Efficient XML query processing

Ioana Manolescu

THESE

Présentée en vue d'obtention de

HABILITATION A DIRIGER DES RECHERCHE

Discipline : Informatique

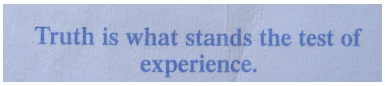
par

Ioana MANOLESCU-GOUJOT
INRIA Saclay–Île-de-France et LRI, Université de Paris XI

Optimization Techniques for XML Query Processing

Jury:

Rapporteurs:	Donald	Kossmann	ETH Zurich
	Torsten	Grust	Universität Tübingen
	Philippe	Pucheral	Université de Versailles Saint-Quentin et INRIA Rocquencourt
Examinateurs:	Serge	Abiteboul	INRIA Saclay –Île-de-France et LRI, Université de Paris Sud
	Véronique	Benzaken	LRI, Université de Paris Sud et CNRS
	Patrick	Valduriez	INRIA Sophia Antipolis - Méditerrannée



Chinese cookie fortune, San Diego, 2005

Contents

Chapter 1

Introduction

Thirty years after the main ideas behind modern database systems have emerged [86], the database field has reached an age of maturity: commercial systems such as those developed e.g. by IBM, Oracle, and Microsoft, and free systems such as Postgres or MySQL are very widely used and back the daily activity of countless organizations. Yet the maturity reached on the well-trodden paths does not preclude vibrant activity at the borders of the models, algorithms, and platforms that we, as a scientific community, are able to build and deploy efficiently.

Research in databases is pushing the boundaries of knowledge in several directions. It reaches towards the very large, considering huge databases such as required e.g. by scientific data management [52], and towards the very small, building database systems on smart cards [31]. It is both abstracting away from the hardware details of any particular computer to process large amounts of data in a highly parallel fashion [72], and seeking to exploit new hardware, such as graphic card processors [54], to efficiently implement database operations. The paradigm of interacting with data now goes beyond querying a static corpus, to getting updates from streams (e.g. RSS), and monitoring changes in large data corpora (e.g. the Web). Going away from the relational model which made its success, database knowledge and tools are by now stable enough to support industrial-scale XML applications, backed by a stack of standards such as XPath [107], XQuery [110], XML Schema [106], SQL/XML [56], and WSDL for Web services [104]. The borders of what used to be called "database-style" resources are being challenged again, to encompass RDF data with little structure [1], and at the other extreme, ordered, large-scale numerical arrays, well-suited for scientific applications [93]. The democratization of databases and the multiplication of electronic data sources led to possibly contradictory or erroneous data, whose exploitation require probabilistic models [42].

Because XML is a standard for Web data, XML query procesing takes a particular importance. The efficient exploitation of XML documents has attracted significant attention since the publication of the XML standard in 1998. XML queries can now be evaluated by mainstream relational database engines extended to support the XML data type and the XQuery language, as well as by in-memory processors. The update extension of the XQuery language is now a standard [112], supported by current XQuery processors.

This thesis describes my work performed in three related, yet complementary areas related to XML query processing. Each of them has involved extensive development. The resulting platforms were demonstrated in major international conferences and were used in the R&D grants in which the group collaborated with various research and industry partners. The research was carried jointly with colleagues and many undergraduate and graduate students, and in particular, three PhD students, two of which have graduated (2007 and 2008), while the third one is set to graduate by the end of 2009.

Section 1.1, Section 1.2 and Section 1.3 present the three lines of research mentioned above. Section 1.4 mentions briefly other research results that are not further detailed in the present document. Each of these sections outlines the content of a corresponding chapter, namely, Chapters 2, 3 and 4. Finally, Chapter 5 outlines perspectives for future work.

1.1 Tree pattern materialized views for XQuery

Chapter 2 is placed in the rather traditional context where XML documents are queried after having been loaded in an XML database. In this context, various indices are established in persistent stores to help processing queries on potentially complex XML structure. Indices typically provide access to all nodes having a given label, or on a given path. Indices for semistructured data have been studied prior to the advent of XML [51], and also later on in the context of XML databases, e.g. [79]. In an XML database using a specific indexing model, each document is automatically indexed. The query processor is built around the indexing model, and uses it to optimize its processing.

If XML indices are required for querying large documents, XML materialized views can provide dramatic performance benefits when a materialized view can be found to closely match a query. Yet, in 2004 when we started to study the problem of materialized views for XQuery processing, no effort had been made in that area, and XPath materialized view proposals only started to emerge.

Chapter 2 outlines our materialized view language, which we term XML Access Modules (or XAMs, in short). XAMs are complex tree patterns, supporting optional and nested edges, value predicates, and multiple-granularity projected attributes for each view node. It is thus possible to store in a single view XML snippets and/or XML text values and/or identifiers of XML nodes, encapsulating various levels of structural information. We have chosen tree patterns as our materialized view language, since, as we show, they capture the essential *data needs* of a query, that is, the nodes that must be accessed in order to verify the query constraints or return its results. Once a materialized tree pattern view matches a query's data needs, the query can be answered by applying simple, efficient result construction steps on the view content.

We explain our query rewriting approach based on the XAM formalism. We proceed by first, extracting from the query the largest possible tree patterns covering its data needs. We then rewrite the query tree patterns based on the available materialized view tree patterns. Finally, we evaluate the rewriting thereby obtained on the materialized views. In Chapter 2, we focus on the novel and difficult problems raised by this approach: the query pattern extraction from a specific XQuery dialect, and on the problem of XAM query rewriting using XAM views. For the latter setting, we first considered an interesting variant of rewriting under structural constraints provided by a path summary, or Dataguide [51], of the database. Some interesting results on the efficient exploitation of a Dataguide for query minimization under constraints were obtained as a spin-off of this work, and are not covered in this chapter [28]. We then study the problem of tree pattern query rewriting without constraints. Interestingly, the time complexity of the rewriting problem is polynomial both with and without Dataguide constraints, although the exponents are not the same.

1.2 ActiveXML optimization

Chapter 3 switches the perspective, moving from the internals of a centralized system, towards data management in the large, in a setting where on distinct peers are deployed interconnected data management applications. We place ourselves in the setting of the ActiveXML declarative language (or AXML, in short). An ActiveXML document is an XML document which may include calls to Web services provided locally

or by remote peers. When a Web service call is activated, its XML results are added to the document, which is thus capable of evolving over the time.

ActiveXML can be seen as an extremely powerful programming language. To start with, an ActiveXML program is able to modify itself, if we allow calls to services that perform arbitrary updates on the document to which they belong. But this is not the most interesting perspective on the language. Rather, we see it as a high-level way of specifying distributed data management applications, expressing *which* data should arrive in our document and *from where*, but not necessarily *how*. Moreover, we consider in particular Web services defined by declarative XML queries over (possibly distributed, possibly Active) XML documents. From this viewpoint, a set of distributed ActiveXML documents can be seen as (and indeed, compiled into) a set of distributed Datalog programs, but unlike the times when Datalog was introduced, ActiveXML benefits from all the stack of XML and Web service-related technologies, such as WSDL [104], XQuery [110] etc.

The ActiveXML research activity has started in the Gemo group at about the time I joined it, in 2002. Chapter 3 outlines three performance-related research questions raised within the AXML framework, and solutions to these questions to which I have contributed. Two older works, from 2003, respectively 2004, are covered very briefly. The first one concerns the replication of ActiveXML documents, complicated by the fact that documents rely on services to build their contents, and services, in turn, may rely on other documents. We devise an automatic replication algorithm capable of producing a standalone replicated document, able to evolve by itself, and we provide a query evaluation algorithm on distributed interconnected documents. The second one studies the issue of evaluating XML queries over AXML documents. In this setting, one may consider fully evaluating the document, and only then applying the query on the result. However, this may lead to activating many unnecessary service calls, whose results do not impact the query result. The problem addressed in this context is then, how to determine which service calls must be activated, and when, in order to compute the query results with as little effort as possible.

Chapter 3 considers in more detail the interesting problem of static AXML optimization. In this setting, we interpret an AXML document as an intensional specification of some data that must arrive in the document after all its service calls have been activated and have finished executing. The AXML framework provides an evaluation engine using a fixed evaluation strategy for each such AXML specification. However, this strategy may be inefficient, and better strategies may be devised. Alternatives include, e.g., *delegating* the evaluation of a service call (or AXML sub-document) to a different peer, *removing* service calls which do not contribute to the query result, *sharing* subtrees produced by equivalent computations, *composing and decomposing* calls to services defined by XML queries etc.

We model the problem as a search space of equivalent ActiveXML documents, each leading to a fixed evaluation strategy, and all producing the same results. We formalize the optimization problem, and provide a generic optimization framework in which *AXML rewriting rules* can be applied to transform a document into another, as suggested by the optimization opportunities above.

The most interesting aspect of the AXML optimization work is the unique position of the optimizer in the overall data management architecture. The optimizer operates at a very high level, actually above the level of XML query processing, since such queries are just components of an AXML application. However, focusing on declarative query services allows understanding the distributed application, and re-formulating it into a more eficient one. The output of the optimizer is again at the high level of AXML, enabling each peer to use (and take advantage of the particular local optimizations of) of any XML database and query processor at the local level.

1.3 XML warehouses on DHTs

The ActiveXML work considered the setting where each resource, whether a document or a service, used in the application resides at a specific location that is known, and specified as such. A separate question is how, in a distributed system, one can identify and exploit efficiently all the available resources, as they are available in the system.

The last decade has seen a profound change in the way digital content is published and shared, due to the advent of peer-to-peer (P2P, in short) networks such as Napster, Gnutella etc. These early platforms adhered to the P2P philosophy by allowing any user to connect and provide his content. At the same time, they sometimes required establishing a centralized catalog, which is a single point of failure, and could not guarantee complete answers or scalable performance. Distributed hash tables (or DHT, in short) have been developed, enabling the routing of messages in a large network of peers guaranteeing both completeness in the absence of peer failure, and performance bounds on the number of messages exchanged in order to route a given message. Moreover, DHTs implement specific mechanisms to cope with peers joining the network and leaving it, as well as with peer failure.

Chapter 4 describes an approach for building XML data management systems on large-scale networks (in the thousands of peers). We consider networks where both peers and document may join and leave, but with a moderate level of churn, i.e. we assume the average uptime of a document in the network is of the order of a few hours at least.

In this context, we have, first, proposed an architecture for indexing the structure and contents of XML documents based on the DHT. The idea is to build *posting lists* for each element name or word occuring inside a text snippet contained in an XML document, and to distribute the posting lists in the DHT in such a way that any peer may find them by looking up the associated element name or word. Based on this index, XML tree pattern queries can be processed by a structural join over the posting lists of the query terms. We have implemented and experimented with this approach. The initial performance results were disastrous. Both document publication, the heaviest operation in terms of network traffic, and index query processing, were prohibitively slow. We have brought several performance improvements which essentially got the DHT up to the task of intensive data transfers. These techniques are independent of the semantics of the data being transferred and would help in any DHT-based application. Moreover, we have proposed two optimization techniques that are specific to the fact that we are managing posting lists, obtained from an XML corpus. These techniques have been implemented in the KadoP system, and shown to scale on networks of up to 1000 peers and 1 GB of published data.

Second, we have considered using DHT networks not only for general-purpose XML indexing, but also for deploying and using access support structures, or materialized views, specifically for certain queries of interest. In this context, independent peers may define views over all the network documents. The network peers are willing to contribute data from their own documents to the views. To enable this, view definitions are indexed in the network, and looked up by document publishing peers. Queries can be processed then, either using a fixed index such as the one of KadoP, or by rewriting them based on the existing materialized views in the network. This approach has been implemented in the ViP2P system, and tested on up to 1000 peers and 3 GB of data.

The complexity of the algorithms involved in KadoP and ViP2P (with the exception of the query rewriting algorithm, polynomial in the size of the query and views) is linear in the size of the data and logarithmic in the size of the peer network. Thus, the systems have the potential to scale beyond the experiment sizes that we considered.

1.4 Other works

Other works to which I have contributed since the obtention of my PhD in 2001 are briefly outlined next.

The XQueC project has focused on the efficient integration of compression in XML databases [27]. A prototype has been developed and demonstrated [26].

Research carried on during my post-doc in Politecnico di Milano, Italy, has lead to several results on the integration of Web services and workflow models in the WebML approach for declarative Web site management [35, 63]. A development of this work, that is taking place now between INRIA and the LRI laboratory of Université Paris XI, concerns the extension of the WebML data-centric workflow model to include support for interactive data visualization; this work frames the PhD thesis of Wael Khemiri (started in 2008).

The XClean project addressed the topic of declarative XML data cleaning. A specific language was proposed for XML data cleaning task, compiled into XQuery programs that could be then processed by efficient XQuery engines [99, 100].

While performing our own XML query processing performance experiments, we have contributed to an open XML micro-benchmark library called MemBeR [18]. Based on an initial set of micro-benchmarks, a performance study on a set of XML query engines was performed [68]. The interest of controlled, pointed experiments such as those of MemBeR was demonstrated by the fact that they have highlighted various performance or correctness bugs in the systems under test, which the respective developers were then able to correct.

A common theme in my research has been to carry out experiments to validate the ideas, stress the techniques, and understand possible shortcomings. This is in my opinion an essential component of applied research. In this spirit, I have participated in various efforts to increase the awareness in the database research community, of the need for higher-quality, extensive experimental evaluation. This started with the creation of the ExpDB (Experimental Evaluation in Databases) workshop in 2006 in cooperation with the ACM SIGMOD conference, and continued through the organization of the SIGMOD 2008 Repeatability evaluation (at the initiative of the SIGMOD 2008 chair, Dennis Shasha), that I have coordinated. The SIGMOD 2009 Repeatability and Workability Evaluation, that I have co-chaired with Stefan Manegold from CWI, enlarged the focus from being able to repeat just the experiments described in the paper, to being able to also perform some others. For all its imperfections, and the hard and often frustrating work that it required, the SIGMOD Repeatability and Workability Evaluation received significant interest from the community [62], and fostered the apparition of many interesting alternatives, such as the experimental track at the VLDB and ICDE conferences and the SIGMOD PubZone [119]. I am confident that when the dust settles, experimental evaluation will have gained in solidity, and we will be able to say the same about the research works validated via experimentation.

Chapter 2

Materialized views and XQuery

This chapter presents *X*ML *A*ccess *M*odules (or XAMs, in short), an extensible and expressive tree pattern language which we introduced in order to speed up the processing of XQuery queries in a persistent store.

Section 2.1 introduces XAMs gradually via a set of examples, discussing at each step which query features are handled by which XAM language feature. Section 2.2 discusses the problem of rewriting XQuery queries based on materialized views described by XAMs, and describes our algorithms for solving this problem. Section 2.3 compares our work with the state of the art both prior to and following our work. Section 2.4 concludes the chapter by presenting the context of this work, i.e. the grants which supported them, and to the PhD and intern students working on the respective projects.

2.1 XML Access Modules

Materialized views for XPath To start, consider the following XPath query:

q_1 //item[description//keyword]//paragraph

The query requires the paragraph descendants of item elements having description children with some keyword descendant. A simple tree pattern representation of query q_1 is depicted at left in Figure 2.1. The paragraph node is gray to highlight that XML components matching this query node are the targets of the query and should be returned.

We now turn to the slightly more complex query q_2:

q_2 //item[description//keyword/text()="gold"]//paragraph

This query features an extra condition: the keyword elements should have a text value equal to the string "gold". The text value of an XML element is computed according to precisely defined rules [111] out of the set of text nodes which are descendants of the XML node. It is important to realize the differences between (i) the text value of an XML element (a string obtained by concatenating only text descendants), (ii) the element itself (a node, that is, a structured object, belonging to the original document), and (iii) the result of serializing the element. The latter is also a string, starting with $<t>$ and ending with $</t>$ (where t is the element's tag), obtained by concatenating the tags and text children of all descendants of the element, in a pre-order traversal [109]. By XPath query semantics, query q_2 should return paragraph

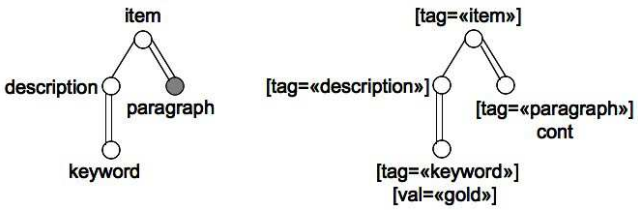


Figure 2.1: Simple query (or view) tree patterns.

elements, corresponding to (ii) above. When one plans to use materialized views, however, given that data is stored in a view for subsequent use, the best one can hope for is to return its serialization, that is, (iii).

The above discussion has highlighted three different information items that can be associated to a node: its label (or tag), its text value, and its serialization. To properly account for these items, we will use henceforth a slightly more verbose, yet more expressive, flavor of tree patterns, as shown at right in Figure 2.1. Here, node names are specified by explicit predicates of the form [tag=t], text values are referred to by the val node attribute, and serialized text images are designated by cont attributes. The pattern at the right hand side in Figure 2.1 thus represents in a precise manner the query q_2 - or, alternatively, a materialized view that could be used to answer the query directly.

