# THESE

*Pour obtenir le grade de*

**DOCTEUR DE L'UNIVERSITE PARIS SUD**

*Discipline : Informatique*

*date de soutenance prévue: 6 Decembre 2007*
*par*

Andrei ARION

# XML Acces Modules: Towards Physical Data Independence in XML Databases

*Jury*

| | | | |
|---|---|---|---|
| **Rapporteurs:** | Sihem | Amer-Yahia | Yahoo Research |
| | Patrick | Valduriez | INRIA-Rennes |
| **Examinateurs:** | Nicole | Bidoit | Université Paris Sud |
| | Philippe | Pucheral | INRIA-Rocquencourt |
| **Directeurs de thèse:** | Véronique | Benzaken | Université Paris Sud |
| | Ioana | Manolescu | INRIA Futurs |

# Abstract

The purpose of this thesis is to design a framework for achieving the goal of physical data independence in XML databases. We first propose the XML Access Modules - a rich tree pattern language featuring multiple returned nodes, nesting, structural identifiers and optional nodes, and we show how it can be used to uniformely describe a large set of XML storage schemes, indices and materialized views.

Next, we study the problem of XQuery rewriting using XML Access Modules. In a first step we present an algorithm to extract XML Access Modules patterns from XQuery and we show that the patterns we identify are strictly larger than in previous works, and in particular may span over nested XQuery blocks.

We characterize the complexity of tree pattern containment (which is a key sub-problem of rewriting) and rewriting itself, under the constraints expressed by a structural summary, whose enhanced form also entails integrity constraints. We also use structural identifiers to enhance the rewriting opportunities.

We have implemented our framework in the ULoad prototype [13] and we evaluate the performance of our approach for query containment and rewriting.

**Keywords:** XML, physical data independence, XQuery materialized views, tree pattern containment and rewriting, path summary

# Introduction

A key factor for the outstanding success of database management systems is *physical data independence*: queries, and application programs, are able to refer to the data at the logical level, ignoring the details on how the data is physically stored and accessed by the system. The corner stone of implementing physical data independence is an *access path selection algorithm*: whenever a disk-resident data item can be accessed in several ways, the access path selection algorithm, which is part of the query optimizer, will identify the possible alternatives, and choose the one likely to provide the best performance for a given query [104].

Surprisingly, physical data independence is not yet achieved by XML database management systems (*XDBMS*s, in short). Numerous methods have been proposed for XML storage, labeling and indexing and implemented in various prototypes. However, the data layout resulting from each of these schemes is hard-coded within the query optimizer of the corresponding system. Thus, adding a different type of storage structure (e.g., a new index) requires rewriting the query optimizer, to inform it that a new access path becomes available. This situation prevents XDBMSs from attaining two important features: flexibility and extensibility. By *flexibility*, we mean that widely different storage schemes must be supported, for the varying needs of different workloads and data sets. By *extensibility*, we mean that the XDBMS must adapt gracefully to changes in the workload and/or data set, which naturally require tuning the storage by adding e.g., an index or a materialized view. Such extensibility is a common feature of current relational database management systems (RDBMSs), endowed with automatic index and materialized view selection [5].

The work described in this thesis aims at acheiving the goal of physical data independence in XML databases. We outline next the contributions made in this thesis and set the roadmap followed by the manuscript.

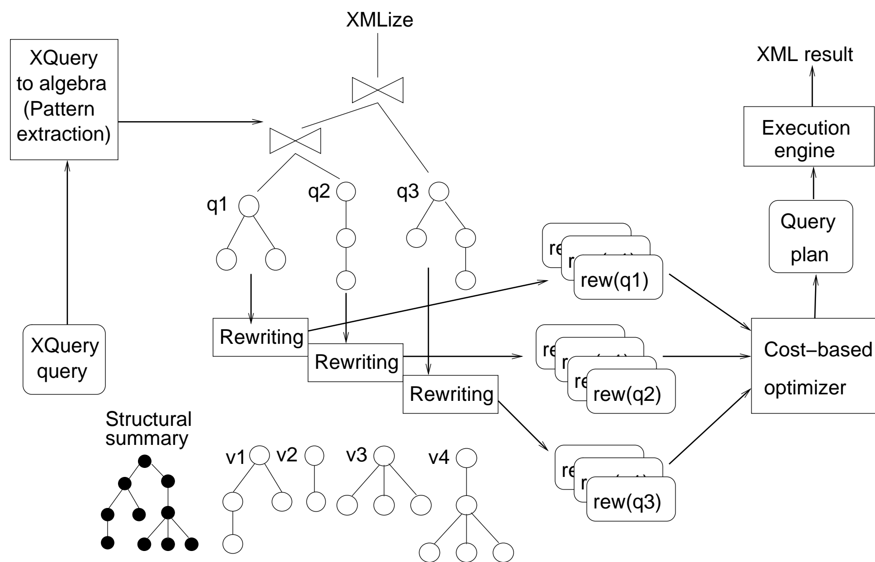



Figure 1: Outline of our query processing approach.

# Contributions and outline

**Uniform description for XML stores, indices and views: XML Access Modules** A first difficulty to overcame is to provide an uniform description of persistent storage structures (storage modules, indices and materialized views) to the optimizer. The language used for this description has to accomodate a variety of features. The degree of fragmentation may go from very low (blob storage) to very high ( node-oriented storage). The clustering criteria may be very simple, e.g., cluster nodes by their name [48], or very complex, e.g., cluster nodes connected by a complex path expression [68, 86]. The clustering criteria may be derived from the document's schema, content, a workload, or a combination thereof. Particular applications may not need to use all the above spectrum of choices. However, just like relational database systems, XML Database Management Systems (XDBMSs) should be able to support different XML data sets and query workloads.

The first contribution of this thesis is to define a tree pattern language - the XML Access Modules (or XAMs for short). A XAM describes, in an algebraic-style formalism, the information contained in a persistent XML storage structure, which may be a storage module, an index, or a materialized view. The set of XAMs describing the storage is used by the optimizer to build data access plans. Using XAMs, a change to the storage (adding or removing a storage structure) is communicated to the optimizer

simply by updating the XAM set.

The XAM language has been introduced in [11]. Chapter 2 focuses on the presentation of this language: first we review the features of persistent XML storage structures that it must handle (Section 2.1) and we introduce its syntax and semantics (Section 2.2). Then we evaluate its expressive power by showing how it can model existing storage, indices and materialized views (Section 2.3).

**Query containment and rewriting in the presence of path summaries** We study the problem of answering queries under path summary constraints using XAM views.

- *Extracting XAMs from XQueries* To take advantage of our tree pattern-shaped materialized views, one has to understand which views can be used for an XQuery query $q$. This process can be seen as a translating $q$ to some *query patterns* $p_{q1}, \ldots, p_{qn}$, followed by a rewriting of every query pattern $p_{qi}$ using the view patterns $p_{v1}, \ldots, p_{vm}$. The first step (query-to-pattern translation) is crucial. Intuitively, the bigger the query patterns, the bigger the view(s) that can be used to rewrite them, thus the less computations remain to be applied on top of the views.

  An important contribution of this thesis is an algorithm identifying tree patterns in queries expressed in a large XQuery subset. The advantage of this method is that the patterns we identify are strictly larger than in previous works, and in particular may span over nested XQuery blocks, which was not the case in previous approaches. We ground our algorithm on an algebra, since the translation is quite complex due to XQuery complexity, and straightforward translation methods may loose the subtle semantic relationships between a pattern and a query.

  The pattern extraction algorithm was introduced in [12]. We present it in Chapter 3 of this thesis.

- *XAM containment and rewriting using XAM views*

  Our approach for rewriting relies on finding for each query tree pattern an algebraic plan, built on the materialized views, equivalent to the pattern under a specific sets of constraints over the queried document. More specifically, we use path summaries (or DataGuides [52]) as sources of structural constraints, describing the structure of the document. We provide a correct and complete algorithm for rewriting XAM queries based on XAM materialized views in Chapter 5. An important contribution of this algorithm is that it exploits interesting ID properties for rewriting.

A necessary ingredient for the rewriting process is an algorithm for judging whether two XAM patterns are equivalent under summary constraints. We decide equivalence by checking that containment holds both ways. Chapter 4 is devoted to the containment decision under summary constraints.

The containment and rewriting algorithms have been published in [16].

We have implemented our approach for physical data independence in the ULoad [13] prototype, and we report in Sections 4.6 and 5.6 on their practical performance.

In [15] we study the problem of tree pattern minimization using path summaries. This topic, although related, is not directly in the scope of this thesis. We will only briefly discuss it in Section 4.5 because it enables an interesting comparison of tree pattern minimization under different types of constraints.

# Contents

**1 Persistent XML databases** **1**

1.1 Preliminaries: XML Data Model . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Generic architecture of a persistent XML database . . . . . . . . . . . 3

1.2.1 Stored data structures . . . . . . . . . . . . . . . . . . . . . . 3

1.2.2 A logical algebra for XML processing . . . . . . . . . . . . . 9

1.2.3 Execution engine . . . . . . . . . . . . . . . . . . . . . . . . 13

1.2.4 Query analyzer . . . . . . . . . . . . . . . . . . . . . . . . . 14

**2 XML Access Modules: describing persistent XML storage structures** **17**

2.1 Persistent storage structures for XML databases . . . . . . . . . . . . 17

2.1.1 XML storage models . . . . . . . . . . . . . . . . . . . . . . 18

2.1.2 XML indexing models . . . . . . . . . . . . . . . . . . . . . 25

2.1.3 XML materialized views . . . . . . . . . . . . . . . . . . . . 27

2.1.4 Summary: interesting features in XML persistent storage structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.2 XML Access Modules syntax and semantics . . . . . . . . . . . . . . 33

2.2.1 XAM syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.2.2 XAM semantics . . . . . . . . . . . . . . . . . . . . . . . . . 36

2.3 XAM expressive power . . . . . . . . . . . . . . . . . . . . . . . . . 45

2.3.1 Modeling relational XML stores . . . . . . . . . . . . . . . . 46

2.3.2 Modeling native XML stores . . . . . . . . . . . . . . . . . . 49

2.3.3 Modeling XML indices . . . . . . . . . . . . . . . . . . . . . . 51
2.3.4 XAM limitations . . . . . . . . . . . . . . . . . . . . . . . . . 56

**3 From XQuery to XML Access Modules** **59**
3.1 Motivating example . . . . . . . . . . . . . . . . . . . . . . . . . . 60
3.2 Query language . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.3 Pattern extraction algorithm . . . . . . . . . . . . . . . . . . . . . . 63
3.3.1 Algebraic translation of path queries . . . . . . . . . . . . . . 64
3.3.2 Algebraic translation of more complex queries . . . . . . . . 66
3.3.3 Isolating patterns from algebraic expressions . . . . . . . . . 70

**4 XAM containment under path summary constraints** **71**
4.1 Alternative XAM semantics based on embeddings . . . . . . . . . . . 71
4.2 Structural summaries . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.2.1 XML path summaries . . . . . . . . . . . . . . . . . . . . . . 77
4.2.2 Complex summaries . . . . . . . . . . . . . . . . . . . . . . . 79
4.3 Canonical models for patterns . . . . . . . . . . . . . . . . . . . . . 80
4.3.1 Canonical model of a conjunctive pattern . . . . . . . . . . . 80
4.3.2 Canonical model of complex patterns . . . . . . . . . . . . . 83
4.4 Pattern containment . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.4.1 Conjunctive patterns . . . . . . . . . . . . . . . . . . . . . . . 86
4.4.2 Decorated patterns . . . . . . . . . . . . . . . . . . . . . . . . 88
4.4.3 Optional pattern edges . . . . . . . . . . . . . . . . . . . . . . 89
4.4.4 Attribute patterns . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.4.5 Nested patterns . . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.5 Pattern minimization under summary constraints . . . . . . . . . . . 91
4.6 Experimental evaluation of XAM containment . . . . . . . . . . . . . 93

**5 Rewriting XQuery queries using XAM views** **97**

5.1 Overview . . . . . . . . . . 97
5.2 Motivating example . . . . . . . . . . 98
5.3 Summary-based rewriting of conjunctive patterns . . . . . . . . . . 101
5.4 Extending rewriting . . . . . . . . . . 106
5.5 Building equivalent plan-pattern pairs . . . . . . . . . . 108
5.5.1 Examples and problem statement . . . . . . . . . . 108
5.5.2 Computing the pattern equivalent to a join plan . . . . . . . . . . 110
5.6 Experimental evaluation of XAM rewriting . . . . . . . . . . 114

**6 Related works 119**
6.1 Materialized views for XML . . . . . . . . . . 119
6.2 Algebras for XML . . . . . . . . . . 120
6.3 Extracting tree patterns from XQuery . . . . . . . . . . 121
6.4 Containment and rewriting under integrity constraints . . . . . . . . . . 122
6.5 Constraint-based tree pattern minimization . . . . . . . . . . 125

**7 Conclusion and perspectives 127**

# Chapter 1

# Persistent XML databases

We consider the problem of processing XML queries on a database of XML documents which have been previously stored. In this chapter, we briefly discuss the main modules involved in storing XML documents and query them subsequently. The purpose of the chapter is to set the stage for the remainder of the thesis by introducing some fundamental notions. Section 1.1 presents the XML data model.

In Section 1.2 we sketch the generic architecture of an XML storage and query processing engine, and for each building block, we illustrate some of the specific solutions and tradeoffs.

## 1.1 Preliminaries: XML Data Model

We view an XML document as a tree $(\mathcal{N}, \mathcal{E})$, such that $\mathcal{N} = \mathcal{N}_d \cup \mathcal{N}_e \cup \mathcal{N}_a$, where $\mathcal{N}_d$ is a set consisting of exactly one document node, $\mathcal{N}_e$ is a non-empty set of element nodes and $\mathcal{N}_a$ is a set of attribute nodes. The set of edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ respects the restrictions imposed by the XML specification [116]. Thus, the node in $\mathcal{N}_d$ is the tree root and it has exactly one child which belongs to $\mathcal{N}_e$. The parent of any node in $\mathcal{N}_e \cup \mathcal{N}_a$ belongs to $\mathcal{N}_e$. For ease of exposition, in the sequel we will ignore the document node and refer to the unique $\mathcal{N}_e$ child of the document node as the document's root.

Let $n_1, n_2$ be two XML nodes. We denote the fact that $n_1$ is $n_2$'s parent as $n_1 \prec n_2$, and the fact that $n_1$ is an ancestor of $n_2$ as $n_1 \prec\!\!\prec n_2$.

All $\mathcal{N}$ nodes are endowed with a unique *identity* and with a *node label*, viewed as a string. Furthermore, all $\mathcal{N}$ nodes have a *value*. All values belong to $\mathcal{A}$, the set of atomic

values. The value of an element node is the result of applying the XPath function *text()* on the node [121]. If an element has multiple text descendant nodes, the *text()* function concatenates them, losing the information about their number and relative order in the document. To avoid this, a simple extension to our model consists of making text nodes first-class $\mathcal{N}$ citizens, endowed with identity. To keep our presentation simple, we omit this in the sequel.

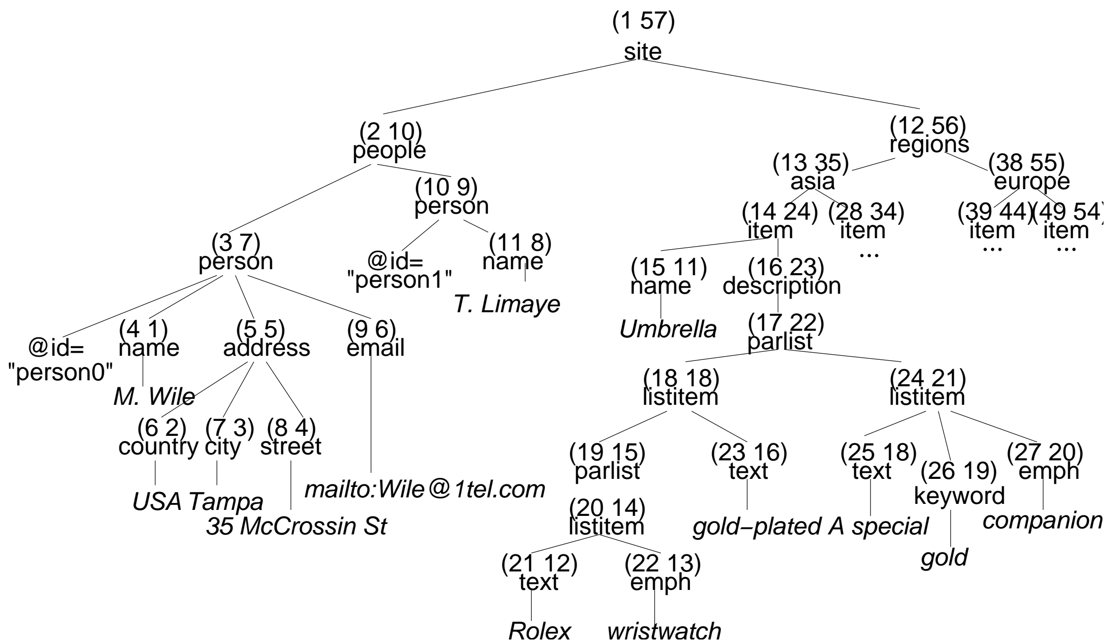



Figure 1.1: A sample XMark document

The *content* of a node $n$ is a string value, obtained by serializing the labels and values of all nodes from the tree rooted in $n$, in a top-down, left-to-right traversal.

**Types** The standard XML query language XQuery [121] is defined as operating over instances of XQuery Data Model [119]. According to XQuery semantics [118], whether an XML Schema of the input documents is available or not, some simple or complex type information is always attached to data model instances, whether nodes, values, or lists, manipulated by a query [121]. Unlike previous XML query languages such as Lorel [3], Quilt [31], UnQL [30], X-OQL [6], XQuery semantics is strongly influenced by types:

1. The query processor may reject ill-typed queries.

2. The result of a value-based comparison among nodes, values and/or lists depends on the operands' types, and the types to which they will be cast prior to the comparison [118].

3. Several XQuery constructs, such as instance of, typeswitch, cast, explicitly manipulate expression types.

We model the type information associated to an XML document as a function $\tau$ assigning to any node $n \in N_d$ a type $\tau(n) \in \mathcal{T}$, where $\mathcal{T}$ is the set of value and node types considered in XQuery's type system [119]. The types assigned by $\tau$ are consistent with the dynamic typing reasoning from [118].

Figure 1.1 depicts a sample XML document, which is a simplified instance of an XMark [115] benchmark document. In this figure, nodes whose label appears in normal fonts are elements. Nodes whose label is of the form $@a = v$ are attribute nodes whose name is $a$ and whose value is $v$. Leaf nodes in italic fonts are text nodes. Above each element node, a pair of integers is used as an identifier for the node. We will discuss the meaning of such IDs in the sequel.

## 1.2 Generic architecture of a persistent XML database

In this section we outline the general architecture of an XML query processor based on a persistent store. We then describe in more details each layer of the architecture.

A simplified view of the XML query processing steps is depicted in Figure 1.2. To process a query expressed in some query language, first, data access plans are built by examining all the possible access paths (e.g. index scan, sequential scan) of the existing storage structures, indices and views. Then, these plans are combined with the help of algebraic operators (such as selections, joins etc.) into increasingly larger plans, aiming towards building plans for the complete query. At this step several join orderings and equivalent algebraic alternatives are developed. Then, the best logical plan is translated into a physical plan, and the latter is executed inside the execution engine, possibly accessing stored data structures.

### 1.2.1 Stored data structures

In this section we present some important notions and techniques related to the storage of XML data in persistent stores.

