q \in Q$. Denote by n_1 the node above e and by n_2 the node below e. If ϕ maps e either to (a) a / path or (b) a path of length greater than one, add to V the view v' obtained by copying v and in the copy, replacing e with the path to which e maps. For example, if the view v $/a//d$ embeds in the query $/a/b/c/d$, add the view v': $/a/b/c/d$.

— Pick a view $v \in V$, a $cont$-labeled node $n \in v$ and an embedding $\phi : v \to q$ such that $\phi(n)$ has some children. Add to V a view v' obtained by copying v and adding as children to (the copy of) n in v', all the child subtrees of $\phi(n)$ in q. For example, if the view $//a_{cont}$ embeds into the query $//a[//c/d]/b_{val}$, add the view $//a_{cont}[//c/d][b]$.

Observe that the symmetric situation to our first adaptation case, i.e., a view $//a/b_{val}$ and a query $//a//b_{val}$, does not occur, since such a view is not a candidate for the query (Section 3.3.1).

PROJECT When a view stores attributes not needed by any of the rewritings, we can remove these stored attributes to diminish view space occupancy. Transformation PROJECT picks a view $v \in V$ and replaces it with a view v' restricted to a subset of the stored attributes (ID, val, $cont$ or $[val = c]$) of v.

RANDOM Adding a candidate view to the materialized view set increases view storage size and may also increase benefit if the new view enables some rewritings efficient enough to offset the storage costs. Transformation RANDOM picks a candidate view not already in V, and adds it to the view set.
Our last transformations may trade rewriting power for space:

REMOVE and REMOVE0 Removing a view decreases view storage size and may reduce the benefit of a state. Transformation REMOVE picks a view and removes it from the view set, while REMOVE0 only removes zero-utility views. REMOVE0 preserves rewriting power and cost, while REMOVE is not guaranteed to do so. Both allow reducing space occupancy.

Importantly, REMOVE0 can be applied after a transition which has added a view v_1, to identify some view v_2 rendered useless by the addition of v_1. In this case, REMOVE0 eliminates v_2.
We also define some variations of our transformations. We consider the repeated exhaustive application of a transition τ, and use the shorthand $V_1 \xrightarrow{\tau*} V_k$ to state that repeated application of τ led from V_1 to V_2, from V_2 to V_3 etc. until V_k. In particular, REMOVE* repeatedly removes the least benefit view from a state V until it fits in the space budget S, REMOVE0* removes all unused views from a state V, while PROJECT* removes *all* unused attributes from *all* views $v \in V$. It can be shown that the state attained by REMOVE0*, or by PROJECT*, does not depend on the order in which the transformations are applied.

Which transition sets suffice to reach all candidate view sets? Since the answer depends on the candidate views and on the initial state, we include them in the definition:

Definition 3.4.2 (Transformation set completeness)**.** *Let Q be a workload, $\mathcal{CS} \subseteq \mathcal{CS}_0(Q)$ be a set of candidate views and V_0 an initial state in $\mathcal{CS}$. A set of transformations $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is $\mathcal{CS}$ and V_0-complete iff for any set of views $V \subseteq \mathcal{CS}$, there exists a sequence of states $V_0 \xrightarrow{\tau_{i_1}} V_1 \xrightarrow{\tau_{i_2}} \ldots \xrightarrow{\tau_{i_k}} V$, where for $1 \leq j \leq k$, $\tau_{i_j} \in \mathcal{T}$.*

Clearly, the set {RANDOM, REMOVE} is complete for any candidate view set $\mathcal{CS} \subseteq \mathcal{CS}_0(Q)$, and initial view set $V_0 \in \mathcal{CS}$, since RANDOM allows generating all possible candidate view sets. However, the transformation path from V_0 to V may be arbitrarily long. Below we present a more practical transformation set.

Proposition 3.4.1. *The transformation set* {BREAK, JOIN, GENERALIZE, PROJECT, REMOVE} *is $\mathcal{CS}_1(Q)$ and V_0 complete for $V_0 = Q$.*

The completeness follows from the ability of BREAK to break the initial view set all the way to one-node ID-annotated views, JOIN to glue back the pieces to build V, PROJECT out possible extra annotations, GENERALIZE to annotate view nodes with $cont$ or val. Extraneous views can be removed with REMOVE.

We end by noting that an exhaustive application of BREAK, JOIN, PROJECT, GENERALIZE and REMOVE on $V_0 = Q$ is still likely to be very costly, first, because of the large number of states reached, and second, because the calls to the rewriting algorithm (needed after *every* transformation) are very expensive, motivating further interest in searching for heuristics.

3.4.3.2 Reduce-Optimize Algorithm (ROA)

Based on the above transformations, we devised a search algorithm with a randomized component, which we term Reduce-Optimize Algorithm (or ROA, in short). ROA repeatedly executes two successive phases: first, the *reduce* phase which seeks to reduce the space occupancy of a state, by applying a chain of transformations on it; then, the *optimize* phase which attempts to increase the benefit

Algorithm 2: Reduce-Optimize Algorithm (ROA)

Input : Query workload $\mathcal{Q}$, candidate view set $\mathcal{CV}$, space budget S
Output: Best view set V_{best}

$V_{best} \leftarrow \emptyset$; $V \leftarrow Q$ // V is the current state
$V \leftarrow$ rewriteAndTrim(V, Q)
$\mathcal{S} \leftarrow \emptyset$ //the set of states on which a *reduce-then-optimize* sequence has been applied
while *!timeout* **do**
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{V\}$
  // *reduce* phase:
  foreach $\tau \in \{$BREAK, JOIN, GENERALIZE, ADAPT, REMOVE*$\}$ **do**
    $V' \leftarrow \tau(V)$ //apply τ to V
    $V' \leftarrow$ rewriteAndTrim(V', Q)
    if $size(V') < size(V)$ **then**
      $V \leftarrow V'$ //we found a smaller state, *reduce* will continue on this one
    else
      //ignore V', *reduce* will continue on V
    if $size(V) \leq S$ **then**
      break //end of *reduce* phase
  //*optimize* phase:
  foreach $\tau \in \{$ADAPT, JOIN$\}$ **do**
    $V' \leftarrow \tau(V)$ //apply τ to V
    $V' \leftarrow$ rewriteAndTrim(V', Q)
    if $b(V', Q) > b(V, Q)$ **then**
      $V \leftarrow V'$
  //seek a new state on which to apply *reduce-then-optimize*:
  while $V \in \mathcal{S}$ **do**
    //at most k attempts of adding a random view
    foreach $i \in 1, 2, \ldots k$ **do**
      $V \leftarrow V \cup \{v$ chosen at random from $\mathcal{CV}$, $v \notin V\}$
      $V \leftarrow$ rewriteAndTrim(V)
      if $V \notin \mathcal{S}$ **then**
        break
    if $V \in \mathcal{S}$ **then**
      $V \leftarrow$ REMOVE(V); $V \leftarrow$rewriteAndTrim(V)
return V_{best} //updated by calls to procedure rewriteAndTrim

of the target state. Both phases follow a trial-and-error approach, that is, an attempted transformation may fail to reduce space during *reduce*, or fail to increase benefit during *optimize*. If this happens, *reduce* (respectively, *optimize*) simply ignores the unsatisfactory target state and continues applying other transformations starting from the previously attained state.

One phase *may* also reach the desirable effect of the other, that is, *optimize*

Algorithm 3: Procedure REWRITEANDTRIM

Input : View set V, workload Q
Output: Restriction of V to the views and attributes needed by the best rewritings of Q. Side effect on V_{best}

```
1 E ← {rewrite(q, V)|q ∈ Q}
2 W ← PROJECT*(REMOVE0*(V)) //remove views and IDs not used in the best
  minimal rewritings
3 if cost(Q|_W) < cost(Q|_{V_best}) and size(W) ≤ S then
4     V_best ← W
5 return W
```

can reduce the space occupation of a state and *reduce* can increase the benefit of a state. However, most often, these objectives conflict, thus each phase strictly attempts to obtain one of the two improvements.

As described above, *reduce* and *optimize* explore the space in quite a linear fashion, that is, the fan-out of the search is low: if, say, the first transition attempted on V_0 during *reduce* does diminish space occupancy, we move to the resulting state V_1 and do not come back to V_0 to apply other transformations to it. A small search fan-out is desirable since exploring all possibilities would lead to too many states and unacceptably slow down the search. However, a disadvantage is that this leads to never visiting large parts of the search space and potentially missing interesting states. To cope with this, whenever *reduce* or *optimize* find a state on which the *reduce-optimize* sequence has already been applied, ROA jumps to a randomly chosen state, and continues the search from there.

As Algorithm 2 shows, the best state returned by ROA is stored in its local variable V_{best}. To simplify the description, we assume V_{best} is modified by the helper procedure **rewriteAndTrim** (Algorithm 3). This procedure is the only place where the (costly) **rewrite** algorithm of [MKVZ11] is called. After each call, the incoming view set V is trimmed down by exhaustively applying PROJECT and REMOVE0, and the trimmed-down version W is returned.

Algorithm's ROA exploration history is stored in the set $\mathcal{S}$ of all states on which *reduce* has been applied followed by *optimize*. During the *reduce* phase, it successively tries several transformations.

If a transformation reduces storage space, the next will follow from the reduced state, otherwise, ROA will apply it again on the start state V. The *reduce* phase ends either when 5 successive transformations have been applied, or when the size of the state found so far has decreased under the space budget S. Similarly, *optimize* attempts to apply two transformations to increase the current state benefit. Finally, after the two phases, we need to find a new state to work on. If the current state (reached at the end of *optimize*) has not yet gone through the two phases, ROA restarts *reduce* from there. Otherwise, we successively draw k random candidate views, and check if they enable new rewritings. Observe that **rewriteAndTrim** may remove these views, or views from V, when they are

rendered useless by a randomly-added view. We have empirically set k to be 40 which gave good results; a large k increases the chances of finding a good view but lengthens the search. Finally, if k successive view additions did not lead to a new state, we REMOVE some views until a non-visited state is reached. ROA needs to stop on a timeline, since completing its randomized search would take unacceptably long.

Remarks on the Implementation To increase ROA efficiency, we let it take hints from the query optimizer, in order to restrict the space of alternatives for its transformations. Specifically, the REMOVE0 and PROJECT transformations are only applied to remove attributes and views unused *by the best rewritings* of the workload queries (instead of "unused by any rewriting"). Similarly, JOIN only attempts to build view joins *that are part of some query's best rewriting*. This is possible thanks to the fact that rewritings are passed as algebraic expressions from the rewriter to the optimizer, and from there to the view selection.

Another concern is to be able to quickly identify (line 23) whether a state is already in $\mathcal{S}$. To efficiently support this, we index states by (string) signatures of their views, as follows. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a view set and assume first that each v_i is a (minimal) tree pattern [AYCLS02]. We compute serialized signatures of the V views $\{s(v_1), s(v_2), \ldots, s(v_n)\}$, and sort them into a list $s(v_{i_1}), s(v_{i_2}), \ldots, s(v_{i_n})$ where $(i_1, i_2, \ldots, i_n)$ is some permutation of $(1, 2, \ldots, n)$. Then, $\mathcal{S}$ can be organized as a multi-level hash structure where V is first indexed by $s(v_{i_1})$, then by $s(v_{i_2})$ and so on up to $s(v_{i_n})$. This structure allows determining with certainty by n look-ups whether a given state V of n *tree pattern* views has already been explored or not. In the general case (tree patterns with value joins), we encode a view of the form $t_1 \bowtie t_2 \bowtie \ldots \bowtie t_k$ as if it was the set of views $\{t_1, t_2, \ldots, t_k\}$. This introduces some imprecision in the state look-up, e.g., when looking up the view set $V_1 = \{t_1 \bowtie t_2\}$, one may (also) find the different view set $V_2 = \{t_1, t_2\}$. In this case (views including value joins), one still needs to check whether the state found by the multi-level look-up really is the same as the one we searched for, but overall, the search remains quite efficient.

3.5 Closest Competitor Algorithms

An early materialized view selection work for downward XPath (including wildcard $*$ nodes) is [MS05]. To select materialized views, they use an algorithm inspired from the greedy polynomial approximation of a set-cover problem [RN03], by defining view utility as the number of queries that it answers. Their set-cover greedy algorithm (which we denote **SCG** from now on) has an upper bound M on the number of views that can be recommended. In our implementation of SCG, to make a meaningful comparison, we dynamically set M to be the number of views selected by SCG that happen to reach our space bound $\mathcal{S}$.

More recently, [TYT+09] studied view selection for the same XPath dialect. From an XPath workload, they identify a subset of candidate views, consisting of

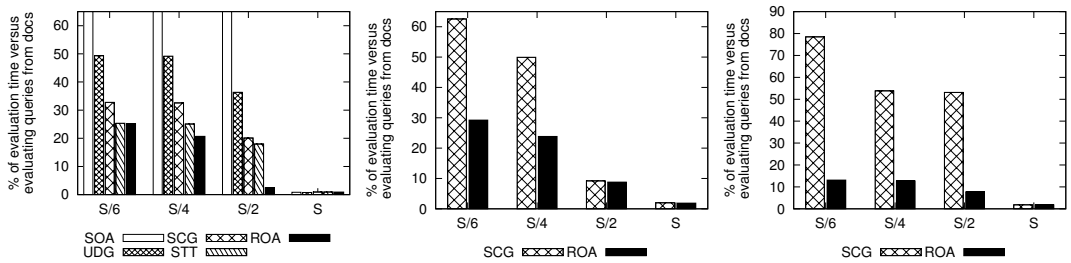


Figure 3.3: Workload execution time for the workloads Q_1, Q_2 and Q_3 (respectively, from left to right) based on views selected by various algorithms.

the minimal XPath queries based on which at least one query may be answered. This set can be organized as a lattice of size $2^{|Q|}$ which two algorithms rely on in order to recommend views. First, the dynamic-programming based *s*pace-*o*ptimized *a*lgorithm (denoted **SOA** in the sequel) searches for the smallest view set that can rewrite all the workload. Their second algorithm which we denote **STT** seeks to optimize a *s*pace/*t*ime *t*rade-off. It assigns to each view a benefit computed by summing the *weights* of the queries it can answer (regardless of the costs), divided by the view size. STT then greedily selects views in the decreasing order of their utility, until the space budget is filled up or all workload queries can be rewritten using the views.

Conceptually, the biggest difference between these and our algorithms is that they only apply for *XPath queries returning single nodes*. To compensate for this, we plugged in our implementation of SCG, SOA and STT the query rewriting, embedding etc. modules relevant for our language (Section 2.2.2).

A second important difference is: when considering XPath with one returning node, as [MS05] and [TYT$^+$09] do, *each query can only be re-written based on at most one view*, whereas we (as well as e.g., in [CDO08, ODPC06, TYÖ$^+$08]) consider query rewritings *based on multiple views*. This significantly complicates our setting, since for each query q and n candidate views, up to 2^n view sets may be used to rewrite q, instead of just n. Also, as our experiments will show, the algorithms [MS05, TYT$^+$09] by design do not capture the opportunities of multiple view-based rewritings, and in our setting, different algorithms exploiting these opportunities can achieve much better savings.

3.6 Experimental Evaluation

We now describe experiments we have performed with our and previous view selection algorithms. Section 3.6.1 outlines our software experimental framework, we describe the data and workloads in Section 3.6.2 and the algorithms with their settings in Section 3.6.3. Section 3.6.4 study candidate view set sizes, Section 3.6.5 algorithm effectiveness, and Section 3.6.6 their efficiency. We end with a conclusion of the experiments.

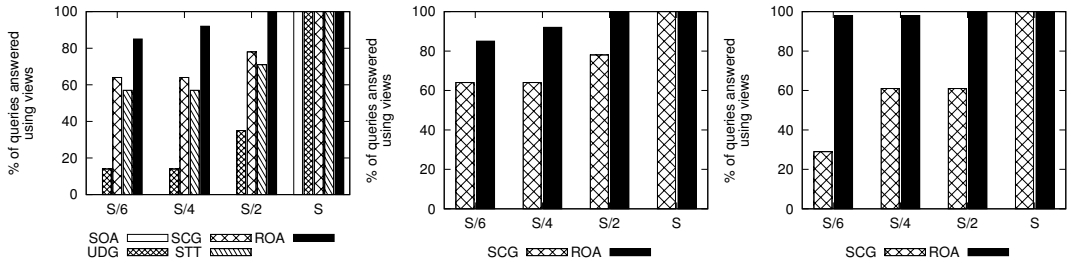


Figure 3.4: Fraction of queries from the workloads Q_1, Q_2 and Q_3 (respectively, from left to right), rewritten using the recommended views of different algorithms.

3.6.1 Framework

We have implemented our view selection algorithms within a Java-based XML data management platform that we developed. The platform supports the materialization of the complex XML views, featuring tree patterns with multiple returning nodes and value joins, described in Section 2.1.4.1. View tuples are stored into a native store that we built using the Berkeley DB library v3.3.75. The platform also provides a view-based rewriter module which, given a query q and a set of materialized views V, returns the best rewriting of q using views in V, as discussed in Chapter 2. Its optimizer takes as input the rewritings (logical plans over the views), pushes selections and projections, re-orders joins, identifies groups of binary structural joins to be transformed in an n-ary holistic twig join etc. Logical plans are then translated into physical plans including operators to: scan materialized views, apply selections, projections, value-based or structural joins (e.g., holistic joins [BKS02]), add Sort operators when needed etc. To evaluate tree patterns directly on the data, as well as nav operators, we implemented an efficient XML stream-based tree pattern matching algorithm [CDZ06]. The *physical plan cost estimation function* $cost^{\epsilon}$ takes into account the I/O cost of scanning views, as well as the CPU costs of selections, projections, joins, navigation, sort etc.

We have implemented our *view size estimation* function $size^{\epsilon}(v)$ (as described in Chapter 2, Section 2.3) based on a DataGuide [GW97] augmented with detailed statistics for each parent-child path p starting from the root of a document d: number of nodes on path p, minimum, average and maximum number of children of a node on path p on each child path, minimum and maximum string value of the nodes, number of distinct string values, average size of $cont$ etc. Our $size^{\epsilon}(v)$ function also makes simple independent-distribution and uniform-distribution assumptions. More elaborate estimations such as e.g., TreeSketch [PGI04] could easily plugged instead in our view selection approach.

3.6.2 Inputs: Data, Queries and Space Budget

We used 10 synthetic XMark benchmark [SWK+02] documents of 100 MB each resulting in a total of 1 GB.

Workload	$\|\mathcal{CS}_0\|^{max}$	$\|\mathcal{CS}_0\|^{max}$	$\|\mathcal{CS}_1\|$	$\|\mathcal{CS}_2\|$
Q_1	10^{12}	10^{13}	5944	992
Q_2	10^{12}	10^{13}	7582	1054
Q_3	10^{12}	10^{13}	10014	1454
Q_4	10^{16}	10^{16}	8570	1250

Table 3.1: Size of candidate view sets $\mathcal{CS}_0$, $\mathcal{CS}_1$ and $\mathcal{CS}_2$.

Four our tests, we generated four random query workloads based on the XMark document structure and content. First, we use three tree pattern query workloads Q_1 of 14 queries, Q_2 having 50 queries and Q_3 of 100 queries. We picked the size of Q_1 as the largest that all algorithms could handle, and Q_2, Q_3 to study further scale-up. Q_1, Q_2 and Q_3 only have tree pattern queries; each query has between 3 and 8 nodes. We added a fourth workload Q_4 of 50 queries, 10 of which have value joins; Q_4 queries have between 6 and 15 nodes. All queries have 2 to 4 returning nodes.

Within each workload, we varied query selectivity as follows. From each document (each $1/10$ of the data), 30% of the queries return just 1 result, 30% return a few hundred results, while 30% return a few thousand results. The remaining 10% of the queries return hundreds of thousands of results.

Clearly, materializing the workload is the best solution if the space budget allows it, but the interesting area is when this is not possible due to space constraints. For that purpose, we have taken $S = \sum_{q \in Q} size(q)$, and tested with the space budgets: $S/6$, $S/4$, $S/2$ and S. The main interest of the S value is to show the minimum possible query processing cost.

3.6.3 Algorithms and Settings

We have implemented our algorithms UDG (Section 3.4.2) and ROA (Section 3.4.3.2), as well as SCG [MS05], SOA and STT [TYT+09] discussed in Section 3.5. Our UDG algorithm is quite similar to STT [TYT+09]: beyond their rewriting differences, their benefits are different (our includes processing costs and query weights, theirs only query weights), but the greedy approach is the same.

The SCG, SOA and STT algorithms start with the workload itself. Concerning our own algorithms, we used $V_0 = \emptyset$ for UDG since it proceeds by adding views (thus we start it with all the allowed space free), and $V_0 = Q$ for ROA which is more powerful and can change (break, join, adapt etc.) views in many ways. We experimented with UDG and ROA both on the $\mathcal{CS}_1$ and $\mathcal{CS}_2$ candidate view sets. For the workloads we tested, they lead to similar results, thus we report on our UDG and ROA experiments using $\mathcal{CS}_1$ as a candidate view set.

We used a desktop having an Intel Xeon CPU 5140 @2.33 Ghz, 4 GB of RAM and a 60 GB SCSI hard disk at 10.000 RPM.

3.6.4 Candidate View Set Size

Our first experiment studies the size of the candidate view sets $\mathcal{CS}_0$, $\mathcal{CS}_1$ and $\mathcal{CS}_2$ (discussed in Section 3.3.2) for the four workloads. Exhaustive enumeration of $\mathcal{CS}_1$ views is not possible even for medium-size queries, e.g., for a tree pattern of 8 nodes, the $|\mathcal{CS}_0|$ bound is $25^8 \approx 152$ billions. Instead, we use a lower bound $\mathcal{CS}_0^{min}$ assuming that all workload queries are the same (thus, their candidate views overlap) and an upper bound $\mathcal{CS}_0^{max}$ assuming the queries have nothing in common (thus all candidate views are different). We built and counted the actual sets $\mathcal{CS}_1$ and $\mathcal{CS}_2$. Table 3.1 shows the candidate view counts. Candidate views are reduced by many orders of magnitude using the pruning techniques presented in Section 3.3.2. This makes view-based rewriting (and thus, candidate view search) feasible.

3.6.5 View Selection Algorithm Effectiveness

We ran the existing algorithms SCG, SOA, STT and our algorithms UDG and ROA on tree pattern queries, which they were all built for (modulo the many returning nodes, for which we adapted the competitors as explained in Section 3.5); we the workloads Q_1, Q_2 and Q_3. We then materialized their recommended views, evaluated the rewritings within our execution framework three times, measured the average time, and show it in Figure 3.3 as a percentage of the time to evaluate the queries directly on the database.

The first observation is that SOA, STT and UDG do not scale beyond Q_1. For SOA and STT, this is because they develop a candidate view set whose size is exponential in the size of the query, and this does not fit in the memory for larger workloads (similarly, [TYT+09] tests them on a 16-queries workload). For the 50-queries workload Q_2, our UDG algorithm had not finished running after two hours, when we stopped it. This is because UDG needs to update the benefit of every candidate view after each view addition, which in turn requires rewriting all the workload queries for every candidate view. Thus, for Q_2 and Q_3 we only measured SCG and ROA.

We also noticed that for Q_1, SOA failed to recommend any view (thus the evaluation time is plotted as 100% of the time to evaluate on the database directly) when given a space budget of at most $S/2$. This is because SOA only seeks view sets that can rewrite the whole workload, which for Q_1 could not be found at an $S/2$ space budget. In contrast, as expected, the greedy STT finds interesting view sets saving 50% of the execution costs or more.

Overall, for any workload and space budget, the ROA algorithm achieved the largest query processing cost reductions. This is because it exploits all rewriting possibilities: it tries to use both navigation (GENERALIZE transformation) and joins of several views (BREAK and JOIN transformations), aggressively prunes needless views and attributes (REWRITEANDTRIM), opportunistically modifies candidates to suit the queries (ADAPT transformation), and finally manages to visit sufficient parts of the search space through random jumps (RANDOM transformation). The

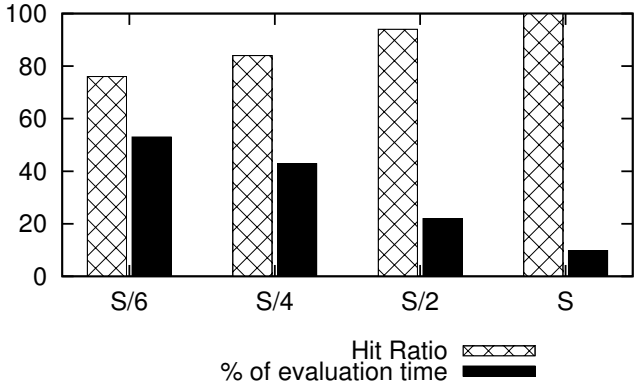


Figure 3.5: ROA performance on the workload Q_4.

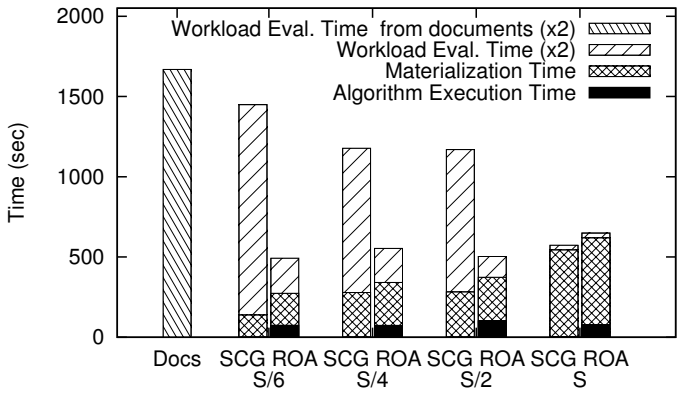


Figure 3.6: Overall performance of SCG and ROA.

greedy SCG does not perform as well because it does not consider multiple-view rewritings. We confirmed this intuition by inspecting the ROA-selected views: queries were typically rewritten using 2-3 views.

Figure 3.4 gives a different perspective on the same experiment: which fraction of the workload queries are rewritten based on the recommended views. ROA also is best in this respect.

For the workload Q_4 with value joins, we only ran ROA since it is the only one handling them. Figure 3.5 shows (in %) the workload evaluation cost reduction *wrt* evaluation on the database, and the ratio of queries rewritten by the ROA-selected views. This confirms the good properties of ROA also for queries with value joins.

	$S/6$			$S/4$			$S/2$		
	60%	80%	100%	60%	80%	100%	60%	80%	100%
Q_1	1	1	5	1	1	15	1	1	1
Q_2	1	1	25	1	1	25	1	2	2
Q_3	1	2	15	2	2	3	2	2	20
Q_4	1	1	13	1	1	1	1	1	1

Table 3.2: ROA time to attain increasing benefits (minutes).

3.6.6 View Selection Algorithm Efficiency

We now study the performance of the view selection algorithms themselves. Based on the findings of the previous section, we study only SCG and ROA, on the 100-query workload Q_3. Figure 3.6 depicts the time needed by SCG and ROA to (i) select the views to materialize (algorithm execution time). We let ROA run for 2 hours, and plot the time it needed to achieve 90% of its maximum benefit; (ii) materialize the views recommended by SCG and ROA (iii) evaluate all queries, *twice* based on the views. As a reference, we also show the time to evaluate all queries *twice* directly on the data. We timed two executions since views are typically materialized to support repeated query execution; in this case, two executions already enable the views to pay off, that is, the time to select, materialize, and exploit the views to evaluate the queries is smaller than the time to evaluate the queries directly on the database.

In this experiment, selecting, materializing and exploiting views paid off even for a single execution (except, of course, when materializing the workload itself). Moreover, SCG is much faster than ROA: SCG execution time is invisible in the Figure. This is because the greedy SCG never comes back on its decisions, whereas ROA investigates more complex view configuration settings, and may search for a long time due to its randomized component.

Showing the ROA time only up to attaining 90% of its biggest benefit may seem to give it an unfair advantage, since in practice we only stopped it after 2 hours. However: (i) increasing the view-based evaluation time in Figure 3.6 by a factor of $100/90$ does not change the overall picture and (ii) the robustness of the relatively quick cost reductions of ROA is confirmed by our next experiment.