Once val and cont are introduced as attributes of any pattern node, observe that nothing prevents annotating several nodes in a given pattern with one or both of these attributes. In this case, it is no longer possible to write down an XPath query equivalent to the pattern, since an XPath expression has at most one target node. Thus, support for these simple node attributes allows to go beyond XPath 1.0 in terms of expressive power for our materialized views - and in particular, it will enable support for complex XQuery queries, as we will see.

Now let us consider a workload consisting of the query q_1, as well as of the following two queries:

q_3 //item//description//keyword/text()
q_4 //item//paragraph

Intuitively, q_3 corresponds to one of the branches in q_2 (except for the selection on the keyword), while q_4 is the other branch of q_2. The common parts among q_2, q_3 and q_4 suggest that two views, one for each branch of q_2, could be used to answer the three queries. Observe that in order to combine the matches of two query branches into matches for a query tree, a join by node identity is needed.

Therefore, we extend our view tree pattern language by allowing to specify that for a given node, an ID (standing for unique identifier) is stored. Two sample tree pattern views are presented in Figure 2.2. Both views store identifiers for the item nodes. Now, q_3 can be answered by a selection over the view at left in Figure 2.2, followed by a projection to eliminate the item identifiers. q_4 can be answered by a similar projection over the view at right. Finally, q_2 can be answered by joining the two views on item IDs, applying a selection on the keyword value, and projecting out the IDs.

Observe that the presence of node identifiers in our tree pattern language goes beyond the XPath 1.0 language [107]. Indeed, such identifiers are not part of the XPath (and XQuery) data model [108]. However, to our knowledge, most (if not all) persistent XML storage systems use some persistent identifiers to capture node identity. Several methods for computing persistent identifiers have been considered in the literature, ranging from simple integers [48] to interval-based identifiers [57], Dewey IDs [96], ORDPaths [73] etc.

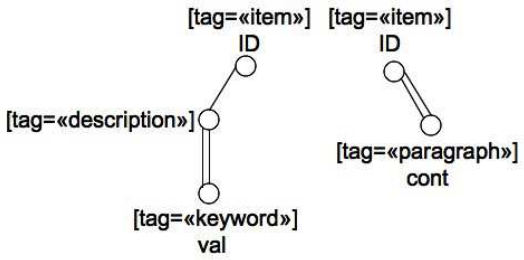


Figure 2.2: Views storing node identifiers.

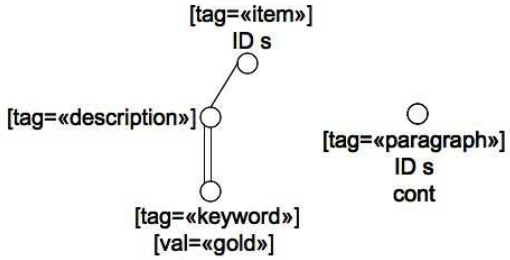


Figure 2.3: Views storing structural node identifiers.

Any of these methods could be used here.

Some methods for computing node IDs exhibit an interesting property: a simple computation on two identifiers allows to establish whether the corresponding nodes are structurally related, i.e., whether one is a parent/an ancestor of the other [57, 73, 96]. This is the case for all the methods cited above, and we term such identifiers *structural IDs*.

When used in materialized views, structural IDs enable more rewritings, as illustrated by the views in Figure 2.3. The two views of this figure can be used to answer q_2, by joining them on the condition that the IDs of item elements designate ancestors of the paragraph elements corresponding to the IDs in the view at right. Such a join is termed a *structural join*, and we consider it here as a logical operator, regardless of its implementation. Several efficient physical structural join operators have been devised in the literature [20, 36]. In our setting of query answering based on views, we stress that the structural properties of the IDs are required in order to answer q_2 based on the views in Figure 2.3 only. The notation ID s is introduced to capture this crucial property.

Materialized views for XQuery The queries q_1-q_4 considered so far were all XPath queries. We now consider the processing of a simple XQuery query, featuring one for-where-return block:

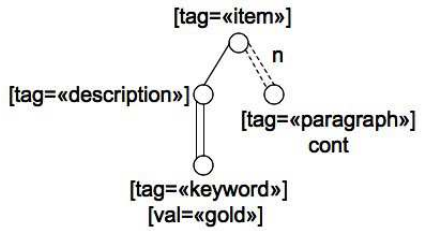


Figure 2.4: Views corresponding to the XQuery query q_5.

q_5 for $x in //item
where $x/description//keyword/text()="gold"
return < res>
{$x//paragraph}
< /res>

Different from all the previous queries, q_5 may return some answers even for an item element having no paragraph descendants, but having the gold keyword in its description. Let us consider how q_5 could be rewritten based on the views previously described. The views in Figure 2.1 cannot be used, as they keep no trace of item elements lacking paragraph descendants. The same can be said of the view at right in Figure 2.2, therefore it is not possible to rewrite q_5 using the views in this figure.

In contrast, one could use the views in Figure 2.3 in order to rewrite q_5 by applying a *structural outer join*, denoted $\bowtie\!\!\!\!\sqsubset$, that pairs every item structural ID from the view at left, with its descendant paragraph elements if they exist, or with a null value otherwise. However, q_5 needs to return only one res element for each item satisfying the for-where conditions, even if it has multiple paragraph descendants. This requires the addition of a group-by operator as follows (denoting by v_1 the view at left and by v_2 the view at right in Figure 2.3):

$$\Gamma_{\text{item.ID}}(v_1 \bowtie\!\!\!\!\sqsubset_{\text{item ancestor of paragraph}} v_2)$$

Once the results of this expression are computed, one only needs to wrap each group produced by Γ in a res element in order to compute the answer to q_5.

We extend our materialized view language to let it model views allowing to directly answer an XQuery query such as q_5. Two orthogonal extensions are needed. First, we introduce *optional* edges, denoting child (or descendant) nodes which are optional with respect to the parent (respectively, ancestor). Such edges are graphically depicted with dashed lines. Second, we distinguish *nested* edges, indicating that the data corresponding to the view subtree rooted at the child (or descendant) node should be stored in groups, one for each corresponding parent (respectively, ancestor) nodes. In the graphical view representation, nested edges are marked with the letter n. As an example, Figure 2.4 shows a materialized view based on which q_5 can be answered by surrounding paragraph groups with <res> tags into the output. Intuitively, the optional and nested edges allow pushing the structural outer join and the subsequent group-by operation directly in the view language.

For illustration, consider the sample document in Figure 2.5, where nodes labeled i1, i2 etc. stand for item elements, nodes labeled d1, d2 for description elements and so on, and assume the keyword elements labeled k4, k51 and k61 in this figure have the text value "gold". The data stored by the view in Figure 2.4 from this document is the following nested relation:

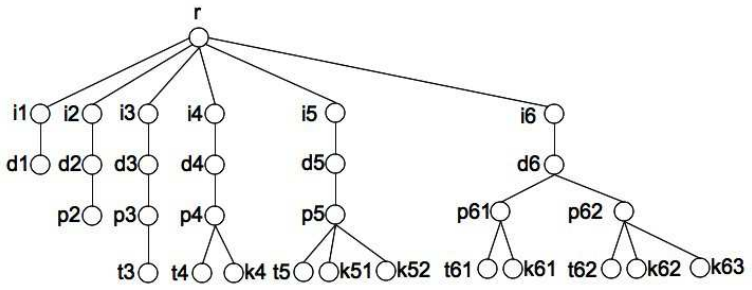


Figure 2.5: Sample XML document.

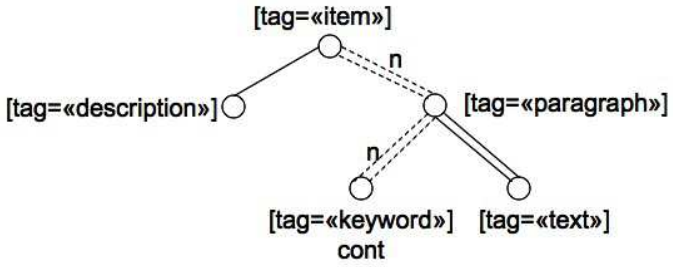


Figure 2.6: Views corresponding to the XQuery query q_6.

$$\{(\{(\text{p4.cont})\}),\ (\{(\text{p5.cont})\}),\ (\{(\text{p61.cont}), (\text{p62.cont})\})\}$$

where $\{\cdot\}$ is the set constructor and $(\cdot)$ is the tuple constructor. In this relation, there is a tuple for each of i4, i5 and i6, with a single attribute of type relation. This relation stores the cont attributes of the paragraphs corresponding to the items.

Multi-level nesting The query q_5 exhibits a single level of for-where-return expressions. Complex XQueries, however, can include multiple such blocks, that can be nested, such as the following query q_6:

```
q6  for    $x in //item[description]
    return <res1>
             {for $y in $x//paragraph[//text]
              return <res2>
                       {$y//keyword}
                       </res2>
             }
           </res1>
```

The XAM tree pattern formalism is able to capture such nesting, as illustrated by the view in Figure 2.6. Observe the two nested, optional edges: they correspond to the relationships between each return clause and

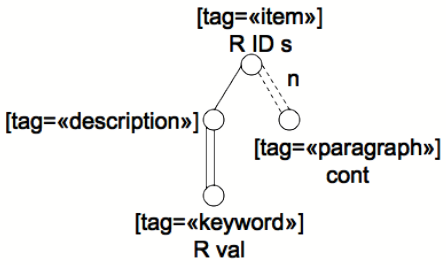


Figure 2.7: Index tree pattern.

its associated for-where block. The evaluation of this view on the sample document in Figure 2.5 can be traced as follows:

- for i1, the tuple $(\{\})$ containing an empty relation, since i1 does not have paragraphs;
- for i2, a similar tuple $(\{\})$, since the only paragraph descendant of i2 does not have text descendants;
- for i3, the tuple $(\{(\{\})\})$ containing a relation with one tuple for the paragraph p3. This inner tuple contains an empty relation, since p3 has no keyword descendants;
- for i4, the tuple $(\{(\{(\text{k4.cont})\})\})$;
- for i5, the tuple $(\{(\{(\text{k51.cont}), (\text{k52.cont})\})\})$;
- for i6, the tuple $(\{(\{(\text{k61.cont})\}), (\{(\text{k62.cont}), (\text{k63.cont})\})\})$.

When rewriting q_6 based on the view in Figure 2.6, the empty tuples for i1 and i2 lead to an empty res1 element for i1, and another empty one for i2. For i3, a res1 element containing an empty res2 element will be constructed, and so on.

XAMs versus query languages The relationship between XAMs and standard query languages can now be characterized as follows. XAMs are designed to select and extract data from an XML database. Compared with a rich XML query language such as XQuery, XAMs lack the ability to join, restructure (change the way XML elements are nested in each other), or create new XML elements. Such features were kept out in order to focus XAMs on the data that is needed by a query from the database, and not on the processing that the query applies on this data. However, XAMs capture aspects related to database system implementation, such as persistent identifiers and their special properties. Such identifiers are not part of the XML data model [1] and as a consequence, are not visible to high-level XML query languages.

[1]The xml:id specification of the W3C [105] describes a syntactic convention for encoding in a specific XML attribute the identifier of each XML element. However, this specification is not widely adopted and persistent stores need to handle XML elements with or without such attributes.

From views to indices Materialized views and indices are the two main classes of access support data structures used in the literature. The difference is that the data from a materialized view can be accessed by a view scan, whereas an index is accessed with some values for the index key attribute(s). In the context of XML processing, indices are also a valuable tool. Therefore, we make the following extension to the XAM tree pattern language, to enable the modeling of indices. A subset of the attributes stored by the XAM can be annotated with the R symbol, denoting that values for those attributes are *required* (bindings for those attributes must be supplied) in order to gain access to the corresponding tuples stored in the index. An example is provided in Figure 2.7. To access the data stored in this index, one must provide a value for the structural identifier of the item element and the value of one of its keyword descendants. For a given pair of (item.ID, keyword.val) values, the index returns the serialized image (cont) of all the paragraph descendents of the item.

Formal XAM semantics We have formally defined XAM semantics based on two distinct formalisms. In [22], we use a nested relational algebra and define the semantics of a XAM by an algebraic expression which recurses over the tree XAM structure. Intuitively, the algebraic expression of a XAM is a join tree, featuring one (possibly nested, possibly outer-) structural join for each XAM edge. In [22] we also formalize the concept of accessing a (potentially nested) XAM-described index with a tuple of bindings.

Second, in [23] we have defined XAM semantics based on the more traditional concept of tree embeddings. This formalism proved more convenient for reasoning about XAM containment.

2.2 XQuery rewriting based on XAMs

The previous section has illustrated the features of the XAM language for describing XML materialized views. We now turn to considering the usage of XAM-described views to rewrite XQuery queries. Section 2.2.1 provides a high-level view of our approach. Section 2.2.2 discusses XAM pattern extraction from XQuery queries. In Section 2.2.3 we consider the problem of rewriting XAM queries with XAM views, and discuss the algorithms we developed to solve this problem.

2.2.1 Overview of our approach

The core idea behind our rewriting approach is that the computations expressed by an XQuery can be classified in two rough categories. The first concerns the *data needs* of the query, i.e. the specification of which parts of the document must be read in order to answer the query. The second concerns the *extra processing* applied on the results of the first step. View support for XQuery processing, in this context, means the ability to efficiently support the query data needs based on the existing materialized views, with the understanding that the extra processing will be applied on the data retrieved from the views (thus, typically, on a dataset much smaller than the original data).

Our approach can be best followed based on Figure 2.8. From an initial XQuery query, a set of query tree patterns (or query XAMs) are extracted. In particular, due to the ability of XAMs to represent nesting and optional edges, a single tree pattern may cover the for, where and return part of a for-where-return block, and may even cover several such blocks, nested into one another. The resulting tree patterns, denoted q_1, q_2, q_3 in Figure 2.8, can be seen as encapsulating the data needs of the query. The algebraic operators connecting them represent the extra processing, that is, the other steps that the query applies, which were not already computed in the query tree patterns.

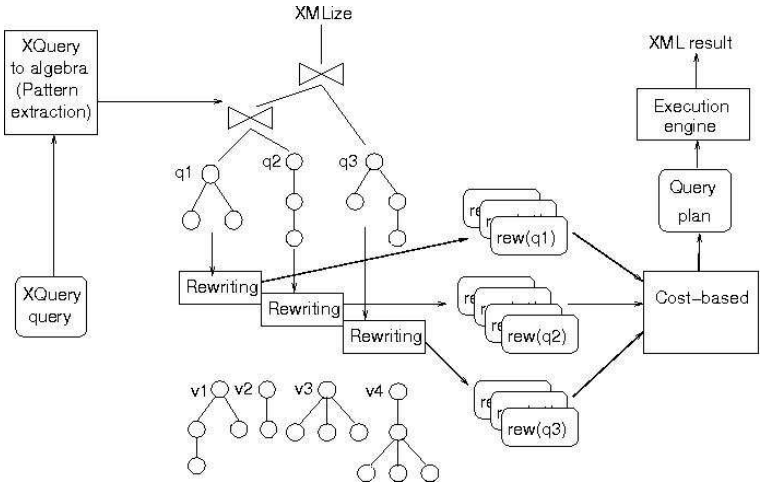


Figure 2.8: Outline of our XQuery processing approach.

Subsequently, each of the query patterns is rewritten based on the available views, denoted v_1, v_2, v_3 and v_4 in the Figure. Several rewritings can be obtained for each query pattern. Picking a combination of rewritings, one for each query pattern, and substituting them to the respective patterns in the initial algebraic plans leads to obtaining a logical plan which is a rewriting of the query. A cost-based optimizer can then be used to pick the best physical rewriting plan, whose execution will yield the desired query results.

This architecture, and all the algorithms discussed in the remainder of this chapter, have been validated by a complete implementation in the ULoad prototype, demonstrated in [25]. In particular, ULoad is able to exploit materialized views stored either in a relational database, or in a native store that we developed, using its own nested-relations execution engine.

The next section discusses XAM pattern extraction from XQuery queries. In Section 2.2.3 we consider the problem of rewriting XAM queries with XAM views, and discuss the algorithms we developed to solve this problem.

2.2.2 XAM extraction from conjunctive XQuery

There may be many ways of extracting tree patterns from an XQuery query. However, in order to efficiently exploit materialized views, one needs to extract as few (and as large) tree patterns as possible. This is because every query pattern will be rewritten as a standalone algebraic plan. The more these plans, the more operations (typically, joins) required to reconnect them into a complete query rewriting, and thus, very likely, the higher the cost.

In this section, we start by formalizing this intuition based on the notion of XAM-driven query decomposition. Meaningful reasoning about query semantics requires a clear specification the subset of XQuery we consider, given that the full language is Turing-complete [59]. Then, we outline our XAM extraction algorithm. The full description can be found in [24].