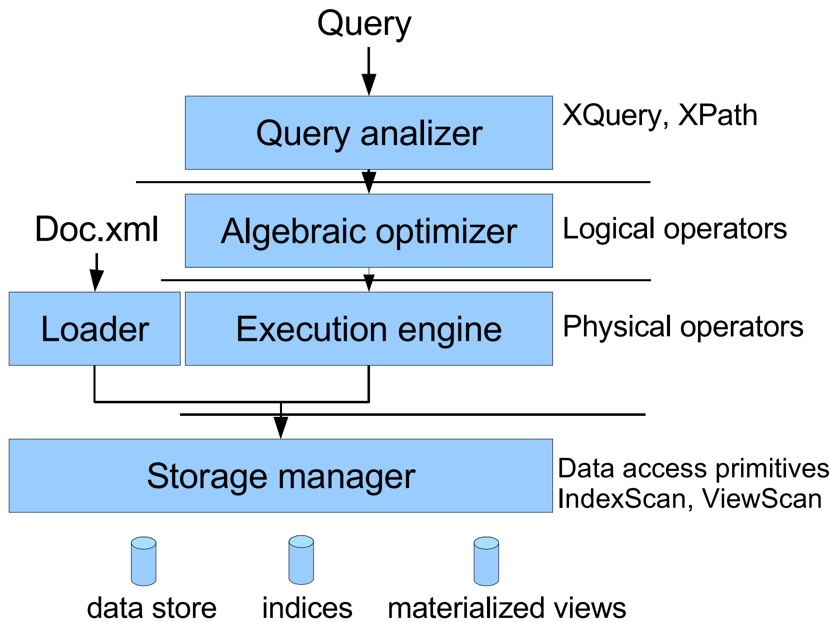



Figure 1.2: General architecture of a persistent XML database

### Structural identifiers for XML nodes

To represent node identity, XML databases use *persistent identifiers*, which are values uniquely identifying a node within a document (or, more generally, within the whole database). We can view node identifiers (or, in short, *IDs*) as values obtained by applying an injective function (or labeling function) $f : \mathcal{N} \rightarrow \mathcal{A}$. We say the identifiers assigned by a given function $f$ are *structural* if, for any nodes $n_1, n_2$ belonging to the same document, we may decide, by comparing $f(n_1)$ and $f(n_2)$, whether $n_1$ is a parent/ancestor of $n_2$ or not.

The ancestor $\prec\!\!\prec$ and parent $\prec$ predicates can be easily extended to structural identifiers. Thus, we write $f(n) \prec\!\!\prec f(m)$ to signify that the structural identifier $f(n)$ corresponds to an ancestor of the node $m$, and similarly for the predicate $\prec$.

A very popular structural identifier scheme is based on tree traversals, as follows:

- traverse the XML tree in pre-order and assign increasing integer labels $1, 2 \ldots 3$ etc. to each encountered node; we call this integer the $pre$ label.
- traverse the XML tree in post-order and similarly assign increasing integer labels to each encountered node; we call this the $post$ label.

- assign to each node a $depth$ label corresponding to its depth in the tree. Thus, the $depth$ of the root is 1, the $depth$ label of the root's children nodes is 2 etc.
- the identifier of each node in the XML document is the triple of its $(pre, post, depth)$ labels.

For example, Figure 1.1 depicts $(pre, post)$ IDs above the elements. The $depth$ number is omitted in the figure to avoid clutter. This scheme was introduced in [7] for XML documents and often used subsequently [34, 59].

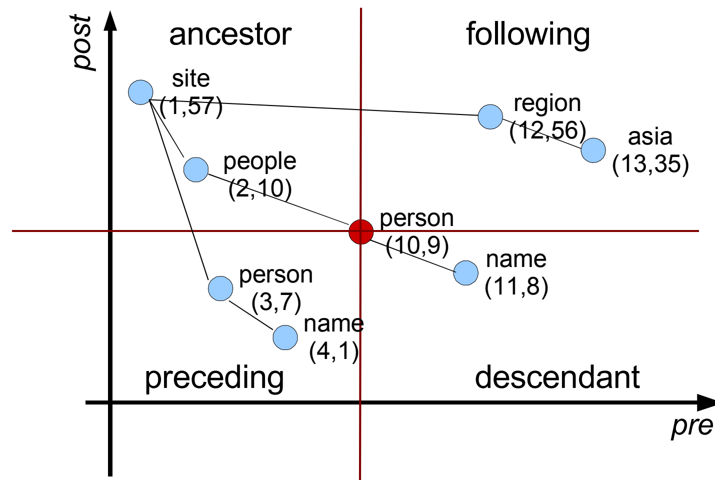



Figure 1.3: A fragment of the pre-post plane for the XMark document.

Given an XML document $d$ and two of its elements $n$ labeled $(pre_n, post_n, depth_n)$ and $m$ labeled $(pre_m, post_m, depth_m)$ we can decide whether $m$ is an ancestor / descendant or preceds/follows the n element using simple comparisons of the identifiers:

- $m$ is a *descendant* of $n$ if and only if $pre_n < pre_m < post_m < post_n$
- $m$ is a *child* of $n$ if and only if $pre_n < pre_m < post_m < post_n$ and, furthermore $depth_n + 1 = depth_m$
- $m$ is an *ancestor* of $n$ if and only if $pre_m < pre_n < post_n < post_m$
- $m$ is the *parent* of $n$ if and only if $pre_m < pre_n < post_n < post_m$ and, furthermore $depth_n = depth_m + 1$
- $m$ precedes the $n$ element if and only if $post_m < pre_n$

- $m$ follows the $n$ element if and only if $post_n < pre_m$

These structural properties induce a partitioning of the nodes from the document $d$ in a so called *pre-post plane* [55] as the following example shows.

EXAMPLE 1.2.1. For the document depicted in Figure 1.1 we represent in the Figure 1.3 a fragment of its *pre-post plane*. As this figure shows, the descendants of the $person$ element identified by $(10, 9)$ can be found at the right, and under, the $(10, 9)$ node. Its following nodes appear at the right and above the level of $(10, 9)$ node. Similarly, the ancestors can be found in the top left part of the plane, while its preceding nodes apear in the bottom left quarter of the plan.

$\triangleleft$

**Navigational structural identifiers** More advanced structural IDs such as DeweyIDs [107] or ORDPATHs [90] allow us to directly compute structural identifiers for ancestor/descendant nodes using the IDs of a given node. We term the structural identifiers that have this property *navigational structural identifiers* and we demonstrate their use for query rewriting in Section 4.4.

**Structural identifiers and updates** One of the drawbacks of using structural identifiers in XML query processing was that the initially proposed labeling schemes were not behaving well in the presence of updates. Adding or removing nodes from the XML tree lead to re-computing the structural identifiers for a large part of the document. This poses performance problems and in some cases severely limits the degree of concurrent updates supported. Recent works [90, 26] propose new labeling schemes and techniques that are also update-resilient and thus eliminate this drawback.

In this thesis, we sometimes rely on structural identifiers for query processing. The work we present, however, is not bound to a given structural identifier model. Moreover, we focus on the performance of read-only queries, and we do not address updates (which are part of the ongoing work beyond this thesis).

**XML storage, indexing and materialized views**

Since the advent of XML numerous approaches have been proposed for storing the information content of an XML document in a persistent database. We briefly outline the main classes of storage solutions. The purpose of this section is not to be exhaustive, but rather to give an overview of existing trends. We explore this topic in more detail in Section 2.1.

**Node based stores** Even before XML caught the attention of database researchers, semi-structured data stores were built for OEM data [92]. The idea is to store XML documents as trees, simply by adding to the store one entry per node, together with some pointers to its child node entries. Natix [67] was the first such native store developed for XML. A more recent node oriented store was described in NOK [123]. Node-based stores typically cluster together related nodes (e.g. a parent and its children) in order to facilitate path traversals.

**Relational stores** Many works have proposed storing XML in relational databases in order to take advantage of the robustness of the available commercial systems. Such solutions suppose that XML queries are translated into “relational“ queries that relational database management systems (RDBMSs) can process. The first proposal dates from 1998 [48] and it investigates the possibility of storing one relational tuple for every XML node into the so called $Edge$-relation. The work also advocates materializing other relational tables defined by select-project-join expressions over $Edge$, to speed up query processing. Subsequent works [23, 39, 105] have focused on choosing the structure of the relations based on documents schema descriptions and/or frequent data patterns.

A particular family of works [122, 66] have proposed storing in relational attributes structural identifiers and XML path information. These paths and identifiers are used both by the query translation process and by the SQL query execution stage.

The MonetDB/XQuery system [25, 55, 108] is the first to propose a full algebraic based query translation methodology from the full XQuery language to SQL. This translation is based on the special properties of the $pre - post$ plane illustrated in Figure 1.2.1.

**Mixed stores** XML documents have varied structures, which lead to the idea that some XML data is best stored in regular relational format, whereas some other documents are best served by a contiguos (node-oriented) store. To accomodate documents having both regular and unstructured parts, *mixed storages* were proposed in some early research prototypes [42, 81]. In these works, fragments of a document may be stored in different data structures, even in different databases. While this has the potential for compact storage and efficient processing, it also complicates the query translation process.

In the industrial community, the need to seamlessly combine relational and XML data lead to the SQL/XML proposal [62]. This language allows retrieving XML results out of a relational store, as well as storing XML fragments in relational tables (XML

has become a new native atomic data type).

**XML indexes** A first category of index proposals for XML are the *structural indexes*; using an idea initially proposed for Object Oriented Databases([70]), this type of indexing relies on local structural properties to group together nodes. For example, path indexes [21] use a strong dataguide to index all the nodes which are on the same root to node path. T(k) [86] indexes generalize this approach by creating equivalence classes for nodes that are indistinguishable w.r.t. a class of paths defined by path templates. Other structural indexing methods (e.g. A(k)-indices [69], D(k) Index [98]) use approximate dataguides [53] and consider incoming paths of bounded length as indexing property.

A second category of indexes for XML are based on structural identifiers. Using variations of the (pre,post) structural identifier scheme, XISS [77] and XPath Accelerator [55] map all element and attribute nodes onto the pre-post plane and rely on structural joins to match path queries.

Annother category of XML indexes are the *sequence based indexes* [37, 99, 112, 113]. The basic idea behind sequence based indexes is to encode both the query and the XML document as strings. Then, in order to compute the answer to the query, subsequence matching techniques on the string-encoded representations are used. For example, Index Fabric [37, 68] uses multi-level persistent Patricia tries in order to speed up the execution of branching queries.

When the structure of XML data is not known by the users, *keyword-based indexing* techniques using the notion of proximity search, and implemented by inverted files can be used to find the most relevant result of keyword queries [49, 57, 60].

**Materialized views** One performance-enhancing technique in XQuery processing is the usage of materialized views. The idea is to pre-compute and store in the database the result of some queries (commonly called *view definitions*), and when a user query arrives, to identify which parts of the query match one of the pre-computed views. The larger parts of the query one can match with a view, the more efficient query processing will be, since a bigger part of the query computation can be obtained directly from the materialized view.

Identifying useful views for a query requires reasoning about containment (e.g., is all the data in view $v$ contained in the result of query $q$ ?) and equivalence (e.g., is the join of views $v_1$ and $v_2$ equivalent to the query $q$ ?). XML query containment and equivalence are well understood when views and queries are represented as *tree*

*patterns*, containing tuples of elements satisfying specific structural relationships [85, 89]. Moreover, popular XML indexing and fragmentation strategies also materialize tree patterns [37, 42, 66, 68]. Therefore, tree patterns are an interesting model for XML materialized views [10, 13, 22, 42, 63].

**Summary** Observe that, whatever the chosen storage model, indices and materialized views, the problem of answering an XML query is a version of the classical access path selection problem and is akin to the problem of query rewriting using materialized views. This is also the angle under which we study this problem in this thesis.

### 1.2.2 A logical algebra for XML processing

We assume available an ID scheme $I$, that is, an injective function assigning to every node a value in $\mathcal{I}$.

Our data model is an instance of a nested relational data model [1, 2], enhanced with order, and further specialized to our setting. This data model features:

- a set of atomic data types $\mathcal{A}$, such as String, integer etc.
- the tuple constructor, denoted $(\cdot)$;
- the set constructor $\{\cdot\}$, the list constructor $[\cdot]$ and the bag constructor $\{\!\{\cdot\}\!\}$.

The value of a tuple attribute is either a value from $\mathcal{A}$, or null ($\perp$), or a collection (set, list or bag) of homogeneous tuples. Notice the alternation between the tuple and the collection constructors. Thus, the model allows nesting of tuples and sets/lists, but only in alternation. This model is well-adapted to the hierarchical, ordered structure of XML data, and conceptually close to the XQuery data model [119].

We use lowercase letters for relation names, and uppercase letters for attribute names, as in $r(A_1, A_2(A_{21}, A_{22}))$. Values are designated by lowercase letters. For instance, a tuple in $r(A_1, A_2(A_{21}, A_{22}))$ may have the value $t(x_1, [(x_3, \perp)\ (x_4, x_5)])$.

To every nested relation $r$, corresponds a Scan operator, also denoted $r$, returning the (possibly nested) corresponding tuples. Other standard operators are the cartesian product $\times$, the union $\cup$ and the set difference $\setminus$ (which do not eliminate duplicates).

We consider predicates of the form $A_i\ \theta\ c$ or $A_i\ \theta\ A_j$, where $c$ is a constant. $\theta$ ranges over the comparators $\{=, \leq, \geq, <, >, \prec, \prec\!\!\prec\}$, and $\prec$, $\prec\!\!\prec$ only apply to $\mathcal{I}$ values.

Let $pred$ be a predicate over atomic attributes from $r$, or $r$ and $s$. Selections $\sigma_{pred}$ have the usual semantics. A join $r \bowtie_{pred} s$ is defined as $\sigma_{pred}(r \times s)$. For convenience, we will also use outerjoins $⟕_{pred}$ and semijoins $\ltimes_{pred}$ (although strictly speaking they are redundant to the algebra). Another set of redundant, yet useful operators, are *nested joins*, denoted $\bowtie^n_{pred}$, and *nested outerjoins*, denoted $⟕^n_{pred}$, with the following semantics:

$$r ⟕^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\}) \mid t_1 \in r\}$$
$$r \bowtie^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\}\ ) \mid t_1 \in r, \{t_2 \in s \mid pred(t_1, t_2)\} \neq \emptyset\ \}$$

An interesting class of logical join operators (resp. nested joins, outerjoins, nested outerjoins, or semijoins) is obtained when the predicate's comparator is $\prec$ or $\prec\!\!\prec$, and the operand attributes are identifiers from $\mathcal{I}$. Such operators are called *structural joins*. Observe that we only refer to *logical structural joins*, independently of any physical implementation algorithm; different algorithms can be devised [7, 28]. Since structural joins play a central role in our work, we include their formal definition here.

DEFINITION 1.2.1 (STRUCTURAL JOINS). Let $R$ and $S$ be tuple sets, and $R.x$ and $S.y$ be attributes of type $ID$. For a given tuple $t_R \in R$, let *child*$(t_R.x, S.y)$ be the set of tuples in $S$ whose $y$ attribute is a child of $t_R.x$.

The parent-child structural join of $R$ and $S$, $R \bowtie^{\prec} S$, is:

$$\bigcup_{t_R \in R} \{t_R \| t_S \mid t_R \in R, t_S \in S, t_S \in \textit{child}(t_R.x, S.y)\}$$

where $\|$ stands for tuple concatenation.

The parent-child structural semijoin of $R$ and $S$, $R \ltimes^{\prec} S$, is:

$$\{t_R \in R \mid \textit{child}(t_R.x, S.y) \neq \emptyset\}$$

The parent-child structural outerjoin of $R$ and $S$, $R ⟕^{\prec} S$, is:

$$\begin{aligned} \textstyle\bigcup_{t_R \in R} \{t_R \| t_S \mid\ & t_R \in R, \textit{child}(t_R.x, S.y) \neq \emptyset, \\ & t_S \in \textit{child}(t_R.x, S.y)\} \ \cup \\ \{t_R \| \perp_{t_S} \mid\ & t_R \in R, \textit{child}(t_R.x, S.y) = \emptyset\} \end{aligned}$$

where $\perp_{t_S}$ denotes a tuple with $t_S$'s schema, and whose attributes are set to null ($\perp$).

When $R$ and $S$ are bags of tuples, the above definitions are modified to consider bag unions (which respect input cardinalities). Finally, when $R$ and $S$ are lists of tuples, *child*$(t_R.x, S.y)$ becomes a list respecting the order of the children in $S$, and the unions are replaced by list concatenation. Thus, the result is ordered, first, by $R$, and then by $S$ order. $\triangleleft$

Ancestor-descendant structural joins are similarly defined, using the set of descendants of $t_R.x$ in $S$ instead of the set of children. We omit the details. Notice that the above definitions also hold for the case when $R.x$ is nested within a tuple collection attribute.

As defined above, structural joins return flat tuples. Sometimes it is desirable to construct nested structural join results; to that purpose, nest structural join operators are introduced next[1].

DEFINITION 1.2.2 (NEST STRUCTURAL JOINS). Let $R$ and $S$ be two set of tuples, and $R.x$ and $S.y$ be two attributes of type $ID$. The nest parent-child structural join of $R$ and $S$, denoted as $R \bowtie_n^{\prec} S$, is:

$$\bigcup_{t_R \in R} \{t_R \| (s = \textit{child}(t_R.x, S.y)) \mid t_R \in R, \textit{child}(t_R.x, S.y) \neq \emptyset\}$$

In the above, we append to tuple $t_R$ a new attribute named s, whose value is the set of all $t_S$ tuples corresponding descendants ot $t_R$. The nest structural outerjoin of $R$ and $S$, $R \bowtie\!\sqsubset_n^{\prec} S$, is:

$$\bigcup_{t_R \in R} \{t_R \| (s = \textit{child}(t_R.x, S.y)) \mid t_R \in R\}$$

These definitions extend to the case when $R$ and $S$ are bags, respectively, lists, as in the case of joins. $\lhd$

Let $A_1, A_2, \ldots, A_k$ be some atomic $r$ attributes. A projection $\pi_{A1,A2,\ldots,Ak}(r)$ by default does not eliminate duplicates. Duplicate-eliminating projections are singled out by a superscript, as in $\pi^0$. The group-by operator $\gamma_{A_1,A_2,\ldots,A_k}$, and unnest $u_B$, where $B$ is a collection attribute, have the usual semantics [2].

We use the $map$ meta-operator to define algebraic operators which apply *inside* nested tuples. Let $op$ be a unary operator, $r.A_1.A_2.\ldots.A_{k-1}$ a collection attribute, and $r.A_1.A_2.\ldots.A_k$ an atomic attribute. Then, $map(op, r, A_1.A_2.\ldots.A_k)$ is a unary operator, and:

- If $k = 1$, $map(op, r, A_1.A_2.\ldots.A_k) = op(r)$.
- If $k > 1$, for every tuple $t \in r$:

[1]Nest structural joins have been mentioned also in [93] we formalized them here within our data model.

  - If for every collection $r' \in t.A_1$, $map(op, r', A_2.\ldots.A_k) = \emptyset$, $t$ is eliminated.
  - Otherwise, a tuple $t'$ is returned, obtained from $t$ by replacing every collection $r' \in r.A_1$ with $map(op, r', A_2.\ldots.A_k)$.