Table 3.2 depicts the evolution of benefit through ROA execution. For each workload and space budget, we show the time (in minutes) it has taken ROA to attain 60%, 80% and 100% of the biggest benefit found in two hours. In all cases, 80% of the benefits were attained in just 2 minutes, while the maximum benefit was always attained within 25 minutes. While such times are still much longer than e.g. the greedy SCG, ROA recommends much better views. Moreover, view selection is typically an off-line process, thus we view the running times as acceptable.

3.6.7 Experiment Conclusion

Our experiments have shown that the candidate sets $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are of manageable size for workloads of up to 100 queries. Working on $\mathcal{CS}_1$, we have demonstrated that ROA and SCG scale up to 100 queries, whereas SOA and STT outgrow the available memory for 50 queries. Moreover, ROA achieved better savings (up to a factor of 8) and finds rewritings for more workload queries than its competitors. ROA (and SCG-) recommended materialized views lead to efficient execution; in our experiments materializing their views paid off starting from 2 workload runs. ROA's disadvantage is that being randomized, it needs to be stopped by a time-out, and is significantly slower than SCG. However, in practice, ROA achieves significant cost reductions (bigger than SCG) after relatively short times, of the order of minutes in our experiments. This confirms its interest for recommending views on complex XQuery workloads featuring many return nodes and value joins.

3.7 Related work

Our view selection approach bears similarities to those used in the relational databases [ACN00, GM99b, HRU96]: breaking and joining views to find common sub-expressions, and especially heavily relying on the (rewriter and) optimizer's recommended best plans, since a materialized view is only useful if the rewriting-optimization pipeline identifies recognizes it as such. A recent survey on view selection methods can be found in [MB12].

The complexity of XML data has lead to several index proposals, such as the DataGuide [GW97], indexes for navigation in a tree [KBNK02], adaptive path indexes of fixed length [QLO03] etc. Indexes can be seen as a specialized class of materialized views, based on which one only retrieves the identifiers of nodes that need to be retrieved from the store in order to return the query results. In contrast, we focus on materialized views that can help to completely answer complex queries, featuring multiple returning nodes and value joins. In the space of XML view-based query rewriting, closest to our work are the equivalent rewriting algorithms: for an XPath query using one view [LP08, MS05, XO05, YLH03], and for XPath/XQuery using several views [ABMP07, BOB$^+$04, CDO08, CC10, DT03, MKVZ11, ODPC06, TYÖ$^+$08]. In this work, we built a view selection framework that exploits the recent multiple-view equivalent rewriting algorithm of [MKVZ11], capable of handling tree patterns with multiple return nodes and value joins. We are the first to study the automated selection of materialized views in this context.

Among the XML view recommender systems, the closest works consider one-view XML query rewriting [MS05, TYT$^+$09]. We discussed them in Section 3.5, implemented adapted versions of their algorithms for our problem and show that our ROA algorithm scales better than [TYT$^+$09] which requires materializing an exponential-size lattice, and is more effective than [MS05] since it exploits multi-view, more sophisticated rewritings.

[CF10] studies view selection to support the reconstruction an XML subtree out of shredded data in relational tables. They show this is NP-hard, and present a PTime approximate solution. The focus in [EAZZ09] is on recommending relational DB2 XMLTable [1] materialized views for XQuery workloads. Their candidate views are inspired from the XPath snippets appearing in the for, where, let and return XQuery clauses; a transformation close to our GENERALIZE is applied to obtain more generic views. XQuery rewriting consists of translating XQuery into SQL queries over the XMLTable views and taking advantage of the XPath rewriting (based on one XPath views) supported by DB2. Their selection algorithm is a knapsack-style greedy, and experimented on a small workload of 10 queries. They explore a more limited space of alternatives, since they do not split and re-compose tree patterns through ID and structural joins, which ROA does extensively. Moreover, as we have shown, greedy algorithms (e.g., UDG) become impractical for complex view and query languages and multiple-view rewritings, since the repeated re-computation of benefit (through rewriting) takes prohibitive time.

3.8 Summary

In this chapter, we considered the selection of materialized views for an XQuery dialect consisting of joined tree patterns, and assuming a rich algebraic rewriting framework capable of value and structural joins, XPath navigation etc. We formalized the space of candidates which is extremely large and showed how to efficiently prune it. We have modeled the view selection problem as a state optimization problem and devised ROA, a heuristic algorithm that efficiently visits part of the search space of sets of candidate views. Finally, we have demonstrated that ROA scales easily up to 100 queries, whereas related works could scale up to 50. Moreover, ROA achieved better savings and finds rewritings for more workload queries than its competitors. This confirmed the interest of ROA for recommending views on complex XQuery workloads featuring many return nodes and value joins.

Acknowledgements We would like to thank Konstantinos Karanasos for his valuable help, comments and suggestions, Federico Ulliana for his input and advice and Julien Leblay for his proofreading and comments. This work was partially supported by the CODEX (ANR 08-DEFIS-004) and DataBridges (ANR 11-EITS-003) projects.

1. http://publib.boulder.ibm.com/infocenter/db2luw/v9/

Chapter 4

Distributed View-based Web Data Dissemination

The growing volumes of XML data sources on the Web or produced by enterprises, organizations etc. raise many performance challenges for data management applications. In this chapter, we are concerned with the distributed, peer-to-peer management of large corpora of XML documents, based on distributed hash table (or DHT, in short) overlay networks. We present ViP2P (standing for *Vi*ews in *P*eer-to-*P*eer), a distributed platform for sharing XML documents based on a structured P2P network infrastructure (DHT). At the core of ViP2P stand *distributed materialized XML views*, defined by arbitrary XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. ViP2P allows user queries to be evaluated over XML documents published by peers in two modes. First, a long-running subscription mode, when a query can be registered in the system and receive answers incrementally when and if published data matches the query. Second, queries can also be asked in an ad-hoc, snapshot mode, where results are required immediately and must be computed based on the results of other long-running, subscription queries. ViP2P innovates over other similar DHT-based XML sharing platforms by using a very expressive structured XML query language. This expressivity leads to a very flexible distribution of XML content in the ViP2P network, and to efficient snapshot query execution. ViP2P has been tested in real deployments of hundreds of machines. We present the platform architecture, its internal algorithms, and demonstrate its efficiency and scalability through a set of experiments. Our experimental results outgrow by orders of magnitude similar competitor systems in terms of data volumes, network size and data dissemination throughput.

ViP2P has been the main subject of previous publications and a thesis [Zou09]. This thesis has mainly focused on communication protocols and optimizations of central parts of the ViP2P platform. Moreover, an extensive validation and experimentation of the platform has been performed. The results of these efforts have been published in a paper [KKMZ12] and a research report [KKMZ11]. To make this thesis self-contained, we provide a complete description of the architecture

and design of ViP2P.

4.1 Motivation and Outline

The volumes of data sources available in the form of XML documents has exploded since the W3C's 1998 standard, and so have the languages, tools and techniques for efficiently processing XML data. The interest of distribution in this context is twofold. First, a distributed storage and processing network can accommodate data volumes going far beyond the capacity of a single computer. Second, as organizations and individuals interact more and more, sharing and consuming one another's information flows, it is often the case that (XML) data sources are produced independently by several distributed sources [ÖV11]. The set of producers and consumers of data related to a specific topic, e.g., IT journals, blogs and online bulletins, is not only distributed, but also dynamic: sources may join or leave the system, the set of information consumers or their topics of interest may also change in time etc. Thus, we are interested in the large-scale management of distributed XML data in a *peer-to-peer* (P2P) setting. To provide users with *precise, detailed and complete* answers to their requests for information, we adopt a database-style approach where such requests are formulated by means of a structured query language, and the system must return complete results. That is, if somewhere in the distributed peer network, an answer to a given query exists, the system will find it and include it in the query result. To achieve this, our goal is to build a P2P XML data management platform based on a distributed hash table (or DHT, in short [DZD+03]).

In this setting, users may formulate two kinds of information requests. First, they may want to *subscribe to interesting data anywhere in the network*, and published before or after the subscription is recorded in the system. Our goal is to persist the subscriptions and ensure that results are eventually returned as soon as possible following the publication of a matching data source. This is in the spirit, e.g., of RSS feeds, but extended to a distributed network where the source from which interesting data will come is not a priori known. Second, users may formulate *ad-hoc (snapshot)* queries, by which they just seek to obtain as fast as possible the results which have already been published in the network.

The challenges raised by a DHT-based XML data management platform are:

— building a *distributed resource catalog*, enabling client producers and consumers to "meet" in the virtual information sharing space; such a catalog is needed both for subscription and ad-hoc queries,

— efficiently distributing the data of the network to the consumers that have subscribed to it and

— providing *efficient distributed query evaluation algorithms* for answering ad-hoc queries fast.

In this chapter, we present ViP2P, standing for *Views in Peer-to-Peer*, a distributed P2P platform for sharing Web data, and in particular XML data. ViP2P

is built on top of a structured P2P network infrastructure, and it allows each peer in the network to share data with all the other peers. Data sharing in ViP2P is twofold. First, each network peer can ask long-running queries which are treated as subscriptions, that is, they receive results if and when a document published in the system matches such queries. Second, once results are stored for such a subscription, they are treated as *materialized views* based on which subsequent ad-hoc queries can be processed with snapshot semantics, i.e., based only on the data already published in the network. Given such an ad-hoc query, a ViP2P peer looks up the ViP2P network for relevant materialized views, runs an algorithm for equivalently rewriting the query, identifies and evaluates a distributed query evaluation plan which, based on the views, computes exactly the results of the query on the data published in the system prior to the query. ViP2P thus fills two kinds of needs: (i) disseminating information in a timely fashion to subscriber peers and (ii) re-using pre-computed results to process ad-hoc queries efficiently on the existing data only.

A critical issue when deploying XML data management applications on a DHT is the division of tasks between the DHT and the upper layers. The DHT software running on each machine allows peers to remain logically connected to each other and to look up data based on search keys: a small set of simple, light-weight operations. In contrast, powerful XML data management requires complex languages (such as the W3C's XPath and XQuery standards), and scalable algorithms to cope with complex processing and large data transfers (known to raise performance issues in any distributed data management setting).

Experience with our previous DHT-based XML data management platform KadoP [AMP+08] has taught us to load the DHT layer *as little as possible*, and keep the heavy-weight query processing operations in the data management layer and outside the DHT. This has enabled us to build and efficiently deploy a system of important size (70.000 lines of Java code), which, as we show, scales on up to 250 computers in a WAN, and hundreds of GBs of XML data. ViP2P improves over the state of the art in DHT-based XML data management, since: (i) it is one of the very few systems actually implemented (together with [AMP+08, RM09b], and opposed to prototypes built on DHT simulators), (ii) is shown to scale on data volumes that are orders of magnitude beyond the cited competitor systems and (iii) has the most expressive XML query language, and the most advanced capabilities of re-using previously stored XML results, among all similar existing platforms [AMP+08, BC06, BMCJ04, GWJD03, KP05, LP08, RM09b].

ViP2P is part of a family of systems aiming at efficient management of XML data in structured peer-to-peer networks [AMP+08, BC06, BMCJ04, GWJD03, KP05, LP08, RM09b, RM09a]. The contributions of this work, with respect to the existing systems, are as follows:

— We present a *complete architecture for query evaluation, both in continuous (subscription) and in snapshot mode*. This architecture enables the efficient dissemination of answers to tree pattern queries (expressed in an XQuery dialect) to peers which are interested in them, regardless of the relative

order in time between the data and the subscription publication. As in [LP08], it also allows to efficiently answer queries in snapshot mode, based on the content of the existing views materialized in the network, but using more expressive views, queries and rewritings.

— We have fully implemented our architecture (about 250 classes and 70.000 lines of Java code), on top of the FreePastry [Fre] P2P infrastructure. We present a *comprehensive set of experiments performed in a WAN*, showing that (i) the performance of a fully deployed large-scale distributed system (and in particular a DHT-based XML management platform) is determined by many parameters, beyond the network size and latency which can be set in typical P2P network simulators and (ii) the ViP2P architecture scales to several hundreds of peers and hundreds of GBs of XML data, both unattained in previous works.

This chapter is organized as follows: Section 4.2 surveys the state of the art in managing XML data in DHT networks and Section 4.3 introduces the ViP2P architecture via an example and describes its main modules. Section 4.4 concentrates on the materialization, indexing and look-up of materialized views, at the core of the platform. Finally, in Section 4.5, we present a set of experiments analyzing the performance of ViP2P data management in a variety of settings and demonstrating its scalability, then we conclude.

4.2 State of the Art

In this section we present the current state of the art in XML data management over P2P networks. In Section 4.2.1 we focus on the differences of structured and unstructured P2P networks and the reasons behind our choice to use a structured P2P network for building our platform. In Section 4.2.2 we present our closest competitor works focusing on the management of XML data over structured DHT networks. Section 4.2.3 stresses the challenges of distributed XML data management in a real, deployed platform as opposed to simulations. Finally, in Section 4.2.4, we present earlier publications of the ViP2P platform.

4.2.1 P2P Data Sharing Networks

Peer-to-peer content sharing platforms can be broadly classified in two groups. Unstructured peer-to-peer networks allow arbitrary connections among peers, that is, each peer may be connected to (or aware of the existence of) one or more network peers of its choice. Such network structure typically mimics some conceptual proximity between peers interested, for instance, in similar topics. Structured peer networks, on the other hand, impose the set of connections among peers. A survey of (structured and unstructured) P2P XML sharing platforms reflects the state of the art and open issues as of 2005 [KP05] and a more recent survey of XML document indexing and retrieval in P2P networks can be found in [Abe11].

The different network structures impact the way in which searches (or queries) can be answered in the network. Thus, in unstructured networks, queries are forwarded from each peer to its set of known peers (or neighbors) and answers are computed gradually as the query reaches more and more peers. For instance, in [SMGC05] peers are logically organized into clusters that are formed on a document schema-similarity basis. The superpeers of the network are organized to form a tree, where each superpeer hosts schema information about its children. When a query arrives it is forwarded to the superpeers. Every superpeer performs location assignment: it examines the schemas of the documents of its children to detect which peers could possibly contribute results to the query. After the contributing peers have been located, the peer that originally posed the query builds a location aware algebraic plan and ships the corresponding subqueries to their respective peers. The results are then retrieved from each peer and the original query is evaluated by performing operations such as joins over the subquery results.

It is easy to see that if query answers reside on a peer very far (in terms of peer connections) from the peer where the query originated, this may lead to numerous messages and a long query response time. To improve the precision, performance and recall of query answering in this context, many approaches have been proposed, from the earliest [YGM03] to the very recent [DLAV10], to name just a few.

In contrast, structured networks (and their best-known representatives, distributed hash tables or DHTs, in short [DZD+03]) provide a simple distributed index functionality implemented jointly by all the peers. The simplest DHT interface provides *put(key, value)* and *get(key)* operations allowing the storage of (key, value) pairs distributed over all the network peers. More advanced DHT structures also allow range searches of the form *get(key range)*, such as Baton [JOT+06, JOV05] or P2PRing [CLM+04, CLM+07]. In a DHT, to answer a *get* request, a bounded number of messages are exchanged in the network, typically in $O(log_2(N))$, where N is the number of network peers.

In this work, we consider the setting of a structured network, based on a DHT, and design an efficient platform for *XML query processing in large scale networks, based on P2P XML materialized views*. The main difference between most of the existing platforms and ViP2P is that our system addresses the whole processing chain involved in evaluating queries, as opposed to only locating the interesting documents and shipping the query to those peers for evaluation. The latter approach may, in some cases, require numerous messages at query evaluation time and possibly increased response times. ViP2P, in contrast, considers the complete chain of query processing based on materialized views incrementally built in the network. This enables answering queries by contacting only a few peers and possibly re-using complex pre-computed results, stored in the views.

4.2.2 XML Data Management Based on DHTs

The first DHT-based platform for XML data management was [GWJD03]. This work proposed a framework for indexing XML documents, based on the parent-child element paths appearing in the document. Processing a query involves (i) extracting from the query a set of paths which could serve as lookup keys, (ii) obtaining via *get* calls the IDs of all documents matching the paths, (iii) shipping the query to all the peers holding such documents and (iv) retrieving the results at the query peer. The approach carries some imprecision in the case of queries featuring the descendant axis (//) or tree branches. For instance, the query /a[b]/c could be forwarded to documents in which the paths /a/b and /a/c occur, but the tree pattern /a[b]/c does not occur. A very similar approach to DHT-based XML indexing by parent-child paths is taken in [SHA05].

The above discussion illustrates a common aspect in DHT-based content management platforms: imprecision in the indexing method leads to more peers being contacted to process a given query. A previous work on managing relational data based on DHTs [LHH+04] has shown that intensive messaging at query time may seriously limit scaling. Therefore, index precision is generally a desirable feature.

The work described in [BC06, BMCJ04] considers the setting where XML documents are divided in fragments distributed among several peers. Each fragment is assigned as identifier the parent-child label path going from the document root to the root of the fragment, and subsequently, fragments are indexed in the DHT by their identifiers. The system uses a particular DHT which can handle *prefix queries*, and thus allows locating XML fragments for which a prefix of the path from the root to the fragment is known. Processing linear queries using only the child axis is simple, however, simple queries using the descendant axis, such as the query //a, need to be forwarded to all the network peers.

The KadoP system [AMP+08] indexes XML documents at fine granularity. Thus, for any element name a, a network peer is in charge of storing the identifiers (or IDs, in short) of all a-labeled elements from all the documents in the network. The IDs reflect the position of the elements in the respective documents. Therefore, any tree pattern query can be answered by retrieving the list of IDs corresponding to each tree pattern node, and combining these lists via a holistic twig join [BKS02]. This indexing model has very high precision, since the output of the holistic twig join includes exactly the documents matching the query. However, the index is much more voluminous than in previous proposals [BC06, BMCJ04, GWJD03, SHA05], highlighting the severe limitations *in terms of volume of the (key, value) pairs* of the DHT index. Several optimizations in the index structure were introduced in [AMP+08], based on which the KadoP platform was tested on hundreds of peers and 1GB of data.

More recently, the *psiX* system [RM09a, RM09b] proposed an XML indexing scheme based on document summaries, corresponding to the backward simulation image of the XML documents (if a DTD is available, summaries can also be built based on the DTD). An algebraic signature is associated to each summary and to each query. When a query arrives, the algebraic query signature is used to look

up in a holistic fashion all document signatures matching the query. The precision of this indexing scheme improves over KadoP [AMP+08] by a better treatment of wildcard ($*$) nodes, which KadoP ignores for the most part of query processing. From the matching summaries, one can identify the concrete corresponding documents, and then push query evaluation to the peers hosting the documents. The approach is implemented over the Chord DHT and shown to be effective by experiments on up to 11 peers in the PlanetLab network.

The main difference between the works described in [GWJD03, RM09a, RM09b, SHA05] and our work lies in the approach taken for query processing. These works, of which *psiX* [RM09b] can be considered the most advanced, are only concerned with locating the documents relevant for a query. In contrast, [BC06, BMCJ04], KadoP [AMP+08] and the ViP2P platform presented here address the P2P XML query processing problem as a whole. They re-distribute data in the P2P network in order to prepare for the evaluation of future queries. KadoP distributes a tag index over the peers independently of the data and the queries, which can be seen as a "one size fits all" approach. ViP2P allows individual peers to choose the particular queries of interest for them, expressed in a rich tree pattern dialect (or, equivalently, a useful XQuery subset) and then allows exploiting the stored results of such queries as views for rewriting future queries. An ongoing development of ViP2P [CRKMR10] focuses on automatically choosing the views to materialize on each peer in order to improve observed query processing performance. Thus, going beyond the problem of *locating* relevant documents, ViP2P aims at making the most out of the existing network storage and processing capacity in order to *evaluate queries* most efficiently to the peers that need them.

Closer in spirit to our work is the cooperative XPath caching approach described in [LP08], where peers can store results of a (peer-chosen or system-imposed) XPath query. The definitions of these stored queries (or views) are indexed in the network, enabling subsequent queries to be rewritten and answered based on these views. ViP2P is more general, since (i) our view and query language is an XQuery dialect with many returning nodes, as opposed to the simple XPath subset in [LP08] and (ii) our approach allows to rewrite a query based on *several* views, whereas [LP08] can only exploit one view for one query.

DHT-based XML indexing methods [AMP+08, BC06, BMCJ04, GWJD03, RM09a, RM09b, SHA05] are *complete*, i.e., for each query, based on the index, all relevant answers can be computed and returned. In ViP2P and [LP08], peer-chosen views replace the compulsory index fragments assigned by the network to each peer. Thus, it is possible that some queries cannot be processed due to the lack of appropriate views. Our focus in ViP2P is on efficiently building and exploiting pre-computed query results under the form of materialized views. To guarantee completeness, our approach can be coupled with an efficient and compact document-level index, such as *psiX* [RM09b], on which to fall back when no suitable views are found for a given query.

We conclude our analysis by considering the *granularity* or level of detail used to index XML, i.e., the granularity of the keys inserted in the DHT. Element la-

bels (or label paths, or document summaries) have been often used. However, this does not allow efficiently locating documents which satisfy specific *value or keyword* search conditions, such as e.g., //item[price=$45] or //item[contains(.,'camera')]. Indexing by keywords or text nodes increases index precision but also significantly increases the index size, since there are many more keywords in an XML document than distinct tags. Therefore, the approaches of [BC06, BMCJ04, GWJD03, RM09a, RM09b, SHA05] cannot be easily extended to support keyword search and preserve their scalability. A *value summary framework* is proposed in [GWJD03] to index element values by trading off precision for index space. KadoP [AMP+08] indexes all keywords just like element labels, and proposes index-level optimization techniques to cope with important scale-related problems. ViP2P allows keyword and value conditions both in the materialized views and in the queries.

4.2.3 Managing XML on a DHT: Platforms vs. Simulations

Developing distributed systems, and in particular a P2P platform, requires significant efforts. This may be a reason why many previous works in this area validate their techniques based on *simulated peer networks*, where a single computer runs an analytical model configured to simulate a given network size. Our INRIA team has invested significant manpower (of the order of 70 man $\times$ month by now) developing the KadoP and then the ViP2P platforms. Our effort has taught us that many architecture and engineering problems arise due to the mismatch between the initial DHT goals (maintaining large dynamic networks connected and providing minimal messaging), and the data-intensive operations required by indexing, storing, and querying large volumes of XML data. We have addressed these problems in ViP2P by careful architecture and engineering, and report in this chapter *experiments at a scale (in peers deployed over a WAN, and in data size) unattained so far by any other platform*. Thus, KadoP [AMP+08] scales up to 1 GB of data over 50 computers peers, psiX [RM09b] used 262 MBs of data and 11 computers, and in this chapter we report on sharing up 160 GB of data over up to 250 computers (in all cases, the computers were distributed in a WAN).

4.2.4 Previous Publications on ViP2P

A first version of the platform was described in an informal setting (no proceedings) in an international workshop [MZ09b] and a national conference [MZ09a]. These works used a more restricted query language than we consider here, and described early experiments on a platform which has been much improved since. Two ViP2P applications have lead to demonstrations: P2P management of RDF annotations on XML documents [KZ10] and adaptive content redistribution [CRKMR10]. The details of view-based query rewriting in ViP2P are described in a separate paper [MKVZ11]. They can be seen as orthogonal to the architecture and performance issues described here.

4.3 ViP2P Platform Overview

XML data flows in ViP2P can be summarized as follows. XML documents are published independently and autonomously by any peer. Peers can also formulate subscriptions, or long-running queries, potentially matching documents published before, or after the subscriptions. The results of each subscription query are stored at the respective peer, and the definition of the query is indexed in the peer network. Finally, peers can ask ad-hoc queries, which are answered in a snapshot fashion (based on the data available in the network so far) by exploiting the existing subscriptions, which can be seen as materialized views. We detail the overall process via an example in Section 4.3.1. We then proceed to describe the ViP2P modules implementing it in Section 4.3.2.

4.3.1 ViP2P by Example

A sample ViP2P instance over six peers is depicted in Figure 4.1 and we use it to base our presentation of the operations which can be carried in each peer. In the Figure, XML documents are denoted by triangles, whereas views are denoted by tables, hinting to the fact that they contain sets of tuples. More details on views and view semantics are provided in Section 4.4, but they are not required to follow the discussion here. For ease of explanation, we make the following naming conventions for the remainder of this chapter:

— **Publisher** is a peer which publishes an XML document
— **Consumer** is a peer which defines a subscription and stores its results (or, equivalently, the respective materialized view)
— **Query peer** is a peer which poses an *ad-hoc* query (to be evaluated over the complete ViP2P network).

Clearly, a peer can play any subset of these roles simultaneously or successively.

4.3.1.1 View Publication

A ViP2P view is a *long-running subscription query* that any peer can freely define. The *definition* (i.e., the actual query) of each newly created view is indexed in the DHT network. For instance, assume peer p_2 in Figure 4.1 publishes the view v_1, defined by the XPath query $//bibliography//book[contains(.,'Databases')]$. The view requires all the books items from a bibliography containing the word 'Databases'. ViP2P indexes v_1 by inserting in the DHT the following three (key, value) pairs: $(bibliography, v_1@p_2)$, $(book, v_1@p_2)$ and $('Databases', v_1@p_2)$. Here, $v_1@p_2$ encapsulates the structured query defining v_1, and a pointer to the concrete database at peer p_2 where v_1 data is stored. As will be shown below, all existing and future documents that can affect v_1, *push* the corresponding data to its database.

Peers look up views in the DHT in two situations: when publishing documents, and when issuing ad-hoc queries. We detail this below.

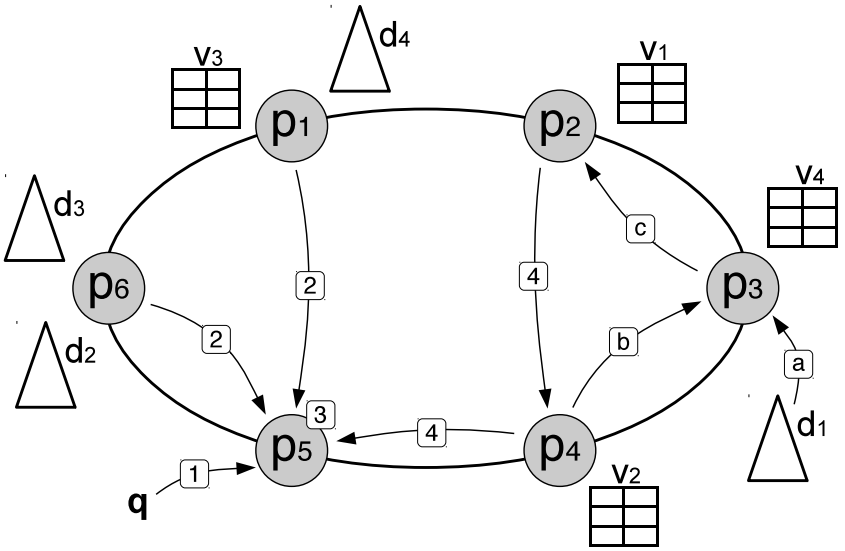


Figure 4.1: System overview.

4.3.1.2 Document Publication

When publishing a document, each peer is in charge of identifying the views within the whole network to which its document may contribute. For instance, in Figure 4.1 (step a), peer p_3 publishes the document d_1 (depicted in Figure 4.2). Document d_1 contains data matching the view v_1 as it contains the element names $bibliography$ and $book$, as well as the word $'Databases'$. Peer p_3 extracts from d_1 all distinct element names and all keywords. For each such element name or keyword k, p_3 looks up in the DHT for view definitions associated to k, and, thus, learns about v_1 (step b). In the publication example above, p_3 extracts from d_1 the results matching v_1; from now on, we will use the notation $v_1(d_1)$ to designate such results. Peer p_3 sends $v_1(d_1)$ to p_2 (step c), which adds them to the database storing v_1 data.

A separate mechanism is needed for a view, say v_x, published after d_1 but having results in d_1. One possibility would be for the peer publishing v_x to look up, among all the network documents, for those that could contain terms from v_x and require them to contribute v_x results. The drawback is that this requires indexing all documents on all terms, which may be wasteful since a large part of published content may not be looked up frequently, or not at all.

Instead, ViP2P associates to each view an *interval timestamp*, corresponding to a time interval during which the view was published. Each peer having published a document d must check the DHT for views published after d. To that effect, each peer performs regular lookups using as key, the time interval which has just passed. Thus, it retrieves the definitions of all the views published during that interval and contributes its data if it hasn't done it already.