XAM-driven decomposition Our goal is to extract *maximal* XAM patterns from a given query, in the following sense. Let q be a query and $alg(q)$ a representation of q in an XQuery algebra such as the one

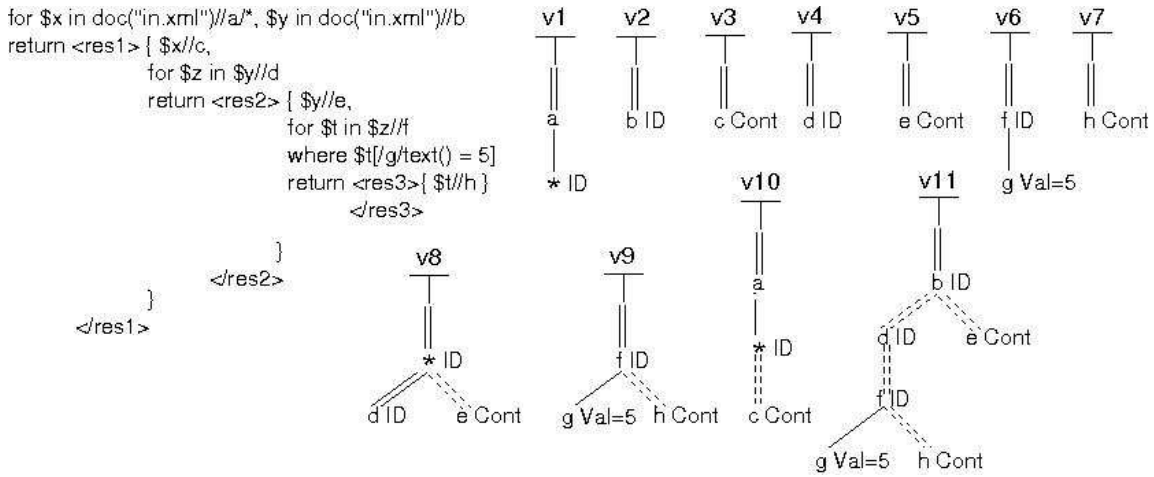


Figure 2.9: Sample XQuery and corresponding tree patterns.

described in [65]. A *XAM-driven decomposition* (or decomposition, in short) of q is an algebraic expression, equivalent to $alg(q)$, and of the form $d = e(tp_1, tp_2, \ldots, tp_n)$, such that each sub-expression tp_i, where $1 \leq i \leq n$ corresponds to a XAM. For one given query q, several decompositions can be identified. A decomposition d *dominates* $d' = e'(tp'_1, tp'_2, \ldots, tp'_m)$ if there do not exist $1 \leq i \leq n, 1 \leq j \leq m$ such that tp_i is a sub-expression of tp'_j. A decomposition d is *maximal* if it dominates all other decompositions of q.

For example, consider the XQuery query in Figure 2.9 and the XAMs depicted at its right. A first decomposition of this query, which we denote d_1, is a join expression over the tree patterns identified by v_1 to v_7 in the Figure. Each tree pattern represents one or two nodes appearing in the query's path expressions, and (structural) join operations are needed to combine then following the query structure.

A second decomposition, denoted d_2, identifies three tree pattern expressions denoted v_8, v_9 and v_{10} in the Figure. Here, v_{10} corresponds to the variable $x and its descendant $c node, v_8 corresponds to $y and its $d and $e descendants, while v_9 corresponds to the nodes named e, f, g and h in the innermost XQuery block. Clearly, d_2 dominates d_1, since, for instance, v_1 and v_3 are sub-expressions of v_{10}.

Finally, a third decomposition, denoted d_3, can be identified. It consists of two tree patterns, namely v_{10} and v_{11} in Figure 2.9. The decomposition d_3 dominates d_2 and d_1.

XQuery dialect We consider a subset of XQuery, denoted $\mathcal{Q}$, obtained as follows.

1. XPath$^{\{/,//,*,[]\}} \subset \mathcal{Q}$, that is, any core XPath [69] query over some document d is in $\mathcal{Q}$. We allow in such expressions the usage of the function *text()*, which on our data model returns the value of the node it is applied on. This represents a subset of XPath's absolute path expressions, whose navigation starts from the document root. Examples include $/a/b$ or $//c[//d/text() = 5]/e$. Navigation branches enclosed in [ ] may include complex paths and comparisons between a node and a constant c. Predicates connecting two nodes are not allowed. They may be expressed in XQuery for-where syntax (see below).

2. Let $\$x$ be a variable bound in the query context [110] to a list of XML nodes, and p be a core XPath expression. Then, $\$x\, p$ belongs to $\mathcal{Q}$, and represents the path expression p applied with $\$x$'s bindings

list as initial context list. For instance, $\$x/a[c]$ returns the a children of $\$x$ bindings having a c child, while $\$x//b$ returns the b descendents of $\$x$ bindings. This class captures *relative* XPath expressions in the case where the context list is obtained from some variable bindings. We denote the set of expressions (1) and (2) above as $\mathcal{P}$, the set of path expressions.

3. For any two expressions e_1 and $e_2 \in \mathcal{Q}$, their concatenation, denoted e_1, e_2, also belongs to $\mathcal{Q}$.
4. If t is an element name and $exp \in \mathcal{Q}$, element constructors of the form $\langle t \rangle \{exp\} \langle /t \rangle$ belong to $\mathcal{Q}$.
5. All expressions of the following form belong to $\mathcal{Q}$:

$$\boxed{xq} \quad \begin{array}{l} \text{for } \$x_1 \text{ in } p_1, \$x_2 \text{ in } p_2, \ldots, \$x_k \text{ in } p_k \\ \text{where } p_{k+1}\ \theta_1\ p_{k+2} \text{ and } \ldots \text{ and } p_{m-1}\ \theta_l\ p_m \\ \text{return } q(x_1, x_2, \ldots, x_k) \end{array}$$

where $p_1, p_2, \ldots, p_k, p_{k+1}, \ldots, p_m \in \mathcal{P}$, any p_i starts either from the root of some document d, or from a variable x_l introduced in the query before p_i, $\theta_1, \ldots, \theta_l$ are some comparators, and $q(x_1, \ldots, x_k) \in \mathcal{Q}$. Note that the return clause of a query may contain several other for-where-return queries, nested and/or concatenated and/or grouped inside constructed elements. The query in Figure 2.9 illustrates our supported fragment.

XAM extraction algorithm For the XQuery dialect $\mathcal{Q}$ identified above, a maximal decomposition exists, and it is unique. To obtain it, we proceed in two steps:

- We have provided an algorithm which starting from a query q, provides its algebraic translation $alg(q)$. In the algebraic expressions, all leaf operators correspond to simple patterns of exactly one node.
- Subsequently, equivalence-preserving algebraic rewriting is performed, in order to group together those one-node patterns corresponding to nodes that are structurally related, according to the query. This step is a particular type of join re-ordering.

With respect to the algebraic translation, we mention that each $\mathcal{P}$ expression is translated, as expected, into an expression that is equivalent to a XAM tree pattern. The other main query primitive is the for-where-return expression. In this case, a nested structural outer join is used to combine the expression in the for-where clauses (outer) with the expression in the return clause (inner). The algorithm is described in [24].

2.2.3 XAM query rewriting

In this section, we consider the problem of rewriting a XAM tree pattern query, based on a set of XAM tree pattern materialized views. We start by illustrating multiple-views XAM rewriting via an example. We then spell out our rewriting approach.

XAM rewriting by example Figure 2.10 depicts one XAM query q and three XAM views v_1, v_2 and v_3. At right in the Figure, we have shown a rewriting of q based on the views.

Recall from the global description of our approach (Section 2.2.1) that a rewriting is an *algebraic plan*. Our algebra includes the regular selection (σ) and projection (π) operators, as well as cartesian products ($\times$), joins ($\bowtie$) and outerjoins ($\bowtie\!\!\!\!\sqsubset$). We consider conjunctive join predicates, such that each atom can express: an equality among attributes and constants, or between two attributes; or a structural predicate between two attributes which correspond to node IDs. More specifically, a predicate of the form $a.id \prec b.id$ denotes

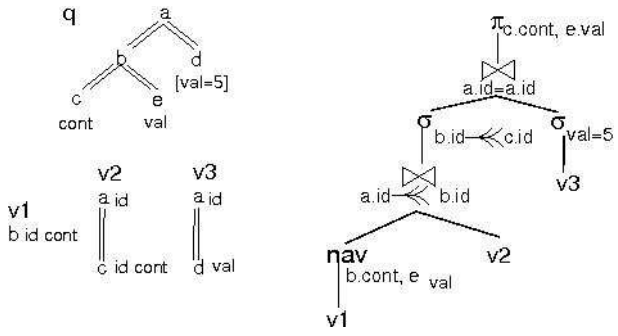


Figure 2.10: Sample XAM rewriting.

that the nodes whose identifiers can be found in the attribute $a.id$ correspond to parents of nodes whose identifiers can be found in the attribute $b.id$. The symbol $\prec\!\!\!\prec$ denotes the ancestor-descendant relationship, based on which atomic predicates can be similarly defined. We also consider a navigation operator, denoted $nav_{i,np}(np)$, which evaluates the pattern np over the i attribute of op. Two restrictions apply. First, i must be a $cont$ attribute, as navigation only makes sense inside an XML fragment. Second, np nodes must not project id attributes, since it is generally not possible to determine the structural identifier corresponding to a node by only considering an XML fragment enclosing the node. In its full generality, our algebra is based on nested relations, and therefore includes a *nested apply* operator, based on which complex plans can be applied on nested tables. Such nesting will be ignored for simplicity in the following discussion.

We are now equipped to explain the rewriting from Figure 2.10. View v_1 matches the b node of the query. The sub-plan $nav_{b.cont,e.val}(v_1)$ adds a new attribute, obtained by extracting the values of the e descendants of each b element. The join on the predicate $a.id \prec\!\!\!\prec b.id$ ensures that the a nodes retrieved from view v_2 are ancestors of the b nodes from v_1. View v_2 also brings matches for the c query node, therefore the selection $\sigma_{b.id \prec\!\!\!\prec c.id}$ is added in order to ensure that these c nodes are also descendants of the b nodes from v_1. The join on the predicate $a.id = a.id$ matches a nodes from v_1, with a nodes from v_3. The left-hand child of this join contains matches for the a, b, c and e query nodes, whereas the right-hand child corresponds to the a and d query nodes. The final projection retains only the $c.cont$ and $e.val$ attributes that the query required.

The rewriting problem Given a query q and a set of views $\mathcal{V} = v_1, v_2, \ldots, v_n$, a first approximation of the rewriting problem may be stated as: find all algebraic plans which are equivalent to q, that is, those plans that will return the same results as q for any input database. This definition, however, leads to an artifficial explosion of the target rewritings, since many algebraic plans differ e.g. by their join order, selection and projection placement etc. while fundamentally they are the same. An optimizer will always have the possibility to explore as many of these join orders as desired after rewriting, but this solution explosion must be avoided when rewriting. To this effect, we defined a *canonical* form of algebraic expressions, such that all rewritings differing only by their join order, selection and projection placement have the same canonical form [66]. We then re-state our rewriting problem as: find all canonical algebraic plans based on $\mathcal{V}$ views.

A final refinement is to check for the presence of redundancy in the rewritings, i.e., a view symbol which could be removed from the algebraic rewriting expression while the result is still an equivalent rewriting. Thus, our final problem statement is:

Given a set of XAM views $\mathcal{V}$ and a XAM query q,
find all minimal canonical equivalent algebraic rewritings of q using views from $\mathcal{V}$.

We now consider the complexity of the rewriting problem. Given a query rewriting, one can clearly derive a cover of all the query nodes with the sets of view nodes. However, not any cover of query nodes with view nodes can be translated translate into a rewriting. This is due to some view nodes which lack IDs, and thus disallow a join that would have allowed building a rewriting.

An important complexity aspect is the following. We have shown in [66] that the number of views involved in a minimal canonical rewriting is smaller than, or equal to, the number of nodes in the query. This places the complexity the rewriting problem in $O(|\mathcal{V}|)^{|q|}$, where $|\mathcal{V}|$ denotes the number of views in $\mathcal{V}$ and $|q|$ denotes the number of query nodes. A different bound on the rewriting size holds in the particular case when rewriting is performed under structural constraints on the XML database [23].

Rewriting algorithms Based on this analysis, we have first considered two rewriting algorithms based on computing covers of the query nodes with the view node sets.

Subset Enumeration (SE) The SE algorithm enumerates all subsets of $\mathcal{V}$ and attempts to build a rewriting out of each subtree. If a rewriting is found, it is tested for minimality.

Increasing Subset Enumeration (ISE) The ISE algorithm is similar to SE, but it differs from it by the $\mathcal{V}$ subset enumeration strategy it uses. More specifically, ISE enumerates subsets in the increasing order of their size. This allows minimal rewritings to be developed before non-minimal ones.

SE and ISE share two disadvantages. First, they repeat work. For instance, let v_1 and v_2 be two views from $\mathcal{V}$. The effort to see if they can be joined will be repeated by SE and ISE, for all $\mathcal{V}$ subsets which contain both v_1 or v_2. It would be more efficient to memoize the partial result obtained by joining v_1 and v_2 (if this join was possible), or memoize the failure to join them otherwise.

Second, SE and ISE blindly test all subsets, whereas we only need to test those view subsets covering *overlapping or related* sets of query nodes. For instance, consider the query $q = //a[//b]//c$ and three views $v_a = //a$, $v_b = //b$ and $v_c = //c$ (assume all query and view nodes are labeled id). One may avoid testing the subset $\{v_b, v_c\}$, and yet identify all minimal canonical rewritings. This is because no equivalent rewriting of a tree pattern query contains a pure cartesian product of two or more views. In other words, assume we represent a rewriting as a graph, having one node for each view and one edge for each join predicate connecting two views. In order for the rewriting to be equivalent to the query, the corresponding graph must be connected. This allows avoiding the enumeration of the $\{v_b, v_c\}$ set in our example.

To alleviate the shortcomings of SE and ISE, we have devised two incremental, bottom-up rewriting algorithms, **Dynamic Programming Rewriting** (or DPR, in short), and **Depth-First Rewriting** (or DFR, in short). The algorithms share several traits:

1. Their first step is to *prune* the set of views by eliminating the views of which it can be established that they cannot be used to answer the query. The pruning is made based on the following criterion: a view v may be used to answer a query q only if $q(d) \neq \emptyset \Rightarrow v(d) \neq \emptyset$ for any document d.

   On our pattern language, this can be checked in $O(|v| \times |q|)$ by evaluating v as a query over q. If the result is empty, then v can be discarded, since no equivalent rewriting of q will use it.

2. Their next step is to *extend* each view to account for all the nodes that it could return if some navigation is applied on some of its $cont$ attributes.

3. In a third step, both algorithms *combine views in left-deep join plans*, using ID equality joins or structural joins.

4. Selections and projections are added once a join plan is found to have sufficient information to be used as a rewriting.

The algorithms differ in the order in which they build their join plans (the third step described above).

The DPR algorithm attempts to join $n + 1$ views only after it has finished exploring partial rewritings of up to n views.

The DFR algorithm orders the search by the number of query nodes covered by a partial rewriting. Thus, at any moment, it tries to join the rewriting developed so far which covers the most query nodes, with a single view.

The DPR and DFR algorithms are *correct* and *complete* for our pattern language and notion of rewriting.

Correctness is ensured by manipulating throughout the rewriting process, (plan, pattern) pairs. Initially, in each pair, the pattern is a view v and the plan is $scan(v)$. After extension, the pattern may change by adding nodes to v, while the plan takes the form $nav_{...}(scan(v))$. For instance, in Figure 2.10, the pattern equivalent to the plan $nav_{b.cont,e.val}(v_1)$ has two nodes, one labeled b, and one underneath labeled e.

Completeness is ensured by the fact that the subsets which DPR and DFR do not explore, are those which include at least one view that is not connected to any other view by a join predicate (thus, subsets including a cartesian product among views).

The advantage of DFR over DPR is that it very often finds its first rewriting very fast, due to its greedy query covering approach. In contrast, DPR develops increasingly larger rewritings and typically finds solutions at the end of the search only. The trade-off is that the total running time of DFR is typically longer than for DPR. This is because DFR explores the search space in a disordered manner, always eager to build on the partial rewriting covering the most query nodes. Thus, to ensure completeness, DFR must re-visit some parts of the search space. Proper memoization prevents it from repeatedly building complex plans, but the extra effort required for this memoization is noticeable. Finally, DFR and DPR are much faster than ISE and ISE, since DFR and DPR avoid exploring certain view subsets as explained above. Experiments demonstrating these aspects are described in [66].

2.3 Related works

Our work on XAMs belongs to the area of choosing the optimal data access path to answer a given query. From this perspective, it can be seen as following the line of research initiated by the seminal System R paper [86] and the GMap framework [98]. More recently and closer to our work, query processing in XML databases has lead to results in two areas, which we survey next: tree pattern formalisms for representing XQuery queries, and access path selection in XML databases.

Tree patterns for XML query processing Trees are natural formalisms for expressing XML queries, and tree patterns have been used to represent many XPath dialects [69]. XPath queries can be captured by conjunctive, unnested tree patterns. In contrast, we consider nested XQuery queries, which require richer patterns. XQuery processing based on tree pattern models has been studied by means of Generalized Tree Patterns [39] and its follow-up work on Tree Logical Classes [76]. Similarly to XAMs, GTPs and TLCs provide for optional edges. They operate on a model with nodes, unlike our model based on nested tuples,

which allows them to capture the nesting of results within return clauses. However, a formal description of the expressive power of TLCs, and therefore of their target XQuery subset, is not available. In our work, we have formalized tree pattern extraction from XQuery in [24].