EXAMPLE 1.2.2. For instance, let $r(A_1(A_{11}, A_{12}), A_2)$ be a nested relation. Then, $map(\sigma_{=5}, r, A_1.A_{11})$ only returns those $r$ tuples $t$ for which *some* value in $t.A_1.A_{11}$ is 5 (existential semantics), and reduces these tuples accordingly. $\triangleleft$

$Map$ applies similarly to $\pi$, $\gamma$ and $u$. By a slight abuse of notation, we will refer to $map(op, r, A_1.A_2.\ldots.A_k)$ as $op_{A_1.A_2.\ldots.A_k}(r)$. For instance, the sample selection above will be denoted $\sigma_{A_1.A_{11}=5}(r)$. Binary operators are similarly extended, via $map$, to nested tuples. More formally, let $op$ be a binary operator, $r.A_1.A_2.\ldots.A_k$ be a collection attribute, and $s.B$ be an atomic attribute. Then, $map(op, r, s, A_1.A_2.\ldots.A_k, B)$ is a binary operator, and:

- if $k = 1$, $map(op, r, s, A_1.\ldots.A_k, B) = op(r, s, A_1, B)$.
- if $k > 1$, for every tuple $t \in r$
  - if for every collection $r' \in t.A_1$, $map(op, r', s, A_2.\ldots.A_k, B) = \emptyset$, $t$ is eliminated.
  - Otherwise, a tuple $t'$ is returned, obtained from $t$ by replacing every collection $r' \in r.A_1$ with $map(op, r', s, A_2.\ldots.A_k, B)$.

EXAMPLE 1.2.3. Let $r(A_1(A_{11}, A_{12}), A_2)$ and $s(B_1, B_2(B_{21}))$ two nested relations and $A_1.A_{12}$ and $B_1$ attributes of type ID. Then, $map(\bowtie_n^{\prec}, r, s, A_1.A_{12}, B_1)$ creates nested tuples with the signature $(A_1(A_{11}, A_{12}, (B_1, B_2(B_{21})), A_2))$ , by nesting tuples from $s$ inside $A_1.A_{12}$ attributes based on the $A_1.A_{12} \prec B_1$ relationship. $\triangleleft$

We assume available a *node creation function* $\nu$, which produces new nodes with fresh identity, given the node's possible children, and label.

The $xml_{templ}$ operator creates new XML nodes. For every (possibly nested) tuple $t$, whose data has already been grouped and structured, $xml_{templ}$ creates a new node by calling function $\nu$. The children of the new node have labels and content taken from the tuple $t$, and are structured as the template $templ$ describes.

EXAMPLE 1.2.4. Let us consider the following query which is to be evaluated on the XMark document in Figure 1.1:

*for $x in doc("XMark.xml")//item*
*return <res_item>*
*{$x/name},*
*for $y in $x/description*
*return <res_desc>{$y//listitem} </res_desc>*
*</res_item>*

We consider that the logical plan precomputed for the query has the signature $R(A_1(A_{11}))$ where $A_1$ and $A_{11}$ correspond to $name$, and respectively, $listitem$ elements. The tagging template $templ$ for the query is

*templ=<res_item>* $A_1$ *<res_desc>* $A_{11}$ *</res_desc></res_item>*

For each nested tuple form the result $xml_{templ}$ creates a new element $< res_item >$, then inserts the value for $A_1$. Next, for all nested children $A_{11}$ it creates a new $< res_desc >$ element filled with the value from $A_{11}$.

◁

### 1.2.3 Execution engine

We assume available a library of physical operators implementing the logical operators described in Section 1.2.2. For each logical operator *op*, we will use $op_\phi$ to designate its corresponding physical operator. Whenever one logical operator is implemented by several alternative physical operators, we will use superscripts to distinguish between the alternative physical operators. All operators apply on, and produce, potentially nested tuples.

Moreover, to each operator we associate an *order descriptor*, specifying the column(s) on which the operator output is ordered. For instance, assume operator *op*, with the output signature $(A_1, A_2(A_{21}, A_{22}), A_3)$. An order descriptor of the form $\downharpoonleft A_1 \downharpoonright$ specifies that *op* output is sorted in increasing order of $A_1$. Alternatively, an order descriptor of the form $\downharpoonleft A_3.A_{22} \downharpoonright$ specifies that the output is sorted by $A_3$, and then, inside each *op* tuple, the tuples in $A_2$ are sorted according to $A_{22}$. Order descriptors are used by the query optimizer to ensure physical operators are properly combined (see below).

The physical operators: $Scan_\phi$, $\sigma_\phi$, $\pi_\phi$, $\cup_\phi$ are implemented in a straightforward tuple based manner. The $Sort_\phi$ operator is based on a persistent B+ tree. A $GroupBy$ operator is implemented using a memory resident hash table.

Several physical operators are used to implement joins based on value predicates. More specifically, we consider nested loops joins and hash joins, the latter backed by a memory-resident hash table.

Of particular interest to us is the family of structural join operators. At its simplest form, when the join attributes are not nested inside their operands, the physical operators we implemented are $StackTreeDesc$ and $StackTreeAnc$ described in [7]. Both these algorithms require that the structural identifiers in their inputs be sorted in increasing order. However, $StackTreeDesc$ produces tuples ordered by the descendant element ID, whereas $StackTreeAnc$ produces tuples ordered by the ancestor element ID.

We have implemented structural outerjoins and structural semijoins as variations of the $StackTreeDesc$ algorithm. For what concerns nested structural joins (and as a matter of fact all nested joins) they are implemented using a simple iterative physical implementation of the $map$ operator.

Observe that the structural join family of operators has a special properties concerning the inputs and outputs order. This is why we had to rely on some relatively complex order descriptors, to make sure that join operators are correctly piped into each other.

For instance, consider again the example of operator $op_1(A_1, A_2(A_{21}, A_{22}), A_3)$ with the output descriptor $\downharpoonleft A_3 \downharpoonright$ and operator $op_2(B_1, B_2)$ with the output descriptor $\downharpoonleft B_1 \downharpoonright$. Assume, for simplicity, that all atomic attributes in $op_1$ and $op_2$ are structural identifiers. It is not possible to join $op_1$ and $op_2$ using the condition $A_1 \prec B_1$, because $op_1$'s output is not sorted by $A_1$. However, if $op_1$ order descriptor is $\downharpoonleft A_2.A_{21} \downharpoonright$, then it is possible to compute the nested structural join $op_1 \bowtie\!\sqsubset_{A_2.A_{21} \prec B_1} op_2$, given that the inputs are sorted on the right attributes.

The XML construction operator $xml_{templ,\phi}$ is implemented in a straightforward pipeline manner. It runs in constant time per constructed element; its memory needs are bounded by the size of the largest element to construct.

### 1.2.4 Query analyzer

The last module we consider is the query analyzer. This module parses the query and extracts a description in an internal format suited for optimization. This format may be the same as the logical algebra or may be an intermediary step between the query syntax and the algebra. In the realm of relational databases, the analyzer's output format is the the classical Query Graph Model (QGM) [58].

XQuery optimizers often use a form of XQuery graphs, whose basic ingredient is some form of *tree patterns* as in [9, 33, 40, 65, 87]. Tree patterns are convenient for several reasons. They capture the elementary navigation operations specific to XML and XML query languages. Their semantics is well-understood, based on the notion of tree embeddings [2]. Such clean semantics have allowed the obtention of important theoretical results concerning tree pattern minimization [9, 40], containment, and rewriting [41, 47, 85, 89, 114]. Incidentally, tree patterns are also the common abstraction for XML query cardinality estimations [4, 73, 94] therefore some preliminary cardinality information can be attached to the result of the query analysis even before the actual optimisation.

This work relies on tree patterns in two crucial ways. First, we use tree paterns as a basis for representing a query inside the query processor, as has been done in previous works. However, in contrast to previous works, the tree patterns that we extract from an XQuery query are able to cross query block boundaries in order to follow structural relationships between query nodes. Thus, our patterns can be seen as maximal. Chapter 3 describes the pattern extraction process, which is one of the contributions of this thesis.

Second, we use tree patterns to model the persistent data structure available to the query optimizer when it needs to chose data access paths for the query. Tree patterns are an interesting option here, since they capture a large family of storage and indexing models advocated in previous works. Moreover, they can be fitted closely to a given query, offering the potential for very convenient materialized views. The next chapter describes this usage of tree patterns in detail.

# Chapter 2

# XML Access Modules: describing persistent XML storage structures

## 2.1 Persistent storage structures for XML databases

In this section, we describe some of the more representative solutions proposed for storing XML documents. We briefly present the idea behind each strategy, and we use an example to illustrate the tight connection between the plans produced by an XML query optimizer, and its knowledge about the persistent storage structures. We will reuse these examples in section 2.3 in which we will see how to model these storages using our flavor of tree patterns.

We organize our presentation following four broad classes of storage structures: based on a relational store (Section 2.1), on native stores (Section 2.1.2), XML indices (Section 2.1.3) and unfragmented XML storages also known as the "blob" approach, (Section 2.1.1). These examples do not attempt to be exhaustive, but merely representative. A more extensive comparison with existing models is provided in Section 2.3.

We use for illustration the sample document in Figure 2.1, representing bibliographic data. Element and attribute identifiers (materialized by simple integer numbers) appear at the left of the elements' begin tags.

### 2.1.1 XML storage models

#### Storing XML documents in RDBMSs

Some of the first proposals for storing XML documents were based on shredding the XML documents using a fixed partitioning strategy and loading it to a relational DBMS. Simple identifiers and foreign keys were used to materialize the tree structure of the document in the relational table. This solution has the advantage of simplicity since it reuses relational databases technologies and optimisation. However, as we'll see next, this approach may lead to inefficient query evaluation plans.

```
1        <bib>
2, 3       <book year=”1999”>
4            <title>Data on the Web</title>
5            <author>Abiteboul</author>
6            <author>Suciu</author>
           </book>
7          <book>
8            <title>The Syntactic Web</title>
9            <author>Tom Lerners-Bee</author>
           </book>
10, 11     <phdthesis year=”2004”>
12           <title>The Web: next generation</title>
13           <author>Jim Smith</author>
           </phdthesis>
         </bib>
```

Figure 2.1: Sample *bib.xml* document.

Assume the document in Figure 2.1 is stored in a relational database, according to the *Hybrid* storage model [105]. Applying this storage model yields the relations depicted in Table 2.1. In these relations, the underlined ID attribute is a primary key, parentID attributes are foreign keys pointing in the parent relation. Finally, attributes such as year, and single children such as title, are *inlined* (stored as attributes) in the relations corresponding to their parent elements. For example the $year$ yields the $yearValue$ attribute in the **book** and **phdthesis** relations. The values for this attribute are the values (as defined in Section 1.1) of the $year$ elements.

Now, consider the following query $q$:

| Relational storage model #1 (*Hybrid*) | |
|---|---|
| **bib** | (<u>ID</u>) |
| **book** | (<u>ID</u>, parentID, yearValue, titleValue) |
| **phdthesis** | (<u>ID</u>, parentID, yearValue, titleValue) |
| **author** | (<u>ID</u>, parentID, parentType, authorValue) |

Table 2.1: *Hybrid* schema for the sample *bib.xml* document.

*for* $x *in doc("bib.xml")//book*
*return* <*info*>{$*x/author*} {$*x/title*}</*info*>

To answer this query on the relations in Table 2.1, the query optimizer has to figure out which relations store book elements. Then, it has to identify where are the years and authors stored, and how to connect them to books. Year values are stored in the book table itself; but, to connect the authors, a join on the book.ID and author.parentID is needed. The algorithm for this reasoning (not described in [105]) is based on the fixed DTD-to-relational schema mapping corresponding to the *Hybrid* scheme.

Based on this reasoning, a possible query plan[1] is:

$$QEP_1 \quad \pi_{\text{authorValue,titleValue}}(\text{sort}(\textbf{book} \bowtie_{\text{ID=parentID}} \textbf{author}), \text{ID})$$

where $\bowtie$ is implemented e.g. by a HashJoin. This plan combines book and author information via a key-foreign key join. Notice that this plan is only correct if we are sure that every book has at most one year (otherwise, some authors would be erroneously duplicated). Furthermore, this plan also assumes that every book has at least one year(otherwise, the titles of books without a year would be left out of the answers, which contradicts the query). A more precise relational plan for $q$, using outer-joins, can be written; for simplicity of exposition, we prefer to rely on$QEP_1$ which suffices for illustration.

Now, let us assume we decide to create a separate relation storing the publication titles. This corresponds to the *Shared* model proposed in [105]; the resulting schema is shown in Table 2.2. A possible QEP for $q$ on this storage is:

$$QEP_2 \quad \pi_{\text{authorValue,titleValue}}(\text{sort}(\textbf{book} \bowtie_{\text{ID=parentID}} \textbf{author} \bowtie_{\text{ID=parentID}} \textbf{title}), \text{ID})$$

[1]For simplicity, in this and the following query execution plans, we ignore some other operations required to compute the answer of $q$, such as data re-structuring and tagging.

| Relational storage model #2 (*Shared*) | |
|---|---|
| **bib** | (ID) |
| **book** | (ID, parentID, yearValue) |
| **phdthesis** | (ID, parentID, yearValue) |
| **title** | (ID, parentID, parentType, titleValue) |
| **author** | (ID, parentID, parentType, authorValue) |

Table 2.2: *Shared* schema for the sample document.

Notice that $QEP_2$ involves one more table than $QEP_1$. Adapting to the change of the storage, between the schema in Table 2.1 and Table 2.2, requires changing the code which identifies the storage structures corresponding to every fragment of XML data.

If our space of possible storage schemas is limited to *Hybrid* and *Shared*, then moving from one to another amounts to switching between two pieces of code responsible for the XML-to-relational mapping and translation.

But the space of options is, in fact, much wider. Assume, for instance, that we decide to speed up the evaluation of $q$, by creating a table **book-author-title**, in which each book is combined with its title and each of its authors (or with *nulls* if these are missing). Such a model was proposed in [51].

| Additional table for relational models #1 or #2 | |
|---|---|
| **book-author-title** | (ID, parentID, yearValue, titleValue, authorValue) |

Based on this relation, another QEP for $q$ is:

$QEP_3$ scan(**book-author-title**)

If we add the **book-author-title** to our relational storage model #1, the mapping code associated to the *Hybrid* model needs to be changed to take into account the new relation. We may also decide to use **book-author-title** in conjunction with the relational storage model #2; this requires modifying the corresponding mapping code. Now assume that we decide to drop the **book** relation itself, since its information is also included in the **book-author-title** relation. This, in turn, requires serious modifications to both the *Hybrid* and the *Shared* mapping and translation code, since this code was conceived assuming the existence of a **book** relation for our sample document.

The potential for similar examples is almost infinite; just add or erase a relation storing some information from our sample document, and some code will have to be modified. Moreover, moving to a different dataset will lead to creating different tables etc. Clearly, this is a far cry from the principle of physical data independence at the core of the success of relational databases [104].

Are the above problems simply due to the mismatch between the XML and a *relational* storage model? The answer is no. As we show next, similar problems arise when changing a *native* storage model, for instance, by adopting a different node labeling scheme, or by better organizing the data.

### Native XML storage models

Consider the native storage model used by an early version of the persistent store of the Galax system [111]. This model consists of several indexed data collections. The most important ones are:

| Native storage model #1 | |
|---|---|
| **main** | (<u>ID</u>, parentID, kind, nameID) |
| **text** | (<u>ID</u>, text) |
| **name** | (<u>ID</u>, name) |

There is one entry in the **main** collection for every node in the XML document (including elements, attributes, and text nodes). The "kind" field allows to distinguish between these various node types, as XQuery requires. The **text** structure stores the textual values of the XML document leaves. Based on this model, a possible QEP for $q$ is:

$$
\begin{array}{ll}
QEP_4 & (\textbf{main} \bowtie (\sigma_{\text{name="book"}}\textbf{name})) \mathbin{⟕} \\
 & (\textbf{main} \bowtie (\sigma_{\text{name="author"}}\textbf{name})) \mathbin{⟕} \\
 & (\textbf{main} \bowtie (\sigma_{\text{name="title"}}\textbf{name}))
\end{array}
$$

In the above, all the joins denoted $\bowtie$ correspond to the condition "main.nameID = name.ID", while the outerjoins connect parents to their children on the join predicate "ID=parentID". All the joins could be implemented e.g. by hash joins.

Now, assume we decide to change this schema, by adopting *structural identifiers* of the form (pre-order, post-order, depth) which were described in Section 1.2.1.

Observe that, in the *Native storage model #1*, the $parentID$ field serves to connect parent and child elements. In the presence of structural identifiers, the $parentID$ field becomes unnecessary, and the storage can be simplified into:

| Native storage model #2 | |
|---|---|
| **main** | (ID, kind, nameID) |
| **text** | (ID, text) |
| **name** | (ID, name) |

Correspondingly, the plan for $q$ now becomes:

$$QEP_5 \quad (\textbf{main} \bowtie (\sigma_{\text{name="book"}}\textbf{name})) \bowtie\!\sqsubset_{main_1.\text{ID} \prec main_2.\text{ID}} (\textbf{main} \bowtie (\sigma_{\text{name="author"}}\textbf{name})) \bowtie\!\sqsubset_{main_1.\text{ID} \prec main_3.\text{ID}} (\textbf{main} \bowtie (\sigma_{\text{name="title"}}\textbf{name}))$$

where in the join condition $main_1$, $main_2$ and $main_3$ indentify the three occurences of the main relation in the query execution plan. The use of structural join algorithms is conditioned by the optimizer's knowledge of the special structural information encoded by the identifiers.

Now, assume we decide to partition the **main** table by the names of the nodes. Thus, node name moves from the **name** table (where it was stored as *data*) to the *metadata* characterizing the storage; the **name** table disappears. The storage schema is now:

| Native storage model #3 | |
|---|---|
| **bib** | (ID) |
| **book** | (ID) |
| **phdthesis** | (ID) |
| **title** | (ID) |
| **author** | (ID) |
| **year** | (ID) |
| **text** | (ID, text) |

The tag-partitioned collections of (pre-order, post-order, depth) identifiers above are exactly the indexes used by Timber [63] and Natix [63]. The **text** table associates element IDs with their text children. Once the optimizer has been modified to inform it of these changes, it may produce the following plan for $q$:

$QEP_6$ $(\textbf{book} \bowtie\!\sqsubset_{\text{book.ID} \prec \text{title.ID}} \textbf{title} \bowtie\!\sqsubset_{\text{book.ID} \prec \text{author.ID}} \textbf{author})$
$\bowtie \textbf{text} \bowtie \textbf{text}$

In the above, the last two joins pair the **author** and **title** IDs with their text children. Notice that all **author** and **title** IDs participate to the structural joins, whether they belong to book or phdthesis elements.

To obtain more selective disk access, we may decide to partition the structural identifiers, not by the *tag*, but by the *path* leading to the elements, as in [83, 103]. This would lead to a new storage scheme:

| Native storage model #4 | |
|---|---|
| **bib** | (ID) |
| **bib-book** | (ID) |
| **bib-book-title** | (ID) |
| **bib-book-year** | (ID) |
| **bib-book-author** | (ID) |
| **bib-phdthesis** | (ID) |
| **bib-phdthesis-title** | (ID) |
| **bib-phdthesis-year** | (ID) |
| **bib-phdthesis-author** | (ID) |
| **text** | (ID, text) |

Modified anew, to learn about the new storage model, the query optimizer will now produce a different plan:

$QEP_7$ $(\textbf{bib-book} \bowtie\!\sqsubset_{\text{bib-book.ID} \prec \text{bib-book-title.ID}}$
$\textbf{bib-book-title} \bowtie\!\sqsubset_{\text{bib-book.ID} \prec \text{bib-book-author.ID}}$
$\textbf{bib-book-author}) \bowtie \textbf{text} \bowtie \textbf{text}$

where the semantics of the joins and outerjoins are the same as for $QEP_5$.

### Non-fragmented storage models

We have so far considered a quite simple and regular XML data set, typically called "data-centric". In these documents, each element encapsulates some useful information which may be queried alone; thus, the corresponding storage models tend to *fragment* the data according to some criteria (e.g. element name or path).