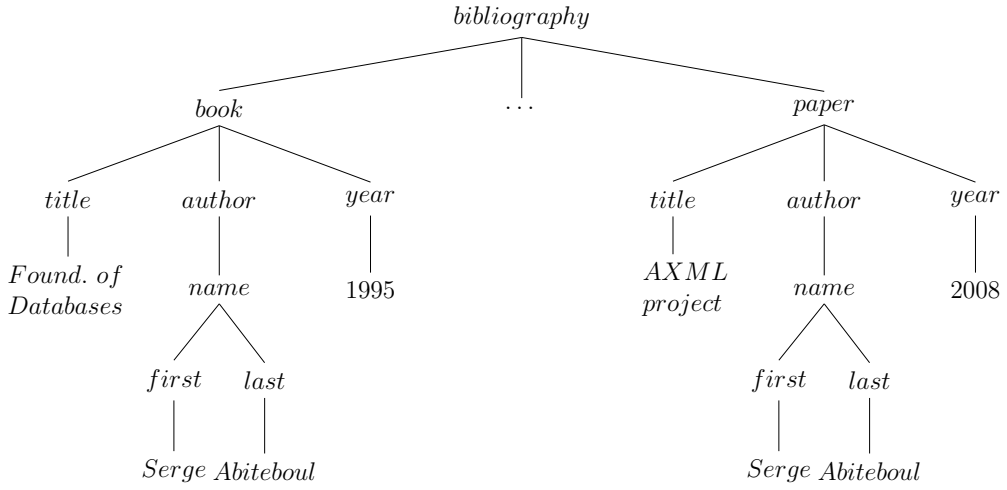


Figure 4.2: Sample XML document d_1.

4.3.1.3 Ad-hoc Query Answering

ViP2P peers may pose *ad-hoc queries*, which must be evaluated immediately (from the previously published data). To evaluate such queries, a ViP2P peer looks up in the network for views which may be used to answer it. For instance, assume the query $q = //bibliography//book[contains(.,'Databases')]//author$ is issued at peer p_5 (step 1, in Figure 4.1). To process q, p_5 looks up the keys $bibliography$, $book$, $'Databases'$ and $author$ in the DHT, and retrieves a set of view definitions (step 2), including that of v_1. Observe that q can be rewritten as $v_1//author$; therefore, p_5 can answer q just by retrieving and extracting q's results out of v_1. A distinguishing feature of ViP2P (step 3) is its ability to combine *several* materialized views in order to rewrite a query. A query rewriting (a logical plan based on some views) is translated by the ViP2P query optimizer into a distributed physical plan, specifying which operators will be used and in which peers they will be executed. The ViP2P optimizer is responsible for selecting the most efficient physical plan, as this choice has a significant impact in the query execution time, especially in a distributed setting such as ours where network communication plays an important role. The execution of the physical plan may require the cooperation of various peers, and leads to results being sent at the query peer (step 4).

4.3.2 ViP2P Peer Architecture

We now present the main modules of ViP2P peers as well as their functionalities and interaction, outlined in Figure 4.3. The *ViP2P Core* box includes the main modules, whereas boxes located outside *ViP2P Core* are independent external subsystems that interact with ViP2P.

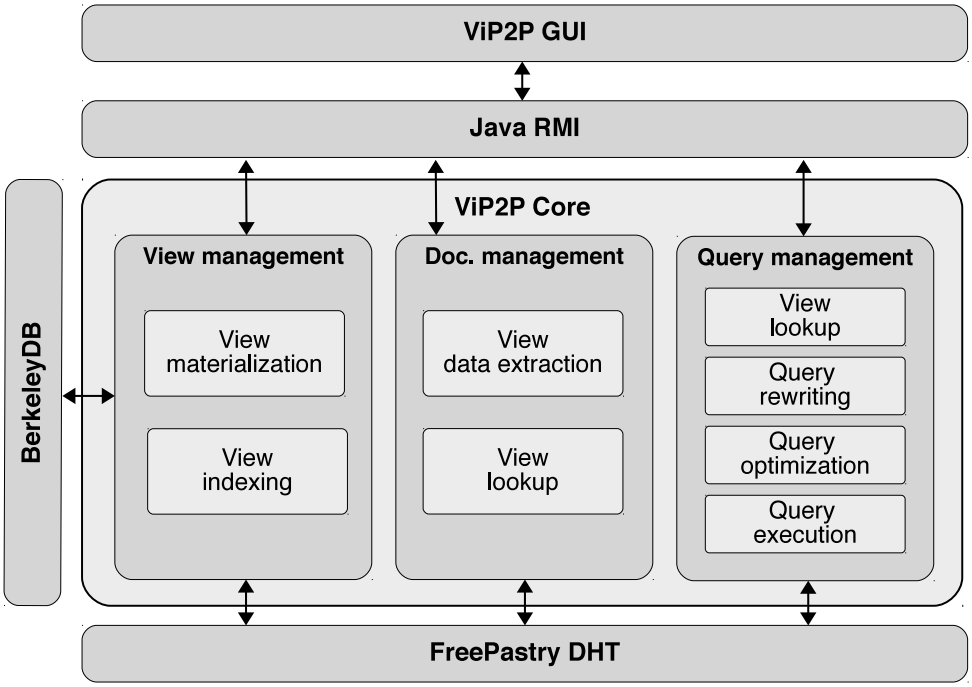


Figure 4.3: Basic architecture of a ViP2P peer.

4.3.2.1 External Subsystems

FreePastry DHT [Fre] provides the underlying DHT layer on which ViP2P is built. FreePastry is an open-source implementation of Pastry [RD01], an efficient, self-organizing and fault-tolerant overlay network. Pastry provides efficient request routing, deterministic object location, and load balancing. ViP2P nodes index and lookup view definitions on FreePastry's DHT during the view materialization and query processing.

Java RMI is used for all large data transfers. Previous work [AMP+08] has shown that the DHT communication primitives were not suitable for such transfers, since (i) the DHT *get* and *put* operations are blocking, that is, data sent via the DHT becomes available at the receiver only when it has been completely received and (ii) message queues in the DHTs overflow easily even after tuning, in which case the DHT peers re-send them, which further clogs the DHT communication pipes. Beyond the degradation of performance, such message overflows are annoying because a peer that is too busy trying to re-send data, may skip sending the regular "ping" to his neighbors to signal that it is still alive. Then, the neighbors suspect the peer is down, this triggers further loss of messages etc.

For all these reasons, we have decided to split inter-peer communication in two categories. The DHT is used to efficiently send small messages, typically to index and look up view definitions. We use RMI (which we were able to fine-tune by writing efficient custom serialization/de-serialization methods, properly con-

trolling concurrency at the send and receiver side etc.) to send larger messages containing view tuples, when views are materialized and queried. We also applied specific techniques to reduce the space occupancy of transmitting tuples. Thus, a document ID (or URI) often appears many times in a view, as many times as there are view tuples obtained from that document. Since the URIs are quite large, they make up an important part of the document data. We use dictionary-based encoding of the document URIs, i.e., the tuple sender dynamically builds a dictionary of all document URIs and sends partial dictionaries with each tuple packet, to enable decoding on the receiver side. One could perhaps improve performance even further by coding data-intensive communications at a lower level (e.g. using plain sockets), but the improvements attained by our way of utilizing RMI are already very significant.

BerkeleyDB Within each peer, view tuples are efficiently stored into a native store that we built using the Berkeley DB [BDB] library. It provides the routines to store, retrieve and sort entries, while guaranteeing ACID transactions when view data are written and read concurrently.

The GUI facilitates the control and inspection of each peer, enabling users to publish views and/or pose queries. Screenshots of the ViP2P GUI, along with other information, can be found on the ViP2P website [1].

We now move to describing the core modules.

4.3.2.2 Document Management Module

This module is responsible for looking up for views to which the peer's documents may contribute, extracting the data from the documents and sending it to the respective consumers.

View Definition Lookup When a new document is published by a peer, the *view lookup* module at this peer first, looks up in the DHT the definitions of the views to which the document may contribute data, and then passes these views definitions to the *view data extraction module*.

View Data Extraction Given a list of view definitions, the *view data extraction module* at a publisher peer extracts from the document the tuples matching each view, and ships them, in a parallel fashion, to the different consumers. The view data extractor is capable of simultaneously matching several views on a given document. Thus, the corresponding tuples are extracted during a single traversal of the document. The extractor maintains a thread pool for setting up RMI communications for shipping tuples to the consumers. As our experiments show in Section 4.5.3, this parallel tuple sending significantly reduces the time needed to materialize the views.

1. http://vip2p.saclay.inria.fr/

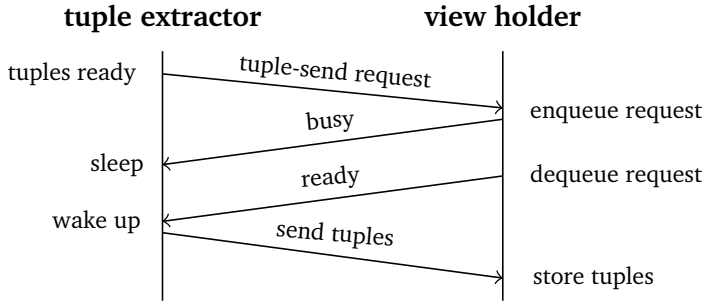


Figure 4.4: Tuple-send/receive protocol use case between a tuple–sender and a view holder.

4.3.2.3 View Management Module

This module handles view indexing and materialization.

View Indexing This module makes visible to all network peers the definitions of all the views declared in the ViP2P network (of course without broadcasting them, since most peers are typically not interested in all views). When a new view is defined, the indexer inserts in the DHT (key,value) pairs used to describe it, based on one of the *indexing strategies* that we will describe in Section 4.4.1.

View Materialization The *view materialization* module receives tuples from remote publishers and stores them in the respective BerkeleyDB database. In a large scale, real-world scenario, thousands of documents might be contributing data to a single view. To avoid overload on its incoming data transfers, this module implements a back-pressure *tuple-send/receive protocol* which informs the publisher when the incoming tuple buffer is full at the consumer side. Thus, a publisher may have to wait until the consumer is ready to accept the tuples. This makes the most out of the available publisher-to-consumer bandwidth, all the while avoiding costly re-transmissions due to messages lost from overflowing queues.

Figure 4.4 traces the tuple-send/receive protocol between a tuple extractor and a view holder. First the tuple extractor extracts the tuples and keeps them in memory being ready to ship them to the view holder. After that, it sends a tuple-send request to the view holder. In this example, the view holder is busy storing tuples (possibly sent by other tuple extractors), thus it enqueues the request and responds to the tuple extractor with a "busy" response. When the view holder is ready to accept the new set of tuples, it dequeues the request and informs the tuple extractor (via a "ready" message). Then, the tuple extractor ships the new tuples to the view holder, who finally stores them in the Berkeley database of the respective view. The view holder can serve multiple tuple-send requests concurrently. Our experiments (Section 4.5.3) show how the concurrency can affect the time needed for a set of views to be materialized.

4.3.2.4 Query Management Module

A sequence of steps are required to evaluate queries, each performed by a dedicated module, as follows.

View Lookup This module, given a query, performs a lookup in the DHT network retrieving the view definitions that can be used to rewrite the query.

Query Rewriting This module takes a given ad-hoc query and a set of available view definitions and produces a logical rewriting plan which, evaluated on some views, produces exactly the results required by the query.

Query Optimization This module receives as input a logical plan which is output by the query rewriting module and translates it to an optimized physical plan. The optimization takes place both at the logical (join reordering, push selections and projections etc.) and physical (dictating the exact flow of data during query execution, selection of appropriate physical operators etc) level.

Query Execution This module provides a set of physical operators which can be executed by any ViP2P peer, implementing the standard iterator-based execution model [Gra90]. Since ViP2P is a distributed application, operators can be deployed to peers and executed in a remote manner. The query optimization module is the one to decide the parts of a physical plan that every peer executes.

Data exchange operators are an essential part of a distributed execution plan. To that end, ViP2P implements two data exchange operators: the *Send* and *Receive* operators that permit data exchange across peers. They are always used in pairs: whenever a data sender peer executes a *Send* operator, the data receiver executes its respective *Receive* operator. *Send* and *Receive* are implemented using asynchronous communication buffers (tuples are not sent through the network one by one but in buckets of specified size) and data is transferred via RMI. To reduce the transferred data volumes, document URIs (present in each view tuple to identify the document the tuple was extracted from) are compressed using a dictionary by the *Send* and decompressed by the *Receive* as described in Section 4.3.2.1.

ViP2P implements the typical *Selection*, *Projection*, *Hash Join*, *Nested Loop Join* and *Merge Join* operators. Moreover, it uses the XML specific operators *Holistic Twig Join* [BKS02], *Structural Ancestor Join* and *Structural Descendant Join* [AKJP+02] performing structural joins based on the structural identifiers (IDs) of the incoming tuples. The *Navigation* operator corresponds to the logical navigation operator, described in Section 2.2.2. Two sorting operators are available: an in-memory sort operator *Memory Sort*, and an external memory sort based on BerkeleyDB.

4.4 ViP2P View Management

Materialized views stand at the heart of data sharing in ViP2P. sections 4.4.1 and 4.4.2 show how view definitions are indexed and looked up in the DHT in order to be retrieved for view materialization and query rewriting, respectively.

4.4.1 View Definition Indexing and Lookup for View Materialization

This section describes how published data and views "meet", i.e., how ViP2P ensures that for each view v, the data obtained by evaluating v over d, denoted $v(d)$, is *eventually* computed and stored at the peer having defined v. Two cases arise, depending on the publication order of v and d.

View Published Before the Document In this case, the view definitions are indexed using as keys all the labels (node names and words) of the view. Figure 4.6 shows eight views. To index v_1, ViP2P issues the calls $put(book, v_1)$ and $put(title, v_1)$ to the DHT. Observe that these calls index the *definition* of v_1 (*not* its data) on the keys $book$ and $title$. Similarly, v_2 is indexed on the keys $book$, $author$ and $last$, v_3 using the keys $paper$, $author$ and $last$ etc. When the document in Figure 4.2 is published, *get* calls are issued with the keys $bibliography$, $book$, $paper$, $title$, $author$, $year$, $Found.\ of\ Databases$ and all the other labels and keywords of the document. The result is a superset of view definitions of the views that the document might affect. In this case the views v_1 to v_8 are retrieved.

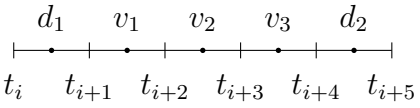


Figure 4.5: Sample timeline of view and document publication.

View Published After the Document ViP2P ensures that views are kept up to date (providing some time for the data to circulate across the network). Thus, when a view is published, it should be filled in with data from all the previously published documents matching the view. To achieve this, ViP2P associates to each view an *interval timestamp*, corresponding to a time interval during which the view was published, and indexes each view definition in the DHT using as key the corresponding timestamp. As illustrated in Figure 4.5, v_1 belongs to (was published in) the interval $(t_{i+1}, t_{i+2}]$, v_2 to the interval $(t_{i+2}, t_{i+3}]$ and v_3 to $(t_{i+3}, t_{i+4}]$.

Each peer having published a document d must check the DHT for views that may have appeared after d. To that effect, each peer performs regular lookups using as key the time interval that has just finished. This retrieves the definitions of all views published during that interval. The peer then checks, for each of its documents, if the document has already contributed to that view (this information

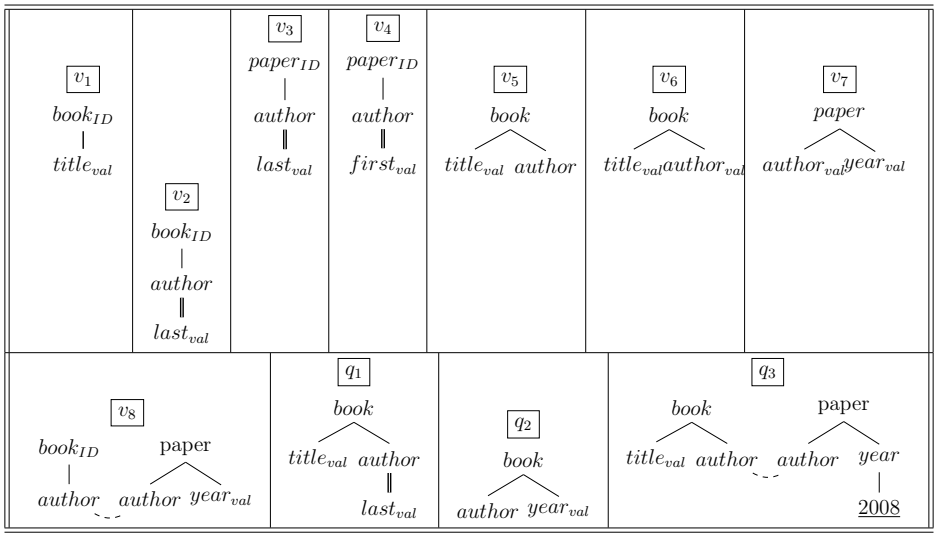


Figure 4.6: Sample views and queries.

is stored locally at the peer). If this is not the case, the peer checks if that document holds any data for these views and if so, extracts and sends the corresponding data to the view holder. In Figure 4.5, document d_1 arrives during the $(t_i, t_{i+1}]$ time interval. With the help of the timestamped view index, we discover the views v_1, v_2 and v_3 which arrived later. Notice also that document d_2 is published after the views and thus is treated according to the first case above.

4.4.2 View Definition Indexing and Lookup for Query Rewriting

View definitions are also indexed in order to find views that may be used to rewrite 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 show that these strategies are *complete*, i.e., they retrieve at least all the views that could be embedded in q and, thus, could potentially lead to q rewritings.

4.4.2.1 Label Indexing (LI)

In this strategy we 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 $O(|v|)$, where $|v|$ the number of nodes of the view.

View Lookup for LI The lookup is performed by all node labels of q. The number of lookups is $\Theta(|q|)$, where $|q|$ is the number of nodes in the query. Figure 4.6 depicts some sample queries. The LI lookup keys for q_1 are $book$, $title$, $author$ and $last$, retrieving all the views of Figure 4.6. Note that some of these cannot be used to equivalently rewrite q_1. For instance, v_3 has data about papers, while q_1 asks for books. Similarly, LI indexing and lookup for q_2 and q_3 leads to retrieving all the views. This shows that LI has many false positives.

LI Completeness If LI is not complete, then there exists a view v that can be used to rewrite a query q, and v is not retrieved when searching by all q labels. It has been shown [TYÖ+08] that in order for a view to appear in an equivalent rewriting of a query, there must exist an embedding (homomorphism) from the view into the query, which entails that some node labels must appear in both. If in our case v and q have no common node label, this contradicts the hypothesis that v was useful to rewrite q.

The LI strategy coincides with the view definition indexing for document-driven lookup (described previously). An interesting variant can furthermore be elaborated.

4.4.2.2 Return Label Indexing (RLI)

Here, we index v by the labels of all v nodes which project some attributes (at most $|v|$). For instance, in Figure 4.6, the index keys for v_1 are $book$ and $title$, for v_2 they are $book$ and $last$, for v_3 $paper$ and $last$ etc. up to v_8 which is indexed by RLI on the keys $book$ and $year$.

View Lookup for RLI The view definition lookup is the same as for LI (look up on all query node labels). In Figure 4.6, the definitions of $v_1 - v_3$, and $v_5 - v_8$ will be retrieved for q_1. For q_2, the definitions of v_1, v_2, v_6, v_7 and v_8 will be retrieved. A RLI lookup for q_3 will retrieve $v_1 - v_8$. Observe that RLI lead to less view definitions retrieved than LI.

RLI Completeness Suppose that there is a view v which can be used to rewrite a query q, yet the definition of v is not retrieved by RLI lookup. This means that either (i) v does not store any attributes or (ii) the labels of v nodes that project an attribute do not appear in q. (i) is not possible because a view that participates to a rewriting should store at least an attribute and (ii) is also not possible since it contradicts the existence of an embedding from v to q, required for v to be useful in rewriting q.

4.4.2.3 Leaf Path Indexing (LPI)

Let $LP(v)$ be the set of all the distinct root-to-leaf label paths of v. Here, a path is just the sequence of labels encountered as one goes down from the root to the node, and does not reflect the type of the edges. We index v using each element of $LP(v)$ as key. The number of (key, value) pairs thus obtained is in $\Theta(|LP(v)|)$.

Going back to Figure 4.6, v_1 is indexed on the key $book.title$, v_2 with the key $book.author.last$ etc. The view v_8, composed of two tree patterns, is indexed using the keys $book.author$, $paper.author$ and $paper.year$.

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. For example, q_1 of Figure 4.6 LPI lookup uses the keys $book.title$, $book$, $title$, $book.author.last$, $book.author$, $author.last$, $book.last$, $book$, $author$ and $last$ etc. Note that LPI lookup for q_1 does not retrieve the definitions of the views v_3, v_4, and v_7, which previous strategies retrieved, although they are not useful to rewrite q_1. LPI can still have some false positives though: a lookup for q_2 retrieves v_5, v_6 and v_8, none of which can be used to rewrite q_2 (in this example, q_2 simply has no rewriting). The lookup for q_3 retrieved the views v_1, v_5, v_6, v_7 and v_8. The filtering is very good in this case because among these only v_5 can not be used to rewrite q_3.

Let $h(q)$ be the height of q and $l(q)$ be the number of leaves in q. The number of LPI lookups is bound by $\Sigma_{p \in LP(q)} 2^{|p|} \leq l(q) \times 2^{h(q)}$. If the query q is a join of tree patterns (tpqs) then the bound becomes $\Sigma_{tpq \in q}(\Sigma_{p \in LP(tpq)} 2^{|p|})$.

LPI Completeness is guaranteed by the fact that if a view v can be embedded in the query q, then $LP(v) \subseteq SP(q)$.

4.4.2.4 Return Path Indexing (RPI)

RPI is the last strategy that we consider. 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 $\Theta(|RP(v)|)$. The indexing keys for v_1 are $book$ and $book.title$, for v_2 are $book$ and $book.author.last$ etc.

View Lookup for RPI coincides exactly with the lookup for LPI. The lookup of q_1 retrieves the definitions of the views v_1, v_2, v_5, v_6 and v_8, the same as LPI. For q_2, RPI lookup retrieves the definitions of v_1, v_2, v_6 and v_8. Observe that unlike LPI, RPI in this situation does not return v_5, which indeed is not useful! We end by noting that this increase of precision of RPI over LPI is not guaranteed. For example, an RPI lookup for q_3 retrieves the definitions of all views in Figure 4.6, which is much less precise than LPI.

RPI Completeness is established in a similar fashion to the LPI case.

4.5 Experimental Results

In this section we present a set of experiments studying ViP2P performance. Section 4.5.1 outlines the experimental setup. ViP2P attempts to speed up query processing by exploiting pre-computed materialized views. This shifts the complexity of extracting and sending interesting data across the network, from query

processing to view materialization, to which we devote the most attention in our experiments. Several parameters determine view materialization performance: the distribution of the documents and views in the network, the documents which contribute to each view, the documents and views size etc. Section 4.5.2 starts by studying view materialization in the context of a single peer. Then, Section 4.5.3 examines view materialization in the large, in widely different network configurations, varying the number and the distribution of publisher and consumer peers. Section 4.5.4 presents an evaluation of the indexing strategies for query rewriting presented in Section 4.4.2. Finally, Section 4.5.5 presents experiments that evaluate the performance of the query execution engine.

4.5.1 Experimentation Settings

Infrastructure Setup We have carried our experiments on the Grid5000 infrastructure (https://www.grid5000.fr), providing computational resources distributed over nine major cities across France. Figure 4.7 shows Grid5000 network topology. Sites are interconnected with a 10Gbps network and within each site, nodes are interconnected with (at least) 1Gbps Ethernet network. The hardware of Grid5000 machines varies from dual-core machines (of at least 1.6 GHz clock speed) with 2GBs of RAM to 16-core machines with 32GBs of RAM. We settled for a random and heterogeneous distribution of hardware, in order to be close to real P2P deployment scenarios. However, in some experiments, we deliberately choose sites being very far away from each other, almost being the two opposite ends of the network, to show the scalability of our platform in the most difficult scenarios imagined within the Grid5000 network.

Data Generation To have fine control over all the parameters impacting our experiments, we have used synthetic data, produced by two existing XML data generators: ToXGene [BMKL02] and MemBeR [AMM05].

Experimentation Parameters We summarize the main parameters characterizing our experiments in Table 4.1. For each set S, we use $|S|$ to denote the size of the set. Thus, $|P|$ is the number of peers in the network etc. Finally, for a document d, we use $|d|$ to denote the size of d, measured in Megabytes (MBs).

Evaluation Metrics In our measurements, we use the following metrics to characterize the system performance:

— **Materialization time** is the time needed for the network to materialize a set of views populating them with the data extracted by all the documents published in the network. The materialization time starts at the time instance that a peer initiates the first extraction of data and ends at the time that all peers have extracted and shipped the tuples to the appropriate view holders.

— **Tuple extraction time** for a view v and a document d is the time needed for the publisher of d to extract from d the tuples which make up $v(d)$.

— **Storage time** for a document d and a view v is the time taken by the con-

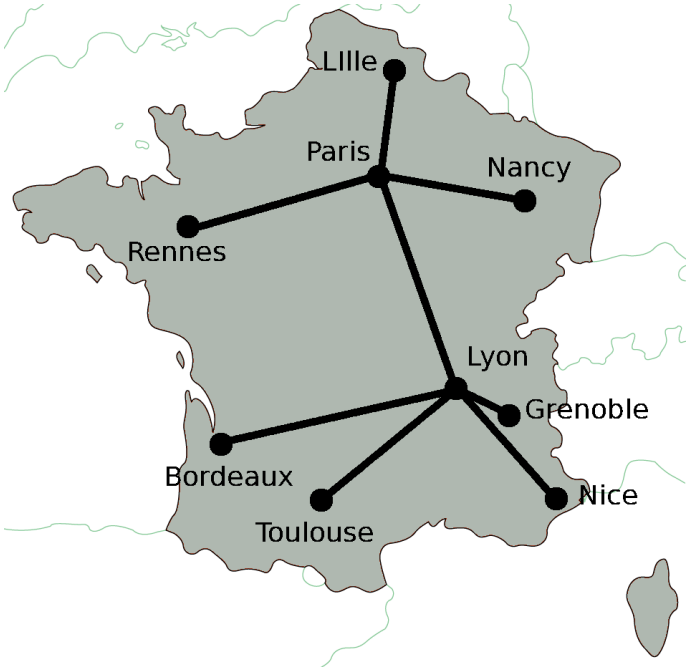


Figure 4.7: Grid5000 network topology.

sumer holding v, to add to the corresponding BerkeleyDB database the set of tuples corresponding to $v(d)$.

— **Data exchange time** for a document d and view v is the time needed for the tuples $v(d)$ to be transferred across the network from the publisher of d to the consumer holding v.

— **Lookup time** for a query q is the time needed for the peer asking q to lookup in the DHT the views that may be useful to rewrite q.

— **Embedding time** for a query q and a set of views V is the time needed by the query peer to verify which of the views may actually be used to rewrite q. Recall from Section 4.4.2 that this is established by checking for the presence of embeddings between each view $v \in V$ and the query q [TYÖ+08].

— **Query response time** for a query q is the time elapsed between the moment when the query has been posed, and the moment when its execution has finished (as observed at the query peer).

— **Time to first result** for a query q is the time between the moment when the query has been posed, and the moment when its first result tuple has been received at the query peer.

Whenever the query, view, or document are not specified for a given metric, *the metric value is understood to be the sum, over all the documents, views, and queries used in the respective experiment, of the respective metric, with the exception of the materialization time*. By nature, this metric accounts for many materialization processes running *in parallel*, and therefore is not the sum of individual materialization times. For instance, assume that publisher p_1 publishes a document which

Symbol	Description
P	The set of peers in the network
P_D	The set of peers holding at least one document
V	The set of views in the network
P_V	The set of peers holding at least one view
D	The set of all published documents
D_V	The set of documents matching at least one view

Table 4.1: Parameters characterizing the experiments.

contributes data to a view at p_2, while publisher p_1' similarly contributes to a view at p_2'. The peers p_1 and p_1' will start at about the same time the materialization process by looking up views to which they could contribute etc. One of them will be the last to report that all its tuples have been stored and acknowledged by the respective consumer peer. The materialization time of this experiment spans between the first materialization start event, and the last materialization end event, while the two processes run in parallel.