Access path selection in XML databases The intricacies of the XML data model (and its precursor, the semistructured data model such as materialized e.g. in OEM) make it unsuitable as a data storage model, bringing the need for complex alternative data access support structures. Among the earliest are value, label, and path indices [51]. Thus, a value index provides access to nodes having a specific value, a label index allows direct access to all nodes of a given label, while a path index allows to directly retrieve the identifiers of nodes found on a given path in the XML document. Other works focused on more complex XML path or graph indices [58, 79], providing access to the identifiers of nodes appearing in positions satisfying specific structural constraints.

XAMs share with these works the ability to describe data structures containing identifiers of nodes found on specific paths. They are tailored to a tree data model, and therefore do not capture the graph constraints of [58, 79]. However, XAM-described indices or views include persistent structural identifiers, which increase the set of scenarios where they can be used to rewrite queries.

Several earlier works [32, 43, 87] have studied alternative ways of storing XML in relational (or relational-style) views (or databases), also known as XML shredding. The focus of these works is on how to chose the relations in which to store a specific XML corpus, given the data, the schema, and/or the query workload. The resulting access path selection method is in each case tied to the specific chosen storage model, which XAMs generalize by introducing nesting, and extend by the usage of structural identifiers. More generic approaches for rewriting XML queries based on materialized views are taken in [44, 64], which, however, do not exploit nesting, nor structural identifiers.

Materialized views for XPath queries Once the first generation of XML databases endowed with XPath query processor was established, research works have considered using XPath materialized views when answering XPath queries. In such works, views store only XML fragments, therefore rewriting is reduced to navigation, and only one view can be used at a time for a specific query [30, 113].

View intersection is used to build rewritings in [38]; the ID equality joins we consider are akin to intersection. Our rewriting problem is complicated by the fact that our views have multiple attributes at various places in the view. Thus, we need joins, and we need to take into account how many times a tuple is multiplied by each extra join. As we explain in [66], this may lead some rewritings to fail. Also, we assume structural ids, which enable e.g. rewriting the query q in Figure 2.11 using the views v_1 and v_2 in the same Figure, whereas [38] does not handle this case.

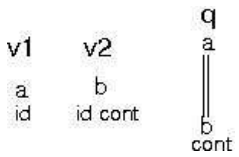


Figure 2.11: Sample views and query for comparing our work with [94].

The recent work of [94] takes structural ids a step further. They use XPath views (including wildcard nodes labeled $*$) where the return node always has *cont and* a powerful structural id, encapsulating the ids and labels of all its ancestors, up to the root [61]. Thus, unlike us and [38], they may rewrite query q in Figure 2.11 using only the view v_2: one only needs to check the $b.id$ for an a-labeled ancestor. We place

fewer conditions on the IDs we use, but our algorithms could be extended easily to handle more complex ones. Due to the presence of more powerful IDs, rewriting is reduced in [94] to finding covers of the query leaves. Our rewritings need to cover the whole query, however the complexity is better than the one derived from set-cover, since bounds can be established on the size of the rewriting, making the complexity of our rewriting problem polynomial. In contrast, in [94] the rewriting size bound is $|\mathcal{V}|$ and the complexity is exponential in the number of query leaves.

View pruning (step 1 in our DPR and DFR algorithms) is very expensive in the presence of wildcard nodes labeled $*$, thus [94] provides an efficient, approximate view pruning technique which builds views a view automaton at a cost of $\Sigma_{v \in \mathcal{V}}(|v|)$, and then runs q through the automaton. This optimization would also apply in our context, but its impact is considerably smaller, since in our pattern language, lacking wildcard nodes, pruning was not expensive to start with.

Materialized views for XQuery queries Few studies have considered this problem so far. XQuery rewriting based on XQuery views is studied in [74], which establishes polynomial complexity for the XPath case. A recent study addressed the problem of choosing SQL/XML views to support XQuery queries [45], whereas [95] considers materialized XML view selection based on the rewriting algorithm of [94].

In our own work, we have addressed the rewriting of queries expressed in the full XAM language using XAM views in [23], under Dataguide constraints which strongly impact the algorithm. This algorithm was implemented in ULoad [25], which was the system capable of XML query rewriting based on multiple views. The DFR algorithm was studied alongside with DPR, in the absence of constraints, for a subset of the XAM language, in [67] and its extended version [66]. This algorithm was implemented within the ViP2P peer-to-peer system for materialized XML view management [103], which we will discuss in Chapter 4.

2.4 Context of the work presented in this chapter

The work on XAMs was carried on within the framework of the Tralala (**Tra**nsformations, **L**ogic and **La**nguages for XML) grant (2005-2008), funded by the French National Research Agency (ANR). The work performed in this period has been at the core of the PhD thesis of Andrei Arion [21] and of several undergraduate and MS internships: Ravi Vijay (2005) and Dushyant Rajput (2008), both from IIT Bombay, India, Thai-Tho Nguyen (2006) from IFIPS Hanoi, Vietnam, and Asmae Fahoum (2006) from Ecole Mohammadia of Rabat, Morocco.

The more recent developments [66, 67] have been carried on within the CODEX (**E**fficiency, **D**ynamicity and **Co**mposition for **X**ML) grant [115] (2009-2012), funded by ANR, which I coordinate. This work is part of the PhD of Spyros Zoupanos, expected to graduate by the end of 2009.

Finally, the work presented in this chapter was also partially supported by the ANR "Young Investigator" grant WebStand (2006-2009), obtained jointly with Dario Colazzo from U. Paris XI, Benjamin Nguyen from U. Versailles and Antoine Vion, a social scientist, from U. Aix.

2.5 Conclusion

This chapter has presented our work on exploiting materialized views described by expressive tree patterns, to answer queries expressed in a subset of XQuery. Our work is based on precise formal semantics, which enabled us to provide algorithms for decomposing queries in maximal patterns, and for rewriting queries based on such pattern views. Interestingly, the rewriting problem is polynomial in the combined size of the

views, in the two variants of the problem that we considered: with and without the presence of structural constraints on the document, expressed by Dataguides.

Chapter 3

Optimization for Active XML

This chapter is placed in the context of the ActiveXML language for distributed XML applications. Section 3.1 provides an overview of the language and the platforms built for ActiveXML. Section 3.2 outlines two optimization techniques we studied in this context: exploiting distribution and replication, and lazily evaluating queries on AXML documents. Section 3.3 presents OptimAX, an approach and a prototype for the static optimization of AXML computations. Section 3.4 discusses related works, while Section 3.5 situates this research work with respect to grants, and to the PhD and intern student involvement.

3.1 Active XML overview

The Web has become the platform of choice for the delivery of business applications. In particular, the popularity of Web service technologies (WSDL [104], BPEL4WS [34] etc.) and their closeness to HTML and XML, the predominant content delivery languages on the Web, has opened the way to the development of complex business applications by integrating Web services provided by different parties. This model has several advantages. From a development viewpoint, it relies on widely accepted standards, and benefits from the plethora of available application building blocks. From a business viewpoint, it allows organizing the activity in cleanly defined modules, each of which is implemented by some Web services. This enables several entities to provide implementations of a given module, and facilitates replacing one entity with another.

The ActiveXML language (or AXML, in short) [29] has been studied since 2001 as a data-driven high-level Web service composition tool. An ActiveXML document is an XML document including calls to Web services. When the Web services are called, possibly with some XML parameters (or inputs), the corresponding Web service call results (or outputs) are inserted in the AXML document as siblings of the service call node. Thus, an AXML document can be seen as having an *extensional* part, i.e., those data nodes actually appearing in the document, and an *intensional* part, i.e., the nodes that are added to the document when service calls are activated.

AXML has been put to work in a peer-to-peer setting [6]. Each AXML peer may host a set of (A)XML documents and a set of Web services. Thus, activating a Web service call found in a document at one peer, acting like a client in this context, may trigger the evaluation of a service implemented by a query on a different peer, which acts like a server. We are particularly interested in *declarative* Web services, whose implementation is realized by evaluating a read-only XQuery query on some (A)XML documents.

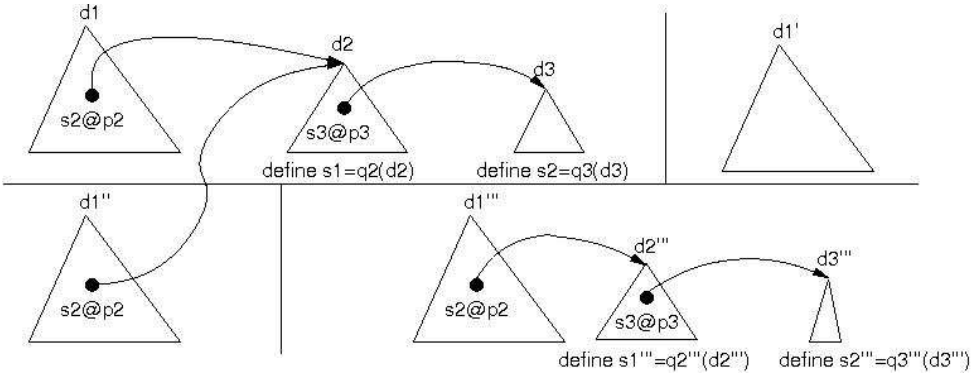


Figure 3.1: ActiveXML document replication options.

Declarative services in a distributed setting provide a series of optimization opportunities, which we set out to exploit.

3.2 Optimization techniques for ActiveXML

In this section, we outline two interesting optimization problems arising in applications based on ActiveXML documents.

Distribution and replication of AXML documents In a distributed setting, it is common for performance reasons to copy, either partially or completely, a data item from one site to another. In our case, a peer may be willing to establish a local cache of *some* AXML documents, in order to speed up the most frequent operations at that peer. In a context where documents are partly intensional, a whole array of replication possibilities open up, as illustrated in Figure 3.1. Here, the document d_1 contains among others a call to the service s_2, implemented at p_2 by a query q_2 over document d_2. A similar connection holds between document d_2 and the document d_3.

1. A very superficial replication would consist of copying only the extensional document nodes, including Web service call results, but excluding Web service call themselves (document d_1' in Figure 3.1. This corresponds to a snapshot of the document's actual contents at a given moment in time. However, such a snapshot, unlike the original document, does not allow replacing the Web service call results with future, potentially changed values.

2. Alternatively, one may copy the document as such, including the Web service calls (document d_1'' in Figure 3.1) This allows the replica owner to make it evolve by performing the Web service calls of the original document. However, if the replica owner is not able to call the service (e.g., being off-line), the calls cannot be performed.

3. Finally, one can envision copying the document with service calls and the implementations (the definitions) of the respective services. This is possible when the service code can be migrated, and in particular when services are defined by declarative XML queries. In this case, observe that if one such query carries over another document, a portion of that document should also be replicated to enable

the replicated service to function. The replicated service code, moreover, may have to be adjusted to return the same result on the partial replica. This is illustrated by the documents d_1''', d_2''' and d_3''' in Figure 3.1. Document d_1''' is a full replica of d_1, but d_2''' is only a partial replica of d_2. We replicate only as much as needed for the service q_2 to function. The copy of q_2's code adapted to the replicated d_2''' is denoted q_2'''. A similar mechanism produces the document d_3''' and the replicated service defined by q_3'''.

In this context, we have, first, proposed an algorithm for evaluating queries over distributed documents. The algorithm computes from the original query, the largest sub-query that can be evaluated on the original document (excluding the remotely distributed elements), and evaluates it using a local XML query processor. This query results in the list of nodes in the original document d, such that some remote children of d must be traversed to evaluate the remainder of the query. This part of the query is then sent to the remote site, which proceeds in a similar fashion, until the query has been completely evaluated and results have been gathered. The interest of the algorithm is that it exploits the capability of standard XML query processors, that is, unaware of distribution and replication.

Second, we have devised an algorithm for replicating intensional data, with the purpose of making a replicated fragment independent, i.e., allow it to function even if it has no connection left with any other document or service provider. A typical application concerns replicating a workspace of data and services to be used on a mobile terminal with little or no connectivity.[1] The algorithm proceeds recursively starting with a document, and then whenever a call to a declarative Web service is encountered, attempts to replicate the service, as well as the documents it depends on.

The above algorithms are described and evaluated in [7]. Their corresponding implementations have been demonstrated in [4].

Lazy query evaluation on ActiveXML documents The full answer of a query on an ActiveXML document can be defined as the answer obtained by first, evaluating the document until a fixpoint is reached, by triggering all existing Web service calls, and then, evaluating the query on the fixpoint thus obtained. However, it may be the case that some Web service calls in the document do not affect the query result at all, therefore, they need not be triggered just in order to compute the result of the query. We say such service calls are not *relevant* to the query.

Interestingly, the relevance of a Web service call included in a document, to a given query asked over the document, is not fixed once and for all. Instead, when one service call is triggered and produces results, this may make other, previously irrelevant service calls become relevant.

Moreover, rather than retrieving the results of some Web service calls and then evaluating (sub-)queries over these results, one could envision pushing the evaluation of the sub-queries to the Web service providers, so that only the needed query results are obtained at the site where the query has originated.

We have studied these issues in [5]. More specifically:

- We provided an algorithm which, given a document and a query, identifies the service calls in the document that are relevant for the query.
- Based on this, we proposed a simple algorithm for evaluating queries over intensional documents. The algorithm triggers a call to a relevant service, then re-computes the set of relevant service calls in the document, then triggers another call etc. We have proposed an optimization which reduces the total

[1] If we exclude Web services, this is the problem solved by Web browsers allowing users to work offline, or iPhone applications building local copies of a given Web site etc.

amount of work to be done, by analyzing dependencies among Web service calls, as well efficient implementation techniques.

- We have provided an algorithm for decomposing queries into a local part, and sub-queries to be pushed to the Web service providers.

All the above algorithms were implemented and experimentally assessed in [5]. Our study shows that due to the relatively high overhead of making a Web service invocation, pruning down the set of Web service calls activations entailed by the evaluation of a query can significantly reduce the query evaluation time.

3.3 OptimAX: a framework for static ActiveXML optimization

In this section, we describe a more general approach we took towards optimizing the performance of ActiveXML applications. The approach we describe is *static* in that it consists of applying a set of transformations to one or several ActiveXML documents without triggering any of Web service calls contained in the document. Rather, the idea we exploited in this work [13, 15] is to *rewrite* an ActiveXML document into another one, such that evaluating all service calls in the two documents lead to producing the same results, but the evaluation of the rewritten document is more efficient.

3.3.1 ActiveXML with continuous services

We consider ActiveXML documents including calls to *continuous services*, whose inputs are streams of XML trees, and which may produce output before having exhausted any of their inputs. A particular class of continuous service have no XML inputs and emit a stream of XML trees. This corresponds to an XML subscription, in the style of RSS.

Let s be a service with n inputs. When the service is running, it expects to receive a stream of XML trees for each input. Any stream finishes with a special token denoted *eof*, which signals that no tree may follow. Trees may arrive in all inputs in parallel. When a tree is received in one input, the service may perform an internal computation and/or may output zero or more trees. More specifically, we consider the class of *distributive* services, such that for each $1 \leq i \leq n$, and for any finite streams of XML trees $T_1, \ldots, T'_i, T''_i, \ldots, T_n$, the following holds:

$$s(T_1, \ldots, (T'_i + T''_i), \ldots, T_n) = s(T_1, \ldots, T'_i, \ldots, T_n) + s(T_1, \ldots, T''_i, \ldots, T_n)$$

where $+$ stands for stream concatenation. Services defined by XPath queries and for-where-return XQuery expression can be seen as distributive, if all input tree as disjoints. If this is not guaranteed, then one can refine the $+$ semantics to include the necessary duplicate elimination. We will not discuss this issue further.

As an example, consider a query service defined by the following query:

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

A possible sequence of inputs and outputs for this service is shown in Figure 3.2. Observe that an output is produced only when equal-valued a elements have been found in the inputs. Services such as this one

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

Figure 3.2: Sample execution scenario of a continuous service defined by a query.

require the maintenance of an internal state for each invocation (both inputs need to be buffered in order to find matching pairs). To avoid dealing with unbounded-state services, in practice, time- or size-determined windows are customarily applied on the input streams. This issue is rather orthogonal to the optimization questions we consider and thus, we do not address it further.

A non-continuous service may be seen as a particular case of a continuous one, delaying output until it has received an eof token from each of its inputs. At this point, the service outputs its complete results followed by eof.

3.3.2 ActiveXML activation orders and schedules

As outlined above, the AXML language does not support specifying when to activate a service call. The issue is of importance even when considering deterministic, time-independent services, i.e. those that return the same output for the same inputs, regardless of the activation moment. Activation moments may make a difference when considering the interplay between several service calls in the same document. For instance, consider a service call $sc_2(sc_1)$, i.e. sc_1 is a parameter of sc_2. The user may choose to activate just sc_2, in this case the sc_1 element *as such* is used as a parameter for sc_2. The call sc_1 may be activated in the future. Or, the user may chose to activate just sc_2, in this case sc_2 is evaluated and its result accumulate as children of sc_1, awaiting its possible future activation.

For our optimization purposes, we introduced a simple, yet flexible approach for deciding when to activate calls. This approach is based on a set of *default activation order* constraints, which apply automatically, and lead to obtaining an intuitive result which the users may expect. For more advanced scenarios, *explicit activation order* constraints may be defined by the application designer. Explicit constraints have higher priority, that is, they override the defaults.