Another important class of XML documents comprises more textual ("document-centric") datasets. For example, consider a modified version of the first book element in our sample document, shown in Figure 2.2: we now assume that the full text of the book is XML-encoded.

```
<bib>
  <book year="1999">
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Suciu</author>
    <body>
        <section no="1">
            In this book, we discuss <it>Web data</it>
            as encountered in HTML and, increasingly,
            XML documents on the <b>Web</b>.
        </section>
        . . .
    </body>
  </book>
    . . .
</bib>
```

Figure 2.2: Fully XML-ized book element.

Within the first book section, notice the XML elements encoding markup, such as "it" (for italics) and "b" (for bold). Such elements are frequent in bibliographic XML data sets, such as the set of 12.000 IEEE computer science publications used in the INEX [61] XML information retrieval benchmark (which uses about two dozens such tags), or in the Library of Congress's dataset [78]. Similarly, in the XMark [115] benchmark, centered on an on-line auction site, the descriptions of items for sale are heavily marked up with XML elements representing bold and italic text, bullet lists, paragraphs etc.

Now, consider the following simple query $q'$ on the XML-ized book element in Figure 2.2:

*doc("bib.xml")//book//section*

Assuming e.g., the native storage model #4, a query plan for $q'$ would be:

$$QEP_8 \quad (\textbf{book} \bowtie_{\text{book.ID}\prec\text{section.ID}} \textbf{section} \bowtie_{\text{section.ID}\prec\text{it.ID}} \textbf{it} \bowtie_{\text{section.ID}\prec\text{b.ID}} \textbf{b}) \bowtie_{\text{section.ID=text.ID}} \textbf{text} \bowtie_{\text{it.ID=text.ID}} \textbf{text} \bowtie_{\text{b.ID=text.ID}} \textbf{text}$$

For some applications, such as XML information retrieval, it does not make sense to store XML elements encoding markup, separately from their enclosing text. Instead, it may be desireable to store the whole *content* of some elements (as defined in Section 1.1) in a single textual field, corresponding to their text image in the original XML file. In the case of the book sections, this yields:

| Non-fragmented storage of book sections |
| --- |
| **sectionContent**(ID, content) |

Such a structure has several benefits. It is better suited to full-text indexing than a fragmented storage, and provides better opportunities for text compression. But most importantly, it avoids joins when recomposition of the textual image of book sections is needed. For instance, based on this structure, $q'$ can be answered by the following plan:

$$QEP_9 \quad \textbf{book} \bowtie_{\text{book.ID}\prec\text{section.ID}} \textbf{sectionContent}$$

Notice that $QEP_9$ is much simpler than $QEP_8$, and is likely to have much better performance. Finally, we note that native storages organized as persistent trees, such as those of Natix [46] and Timber [63], are also examples of (logically) unfragmented storage. Element content is clustered with the element, thus it can be accessed without joining several storage structures.

### 2.1.2 XML indexing models

We have so far considered various storage models, and resulting query plans, for XML queries such as $q$, which are mainly focused on navigation. Now let us consider a query applying a selection on the data, such as $q''$:

*for* $x *in doc("bib.xml")//book*
*where* $*x/year="1999" and* $*x/title="Data on the Web"*
*return* $*x/author*

A simple QEP for $q''$, based e.g., on the native storage model #4, would be:

$QEP_{10}$ (**bib-book** $\bowtie\!\sqsubset_{\text{bib-book.ID}\prec\text{bib-book-title.ID}}$ **bib-book-title**
$\bowtie\!\sqsubset_{\text{bib-book.ID}\prec\text{bib-book-year.ID}}$ **bib-book-year**
$\bowtie\!\sqsubset_{\text{bib-book.ID}\prec\text{bib-book-author.ID}}$ **bib-book-author**) $\bowtie$
$\sigma_{\text{“Data on the Web”}}$(**text**) $\bowtie \sigma_{\text{“1999”}}$(**text**) $\bowtie$ **text**

Now, assume that, in order to speed up the processing of $q''$, we build an index associating to each (year, title) pair, the identifier(s) of the book element(s) appearing in that year, with that title.

| Index: book IDs by (book year, book title) |
| --- |
| **booksByYearTitle** (yearVal, titleVal, bookID) |

Then, the optimizer could use this index to construct a potentially more efficient plan:

$QEP_{11}$ idxLookup(**booksByYearTitle**,(1999,“Data on the Web”))
$\bowtie\!\sqsubset_{\text{booksByYearTitle.book.ID}\prec\text{bib-book-author.ID}}$ **bib-book-author** $\bowtie$ **text**

To construct $QEP_{11}$, the optimizer needs to know that the index exists, what is the index key, and what is the lookup result. For the general case, we should also specify whether books without a year, or without a title, are covered by the index, and how.

We have so far only considered structure-oriented queries, such as $q$, $q'$ and $q''$. Another important class of XML queries involves text search. As a very simple example, consider the following query $q'''$:

*for* $x *in doc(“bib.xml”)//book/title*
*where* $x *ftcontains “Web”*

This query retrieves all book titles containing the word “Web”. A simple query plan for $q'''$, based on our native model #4, is:

$QEP_{12}$ $\sigma_{contains(\text{text},\text{”Web”})}$(**text**) $\bowtie$ **bib-book-title**

where the function $contains(t, w)$ returns true if the text $t$ contains the word $w$, and false otherwise. The function $contains$ can be implemented directly by a string pattern matching algorithm. However, this would require calling it for every distinct text value. A more efficient alternative is to materialize a *full-text index* (FTI), e.g., following the IndexFabric [37] model:

| Index: book title IDs by title words |
|---|
| **bib-book-title-fabric**(word, ID) |

The index **bib-book-title-fabric** returns, for any given word $w$, the identifiers of all book titles containing the word $w$. As noted by its authors, the IndexFabric can be combined with either a relational or a native store. Assuming we use it together with the native model #4, a more efficient plan for $q'$ is:

$QEP_{13}$ idxLookup(**bib-book-title-fabric**, “Web”) $\bowtie$
**bib-book-title**

To construct this plan, the optimizer has to be told that the FTI exists, and also how the FTI is organized, that is: what is the scope of the indexed texts, what are the index inputs, and outputs. For instance, the IndexFabric stores word occurence information within precise parent-child paths. Other variants, such as the Natix FTI [46], return word occurences found on any path in the document; the FTI described in [49] returns occurences found as direct children of elements of a given tag etc.

### 2.1.3 XML materialized views

Let us consider the following query $q_1$ that evaluated on the XMark document from Figure 1.1 produces all the names of the auctioned items paired with the list of keywords from the item’s description:

```
for $x in doc("XMark.xml")//item
return <result>{$x/name/text()}
               {$x//keyword}</result>
```

We’ll briefly evaluate what are the alternatives for current available choices for the appropriate materialized views to speed-up the query execution:

1. A first solution would be to use an XQuery view that materializes the whole query. Indeed, recent works have explored XQuery view rewriting using XQuery views [38, 43]. This solution clearly provides the materialized view which is the best fit for the query. Thus, the complexity of computing the answer to $q_1$ is reduced (in general, only a simple table scan is needed). However, due to the compexity of the XQuery language[2], it is difficult to understand and combine multiple XQuery views in order to answer a query.

2. Many works studied materializing XPath expressions that correspond to the navigation part of the XQuery [18, 120] and use them to rewrite XQuery queries both in the presence of structural constraints and without constraints. Usually the approach consist in finding a view (or view set) covering the query, materializing this view in a flat table, and then use navigation to gather all the data needs of the query. For the given query we have several XPath views that could be used in the rewriting:

   - One candidate XPath view is $V_1 = //item$. In fact using only $V_1$ we can fully rewrite $q_1$: using navigation we would compute an algebraic expression $q_2 = E_1(V_1)$ where $q_2 \equiv q_1$. However, materializing $V_1$ is not a good choice for our XMark document: the view is very big, so the cost of storing its result can be prohibitive. Moreover, $E_1$ is complex and possibly inefficient.

   - A second solution is to materialize the views $V_1 = //item/name$, $V_2 = //item//keyword$ and to use the rewriting $q_1 = E_2(V_2, V_3)$. This solution turns out to be more efficient because we store now only $name$ and $keyword$ elements not full $item$ elements. However when constructing the result of $q_1$ we have to pair the item names with their corresponding keywords which is impossible in the absence of elements identifiers. Moreover, due to the impossibility to express nesting or optional elements in XPath, the optimizer must do a lot of work in rewriting complex nested queries.

[2] We recall the classical identity loss problem of XQuery semantics: if an XQuery view builds new elements including elements from the input, the identity of the input elements is lost (element constructors have COPY semantics). XQuery-based rewritings are thus correct as long as node identity is not an issue.

### 2.1.4 Summary: interesting features in XML persistent storage structures

From the previous sections, we derived a set of interesting features of XML storage models. We summarize them next, as they were the basis for designing the XML Access Modules tree pattern language.

**Labels stored as data or metadata** Node labels may be stored as plain data values; such was the case, for instance, in the native storage model #1 we considered. Other models encode node labels within the metadata, that is, the schema describing the persistent storage structures. Examples of this approach are the relational storage models #1 and #2, and the native storage models #3 and #4.

**Node-driven storage** Simple storage schemes may choose the XML document node (whether element, attribute, or text) as the storage granularity. This was the case for our native storage model #1.

**Label- or path-driven clustering** A storage model may separate information about XML nodes, according to their labels. This was the case, for instance, in the relational models #1 and #2, and the native models #2 and #3 previously presented. Another simple fragmentation strategy is based on the paths in the XML document; an example is provided by our native storage model #4.

**Complex clustering** A storage model may choose to cluster data according to a DTD, as in the relational models #1 and #2. More generally, custom clustering strategies can greatly improve the performance of specific queries; such was the case of the **book-author-title** relation in Section 2.1.1. Establishing custom clusterized structures is akin to using materialized views, or indexes to speed up specific queries, such as the T-index [86] and the F&B index [68].

**Identifiers: simple, complex, or absent** Node identity is a crucial notion for the semantics of several XQuery operations, thus the need for persistent identifiers. All existing storage models assign persistent identifiers to (some of) the XML document nodes. IDs may be very simple, such as the integers used in our relational models #2 and #3, and the native model #1. IDs may, on the contrary, incorporate structural information, as in the native models #2 and #3 that we used. Finally, IDs may not be

created for *all* XML nodes: for instance, years and titles are not assigned IDs by our relational model #1.

**Generic indexes** The complexity of XML data, and queries, requires complex indexes. An index may refer to XML document structure *and* content, and may have a composite key, as was the case for our index **booksByYearTitle** in Section 2.1.1.

**Fine granularity** XML query processing requires at least some minimal capability of full-text search in the document. To that purpose, FTIs of various styles may be included in the storage, as discussed in Section 2.1.2.

**Coarse granularity** In some applications, coarse granularity (storing the full textual content of an XML element as such) may yield good performance. This is particularly true for complex elements which are often returned by queries, but never (or rarely) *traversed*. An example has been provided in Section 2.1.1.

**Typed and untyped documents** Semi-structured data, and XML, were initially touted for their *schema-less, self-describing* aspect. XML type descriptions, e.g., DTDs and XML Schemas, have changed this perspective. However, a study of the XML documents found on the Web towards the end of 2002 [84] found that barely 40% have a DTD, and about 1% have an XML Schema. The situation may have changed since, but radical changes are unlikely. Thus, it must be possible to store (and efficiently process queries over) typed and untyped data.

**Putting it all together** Does an optimizer really need to handle *all* the above features ?

For a given data set and query workload, the optimal storage will probably need only a small subset of the spectrum of choices enumerated above. However, a (realistic) XML DBMS needs to handle, with reasonable, if not optimal, performance, *any* XML data set, and *any* query workload. DBMSs are rarely developed with the goal of handling one single data set and workload. Changing the storage model (e.g., by changing the fragmentation strategy, moving tags from data to metadata, adding a custom index or materialized view, or trying a new node labeling scheme) should not require changing the optimizer code. Moreover, locking an optimizer inside a storage model rules out the usage of useful techniques, such as indexing models and labeling schemes, that may be proposed in the future.

We conclude that a *high-level description of the storage* is a crucial component of an extensible and efficient XML query optimizer. A change in the storage model is, in this case, reflected simply by a change in the storage description (which is compact and easy to understand and modify); no change to the optimizer code is needed. We propose our formalism for high-level storage model descriptions in the next section.

## 2.2 XML Access Modules syntax and semantics

XML Access Modules (in short XAMs) are a formalism for representing an XML storage, index or materialized view. This formalism must be general enough to capture many existing (and future !) proposed structures. It must have clearly defined semantics, to allow the optimizer to make informed decisions. As we concluded from the previous sections, the formalism should be able to express value and structure conditions, fine and coarse granularity, as well as ID presence and their interesting properties. Finally, it should be relatively easy to understand and to express. We describe XAM syntax and semantics next.

### 2.2.1 XAM syntax

An *XML Access Module* describes a fragment of an XML document stored in a persistent data structure. Formally, a XAM is an ordered tree $(NS, ES, o)$, where: $NS$ is a *node specification*, $ES$ is an *edge specification*, and $o$ is an *order flag*. If the XAM data is stored in document order, $o$ is set to true; otherwise, $o$ is false.

We now describe XAM specifications, using a grammar-like notation (Figure 2.3). We use bold font for terminal symbols of the grammar, i.e., constants.

Any XAM specification contains a special node $\top$, corresponding to the document root (the ancestor of all elements and attributes in a document). The other nodes represent elements or attributes, and have an associated $name$; by convention, names starting with @ are used for XAM nodes representing XML attributes.

A node may be annotated with: an identifier specification $IDSpec$, a tag specification $TSpec$, a value specification $VSpec$, and a content specification $CSpec$.[3] As introduced in Section 1.1, by content, we mean the full (serialized) representation of the XML element or attribute.[4] An ID (resp. tag, value, content) specification, attached to a XAM node, denotes the fact that the element/attribute ID (respectively, tag, value, or full textual content) is stored in the XAM.

Node identity is a crucial notion in XQuery processing. Any XML store provides some persistent identifiers; the particular type of IDs used determines the efficiency of

[3]Attribute nodes are uniquely identified by their parent's ID and the attribute name. We use explicit atribute IDs for simplicity.

[4]Clearly, the content of an XML element can always be retrieved from a non-lossy storage, by combining accesses to *several* storage modules. Here, we use Cont only for the storage models able to retrieve it from a *single* persistent data structure.

matching structural query conditions. The persistent identifiers stored in a XAM are described by the ID specification (line 3); it consists of the symbol **ID**, and one of four symbols, depending on the level of information reflected by the element identifier:

1. We use **i** for simple IDs, for which we only know that they uniquely identify elements.

2. The symbol **o** stands for IDs reflecting document order; simple integer IDs used e.g., in [39, 48, 105] are a typical example.

3. We use **s** to designate structural identifiers, which allow to infer, by comparing two element IDs, whether one is a parent/ancestor the other. They are produced e.g., by the popular (preorder, postorder, depth) labeling schemes based on Dietz's model [44].

4. We use **p** to designate structural identifiers which allow to *directly derive* the identifier of the parent from that of the child, such as the Dewey scheme in [107], or ORDPATHs [90].

$$NS ::= \top \ \ N^{+} \tag{2.1}$$

$$N ::= \ name\ IDSpec?\ TSpec?\ VSpec?\ CSpec? \tag{2.2}$$

$$IDSpec ::= \ \mathbf{ID}^{\mathbf{i}\ |\ \mathbf{o}\ |\ \mathbf{s}\ |\ \mathbf{p}}\ \mathbf{(R?)} \tag{2.3}$$

$$TSpec ::= \ (\mathbf{Tag}(\mathbf{R}?))\ |\ \mathbf{[Tag=}c\mathbf{]} \tag{2.4}$$

$$VSpec ::= \ (\mathbf{Val}(\mathbf{R}?))\ |\ \mathbf{[Val=}c\mathbf{]} \tag{2.5}$$

$$CSpec ::= \ \mathbf{Cont} \tag{2.6}$$

$$ES ::= \ E* \tag{2.7}$$

$$E ::= \ name_1\ (/\ |\ //)(\mathbf{o}\ |\ \mathbf{j}\ |\ \mathbf{s}\ |\ \mathbf{nj}\ |\ \mathbf{no})\ name_2 \tag{2.8}$$

Figure 2.3: Full XAM syntax.

An **R** symbol in an ID specification denotes an *access restriction:* the ID of this XAM node is *required* (must be known) in order to access the data stored in the XAM. This feature is important to model persistent tree storage structures, which enable navigation from a parent node to its children, as in [63, 46]. More generally, **R** symbols allow to model arbitrary XML indexes, on structure and values: key values must be known to perform an index lookup.

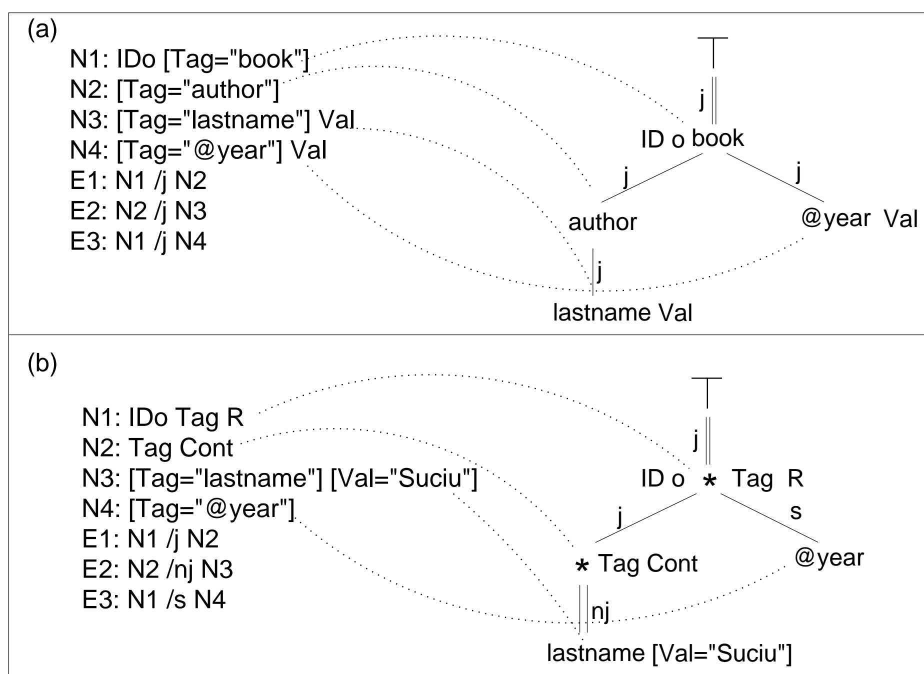



Figure 2.4: Sample XAMs: grammar (left) and graphical representations (at the right).

A tag specification of the form **Tag** denotes the fact that the element tag (or attribute name) can be retrieved from the XAM. Alternatively, a tag specification predicate of the form **[Tag=**$c$**]** signals that only data from the subtrees satisfying the predicate is stored by the XAM. The tag value can also be required; this is also marked by the symbol **R**. Value and content specifications are very similar. The value(s) stored in a node corresponding to elements are the textual children of the elements. The value(s) described by a node corresponding to attributes are the attribute value(s).

XAM edges can be either parent-child edges, marked /, or ancestor-descendant edges, marked //. We distinguish *join*, *left outerjoin*, *left semijoin*, *nest join* and *left nest outer join* semantics for the XAM edges, considering the parent node on the left hand). These are marked by the symbols **j**, **o**, **s**, **nj**, respectively **no**. All these joins correspond to structural relationships; nest join variants furthermore allow the construction of complex nested tuples.