4.5.2 View Materialization Micro-benchmarks

We start by studying the performance of extracting from a document d, the tuples corresponding to a view v, and sending these $v(d)$ tuples from the peer holding d to the one storing v. To focus exactly on the process of extraction, we use very simplistic network settings. View materialization in more complex settings and larger scale will be studied next.

Experiment 1: Sequential vs. Parallel Extraction of Views As described in Section 4.3.2, a ViP2P peer p is capable of simultaneously matching several views $v_1, v_2, \ldots, v_k$ on a given document d residing at p. The corresponding tuples $v_1(d)$, $v_2(d)$, ..., $v_k(d)$ are extracted during a single traversal of the document d, instead of k traversals (one for each of the k views). This is important when publishing a document d in case the publisher finds out that many previously defined views could match d, and therefore it has to match all of them against d. While parallel extraction is faster, it may require more memory, since matches for the various views have to be constructed and kept in memory at the same time.

Our first experiment studies the effect of extracting data for several views in parallel. We use a document d and two distinct sets of views. First, we consider a four-view set of the form $\{//t_{i\,ID}\}$ for $i \in \{1, \ldots, 4\}$. Second, we consider a larger set including views of the form $\{//t_{i\,ID}\}$ for $i \in \{1, \ldots, 8\}$. The views and d are chosen so that d contributes 130,000 tuples to each published view v_i. The parameters characterizing the experiment are as follows:

$\lvert P\rvert$	$\lvert P_D\rvert$	$\lvert V\rvert$	$\lvert P_V\rvert$	$\lvert D\rvert$	$\lvert D_V\rvert$	$\lvert d\rvert$
2	1	{4, 8}	1	1	1	100MB

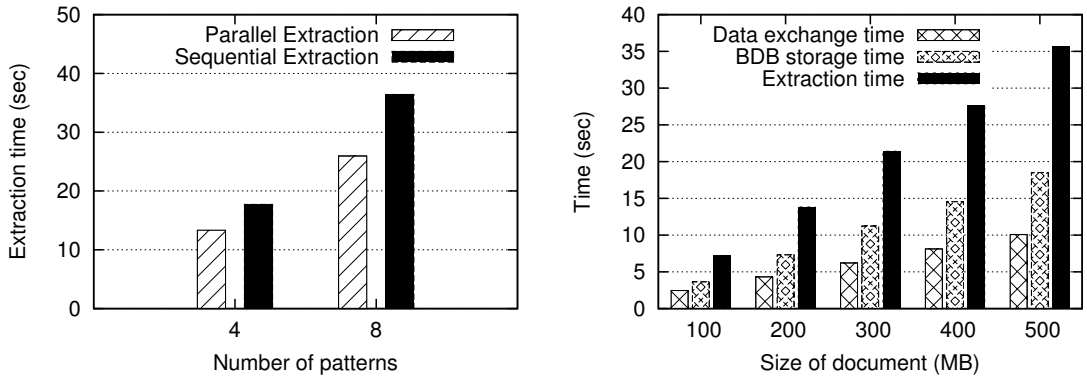


Figure 4.8: Experiment 1: parallel vs. sequential extraction time (left); experiment 2: view materialization over different-size documents (right).

Figure 4.8 (left) depicts the extraction time when extracting data out of d for four and for eight views, in a parallel and sequential fashion. We observe that parallel extraction accelerates data extraction (in this case, up to 40%). Therefore, we will always use parallel extraction in the subsequent experiments.

Experiment 2: Studying One Data Transfer Pipe We now study the materialization of documents of various sizes, in order to identify the bottleneck of the materialization process. Possible bottlenecks are (i) data extraction at the document publisher; (ii) network bandwidth between a consumer and a publisher; (iii) view storage time at the consumer. For this experiment, the following parameters are used:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
2	1	1	1	1	1	$\{100, \ldots, 500\}$MB

One peer plays the role of the publisher, while the other is the consumer. The peers are located at two opposite ends of France (Lille and Grenoble). The document and the view are chosen so that the complete content of the document is extracted and sent to the consumer, thus, the materialized view size increases linearly to the size of the document.

Let us now detail the synchronization of the various processes involved when a publisher sends data to a consumer to be added in a view.

1. The publisher extracts data locally. *After* all the tuples from $v(d)$ have been computed, the publisher starts sending them to the consumer [2].
2. Packets of tuples are sent over the network to the consumer in an asynchronous way using buffers at the consumer side.

2. This could be improved to parallelize extraction and sending in some cases, but there are fundamental limitations: for some of the views we support, one needs to wait for the full traversal of the document before producing an output tuple [GNT09].

3. At the consumer, a thread picks packets of tuples from the buffer and stores them in the BerkeleyDB database.

The buffer at the consumer can be parameterized to control the data transfer speed: when the buffer is full because the storage thread is not sufficiently fast, data transfer stalls. For this experiment, the size of the data buffer was set to *unlimited* (making sure in advance that the memory of the consumer is enough to store all the produced tuples), so that the data exchange thread can use as much as possible of the available bandwidth between the two peers.

Figure 4.8 (right) depicts the time needed for the view tuples to be (i) extracted from the document, (ii) sent over the network and (iii) stored in BerkeleyDB at the consumer. We observe that the three times increase linearly in the size of the data. Data extraction is the slowest component, however, overall, times were comparable (also recall that the network connection is fast within the Grid, thus transfer times may be higher in other contexts).

Conclusion From the above two experiments, we conclude that (i) parallelizing data extraction does speed up the time to compute view tuples; (ii) extraction time grows linearly to the size of the input document and (iii) data transfer and data storage time grow linearly with the size of the extracted tuples.

4.5.3 View Materialization in Large Networks

We now consider view materialization in larger and more complex environments, with many publishers and/or many consumers.

Documents For these experiments, we needed to tightly control which parts of the published data are relevant to which views on each peer. Therefore, unless stated otherwise, we rely on documents whose shape is outlined on the left of Figure 4.9. There are always 64 camera elements under one $catalog$, and each $camera$ has 4 children. To obtain different document sizes, we insert text of varying length in the $description$ of each $camera$.

Experiment 3: One Publisher, Fixed Data, Varying Number of Consumers In this experiment, we use a single publisher, a fixed data set (5 documents of 50 MBs each), and a varying number of consumers (from 1 to 64). Each consumer always holds exactly one view. All the published data is relevant for some view; moreover the view contents do not overlap, i.e., the data is practically "partitioned" over the views. Thus, when there is a single consumer, its view stores the $cont$ of all cameras from the catalog. When there are two consumers, the view of the first consumer stores the $cont$ of the cameras from camera_1 to camera_{32}, while the other consumer's view stores the $cont$ of the rest of the cameras (camera_{33} to camera_{64}) and so on. This way, the views absorb all the data published. The producer is located in Lille and the consumers in Sophia-Antipolis (two opposite ends of France). The parameters values for this experiment are given in the table below:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	1	{1,2,4,...,32,64}	$\{1, 2, 4, \ldots, 32, 64\}$	5	5	50MB

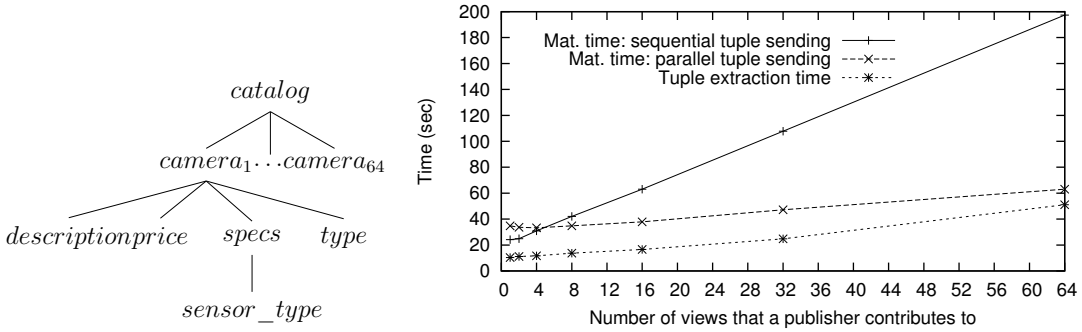


Figure 4.9: Outline of a controlled synthetic document for our experiments (left); experiment 3: view extraction and materialization time depending on the number of consumers (right).

Once the tuples are extracted by a publisher, they can be shipped to the view holders sequentially (the publisher contacts the consumers one after the other) or in parallel (the publisher ships all the tuples to all consumers concurrently). At right in Figure 4.9, we show the time needed to extract the tuples, and the materialization time for the two variations of tuple sending: sequential or parallel. In both cases, as expected, the extraction time is the same and it increases linearly with the number of consumers.

When sending tuples sequentially, we observe that the materialization time increases linearly with the number of consumers (views). In the case of 64 consumers, data extraction takes about 45 seconds, but materialization takes about 200 seconds. Materialization time increases drastically with sequential tuple sending since more and more consumers need to be contacted one after another.

When sending tuples in parallel, we observe that the materialization time is notably lower than in the case of sequential tuple shipping and that its slope is almost the same as the one of the extraction time. This is because, as soon as the tuples are extracted, a pool of threads (one thread for each packet of tuples) takes over the task of shipping all the tuples in parallel. The bottleneck in this situation is the upload link of each consumer.

Experiment 4: One Publisher, Varying Data Size, 64 Consumers We study how materialization time is affected when the total size of published data is increased. We use one publisher. The size of the published data varies from 64MBs to 1024MBs.

Each of the 64 consumers holds one view of the form $//catalog//camera_{K\ cont}$ where K varies according to the peer that holds the view. For example, the first consumer holds the view $//catalog//camera_{1\ cont}$, the second holds the view $//catalog//\ camera_{2\ cont}$ etc. This way, from each document the publisher extracts 64 tuples, each of which is sent to a different consumer. All the content of the

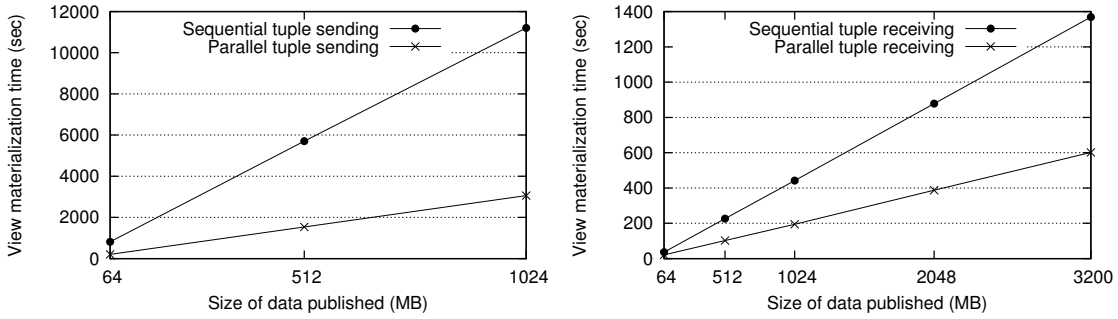


Figure 4.10: Experiment 4: one publisher, varying size of data, 64 consumers (left); experiment 5: 64 publishers, varying data size, one consumer (right).

documents is absorbed by the 64 views. The parameter values used for this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	1	64	64	{64,512,1024}	{64,512,1024}	1MB

Like in Experiment 3, we run two variations of the same experiment: (i) one for sequential tuple sending and (ii) one for parallel tuple sending. The graph at left in Figure 4.10 shows, as expected, that the materialization time increases linearly with the size of data published in the network in both cases. It also shows that the materialization time in the case of parallel tuple sending is considerably shorter (about 3000 sec. instead of 11500 sec. for absorbing 1024MBs of data).

Experiment 5: 64 Publishers, Varying Data Size, One Consumer We now study the potential for parallel publishing, i.e., the impact of the number of (simultaneous) publishers on the capacity of absorbing the data into a single view. The published data size varies from 64MBs to 3.2GBs, and all the published data ends up in the view. The parameter values for this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	64	1	1	{64,...,3200}	{64,...,3200}	1MB

Recall from Section 4.3.2 that the view materialization module maintains a queue of tuple-send requests and allows only a certain number of concurrent tuple-extractors to send data to it concurrently. In this experiment we test 2 modes of tuple-receiving concurrency: (i) the consumer accepts only one tuple-send request at any given time (sequential tuple receiving); (ii) the consumer accepts at most 64 tuple-send requests concurrently (parallel tuple receiving).

Figure 4.10 (right) depicts the materialization time as the size of the published data increases. We observe that the materialization time increases proportionally to the size of published data in both sequential and parallel tuple receiving modes. Also, parallel tuple receiving reduces the view materialization time by more than 50% (600 sec. instead of about 1400 sec. to absorb 3.2GBs of data).

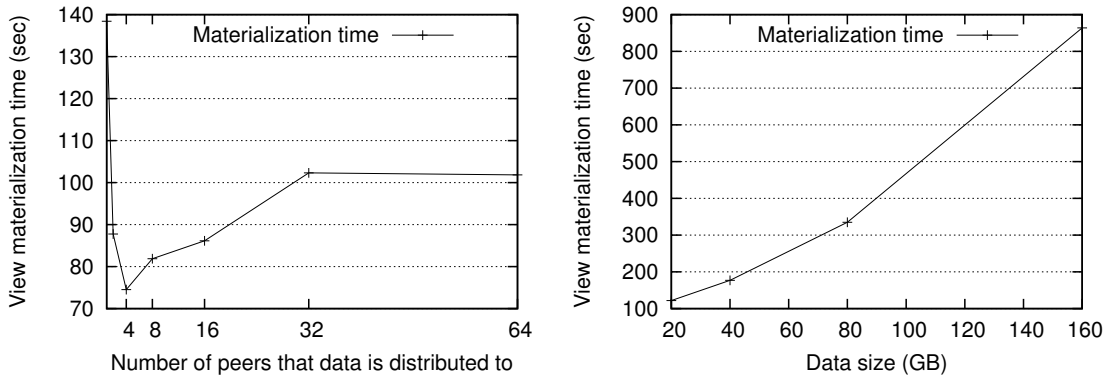


Figure 4.11: Experiment 6: publishing the same amount of data from an increasing number of publishers (left); experiment 7: publishing varying size of data in 50 groups of 5 peers each (right).

From the two graphs in Figure 4.10, we conclude that it is faster for the network to absorb data using one consumer and many publishers rather than many consumers and one publisher. For example, for absorbing 1024MBs of data, the view materialization time is less than 200 seconds (Figure 4.10 right) for 64 publishers and one consumer, and about 3000 seconds in the case of one publisher and 64 consumers (Figure 4.10 left). This is explained by the fact that data extraction is proven to be a slow process (Experiment 2) thus it is slow for a peer to extract all the available data by itself and ship them to the consumers.

Experiment 6: Varying Number of Publishers, Fixed Data, One Consumer The purpose of this experiment is to study the impact that the parallelization of document publication has on the view materialization time. We use 250MBs of data distributed evenly across an increasing number of publishers. First, one peer publishes all the data, then two peers publish half of the data each, then 4, then 8 peers etc. The parameter values for this experiment are as follows:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	{1,2,...,64}	1	1	512	512	0.49MB

Figure 4.11 (left) shows how materialization time varies depending on the number of parallel publishers. The time decreases as the data is distributed to two and then 4 publishers, as the extraction effort is parallelized. From 8 publishers onwards, the materialization time increases again, until it stabilizes from 32 to 64 publishers. This increase is due to publishers simultaneously trying to connect to the consumer and making the consumer's storage module the bottleneck.

Exp. No.	Experiment description	Throughput (MB/sec)
3	One publisher, fixed data, varying number of consumers	10.30
4	One publisher, varying data size, 64 consumers	0.34
5	64 publishers, varying data size, one consumer	5.31
6	Varying number of publishers, fixed data, one consumer	8.05
7	Community publishing	238.80

Table 4.2: Maximum data absorption throughput during view materialization.

Experiment 7: Community Publishing We now consider a more complex scenario. We study materialization time in a setting with (logical) sub-networks, i.e., such that no single publisher has data of interest to all views, and no single view needs data from all publishers. The parameters of this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
250	250	50	50	{20K,...,160K}	{20K,...,160K}	1MB

We use a network of 250 peers, each of which holds the same number of 1MB documents. We logically divide the network into 50 groups of 5 peers each, such that in each group there are five publishers and one consumer (one peer is both a publisher and a consumer). The data of all publishers in a group is of interest for the consumer of that group, but it is not relevant for any of the other groups' views. The group peers are randomly chosen, i.e., they do not enjoy any special geographic or network locality etc. The total amount of data published (and shipped to the views) varies from 20GBs to 160GBs. Figure 4.11 (right) shows that the materialization time grows linearly with the published data size.

Conclusion This section has studied several extreme cases of view materialization (very skewed / very evenly distributed, with one or many publishers or consumers etc.), in order to traverse the space of possibilities. Overall, the experiments demonstrate the good scalability properties of ViP2P as the data volume increases, and that ViP2P exploits many parallelization opportunities when extracting, sending, receiving and storing view tuples. Table 4.2 summarizes the results by providing a global metric, the view materialization throughput, reflecting the quantity of data that can be published (from documents to views) simultaneously in the network. Table 4.2 demonstrates that ViP2P properly exploits all opportunities for parallelism in the "community publishing" scenario: the throughput is of 238 MB/s, while the best comparable result in this area from KadoP is of 0.33 MB/s only [AMP+08].

4.5.4 View Indexing and Retrieval Evaluation

We now compare the view indexing and lookup strategies LI, RLI, LPI and RPI described in Section 4.4.2.

Experiment 8: View Indexing and Retrieval We start with a random synthetic query q of height 5, having 30 nodes labeled $a_1, \ldots, a_{30}$. Each node of q has between 0 and 2 children. We then create three variants of q:

— q' has the same labels as q, but totally disagrees with q on the structure (if a_i is an ancestor of a_j in q, a_i is not an ancestor of a_j in q')

— q'' coincides with q for half of the query, while the other half conserves the labels of q but totally disagrees on the structure (as in q')

— 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 labels $b_1, \ldots, b_{15}$ (that replace $a_{16}, \ldots, a_{30}$ respectively).

From each of q, q', q'' and q''' we automatically generate 360 views of 2 to 5 nodes, totaling 1440 views, such that: the views can all be embedded into their 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 resembling the original query q to various degrees.

We have indexed the resulting 1440 views in a network of 250 peers, following the LI, RLI, LPI and RPI strategies described in Section 4.4.2. We then performed lookups using the four different indexing strategies. The parameters characterizing this experiment are the following:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
250	0	1440	250	0	0	0

Figure 4.12 (left) depicts the number of views retrieved by each strategy, compared to the number of useful views, which can be embedded into q. We observe, as expected, that the path indexing-lookup strategies (LPI and RPI) are more precise than the label based ones (LI and RLI). Moreover, LPI is the most precise, since it uses as keys longer paths, describing views more precisely.

Figure 4.12 (right) depicts the time spent looking up in the DHT the set of (possibly) useful views in order to rewrite q, as well as the time spent to check whether embeddings exist from those views into q. We observe that from this angle, the label strategies (LI and RLI) perform better than the path strategies, since the more numerous lookups performed by the path strategies take up too much time when processing queries.

Figure 4.13 (left) depicts the number of view definitions that were indexed in the DHT by each view indexing strategy. Figure 4.13 (right) depicts the number of lookups performed by each strategy for the query we consider. As expected, LI inserts the largest number of DHT entries. With respect to query-driven lookup, LI and RLI perform 30 lookups, much less than LPI and RPI that perform 370 lookups each.

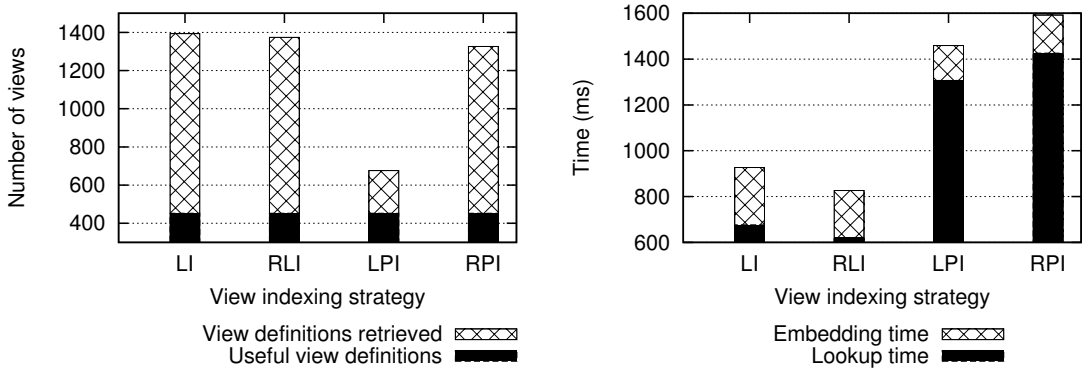


Figure 4.12: Experiment 8: view definition retrieval (left); embedding vs lookup time (right).

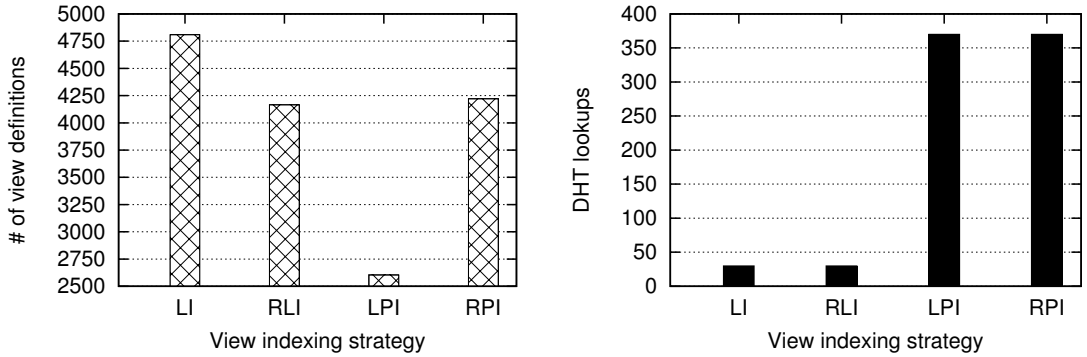


Figure 4.13: Experiment 8: lookups generated for retrieving views (left); embedding vs lookup time (right).

From this experiment, we conclude that label-based strategies are preferable, since the savings at query processing time are more critical than the DHT index size (which is very modest in all cases) or the precision of look-up, as the retrieved view definitions are further filtered at the query peer (after the embedding filtering, the rewriting is run with the same set of views no matter the used strategy).

4.5.5 Query Engine Evaluation

Experiment 9: Query Response Time vs. Query Selectivity and Number of Results We now investigate the query processing performance as the data size increases. We use 20 peers, all of which are publishers, 2 are consumers and 1 is a query peer. The query peer and the 2 consumers are located in 3 different locations of France (Bordeaux, Lille and Orsay). The parameter values characterizing this experiment are the following:

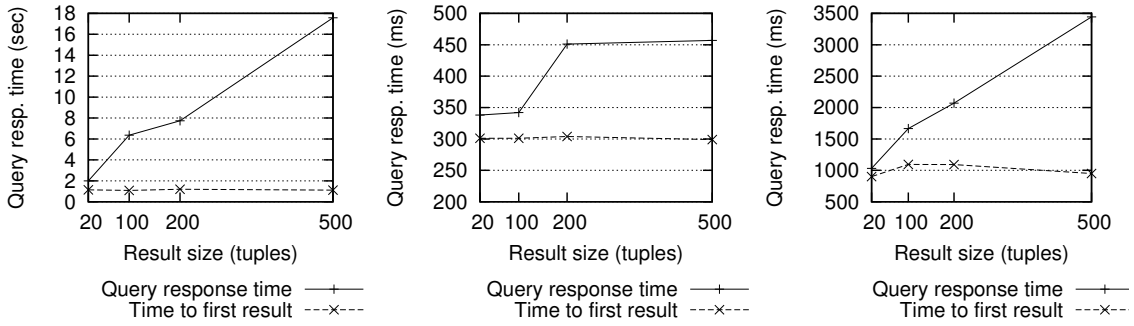


Figure 4.14: Experiment 9: query execution time vs. number of result tuples for three queries.

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
20	20	2	2	{20,...,500}	{20,...,500}	0.5MB

The document used in this experiment is the same as the one of Figure 4.9 (left) with a slight difference: the root element $catalog$ has only one child, named $camera$.

The views defined in the network are the following:

— v_1 is $//catalog_{ID}//camera_{ID}//description_{ID,cont}$

— v_2 is $//catalog_{ID}//camera_{ID}//\{description_{ID}, price_{ID,val}, specs_{ID,cont}\}$

Each view stores one tuple from each document. A v_1 tuple from document d roughly contains all of d (since the $description$ element is the most voluminous in each $camera$). A v_2 tuple is quite smaller since it does not store the full camera descriptions. We use three queries:

— q_1 asks for the $description_{cont}$, $specs_{cont}$ and $price_{val}$ of each $camera$. To evaluate q_1, ViP2P joins the views v_1 and v_2. Observe that q_1 returns full XML elements, and in particular, product descriptions, which are voluminous in our data set. Therefore, q_1 returns roughly all the published data (from 10MB in 20 tuples, to 250MB in 500 tuples).

— q_2 requires the $description_{ID}$, $specs_{ID}$ and $price_{ID}$ of each $camera$. This is very similar to q_1 but it can be answered based on v_2 only. The returned data is much smaller since there are only IDs and no XML elements: from 2KB in 20 tuples, to 40KB in 500 tuples.

— q_3 returns the $specs//sensor\_type_{val}$ of each $camera$. The rewriting of q_3 applies *navigation* over $specs_{cont}$ that is stored by v_2. The result size varies from 2KB in 20 tuples to 40KB in 500 tuples.

Figure 4.14 shows the query response time and the time to get the first result for the 3 queries. The low selectivity query q_1 (at left in Figure 4.14) takes longer than q_2, due to the larger data transfers and the necessary view join. The time to first result is always constant for both q_1 and q_2 and does not depend on the result size. For q_1, a hash join is used to combine v_1 and v_2, and thus no tuple is

output before the view v_2 has been built into the buckets of the hash join. This is done in more or less one second in the case of q_1 and about 300ms for q_2. Note that the join is performed on the peer holding v_1 as it is faster to transfer v_2 at the peer holding v_1. Increases in the total running time appear when more data-sending messages are needed to transfer increasing amounts of results. For q_3, which applies navigation on the view v_2, the time to the first tuple is the time to evaluate the navigation query locally at v_2's peer and send the first message with result tuples to the query peer, and this does not grow with the data size.

Conclusion The ViP2P query processing engine scales close to linearly when answering queries in a wide-area network. The fact that ViP2P rewrites queries into logical plans which are then passed to an optimizer enables it to take advantage of known optimization techniques used in XML and/or distributed databases, to reduce the total query evaluation time, and (depending on the characteristics of the particular physical operators chosen) the time to the first answer. Given the ViP2P architecture, the peers involved in processing a query are only those holding the views used in the query rewriting; this is why using only 20 peers for this experiment does not affect its interpretation, since ViP2P query processing involves only three peers. The network size may only impact the view look-up time, which is very modest (Section 4.5.4).

4.5.6 Conclusion of the Experiments

Our study leads to the conclusion that the ViP2P architecture scales up well. In particular, view materialization scales in the number of publishers and consumers, in the size of the network, and in the size of the data. High contention at a single consumer receiving data from many publishers, and especially at a single publisher contributing to many consumers' views, degrades the ability of the view holders to efficiently absorb data. However, these contention effects are to be expected in a large distributed system. Moreover, we showed that when interest in the published data is more evenly distributed among sub-communities, ViP2P takes advantage of all parallelization opportunities to increase the data transfer rate between publishers and consumers by 3 orders of magnitude. Our view materialization experiments also show the importance of carefully tuning all stages in the data extraction and data transfer process, including asynchronous communication and parallelization whenever possible. The cumulated impact of these optimizations on the data transfer rate between peers are dramatic (more than 4 orders of magnitude increase).

Our query processing experiments show that label-based view indexing strategies are preferable, and indeed we use RLI by default. They also demonstrate that the ViP2P execution engine scales linearly up to large data volumes, orders of magnitude more than in previous real DHT deployments [AMP+08, RM09b].