The first default activation order rule is dao_1 in Figure 3.3. The reason for this rule is that the majority of the services available today requires plain XML inputs and returns plain XML outputs. Activating the inner call first is more likely to lead to call sc_2 with XML input. Rule dao_1 cannot influence the activation order of two calls when none is an ancestor of the other. To capture such constraints, we enable users to specify that a given service call should be activated only after another call's activation. Moreover, for continuous services, we may wish to distinguish between activating service call sc_1 after sc_2 has been *activated* (but has not finished executing), and activating sc_1 after sc_2 has been activated and has finished, i.e. it has sent its eof. Syntactically, such constraints are expressed using two attributes afterActivated and afterTerminated, whose interpretation is provided by the rules eao_1 and eao_2 in Figure 3.3.

For example, Figure 3.4 depicts a simple AXML document at peer p_0, and a possible timeline of the activations of its calls. The period between the activation and termination of each service is shown by a horizontal bar. The calls to $h@p_2$, $k@p_3$, and $g@p_4$ are activated first, since they have no implicit or explicit dependencies on other calls. The call to $j@p_2$ is activated sometimes after the call to $g@p_4$. Finally, the call

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

Figure 3.3: Activation order rules.

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

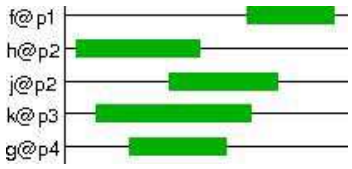


Figure 3.4: Sample AXML document and timeline for its service call activation.

to $f@p_1$ is activated after the call to j has finished, as required by the explicit order constraints.

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

Figure 3.5: Sample activation of calls to $send$ and $receive$.

Send and receive services For optimization purposes, to the basic AXML model above, we add a small set of predefined services, which we assume available on all peers.

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

One of the main applications of $send$ and $receive$ concerns the sending of data streams, as Figure 3.5 illustrates. Consider for now only the XML content shown in upright part of the table. The document $doc_1@p_1$ contains a call to the local service $send@p_1$, with a call to $f@p_3$ as parameter. The destination address is the node identified by $(p_2, doc_2, \#3)$, which is the call to $receive$. Once the call to $f@p_3$ is activated, it returns $fres$ elements shown in italic font in $doc_1.xml$ in Figure 3.5. Activating the calls to $send$ and $receive$ transmits these elements into the document $doc_2@p_2$. In the figure, the last element has not yet arrived in $doc_2.xml$.

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

The introduction of the $send$ and $receive$ services requires new activation order rules, namely dao_2, dao_3 and noa_1 in Table 3.3. Rule dao_2 specifies that by default, $send$ distributes the computation (not its result). Rule dao_3 shows that $receive$ calls are not activated individually but only as a consequence of receiving a message. Finally, the *no* activation rule noa_1 states that if a service call sc is sent from p to p' by a $send$, sc is not be evaluated at p.

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

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

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

Activation schedule (revisited) Let d be a document and $SC(D) = \{sc_1, sc_2, \ldots, sc_k\}$ be the set of the service calls from d. An activation schedule (or schedule, in short) for S is a list of pairs:

$$[(sc_1, \tau_1), (sc_2, \tau_2), \ldots, (sc_k, \tau_k)]$$

such that for $1 \leq i \leq k$, τ_k is a moment in time, sc_1 is activated at the moment τ_1, sc_2 is activated at the moment τ_2 etc. The schedule is said *valid* iff it respects: (i) all the eao_1 and eao_2 constraints of d, and (ii) as many of the $dao_1 - dao_4$ constraints as possible without violating the (eao_1) and (eao_2) constraints.

Note that valid schedules largely allow parallel activation of continuous services (the only limitation being the explicit use of afterTerminated). Figure 3.4 exemplifies a valid schedule (bottom) for a given ActiveXML document (top). We focus on valid schedules from now on.

3.3.3 AXML optimization problems

We now introduce the fundamental notion of schedule equivalence, as well as a cost model. Based on these, we define our optimization problem.

Equivalent schedules Let d be an AXML document and T_1, T_2 be two schedules over two sets of services $S_1, S_2 \subseteq SC(d)$. We say the schedules are equivalent, denoted $T_1 \equiv T_2$, iff applying T_1, resp. T_2 on the document leads to documents that are equal. Observe that S_1 and S_2 do not have to coincide.

Valid schedule equivalence Let d be an AXML document and $S \subseteq SC(d)$ be a subset of d's service calls. All valid schedules for S are equivalent. This is due to the distributive and deterministic character of the services which, called with the same parameters, produce the same results, even if some streams are created at different moments and progress at different rates in different schedules.

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

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

where: α and β are some numerical weights, c_f is the cost associated to the computation of f at the peer p, $bw_{p \to p_0}$ and $b_{p_0 \to p}$, respectively, are the bandwidths from p to p_0, respectively from p_0 to p, s_p is the size of the parameters of the calls to $f@p$, and s_f is the size of the results produced by the calls to $f@p$.

We focus on activations with a finite cost, which requires that c_f, s_p and s_f be finite. The latter implies that f returns a finite number of answers. (A simple extensions to infinite streams would consider the cost per tree in the stream.)

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

Using the above cost model, it can be shown that all valid schedules for a given subset $S \subseteq SQ(d)$, besides being equivalent, have the same cost.

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

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

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

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

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

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

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

We now consider the problem of *AXML cost-based optimization*, focusing first on one-stage.

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

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

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

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

Document equivalence Let $d@p$, $d'@p$ be two documents at the same peer. We say d and d' are equivalent, denoted $d \equiv d'$, if the result of full activation on d and d' coincide (up to some terminated service calls).

This notion of equivalence characterizes documents that are *eventually* equal after their full activation. The documents may go through different states during the process, may call services from different peers etc. From the perspective of the user requiring the full activation result of $d@p$, the result of $d'@p$ is the same. From the system perspective, given a document d and a set $\mathcal{R}$ of rewriting rules, optimization can be seen as repeatedly applying $\mathcal{R}$ rules to obtain new documents equivalent to d, and keeping the one with the lowest evaluation cost.

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

- pick a service call currently in d which can be activated according to the ordering constraints of d, and activate it
- apply a rewriting rule from $\mathcal{R}$, rewriting d into d'

until all services calls in d have been activated, such that the total activation cost (including the past and possibly future service call activations) plus the total cost of optimization is the smallest among all possible such sequences of steps.

In the above, we also took into account the cost of optimization, which can be approximated by some constant c_o. Observe that the problem is more complex than just inserting optimization steps in some places into a given full schedule, because optimization may remove or add service calls, leading to re-scheduling. We make the following important remarks:

1. If all service calls return plain XML results, then invoking the optimizer only once, prior to any activation, is a solution to the full optimization problem.
2. If service calls are allowed to return any AXML trees with service calls, the problem is undecidable. The intuition for this is the following. Optimizing too rarely may lead to poor activation decisions, which could have been avoided if we had chosen to invoke the optimizer more often. Optimizing too often, on the other hand, may also be suboptimal. For instance, assume a document d has two subtrees t_1 and t_2. The optimizer may rewrite t_1 into t'_1. Instead, we may have performed some service call activations, which might have turned t_2 into t'_2 such that considering simultaneously t_1 and t'_2 would have enabled an interesting optimization - for instance, the elimination of a common sub-expression. If we pick the optimization step initially mentioned, and rewrite t_1 into t'_1, the opportunities to which lead t'_1 and t_2 may be missed. Thus, one can exhibit a document when optimizing *before each call activation*, even assuming optimization costs zero, is suboptimal. Moreover, in reality, optimization also takes some time, thus very frequent optimization is impractical.

Based on this analysis was devised and implemented OptimAX, an optimizer for *one-stage AXML optimization*. OptimAX was demonstrated in [14]. The optimizer is highly customizable for the needs of different applications, both in the set of rules it uses and in its search strategies, to cope with the explosion which may be entailed by exhaustive application of the optimization rules. These issues are discussed in [15], which also presents experiments demonstrating OptimAX efficiency.

In the full evaluation setting, we recommend invoking the optimizer once on SC_0, then on SC_1, then on SC_2 and so forth, a heuristic which we find a reasonable compromise between optimization costs and optimization-brought benefits.

3.4 Related works

Clearly, the work on ActiveXML optimization can be seen as a subset of the larger area of work on distributed data management [60, 75]. Many classical techniques described in these works have been applied to our settings, such as: replication, distribution of computations (which has a special flavor in ActiveXML due to the presence of location in the language) etc.

The ActiveXML model and language have lead to several theoretic works, on issues such as the impact of types in ActiveXML evaluation [16, 70] and static analysis [17]. In contrast, the works described in this

chapter are concerned with performance issues arising from the evaluation of ActiveXML documents. While these issues are clearly related to the general setting of distributed query processing [60], ActiveXML brings specific challenges due to its expressivity, and in particular its ability to add new service calls in a document whose evaluation is ongoing.

The work described in [8] and [9] reconsiders the problem of query evaluation over ActiveXML documents from a new perspective. All the Web service calls considered in these works have already been activated. The question then is what needs to be done when one (running) Web service call brings a new result: do we need to re-compute the query result, or not? Other related problems are deciding whether a given result can ever belong to the query. The solution is based on implementing a Datalog style view maintenance engine over an ActiveXML document.

The work described in this chapter can be placed in the larger setting of distributed XML data management. The interest of combining XML documents, queries, and services is shared by the early XL work [46], more oriented towards an imperative programming style. In a quite similar manner, subsequent works have considered the efficient evaluation of XQuery queries over documents distributed at several sites [121, 122]. These works can be seen as providing XQuery-specific solutions to the general problems raised by distributed query evaluation, changing the data model, but preserving the overall paradigm. The ActiveXML general approach, in contrast, gives proeminence to a document with intensional components, and can be seen as more declarative. Moreover, and perhaps most interestingly, OptimAX is an *external* optimizer, in that it does not require changes to the optimization and evaluation modules of a (distributed) query processor. Rather, its role can be assimilated to that of a mediator, which works on top (and cooperates with) independent query processors via a high-level query (or in our case, Web service) interface.

Other works have targeted more specifically the efficient implementation of an XML streaming query processor [47]. In our work, we have used a simple one, obtained by wrapping an off-the-shelf non-streaming query processor with state buffers.

Finally, new interesting work is still going on in the area of relational distributed query processing. More specifically, new negotiation strategies are being proposed to enable sites to agree on some distribution of the work entailed by distributed query processing [77]. Such strategies could probably be studied to see how they can adapt to the ActiveXML setting, which does provide the possibility to install new data and new computations to a remote peer (via the $newNode$ service we have presented). Currently, the OptimAX approach for modeling the interest and benefits of each peer is quite simplistic, i.e. we assume that all peers are willing to cooperate, and optimization is performed for the benefit of only the respective peer.

3.5 Context of the work presented in this chapter

ActiveXML has been used in several contracts of the group, most notably in EDOS (**E**lectronic **D**istribution of **O**pen-source **S**oftware) [116], an European project involving the company which builds the Mandriva Linux distribution. More recently, we have been involved in the WebContent [120] project, aiming at building warehouses of semantically rich Web documents. WebContent involves among other partners EADS, the European defense company, and the French companies CEA and Thales. Within WebContent, I have coordinated the work package concerned with the peer-to-peer architecture. We have proposed OptimAX to integrate the project partners' distributed Web services in the peer-to-peer WebContent platform [3].

The work on ActiveXML distribution, replication, and lazy query evaluation has involved Nicoleta Preda, Florin Dragan, and Cosmin Cremarenco during their final year of engineering studies in 2003. Some preliminary work on ActiveXML optimization was performed in 2004 with the help of Nicolaas Ruberg and Gabriela Ruberg while were visiting the group, from their home university of Rio de Janeiro, and later

involved Emanuel Taropa, visiting from Yonsei University in Korea in 2006. The core of the AXML optimization work presented here was part of the PhD thesis of Spyros Zoupanos.

3.6 Conclusion

ActiveXML is a very expressive language capable of specifying complex distributed XML data management applications. Many different perspectives exist on the language, such as: data integration tool, platform for implementing mash-ups (combining independent Web services without resorting to imperative programming), distributed Datalog queries on trees etc.

The works to which I contributed and which were described in this chapter aim at developing specific optimization techniques to reduce the total work involved the execution of specific AXML-related tasks: querying intensional documents, and fully evaluating an AXML document. Due to the expressive power of the language, optimization in the customary sense (complete exploration of a search space and choice of the optimal strategy) is undecidable. Therefore, we proposed and deployed a one-step static optimization approach, likely to be efficient in most practical applications, and whose overhead can be kept in check by appropriate limits on the size of the search space it explores.

Chapter 4

XML warehouses on DHTs

The current development of peer-to-peer (P2P) information sharing has opened the way for supporting rich data management applications in a distributed environment. Of particular interest is the support of structured queries over distributed XML data, as XML is well suited to represent the variety of data that may be shared within a P2P system.

We consider P2P applications with a possibly very large number (millions) of XML documents stored in a large number (up to the thousands) of peers. We assume that peer volatility is moderate, i.e. a peers typical online span is of the order of a few hours (as opposed to a few minutes). We note that this model encompasses a wide range of reallife applications, such as scientific data sharing platforms, shared bookmarks, or virtual libraries (e.g., the French federated online scientific library HAL [118]). We consider in particular *structured peer to peer networks*, which guarantee acceptable bounds on the number of hops needed to route a message from any network peer to any other. Such bounds are crucial for the scalability of a peer-to-peer data management architecture.

In this chapter, Section 4.1 outlines the document and query model we consider, and the generic DHT model of the underlying structured P2P network. Then, Section 4.2 describes the KadoP architecture and algorithms for indexing XML documents on DHTs, as well as crucial optimizations which were brought to make the platform scale. The KadoP index enables the processing of conjunctive tree pattern queries quite efficiently, however, it cannot be tuned to provide better support for specific queries. In Section 4.3, we turn to the more complex problem of building and exploiting materialized XML views in DHT networks. Thus, ViP2P complements KadoP's generic indexing abilities with more flexible, application-driven access support structures.

The KadoP and ViP2P distributed algorithms were validated through large-scale experiments, some of which are discussed in this chapter for their interesting insights. It is worth pointing out that no other DHT-based XML sharing platform has demonstrated practical scalability at the data and network sizes which we consider. We compare our work with similar related works in Section 4.4, and relate it to research contracts and Master and PhD student activities in Section 4.5.

4.1 Preliminaries

The KadoP and ViP2P platforms described in this chapter share a set of basic settings which we outline here: the data and query model, described in Section 4.1.1, and the underlying P2P network model, which we present in Section 4.1.2.

4.1.1 Document and query model

In our setting, each peer may publish a set of XML documents. Each document d published by a peer is uniquely identified by the pair (p, d), where p is the numerical identifier of the peer that checked it in, and d the document identifier within this peer. A document (p, d) is a labeled unranked tree, comprising element and text nodes. (For simplicity, we do not distinguish between elements and attributes.) Each element is labeled with a string symbol from some alphabet $Label$, and is is uniquely identified by a structural identifier (sid for short). As previously explained in Section 2.1, structural identifiers allow deciding by comparing two identifiers, whether their corresponding elements are structurally related, i.e, one is an ancestor (or parent) of the other, or not. In this chapter, we use structural identifiers of the form $sid = (start, end, lev)$. Here, $start$ (resp. end) is the number assigned to the opening (resp. closing) tag of the element, when reading the document and numbering its tags in the order they appear in the document. The third value lev denotes the elements level in the tree. An element e_1 is an ancestor of element e_2 iff $e_1.start < e_2.start$ and $e_1.end > e_2.end$. Moreover, if $e_1.lev = e_2.lev - 1$, e_1 is the parent of e_2.

We consider queries described by conjunctive tree patterns. In these patterns, each node is labeled either with an element or attribute name, or with a keyword, while edges can correspond to ancestor-descendant relationships (denoted $//$) or to parentchild relationships (denoted $/$). A subset of the pattern nodes are return nodes. This tree pattern dialect can be seen as a variant of the XAM language presented in Chapter 2, obtained by:

- (restriction:) allowing only non-optional, unnested edges;
- (restriction:) allowing only cont annotations and only on the return nodes;
- (extension:) allowing nodes to be labeled not only with element names, but also with words occuring in XML text nodes.

The extension to support pattern nodes labeled with words is conceptually quite simple, but it is of important interest in a distributed context, where users may have little or no knowledge of the structure of the documents, and may prefer to explore the data available in the network by means of keyword searches.

4.1.2 Distributed hash tables

By the name of distributed hash tables, or DHTs in short, are referenced a family of distributed data structures providing the abstraction of a hash table endowed with the usual operations *put(key,value)* and *get(key)*. DHT implementations distribute the *(key, value)* pairs across several sites or peers. Two remarkable properties of DHTs are interesting from the viewpoint of distributed data management:

Performance: the operations *put(key, value)* and *get(key)* are realized with a typical upper bound, in the number of messages exchanged across the distributed network, of $O(log(N))$, where N is the size of the network;

Dynamicity: DHTs provide mechanisms for dynamically adjusting to network changes, i.e. peers joining or leaving the network. Modulo some network stabilization time, the results of *put* and *get* operations are not affected by such changes.

Well-known examples of DHTs include Pastry from the Rice University [81], Chord from the MIT [91], Bamboo [114] etc. A common DHT API along the lines described above is presented in [41].

4.2 KadoP: efficient XML indexing on DHTs

This section presents KadoP, a system for efficiently querying XML documents published on top of a DHT network. In a nutshell, KadoP implements a full-text index of the XML documents published in the DHT nework, and distributes it over the DHT. Queries are processed by evaluating structural joins over the distributed index entries.