EXAMPLE 2.2.1. Figure 2.4 depicts the grammar and graphical representation of two XAMs. For the graphical representation, we use node and edge-annotated tree

patterns. The dotted lines link together the representations for the same pattern node. Notice that XAM nodes annotated with **[Tag=$c$]** specification are depicted by $c$ nodes, whereas nodes annotated with **Tag** are depicted by $*$ nodes. The rest of node and edge annotations translate directly from the grammar notation. $\triangleleft$

The data from $D$ stored by a XAM is a set (or list) of possibly nested tuples, whose schema is derived from the XAM definition. This is formally defined next.

```
1          <library>
2, 3         <book year="1999">
4              <title>Data on the Web</title>
5              <author>Abiteboul</author>
6              <author>Suciu</author>
             </book>
7            <book>
8              <title>The Syntactic Web</title>
9              <author>Tom Lerners-Bee</author>
             </book>
10, 11       <phdthesis year="2004">
12             <title>The Web: next generation</title>
13             <author>Jim Smith</author>
             </phdthesis>
           </library>
```

Figure 2.5: Sample XML document.

### 2.2.2 XAM semantics

The semantics of a XAM $\chi$ over a document $d$ is a set of nested tuples, corresponding to the data contained in a storage module described by $\chi$ for the document $d$. However, XAM semantics adapts it to the needs of storage description, as we explain next.
We define XAM semantics in two stages: first, omitting the **R** annotation, then including it too. We start by introducing some useful notions.

We use the notation $d.root$ to denote the root of an XML document, which is the parent of the top XML element in $d$.

DEFINITION 2.2.1 (TAG-DERIVED COLLECTION). Let $t$ be an element name and $d$ be an XML document. We define the *tag-derived collection (set/list) of* $t$ as a set/list of tuples $R_t$(ID:$ID$, Val: $\mathcal{A}$, Tag: String, Cont: String):

$R_{book}$

| ID | Tag | Val | Cont |
|---|---|---|---|
| 2 | book | ⊥ | <book year="1999"><br><title>Data on the Web</title><br><author>Abiteboul</author><br><author>Suciu</author><br></book> |
| 7 | book | ⊥ | <book><br><title>The Syntactic Web</title><br><author>Tom Lerners-Bee</author><br></book> |

$R^{@}_{year}$

| ID | Tag | Val | Cont |
|---|---|---|---|
| 3 | year | "1999" | year="1999" |
| 11 | year | "2004" | year="2004" |

$R_{title}$

| ID | Tag | Val | Cont |
|---|---|---|---|
| 4 | title | "Data on the Web" | <title>Data on the Web</title> |
| 8 | title | "The Syntactic Web" | <title>The Syntactic Web</title> |
| 12 | title | "The Web: next generation" | <title>The Web: next generation</title> |

Figure 2.6: Tag-derived lists on the document in Figure 2.5.

$$R_t(d) = \{(n.ID, n.Val, n.Tag, n.Cont) \mid n \in d, n.Tag = t\}$$

$R_t$ contains a tuple for each element $n \in d$ whose tag is $t$. If $R_t$ is a list, then tuples follow the document order.

We similarly define the collection $R_*$(ID:$ID$, Val: $\mathcal{A}$, Tag: String, Cont: String) as:

$$R_*(d) = \{(n.ID, n.Val, n.Tag, n.Cont) \mid n \in d\}$$

Similarly, the collection $R^{@}_t$ reflects all attributes nodes labeled $t$, and $R^{@}_*$ reflects all attribute nodes. ◁

As an example, Figure 2.6 shows the tag-derived lists $R_{book}(d)$, $R_{title}(d)$ and $R^{@}_{year}(d)$, where $d$ is the sample document in Figure 2.5. For simplicity, we will only use the attribute names ID, Val, Tag and Cont in association with the above types, and omit the atomic attribute types.

The next ingredient of XAM semantics is *logical* structural joins. Structural joins combine two collections of tuples based on a structural relationship between nodes whose IDs appear in the collections. We consider the *parent-child* and *ancestor-descendant* relationships; accordingly, structural joins are denoted as $\bowtie^{pc}$ for parent-child, and $\bowtie^{ad}$ for ancestor-descendant. Notice that structural joins are asymmetric; we distinguish e.g. $\bowtie^{pc}$ from $\bowtie^{cp}$, depending on which input contains the parent IDs. Furthermore, we also use *structural semi-joins* such as $\rhd\!<^{pc}$, and *structural outer-joins* such as $\bowtie\!\sqsubset^{pc}$.

### Algebraic semantics of a XAM without access restrictions

Let $\chi$ be a XAM without **R** annotations. Without loss of generality, we assume $\chi$ to be ordered.

**Notation** We denote by $[\![\chi]\!]_d$ the *semantics of* $\chi$ *over a document* $d$, namely a set (or list, if $\chi$ is ordered) of (possibly nested) tuples whose content is extracted from $d$.

DEFINITION 2.2.2 ($\top$ SEMANTICS). Let $\chi$ consist of the single node $\top$. In this case, we have:

$$[\![\chi]\!]_d = \{(\text{root}{=}d.root.ID)\}$$

Thus, the document root is the only one matching $\top$. $\lhd$

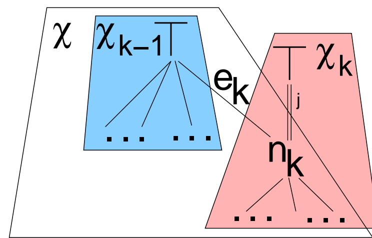



Figure 2.7: Illustration for the algebraic XAM semantics.

DEFINITION 2.2.3 (TWO-NODE XAM SEMANTICS). Let $\chi$ be a XAM consisting of a node $\top$ connected to a node $n_1$ by an edge labeled with $//$ and **j**. The semantics of $\chi$ over a document $d$ is:

1. If $\mathsf{n}_1$ is an element node, with a $TagSpec$ of the form **[Tag=$t$]**, then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R_t(d)))$.
2. If $\mathsf{n}_1$ is an element node with a different $TagSpec$ (or none), then, $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R_*(d)))$.
3. If $\mathsf{n}_1$ is an attribute node, with a $TagSpec$ of the form **[Tag=$t$]**, then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R^@_t(d)))$.
4. If $\mathsf{n}_1$ is an attribute node with a different $TagSpec$ (or none), then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R^@_*(d)))$.

In the above formula:

1. $\sigma_\chi$ is a selection on the conjunction of all predicates of the form **Val=**$c$ that appear in the value specifications of $\chi$ nodes. $\sigma_\chi$ checks each such predicate against the respective Val attributes.
2. $\Pi_\chi$ is a projection which: $(i)$ eliminates the root attribute, $(ii)$ for every non-$\top$ node in $\chi$, retains the ID (respectively, the Val, Tag, Cont attribute) only if the node has an ID specification of the form **ID** (respectively, value specification of the form **Val**, tag specification of the form **Tag**, and content specification of the form **Cont**) and $(iii)$ eliminates duplicate tuples.

◁

Now, consider a larger XAM, such as $\chi$ in Figure 2.7. In this figure, $\chi_{k-1}$ is the same as $\chi$ but for the rightmost subtree, rooted in node $\mathsf{n}_k$. Furthermore, $\chi_k$ is obtained by adding a $\top$ node on top of $\mathsf{n}_k$, connected to $\mathsf{n}_k$ by a descendant edge annotated j (join).

**Simplifying assumption: IDs present** We start by assuming that $\mathsf{n}_k$ has an ID specification of the form **ID**.

DEFINITION 2.2.4 (SEMANTICS OF A XAM WITH IDS). Let $\chi$ be a XAM, $\chi_{k-1}$ and $\chi_k$ be the XAMs derived from $\chi$ as in Figure 2.7. The semantics of $\chi$ over a document $d$, denoted $[\![\chi]\!]_d$, is:

$$[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi([\![\chi_{k-1}]\!]_d \circ [\![\chi_k]\!]))$$

In the above, $\sigma_\chi$ and $\Pi_\chi$ are defined as in the previous definition, while $\circ$ stands for:

- structural join if $\mathsf{e}_k$ is labeled **j**
- structural semijoin if $\mathsf{e}_k$ is labeled **s**
- structural outerjoin if $\mathsf{e}_k$ is labeled **o**
- nest structural join if $\mathsf{e}_k$ is labeled **nj**
- nest-structural outerjoin if $\mathsf{e}_k$ is labeled **no**

The structural relation tested by the above structural (possibly nest) join, outerjoin or semijoin depends on the edge $\mathsf{e}_k$: it is parent-child if $\mathsf{e}_k$ is labeled $/$, and ancestor-descendant if it is labeled $//$. ◁

This definition constructs a structural join tree isomorphic to the XAM tree itself. In this join expression, structural joins are paranthesized bottom-up. Consider, for example, the XAM from Figure 2.4(a). Its annotated semantics over the document $d$ is:

$$[\![\chi]\!]_d = ((R_{lastname} \bowtie R_{author}) \bowtie R_{book}) \bowtie R^{@}_{year}$$

**General case: XAMs without IDs** Now assume $\mathsf{n}_k$ does not have an ID specification of the form **ID**, and let $\chi'$ be a XAM identical to $\chi$ but where $\mathsf{n}_k$ has such an ID specification. Intuitively, the semantics of $\chi$ and $\chi'$ are identical except for the missing IDs in $\chi$. Thus:

DEFINITION 2.2.5 (SEMANTICS OF A XAM IN GENERAL). Let $\chi$ be a XAM and $\chi'$ be the XAM obtained from $\chi$ by adding ID specifications to $\chi$'s node $\mathsf{n}_k$ as above. The semantics of $\chi$ is:

$$[\![\chi]\!]_d = \Pi_\chi([\![\chi']\!]_d)$$

where the semantics of $\Pi_\chi$ is specified in Definition 2.2.3. ◁

For example, consider the XAMs in Figure 2.8. Let $d$ be the XML document in Figure 2.5, where node numbers are used as order-preserving IDs. By Definition 2.2.3, we obtain:

$[\![\chi_1]\!]_d$= [$\mathsf{e}_1$(ID=2, Tag="book"), $\mathsf{e}_1$(ID=7, Tag="book")]

In the above, and in the sequel, XAM node names appear explicitly in every tuple, to facilitate reading.

We obtain $[\![\chi_2]\!]_d$ by a structural semijoin on $[\![\chi_1]\!]_d$ and $R^{@}_{year}$:

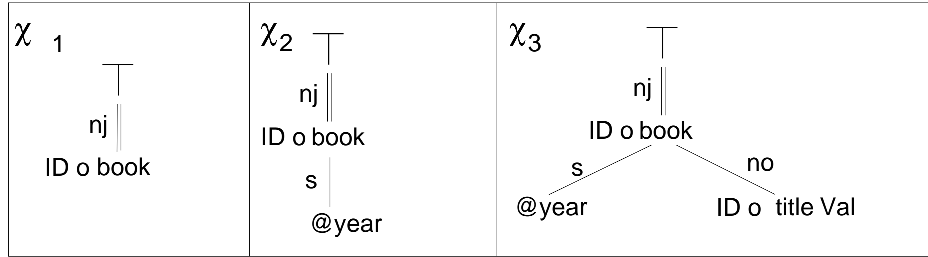



Figure 2.8: Sample XAMs to illustrate XAM semantics.

$[\![\chi_2]\!]_d$= [e$_2$(ID=2, Tag="book")]

Only the first tuple from $[\![\chi_1]\!]_d$ contributed to $[\![\chi_2]\!]_d$, since only the first book had a match in $R^{@}_{year}$.

Applying again Definition 2.2.4 on $[\![\chi_2]\!]_d$, we obtain:

$[\![\chi_3]\!]_d$= [e$_1$(ID=2,Tag="book",
e$_2$[(ID=4,Tag="title",Val="Data on the Web")])]

### Algebraic semantics of a XAM with access restrictions

We now extend the XAM semantics to account for the **R** (required) marker. Intuitively, values for the required fields have to be known in order to be able to access the data stored by the XAM. Thus, XAMs with access restrictions intuitively correspond to indices over XML data.

The semantics of a XAM with access restrictions (represented by **R** markers) can only be defined *with respect to a set of bindings*, that is, a set of values for the required attributes. Bindings for a XAM $\chi$ consist of (possibly nested) tuples of values; the type of these tuples is the projection of $\chi$'s type, over the attributes marked with **R**.

For instance, consider the XAM $\chi_4$ in Figure 2.9. $\chi_4$ contains information about elements having "title" and "author" sub-elements. However, in order to access this information, one must provide the tag of the elements corresponding to e$_1$, and the title associated to these elements. A typical storage structure modeled by $\chi_4$ would be an index on publications, with a composite index key consisting of the publication type and title (the required attributes in $\chi_4$).

For instance, consider the following binding tuple for $\chi_4$:

$t_{B1}$= e$_1$(Tag="book", e$_3$[(Val="Data on the Web")])

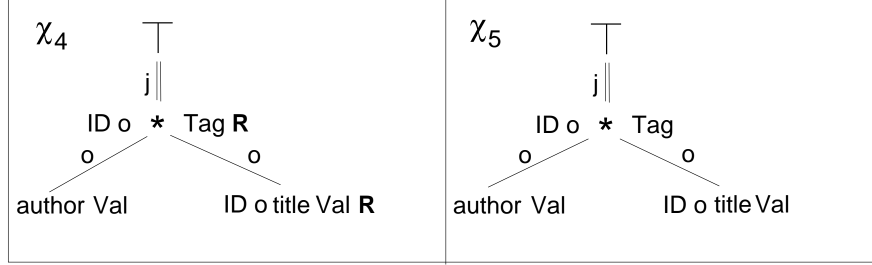



Figure 2.9: Sample XAM with access restrictions.

The information content of $\chi_4$ on the document $d$ from Figure 2.5, with the bindings list $[t_{B1}]$, is:

e$_1$(ID=2, Tag="book",
e$_2$[(Val="Abiteboul"), (Val="Suciu")],
e$_3$[(ID=4, Val="Data on the Web")])

Now consider another binding tuple $t_{B2}$ for $\chi_7$:

$t_{B2}$=e$_1$(Tag="article", e$_3$[(Val="Data on the Web")])

The information content of $\chi_4$ on document $d$ with the binding list $[t_{B2}]$ is empty, since $d$ does not contain any article called "Data on the Web".

Let $t_{B3}$ be the binding tuple:

$t_{B3}$=e$_1$(Tag="book", e$_3$[(Val="The Syntactic Web")])

The information content of $\chi_4$ on document $d$, with the bindings $[t_{B1}, t_{B3}]$ is:

[e$_1$(ID=2, Tag="book",
e$_2$[(Val="Abiteboul"), (Val="Suciu")],
e$_3$[(ID=4, Val="Data on the Web")]),
e$_1$(ID=7, Tag="book",
e$_2$[(Val="Tom Lerners-Bee")],
e$_3$[(ID=12, Val="The Web: next generation")]) ]

To formalize the above, we need the notion of *tuple intersection*. Let $t$ and $b$ be two tuples such that the signature of $b$ is a projection on the signature of $t$. Then, $t \cap b$

**Algorithm 1**: Data accessible from tuple $t$ with a binding tuple $b$

```
   input : t(a_1, a_2, ..., a_k), where a_{i_1}, a_{i_2}, ..., a_{i_n} are marked R;
           binding tuple t_B(a_{i_1}, a_{i_2}, ..., a_{i_n})
   output: t ∩ b
 1 t_RES ← 0_t
   /* t_RES is a tuple having t's type, atomic attributes set to ⊥, and collection
      attributes set to empty collections                                          */
 2 foreach a_{i_j} in a_{i_1}, a_{i_2}, ..., a_{i_n} do
       /* check if t matches the binding in b                                      */
 3     if a_{i_j} is of an atomic type then
 4         if t.a_{i_j} = b.a_{i_j} then
 5             t_RES.a_{i_j} ← t.a_{i_j}
 6         else
 7             return [ ]
 8     else
           /*a_{i_j}'s type is a list of tuples                                     */
 9         t_RES.a_{i_j} ← ⊔_{t'∈t.a_{i_j}, t''∈b.a_{i_j}} t' ∩ t''
10         if t_RES.a_{i_j} = [ ] then
11             return [ ]
12 foreach attribute a_j, 1 ≤ j ≤ k, a_j ∉ a_{i_1}, ..., a_{i_k} do
13     t_RES.a_j ← t.a_j
14 return [t_RES]
```

represents the data accessible from $t$ *given* $b$; this data can consist of zero or one tuple, containing (possibly part of) the data from $t$.[5]

Tuple intersection is described in Algorithm 1, which computes the data accessible from a tuple $t$, given a binding tuple $b$. If $t$ and $b$ disagree on the values of some atomic attributes, then no information from $t$ can be accessed using $b$ (lines 2-7 of the algorithm). This is similar to an unsuccessful index lookup, with a search key absent from the index. If $t$ and $b$ agree on their common atomic attributes, lines 8-11 describe which part of their common complex attributes can be obtained from $t$: the intersection of $t$'s and $b$'s values for these attributes ($\bigsqcup$ stands for list concatenation). Again, if such an intersection is empty, no data is reachable from $t$ using $b$. Finally, the values of $t$ attributes whose names do not appear in $b$'s types are accessible (lines 12-13).

[5]Notice that in this context, tuple intersection is not commutative.

We illustrate nested tuple intersection with an example.

Consider the following tuple $t$ and binding tuple $b_1$:

$t$=e$_1$(ID=2, Tag="book",
e$_2$[(Val="Abiteboul"), (Val="Suciu")],
e$_3$[(ID=4, Val="Data on the Web")])
$b_1$=e$_1$(ID=2, e$_2$[(Val="Suciu"), (Val="Buneman")])

The computation of $t \cap b_1$ initially sets:

$t_{RES}$=e$_1$(ID=$\perp$, Tag=$\perp$, e$_2$[ ], e$_3$[ ])

Applying lines 2-9 in Algorithm 1 transforms $t_{RES}$ into:

$t_{RES}$=e$_1$(ID=2, Tag=$\perp$, e$_2$[ ], e$_3$[ ])

and then successively into:

$t_{RES}$=e$_1$(ID=2, Tag=$\perp$,
e$_2$[(Val="Abiteboul") $\cap$ (Val="Buneman") $\bigsqcup$
(Val="Suciu") $\cap$ (Val="Buneman") $\bigsqcup$
(Val="Abiteboul") $\cap$ (Val="Suciu") $\bigsqcup$
(Val="Suciu") $\cap$ (Val="Suciu")],
e$_3$[ ]),

$t_{RES}$=e$_1$(ID=2, Tag=$\perp$, e$_2$[(Val="Suciu")], e$_3$[ ])

Lines 10-11 in Algorithm 1 copy $t$'s attributes not appearing in $b_1$ into $t_{RES}$, and thus:

$t_{RES}$=e$_1$(ID=2, Tag="book", e$_2$[(Val="Suciu")],
e$_3$[(ID=4, Val="Data on the Web")])

Finally, $t \cap b_1 = [t_{RES}]$.

We now formally define the semantics of a restricted-access XAM $\chi$ with respect to a set of bindings.

DEFINITION 2.2.6 (RESTRICTED XAM SEMANTICS). Let $\chi$ be a XAM with some required attributes, and $\chi^0$ be a XAM obtained from $\chi$ by erasing all **R** markers. Let $B$ be a list of binding tuples for $\chi$. The semantics of $\chi$ over a document $d$, with bindings $B$, is defined as:

$$[\![\chi(B)]\!]_d = \bigsqcup_{b \in B,\ t \in [\![\chi^0]\!]_d} t \cap b$$

◁

The following example illustrates restricted XAM semantics.