4.6 Summary

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-indexed views adds to query processing the (modest) cost of locating relevant views and rewriting the query using the views, in exchange for the benefits of using pre-computed results stored in views. We studied several view indexing strategies and associated complete view lookup methods. Moreover, we did an extensive study of our platform's main aspects (view materialization, indexing and retrieval, and query processing) in different scenarios and settings. ViP2P was able to extract and disseminate 160GB of data in less than 15 minutes over 250 computers in a WAN network [Gri]. These results largely improve over the closest competing XML management platforms based on DHTs, and actually implemented and deployed (1 GB of data indexed in 50 minutes in KadoP [AMP+08], hundreds of MB of data on 11 peers in psiX [RM09b], which, unlike us, focused only on document indexing and look-up).

Acknowledgements Part of the ViP2P code comes from ULoad [ABMP07]. We thank Alin Tilea, Jesús Camacho-Rodríguez, Alexandra Roatis, Varunesh Mishra and Julien Leblay for their help developing and testing ViP2P. Experiments presented in this chapter were carried out using 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 funding bodies (see https://www.grid5000.fr). This work was partially funded by the CODEX (ANR 08-DEFIS-004) and DataRing (ANR 08-VERS-007) projects.

Chapter 5

Delta: Scalable View-based Publish/Subscribe

In content-based publish/subscribe (pub/sub, in short) systems, users express their interests as queries over a stream of publications. Scaling up content-based pub/sub to very large numbers of subscriptions is challenging: users are interested in low *latency*, that is, getting subscription results fast, while the pub/sub system provider is mostly interested in *scaling*, i.e., being able to serve large numbers of subscribers, with low *computational resources utilization*.

We present a novel approach for scalable content-based pub/sub in the presence of constraints on the available CPU and network resources, implemented within our pub/sub system Delta. We achieve scalability by off-loading some subscriptions from the pub/sub server, and leveraging view-based query rewriting to feed these subscriptions from the data accumulated in others [1]. Our main contribution is a novel algorithm for organizing views in a multi-level dissemination network, exploiting view-based rewriting and powerful integer linear programming capabilities to scale to many views, respect capacity constraints, and minimize latency. The efficiency and effectiveness of our algorithm are confirmed through extensive experiments and a large deployment in a WAN.

The results of this chapter are part of an article submitted for publication on May 1st, 2013 and which is currently being reviewed. Unlike in previous chapters, in this chapter we will refer to the *cost* of evaluating a distributed rewriting plan as *computational resources utilization* or, simply, *utilization*.

5.1 Motivation and Outline

Publish/subscribe (*pub/sub*, in short) is a popular model for disseminating content to large numbers of distributed subscribers. The literature distinguishes *topic-based* pub/sub, where users subscribe to a set of predefined topics, from

1. This can be seen as organizing subscriptions in a dissemination network where data flows from the source through a network of subscriptions, similarly to water flow in a river delta.

content-based pub/sub, where users express their subscriptions as custom complex-structured queries on the published data. Topic-based pub/sub offer better scalability at the expense of subscription expressiveness, while in more complex systems, the increased expressive power of content-based pub/sub makes it preferable. For instance, within a large company ACME, *"senior positions representing ACME in Singapore"* should be pushed to the senior staff which may be interested, while *"sales seminar in Singapore"* interests the sales department plus the administrative staff that must make the travel arrangements.

Pub/sub *subscribers* are interested in *low latency*, that is, getting all the results to their subscriptions, as soon as possible after the data is published. The *publisher* of a pub/sub system faces several performance challenges in order to meet subscriber requirements. The first is matching published items against the set of subscriptions, a CPU-intensive task. Then, the publisher's outgoing bandwidth is another physical limitation, as more and more updates must be sent to the interested subscribers. Third, the speed of the network connecting the publisher to the subscribers imposes a lower bound on the dissemination latency.

Both centralized and distributed approaches have been proposed to address the above issues, while aiming at latency minimization. The centralized ones [CDTW00, DAF+03] mostly rely on efficient filtering algorithms for matching the data against subscriptions. However, for more expressive and numerous subscriptions, subscription matching remains an onerous task. To this end, distributed pub/sub systems have been proposed [DRF04, GSAA04, Pap05, TBF+03], providing solutions for serving thousands or millions of subscribers with minimum resources utilization and low latency. In most cases, they focus on distributed filtering and design overlay networks in the form of logical multicast trees. Those trees are formed by specialized nodes, called *brokers*, able to efficiently filter and move the data from the publisher to the subscribers, or by the subscribers themselves. Nevertheless, as the amount of subscribers and data increases, the publisher's (or broker's) resource capacity becomes insufficient.

Problem Statement To overcome the above resource constraints, we allow the subscribers to *take part in the dissemination of data* (i.e. serve other subscribers that have similar interests) in order to offload the data publisher. Due to their similarity of interests, the subscribers can form a logical overlay network, over which subscription results can flow from the data publisher to the subscribers. Since subscribers have to use their resources to serve others, the problem we consider is how to (i) minimize the total resource utilization (e.g., CPU and bandwidth), while (ii) keeping the subscription latency as low as possible, and (iii) respecting the given resource capacity constraints.

The key idea on which we build our approach is that subscriptions often overlap, completely or partially, when user interests are close. In such a case, results of several subscriptions can be combined to compute the results of other subscriptions. For instance, from the subscriptions s_1: *"open positions in Asia"* and s_2: *"open positions in Sales"*, one can compute s_3: *"open Sales positions in Asia"* by joining s_1 and s_2.

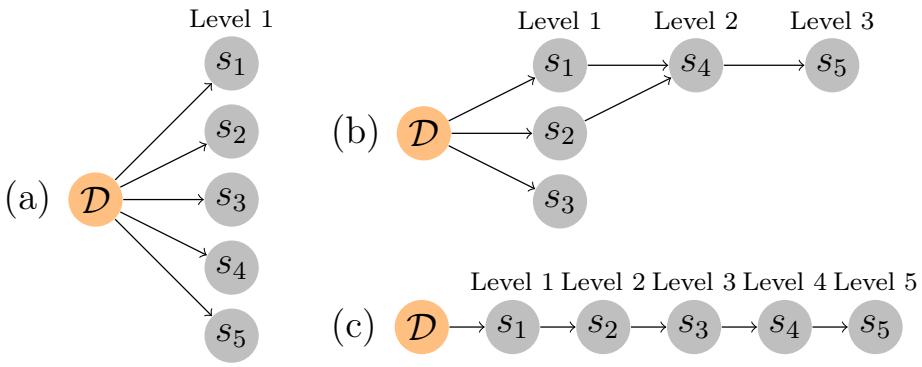


Figure 5.1: Sample dissemination networks.

Rewriting Subscriptions More formally, a subscription can be *rewritten based on other subscriptions*, by filtering their results, e.g., through classic database selections and projections, combining them through joins, etc. For instance, rewriting and serving s_3 based on s_1 and s_2 instead of the publisher, relieves the publisher from the effort of computing s_3 against the published data, and saves bandwidth between the publisher and the site of s_3. At the same time, rewriting s_3 from s_1 and s_2 incurs computations to the sites of s_1, s_2 and/or s_3 to evaluate the rewriting, and also bandwidth consumption from the sites of s_1 and s_2, to the site of s_3. Notice that if we consider subscriptions as queries (or views), deciding how to serve a subscription based on others, can be turned to a problem of *view-based query rewriting*, which has been extensively studied in the database literature (e.g. [PH01, MKVZ11]).

Multi-level Subscriptions Moving a subscription from being served directly by the publisher (we call this a *level 1* subscription), to being served from other subscriptions by rewriting (we call this a *level 2*, *level 3* subscription, etc.), changes the data transfer and processing paths, with many possible consequences on subscription latency and resources utilization for data dissemination.

For illustration, Figure 5.1 shows three possible dissemination networks. At left (a), there is only one level and all subscriptions are filled from the publisher $\mathcal{D}$. The data paths from $\mathcal{D}$ to all subscriptions are as short as possible, however all the load is on $\mathcal{D}$. At (b), the subscription s_5 gets its data from s_4 instead of the publisher, while s_4 results are computed based on s_1 and s_2. At (c), only s_1 is filled from $\mathcal{D}$, while s_2 gets data from s_1, s_3 from s_2, etc. The load on the publisher is minimal, but the four hops from $\mathcal{D}$ to s_5, increase the latency of this subscription

More generally, dissemination effort decreases at the publisher, at the expense of subscribers joining this effort. A less-loaded publisher will likely match data against the rest of the subscriptions faster, which may reduce the total latency for all the subscriptions. However, moving a subscription to a higher level lengthens the data path from the publisher to that subscription, which may increase its latency. Finally, pushing some processing at the subscribers require taking into

account a new set of capacity constraints, since subscriber resources should be sparingly used, to keep the respective sites willing to participate in the system.

Contributions and Outline Given a set S of subscriptions and a data publisher $\mathcal{D}$, we term *configuration* a choice for each subscription $s \in S$ of filling s either (i) directly from $\mathcal{D}$ or (ii) by rewriting s over some other S subscriptions and thus computing s results from these other subscriptions' results. The *cost* of a configuration is a weighted sum of the resource utilization and subscription latencies incurred by the configuration. This work makes the following contributions:

— We show how to model the problem of finding a minimum-cost configuration under some resource capacity constraints as a graph problem, related to the known Degree-bounded Arborescence problem [BKN09], but departing from it through our interest in minimizing both resource utilization and latency. As we will explain, resource utilization and latency differ in fundamental ways, making existing solutions inapplicable in our setting.

— Based on this insight, we provide a novel two-step algorithm for selecting a configuration. First, we employ an Integer Linear Programming (ILP) approach to find a resource utilization-optimal solution (ignoring latency); second, we provide a latency optimization algorithm which starts from the configuration found by the ILP solver and modifies it to reduce latency.

— We have implemented all our algorithms and performed extensive experiments, including a deployment of Delta on a significant-size pub/sub scenario on a WAN. Our experiments demonstrate the efficiency and effectiveness of our algorithms and the practical interest of multi-level subscriptions in large data dissemination networks.

The chapter is organized as follows. Section 5.2 introduces our problem and presents its graph-based formalization. Section 5.3 describes our algorithms for selecting an efficient configuration, based on the graph models previously introduced. Section 5.4 details our view-based approach for rewriting subscriptions based on other subscriptions, given the large number of subscribers. Section 3.6 describes our experiments, we then discuss related works and conclude.

5.2 Problem Model

We now describe our multi-level subscription problem model.

Let $\mathcal{D}$ denote a data source publishing a set of data items $i_1, i_2, \ldots$ and $S = \{s_1, s_2, \ldots, s_n\}$ be a finite set of subscriptions, each defined by a query and established on some network site. The semantics of a subscription s defined by query q_s and issued at site n_s is that s must receive the results of $q_s(i)$ for any data item i published by the data source $\mathcal{D}$ after s was created. From now on, for simplicity, whenever possible we will simply use s to denote both a subscription and the query defining it.

At the core of our work is the observation that it may be possible to compute results of a subscription out of the results of others. We say subscrip-

tion s can be *rewritten* based on subscriptions $s_1, s_2, \ldots, s_k$, if there exists a query r, which, evaluated over the results of $s_1, s_2, \ldots, s_k$, produces exactly the results of subscription s, regardless of the actual data items published by $\mathcal{D}$: $r(s_1(\mathcal{D}), s_2(\mathcal{D}), \ldots, s_k(\mathcal{D})) = s(\mathcal{D})$ for any $\mathcal{D}$, or more simply, $r(s_1, s_2, \ldots, s_k) \equiv s$, where $\equiv$ denotes query equivalence.

Subscriptions = Views Observe that we are interested in complete rewritings, that is, we assume that r can either rely completely on the data source, or on the results of other subscriptions $s_1, s_2, \ldots, s_k$. This is because our goal is to off-load subscriptions from the data source and serve them solely from other subscriptions instead. In turn, a subscription s rewritten based on $s_1, \ldots, s_k$ as above, may be used to rewrite another subscription s'. This shows that every subscription may be considered as a (materialized) view, based on which to rewrite the others. Thus, from now on, for conciseness, we will simply use *view* to designate a subscription.

In the sequel, we introduce the central concepts and data structures of our work. We define rewritability graphs (RGs) and configurations in Section 5.2.1. Section 5.2.2 presents the basic metrics we use to gauge the interest of a configuration, namely utilization and latency, and shows how to incorporate load balancing in the discussion under the form of constraints over the configurations. Based on these notions, Section 5.2.3 formalizes our problem statement.

5.2.1 Rewritability Graph (RG)

A rewritability graph (RG) indicates which views can be rewritten based on other views. Its simplest representation is an AND-OR rewritability graph as in, e.g., [Gup97]. For each view v at site s, there is a corresponding node in the AND-OR graph (if the same v is declared at n distinct sites $s_1, s_2, \ldots s_n$, there are n corresponding nodes in the graph). Moreover, for every view set $v_1, v_2, \ldots, v_k$, based on which v can be equivalently rewritten, there exists a $\wedge$ (AND) node a_v such that: (i) each of the nodes corresponding to $v_1, v_2, \ldots, v_k$ points to a_v, and (ii) a_v points to the v node. If v can be rewritten based on several view sets, there will be one $\wedge$ node pointing to v for each such rewriting possibility[2].

A sample RG over seven views is depicted in Figure 5.2. Each view can always be evaluated directly from the data source $\mathcal{D}$, thus, for each view v, there is a $\wedge$ node through which $\mathcal{D}$ is connected to v. Further, in Figure 5.2, v_2 and v_3 can be used to rewrite v_5, as shown by the lower $\wedge$ node pointing to v_5; v_3 and v_4 can be used to rewrite v_6, etc. Observe that there may be cycles in the RG: v_6 can be used to rewrite v_7 and vice versa. This entails that v_6 and v_7 are equivalent.

Formally, given a view set S, an RG is a directed graph, defined by the pair $(V \cup \{\mathcal{D}\} \cup A, E)$, such that:

— $V \cup \{\mathcal{D}\} \cup A$ is the set of nodes:
  — For each view $s_i \in S$, there exists a corresponding node $v_i \in V$.

2. To keep the AND-OR graph shape, one would have needed to use a $\vee$ node pointing to v and have the $\wedge$ nodes pointing to that $\vee$ node instead of v directly. We omit the $\vee$ nodes for simplicity.

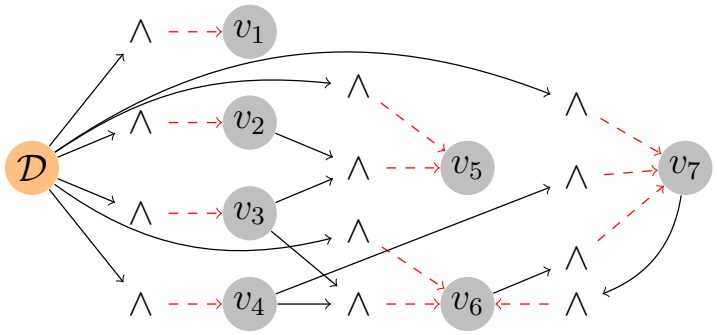


Figure 5.2: Rewritability Graph (RG).

- $\mathcal{D}$ is the node corresponding to the data source.
- A is the set of $\wedge$ nodes, each of which represents a rewriting of a view $s \in S$ based on a set of other views $\{s_1, s_2, \ldots, s_k\} \subseteq S \setminus \{s\}$.
- $E \subseteq ((V \cup \{\mathcal{D}\}) \times A) \cup (A \times V)$ is the set of directed edges that connect the graph's nodes as follows:
  - V nodes (as well as $\mathcal{D}$) can only point to A nodes, while A nodes can only point to V nodes.
  - Each node $a \in A$ has an indegree of at least one, and an outdegree equal to one.
  - For each view $v \in V$, there exists a $\wedge$ node $a_v \in A$ such that (i) $\mathcal{D} \rightarrow a_v \rightarrow v$ and (ii) $\mathcal{D}$ is the only node pointing to a_v.
  - For each view set $\{s_1, s_2, \ldots, s_k\}$ based on which another view s can be rewritten, there exists a $\wedge$ node $a_v \in A$ such that the edges (v_1, a_v), (v_2, a_v), $\ldots$, (v_k, a_v), $(a_v, v) \in E$.

Size of RG The number of nodes in an RG is $|V| + |A| + 1$ (where 1 corresponds to $\mathcal{D}$). We have $|V| = |S|$, which is the number of views (subscriptions). As for the A nodes, there is one for every V node v, connecting $\mathcal{D}$ to v (thus, $|S|$ such A nodes). Moreover, we have one A node for every view set that can rewrite a view v. Since there are $|S| - 1$ views that can be used to rewrite v (we exclude v itself), we can have at most $2^{|S|-1}$ such A nodes for v. Thus, we have $|A| \leq |S| \times (2^{|S|-1} + 1)$.

We now turn to the number of edges. Since by definition the outdegree of each A node is one, there are $|A|$ edges from A to V nodes. Furthermore, an A node has at most $|S| - 1$ incoming edges (a rewriting can involve at most that many views), leading to at most $|A| \times (|S| - 1)$ edges from V to A nodes. Hence, we have $|E| \leq |S|^2 \times (2^{|S|-1} + 1) \approx |S|^2 \times 2^{|S|}$.

Clearly, an RG may be very large when there are many views. Therefore, it is also of interest to develop *partial rewritability graphs*, each of which can be seen as the RG from which some $\wedge$ nodes (and their corresponding input and output edges) have been erased.

Configuration (CFG) Given an RG, a configuration (CFG) is a subgraph of RG encapsulating a concrete choice of how to rewrite every view $v \in V$. Specifically,

in a configuration, only a single $\wedge$ node points to each view. Moreover, there exists a directed path from $\mathcal{D}$ to each view of the RG [3].

Formally, given an RG $rg = (V \cup \{\mathcal{D}\} \cup A, E)$, a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ is a subgraph of rg such that:

— $A' \subseteq A$ and $E' \subseteq E$;
— for any $v \in V$, there exists exactly one $a \in A'$ such that $a \rightarrow v$;
— there exists a path from $\mathcal{D}$ to any view $v \in V$;
— for each node $a \in A'$, if edge $(v_i, a) \in E$ (for each $v_i \in V$), then $(v_i, a) \in E'$.

The last point in the above definition guarantees that when we select an A node to be included in cfg, we also select all its incoming edges that constitute the rewriting. Observe that a CFG completely specifies the paths along which data is disseminated to all the subscribers. Moreover, multiple data dissemination paths starting from the source $\mathcal{D}$ may meet, for instance, when two views v_1 and v_2, together, rewrite another view v_3.

The number of CFGs which may be derived from an RG is $\Pi_{v \in V}(in(v))$ where in denotes the indegree of a view node. It follows from the RG size estimations that the upper bound for the number of CFGs is $|S|^{2^{|S|}}$, which is extremely high.

5.2.2 Characteristics of a Configuration

We now discuss how to quantify the cost of a CFG.

For each rewriting ($\wedge$) node in a CFG, there can be several ways of distributing the effort entailed by the rewriting (typically selections and joins) across the network nodes in which the views reside. For example, consider the views v_2, v_3 and v_5 of Figure 5.2. Assume that v_2 resides on site n_2, v_3 on n_3 and v_5 on n_5. To join v_2 and v_3, they could both be shipped to the site n_5 and joined there. Alternatively, v_3 could be shipped to n_2, the join could be evaluated at n_2 and the results shipped to n_5, at a different resources utilization. More generally, the utilization incurred by the operations of a $\wedge$ node depend on the operations' types and ordering, where each operation runs etc.

Distributed Resources Utilization To estimate the resources *utilization* of a given $\wedge$ node, we quantify the resources (e.g., I/O, CPU, bandwidth) needed for its execution over the various sites.

Let N be the set of network sites on which work can be distributed (we assume for simplicity N is the set of all the sites having subscriptions), and k be the number of distinct resources considered for each site, such as: I/O at that site, CPU, incoming and outgoing bandwidth, etc. Let $P_\wedge$ be the set of all physical plans for a given $\wedge$ node. We define the utilization function $u : P_\wedge \rightarrow \mathbb{R}^{|N| \times k}$, assigning to each plan $p \in P_\wedge$, the estimated resources utilization, along different resource dimensions, entailed by the evaluation of p. Observe that each result of u is a matrix stating the consumption along each dimension and at each site.

To enable comparing utilizations, we rely on a single *utilization aggregator*

3. This also guarantees that a configuration is acyclic.

$\mathcal{U} : \mathbb{R}^{|N|\times k} \rightarrow \mathbb{R}$, which combines the utilization of all the different resource components of the sites involved in the execution of a plan, and returns a single (real) number. The aggregator may for instance sums up all the utilization components, possibly assigning them various weights depending on the metric and/or the site involved. In the sequel, for a plan $p \in P_\wedge$, we will simply write $\mathcal{U}(p)$ to denote the scalar aggregation $\mathcal{U}(u(p))$ of p's multidimensional utilization.

Finally, for a given $\wedge$ node $a \in A$, we denote by $\mathcal{U}(a)$ the smallest value of $\mathcal{U}(p)$, over all the plans $p \in P_\wedge$. Moreover, the utilization of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ is:

$$\mathcal{U}(cfg) = \sum_{a \in A'} \mathcal{U}(a).$$

Latency In a CFG, given a data item i and subscription v such that $v(i) \neq \emptyset$, the data dissemination *latency* of v with respect to i, denoted $\lambda(v, i)$, is the time interval between the publication of i and the moment when $v(i)$ reaches the site of v. In the sequel, we may simply use $\lambda(v)$ to denote v's latency.

Clearly, $\lambda(v)$ is determined by the paths in CFG followed by the data that is moving from $\mathcal{D}$ to v. Each $\wedge$ node a encountered along these paths adds to the latency its contribution, which we term *local latency* of a. That reflects the delays introduced on the propagation of data in the rewriting graph, by evaluating that rewriting. For instance, if the best physical plan for a $\wedge$ node requires shipping data across the network from n_1 to n_2 and performing a join at n_2, the local latency of this node will reflect the data transfer and the processing time in the join. We assume available a *local latency estimation function* l, which estimates the local latency introduced by a. We stress that $l(a)$ characterizes only the operations at the rewriting node a, and not the behaviour of its input(s).

Given that for every subscription v there is a single $\wedge$ node a_v pointing to v (see RG definition, Section 5.2.1), v's latency is equal to the *total latency* of a_v (denoted $\lambda(a_v)$), thus $\lambda(v) = \lambda(a_v)$. This latency can be computed by adding a_v's local latency $l(a_v)$ to the maximum latency of the subscriptions $\{v_i\}$ that are inputs to a_v. Denoting by $v_i \rightarrow a_v$ the fact that node v_i points to a_v in the RG, we have:

$$\lambda(a_v) = \lambda(v) = max_{v_i \rightarrow a_v}(\{\lambda(v_i)\}) + l(a_v)$$

Note that the latency of $\mathcal{D}$ is defined as 0. We also define the latency of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ as follows:

$$\lambda(cfg) = \sum_{v \in V} \lambda(v).$$

Cost We define the cost of a $\wedge$ node a in a CFG as a linear combination of its utilization and latency:

$$\mathcal{C}(a) = \alpha \times \mathcal{U}(a) + \beta \times \lambda(a)$$

where α and β are coefficients controling the importance given to the utilization and latency. A high α prioritizes solutions of low utilization, incurring a low consumption of resources across the network, while a high β prefers solutions having a low latency, favoring quick dissemination of data to the subscribers. Finally, we define the cost of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$:

$$\mathcal{C}(cfg) = \sum_{a \in A'} \mathcal{C}(a).$$

Constraints In practice, resources such as CPU, memory, incoming and outgoing network bandwidth, are limited on each site. This has to be taken into account when deciding whether to use a view v_1 to feed another view v_2 with data, since doing so incurs some consumption of resources on the site of v_1: such resource consumption should be kept within the capacity limits. Each site may have different such *capacity constraints*, according, for instance, to its specific infrastructure or available bandwidth.

We make the simplifying assumption that there is a single view published in each network site. We model capacity constraints by a single integer B_v^{out}, which is the maximum number of views that can be served by v (and which coincides with the maximum number of views served by a network site, since there is one view per site), and design our algorithms to operate within these constraints. This can be easily extended to more (and more complex) constraints.

5.2.3 Problem Statement

Given an RG $rg = (V \cup \{\mathcal{D}\} \cup A, E)$, a cost function $\mathcal{C}$, a limit B_v^{out} for each $v \in V$, as well as a limit $B_{\mathcal{D}}^{out}$ for the data source, the problem we address is to find a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$, such that:

1. Capacity constraints are respected:

$$\forall v \in V \cup \{\mathcal{D}\},\ out(v) \leq B_v^{out}$$

   where $out(v)$ denotes the outdegree of node v in the CFG;

2. The cost of CFG $\mathcal{C}(cfg)$ is minimized.

5.3 Configuration Selection

We now describe our approach for selecting a low-cost configuration. We start by discussing RG construction in Section 5.3.1. Section 5.3.2 provides an overview of the CFG selection, a two-step process described in detail in Section 5.3.3 and 5.3.4, respectively. Section 5.3.5 shows how we treat with CFG updates (view addition/removal).

Algorithm 4: Partial RG Generation

Input : View set V, maximum number k of rewritings per view
Output: RG of V with at most k rewritings per view

```
// RG initially contains only V and D
A ← ∅, E ← ∅, G ← (V ∪ {D} ∪ A, E)
foreach v ∈ V do
    rewrNo ← 0
    while hasNextRewriting(v, V \ {v}) and (rewrNo < k) do
        // Get next rewriting
        rw ← nextRewriting(v, V \ {v})
        A ← A ∪ {rw}                          // Add rewriting (∧) node rw
        E ← E ∪ {(u_i, rw)}, ∀u_i ∈ rw        // Add edges to rw
        E ← E ∪ {(rw, v)}                     // Add edges to v
        rewrNo++
// All views are also fed by D
E ← E ∪ {(D, u)}, ∀u ∈ V
return G
```

5.3.1 Rewritability Graph Generation

Given a set of views, we show how to construct the corresponding RG, modelling the ways to rewrite views based on other views.

Naive RG Generation Assume we initially create a graph that contains the nodes $(V \cup \{\mathcal{D}\})$, as well as the $\wedge$ nodes that are needed to connect $\mathcal{D}$ with each view $v \in V$ (along with the corresponding edges). Based on this graph, the most direct way of building the RG is by calling the view-based rewriting algorithm exhaustively, and adding, each time a rewriting is found, the corresponding $\wedge$ nodes and edges. This simple method requires calling the rewriting algorithm $|V|$ times, using each time $|V| - 1$ views. Given the typically high complexity of view-based query rewriting algorithms, this method is unlikely to scale to large problems. Moreover, even if we optimize the calls to the rewriting algorithm (e.g., by reducing the number of views we use as input each time, as discussed in Section 5.4), the resulting *complete* RG is usually too dense, hampering in turn the process of choosing a CFG from RG.

Partial RG Generation In the interest of efficiency, one can limit the search performed during each call to the rewriting algorithm to at most k rewritings. In other words, we only consider the first (at most) k alternative ways we find to rewrite a given query. Clearly, the internals of the rewriting algorithm affect the order in which rewritings are explored and, thus, the first k rewritings found; we will revisit this issue in Section 5.4. Algorithm 4 outlines the construction of the partial RG, obtained through this limited exploration of rewritings. When a view cannot be rewritten based on the others, Algorithm 4 connects it directly to the data source $\mathcal{D}$.

5.3.2 Configuration Selection Overview

We now turn to the problem of selecting out of a (possibly partial) RG, a CFG that minimizes the cost as a weighted sum of *utilization* and *latency*, under *capacity constraints* (as per our problem statement in Section 5.2.3).

Complexity and Relationship with Known Problems We now discuss how our problem relates to already studied graph problems.

First, consider *resources utilization optimization alone*, that is, ignore the latency and capacity constraints. This simplified problem can be solved in linear time, by selecting for each view v in an RG, the lowest resources utilization $\wedge$ node pointing to v, together with the corresponding edge and the $\wedge$ node's incoming edges.

Now assume given bounds on the number of views that can be fed (i) from $\mathcal{D}$ and (ii) from each view, and consider the problem of finding a CFG that respects these *capacity constraints, without considering the cost*. This version of the problem is more complex than the previous one, as choosing $\wedge$ nodes is no longer a local decision for each view v in the RG: selecting an $\wedge$ node can break the capacity constraints of any of the nodes that are serving it.