Section 4.2.1 describes the system architecture and basic algorithms. These are fairly simple, however, making them scale was not, and several techniques achieving this are presented in Section 4.2.2. These techniques are validated by experiments outlined in Section 4.2.3.

4.2.1 KadoP overview

The KadoP system leverages a DHT (in particular, we use Pastry [81]) to implement a distributed index of element names and words[1] appearing in all the documents of the network. We henceforth use term to refer to an element name or a word occuring in text inside an XML document, without distinguishing among them. The indexing scheme of KadoP is based on the Term relation, defined as follows:

- *Term(p,d,sid,l)*, where l is the label of the element (p, d, sid);
- *Term(p,d,sid,w)*, where w is a word occuring inside a text node whose parent is the element characterized by (p, d, sid)

We refer to a tuple in *Term* as a posting. Given a term a, we refer to the set of its postings as the posting list for a and denote it as L_a. KadoP distributes the *Term* relation among the peers of the system using the DHT. More specifically, *Term* is split horizontally among peers, with peer p in charge of a portion $Term_p$ defined as follows:

$$Term_p = \{Term(p, d, sid, a) \mid locate(a) = p\}$$

The posting lists in $Term_p$ are clustered based on the term value, and the postings of a term are ordered in the lexicographic order dictated by the (p, d, sid) attributes.

The publication of an XML document d in KadoP proceeds as follows. The document is parsed, and for each element name and word occuring in the document, the list of postings for a in d is built. This list is then sent to the peer p such that $locate(a) = p$. The peer will add it to the $Term_a$ list (and create this list in the process if no posting for a had been previously stored at p).

KadoP query processing The distributed *Term* relation essentially implements a data catalog that enables KadoP to locate documents matching a given conjunctive tree pattern query q. Let p be the peer to which the query is submitted. For each term a in q, p asks the peer in charge of a for the posting list L_a, and performs a holistic twig join [36] over all the received lists. For instance, assume that the query q is: $//abstract[contains(., xml)]$. The query peer p uses the *Term* relation to find the $sids$ of all elements labeled abstract and all parents of text nodes containing the word xml. A structural join on these sid collections computes the IDs of all $abstract$ elements that should be returned by the query. At this point, one can contact the corresponding peers and ask for the full $abstract$ elements to be returned, based on their sids.

As described above, the KadoP query processing algorithm is capable of handling conjunctive tree pattern queries. One may however consider the processing of more complex queries, such as for instance those expressed in some XQuery dialect. Given such an XQuery query Q, one could extract the largest possible

[1] Common stop words such as *a*, *the*, *of* etc. are excluded from the index.

tree patterns, as we have discussed in Section 2.2.2. Such patterns are not necessarily conjunctive. In fact, as soon as the query Q has a return clause, the query patterns are sure not to be conjunctive, i.e., they will feature some optional nodes. In a large network setting, optional nodes are not helpful for identifying the documents which may contain supersets of the query results, as a document may lack matches to one or several of the optional nodes, yet contribute to the query result.

Therefore, from each pattern p^Q extracted from Q, one can obtain a corresponding conjunctive-only pattern p_c^Q, by:

- copying p^Q from the root downwards, and stopping as soon as a non-conjunctive (or optional) edge is encountered;
- replacing all node annotation attributes with id.

KadoP is used to evaluate all the conjunctive patterns such as p_c^Q, resulting from Q. Then, Q is sent to the peers hosting documents that had matches for the p_c^Q query. Observe that this may push quite a significant part of the XQuery processing to the peers hosting the documents. However, the query has been exploited to narrow as much as possible the set of peers which are contacted in this stage.

4.2.2 Scaling up KadoP

As discussed earlier, the distributed index is an inherent component of query evaluation and its efficiency is thus crucial for good system performance. However, our experiments with an initial implementation of KadoP [11, 12] have demonstrated that the index has severe scaling limitations, caused primarily by the processing of long posting lists. (Similar points have been raised by other works on DHT-based systems [2, 55]). We identify three important metrics that assess the performance of a distributed index (and essentially form the guiding criteria behind our techniques):

1. indexing time;
2. query response time;
3. bandwidth consumption.

Observe that bandwidth consumption can only be reduced for the processing of the conjunctive tree pattern query up to identifying the documents which contain query results. The bandwidth consumption entailed by transferring the results to the query site cannot be reduced unless one introduces some extra access support structures.

Reducing indexing time To index a document, the system constructs in one traversal the element postings, and routes each posting, via the DHT, to the peer in charge of the corresponding term. Postings of the same term are buffered and sent in batches, which reduces slightly the index latency (the time it takes to index a document) compared to the naive method of routing each posting separately. More important gains are obtained by extending the DHT API, and replacing its data store. We discuss these two points next.

The original *insert* operation in a local DHT index is very inefficient. According to the standard DHT API [35], when a peer in charge of a key k receives a put request, it (1) reads the old value for k, (2) applies a DHT-specific reconciliation of the old value and the new entry, and (3) inserts the result in the repository. Performing this operation for n successive entries associated to key k leads to a total I/O complexity of n^2, since i entries must be read in order to append the $(i+1)$th entry. To overcome this issue, we extend the DHT API with a new operation, namely *append(key, entry)*, to obtain an indexing of linear cost.

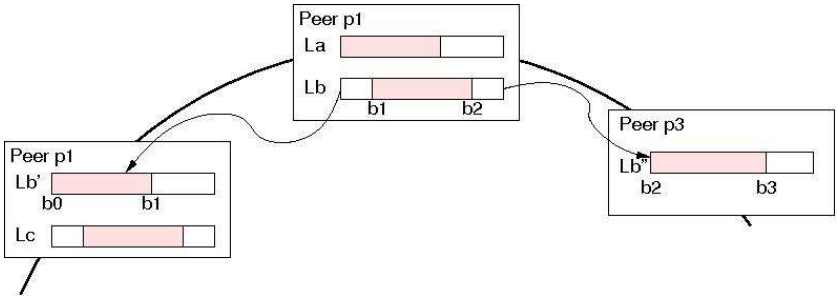


Figure 4.1: Distributed Posting Partitioning example.

To speed up indexing, we tuned the DHTs communication buffers to cope with many small messages generated by small posting lists. Another important improvement is the tuning of the index storage. More precisely, $Term_p$ at peer p is organized as a B+-Tree, using term as the search key, and the postings associated with a given term are lexicographically ordered by (p, d, sid). (KadoP was built on PAST [82], a storage facility on top of the Pastry DHT, which by default uses gzipped XML files.)

In our experiments, enhancing the API, buffer tuning, and replacing the index storage has sped up publishing by two to three orders of magnitude. As a side effect, replacing the index storage has also reduced index query processing time by one order of magnitude.

Reducing query response time As discussed earlier, the peer p in charge of a query q performs a holistic twig join on the posting lists received from other peers. We present three optimization techniques which we implemented to reduce both the time to receive the first results from the holistic twig join, and the time until the last result has been obtained (thus, the total evaluation time).

First, let us see what determines the moment when the first answers can be received. The holistic twig join algorithm can only start working when data is available in at least two inputs, and can only start producing results when data is available in all the inputs. Obviously, the calls to all input posting lists are made in parallel. However, the *get()* operation in the DHT API is defined as being blocking operation. That is, it returns only when the content of the posting list has been fully retrieved. Therefore, the holistic twig join must wait until at least two lists have been entirely received before it can start processing. To shorten the time to the first result, we modified the DHT API (and the actual DHT system) by adding a pipelined *get* method, which transfers posting lists asynchronously. This simple modification brought important performance improvements.

Once this was achieved, the total time to process queries involving long posting lists was still quite high if one or several of the posting lists were very long. Such long posting lists occur in the presence of frequent terms, which appear many times and/or in many documents. Long posting lists influence, first, the time to transfer the inputs to the holistic twig join operator, and second, the time to process these inputs.

To reduce these times, we devised a new physical organization of a posting list, which we call *Distributed Posting Partitioning*, or DPP in short. Recall that each posting list La must be stored in the increasing order dictated by the document identifier and then the sid of the a nodes. A DPP has the flavor of a distributed B+ tree. Originally, the entries of one posting list are all in one data block, which may overflow in B+-tree fashion after reaching a given threshold. The overflow blocks are shipped to different peers, and the original peer keeps the knowledge of (i) the smallest and largest value in each such remote block, and (ii) the logical ID of the peer to which the block was shipped. Figure 4.1 illustrates this on a DHT fragment containing

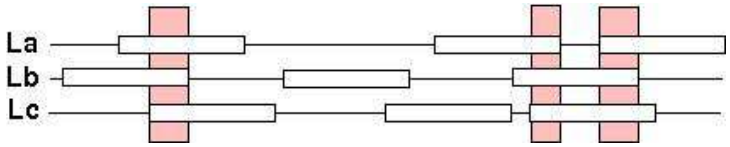


Figure 4.2: Block-oriented joining of posting lists.

three peers, which, together, store the lists L_a, L_b and L_c. The shadowed areas of each posting list represent the portion that is full with data. The list L_b is split using a DPP.

Based on the DPP, instead of running one single holistic twig join, we may run several in parallel, each of which exploits one block from each posting list involved in the original join. Consider a query q with n nodes and assume that the postings for each query node, say i, are split into several blocks C_1^i, ..., $C_{m_i}^i$. For instance, in Figure 4.1, the three L_b blocks are determined by the value intervals: $[b_0, b_1)$, $[b_1, b_2)$, and $[b_2, b_3)$. The idea is to perform the join on a per-block basis, allowing up to K parallel-running joins (where K is a parameter set in advance). The processing starts with an initialization step where only the conditions describing the DPP blocks are fetched, and a description for each parallel join is generated. During the join, the first K blocks for each posting are fetched in parallel, e.g., C_1^1, ..., C_1^K, C_2^1, ..., C_2^K, and so on, until C_n^1, ..., C_n^K. For each combination obtained by taking one block from each input, it can be determined in constant time whether the intervals delimiting the blocks intersect or not. If they do, then the corresponding holistic twig join is evaluated. At a given moment, all the block-oriented holistic twig join algorithms thus obtained run in parallel. To increase the throughput, we do not require that results be produced in lexicographical order, and return to the user the first results produced by each join. When an active block in a posting list has been traversed by all the joins which need it, the next block in this list is activated etc.

An issue is whether the algorithm generates too many joins. The answer is no, because the postings are lexicographically ordered. Indeed, suppose we are joining n posting lists, consisting respectively of $m_1, \ldots, m_n$ blocks. Then one can prove that we do not have to consider $m_1 \times \ldots \times m_n$, but at most $m_1 + \ldots + m_n$ joins, and in practice, much less. Figure 4.2 illustrates this on an example, where we join the posting lists for three query terms. Each horizontal narrow rectangle represents a block of postings. Their relative positions along the horizontal axes reflect the relative order of the postings in each block. The vertical rectangles outline the posting list intervals in which postings exist in all the lists involved in the join. Several such block joins can be evaluated in parallel, which allows exploiting the possibilities of parallel data transfer through the network.

Moreover, the join operator learns the block boundaries for each posting list before actually starting to fetch them, and thus can only require the block portions that are involved in the joins, i.e., those corresponding to the colored rectangles in Figure 4.2. This recalls the classical Zig-Zag joins [50].

Reducing bandwidth consumption This is the third important performance aspect in KadoP. A set of techniques are implemented to reduce bandwidth consumption. One example is the space-efficient encoding of sids. Recall from the KadoP data model that to each word, we assign not an sid of its own, but that of its closest enclosing element. This allows counting only the element nodes, not the words, which has a sensible impact on sid ranges and the space needed to represent them. At a higher level, in the WebContent platform built on KadoP [3], bandwidth consumption was reduced by placing the holistic twig join on the peer where the longest posting list resides.

A more ellaborate technique consists of employing Bloom filters specially devised to compactly represent the entries in a posting list, and then reduce the data transfers entailed by query processing based on

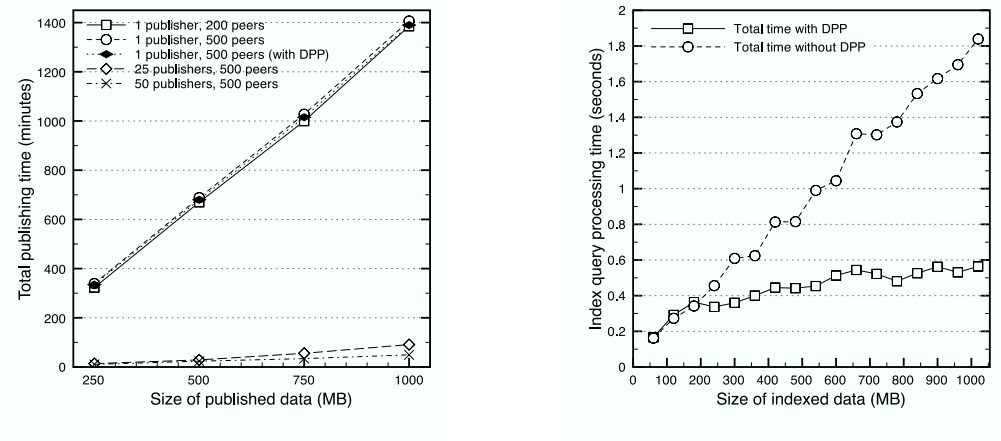


Figure 4.3: KadoP performance: document publication (left) and query processing (right).

these filters. The technique is described in details in [10].

4.2.3 KadoP performance

The KadoP platform to which these improvements were made achieved the linear scalability with the size of the data that one may expect, on networks of up to 1000 peers, and 1 GB of published XML documents. The detailed description of the experiments can be found in [10]. We briefly outline here their most interesting aspects, based on the two graphs in Figure 4.3.

In these experiments, a set of documents are indexed and queried in a network of 200, respectively, 500 peers. The peers are deployed in Grid5000 [117], a research-oriented infrastructure hosted in French laboratories and spanning 9 cities across the country.

The graph at the left in Figure 4.3 depicts the document publishing time as a function of the total indexed data size. The number of peers publishing the documents is either 1, 25, or 50. One can immediately see that when documents are published in parallel by many peers, the total publication time is much shorter than when they are published from a single peer, whose available outgoing bandwidth becomes the bottleneck.

The graph at the right in Figure 4.3 plots the execution time for a simple tree pattern query of three nodes, on the documents indexed in the previous experiment. One query node involves a large posting list, which has been split by the DPP structure. We can see that using the DPP indeed shortens the total evaluation time as expected, due to the parallel evaluation of several holistic twig joins.

4.3 ViP2P: materialized XML views on DHTs

As we have demonstrated in KadoP, DHTs can be used to build efficient large-scale XML data management platforms. KadoP achieves good scalability for tree pattern queries in such large networks, by means of a distributed index consisting of posting lists and automatically assigned to various peers in the network.

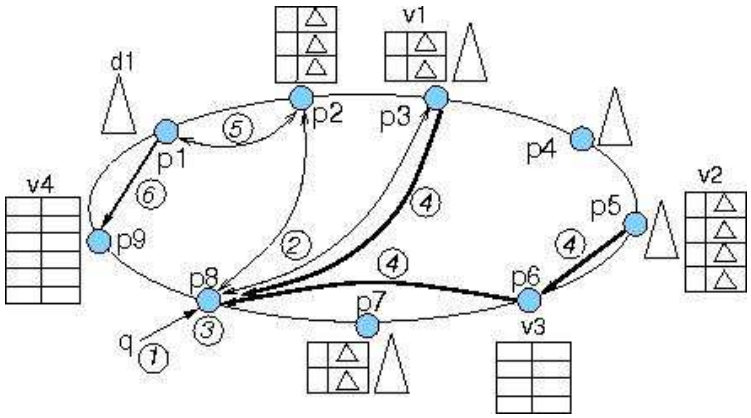


Figure 4.4: Overview of the ViP2P architecture.

We have taken the DHT-based approach a step further, and set out to devise a model where in a DHT-based XML sharing network, one can define specific *materialized views* as XML queries to be continuously evaluated over the documents ever to be published in the network. The *definitions* of the materialized views are then indexed in the peer network, to enable other peers to find them. Then, when a peer must process a query, it starts by looking up in the network for view definitions, then rewrites the query using the views and finally, executes a distributed rewriting plan in order to obtain the query results.

We present the ViP2P architecture in Section 4.3.1. We show in Section 4.3.2 how materialized views are filled with data as documents are published in the network. Section 4.3.3 discusses several strategies for indexing view definitions, and looking them up when a query must be rewritten and evaluated. This section and the previous one also include (abridged) experiment reports.

4.3.1 ViP2P architecture

The architecture we envision is depicted in Figure 4.4. Network peers labeled p_1 to p_9 store documents, shown as triangles, and/or views, shown as tables. Such tables attributes may be of type *xml* (whose values are serialized XML subtrees), in the style of the native XML data type in SQL/XML 2003. Such attributes are shown as triangles inside tuples. We designate a document d or view v at peer p by the notation $d@p$, respectively, $v@p$.