EXAMPLE 2.2.2 (RESTRICTED XAM SEMANTICS). Consider the XAM $\chi_4$ in Figure 2.9. Erasing all its **R** marks leads to the XAM $\chi_5$ shown next to it. Let $d$ be the document in Figure 2.5. By Definition 2.2.5, we have:

$[\![\chi_5]\!]_d$=[e$_1$(ID=2, Tag="book", e$_2$[(Val="Data on the Web")],
e$_3$[(ID=5,Val="Abiteboul"), (ID=6,Val="Suciu")]),
e$_1$(ID=7, Tag="book", e$_2$[(Val="The Syntactic Web")],
e$_3$[(ID=9, Val="Tom Lerners-Bee")]),
e$_1$(ID=10, Tag="phDThesis",
e$_2$[(Val="The Web: next generation")],
e$_3$[(ID=13, Val="Jim Smith")])]

Denoting the three tuples above as $t_1$, $t_2$ and $t_3$, we have $[\![\chi_5]\!]_d = [t_1, t_2, t_3]$. Let $B$ be the following bindings for $\chi_4$:

$B$=[e$_1$(Tag="book", e$_3$[(Val="Data on the Web")]),
e$_1$(Tag="book", e$_3$[(Val="The Syntactic Web")])]
=$[b_1, b_2]$

Applying Definition 2.2.6, we obtain:

$$\begin{aligned}[\![\chi_4(B)]\!]_d =& (t_1 \cap t_{B1}) \bigsqcup (t_2 \cap t_{B1}) \bigsqcup (t_3 \cap t_{B1}) \bigsqcup \\ & (t_1 \cap t_{B2}) \bigsqcup (t_2 \cap t_{B2}) \bigsqcup (t_3 \cap t_{B2}) \\ =& (t_1 \cap t_{B1}) \bigsqcup (t_2 \cap t_{B2}) = [t_1, t_2].\end{aligned}$$

◁

## 2.3 XAM expressive power

In this section, we demonstrate XAM expressive power by showing how several existing storage and indexing strategies can be expressed using XAMs. We organize this

presentation in three parts. Section 2.3.1 discusses existing relational storage models; Section 2.3.2 tackles the case of native storages, while Section 2.3.3 discusses the cases of several XML indexing schemes. Finally, Section 2.3.4 discusses the storage models that are not easy to cover using XAMs, explains why this happens and what are the consequences.

### 2.3.1 Modeling relational XML stores

Many XML-to-relational mappings have been explored in order to store XML documents in RDBMSs. All these techniques rely on the use of foreign keys to capture parent/child relationships, and represent node order by means of an explicit order attribute stored in the tables.

```
<book>
      <booktitle>Data on the web</>
       <author id="abitebou">
            <name>
              <first>Serge</first>
              <last>Abiteboul</last>
            </name>
       </author>
       <author id="suciu">
            <name>
              <first>Dan</first>
              <last>Suciu</last>
            </name>
       </author>
       ...
 </book>
```

Figure 2.10: Sample bib document.

**Schema-independent relational storage strategies** A first class of approaches for storing XML into relations do not rely on schema information. Among these, we describe in the sequel *Edge* and *Universal table* [48] which have been widely considerated in the literature and show how to express them using XAMs.

**The Edge approach and variants** One of the first attempts to store XML in relations is based on the *Edge* approach [48]. This storage scheme establishes two tables:

**Edge** (source, target, ordinal, name, flag)
**Value** (vID, value)

In this schema, the **Edge** table stores a tuple for each parent-child pair of nodes in an XML document. The simple (integer) IDs of the parent and the child provide the "source", respectively, "target" attributes. The "name" attribute designates the name of the child node. The "ordinal" field designates the order of the child among the children of its parent. The "flag" attribute signals whether the target node is an attribute or an element. If the child name is a value, then the "ordinal" attribute is a foreign key into the Value table, which stores all values (leaves) of the document. The XAMs modeling *Edge* are depicted in Figure 2.11(a). The first simple XAMs model the access to element and attribute values, the third XAM models the access to the XML elements, while the last corresponds to attributes. The authors further propose to index $Edge$ on the $source$ column, which can be modeled by a specification ID R in our notation. We omit the graphical XAM representation of this index.

In the *Universal* approach a single table is used to store all the edges. The table has the structure:

**Universal** $(source, ordinal_{n1}, flag_{n1}, target_{n1}, \ldots, ordinal_{nk}, flag_{nk}, target_{nk})$
**Value** (vID, value)

where $n_1, \ldots, n_k$ are the labels present in the document. Given that the universal table is defined in [48] as the full outerjoin of all *Edge* tables, the XAM for *UniversalTable* is depicted in Figure 2.11(b).

**Schema-driven relational storage schemes** Another category of XML storages uses DTD-driven mappings to store the XML documents in RDBMS. In this category we find the *Basic*, *Shared* and *Hybrid* approaches evaluated in [105], *XRel* [71] and *XParent* [66].

In [105], the decision of whether to create a table for an element, or to "inline" its information in the table corresponding to its parent, is made on the basis of whether the element is "shared" as a child' by other elements in the DTD. Since an element could be potentially inlined in several elements that refer to it, the *Basic*, *Shared* and *Hybrid* schemes differ in the degree of redundancy they support. For example, the

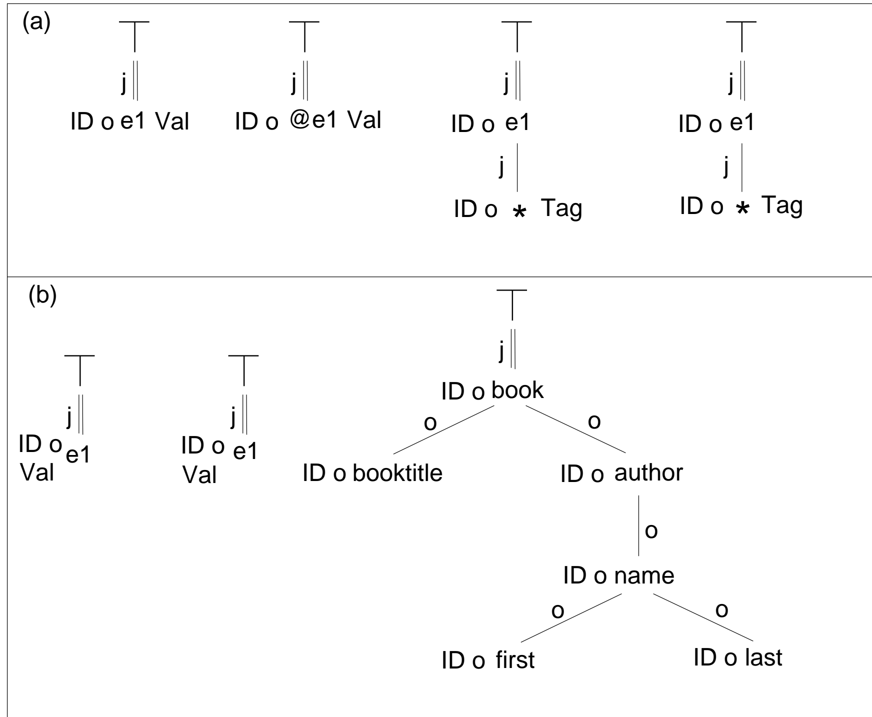



Figure 2.11: XAMs for the *Edge* (a) and *Universal table* (b) approaches.

*Basic* approach inlines as many descendants of an element as possible into a single table. However, relations are created for element because the XML document can be rooted at any element of the DTD. To see how XAMs model the *Basic* storage scheme, we consider the XML document in Figure 2.10. The corresponding relations are the following:

**Book**(bookID:integer, book.booktitle: string,
book.author.name.first: string, book.author.name.last :string)
**Booktitle**(booktitleID: integer, booktitle: string)
**First**(firstnameID: integer, firstname: string)
**Last**(lastnameID: integer, lastname: string)
**Name**(nameID: integer, name.first: string, name.last: string)
**Author**(authorID: integer, author.name.first: string,
author.name.last: string)

Each table from the relational storage is modeled by a XAM, as shown in Figure 2.12.

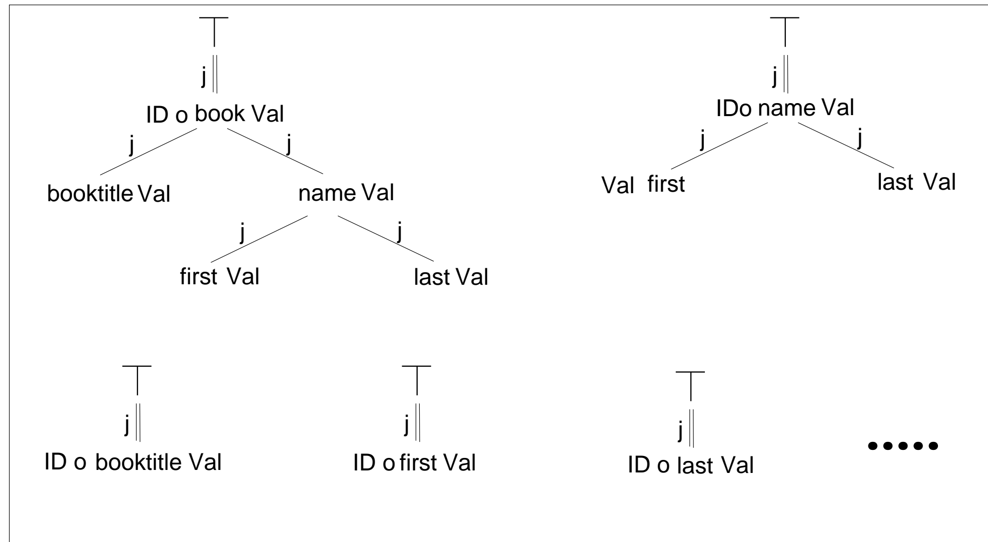



Figure 2.12: XAMs for the $Basic$ storage scheme.

The *XRel* [71] and *XParent* [66] storage schemes are based on storing information about paths in a RDBMS. In this approach, the Path table stores all the parent-child paths starting from the root, encountered in the XML document. To each path is assigned a numeric ID. The other relations store information describing the elements, attributes, respectively, text nodes of the document. A tuple in each of the Element, Attribute and Text tables includes a foreign key, pointing to the Path table. Moreover, $(start, end)$ IDs are assigned to element, attributes, and text nodes. The schema of *XRel* stored relations is:

| | |
|---|---|
| **Element** | (docID, pathID, start, end, index, reindex) |
| **Attribute** | (docID, pathID, start, end, value) |
| **Text** | (docID, pathID, start, end, value) |
| **Path** | ( pathID, pathexpr) |

## 2.3.2 Modeling native XML stores

While relational XML stores rely on an RDBMS for storing the data, many alternatives, commonly termed "native" stores, have been proposed. In this section we study the expressive power of XAMs to model such cases.

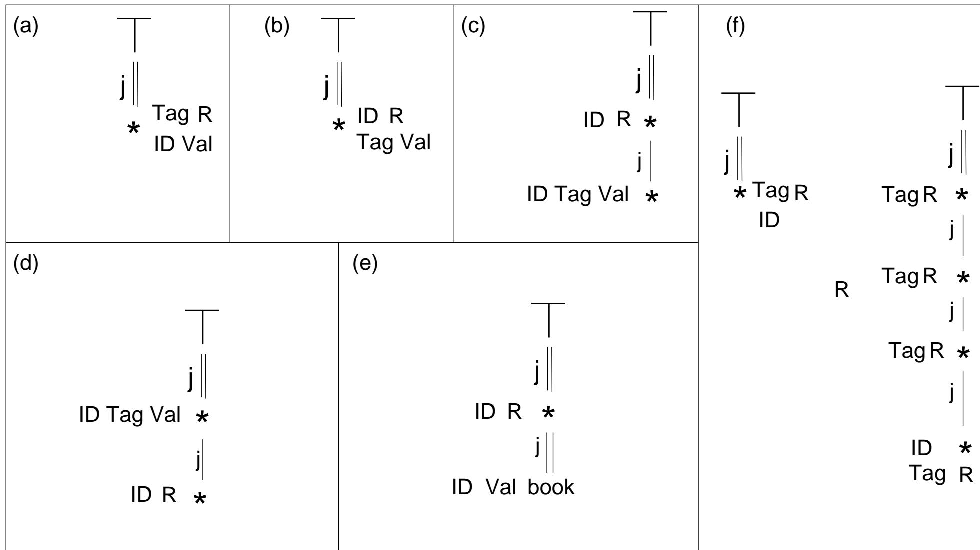



Figure 2.13: XAMs for DOM (a-e) and path partitioning (f).

**The DOM model** The Document Object Model (DOM) [117] is a common tree-based application programming interface (API). It provides simple access methods to the tree's nodes. We discuss it here, as there are many XML processing engines accessing their data through a DOM tree [96, 97, 101, 106].

The DOM model provides access to the children of a given node, access to elements of a given tag, and navigation among siblings. For modeling this with XAMs, we assimilate a pointer to a DOM node (of type Element) to the element's unique ID. XAMs allow to model such access methods, with the exception of sibling navigation. For example, the $GetElementsByTagName$ DOM primitive provides access to a set of elements, if we know the value of their tag. This can be modeled by the XAM in Figure 2.13(a).

Furthermore, DOM provides parent-to-child and child-to-parent navigation (the $getParentNode$ and $getChildNodes$ primitives). We describe these access methods by a 2-node XAM, one node for the parent and one for the child, as shown in Figure 2.13(c) and Figure 2.13(d). Finally, DOM gives access to a descendant of a node, if we know the node ID and the descendant tag name. The XAM for modeling this is similar, and is shown in Figure 2.13(e).

**Tag partition based storage models** This method has been used in many systems, such as Timber [63, 64] and Natix [46]. These systems partition the data according to the tag name of the XML elements and use the tag name as an index of element content.

Conceptually, the effect of tag partitioning is very similar to the $GetElementsByTagName$ DOM primitive, thus the model in Figure 2.13(a) is still applicable.

**Path partition based storage models (PP)** Path-partitioned storages have been proposed in the early versions of Monet [103] and in the XQueC [14, 17, 83] systems. Such storage models partition XML data and content according to the document *paths*. Furthermore, the XQueC system uses structural identifiers and organizes the path-partitioned contents into ordered sequences. Thus, for any path of the form $/tag_1/tag_2/.../tag_k$, where $tag_1$ is the root element tag, $k \geq 1$, and all tags are connected by / only, we can access the IDs of the elements of tag $tag_k$. Similarly, for any path of the form $/tag_1/tag_2/.../tag_k/\#PCData$ or $/tag_1/tag_2/.../tag_k/@attr$ (where $attr$ is an attribute name), we have access to $(ID, value)$ pairs, where $value$ is the text content of element $tag_k$ (in the case of $\#PCData$), or the value of attribute (in the case of $@attrName$). The access to IDs of nodes on a given path is modeled by the XAM in Figure 2.13(f).

*Alternatives when modeling path partitioning with XAMs* We can see that for the path partitioning stores we have two distinct possible XAM descriptions, with distinct degrees of genericity:

- We may use one XAM for each *length* of simple parent-child path encountered in the document, as illustrated in Figure 2.13(f). In this case, the labels have to be known in order to obtain the tuples corresponding to a path. This encoding is quite compact (few XAMs are used).
- A more precise modeling is to use element filtering provided by $[Tag = value]$ predicates. This leads to a XAM for each simple path in the document, as illustrated in Figure 2.14. We prefer this description, due to its extra precision.

### 2.3.3 Modeling XML indices

Many indexing approaches have been considered in the proposed XML storage systems. In order to explain how we can model these indexing methods with XAMs, we

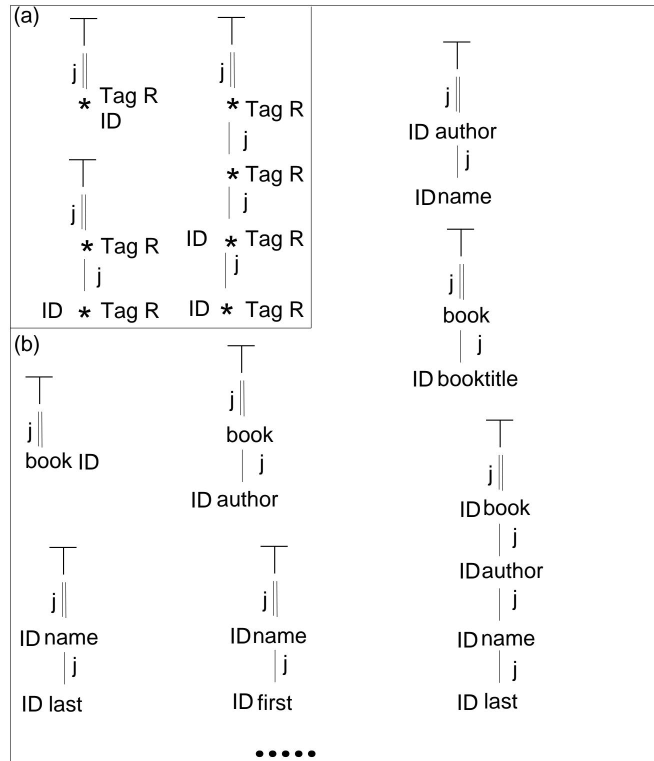



Figure 2.14: XAM description of the Path-Partitioned storage of a simple document: a compact representation (a), and the preferred representation (b).

omit details and group these approaches into three different categories, based on the structures employed in indexing.

**Node-based indexing methods** Earlier indexing methods for XML documents considered single elements/attributes as the basic unit of query. Complex path expressions were decomposed in collections of basic path expressions. Only the access to a single element or attribute is directly provided by the index structure. This can be considered similar to the $getElementByTag$ primitive from the DOM model discussed in Section 2.3.2.

*XISS* [76] proposes a new numbering scheme extending the structural IDs for supporting updates. The major indexing structures in XISS are:

- *An element index*: for a given element name T, find the list of elements having this name ; this is similar to the $getElementByTag$ DOM primitive as discussed in Section 2.3.2.
- *An attribute index*: for a given attribute name A, find the list of attributes having the same name.
- *A structural index*: for a given element find its parent element and child elements or attributes belonging to the element.
- *A name index*: this is a dictionary of all elements/attributes names occuring in the document. Each of them has associated an integer ID, which is subsequently used for query processing. This allows avoiding to perform expensive string comparisons.
- *A value index*: is a similar dictionary over the string values (attributes values or text values) found in the XML document.

The associated XAMs are depicted in Figure 2.15. We note that the *name index* cannot be modeled by a XAM since XAMs only assign IDs to nodes not to distinct values. From this viewpoint, XAMs are closer to the XQuery Data Model [119] than to lower-level techniques such as dictionaries.

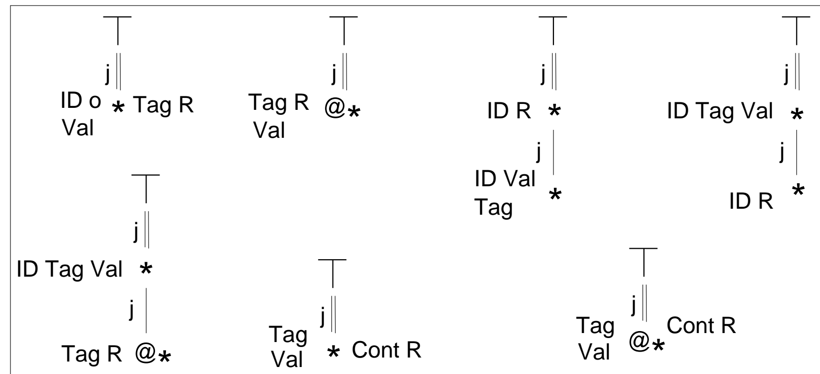



Figure 2.15: XISS indexes: Element index (a), attribute index (b), structural index (c-e), value index (f).

**Path-based indexing methods** Paths were also used as the basic indexing unit. In this approach, the indexing engine provides concise summaries of path structures that

the query processor uses to speed up the traversal of the irregularly structured XML data.

*DataGuides* [52] and *1-indexes* [86] are general path indexes that represent all paths starting from the root of an XML document. Their functionality, from the data access point of view, is thus equivalent to the functionality presented for the path partition based storages discussed in Section 2.3.2. The XAMs that model these indices are thus the same as those presented for path partition. (Figure 2.14).