This last problem of selecting a CFG under capacity constraints is largely connected to the problem of finding a *Degree-bounded Arborescence* (DBA, for short) in a given graph. An arborescence is a spanning tree of a directed graph rooted at a given root node. Although efficient, polynomial-time algorithms have been proposed for solving the *Minimum Cost* Arborescence problem [Edm67], finding a DBA is NP-hard [BKN09]; the NP-hardness is due to the fact that, in order to respect the degree bounds, the edge-selection decisions cannot be local. We have shown that the DBA problem can be reduced in polynomial time to finding a capacity-constrained CFG, which is already a specialization of the general problem we consider (Section 5.2.3), since it does not take into account the cost. This gives the following proposition:

Proposition 5.3.1. *Finding a minimum-cost CFG under capacity constraints is NP-hard.*

Proof. To show that a problem Π is NP-hard, it suffices to show that there exists another NP-hard problem Π' that can be reduced in polynomial time to problem Π. This can be done with the *proof by restriction* [GJ79]: to show that Π is NP-hard, we can simply show that the NP-hard problem Π' is a special case of Π.

We first recal the definition of the Degree-bounded Arborescence (DBA) problem (which is known to be NP-hard [BKN09]), then introduce a specialization of the general problem we consider (we term this specialization *SCFG*). Finally, we show that finding a DBA can be reduced to the problem of finding an SCFG.

The Degree-bounded Arborescence Problem (DBA). Let $G = (V \cup \{\mathcal{D}\}, E)$ be a directed graph with root $\mathcal{D}$, and let B_v^{out} be the bounds on the out-degree of each vertex $v \in V$. The DBA problem consists of finding an (out-)arborescence starting

from $\mathcal{D}$ that satisfies the degree bounds, or declare that no such arborescence exists (if that is the case). Since in an arborescence each vertex except the root has an in-degree of exactly one, the DBA problem does not consider bounds on the in-degree.

We now show that we can specialize the problem of finding a capacity-constrained CFG so that it coincides with the Degree-bounded Arborescence Problem. This can be done by restricting $\wedge$ nodes in an RG to have *only one incoming node* and by *ignoring the resources utilization and latency of the selected CFGs*. We formalize this specialization of our problem below.

The Specialized CFG Problem (SCFG). Let $G = (V \cup A \cup \{\mathcal{D}\}, E)$ be an RG with root $\mathcal{D}$, and let B_v^{out} be the bounds on the out-degree of each vertex $v \in V \cup \{\mathcal{D}\}$. Each $a \in A$ is allowed to have only one input edge. The goal of the SCFG problem is to find a feasible SCFG rooted at $\mathcal{D}$ that respects the bounds B_v^{out}, or declare that it is unfeasible.

First, it is easy to see that the SCFG problem is a specialization of the original problem presented in Section 5.2.3, since: (i) it ignores the cost of the CFGs (resource utilization and latency), and (ii) it restricts all $\wedge$ nodes $a \in A$ to have exactly one input edge. Interestingly, this last restriction allows only for CFGs in which each subscription is fed by another single subscription, as opposed to the more general problem we are tackling (Section 5.2.3), in which subscriptions can be combined (joined) in order to feed other subscriptions.

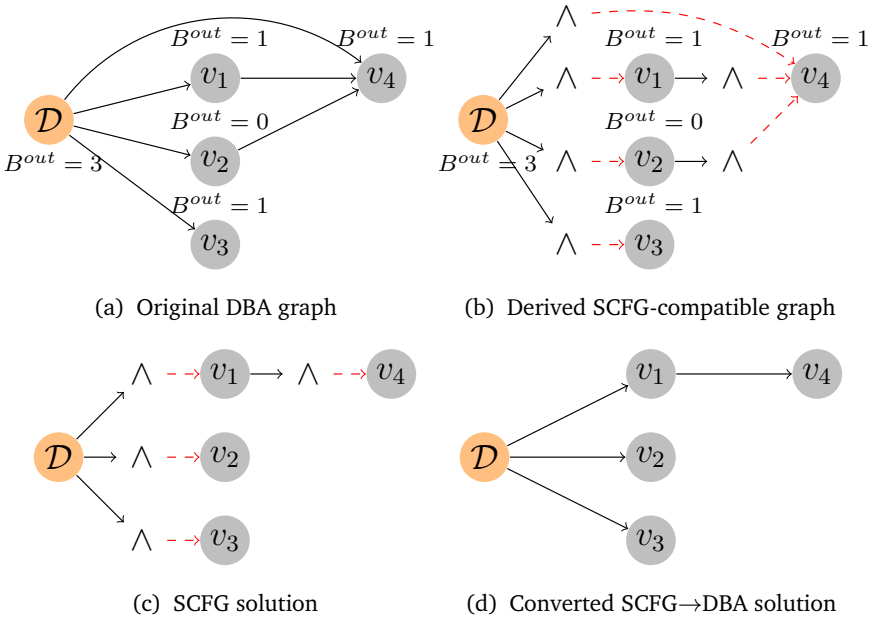
Second, an SCFG is an arborescence, since:

— all views are reachable from the publisher (or root) $\mathcal{D}$;
— there is exactly one $\wedge$ node that points to each view node $v \in V$;
— each $\wedge$ node has exactly one input and one output edge.

Since an SCFG is an arborescence, the next question is whether the DBA problem can be solved with an SCFG solver. If this is possible, and transformations between DBA and SCFG graphs can be done in polynomial time, then the SCFG problem is at least as hard as the DBA problem (thus, NP-hard). To answer this question, we will show how to polynomially transform a graph from the DBA format into an SCFG solver-compatible graph (SCFG-compatible graphs contain $\wedge$ nodes), and polynomially convert the solution that the SCFG solver has produced back to a DBA graph (with no $\wedge$ nodes).

In order to transform a DBA input into an SCFG solver-compatible input, one has to simply replace each edge of the form $n_1 \rightarrow n_2$ of the DBA input graph by one edge $n_1 \rightarrow \wedge$ pointing to a new $\wedge$ node and a second edge $\wedge \rightarrow n_2$. In order to transform an SCFG solution into a DBA solution, one needs to remove the $\wedge$ nodes from the resulting SCFG and connect the input/output edges of all $\wedge$ nodes. Since the transformations are straightforward, we omit their formal description and instead illustrate through an example.

Figure 5.3a shows a graph for which we want to solve the DBA problem. Figure 5.3b shows a derived SCFG solver-compatible graph. Note that, since all $\wedge$ nodes in an SCFG graph have only one input and one output, converting simple edges from a DBA graph into edge-node-edge triplets ($\rightarrow \wedge \rightarrow$) and vice versa is



(a) Original DBA graph

(b) Derived SCFG-compatible graph

(c) SCFG solution

(d) Converted SCFG→DBA solution

Figure 5.3: Solving a DBA instance by converting it into an SCFG.

straightforward. Figure 5.3c shows the (only) feasible solution to the SCFG problem that respects the bounds B^{out} for all views. Finally, Figure 5.3d shows the conversion of the SCFG solution into a DBA solution, i.e., a Degree-bounded Arborescence. Clearly, both conversions DBA→SCFG and SCFG→DBA can be done in linear time.

Finally, the problem of finding an SCFG $\in NP$ since a non-deterministic algorithm only needs to guess a solution for a given graph G and then check in polynomial time whether this solution is indeed an SCFG. To do this, a traversal of the graph is sufficient (linear time).

In this proof we have reduced the NP-hard DBA problem to the SCFG problem, and therefore the SCFG problem is NP-hard. Since the SCFG problem is a specialization of our original problem (Section 5.2.3), we have shown that our original problem is also NP-hard. □

Importantly, the latest effective techniques for solving DBA and even more general network design problems, rely on solving linear relaxations of *Integer Linear Programs* [LNSS09]. The idea is to use one boolean variable x_i to encode whether a node (or edge) is part of the solution, and to formulate the total utilization (objective function) as a weighted sum of all the variables, with the weights being the respective node (or edge) utilizations. Such an ILP formulation can be handed to an ILP solver, which takes advantage of advanced techniques that enable it to solve large-size problems corresponding in our context to many views and many rewritings.

Two-step Optimization Approach Although our problem (Section 5.2.3) is naturally expressed as a linear program when one considers capacity constraints and

optimizes for utilization (ignoring latency), and can thus be delegated to an ILP solver, it turns out that one cannot rely on an ILP solver to also reduce *latency* (as explained in Section 5.3.3). Thus, our approach for addressing the problem is organized in two steps:

1. Formulate our optimization problem *considering utilization and constraints only* as a linear program and delegate it to an efficient ILP solver. We describe this next in Section 5.3.3.
2. Post-process the utilization-optimal configuration returned by the solver (if one exists under the given constraints) to reduce latency in a heuristic fashion, as described in Section 5.3.4.

5.3.3 CFG Utilization Optimization Through ILP

Integer Linear programming (ILP) is a well-explored branch of mathematical optimizations. A wide class of problems can be expressed as: given a set of linear inequality constraints over a set of variables, find value assignments for the variables, such that a target expression on these variables is minimized. Such problems can be tackled by dedicated *ILP solvers*, some of which are by now extremely efficient, benefiting from many years of research and development efforts. Following the model for directed graphs of [LNSS09] (with some changes), we formulate our problem as an integer linear program as follows.

Variables For each node $n \in V \cup \{\mathcal{D}\} \cup A$, we denote by E_n^{in} and E_n^{out} the sets of its incoming and respectively outgoing edges. Selecting a CFG amounts to selecting *one way to compute each view*, which is equivalent to selecting for each view v, one of the $\wedge$ nodes pointing to v, or, equivalently, one edge from E_v^{in}. Thus, for each $v \in V$ and $e \in E_v^{in}$, we introduce a variable x_e, taking values in the set $\{0, 1\}$, denoting whether or not e is part of the CFG.

Coefficients Our problem model attached rewriting evaluation utilization to the rewriting nodes, through the utilization function Ureturning for each $\wedge$ node $a \in A$, the associated utilization $\mathcal{U}(a)$ which aggregates various types of utilizations (CPU, I/O, network, etc.) Further, as explained in Section 5.2.2, $\mathcal{U}(a)$ is the smallest over the utilizations of all physical plans that could be used for this rewriting. To simplify the presentation, and since there is a bijection between A, the set of $\wedge$ node sets, and the set of edges entering view nodes, namely $\cup_{v \in V} E_v^{in}$, we *move the utilization of each rewriting, to the edge going from the rewriting $\wedge$ node, to the corresponding rewritten view*. The other edges, in particular all those entering $\wedge$ nodes, are assumed to have zero utilization. Thus, for each rewriting node $a \in A$ and edge $e \in E_a^{out}$ (recall that $E_a^{out} = \{e\}$, that is, each a node has exactly one outgoing edge), we denote by $\mathcal{U}_e$ the utilization $\mathcal{U}(a)$. Our final ingredient is the B_v^{out} bounds on the views fan-out, introduced in Section 5.2.2.

Putting it All Together Our problem's ILP statement is given in Table 5.1. Equation (1) states that each x_e variable takes values in $\{0, 1\}$, (2) ensures that every

$$\text{Minimize: } \mathcal{U} = \sum_{e \in E} \mathcal{U}_e x_e$$

subject to:

$$x_e \in \{0,1\} \qquad \forall e \in E \qquad (1)$$

$$\sum_{e \in E_v^{in}} x_e = 1 \qquad \forall v \in V \qquad (2)$$

$$\sum_{e \in E_a^{in}} x_e = x_{E_a^{out}} \times |E_a^{in}| \qquad \forall a \in A \qquad (3)$$

$$\sum_{e \in E_v^{out}} x_e \leq B_v^{out} \qquad \forall v \in V \cup \{\mathcal{D}\} \qquad (4)$$

Table 5.1: Utilization optimization problem as a linear program.

view is fed exactly by one rewriting, (3) states that if the (only) outgoing edge of a $\wedge$ node is selected, all of its inputs are selected as well, and finally (4) ensures the respect of the B_v^{out} constraint.

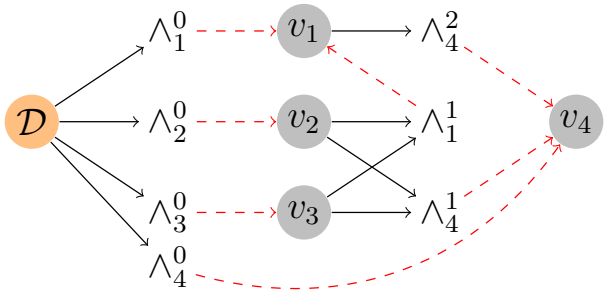
LP Example Consider the RG shown at the top of Figure 5.4, where for illustration we have added to each $\wedge$ node leading to the view v_i, the subscript $_i$ and a superscript ^{j} with $j = 0, 1, \ldots$. For each edge (n, m) in the RG, where n and m are two RG nodes, we introduce a variable $x_{n \to m}$ stating whether that edge is part of the chosen configuration. For simplicity, for each node $\wedge_i^j$ pointing to the view v_i, we write x_i^j instead of $x_{\wedge_i^j \to v_i}$. Thus, x_i^j is a boolean variable whose value 1 indicates that the view v_i is filled by its rewriting $\wedge_i^j$. Moreover, for each $\wedge_i^j$, let c_i^j be the utilization of the processing incurred by that rewriting.

The linear program whose solution is a minimum-utilization CFG for this graph is shown in the lower part of Figure 5.4. Equation numbers at the left refer to the generic equations in Table 5.1.

Non-linearity of Latency Still on the RG in Figure 5.4, we now turn to quantifying the latency of each view. Let l_i^j be the latency of each rewriting $\wedge_i^j$; for simplicity we include therein the impact of all the transfers and processing incurred by the rewriting.

We consider that $\mathcal{D}$ implements an efficient algorithm allowing it to match *simultaneously* all the subscriptions it serves, against each newly published document. This is the case in state-of-the-art algorithms such as [DAF+03], and also in our simpler implementation. Thus, the latency component that is due to *subscription matching at* $\mathcal{D}$ (as opposed to latency incurred by shipping data from $\mathcal{D}$ and possibly further processing and shipping of data) is the same for all views, and we ignore it without loss of generality.

Applying our formulas defining latency, we obtain $\lambda(v_2) = l_2^0$, $\lambda(v_3) = l_3^0$, since



Minimize: $\mathcal{U}_1^0 x_1^0 + \mathcal{U}_1^1 x_1^1 + \mathcal{U}_2^0 x_2^0 + \mathcal{U}_3^0 x_3^0 + \mathcal{U}_4^0 x_4^0 + \mathcal{U}_4^1 x_4^1 + \mathcal{U}_4^2 x_4^2$
subject to:

eq.(1)	$x_i^j \in \{0,1\}, \forall i,j$
eq.(2)	$x_1^0 + x_1^1 = 1;\ x_2^0 = 1;\quad x_3^0 = 1;\quad x_4^0 + x_4^1 + x_4^2 = 1;$
eq.(3)	$x_{\mathcal{D}\to\wedge_1^0} = x_1^0;\ x_{\mathcal{D}\to\wedge_2^0} = x_2^0;\ x_{\mathcal{D}\to\wedge_3^0} = x_3^0;$ $x_{\mathcal{D}\to\wedge_4^0} = x_4^0;\ x_{v_1\to\wedge_4^2} = x_4^2;$ $x_{v_2\to\wedge_1^1} + x_{v_3\to\wedge_1^1} = 2x_1^1;\ x_{v_2\to\wedge_4^1} + x_{v_3\to\wedge_4^1} = 2x_4^1;$
eq.(4)	$x_{v_1\to\wedge_4^2} \leq B_{v_1}^{out};\ x_{v_2\to\wedge_1^1} + x_{v_2\to\wedge_4^1} \leq B_{v_2}^{out};$ $x_{v_3\to\wedge_1^1} + x_{v_3\to\wedge_4^1} \leq B_{v_3}^{out};$ $x_{\mathcal{D}\to\wedge_1^0} + x_{\mathcal{D}\to\wedge_2^0} + x_{\mathcal{D}\to\wedge_3^0} + x_{\mathcal{D}\to\wedge_4^0} \leq B_{\mathcal{D}}^{out};$

Figure 5.4: Sample RG and corresponding ILP model.

v_2 and v_3 are fed directly from the publisher. Since v_1 can be fed either through $\wedge_1^0$ or $\wedge_1^1$, its latency is:

$$\lambda(v_1) = x_1^0 l_1^0 + x_1^1(l_1^1 + max(\lambda(v_2), \lambda(v_3))) = x_1^0 l_1^0 + x_1^1(l_1^1 + max(l_2^0, l_3^0))$$

Similarly, given that v_3 can be fed through three different $\wedge$ nodes, we have:

$$\lambda(v_4) = x_4^0 l_4^0 + x_4^1(l_4^1 + max(\lambda(v_2), \lambda(v_3)) + x_4^2(l_4^2 + \lambda(v_1)) =$$
$$x_4^0 l_4^0 + x_4^1(l_4^1 + max(l_2^0, l_3^0)) + x_4^2(l_4^2 + x_1^0 l_1^0 + x_1^1(l_1^1 + max(l_2^0, l_3^0)))$$

Observe that the above expression unfolds into a sum having among its terms $x_4^2 x_1^0 l_1^0$ and $x_4^2 x_1^1 l_1^1$, which is *non-linear in the problem's variables* x_i^j; in contrast, the latencies of v_1, v_2 and v_3 are *linear* combination of these variables. As a consequence, in these examples and in general, *configuration latency cannot be pushed into the ILP objective function*, which only admits linear combinations of variables.

The intuition behind this non-linear behavior is easy to trace on the RG in Figure 5.4. The variables which end up multiplied correspond to paths of length 2, leading to v_4 *through* v_1. If $x_1^0 = x_4^2 = 1$, v_1 is fed from the source and v_4 from v_1. If $x_1^1 = x_4^2 = 1$, v_1 is fed from v_2 and v_3 and v_4 from v_1. The multiplication of variables corresponds to the logical conjunction of the edge selection decisions they correspond to.

Concluding this discussion, we will rely on ILP to solve efficiently and exactly the utilization optimization problem, and reduce in a second step the latency of the configuration thus obtained.

5.3.4 CFG Latency Optimization

In this second stage, we seek to improve the latency of the CFG obtained by solving the ILP problem (corresponding to the utilization minimization under constraints), by incremental changes on this CFG. We start by introducing a helper notion:

Impact of a View on CFG Latency Given a CFG cfg, we define the *impact* of a view v, denoted by $\mathcal{I}(v)$, as an estimation of v's impact on the latency of all of the views that are fed with data by v, directly or indirectly. Formally:

$$\mathcal{I}(v) = \lambda(v) \times |\text{nodes of } rg \text{ reachable from } v|$$

In the above, we consider that any rg node reachable from v is potentially impacted by the latency introduced by v, and, thus, multiply v's latency by the number of such nodes. We also define the impact of a rewriting rw_v pointing to view v to be equal to the impact of v: $\mathcal{I}(rw_v) = \mathcal{I}(v)$.

The LOGA Algorithm We have devised a Latency Optimization Greedy Algorithm (LOGA, in short), given in Algorithm 5, which incrementally tries to improve the latency of a CFG cfg obtained from an RG rg. The algorithm uses the original rg in order to replace a rewriting in cfg with another one that leads to a CFG with a globally smaller latency. It initially orders the rewritings of cfg in descending order of impact, and then tries to replace first the rewritings with the biggest impact. Such replacements are made (i) without violating the B^{out} bounds, and (ii) without assigning views again to $\mathcal{D}$, since the goal of our work is precisely to spread the data dissemination work.

Incremental Re-computation of Latency As explained above, a change in the latency of a view v in a CFG cfg might affect the latency of every view in cfg accessible from v. Therefore, when the latency of v changes as a consequence of a replacement, LOGA performs a traversal in topological order of the cfg sub-DAG rooted at v, to recompute the latency only of the affected views.

Recomputing Impact of Views As the CFG changes through rewriting replacements, the number of nodes reachable from any given view node v must be recomputed. This number is needed in order to update the impact $\mathcal{I}(v)$, at line 5 of Algorithm 5. The number of nodes reachable from v is determined by the rewriting opportunities, which in turn depend on the actual views etc. In the worst case this may require a costly traversal of the whole CFG, however, as our experiments show (Section 5.5), much fewer nodes are traversed and thus this operation is not expensive in practice.

5.3.5 Incremental CFG Computation

Adding a new view v to an existing configuration cfg, goes as follows: we compute v's rewritings and add them to the existing RG. We then search the RG for a rewriting rw with the least cost $\mathcal{C}(rw)$ such that no bounds are violated in

Algorithm 5: Latency Optimization Greedy Algorithm (LOGA)

Input : CFG cfg, RG rg
Output: Latency optimized version of cfg

```
1  newLat ← λ(cfg)
2  repeat
3      prevLat ← λ(cfg)
4      rwList ← {rw ∈ cfg | ∄ edge (D, rw)}
5      rwList ← reorder(rwList) in desc. order of interest I(rw)
6      foreach rw ∈ rwList do
7          minLat ← λ(cfg); best_rw ← null
           // Replace rw with its latency-optimal alternative (if any)
8          foreach rw' ∈ rg s.t. rw, rw' feed the same view do
9              replace rw with rw' in cfg
10             if (∀v ∈ cfg, outdegree(v) ≤ B_v^out) and (λ(cfg) < minLat) then
11                 minLat ← λ(cfg)
12                 best_rw ← rw'
13             replace rw' with rw in cfg                    // leave cfg intact
14         if best_rw ≠ null then
15             replace rw with best_rw in cfg
16             newLat ← λ(cfg)
17 until prevLat = newLat
18 return cfg
```

cfg. If such a rewriting rw exists, we add it to cfg; otherwise, v is assigned to the data source. After a certain number of new subscriptions have been added, or when the data source's are been reached, the solver and LOGA are re-invoked and a full CFG selection takes place.

When a subscription v is withdrawn or its site fails, the views depending on v, that is those to whose cfg rewritings v contributes, are treated as new and the above incremental process is followed for each of them.

5.4 View-based Rewriting

We now describe the view-based rewriting framework underlying Delta. Section 5.4.1 presents some preliminary notions on views and rewritigs, whereas Section 5.4.2 describes an auxiliary structure, the embedding graph, which is used for building the RG. Then, Section 5.4.3 presents our algorithm for efficiently rewriting a subscription (view) based on the others. Its novelty resides in its capability to produce a specified number of solutions, crucial in our setting where not all rewriting opportunities are explored. Finally, Section 5.4.4 discusses how other view-based rewriting algorithms could substitute ours, to port the Delta architecture in other distributed dissemination contexts.

5.4.1 Views and Rewritings

Since our target applications concern the dissemination of structured text news, and in order to leverage our previous system development [KKMZ12, MKVZ11], we built our system for disseminating XML documents to a network of subscriptions expressed in a rich flavor of XML queries.

Each view is defined by a *tree pattern query* (as described in Section 2.1.4) where nodes are labeled with XML element or attribute names, while edges encode parent-child (single) or ancestor-descendant (double) relationships. Unlike XPath 1.0, and close to XPath 2.0 and to simple XQuery for-let-where-return (FLWR) expressions, our tree patterns may return content from multiple nodes.

Note that unlike in the previous two chapters, Delta does not consider joins between tree patterns for two, mostly technical, reasons. The first reason is the fact that more engineering effort was required in order to develop a fast rewriting algorithm for joined tree patterns. The second reason is that a more sophisticated update mechanism would be required in order to update joined tree pattern views (as they allow joins of tuples that may come from different documents). In this work we focus more on the algorithms for building rewriting graphs and retaining low-utilization and low-latency configurations. We use tree patterns as an example data model that Delta can be based on.

Node IDs are implemented by virtually all efficient XML engines. Therefore, we include IDs in our views, since, as we have shown in [KMV12], view joins based on such IDs may lead to very efficient rewritings. As a simple example, consider the query q defined as $//a[//c]//b$ and the views $v_1 = //a$, $v_2 = //a_{ID}[//c]$ and $v_3 = //a_{ID}//b$, where v_2 and v_3 store IDs for the a nodes. One can rewrite q as $v_2 \bowtie_{a.ID} v_3$, or alternatively as $v_1[//c]//b$. The former is likely to be much more efficient than the latter, because v_2 and v_3 are more selective than v_1, especially if few a elements have b and/or c descendants.

[MKVZ11] provides an equivalent view-based rewriting algorithm for this language. Unsurprisingly, this algorithm has high complexity, therefore, it is not applicable in a setting like ours with a very large numbers of views. Therefore, we consider here a sub-language of the one considered in [KMV12, MKVZ11] where we assume that *all nodes are annotated with* ID. Moreover, to increase the possibilities of view-based rewriting, we assume *IDs are structural*: by comparing two node IDs one can decide if the node corresponding to the one is a parent/ancestor of the node corresponding to the other. Node IDs are invisible to the user; they are added by the system to the user-issued tree patterns. Storing IDs in subscription data brings a space overhead, but not a very significant one, especially if one relies on space-efficient encodings of such views [WTW09]. Restricting the view language to endow all nodes with ID reduces view-based rewriting to a set-cover problem, as we explain shortly below.

View Embedding It has been shown [MKVZ11, TYÖ+08] that a tree pattern view v may participate in an equivalent rewriting of another tree pattern view q only if there exists an embedding $\phi : v \to q$ respecting (1) node labels, i.e., for any node

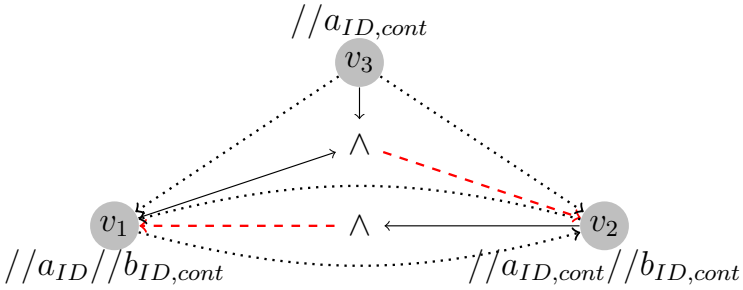


Figure 5.5: Superposed EG and RG over three views.

$n \in v$, $label(n) = label(\phi(n))$, and (2) structural relationships between nodes, that is, for any two nodes $n, m \in v$, if n is a $/$-child (resp., $//$-child) of m, then $\phi(n)$ is a $/$-child (resp., descendant) of $\phi(m)$. Finally, ϕ must not contradict value predicates from the query, i.e., for any node $n \in v$, such that $m = \phi(n) \in q$, if m is annotated with predicate $[val = c_1]$ for some constant c_1, then n must not be annotated with predicate $[val = c_2]$ for some constant $c_2 \neq c_1$. It follows readily from the above properties of embeddings that:

Corrolary 5.4.1. *If a view v embeds into a query q, the labels of v are a subset of the labels of q.*

View Coverage We say that a set of views V *covers* a given view q, iff, for every attribute att of a node $n_q \in q$, there exists a node n_v belonging to a view $v \in V$ and an embedding $\phi : v \rightarrow q$ such that $\phi(n_v) = n_q$ and n_v is also annotated with att. We call such a view set V an *embedded attribute set cover* (EAC) for q.

If we restrict the rewriting algorithm [MKVZ11] to the case when all view nodes are annotated with ID, it can be shown that the existence of an EAC V for q is a necessary and sufficient condition for an equivalent rewriting of q based on V to exist. Indeed, given an EAC V for q, the rewriting can be built using structural joins (based on the node IDs) between all the involved views, and adding all required structural predicates (imposing structural relationships present in the query but not in the views), as well as possible value selection predicates still needed. We formalize this as follows:

Proposition 5.4.1. *A query q can be equivalently rewritten based on a set of views V, iff V is an EAC for q.*

Observe that such a rewriting may be non-minimal; we revisit this issue in Section 5.4.3.

5.4.2 Embedding Graph (EG)

Given a view set V, in order to build the corresponding RG, we must solve $|V|$ view-based rewriting problems, one for each view based on the others. To speed

up the rewriting process, we can exploit Proposition 5.4.1 to attempt to rewrite a given view v, only using those views that embed into v. Thus, we are interested in all view pairs (v_1, v_2) such that v_1 embeds into v_2. We encode this embedding information in an *embedding graph* (EG, in short), which is a directed graph having a node for each view $v \in V$ and an edge (v_1, v_2), with $v_1, v_2 \in V$, iff v_1 embeds in v_2. Figure 5.5 depicts a sample EG (view nodes, dotted edges), along with the corresponding RG (view and $\wedge$ nodes, solid and dashed edges). Next to each view node, we give its view definition. For instance, v_3 embeds in v_1 and v_2 (as shown by the dotted edges).