Views (and queries) are defined by tree patterns. We use the same tree pattern language as in KadoP, for the same considerations, explained in Section 4.1. *The tree pattern defining each view (not the tuples in the view extent) is indexed in the DHT*. Query processing can be traced following the numbered arrows in the Figure. Assume query q is asked at peer p_8 (step 1). Then, p_8 will perform a DHT look-up to find which view definitions may be useful to rewrite the query. For instance, p_2 and p_3 may return to p_8 relevant view definitions (step 2). Peer p_8 will then run a view-based query rewriting algorithm, trying to reformulate q based on these definitions (step 3). A query rewriting is a logical algebraic plan based on some views, in our example, $v_1@p_3$, $v_2@p_5$, and $v_3@p_6$. After picking a rewriting, p_8 transforms it into a distributed physical plan, which runs in a distributed fashion (step 4, thick arrows denote data transfer). In our example, v_2 is sent to p_6 which joins it with v_3 and sends the result to p_8. At p_8 it joins with v_1 which is sent from p_3.

Each ViP2P view v is complete, i.e. it includes $v(d)$ for any document d in the network (modulo some

update propagation time). To obtain such views, whenever a new document, say $d_1@p_1$ in Figure 4.4, is published, the publishing peer performs another type of lookup (step 5) to determine (possibly a superset of) the view definitions to which the new document may contribute tuples. In the Figure 4.4, such definitions are returned by p_2, and p_1 finds out that d_1 contributes some tuples to the view $v_4@p_9$. The tuples are sent to p_9 (step 6), which adds them to the view extent.

The notion of query rewriting considered in ViP2P, and the rewriting algorithms employed, have been described in Section 2.2.3.

4.3.2 Materializing tree pattern views in DHTs

Given a document d and a view v, we say d affects v if $v(d) \neq \emptyset$, in other words, if d contributes some results to the view v.

Assume peer p decides to establish a view v. Then, when a peer p_d publishes a document d affecting v, p_d needs to find out that v exists. To that effect, peer p indexes its view definitions as follows. For any label (node name or word) appearing in the definition of the views $v_1, v2, \ldots, v_k$, the DHT will contain a pair where the key is the label, and the value is the set of view URLs $v_1, v_2, \ldots, v_k$.

When a peer p_d publishes a document d, p_d performs a lookup with all d labels (node names or words) to find a superset S of the views that d might affect. Then, p_d evaluates $v(d)$ for each $v \in S$. Finally, p_d sends, for each view v, the tuple set $v(d)$ (if it is not empty) to the peer p_v publishing v.

We have so far considered that v is published before the documents which affect it. The opposite may also happen, i.e. when v is published, a document d affecting v may already exist, and $v(d)$ needs to be added to vs extent. Two solutions can be envisioned for such cases.

- If a DHT-based index of all published documents is available, one may use it when v is published to evaluate v, against the set of existing documents. The index may be at the granularity of individual nodes or keywords, as in KadoP, or at the granularity of documents as in [49] etc.
- If no predefined index is established, the publisher p_d of a document d must to periodically look up the set of views potentially affected by d, and send $v(d)$ to those views as described above.

View maintenance must be performed when documents change or disappear. When a document is deleted from the system, a similar view lookup is performed, and the peers holding the views are notified to remove the respective data. We model document changes as deletions followed by insertions.

View materialization experiment To demonstrate that our techniques scale, we present a view materialization experiment. We deployed 1000 peers on the Grid5000 [117] research network. The peers are distributed over 250 machines, spread across 7 sites.

The peers publish a total of 2000 XMark benchmark [85] documents of equal size; the total size of the network documents varies across successive runs, from 400 MB to 3.2 GB. The peers also publish 500 views of up to 7 nodes. 70 views are affected by all documents. The others use XMark node names but have no results on XMark documents. The documents and views are split uniformly over the network. All 500 views are retrieved for all 2000 documents.

Figure 4.5 shows the time to evaluate the 500 views on the 2000 documents, in parallel on the 1000 peers of our experiment. This is the time between the moment when all peers start looking up views which may be affected by their documents, and the moment when the last tuple thus obtained has been stored in the respective view. Since the degree of parallelism is maximized in this experiment (the publication work is spread equally across peers), this is the fastest view materialization that can be achieved by the current platform on this document and view load. It involved exchanging 140:000 messages, containing the tuples

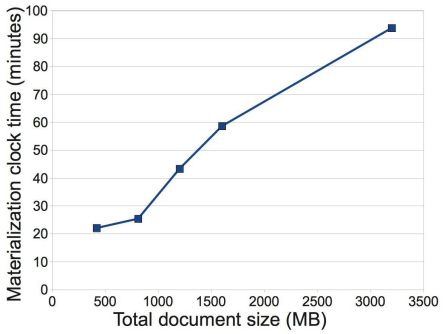
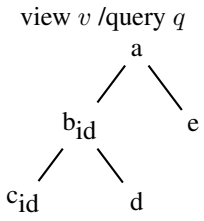


Figure 4.5: View materialization time in ViP2P.



LI index keys	LI&LRI lookup keys	RLI index keys	LPI index keys	LPI&RPI lookup keys	RPI index keys
a, b c, d e	a, b c, d e	b, c	a/b/c, a/b/d, a/e	a/b, a/c a/d, a/e b/c, b/d a/b/c, a/b/d	a/b a/b/c

Figure 4.6: Sample view (or query) and corresponding indexing and lookup keys.

from 2000 documents each of which affects 70 views. The time grows linearly with the total document size. We are currently experimenting various optimizations to further reduce data transfers and thus, view materialization time.

4.3.3 Identifying the views useful in rewriting a query

A second form of view definition indexing is performed in order to find views that may be helpful for rewriting a given query. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy*. For each such strategy, a *view lookup* method is needed, in order to identify, given a query q, (a superset of) the views which could be used to rewrite q. Many strategies can be devised and we present four that we have experimented with. They are all complete, i.e. they retrieve at least all the views that could lead to q rewritings. We analyze their complexity in terms of generated (key, value) pairs and messages exchanged over the DHT in [10].

The four strategies we present, namely LI, RLI, LPI and RPI can be easily followed on the example shown in Figure 4.6. The tree pattern at left plays successively the role of a view v whose definition must be indexed, and of a query q for which we must find interesting views.

Label indexing (LI): index v by each v node label (either some element or attribute name, or word). Given a query q, look up the views associated to all q labels. The LI strategy coincides with the view definition indexing for document-driven lookup (described in the previous section). An interesting variant can furthermore be devised:

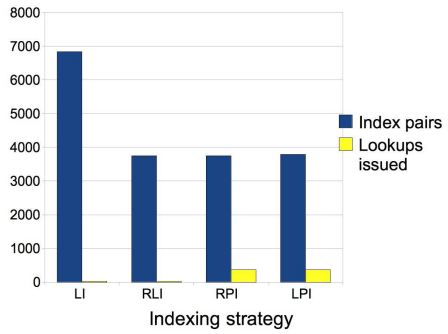


Figure 4.7: Index entries and lookups generated for the views.

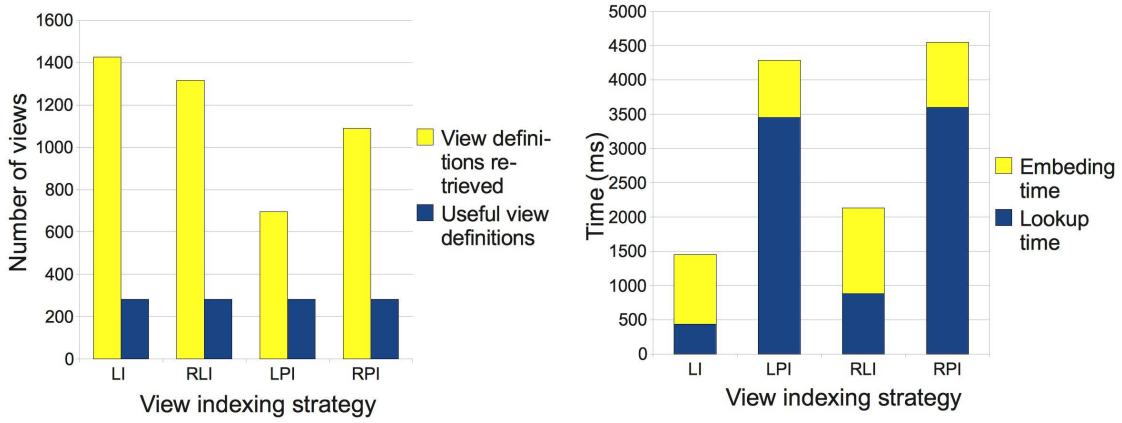


Figure 4.8: View definition retrieval: precision (left) and performance (right).

Return label indexing (RLI): we index v by the labels of all v nodes which return some attributes. View lookup proceeds as for LI: look up all the labels of q nodes. The drawback of LI and RLI is their lack of precision. For instance, the view in Figure 4.6 will be retrieved when trying to answer any query containing one of the tags $\{a, b, c, d, e\}$. In particular, it will be retrieved when attempting to answer the query $//a//b$, although the view cannot be used to equivalenty rewrite this query, due to its nodes labeled c, d and e.

Leaf path indexing (LPI): index v by each distinct root-to-leaf label path of v. Lookup is performed with all the sub-paths of some path in q.

Return Path Indexing (RPI): index v by all rooted paths in v, ending in a node that returns some attribute. View lookup is performed as for LPI.

View indexing/lookup experiment We now describe an experiment comparing the four strategies described above. We use a synthetic query q of 30 nodes labeled $a_1, \ldots, a_{30}$. Each node of q has between 0 and 2

children, and qs height is 5. We generate 1440 views of 2 to 5 nodes based on q, a quarter of which agree with q's labels and structure (whenever a node labeled a_i is an ancestor of a_j in the view, this was also the case in q), another half of which may or may not agree with the tags and the structure, and the final quarter of which agrees with the tags but totally disagrees with the structure of q (for a detailed explanation of how the view are obtained, see [66]). We have indexed these views in the same peer network as described in the previous Section, following the LI, RLI, LPI and RPI strategies. We performed the four corresponding lookups. On each retrieved set of view definitions, we apply the same pruning step (discussed in Section 2.2.3) to retain a superset of the views that actually may lead to rewriting q. Since all the strategies are complete, all the sets obtained after pruning coincide.

Figure 4.7 presents the number of (key, value) pairs added to the index by each view indexing strategy, and the number of lookups needed by each strategy for the query we considered. As expected, LI leads to most index pairs. With respect to query-driven lookup, LI and RLI lead to 30 lookups, much less than LPI and RPI lead to 370 lookups.

Figure 4.8 (left) shows how many views have been retrieved for each strategy, compared with the number of views after pruning. We see that the path indexing-lookup strategies (LPI and RPI) are more precise than label based ones (LI and RLI). Moreover, LPI is the most precise. This is because LPI uses longer paths as keys, thus, it describes views more precisely, eliminating some false positives.

Figure 4.8 (right) shows the time to obtain the view definitions via lookups, and the time to prune the resulting view sets. The Figure shows that the simple LI strategy is the best. Indeed, even though Pastry lookups are asynchronous, issuing many lookups from the same peer is expensive, thus, LPI and RPI, which needed 370 lookups, are significantly slower. LI makes up for its low precision by requiring few lookups.

In this example, rewriting the query based on the relevant views (282 in this example) takes around 6 seconds, whereas finding the first solution takes around 0.5 seconds. Comparing this with the times in Figure 4.8, one notices that view definition look-up is quite short, which validates the feasibility of retrieving view definitions at query rewrite time. One may also consider locally caching view definitions, to completely avoid look-ups.

4.4 Related works

Several recent works have targeted the management of relational data in distributed hash tables, such as [53, 19, 55]. Indexing XML data raises specific challenges due to the rich structure of XML documents.

XML data management on DHTs has been addressed in some recent works [33, 90, 49, 80]. All these works propose establishing a fixed XML indexing model and thus compare directly with KadoP.

The earliest work [49] appeared in 2003, prior to the first KadoP publication of 2004 [11]. In [49], a document is indexed in the DHT by all the parent-child paths appearing in the document. Words are not indexed. The authors suggest adding some text histograms in the index, to account for the appearances of words or more generally, values in the documents text. Thus, the index allows identifying a superset of the documents containing matches to tree pattern queries. The index is precise for linear queries only, i.e., those of the form $a_1(/|//)a_2(/|//)\ldots(/|//)a_k$, where all the a_is are element names. Branching queries or keywords lead to imprecision in the index queries. In contrast, the KadoP index is precise for the conjunctive tree pattern language we consider, including branching and keywords. The index in [90] is also established at document granularity level, and does not allow keyword search. Tree pattern queries are not considered. In [33], XML fragments are distributed in the network and the identifiers of fragment roots are indexed. Query processing starts from root fragments and proceeds downwards, unlike KadoP where the entry points for query processing are found directly from the index. Finally, the recent work demonstrated in [80] uses

a polynomial encoding scheme on XML trees, on one hand, and tree pattern queries, on the other hand, to locate documents matching tree pattern queries. The claim is that this encoding leads to few false positives (documents identified by the index query which do not contain query results). However, large-scale experimental reports on this system are not yet available.

ViP2P is the first system which manages materialized XML views on DHTs. It differs from all the XML DHT-based indexing proposals in its ability to support generic views. The closest similar work considered the DHT indexing of RDF materialized views [89], leading to quite different rewriting problems.

XML query rewriting works to which ViP2P relates were surveyed in Section 2.3.

4.5 Context of the work presented in this chapter

The work on KadoP has taken place within the framework of the MDP2P (Large-scale **P2P D**ata **M**anagement) ANR project (2004-2007). The project was a collaboration with the ATLAS INRIA team, lead by Patrick Valduriez. I have been responsible of our group's work within this project. The KadoP work has been the topic of the MS internship (2004), and then the PhD thesis [78] (2005-2008) of Nicoleta Preda, who developed most of the code. KadoP has also been used in the EDOS [116] and WebContent [120] projects. The engineers Gabriel Vasile and Mohamed Ouazara each spent two years in our group working on the EDOS, respectively, WebContent project, consolidating the code and adapting it to the specific needs of these projects.

The ViP2P project started in 2007 within the thesis of Spyros Zoupanos (to graduate by the end of 2009) and an internship of Ravi Vijay (MS student at UCSD, in 2007). I have implemented the query rewriting component, the local view evaluation module, and the execution engine (the latter two are borrowed from the ULoad prototype [23]). The remaining modules have been developed by Spyros and the engineer Alin Tilea. Two interns (Silviu Julean from UVT, Romania, and Julien Leblay from U. Paris XI) are currently working on introducing a semi-join technique in the execution engine, respectively, replicating views to increase availability. The PhD thesis work of Konstantinos Karanasos (started in 2009) extends the ViP2P platform with support for RDF and RDF annotations on XML corpora.

Most ViP2P results date from 2009 and are part of the work on the CODEX project previously mentioned.

4.6 Conclusion

Integrating data from many distributed sites is still an open problem, although many interesting techniques exist [92, 97]. Distributed Hash Tables bring important assets in the context of large-scale data distribution, notably scalable message routing and graceful adaptation to sites joining and leaving.

We have thus set out to build efficient large-scale XML data management platforms exploiting DHTs to build distributed content indices. Several impedance mismatch issues were detected and solved, mostly due to the difficulties of DHTs when handling numerous or large messages. In KadoP, specific lower-level optimizations were brought to speed up the processing of XML tree pattern queries, notably: DPPs for parallelizing structural join evaluation, and Bloom filters to reduce the data transfers entailed by query processing. In ViP2P, we went one step further and gave each peer the flexibility to declare the tree pattern views that it wishes to store. Such views are then filled with data, in the manner of subscriptions. Views can then be re-used for rewriting queries.

Our DHT-based platforms were tested on a few hundred physical machines spread around France. We have demonstrated the good scalability of our index and view building algorithms. We also have established that the overhead of *small* messages, such as needed in order e.g. to lookup some view definitions, is typically modest.

We are looking forward to continuing this work to support for data and network changes, and more complex data models. These issues are described in the next Chapter.

Chapter 5

Perspectives

The core of the research work which I plan to pursue in the next three to five years is related to fulfilling the vision of transparently distributed large volumes of data. The purpose is establish models and tools for efficient data exchange in large networks, so that data can be produced, enhanced, changed, and queried anywhere in an efficient and reliable way.

The directions to be pursued are described in the next two sections: large-scale XML data management (Section 5.1) and query processing for Semantic Web data and annotated document corpora (Section 5.2).

5.1 Massive XML data networks

Much remains to be do done to complete, expand, and consolidate the distributed XML data management approach put forward by systems such as KadoP and ViP2P. The work can be organized around the two research directions described next.

5.1.1 Data dynamicity in DHT networks

In a context of XML data management on DHTs, efficient algorithms are needed for handling changes that may occur in the contents of documents so also the views, and in the set of peers present in the network.

First, one needs to propagate changes to an XML document, towards all the materialized views affected by the change. This comprises the views which stored some data from the previous version of the document, as well as those that need to store data from the new version. The problem of XPath view maintenance has been addressed in the setting where the data and the views are stored in a relational database [83, 84]. Incremental XAM maintenance may offer some interesting optimization if the XAMs use well-chosen structural identifiers, since some provenance information, for instance, the path leading to a given node, can be encoded in the ID and thus speed up reasoning about the incremental maintenance operations. The distribution of views in DHTs raises its own performance issues, since one must manage the data structures used for view maintenance, in a distributed network.