The *template Index (T-Index)* [86] generalizes dataguides, providing a flexible tradeoff between space and generality. Given a query template of the form:

$$select\ x\ from\ t = (*.book)x(name/last[val = "Suciu"])$$

A *T-index* could be established to provide direct access to the result of this query. Observe that in this case the *T-index* plays a role reminiscent of materialized views. The XAM describing such a *T-index* is depicted in Figure 2.16.

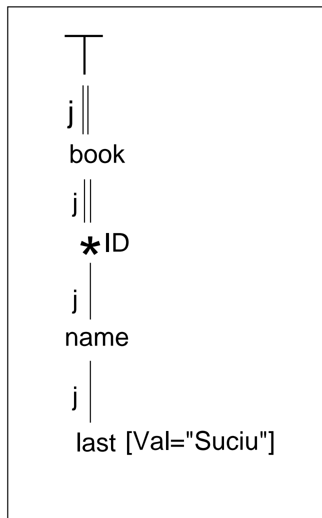



Figure 2.16: A *T-index* for a specific query.

Some other approaches establish indexes for the frequently used paths, possibly including branches. Two representatives are *IndexFabric* [37] and *APEX* [36]).

*Index Fabric* encodes paths as strings and inserts them into a special index, a Patricia trie-like structure. In this approach two kinds of index paths are considered:

- Raw paths - are root to leaf paths from the XML document. These are encoded using for each element name an unique designator (a prefix encoded in a string identifier). The Patricia trie index on these paths can be modeled by the XAM used for the path partition stores, as depicted in Figure 2.17(b).
- Refined paths - path expressions (possibly including ancestor-descendant steps) which have been used by the queries. Some of these paths may also be added to the Index Fabric.

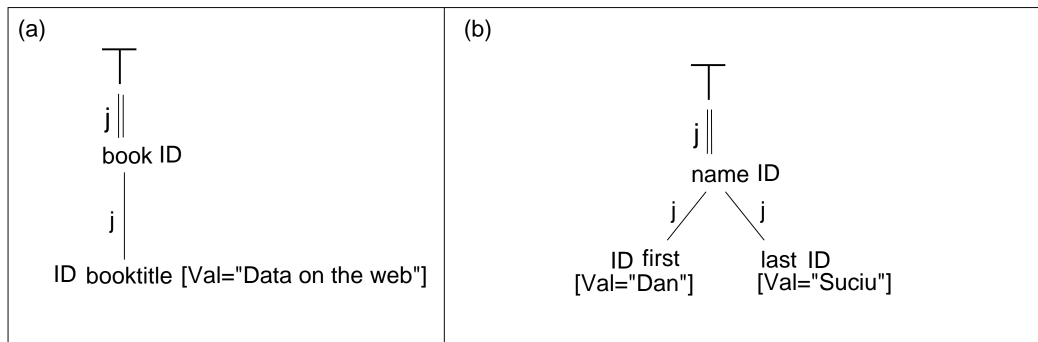



Figure 2.17: Index Fabric: raw path index (a), refined path index (b).

*APEX* [36] is an indexing technique similar to the *refined path*-based indexes, using only the frequently traversed paths to improve query performance. In contrast with *Index Fabric*, it uses data mining algorithms to summarize the frequently used paths. *APEX* also guarantees to maintain all paths of length two. The XAMs for this indexing technique are similar to those that model the *refined path*-based indexes of *Index Fabric*.

**Tree pattern-based indexing methods** To avoid decomposing a query into several subqueries, and then join the subqueries, other approaches rely on indexing full tree structures. We describe ViST [113] that is based on the *refined paths* approach introduced in *Index Fabric*. Instead of using workload information to determine the frequently occuring multiple-path queries, in *ViST* the XML documents, and the XML queries, are encoded in a sequential representation. Thus, querying the document is equivalent to finding subsequence matches between the two encoded entities (XML data and XML query). From the data access point of view, *ViST* is equivalent to an *Index Fabric* index, with a refined path for all possible branching queries with differently-

named children at any level. Observe that the string encoding of *ViST* leads to the same false positives, since not all the structural information is preserved (see [99] for details)

### 2.3.4 XAM limitations

In this section, we discuss some storage and indexing models that the XAM formalism does not capture, and we comment on the reasons and the impact of these limitations.

**Backward navigation** The F&B index [68] is a very generic index structure for XML. The F&B index groups together all identifiers of XML elements that *cannot be separated* by any query performing downward and upward navigation in the document. This grouping condition is very strong, and, as a consequence, the F&B index is often too large to be useful. The authors have investigated several weaker notions of grouping, which keep forward and backward navigation in the query language, but group nodes only when they are not discernable by *some* queries (e.g. the queries in the workload, or the queries of depth smaller than some fixed $k$).

To keep XAMs clear and simple, we relied on a tree pattern formalism which does not offer a direct way of expressing backward navigation. However, in the presence of a set of structural constraints (such as a DTD or a path summary), it is possible to express the nodes reachable by a given forward-and-backward navigation query by a XAM.

**Storage models based on bisimulation** The XAMs are defined based on an algebraic formalism, while the F&B index is defined based on bisimulation [2]. Another storage model based on bisimulation is presented in [54]. We preferred to rely on a simple algebraic model, since it is at the same time easier to understand, to implement, and to use for query rewriting purposes. Incorporating bisimulation in the XAM formalism would likely complicate it a lot; furthermore, the F&B index and the compressed instance of [54] do not capture specific storage properties such as structural identifiers.

**Sibling order** XAMs do not allow to say e.g. that "the first two authors of an article are stored with the article, the third one is stored in a different place". This type of storage may be obtained by storage models such as STORED [39], which applies data mining techniques to extract the structural paterrns most frequently appearing in the data. However, there is a workaround, which is based on the structural constraints that

the document satisfies (remember that a path summary, which is the basis of our work, can be extracted even if an XML schema or DTD is absent, as was the case in Stored).

Consider that the structural constraints say:

$\mathcal{C}$ book ::= (author+) title

that is, a book may have one or more authors, and a single title. Then, we replace this by:

$\mathcal{C}'$
book ::= book1 | book2 | book3
book1 ::= author1 title
book2 ::= author1 author2 title
book3 ::= author1 author2 (author+) title
author1 ::= author
author2 ::= author

The set of documents conforming to $\mathcal{C}$ is the same as the set of documents conforming to $\mathcal{C}'$, and it is now possible to write a XAM that specifies how "author1" and "author2" elements are stored with their parent, whatever the parent's tag is.

# Chapter 3

# From XQuery to XML Access Modules

In the previous chapter we have introduced the XAM language which allows describing complex tree-pattern-organized materialized views. To take advantage of such views, one has to understand which views can be used for a query $q$. The approach followed in our work consists of extracting from $q$ a set of *query patterns* $p_{q1}, \ldots, p_{qn}$, and rewriting every query pattern $p_{qi}$ using the view patterns $p_{v1}, \ldots, p_{vm}$. The first step (query-to-pattern translation) is crucial. Intuitively, the *bigger* the query patterns, the *bigger* the view(s) that can be used to rewrite them, thus the less computations remain to be applied on top of the views.

The contribution of this chapter is a provably correct algorithm identifying tree patterns in queries expressed in a large XQuery subset. The advantage of this method over existing ones [22, 33, 93] is that the patterns we identify are strictly larger than in previous works, and in particular may span over nested XQuery blocks, which was not the case in previous approaches. We ground our algorithm on an algebra, since (as we will show) the translation is quite complex due to XQuery complexity, and straightforward translation methods may lose the subtle semantic relationships between a pattern and a query.

The chapter is organized as follows. Section 3.1 motivates the need for pattern recog-

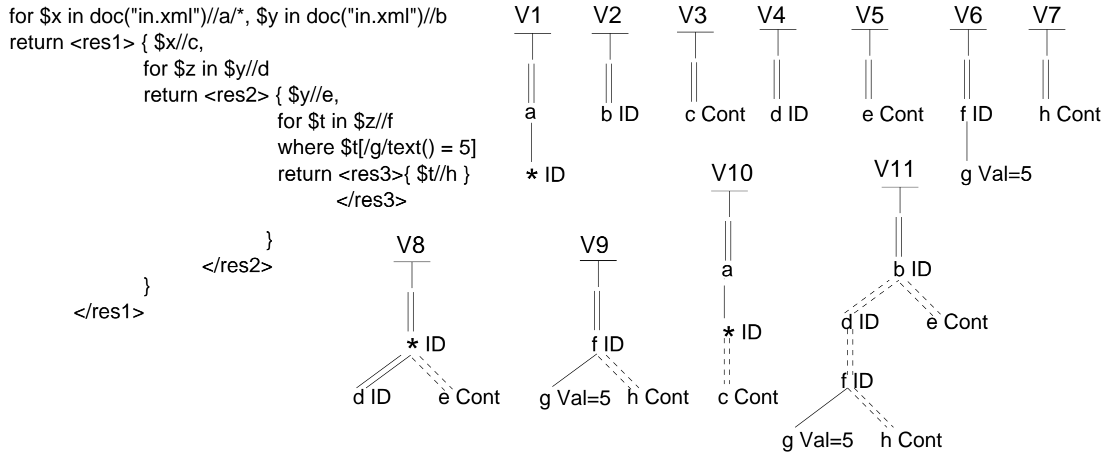



Figure 3.1: Sample XQuery query and corresponding tree patterns/views.

nition in XQuery queries. Section 3.2 describe the query language handled by our approach. The translation algorithm is presented in Section 3.3.

## 3.1 Motivating example

We illustrate the benefits of our tree pattern extraction approach on the sample XQuery query in Fig. 3.1, featuring three nested for-where-return blocks. An important XQuery feature to keep in mind is that when a return clause constructs new elements, if an expression found in the element constructor evaluates to $\emptyset$ (the empty result), an element must still be constructed, albeit with no content corresponding to that particular expression. For instance, in Fig. 3.1, if for some bindings of the variables $\$x$ and $\$y$, the expression $\$x//c$ yields an empty result, a res1 element will still be constructed, with no content corresponding to $\$x//c$ (but perhaps with some content produced by the nested for-where-return expression).

Next to the query, Fig. 3.1 depicts eleven possible corresponding query tree patterns expressed in the XAM formalism introduced in Chapter 2.2. Observe that the patterns $V_1$ to $V_7$ are conjunctive, while $V_8$-$V_{11}$ also include optional edges.

**Patterns for views and queries** As previously mentioned, patterns play a dual role in our approach: view definitions, and query sub-expressions. Thus, each pattern $V_1, \ldots, V_{11}$ is a subexpression of the query at left, and (for our explanation) we also assume it is available as a materialized view. When a pattern is interpreted as a view, we say it *stores* various $ID$, $Cont$ and $Val$ attributes; when it is interpreted as a query subexpression, we say it *needs* such attributes.

Several existing view-based XML query rewriting frameworks [22, 120] concentrate on XPath views, storing data for one pattern node only (since XPath queries have one return node), and lacking optional edges. Similar indexes are described in [37, 68]. In Fig. 3.1, $V_1$-$V_7$ are XPath.

The algorithm which we will present in this chapter extracts from the query in Fig. 3.1 two patterns: $V_{10}$ and $V_{11}$. Based on these, we rewrite the query by a single join (more exactly, a cartesian product) of the corresponding $V_{10}$ and $V_{11}$ views. As we will see when comparing our approach with similar related works in Section 6, extracting large patterns with optional nodes is more efficient than if we were restricted to XPath patterns.

Observe that the pattern extraction process must be grounded o a formal model. This is needed to ensure the patterns have exactly the same meaning as query subexpressions, or, when this is not the case, to compute *compensating actions* on the views. For instance, consider $V_{11}$ in Fig. 3.1. Here, the $d$ and the $e$ nodes are optional descendants of the $b$ nodes, and so they should be, according to the query. However, due to the query nesting, no $e$ element should appear in the result, if its $b$ ancestor does not have $d$ descendants. This $d \rightarrow e$ dependency is not expressed by $V_{11}$, and is not expressible by any tree pattern, because such patterns only account for ancestor-descendant relationships. Thus, $V_{11}$ is the best possible tree pattern view for the part of the query related to variable $\$y$, yet it is not exactly what we need. An (inexpensive) selection on $V_{11}$, on the condition $(d.ID \neq \bot) \vee (d.ID = \bot \wedge e.Cont = \bot)$, needs to be applied to adapt the view to the query.

For simplicity, in this section, no nesting or grouping has been considered, neither in the patterns in Fig. 3.1, nor in the query rewriting strategies. However, given that

XQuery does construct complex grouped results (e.g., all $c$ descendants of the same $\$x$ must be output together in Fig. 3.1), pattern models considered in [93], as well as our translation method, do take nesting into account.

## 3.2 Query language

The target language for our tree pattern extraction process is an XQuery subset we denote $\mathcal{Q}$, obtained as follows:

1. XPath$^{\{/,//,*,[\,]\}} \subset \mathcal{Q}$, that is, any core XPath [85] 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. Navigation branches enclosed in [ ] may include complex paths and comparisons between a node and a constant $c \in \mathcal{A}$. Sample queries from this subset are $/a/b$ or $//c[//d/text() = 5]/e$. Observe that 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 [121] 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 $b$ descendants 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 $t \in \mathcal{L}$ 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 3.1 illustrates category (5) of our supported XQuery fragment.

## 3.3 Pattern extraction algorithm

Our algorithm proceeds in two steps. First, $\mathcal{Q}$ queries are translated into expressions in the algebra described in Section 1.2.2; Sections 3.3.1 and 3.3.2 explain this translation. Second, algebraic equivalence and decomposition rules are applied to identify, in the resulting expressions, subexpressions corresponding to tree patterns. The algebraic rules are quite straightforward. The ability to recognize pattern subexpressions is due to the formal algebraic XAM pattern semantics provided in Chapter 2.2. Section 3.3.3 illustrates it on our running example.

Recall from Section 1.2.2 that $\mathcal{A}$ denotes the set of atomic values used in our algebra. Queries are translated to algebraic expressions producing one $\mathcal{A}$ attribute, corresponding to the serialized query result. We describe query translation as a translation function $alg(q)$ for every $q \in \mathcal{Q}$. We will also use an auxiliary function $full$; intuitively, $full$ returns "larger" algebraic expressions, out of which $alg$ is easily computed.

### 3.3.1 Algebraic translation of path queries

For any $q \in \mathcal{P}$, let $ret(q)$ denote the return node of $q$. Let $d$ be a document, and $a$ an element name. Then:

$$full(d//*) \stackrel{\text{def}}{=} e, \text{ and } alg(d//*) \stackrel{\text{def}}{=} \pi_C(e)$$
$$full(d//a) \stackrel{\text{def}}{=} (e_a), \text{ and } alg(d//a) \stackrel{\text{def}}{=} \pi_C(e_a)$$

Translating $d/*$ and $d/a$ requires care to separate just the root element from $e$:

$$full(d/*) \stackrel{\text{def}}{=} e_1 \setminus \pi_{e_3}(e_2 \bowtie_{e_2.ID \prec e_3.ID} e_3), \text{ and } alg(d/*) \stackrel{\text{def}}{=} \pi_C(full(d/*))$$

where $e_1, e_2$ and $e_3$ are three occurences of the $e$ relation, $e_2.ID$ (respectively, $e_3.ID$) is the $ID$ attribute in $e_2$ (respectively $e_3$), and the projection $\pi_{e_3}$ retains only the attributes of $e_3$. The set difference computes the $e$ tuple corresponding to the element that does not have a parent in $e$ (thus, the root element). Similarly,

$$full(d/a) \stackrel{\text{def}}{=} e_a \setminus \pi_{e_3}(e_2 \ltimes_{e_2.ID \prec e_3.ID} e_3), \text{ and } alg(d/a) \stackrel{\text{def}}{=} \pi_C(full(d/a))$$

In general, for any $\mathcal{P}$ query $q$:

- If $q$ ends in $text()$, then $alg(q) = \pi_{V_{last}}(\pi^0(full(q)))$, where $V_{last}$ is the $V$ attribute from the $e_d$ relation corresponding to $ret(q)$. The inner projection $\pi^0$ eliminates possible duplicate nodes, in accordance with XPath semantics [121]. The outer projection ensures only the text value is retained.
- If $q$ does not end in $text()$, then $alg(q) = \pi_{C_{last}}(\pi^0(full(q)))$, where $C_{last}$ is the $C$ attribute from the $e_d$ relation corresponding to $ret(q)$.

Note that the resulting algebraic expressions return node *value* or *content*, while in general XPath queries may return *nodes*. Alternatively, node identifiers can be returned by setting, for node-selecting XPath queries, $alg(q) \stackrel{\text{def}}{=} \pi_{ID_{last}}(\pi^0(full(q)))$,

where $ID_{last}$ is the $ID$ attribute from the $e_d$ relation corresponding to $ret(q)$. Since XPath results frequently need to be returned in a serialized form, e.g., to be shown to a user, or sent in a Web service, we consider the $C$ attribute is really returned, thus use $\pi_{C_{last}}$ in the translation.

We now focus on defining the $full$ algebraic function for path queries, keeping in mind how $alg$ derives from $full$ for such queries. For any query $q \in \mathcal{P}$, we have:

$$full(q//a) \stackrel{\text{def}}{=} full(q) \bowtie_{e_q.ID \prec\prec e_a.ID} e_a$$

where $e_q.ID$ is the ID attribute in $full(q)$ corresponding to $ret(q)$, while $e_a.ID$ is the ID from the $e_a$ relation at right in the above formula. When $//$ is replaced with $/$, the translation involves $\prec$ instead of $\prec\prec$. We also have:

$$full(q[text() = c]) \stackrel{\text{def}}{=} \sigma_{V=c}(full(q))$$

If $q_1 \in \mathcal{P}$ and $q_2 \in \text{XPath}^{\{/,//,*,[]\}}$ is a relative path expression starting with a child navigation step, we have:

$$full(q_1[q_2]) \stackrel{\text{def}}{=} full(q_1) \ltimes_{e_1.ID \prec e_2.ID} full(//q_2)$$

where $e_1.ID$ is the ID corresponding to $ret(q_1)$, $//q_2$ is an absolute path expression obtained by adding a descendant navigation step, starting from the root, in front of $q_2$, and $e_2.ID$ corresponds to the first node of $q_2$. Here and from now on, we consider all relative path expressions start with a child step. If the first step is to a descendant, $\prec$ should be replaced with $\prec\prec$ in the translation.

Let $\$x$ be a variable bound to the result of query $q_{\$x}$, and $q$ be a relative path expression starting with a child navigation step. Then:

$$full(\$x\ q) \stackrel{\text{def}}{=} full(q_{\$x}) \bowtie_{e_1.ID \prec e_2.ID} (full(q))$$

where $e_1.ID$ is the ID corresponding to $ret(q_{\$x})$, and $e_2.ID$ is the ID corresponding to the top node in $full(q)$.