Testing whether v embeds into v' takes at most $|v| \times |v'|$ operations [MKVZ11], leading to a total complexity of $O(|V|^2 \times |v|^2_{max})$ for creating the EG, where $|v|_{max}$ is the size of the largest view in V. Such tests may get quite expensive for large V sets.

To improve performance, we pre-filter views, based on Corrolary 5.4.1: for v to embed into v', the labels of v must be among the labels of v'. We organize the view definitions in a prefix trie specifically designed to support subset queries [HK99]. Using this trie, given a view v, we can efficiently identify all the views u_i such that $labels(u_i) \subseteq labels(v)$.

Algorithm 6 shows how to construct an EG given a set of views V. The algorithm starts by constructing a trie as explained above. Then, it uses the trie as an index to efficiently build the EG: for a given view v, the trie returns all views whose labels are a subset of v's labels. Only the views thus obtained are tested for embedding into v. Since our pre-filtering has no false negative, Algorithm 6 generates the complete EG.

EG Cycles and their Consequences It is possible for two views to embed into each other, as for example v_1 and v_2 in Figure 5.5, leading to cycles in the EG. In some cases, cycles in the EG lead to cycles in the RG. For instance, in Figure 5.5, although the EG cycle between v_1 and v_2 does not directly translate to an RG cycle, view v_3 enables some additional rewritings (such as the one represented by the upper $\wedge$ node), and in turn these lead to an RG cycle (involving v_1, v_2 and the two $\wedge$ edges).

RGs featuring such cycles pose an issue since the ILP solver may return a CFG with cycles, e.g., feeding v_1 from v_2 and v_2 from v_1 in this example, without using the publisher $\mathcal{D}$ at all. Such CFGs do not make sense from the application perspective, since the data path feeding each view must start at the publisher $\mathcal{D}$.

It can be shown that an RG has cycles only if the EG it has been built from had cycles. To avoid RGs (and CFG) cycles, we break EG cycles using the cycle removal algorithm [ELS93].

5.4.3 View-based Rewriting Algorithm

We now describe our rewriting algorithm (Algorithm 7). As stated in Proposition 5.4.1, to find rewritings of v it suffices to find all embedded attribute set covers (EACs) of v, and to build an efficient rewriting from each such EAC.

Algorithm 6: Trie-based EG Construction Algorithm

Input : View set V
Output: EG of V

```
1  E ← ∅; EG ← (V, E)                                  // Initially empty edge set
2  T ← createTrie(V)                                    // Create the trie for V
3  foreach v ∈ V do
       //Retrieve from T all u s.t. labels(u) ⊆ labels(v) and add edges
         corresponding to embeddings
4      foreach u ∈ {T.lookUp(v)} do
5          if u embeds into v then E ← E ∪ {(u, v)}
6  return EG
```

Algorithm 7: Cover-based greedy rewriting (CGR)

Input : View v, EG $eg = (V_{eg}, E_{eg})$, max. number k of rewritings
Output: List with at most k rewritings of v based on the views of eg

```
   // Get from eg all views embeddable in v
1  V ← {u_i | (u_i, v) ∈ E_eg}
2  rwList ← ∅                                  // List with rewritings for v
3  visited ← ∅                                 // Set of already visited EACs
4  if ∃ attribute att ∈ v, not covered by any u ∈ V then return ∅
5  crtEAC ← ∅                                           // Current EAC view set
6  backtrackFindEAC(v, V, crtEAC)
7  return rwList
8  Procedure backtrackFindEAC(v, V, crtEAC)
9      if crtEAC covers all v's attributes and crtEAC ∉ visited then
10         visited ← visited ∪ {crtEAC}
           // Get rewriting from EAC and add to rwList
11         rwList.add(EACtoRw(crtEAC))
12         if (rwList.size = k) then return
       // Get views not yet used in crtEAC
13     remainViews ← V \ crtEAC
14     if remainViews = ∅ then return
15     remainViews ← sort(altViews) in desc. order of interest i
16     foreach v_alt ∈ altViews do
17         crtEAC ← crtEAC ∪ {v_alt} backtrackFindEAC(v, V, crtEAC)
18         crtEAC ← crtEAC \ {v_alt}
```

The novelty of our algorithm is that it generates solutions *incrementally on-demand*, a useful feature given that we only consider k alternative rewritings for each subscription (recall Section 5.3.1). Since all rewritings may never be developed, Algorithm 7 strives to develop the most promising rewritings first, that is those whose evaluation utilization is likely to be low. This is done by ordering

candidate views in decreasing order of their *interest* w.r.t. rewriting (covering) a given view v: the more v attributes *currently uncovered* by a partial rewriting are covered by a view v', the more interesting it is to add v' to (join it with) the respective partial rewriting. Clearly, as views are added to the rewriting, view interests have to be recomputed. The algorithm is based on depth-first exploration and backtracks to move from one rewriting to the next one.

First, the algorithm uses the EG to retrieve the view set V containing only the views embeddable in v. The EAC exploration starts with an empty EAC, and at each point the highest-interest view not already in the current EAC is added to it. We compute the interest of adding a candidate view u to the EAC, given that a subset of V has already been selected, by counting *how many attributes of v not covered by the EAC views, are covered by the candidate u*.

For example, when rewriting view v $/a_{ID,cont}/b_{ID,cont}$ and considering a candidate view $u_1 = /a_{ID}/b_{ID,cont}$, the interest of u_1 is 3, since u_1 covers the attribute ID in two nodes of v as well as $b.cont$. Once u_1 is selected, the interest of another candidate view $u_2 = /a_{ID,cont}/b_{ID}$ is 1, since the only attribute of v not previously covered by u_1 and covered by u_2 is $a.cont$. When several views have the same interest, the tie is broken by picking the one that covers attributes from *the largest number of v nodes*. Once an EAC for v is found, we transform it to a rewriting expression and add it to the list of rewriting solutions.

In the worst case, Algorithm 7 will develop all subsets of V. However, in practice, since we only seek k rewritings, the number is typically much less, as we verified through our experiments.

Rewriting Minimization Algorithm 7 may generate rewritings which include redundant views. These views may be removed from the rewriting while leaving it still equivalent to the target view. Non-minimality is due to the greedy nature of Algorithm 7: after a view u was included in a rewriting, another set of views $\{u_1, u_2, \ldots, u_k\}$ may be added such that, together, the views in the set cover all attributes that u was selected for. This makes u redundant although it was not when initially added. To build efficient (non-redundant) rewritings, we minimize them in a post-processing fashion as in [TYÖ+08]: remove a random view from a non-minimal rewriting, then check if this has compromised the rewriting. If yes, the view is put back in the rewriting, another view is removed, etc.

5.4.4 Generality of our Approach

The core concepts and framework of Delta, discussed in Section 5.2, are independent of the concrete underlying data model, query language and query rewriting algorithm. While Delta is currently implemented and deployed for XML subscriptions, it can be easily adapted to another data model and subscription language. We briefly discuss the rewriting-related components needed to do so.

First, an algorithm for equivalent view-based query rewriting is needed, such as proposed in the literature, e.g., for relational [PH01] or XML data [TYÖ+08, MKVZ11]. In particular, the set-cover-based algorithm described above can be

used as-is if we model subscriptions simply as *key-value pairs*, e.g., "topic=sport and location=England", as considered in many publish-subscribe data management settings such as e.g., [CDTW00]. We rely on this algorithm to build the RG.

Second, while building the EG is optional, for many-view settings it is likely to significantly improve performance, by limiting the view set input to the rewriting algorithm. The embedding criterium we used to build the EG has natural counterparts in other data models, e.g., the classical containment mappings [CM77]. If these are not implemented or their computational cost is high, the EG can be approximated using any non-lossy pruning. For instance, if one considers relational queries as subscriptions, we could add an edge (v_1, v_2) in the EG as soon as the tables in v_1 are a subset of those in v_2, and for each table, the constants used in selections on that table in v_1 are used in selections over the same tables in v_2.

5.5 Experimental Evaluation

In this Section we present the experimental evaluation of our system. We describe our setup in Section 5.5.1, and discuss the construction of EGs and RGs in Section 5.5.2. Section 5.5.3 studies the utilization-based selection of CFGs through ILP, while Section 5.5.4 discusses how to improve the latency of such CFGs. Finally, Section 5.5.5 presents the deployment of Delta in a wide area network.

5.5.1 Experimental Setup

We implemented all our algorithms in Java, except for the utilization based CFG selection algorithm (Section 5.3.3), for which we made use of the Gurobi ILP solver [Gur13]. We relied on YFilter [DAF+03] to generate our views, based on the XMark DTD [SWK+02].

We have generated two view sets, V_1 and V_2 of 100,000 views, the characteristics of which are shown in Table 5.2. For V_1, we opted for unique views in order to examine the scalability and efficiency of our algorithms in the absence of trivial rewritings (where equivalent views rewrite one another) and force our utilization and latency optimizations algorithms to consider more complicated CFGs (rather than chains of equivalent views that can be easily optimized).

For V_2 we opted for only 31,925 unique views, whereas the rest are duplicates. This view set is chosen so as to observe the impact of duplicate views in the shape of the RGs and CFGs that are generated by our algorithms.

All our experiments ran on an 8-core server (2 CPUs, Intel Xeon @2.93GHz), with 16GBs of RAM and running CentOS Linux 6.4.

	View Set Metric	Value
	Number of views (unique)	100,000
	Avg. number of predicates per view	0.72
	Avg. number of predicates per node	0.11
	Avg. number of nodes per view	6.13
	Avg. number of return nodes per view	2.52
	EG Metric	**Value**
V_1	Number of edges	10,592,053
	Number of edges deleted to remove cycles	18,665
	% of views in which at least one view is embedded	99.95
	Generation time (sec)	452
V_2	Number of edges	2,033,296
	Number of edges deleted to remove cycles	4,692
	% of views embedded by at least another view	100%
	Generation time	56 sec
	RG Metric	**Value**
V_1	Number of rewritings ($\wedge$ nodes)	2,692,139
	Number of edges	8,589,822
	Generation time (sec)	127
	Views rewritten by other views	94,835
	Avg. number of views used in a rewriting	2.15
	Avg. $\|E^{out}\|$	57.9
V_2	Number of rewritings ($\wedge$ nodes)	2,587,687
	Number of edges	6,527,422
	Generation time	80 sec
	Views rewritten by other views	96,736
	Avg. number of views used in a rewriting	1.48
	Avg. $\|\delta^{out}\|$	38.3

Table 5.2: Experiment settings and EG/RG statistics.

5.5.2 EG and RG Generation

We have generated the EG using Algorithm 6, then removed cycles from it, and finally generated the RG using Algorithm 7. Algorithm 7 was instructed to generate no more than $k = 30$ rewritings for each view. The sizes and generation times for the EG and RG appear respectively in the middle and bottom of Table 5.2. Every time Algorithm 7 finds a rewriting, we create the corresponding $\wedge$ node, with an outgoing edge toward the rewritten view, and with an incoming edge from each view used in the rewriting. Table 5.2 shows that the number of rewritings (and thus, the size of the unrestricted RG) is very high, more than 2.5 millions.

5.5.3 CFG Utilization Optimization Through ILP

In the experiments involving the view set V_1, we have set the upper bound of the data source as $B^{out}_{\mathcal{D}} = 6,198$, that is, the number of views that cannot

	B^{out}	30	50	100	∞
V_1	% rewritten views	94.3	94.7	94.7	94.7
	CFG utilization ($\times 10^{13}$)	3.49	3.32	3.31	3.13
	Average views per rewriting	1.77	1.78	1.79	1.8
V_2	% rewritten views	-	-	96.7	96.7
	CFG utilization ($\times 10^{13}$)	-	-	1.93	1.93
	Average views per rewriting	-	-	1.24	1.24

Table 5.3: Impact of B^{out} on the selected CFGs.

be rewritten by other views (see Table 5.2) plus a 20% margin. The respective bound for the view set V_2 was set to $B^{out}_{\mathcal{D}} = 3,916$. We did this in order to push to the data source $\mathcal{D}$ the least possible load, while giving the ILP solver some margin to assign some extra views to $\mathcal{D}$ if needed. We have also set a common $B^{out} = \{30, 50, 100, \infty\}$ for all views (to see the effect of bounds on the shape of the resulting CFGs).

The Gurobi solver was then used to select utilization-optimal CFGs. A first observation was that the running time *decreases* as B^{out} increases, from about four minutes for $B^{out} = 30$ to less than two minutes for $B^{out} = \infty$. The reason is that a small B^{out} corresponds to highly restricted settings where the solver must search longer in order to find acceptable solutions.

Table 5.3 depicts the percentage of views rewritten using other views (and not filled from the data source $\mathcal{D}$) in the CFGs returned by the ILP solver, as well as the utilization of the CFGs and the average number of views that take part in the rewritings. First, notice that even when we keep the load on the views under tight control ($B^{out} = 30$), we achieve a high degree of off-loading (94.3% for V_1 and 96.7% for $V2$) from the data publisher $\mathcal{D}$. Moreover, as can be seen, by decreasing B^{out} in V_1, the utilization of the CFG increases (due to tighter constraints), while the number of views participating in a rewriting decreases (since each view is allowed to serve less views).

In Table 5.3 we can see that the cost of CFGs for V_2 are considerably lower than the ones of V_1. This is due to the fact that the cost of serving one view from (another) identical view, is very low compared to performing joins that are expensive. Note also the difference in the average views per rewriting: in V_1 almost 1.8 views in average are used to serve another view, while for in V_1, this number is considerably lower (1.24).

Finally, we observed that the solver could not generate CFGs (i.e. no configuration was feasible) for the duplicate-rich view set V_2 for $B^{out} < 100$. This happened because some of the views in V_2 were too popular. Assigning all other views that depended on the popular one could not be done without breaking the bound B^{out} that was given to the solver and they had to be assigned to the data source. Since there was also a relatively low bound on the data source ($B^{out}_{\mathcal{D}} = 3,916$) that had to also be respected, the ILP was infeasible.

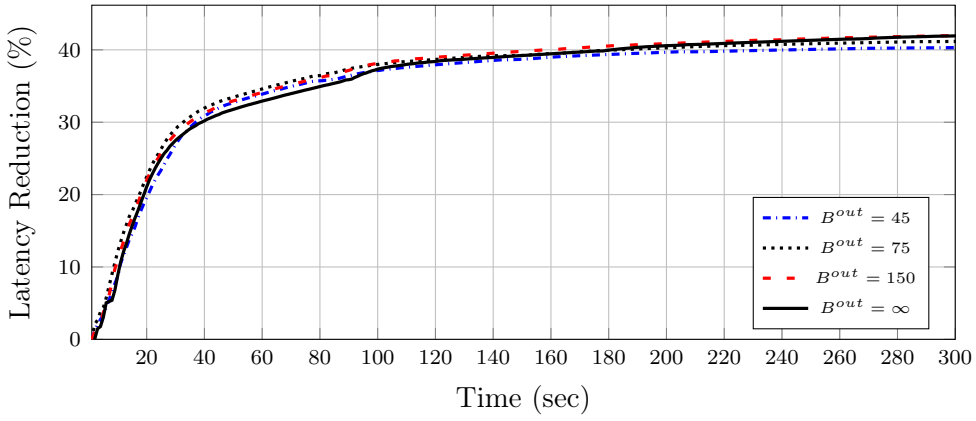


Figure 5.6: Latency reduction while running LOGA for V_1.

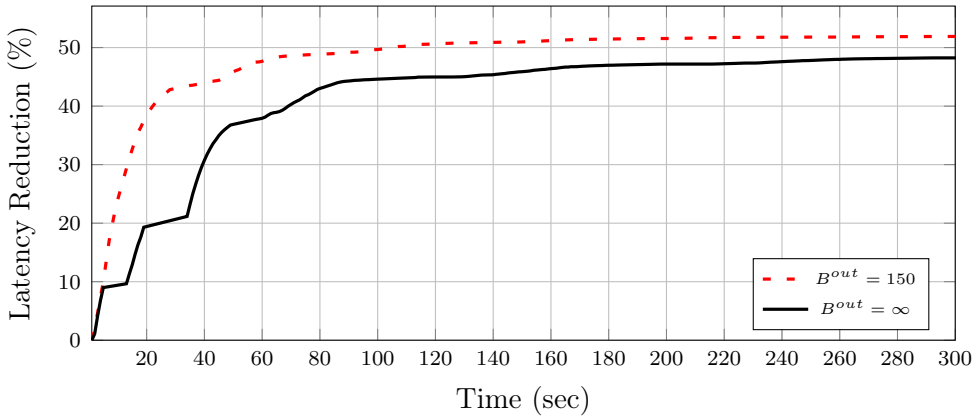


Figure 5.7: Latency reduction while running LOGA for V_2.

This experiment showed that in case a view set contains some very popular views, one has to increase the bound of the data source $B^{out}_{\mathcal{D}}$. We have experimented further in that direction and we saw that increasing the bound of the data source to $B^{out}_{\mathcal{D}} = 6000$ (almost double the previous bound), the solver could finally generate CFGs for $B^{out} = 50$ but not for $B^{out} = 30$.

5.5.4 Greedy CFG Latency Optimization

We now study the performance of Algorithm 5 (LOGA, Section 5.3.4), applied on CFGs obtained through ILP optimization. Our initial experiments did not show significant latency improvement, because the ILP-selected CFGs exploited most of the freedom we gave them (almost every view was feeding B^{out} other views). Hence, there was very little leeway for LOGA to make changes. To circumvent this problem, we allowed LOGA to use as bound 1.5 times the B^{out} given to the ILP solver. Thus, where the ILP solver had $B^{out} = 30, 50, 100$, LOGA used $45, 75, 150$, respectively.

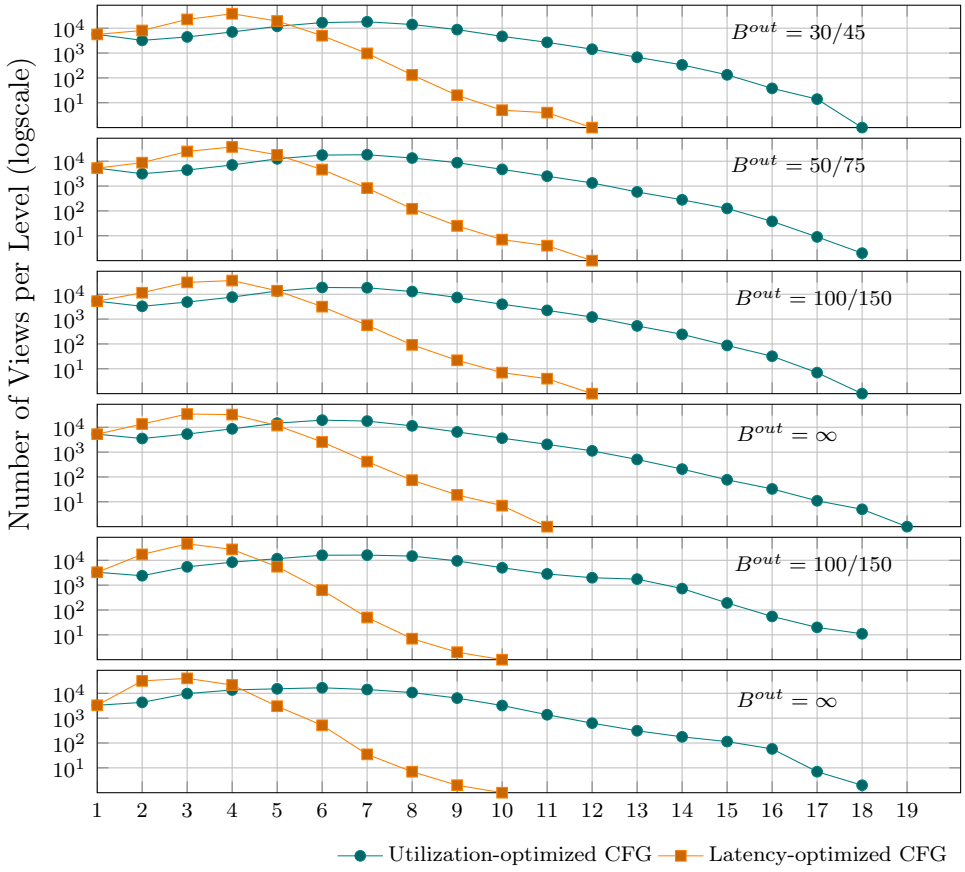


Figure 5.8: Distribution of views across CFG levels for view sets V_1 (top 4) and V_2 (bottom 2).

Latency Optimization Figure 5.6 depicts the latency improvement as a function of the LOGA running time. We see that LOGA is very effective, achieving a 43% reduction with respect to the latency of the CFG returned by the initial ILP solver. Moreover, such savings are obtained within 150-200 seconds. They stabilize when the data propagation paths to all the high-impact views have been altered and there is not much room for further optimization.

Similarly, Figure 5.7 depicts the latency improvement for the view set V_2. We see that in this case LOGA is more effective, achieving a 50% reduction with respect to the latency of the CFG returned by the initial ILP solver. This is explained by the fact that the duplicates in the graph leave a room for further improvement.

Distribution of Views into Levels Figure 5.8 depicts the distribution of views into levels in the CFGs for varying B^{out}, as produced (i) by the ILP solver, and (ii) after LOGA optimization. Note the logarithmic vertical axis. We see that the latency-optimized CFGs have less than 2/3 of the number of rewriting levels of the CFGs produced by ILP. Moreover, in the latency-optimized CFGs, most of the views lie in levels 1-6, leaving approx. only 1.5% of the views on levels 6-12. Thus, most views are only 4-5 hops away from the data source. This "flattening of rewriting

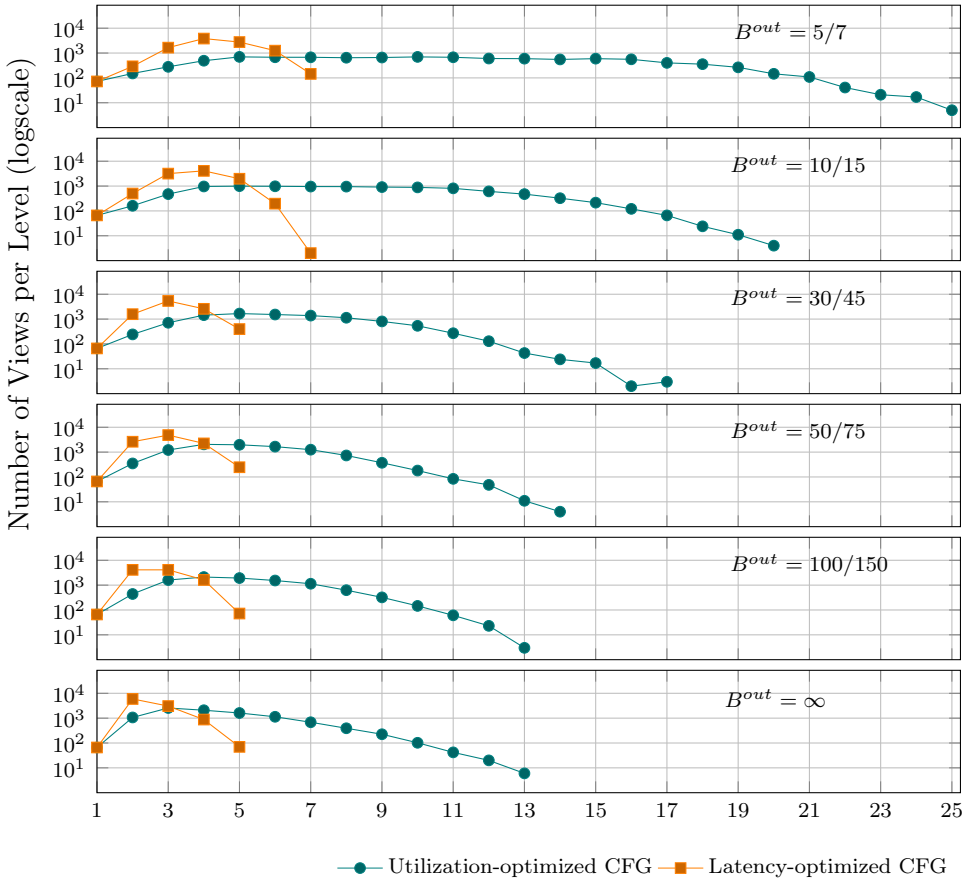


Figure 5.9: Distribution of views across deployed CFG levels.

levels" is an expected result of LOGA, since the more levels the data passes through from the publisher to a view, the more latency is added.

Utilization vs. Latency Although one may expect latency optimization (that reached 50%) to re-increase utilization, the increase was very moderate (5-7%). LOGA is only making greedy incremental fine-tuning over utilization-optimized CFGs (whose bounds were already attained), and therefore, the changes in the graph could not significantly change utilization.

5.5.5 Experiments in a WAN Deployment

We deployed Delta's algorithms on top of the distributed query execution engine of ViP2P [KKMZ12]. We implemented a full set of continuous physical operators (structural joins, selections, buffers etc.) mostly based on the physical operators of ViP2P. We report here on experiments we carried deploying Delta in a WAN.

Infrastructure We conducted our experiments in the Grid5000 infrastructure [Gri], using 300 machines distributed over nine major cities across France and

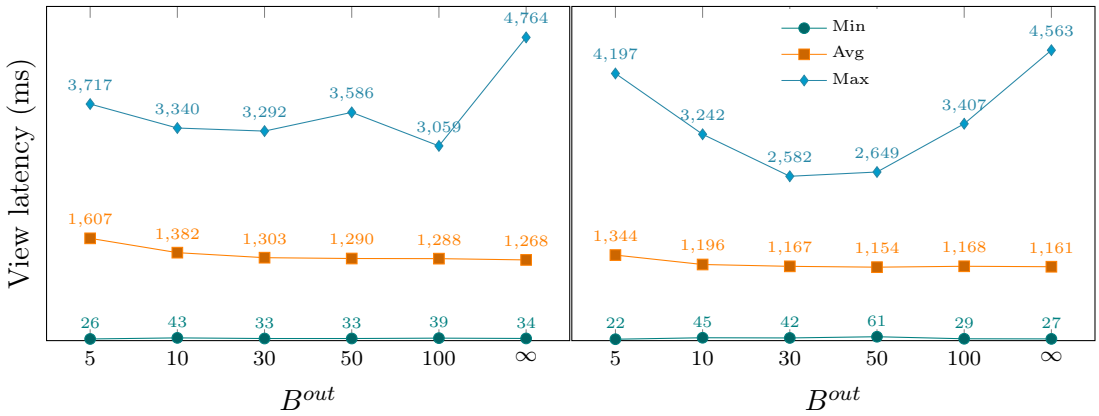


Figure 5.10: View latency for utilization-optimized CFGs (left) and latency-optimized CFGs (right).

Luxembourg. The hardware of Grid5000 machines varies from dual-core machines with 2GBs of RAM to 16-core machines with 32GBs of RAM. This heterogeneous hardware distribution is likely to occur with real settings as subscribers have varied-capacity machines.

Views and Documents We have generated a set of 10,000 views, along with a set of 200 small (10-40KB) XMark [SWK+02] documents, in a way such that each document matches almost all of the views. Unlike our previous experiment, this view set has only 3,000 unique views, which is more representative of real-life scenarios where some subscription topics are popular.

We have created the corresponding EG and RG and invoked the ILP solver to generate utilization-optimized configurations for $B^{out} \in \{5, 10, 30, 50, 100, \infty\}$ and $B_{\mathcal{D}}^{out} = 72$. The resulting CFGs were optimized for latency with the LOGA Algorithm with bounds $\{7, 15, 45, 75, 150, \infty\}$.

The distribution of views into levels is depicted in Figure 5.9. A first observation is that in the presence of duplicate views, the latency-optimized CFGs can have less than half of the levels of their utilization-optimized counterparts. A CFG with duplicate views is easier to optimize through the LOGA Algorithm since equivalent views may be served from one another.

We now move to presenting our results from deploying the generated CFGs. To characterize the performance of Delta, we have measured two important metrics, namely the *observed latency* and the *document delivery time*.

Observed View Latency We measured the latency of a view v for a document d as the time elapsed between: (i) the moment when the first tuple of d leaves the data source, and (ii) the instance when the last tuple of d reaches the view v. Note that in the observed view latency we do not include the time needed to extract the level 1 view tuples from a document. We do not include this [4] since this

4. For completeness: our view matcher took an average of 100ms to extract from each document the tuples for the 72 first-level views.