Second, one may consider the opposite problem, where updates to an XML view must be propagated towards the documents. Database update through views is a known hard problem that is starting to be addressed for XML views of relational databases [40]. We plan to work on identifying possible and meaningful updates to documents through views, and provide algorithms for supporting update propagation based on tree

pattern (XAM) views. From our preliminary study, it appears again that information-rich XML identifiers can increase the set of situations when an update on the view, can be propagated to the original documents.

Finally, a third dimension of dynamicity concerns the peers joining and leaving the network. Peers in our context are rich with data, both original data (their documents) and derived data (views). A first issue when peers leave the network is to allow their data to remain in the network, by means of usual replication protocol. A second issue is graceful re-insertion. When a peer leaves the network, we may propagate this event to remove his data from all views, and his views from all the view indices. However, if the peer having left, joins again the network, starting with an empty data space if not appropriate, and re-publishing all its resources again may be too lengthy. Thus, we will investigate the possibility to define a peer data space to represent its original and derived data, persist the data space through disconnects, and attempt to smoothly re-insert this data space in the network when the peer rejoins it.

Context and collaborations This work is to be carried over with Konstantinos Karanasos (PhD started in January 2009) and Asterios Katsifodimos (PhD to start in October 2009). Spyros Zoupanos is likely to also contribute after his defense (while holding a temporary teaching position at U. Dauphine.)

The grant framework is provided by the CODEX project, that was already mentioned, the ANR project DataRing (2009-2012, coordinated by Patrick Valduriez from INRIA Sophia Antipolis - Méditerranée). With the help of two INRIA engineers, Alin Tilea (until September 2010) and Jesús Camacho-Rodríguez (October 2009 to September 2011), we plan to consolidate ViP2P and start distributing it for others to experiment with.

5.1.2 Indexing Intensional Information

This work is concerned with the querying of large-scale corpora of inter-related documents. Databases of Web pages are one example, ActiveXML documents where service calls are seen as links pointing to a materialized document are another perspective on the problem. The goal is to be able to efficiently process queries over such distributed documents with shared fragments. We are currently working on specific distributed index structures to be maintained efficiently in the DHT, and an associated algebraic approach for optimization.

Context and collaborations This area of work concerns ongoing work with Serge Abiteboul and Neoklis Polyzotis, and is associated to the WebDam ERC grant headed by Serge Abiteboul.

5.2 Query processing meets Semantic Web

The development of of Semantic Web technologies and in particular the adoption of RDF as a generic language for representing knowledge, has lead to several research prototypes aiming at processing RDF queries on large, centralized databases [1, 101]. Intelligently mapping RDF query processing problems to the existing query processing paradigms and tools is in itself quite a challenging task [88].

If RDF can be seen as lacking structure, it often comes with some associated semantics, expressed in terms of a Description Logic-style language, such as OWL or DL-Lite [37]. Such semantics can clearly be exploited to simplify the processing required by an RDF query, since one can use semantic constraints at the schema level, to avoid saturating the underlying databases of facts, and yet compute the same results.

5.2.1 RDFLens: flexible lenses on RDF data

In the RDFLens project, we aim at building flexible lenses for the interrogation of large RDF corpora. A lens in this context starts with a conjunctive query over the RDF graph, which may be either too general, i.e. return the whole database, or unsatisfiable, that is, may return an empty answer. Given such a query, the system may analyze it and identify "interesting" sub-queries, that are satisfiable and return sufficiently many results, or are very selective with respect to closely similar queries. For instance, consider an unsatisfiable query joining $n + 2$ RDF triple atoms. It may be the case that removing the $n + 1$-th atom makes it satisfiable and leads to obtaining 100 tuples, while removing the $n + 2$-th makes it satisfiable again and leads to obtaining 200 tuples. However, removing both the $n + 1$-th and the $n + 2$-th atoms may lead to an explosion in the size, i.e., the query becomes very non-selective. In this case, the variants obtained from the original query by removing one of the last two atoms are intuitively more informative than the one obtained by removing them both.

The goal is to build and maintain a hierarchy of inter-related conjunctive queries, much in the style of data cubes in traditional warehouses - but in a context where numerical aggregation, such as the typical sums and averages of data wareahouses, are not necessarily pertinent. These queries can be organized in a lattice, each node representing a query, and edges connecting queries that can be obtained from one another. Materializing (or even enumerating) the lattice nodes has prohibitive costs, therefore the lattice must be compactly encoded and unfolded lazily, only on demand. The lattice can also be exploited to pick sub-queries to pre-compute to facilitate the processing of a set of queries etc. A suitable model for RDF data statistics must be identified to build the model upon, e.g. along the lines of the one described in [71].

Semantic schemas could also be called to play a role in this context, for pruning the lattice nodes of which it can be inferred that they are unsatisfiable, and possibly for recommending subqueries to materialize.

Once the model becomes sufficiently stable, we intend to interact with a specialist in data visualization, and in particular, visualization of RDF data, to see whether and how a visualization layer could complement our approach. Visualization in this context is not just icing on the cake. It is much more difficult to approach, understand and manipulate a set of triples, than a set of interconnected relations. Allowing users to start with some suggested queries and expand on them, build variations, and get some early feedback on the satisfiability and selective power of their queries would provide an intuitive way of manipulating the data, based on the lattice structure.

5.2.2 AnnoVIP: exploiting corpora of annotated documents

The ongoing work of Konstantinos' thesis is concerned with the efficient management of corpora of *annotated* documents. Here, by annotation we designate any stand-alone information that comes to complement or characterize the content or significance of a structured document, typically XML. Annotations, or loosely speaking, resource descriptions, can be found easily under the form of HTML Meta tags, user-specified tags on social media such as blogs or Facebook etc. Annotations are also produced by numerous text analysis tools such as, on one hand, those produced by the linguistic databases community, and on the other hand, IR-style analysis modules trying to understand the main topics in a given document. Given the increasing volumes of electronic contents, automated tools for making sense of the documents (typically by means of annotations) and semi-automated tools for connected independently-produced resources together is likely to become more and more important. We have encountered such corpora of annotated documents in the WebContent [120] project. More generally, they are used in applications where a database of documents or text is analyzed to identify specific types of content. For instance, in the Shiri project involving members of the Gemo/IASI team, a set of HTML documents are automatically annotated based on domain-specific ontologies [102].

We plan to study the management of corpora of annotated documents at several levels.

1. The first level concerns the choice of language and model for such hybrid data, part XML, part RDF; RDF query languages are clearly insufficient, whereas using XQuery on the XML serialization of RDF has certainly sufficient expressive power, but is quite awkward.

2. The second level concerns efficient storage models for such mixed corpora. RDF and especially XML are each well served by now, by dedicated storage and query processing engines, however, one may envision brining the two in a single engine, to enable global optimization. An idea is to exploit as such XML engines on one hand, and RDF engines on the other, while integrating the two with the help of an external engine including an optimizer and *cross-model* data access support structures, i.e., indices and views over XML *and* related RDF data.

3. Third, we plan to consider the management of annotated documents in a distributed setting, which is quite natural for such corpora. This is because different viewpoints and different levels of analysis of a given document typically correspond to different actors on distant sites.

Context and collaborations This area of work concerns an ongoing collaboration with François Goasdoué from IASI/Gemo, and the PhD work of Konstantinos Karanasos (started in January 2009). Asterios Katsifodimos (PhD started in October 2009) may also participate. A collaboration with N. Pernelle around the Shiri project corpora is also possible.

Bibliography

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB*, pages 685–696, 2005.

[3] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos. WebContent: efficient P2P warehousing of web data. *PVLDB*, 1(2):1428–1431, 2008.

[4] S. Abiteboul, J. Baumgarten, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing distributed workspaces with Active XML. In *VLDB*, pages 1061–1064, 2003.

[5] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD Conference*, pages 227–238, 2004.

[6] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and web services integration. In *VLDB*, pages 1087–1090, 2002.

[7] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD Conference*, pages 527–538, 2003.

[8] S. Abiteboul, P. Bourhis, and B. Marinoiu. Efficient maintenance techniques for views over active documents. In *EDBT*, pages 1076–1087, 2009.

[9] S. Abiteboul, P. Bourhis, and B. Marinoiu. Satisfiability and relevance for queries over active documents. In *PODS*, pages 87–96, 2009.

[10] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, pages 606–615, 2008.

[11] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying peer-to-peer warehouses of XML resources. In *SWDB*, pages 219–225, 2004.

[12] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying peer-to-peer warehouses of XML resources. In *ICDE*, pages 1122–1123, 2005.

[13] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.

[14] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: efficient support for data-intensive mashups. In *ICDE*, pages 1564–1567, 2008.

[15] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: Optimizing distributed ActiveXML applications. In *ICWE*, pages 299–310, 2008.

[16] S. Abiteboul, T. Milo, and O. Benjelloun. Regular rewriting of active XML and unambiguity. In *PODS*, pages 295–303, 2005.

[17] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. In *PODS*, pages 221–230, 2008.

[18] L. Afanasiev, I. Manolescu, and P. Michiels. MemBeR: A micro-benchmark repository for XQuery. In *XSym*, pages 144–161, 2005.

[19] R. Akbarinia, E. Pacitti, and P. Valduriez. Data currency in replicated DHTs. In *SIGMOD Conference*, pages 211–222, 2007.

[20] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–, 2002.

[21] A. Arion. *XML Acces Modules: Towards Physical Data Independence in XML Databases*. PhD thesis, Université de Paris XI, 2007.

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

[23] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, pages 87–98, 2007.

[24] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *FQAS*, pages 13–25, 2006.

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

[26] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing queries to compressed XML data. In *VLDB*, pages 1065–1068, 2003.

[27] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. XQueC: A query-conscious compressed XML database. *ACM Trans. Internet Techn.*, 7(2), 2007.

[28] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. *World Wide Web*, 11(1):117–151, 2008.

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

[30] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, pages 60–71, 2004.

[31] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling down database techniques for the smartcard. In *VLDB*, pages 11–20, 2000.

[32] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Bridging the XML relational divide with LegoDB. In *ICDE*, pages 759–760, 2003.

[33] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2):247–269, 2006.

[34] Business process execution language for web services. Available at http://www.ibm.com/developerworks/library/ws-bpel.

[35] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.

[36] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[37] D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati, and G. Vetere. Dl-lite: Practical reasoning for rich dls. In *Description Logics*, 2004.

[38] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[39] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, pages 237–248, 2003.

[40] B. Choi, G. Cong, W. Fan, and S. Viglas. Updating recursive XML views of relations. *J. Comput. Sci. Technol.*, 23(4):516–537, 2008.

[41] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *IPTPS*, pages 33–44, 2003.

[42] N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[43] A. Deutsch, M. F. Fernández, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD Conference*, pages 431–442, 1999.

[44] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[45] I. Elghandour, A. Aboulnaga, D. C. Zilio, and C. Zuzarte. Recommending XML table views for XQuery workloads. In *XSym workshop*, 2009.

[46] D. Florescu, A. Grünhagen, and D. Kossmann. XL: an XML programming language for web service specification and composition. In *WWW*, pages 65–76, 2002.

[47] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB J.*, 13(3):294–315, 2004.

[48] D. Florescu and D. Kossmann. Storing and querying XML data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[49] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating data sources in large distributed systems. In *VLDB*, pages 874–885, 2003.

[50] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[51] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[52] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *CoRR*, abs/cs/0502008, 2005.

[53] R. Hayek, G. Raschia, P. Valduriez, and N. Mouaddib. Summary management in P2P systems. In *EDBT*, pages 16–25, 2008.

[54] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, pages 511–524, 2008.

[55] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, pages 321–332, 2003.

[56] ISO/IEC 9075-14:2003, information technology – database languages – sql – part 14: XML-related specifications (SQL/XML), 2003.

[57] H. Jiang, H. Lu, W. Wang, and J. X. Yu. XParent: An efficient RDBMS-based XML database system. In *ICDE*, pages 335–336, 2002.

[58] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*, pages 133–144, 2002.

[59] S. Kepser. A simple proof for the Turing-completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.

[60] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[61] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, pages 193–204, 2005.

[62] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha. The repeatability experiment of SIGMOD 2008. *SIGMOD Record*, 37(1):39–45, 2008.

[63] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-driven design and deployment of service-enabled web applications. *ACM Trans. Internet Techn.*, 5(3):439–479, 2005.

[64] I. Manolescu, D. Florescu, and D. Kossmann. Answering xml queries on heterogeneous data sources. In *VLDB*, pages 241–250, 2001.

[65] I. Manolescu, Y. Papakonstantinou, and V. Vassalos. XML tuple algebra. To appear in Encyclopedia of Database Systems (Springer), 2008.

[66] I. Manolescu and S. Zoupanos. Materialized views for P2P XML warehousing. Submitted for publication, available at http://vip2p.saclay.inria.fr/paper.pdf, 2009.

[67] I. Manolescu and S. Zoupanos. XML materialized views in P2P. DataX (International Workshop on Database Technologies for XML) workshop (not in the proceedings), 2009.

[68] P. Michiels, I. Manolescu, and C. Miachon. Toward microbenchmarking xquery. *Inf. Syst.*, 33(2):182–202, 2008.

[69] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[70] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional XML data. In *SIGMOD Conference*, pages 289–300, 2003.

[71] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, 2009.

[72] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[73] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly XML node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[74] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD Conference*, pages 443–454, 2006.

[75] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

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

[77] F. Pentaris and Y. E. Ioannidis. Query optimization in distributed networks of autonomous database systems. *ACM Trans. Database Syst.*, 31(2):537–583, 2006.

[78] N. Preda. *Efficient Web resource management in structured peer-to-peer networks*. PhD thesis, Université de Paris XI, 2008.

[79] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD Conference*, pages 134–144, 2003.

[80] P. Rao and B. Moon. An internet-scale service for publishing and locating XML documents. In *ICDE*, pages 1459–1462, 2009.

[81] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[82] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, pages 188–201, 2001.

[83] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. Maintaining xpath views in loosely coupled systems. In *VLDB*, pages 583–594, 2006.

[84] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, pages 443–454, 2005.

[85] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[86] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[87] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[88] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[89] L. Sidirourgos, G. Kokkinidis, T. Dalamagas, V. Christophides, and T. K. Sellis. Indexing views to route queries in a PDMS. *Distributed and Parallel Databases*, 23(1):45–68, 2008.

[90] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of xpath queries with structured overlay networks. In *OTM Conferences (2)*, pages 1243–1260, 2005.

[91] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[92] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[93] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, 2009.

[94] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, pages 873–882, 2008.

[95] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz. Materialized view selection in XML databases. In *DASFAA*, pages 616–630, 2009.

[96] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[97] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with disco. *IEEE Trans. Knowl. Data Eng.*, 10(5):808–823, 1998.

[98] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB J.*, 5(2):101–118, 1996.

[99] M. Weis and I. Manolescu. Declarative XML data cleaning with XClean. In *CAiSE*, pages 96–110, 2007.

[100] M. Weis and I. Manolescu. Xclean in action (demo). In *CIDR*, pages 259–262, 2007.

[101] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[102] Shiri: Système hybride d'intégration pour la recherche d'information dans les ressources. http://www.lri.fr/contrat.en.cours.php?con=11, 2007.

[103] Views in Peer-to-Peer. Project web site available at http://vip2p.saclay.inria.fr, 2009.

[104] WSDL: Web Services Definition Language 1.1. Available at http://www.w3.org/wsdl, 2001.

[105] xml:id Version 1.0. Available at http://www.w3.org/TR/xml-id/, 2005.

[106] Xml schema. Available at http://www.w3.org/XML/Schema, 2004.

[107] XML Path Language (XPath) Version 1.0. Available at http://www.w3.org/TR/xpath, 1999.

[108] XQuery 1.0 and XPath 2.0 Data Model (XDM). Available at http://www.w3.org/TR/xpath-datamodel/, 2007.

[109] XSLT 2.0 and XQuery 1.0 Serialization. Available at http://www.w3.org/TR/xslt-xquery-serialization/, 2007.

[110] XQuery 1.0: An XML Query Language. Available at http://www.w3.org/TR/xquery/, 2007.

[111] XQuery 1.0 and XPath 2.0 Functions and Operators. Available at http://www.w3.org/TR/xpath-functions/, 2004.

[112] XQuery Update Facility 1.0. Available at http://www.w3.org/TR/xquery-update-10/, 2008.

[113] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, pages 121–132, 2005.

[114] The Bamboo distributed hash table - a robust, open-source DHT. Available at http://bamboo-dht.org, 2005.

[115] Codex project: Efficiency, dynamicity and composition for XML. models, algorithms and systems. Project web site available at http://codex.saclay.inria.fr.

[116] The eDos project. Project web site available at http://www.edos-project.org.

[117] Grid5000, a distributed research network. Project web site available at: https://www.grid5000.fr, 2009.

[118] Archivage ouvert du Web. Available at http://hal.archives-ouvertes.fr.

[119] Pubzone: Scientific publication discussion forum. Available at http://www.pubzone.org, 2009.

[120] WebContent, the Semantic Web platform. Project web site available at http://www.webcontent.fr.

[121] Y. Zhang and P. A. Boncz. XRPC: distributed XQuery and update processing with heterogeneous XQuery engines. In *SIGMOD Conference*, pages 1331–1336, 2008.

[122] Y. Zhang, N. Tang, and P. A. Boncz. Efficient distribution of full-fledged XQuery. In *ICDE*, pages 565–576, 2009.