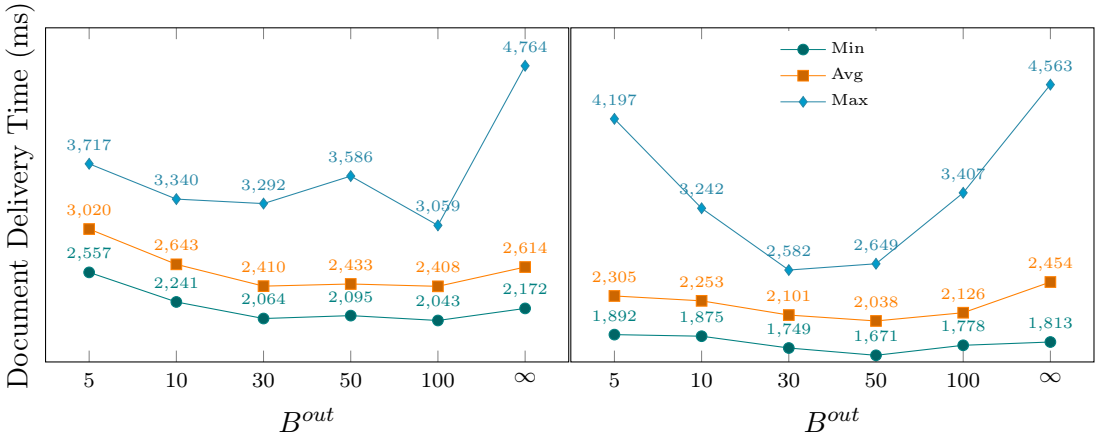


Figure 5.11: Document delivery time for utilization-optimized CFGs (left) and latency-optimized CFGs (right).

extraction step is not the main scope of the paper and has been studied in other works [DAF+03, HDG+07].

Figure 5.10 depicts the average observed view latency for all pairs of views and documents in our CFGs. A first observation is that on average, views get their results in just 1.6 seconds after a document is published. This translates to a throughput of many thousands of subscriptions served per second, with a data source having to serve only 0.7% (72 out of 10.000) of the views. This demonstrates how Delta makes it possible to serve large numbers of subscribers using very little publisher computing resources.

Our second remark regards the minimum/maximum latencies for $B^{out} = 5$ in utilization-optimized CFGs. Some views in the network receive their results extremely fast (30ms) while some others considerably slower (3.7s). This is an inherent feature of Delta: views that are close to the data source receive their data faster than the ones that reside in deeper levels.

The LOGA algorithm reduces the observed latency of views up to 20% ($B^{out} = 5$) compared to the utilization-optimized CFGs. This also shows that our latency estimation models (used by our algorithms) are quite accurate.

An interesting phenomenon is the following: in the utilization-optimized CFG where $B^{out} = \infty$ we notice a very large increase in the maximum latency (4.7s) while the CFG is not too deep (13 levels) compared to other CFGs that showed lower latency. This is explained by the fact that when a view serves a very large number of other views, it can be overloaded and the data processing/transmission throughput is reduced. This shows the importance of the bounds B^{out} in Delta: for optimal performance, B^{out} must be set in the "sweet spot" between values too large (to avoid overloading) and too low (to avoid very deep CFGs). In practice, a simple test can be performed at each subscriber machine to tailor its B^{out} to its observed hardware performance.

Document Delivery Time For a view v and a document d that matches v, we term *document delivery time*, or simply DDT, the total time needed for all the matching tuples of document d to reach the view v. For a set of views V, the DDT is measured as the interval between: (i) the moment when the first tuple of the document d *leaves* the data source and (ii) the instance when the last tuple of the document d has reached the slowest view $v \in V$. In other words, this metric captures the time it takes for a document to reach its slowest interested view.

Figure 5.11 shows the average, minimum and maximum DDT over all published documents in our experiment. In general, in all CFGs, a document is delivered to all views in the network, in an average of 2-2.5 seconds. Note that the maximum observed latency coincides with the maximum DDT (see Figures 5.10 and 5.11) as the slowest view in the network actually defines the DDT. Thus, we observe the same phenomenon as in the observed latency: DDT slows down for the extreme $B^{out} = \{5, \infty\}$ values.

5.5.6 Experiment Conclusion

Our experiments have demonstrated the efficiency and effectiveness of Delta's multi-level dissemination approach. With respect to efficiency, for 100,000 distinct subscriptions, the full graph generation, optimization for utilization and then latency took less than 13 minutes. As for effectiveness, the configurations retained have low cost scores. This is confirmed by the WAN deployment of 10,000 subscriptions, which showed a high message delivery throughput and low latency: documents are propagated to 10,000 subscriptions, which are fed with data within 1.5 seconds on average.

5.6 Related Works

Our work belongs to the class of content-based publish subscribe systems, disseminating to users the results of their specified subscriptions over a stream of published data. This work is related to several themes of existing works.

Filtering Systems A large part of the literature addresses the problem of optimizing the publisher so that it handles the filtering of incoming data for very large numbers of subscribers.

YFilter [DAF+03] stands out as a widely-known system for XML publish-subscribe. It is able to feed many XPath 1.0 subscriptions very efficiently by matching them simultaneously against documents through a single automaton. NiagaraCQ [CDTW00] relies on multi-query optimization for continuous queries, taking advantage of the similarity of subscriptions in order to share operators during evaluation. Similarly, [HDG+07] addressed the same problem but for a more expressive subscription language, supporting joins over multiple documents while [TATV11a, TATV11b] use methods of multi-query optimizations for continuous queries over RSS feeds. Finally, [TRP+04] proposes a pub/sub system where the evaluation of subscriptions is done inside a relational database.

The above do not consider distributed data dissemination. Instead, they focus on optimizing the publisher task, to support very large numbers of subscribers. Our work can be seen as complementary since we focus on the design of a logical overlay network (CFG), that exploits the subscribers in order to scale up. Any efficient filtering at the publisher can be adopted in our setting.

Distributed Publish/Subscribe Onyx [DRF04] connects multiple publishers and subscribers by employing multiple YFilter instances running on connected brokers. Recently, FoXtrot [MK12] has distributed YFilter automata on top of a DHT network. Other DHT-based pub/sub systems are, e.g., [CIKN04, GSAA04]. Closer to our work, SemCast [Pap05] leverages commonalities between subscriptions and creates logical *channels* between brokers and subscribers to form multicast trees of low utilization and latency. However, the system relies on a network of brokers, and the subscribers do not help in the dissemination of data. Finally, [TBF+03] builds one multicast tree per broker aiming at redundancy and fault tolerance.

Contrariwise, in [CF05], *every peer* can forward messages to its neighbors if the message matches its own interests. Peers are organized in an hierarchy tree based on subscription similarity. However, by design, the peers do not know the subscriptions of their neighbors, and as a result, their routing protocol allows for false positives (peers may receive messages which do not interest them).

In contrast with these works, Delta builds multi-level dissemination networks involving the subscribers, leveraging query rewriting to determine whether some subscriptions can be used to compute results of other subscriptions. One of the consequences unique to Delta is the ability to combine the results of *multiple* subscriptions in order to serve another one.

View-based Data Management As explained in Section 5.4.4, any efficient view-based rewriting algorithm (e.g., [PH01]) can be used instead of our Algorithm 7. View maintenance has been investigated in the centralized context of data warehousing [SF90, RSS96]. In the ViP2P project, incremental algebraic techniques for maintaining materialized views have been described in [BGMS11, BGMS13]. Finally in [DLZ05], the authors consider "stacked" views, specified as queries over other defined views, study their maintenance and the efficient evaluation of queries using such views; these resemble our multi-level configurations, but in [DLZ05] the connections between views are given, whereas we choose them for performance through our algorithms.

5.7 Future Work

Delta, considers complete rewritings, that is, subscriptions are either served from the data publisher or solely from other subscriptions. When the system starts to run and the number of subscribers is limited, subscriptions do not overlap enough to create lots of rewriting opportunities. In that case, Delta will fail to rewrite all subscriptions based solely on other subscriptions and the publisher will have to serve them all individually. In that case, the publisher will be overloaded

or even refuse to serve some of the subscriptions.

A possible solution to this overloading would be to use *partial rewritings*. A partial rewriting would allow a subscription to be served partly by other subscriptions and partly by the publisher. This way, the load on the publisher can be reduced since some of the processing effort is pushed to other subscribers.

Supporting partial rewritings in Delta's algorithms is relatively straightforward: given an appropriate rewriting algorithm, the rewritability graph can be easily enriched with partial rewritings. In turn, the enriched rewritability graph can be used by our algorithms without any other change.

In contrast, in order to support partial rewritings a publisher should be slightly modified. Instead of materializing only level 1 views of a given configuration, the publisher would have to also store and serve the extra views for the evaluation of partial rewritings. To sustain the extra effort on the data publisher, one could use the view selection algorithm that was presented in Chapter 3. The possibility of supporting partial rewritings in Delta and selecting custom materialized views for the publisher is left for future work.

5.8 Summary

In this chapter we considered the problem of scaling up content-based publish/subscribe systems under resource constraints (such as finite CPU and network capacity) by off-loading some of the data publisher's effort on the subscriber sites. This is achieved by organizing subscriptions in a rewritability graph which materializes the ways in which one subscription could be served from others, through view-based rewriting. We provide a novel two-step algorithm for organizing the views in a network minimizing a combination of resource utilization and data dissemination latency. First, we express the utilization minimization problem as a linear program and solve it exactly; as we show, latency cannot be included in the ILP formulation due to its non-linear nature. We reduce latency in a second step based on the result obtained from the ILP solver. Our configuration choice algorithm scale well to 100.000 unique subscriptions, whereas in a WAN deployment, Delta succeeds in filling in 10.000 subscriptions with a latency of under 2 seconds.

Acknowledgments We would like to thank Cédric Bentz for his valuable help and discussions on the ILP modeling and Yannis Manoussakis for his guidance in proving the NP-Hardness of our problem.

Chapter 6

Conclusion and Future Work

Vastly increasing quantities of Web data is published every day, and this growth shows no signs of stopping. What is more, a large percentage of this data is published in XML format. Thus, the urgency for new systems in the area of data management that will enable scalable distributed data management has never been greater. In this thesis we have focused on the (distributed) management of XML data based on techniques that employed materialized views.

6.1 Thesis Summary

In this thesis we focused on three problems that we summarize below.

Materialized View Selection for XQuery Workloads considers the problem of choosing the best views to materialize within a given space budget in order to improve the performance of a query workload.

— This work is the first to formalize and address view selection problem for queries and views expressed in a rich subset of XQuery.
— We analyzed the space of potential candidate views and present several effective candidate pruning criteria.
— We proposed ROA, a heuristic algorithm based on state search that uses a set of transformations to navigate in the search space of candidate view sets.
— We experimentally demonstrated the superiority or the ROA algorithm, compared to the state of the art.

Distributed View-based Web Data Dissemination We presented the ViP2P platform, a distributed platform for sharing XML documents based on a structured P2P DHTs.

— ViP2P uses *distributed materialized XML views*, defined by arbitrary XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. More specifically:

— We presented the complete architecture of ViP2P enabling views indexing, XML data publication, view materialization and finally query evaluation over distributed materialized views.
— We have fully implemented our architecture, on top of the FreePastry [Fre] P2P infrastructure. and presented a comprehensive set of experiments performed in a WAN, demonstrating ViP2P's superiority compared to the state of the art.
— The ViP2P platform scaled to several hundreds of peers and hundreds of GBs of XML data, both unattained in previous works.

Delta: Scalable View-based Publish/Subscribe We presented Delta, a novel approach for scalable content-based publish/ subscribe. Delta achieves scalability by off-loading subscriptions from the publisher, and leveraging view-based query rewriting to feed these subscriptions from the data accumulated in others. The main contributions that Delta brings to the literature are:

— Delta is the first pub/sub system that considers that uses distributed materialized views for the dissemination of data.
— We presented a novel algorithm for organizing views in a multi-level dissemination network, exploiting view-based rewriting, using integer linear programming capabilities to scale to many views, respect capacity constraints, and minimize latency.
— We presented LOGA, a heuristic algorithm that works over given configurations and optimizes them for latency.
— We provided a full implementation of our architecture and we presented extensive experiments validating the efficiency and effectiveness of our algorithms. We confirmed our algorithms in practice, through a large deployment in a WAN.

6.2 Perspectives

Materialized View Selection for XQuery Workloads We plan to extend our problem definition and include the view maintenance costs as well as storage costs during the state search. In that case, more algorithms utilizing our transformations could be devised.

Our algorithms could incorporate *query templates* as in [MS05, YLH03], grouping together similar queries under a single representative, to support larger workloads. We also plan to investigate the extension of the query template mining approach of [YLH03], focused on single-view XPath rewritings, to our more complex language and query rewriting framework.

Finally, since XML data management highly concerns distributed databases, an obvious next step for our view selection algorithms would be to consider distributed databases for which views should be selected. In that case, the problem of view selection for a distributed network of sites becomes even more complex: one has to incorporate view placement methods to our algorithms as well.

Delta: Scalable View-based Publish/Subscribe We plan to demonstrate the applicability of our configuration algorithms to a non-XML data model, for instance relational or RDF, and plug into our architecture other rewriting algorithms such as [PH01].

So far, we assumed that the views included in a configuration can be only the ones requested by the subscribers. We are interested in the inclusion of a view selection method (like the one of Chapter 3) that could modify existing or, add extra intermediate views (some form of "broker" views) in a configuration in order to optimize it for lower overall costs.

MapReduce-based Web Data Processing Since 2004 and especially over the duration of this thesis, interest has been growing in processing of data through massively parallel frameworks such as MapReduce [DG04] and related platforms and implementations [Apa, DIS, BEH+10]. At the same time, enterprises are increasingly shifting away from deploying analytical databases on high-end proprietary machines and systems, and moving towards cheaper, lower-end hardware [TSJ+09].

MapReduce-based systems are better suited due to their scalability, fault tolerance, and flexibility to handle un- or semi- structured data. Moreover, in traditional DBMS, data needs to first be loaded and indexed before it is query-able. However, real-time analytics and the very nature of data publication on the Web (continuously grows and old data does not change) dictate that data should be query-able in an online, continuous and incremental fashion. Previous research has focused on the incremental processing of data in MapReduce environments without the need of loading all data in advance [LMD+12] and for reusing computations [BWR+11, PBYI09].

For all the above reasons, we expect that the efficient large-scale processing of Web data in massively distributed environments, will attract significant interest in the near future.

Traditional database operators such as joins have been "ported" and optimized for execution in Map-Reduce environments [AU10, OR11], bringing MapReduce one step closer to database systems. This opens the the way for materialized views in MapReduce environments: materialized views can be used as a means of storing and reusing intermediate results of query computations. To that direction, one could analyze MapReduce programs and decide where and how to incorporate materialized views that would be incrementally updated whenever new data arrives.

Bibliography

[Abe11] Karl Aberer. Peer-to-peer data management. *Synthesis Lectures on Data Management, Morgan & Claypool Publishers*, Volume 3:p.87–94, 2011.

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[ABMP08] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path Summaries and Path Partitioning in Modern XML Databases. *World Wide Web Journal*, 2008.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[AMM05] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. MemBeR: A Micro-benchmark Repository for XQuery. In *EXPDB*, 2005.

[AMP+08] Serge Abiteboul, Ioana Manolescu, Neoklis Polyzotis, Nicoleta Preda, and Chong Sun. XML processing in DHT networks. In *ICDE*, 2008.

[AMR+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[AMR+12] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2012.

[Apa] Apache Foundation. The Apache Hadoop Project. http://hadoop.apache.org/.

[AU10] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110. ACM, 2010.

[AYCLS02] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[BC06] Angela Bonifati and Alfredo Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.

[BCD+10] Anastasia Bezerianos, Fanny Chevalier, Pierre Dragicevic, Niklas Elmqvist, and Jean-Daniel Fekete. Graphdice: A system for exploring multivariate social networks. In *Computer Graphics Forum*, volume 29, pages 863–872. Wiley Online Library, 2010.

[BCH+06] Kevin S. Beyer, Roberta Cochrane, M. Hvizdos, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Matthias Nicola, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Ashutosh Singh, Tuong C. Truong, Robbert C. Van der Linden, Brian Vickery, Chun Zhang, and Guogen Zhang. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006.

[BDB] Oracle Berkeley DB Java Edition. http://www.oracle.com/technology/products/berkeley-db/je/index.html.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *ACM symposium on Cloud computing*, New York, NY, USA, 2010. ACM.

[BGMS11] Angela Bonifati, Martin Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. In *EDBT*, 2011.

[BGMS13] Angela Bonifati, Martin Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. *ACM TODS*, 2013.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006.

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

[BK09a] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4), 2009.

[BK09b] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41, 2009.

[BKN09] Nikhil Bansal, Rohit Khandekar, and Viswanath Nagarajan. Additive guarantees for degree-bounded directed network design. *SICOMP*, 2009.

[BKS02] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[BMCJ04] Angela Bonifati, Ugo Matrangolo, Alfredo Cuzzocrea, and Mayank Jain. XPath lookup queries in P2P networks. In *WIDM*, 2004.

[BMKL02] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. ToXgene: a template-based data generator for XML. In *SIGMOD*. ACM, 2002.

[BOB+04] 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.

[BSX] BaseX. The XML database. http://basex.org/.

[BWR+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

[CC10] Ding Chen and Chee-Yong Chan. ViewJoin: Efficient view-based evaluation of tree pattern queries. In *ICDE*, pages 816–827, 2010.

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

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

[CF05] Raphaël Chand and Pascal Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Euro-Par*, 2005.

[CF10] Artem Chebotko and Bin Fu. XML reconstruction view selection in XML databases: Complexity analysis and approximation scheme. *LNCS*, 6509, 2010.

[CHS02] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *VLDB J.*, 11(3):216–237, 2002.

[CIKN04] P. A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P networks. In *ESWS*, 2004.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[CLM+04] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. An indexing framework for peer-to-peer systems. In *SIGMOD*, 2004.

[CLM+07] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-Ring: An efficient and robust P2P range index structure. In *SIGMOD*, 2007.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[CRKMR10] Jesús Camacho-Rodríguez, Asterios Katsifodimos, Ioana Manolescu, and Alexandra Roatis. LiquidXML: Adaptive XML Content Redistribution. In *CIKM (demo)*, 2010.

[DAF+03] Yanlei Diao, Mehmet Altinel, Michael J Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 2003.

[DFS99] Alin Deutsch, Mary F. Fernández, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD Conference*, pages 431–442, 1999.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[DGL00] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *J. Log. Program.*, 43(1):49–73, 2000.

[DIS] The Disco Project. http://discoproject.org/.

[DKP12] Marios D Dikaiakos, Asterios Katsifodimos, and George Pallis. Minersoft: Software Retrieval in Grid and Cloud Computing Infrastructures. *TOIT*, 12(1):2, 2012.

[DLAV10] William Kokou Dedzoe, Philippe Lamarre, Reza Akbarinia, and Patrick Valduriez. ASAP Top-k query processing in unstructured P2P systems. In *Peer-to-Peer Computing*, 2010.

[DLZ05] David DeHaan, Per-Ake Larson, and Jingren Zhou. Stacked Indexed Views in Microsoft SQL Server. In *SIGMOD*, 2005.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470, 1999.

[DRF04] Y. Diao, S. Rizvi, and M.J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.

[DT03] Alin Deutsch and Val Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[DZD+03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *IPTPS*, 2003.

[EAZZ09] Iman Elghandour, Ashraf Aboulnaga, Daniel Zilio, and Calisto Zuzarte. Recommending XMLTable Views for XQuery Workloads. In *XSym workshop*, 2009.

[Edm67] Jack Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 1967.

[ELS93] Peter Eades, Xuemin Lin, and W.F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 1993.

[eXi] eXist, an Open Source Native XML Database. http://exist.sourceforge.net.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[FM00] Thorsten Fiebig and Guido Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB (Selected Papers)*, pages 125–136, 2000.

[Fre] Freepastry, an open-source implementation of pastry. http://freepastry.org/FreePastry/.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability*. Freeman New York, 1979.

[GKLM10] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In *CIKM*, 2010.

[GKLM12] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(1), 2012.

[GKM09] Michaela Gotz, Christoph Koch, and Wim Martens. Efficient algorithms for descendant-only tree pattern queries. *Information Systems*, 2009.

[GL01] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[GM99a] Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, 1999.

[GM99b] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, 1999.

[GM05] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[GNT09] Olivier Gauwin, Joachim Niehren, and Sophie Tison. Bounded delay and concurrency for earliest query answering. In *LATA*, 2009.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD Conference*, 1990.

[Gri] Grid'5000 network infrastructure. https://www.grid5000.fr/.

[GSAA04] A. Gupta, O.D. Sahin, D. Agrawal, and A.E. Abbadi. Meghdoot: Content-based Publish/Subscribe over P2P networks. In *Middleware*, 2004.

[Gup97] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.

[Gur13] Gurobi Optimizer. `http://www.gurobi.com`, 2013.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[GWJD03] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[HDG+07] M. Hong, A.J. Demers, J.E. Gehrke, C. Koch, M. Riedewald, and W.M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[HK99] J. Hoffmann and J. Koehler. A new method to index and query sets. In *JCAI*, 1999.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[IHW01] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating Network-Bound XML Data. *IEEE Data Eng. Bull.*, 24(2):20–26, 2001.

[JOT+06] H.V. Jagadish, B.C. Ooi, K. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD*, 2006.

[JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. BATON: a balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

[JSO] JSON. http://www.json.org.

[Kar12] Konstantinos Karanasos. *View-Based Techniques for the Efficient Management of Web Data*. Phd thesis, INRIA Saclay & Université Paris Sud, June 2012.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[KFCGR10] Asterios Katsifodimos, Jean-Daniel Fekete, Alain Cady, and Cecile Germain-Renaud. Visualizing the Dynamics of e-Science social networks (poster). *The 5th EGEE User Forum*, 2010.

[KKMZ11] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. The ViP2P Platform: XML Views in P2P. Technical Report RR-7812, INRIA, November 2011.

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

[KMV12] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized View Selection for XQuery Workloads. In *SIGMOD*, 2012.

[KP05] Georgia Koloniari and Evaggelia Pitoura. Peer-to-peer management of XML data: issues and research challenges. *SIGMOD Record*, 34(2), 2005.

[KRML05] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. FiST: Scalable XML document filtering by sequencing twig patterns. In *VLDB*, 2005.

[KZ10] Konstantinos Karanasos and Spyros Zoupanos. Viewing a world of annotations through AnnoVIP. In *ICDE*, 2010.

[LHH+04] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *VLDB*, 2004.

[LLCC05] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, 2005.

[LM09] Zhen Hua Liu and Ravi Murthy. A Decade of XML Data Management: An Industrial Experience Report from Oracle. In *ICDE*, pages 1351–1362, 2009.

[LMD+12] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. SCALLA: A Platform for Scalable One-Pass Analytics Using MapReduce. *ACM Transactions on Database Systems (TODS)*, 2012.

[LNSS09] Lap Chi Lau, Joseph (Seffi) Naor, Mohammad R. Salavatipour, and Mohit Singh. Survivable network design with degree or order constraints. *SIAM J. Comput.*, 2009.

[LP08] Kostas Lillis and Evaggelia Pitoura. Cooperative XPath caching. In *SIGMOD*, 2008.

[MB12] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Record*, pages 20–29, 2012.

[MCB11] Imene Mami, Remi Coletta, and Zohra Bellahsene. Modeling View Selection as a Constraint Satisfaction Problem. In *DEXA*, 2011.

[MK12] Iris Miliaraki and Manolis Koubarakis. Foxtrot: Distributed structural and value XML filtering. *ACM TWEB*, 2012.

[MKVZ11] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.

[MPV09] Ioana Manolescu, Yannis Papakonstantinou, and Vasilis Vassalos. XML tuple algebra. In *Encyclopedia of Database Systems*. Springer, 2009.

[MS04] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1), 2004.

[MS05] Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[MW99] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *VLDB*, pages 315–326, 1999.

[MZ09a] Ioana Manolescu and Spyros Zoupanos. Materialized views for P2P XML warehousing. In *BDA (informal proceedings)*, 2009.

[MZ09b] Ioana Manolescu and Spyros Zoupanos. XML materialized views in P2P. In *DataX workshop (not included in the proceedings)*, 2009.

[ODPC06] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960. ACM, 2011.

[ÖV11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

[Pap05] Olga Papaemmanouil. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

[PBYI09] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. Dryadinc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing*, 2009.

[PCS+04] Shankar Pal, Istvan Cseri, Gideon Schaller, Oliver Seeliger, Leo Giakoumakis, and Vasili Vasili Zolotov. Indexing XML Data Stored in a Relational Database. In *VLDB*, pages 1134–1145, 2004.

[PGI04] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Approximate XML query answers. In *SIGMOD*, 2004.

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

[PKD10] George Pallis, Asterios Katsifodimos, and Marios D. Dikaiakos. Searching for software on the egee infrastructure. *Journal of Grid Computing*, 8(2):281–304, 2010.

[PZIÖ06] Derek Phillips, Ning Zhang, Ihab F. Ilyas, and M. Tamer Özsu. InterJoin: Exploiting indexes and materialized views in XPath evaluation. In *SSDBM*, pages 13–22, 2006.

[QLO03] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ICDSP*, November 2001.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, 2nd edition, 2000.

[RM09a] Praveen Rao and Bongki Moon. An internet-scale service for publishing and locating XML documents. In *ICDE*, 2009.

[RM09b] Praveen R. Rao and Bongki Moon. Locating XML documents in a peer-to-peer network using distributed hash tables. *IEEE TKDE*, 21, 2009.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. In *SIGMOD*, 1996.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD*, 1979.

[Sax] SAXON: The XSLT and XQuery Processor. http://saxon.sourceforge.net.

[SF90] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *ICDE*, 1990.

[SHA05] Gleb Skobeltsyn, Manfred Hauswirth, and Karl Aberer. Efficient processing of XPath queries with structured overlay networks. In *CoopIS*, 2005.

[SMGC05] Carlo Sartiani, Paolo Manghi, Giorgio Ghelli, and Giovanni Conforti. XPeer : A Self-Organizing XML P2P Database System. In *Current Trends in Database Technology (EDBT 2004 Workshops)*, 2005.

[SSB+00] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In *VLDB*, 2000.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[TATV11a] Jordi Creus Tomàs, Bernd Amann, Nicolas Travers, and Dan Vodislav. RoSeS: A continuous content-based query engine for RSS feeds. In *Database and Expert Systems Applications*. Springer, 2011.

[TATV11b] Jordi Creus Tomàs, Bernd Amann, Nicolas Travers, and Dan Vodislav. RoSeS: a continuous query processor for large-scale RSS filtering and aggregation. In *CIKM*. ACM, 2011.

[TBF+03] Wesley Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[tCM07] Balder ten Cate and Maarten Marx. Navigational XPath: calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.

[TGMS08] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable cardinality forecasts for XQuery. *VLDB*, 2008.

[TRP+04] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *ACM SIGMOD*, 2004.

[TS97] Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, pages 1626–1629, 2009.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, 2002.

[TYÖ+08] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[TYT+09] Nan Tang, Jeffrey Xu Yu, Hao Tang, M. Tamer Özsu, and Peter A. Boncz. Materialized view selection in XML databases. In *DASFAA*, 2009.

[W3C04] W3C. Extensible Markup Language (XML) 1.0, 2004.

[W3C07a] W3C. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007.

[W3C07b] W3C. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, January 2007.

[W3C07c] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/, January 2007.

[w3c07d] XPath Functions and Operators. www.w3.org/TR/xpath-functions, 2007.

[W3C08] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/, November 2008.

[WPJ03] Yuqing Wu, Jignesh M. Patel, and H.V. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, 2003.

[WTW09] Xiaoying Wu, Dimitri Theodoratos, and Wendy Hui Wang. Answering XML queries using materialized views revisited. In *CIKM*, 2009.

[XML] XML Schema. http://www.w3.org/TR/XML/Schema.

[XO05] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[YGM03] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *ICDE*, 2003.

[YLH03] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, 2003.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

[Zou09] Spyros Zoupanos. *Efficient Peer-to-Peer Data Management*. Phd thesis, Université Paris Sud, December 2009.