EXAMPLE 3.3.1. Consider the path expressions $p_{\$x}$ =//$a/*$, $p_{\$y} = //b$, $p_{\$z} = \$y$//$d$ and $p_{\$t} = \$z$//$f$ (see Fig. 3.1). Applying the above rules, we obtain:

$$full(p_{\$x}) = e_a \bowtie_{e_a.ID \prec e.ID} e, \quad full(p_{\$y}) = e_b$$
$$full(p_{\$z}) = full(p_{\$y}) \bowtie_{e_{\$y}.ID \ll e_d.ID} e_d,\ full(p_{\$t}) = full(p_{\$z}) \bowtie_{e_{\$z}.ID \ll e_f.ID} e_f$$

Now consider the path expressions $p_1 = \$x//c$, $p_2 = \$y//e$, $p_3 = \$t[g/text() = 5]$ and $p_4 = \$t//h$, also extracted from the query in Fig. 3.1. We have:

$$full(p_1) = full(p_{\$x}) \bowtie_{e_{\$x}.ID \ll e_c.ID} e_c,\ full(p_2) = full(p_{\$y}) \bowtie_{e_{\$y}.ID \ll e_e.ID} e_e,$$
$$full(p_3) = full(p_{\$t}) \ltimes_{e_{\$t}.ID \prec e_g.ID} \sigma_{V=5}(e_g),$$
$$full(p_4) = full(p_{\$t}) \bowtie_{e_{\$t}.ID \ll e_h.ID} e_h$$

In the above, $e_{\$y}$, $e_{\$z}$ and $e_{\$t}$ are the $e$ relations corresponding to the return nodes in the translations of $p_{\$x}$, $p_{\$y}$ and $p_{\$t}$. The $alg$ expressions are easily obtained from $full$. $\triangleleft$

$xq_1$ for $\$x_1$ in $p_1$, $\$x_2$ in $\$x_1/p_2, \ldots, \$x_k$ in $\$x_1/p_k$
where $\$x_1/p_{k+1}\,\theta\,\$x_1/p_{k+2}$ and $\ldots\ \$x_1/p_{m-1}\,\theta\,\$x_1/p_m$
return $\$x_1/p_{m+1}, \$x_1/p_{m+2}, \ldots, \$x_1/p_n$

$$full(xq_1) \stackrel{\text{def}}{=}$$
$$\sigma_{A_{k+1}\,\theta\,A_{k+2},\ldots,A_{m-1}\,\theta\,A_m}(full(p_1) \bowtie_{ID_1 \prec ID_2} full(//p_2) \ldots \bowtie_{ID_1 \prec ID_k} full(//p_k)$$
$$\bowtie^n_{ID_1 \prec ID_{k+1}} full(//p_{k+1}) \bowtie^n_{ID_1 \prec ID_{k+2}} \cdots \bowtie^n_{ID_1 \prec ID_m} full(//p_m)$$
$$\bowtie^n_{ID_1 \prec ID_{m+1}} full(//p_{m+1}) \bowtie^n_{ID_1 \prec ID_{m+2}} \cdots \bowtie^n_{ID_1 \prec ID_n} full(//p_n)\ )$$

$$alg(xq_1) \stackrel{\text{def}}{=} \pi_{A_{m+1},A_{m+2},\ldots,A_n}(full(xq_1))$$

Figure 3.2: Generic XQuery query with simple return expression.

### 3.3.2 Algebraic translation of more complex queries

This section describes the translation of $\mathcal{Q}$ queries other than path expressions.

**Concatenation** We have $alg(q_1, q_2) \stackrel{\text{def}}{=} alg(q_1) \parallel alg(q_2)$ and $full(q_1, q_2) \stackrel{\text{def}}{=} full(q_1) \parallel full(q_2)$, where , denotes query concatenation, and $\parallel$ concatenation of tuple lists.

**Element constructors** Element constructor queries are translated by the following rule:

$$alg(\langle t\rangle\{q\}\langle /t\rangle) \stackrel{\text{def}}{=} xml(n(alg(q)), \langle t\rangle A_1 \langle /t\rangle)$$

where the nest operator $n$ packs all tuples from $alg(q)$ in a single tuple with a single collection attribute named $A_1$. The second argument of the $xml$ operator is a tagging template, indicating that values of the attribute named $A_1$ have to be packed in $t$ elements. Furthermore, $full(\langle t\rangle\{q\}\langle /t\rangle) = n(full(q))$.

**For-where-return expressions** The translation rules for such query expressions are outlined in Fig. 3.2 and Fig. 3.3. For simplicity, these rules use a single $\theta$ symbol for some arbitrary, potentially different, comparison operators.

**(1) Simple return clauses** In the generic query $xq_1$ (Fig. 3.2), path expression $p_1$ is absolute, while all others are relative and start from a query variable $\$x_1$. Attribute $ID_1$ corresponds to $ret(p_1)$. The query returns bindings for some variables. Attribute $ID_i$ is the attribute in $full(//p_i)$ corresponding to the top node of $p_i$, for every path expression $p_i$ in $p_2, \ldots, p_m$. Attributes $A_{k+1}$, $A_{k+2}$, $\ldots$, $A_m$ are those returned by the algebraic translations of the relative path expressions of the where clause, more precisely, the attributes in $alg(//p_{k+1})$, $alg(//p_{k+2})$, $\ldots$, $alg(//p_m)$. Each $A_{k+i}$ is $V$ or $C$, depending on $p_{k+i}$. Note that once $//$ is added in front of such a relative path expression, $//p_{k+i}$ is an absolute expression, thus translatable to the algebra. The child navigation step connecting $\$x_1$ and an expression $p_{k+i}$ is captured by the join $\bowtie^n_{ID_1 \prec ID_i}$. As an effect of this nested structural join, $A_i$ may be nested in $\sigma$'s input, therefore, the selection has existential semantics (recall the $map$-based extension of $\sigma$ to nested attributes from Section 1.2.2).

The $xq_1$ rule easily extends to queries where the for clause features several unrelated variables, the where clause contains predicates over one or two variables, and the return clause returns only variables. Each subquery corresponding to an independent variable in the for clause is then translated separately, and the resulting expressions

| | |
|---|---|
| $xq_2$ | for $\$x$ in $p_f$ where $pred(p_w(\$x))$<br>return $fwr(\$x)$ |
| $full(xq_2) \stackrel{\text{def}}{=} \sigma_{pred}(full(p_f) \bowtie^{n}_{ID_1 \prec ID_2} full(//p_w) \bowtie_{ID_1=ID_1} full(fwr(p_f))$<br>$alg(xq_2) \stackrel{\text{def}}{=} \pi_{fwr}(full(xq_2))$, respectively $xml_{templ(fwr)}(\pi_{fwr}(full(xq_2)))$ | |
| $xq_3$ | for $\$x$ in $p_f$ where $pred(p_w(\$x))$<br>return $\langle a \rangle$ { $fwr(\$x)$ } $\langle /a \rangle$ |
| $full(xq_3) \stackrel{\text{def}}{=} \sigma_{pred}(full(p_f) \bowtie^{n}_{ID_1 \prec ID_2} full(//p_w) ⟕^{n}_{ID_1=ID_1} full(fwr(p_f))$<br>$alg(xq_3) \stackrel{\text{def}}{=} xml_{\langle a \rangle \cdot \langle /a \rangle}(\pi_{fwr}(full(xq_3)))$, resp. $xml_{\langle a \rangle templ(fwr) \langle /a \rangle}(\pi_{fwr}(full(xq_3)))$ | |

Figure 3.3: Generic XQuery queries with complex return clauses.

are joined. From now on, without loss of generality, we will use one variable in each for clause; adding more variables depending on the first one leads to structural join subexpression in the style of $full(xq_1)$, while adding more unrelated variables leads to value joins as sketched above.

**(2) Nested for-where-return queries** Such queries are illustrated by $xq_2$ and $xq_3$ in Fig. 3.3. Here, $p_f$ is an absolute path expression, $p_w$ a relative one, $pred$ a simple comparison predicate, and $fwr$ a (potentially complex, nested) for-where-return query.

**(2.1) The outer query does not construct new elements** This is the case for $xq_2$ in Fig. 3.3. In $full(xq_2)$, $ID_1$ corresponds to $ret(p_f)$ and $ID_2$ to the top node in $p_w$. We add $//$ in front of $p_w$ to make it absolute. The query $fwr(p_f)$ is obtained from $fwr$ by adding a new "for" variable $\$x'$ bound to $p_f$, and replacing $\$x$ by $\$x'$. Thus, $fwr(p_f)$ is decorrelated from (it does no longer depend on) $\$x$; the dependency is replaced by the join on $ID_1$. In $alg(xq_2)$, the projection $\pi_{fwr}$ retains only the attributes from $alg(fwr(p_f))$. Two alternatives exist for $alg(xq_2)$, as shown in Fig. 3.3:

- If $fwr$ does not construct new elements, the query (and its translation) recall $xq_1$.
- If $fwr$ constructs new elements, the top operator in $alg(fwr(p_f))$ is $xml_{templ(fwr)}$, for some given tagging template $templ(fwr)$. In this case, $full(xq_2)$ is built us-

ing exactly the same template. Note how the XML constructor "sifts up" as the top algebraic operator in the translation, in this case, from $alg(fwr)$ to $alg(xq_2)$. All algebraic translations have at most one $xml$ operator.

**(2.2) The outer query constructs new elements** Query $xq_3$ in Fig. 3.3 encloses the results of some correlated query $fwr(\$x)$ in $\langle a \rangle$ elements. Therefore, in $full(xq_3)$ an outerjoin is used to ensure that $xq_3$ produces some output even for $\$x$ bindings for which $fwr(\$x)$ has an empty result. The outerjoin is nested, because all the results of $fwr(\$x)$ generated for a given $\$x$ must be included in a single $\langle a \rangle$ element. For $alg(xq_3)$, there are again two cases. If $fwr$ does not construct new elements, and since $xq_3$ does, the tagging template is a simple $\langle a \rangle$ element. If $fwr$ also constructs some elements, the $xml$ operator in $alg(xq_3)$ builds a bigger tagging template, by enclosing $templ(fwr)$ in an $\langle a \rangle$ element.

The translation of more complex $\mathcal{Q}$ queries can be derived from the above rules.

**Example.** Let us translate the query $q$ in Fig. 3.1 (also recall the path expressions at the end of Section 3.3.1, and their translations). We can write $q$ as:

for $\$x$ in $p_{\$x}$, $\$y$ in $p_{\$y}$
return ⟨res1⟩{ $p_1$,
⟨res2⟩{ $p_2$, for $\$z$ in $p_{\$z}$ where $p_3$ return ⟨res3⟩{ $p_4$ }⟨/res3⟩}⟨/res2⟩
}
⟨/res1⟩

which can be furthermore abstracted into:

for $\$x$ in $p_{\$x}$, $\$y$ in $p_{\$y}$ return ⟨res1⟩ { $p_1$, ⟨res2⟩{ $p_2$, $q_2$ }⟨/res2⟩ } ⟨/res1⟩

where query $q_2$ is: for $\$z$ in $p_{\$z}$ where $p_3$ return ⟨res3⟩{ $p_4$ }⟨/res3⟩. Let us first translate $q_2$. Applying the $xq_3$ translation rule from Fig. 3.3, we obtain:

$$full(q_2) = full(p_{\$z}) \bowtie^{n}_{e_{\$z}.ID \prec e_3.ID} full(//p_3) \;⟗^{n}_{e_{\$z}.ID = e_{\$z}.ID} full(p_{\$z}//p_4)$$

Applying the $xq_3$ rule again, twice, for $q$ leads to:

$$(*) \quad full(q) = full(p_{\$x}) \times full(p_{\$y}) \bowtie\!\sqsubset^{n}_{e_{\$x}.ID=e_{\$x}.ID} full(p_1) \\ \bowtie\!\sqsubset^{n}_{e_{\$y}.ID=e_{\$y}.ID} full(p_2) \bowtie\!\sqsubset^{n}_{e_{\$y}.ID=e_{\$y}.ID} full(q_2)$$

Finalizing $q$'s translation, we have $alg(q) = xml_{templ}(full(q))$, where $xml_{templ}$ is:

$$\langle \text{res1} \rangle e_1.C \langle \text{res2} \rangle e_2.C \langle \text{ res3} \rangle e_3.C \langle /\text{res3} \rangle \langle /\text{res2} \rangle \langle /\text{res1} \rangle$$

where $e_1.C, e_2.C, e_3.C$ are the $C$ attributes corresponding to the path expressions $p_1, p_2$ and $p_3$ (those producing returned nodes). Observe that no data restructuring is needed in $xml_{templ}$, since the nested joins in $full(q)$ have grouped the data as the query required.

### 3.3.3 Isolating patterns from algebraic expressions

Algebraic equivalence rules applied on $(*)$ bring $full(q)$ to the equivalent form:

$$\sigma_{(e_{\$z}.ID\neq\perp)\vee(e_{\$z}.ID=\perp\wedge e_2=\perp)}(\, full(p_{\$x}) \bowtie\!\sqsubset^{n}_{e_{\$x}.ID \prec\prec e_c.ID} e_c \;\times \\ full(p_{\$y}) \bowtie\!\sqsubset^{n}_{e_{\$y}.ID \prec\prec e_e.ID} e_e \bowtie\!\sqsubset^{n}_{e_{\$y}.ID \prec\prec e_d.ID} \\ (e_d \bowtie\!\sqsubset^{n}_{e_d.ID \prec\prec e_f.ID}(e_f \bowtie^{n}_{e_f.ID \prec\prec e_g.ID} \sigma_{V=5}(e_g) \bowtie\!\sqsubset^{n}_{e_d.ID \prec\prec e_h.ID} e_h)))$$

This rewriting has grouped together $full(p_{\$x})$ with the other subexpressions structurally related to $\$x$ (the join product before the $\times$). It has also grouped $full(p_{\$y})$ and the subexpressions structurally related to $\$y$ (the last two lines). It turns out that these correspond exactly to the algebraic semantics of patterns $V_{10}$ and $V_{11}$ in Figure 3.1. Thus:

$$alg(q) = xml_{templ}(\sigma_{(e_{\$z}.ID\neq\perp)\vee(e_{\$z}.ID=\perp\wedge e_2=\perp)}(V_{10} \times V_{11}))$$

The $\sigma$ is a by-product of transforming the equality joins in $(*)$ in structural joins.

# Chapter 4

# XAM containment under path summary constraints

In this chapter we study the problem of XAM containment under path summary constraints. First, we provide an alternate semantics for XAMs (Section 4.1) and we introduce the structural summaries (Section 4.2). In Section 4.3 we present the canonical model technique. This method was introduced in [85] for XPath containment without constraints. We extend-it here to XAM containment under path summary constraints. Next, we present our pattern containment approach in layers starting with simple conjunctive patterns and gradually adding value predicates, optional edges, multiple attributes per returned nodes and nesting (Section 4.4). We describe a technique for pattern minimization under path summary constraints in Section 4.5 and we evaluate our containment algorithms in Section 4.6.

## 4.1 Alternative XAM semantics based on embeddings

In this section we provide alternative (yet equivalent) semantics for XAMs based on tree embeddings. This differs from the algebra-based semantics we introduced in Section 2.2. While the latter proves to be the most convenient for extracting tree patterns out of XQuery queries, the embedding-based semantics we describe here is more appropriate when considering XAM containment and rewriting.

For ease of exposition, we define the embedding-based semantics for increasingly larger subsets of the XAM language. These subsets will also be used in the remaining of the chapter to present gradually more complex algorithms.

**Conjunctive tree patterns** The first XAM subset we consider corresponds to the commonly used notion of conjunctive tree patterns [85]. A grammar of this XAM subset can be extracted from the one we presented in Section 2.2 by restricting it to the following productions:

$$NS ::= \top \;\; N^{+} \tag{4.1}$$
$$N ::= \; name \; IDSpec? \tag{4.2}$$
$$IDSpec ::= \; \textbf{ID} \tag{4.3}$$
$$ES ::= \; E* \tag{4.4}$$
$$E ::= \; name_1 \; (/ \mid //) \; \textbf{j} \; name_2 \tag{4.5}$$

Intuitively, a conjunctive tree pattern $p$ is a tree, whose nodes are labeled from members of $\mathcal{L} \cup \{*\}$, and whose edges are labeled / or //. A distinguished subset of $p$ nodes are called *return nodes of* $p$.

Figure 4.1 shows a pattern $p$, whose return nodes are enclosed in boxes.

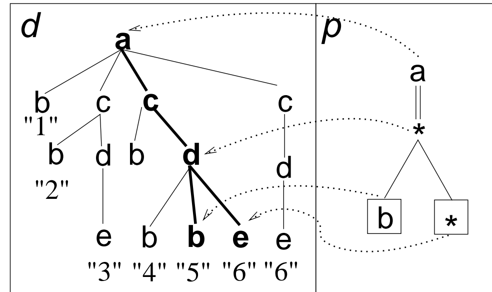



Figure 4.1: Sample document $d$ and a conjunctive pattern $p$.

DEFINITION 4.1.1. An *embedding* of a conjunctive tree pattern $p$ into a document $d$ is a function $e : nodes(p) \rightarrow nodes(d)$ such that:

- For any $n \in nodes(p)$, if $label(n) \neq *$, then $label(e(n)) = label(n)$.
- $e$ maps the root of $p$ into the root of $d$.
- For any $n_1, n_2 \in nodes(p)$ such that $n_2$ is a /-child of $n_1$, $e(n_2)$ is a child of $e(n_1)$.

- For any $n_1, n_2 \in nodes(p)$ such that $n_2$ is a $//$-child of $n_1$, $e(n_2)$ is a descendant of $e(n_1)$.

$\triangleleft$

Dotted arrows in Figure 4.1 illustrate an embedding.

The result of evaluating a conjunctive tree pattern $p$, whose return nodes are $n_1^p, \ldots, n_k^p$, on an XML document $d$, is the set $p(d)$ consisting of all tuples $(n_1^d, \ldots, n_k^d)$ where $n_1^d, \ldots, n_k^d$ are document nodes and there exists an embedding $e$ of $p$ in $d$ such that $e(n_i^p) = n_i^d, i = 1, \ldots, k$.

Given a pattern $p$, a tree $t$ and an embedding $e : p \to t$, we denote by $e(p)$ the tree that consists of the nodes $e(n)$ to which the nodes of $p$ map to, and the edges that connect such nodes. For example, in Figure 4.1, $e(p)$ is shown in bold. We may use the notation $u \in e(p)$ to denote that node $u$ appears in the tree $e(p)$. In Figure 4.1, $e(p)$ has more nodes than $p$, since the intermediary $c$ node also belongs to $e(p)$.

**Decorated tree patterns** A *decorated conjunctive pattern* is a conjunctive pattern where each node $n$ is annotated with a logical formula $\phi_n(v)$, where the free variable $v$ represents the node's value. The formula $\phi_n(v)$ is either $T$ (true), $F$ (false), or an expression composed of atoms of the form $v\,\theta\,c$, where $\theta \in \{=, <, >\}$, $c$ is some $\mathcal{A}$ constant, using $\vee$ and $\wedge$. In Figure 4.2, $p_{\phi 1} - p_{\phi 4}$ are decorated patterns.

We assume $\mathcal{A}$, the domain of atomic values, is totally ordered and enumerable (corresponding to machine-representable atomic values). Then, any $\phi(v)$ can be represented compactly (e.g. by a union of disjoint intervals of $\mathcal{A}$ on which $\phi(v)$ holds), and for any formulas $\phi_1(v), \phi_2(v)$, one can easily compute $\neg\phi_1(v)$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, and $\phi_1(v) \Rightarrow \phi_2(v)$.

We extend our model of labeled trees to *decorated labeled trees*, whereas instead of an $\mathcal{A}$ value, every node $n$ is decorated with a (non-$F$) formula $\phi_n(v)$ as described above. Observe that simple labeled trees are particular cases of decorated ones, where for every $n$, $\phi_n(v)$ is $v = v_n$, where $v_n \in \mathcal{A}$ is $n$'s value.

A *decorated embedding* of a decorated pattern $p_\phi$ into a decorated tree $t_\phi$ is an embedding $e$, such that for any $n \in nodes(p_\phi)$, $\phi_{e(n)}(v) \Rightarrow \phi_n(v)$. Figure 4.2 illustrates a decorated embedding from $p_{\phi 1}$ to $t$. The semantics of a decorated pattern is defined similarly to the simple ones, based on decorated embeddings.

**Optional pattern edges** We extend patterns to allow a distinguished subset of *optional* edges, depicted with dashed lines in patterns $p_1$ and $p_2$ in Figure 4.3. Pattern

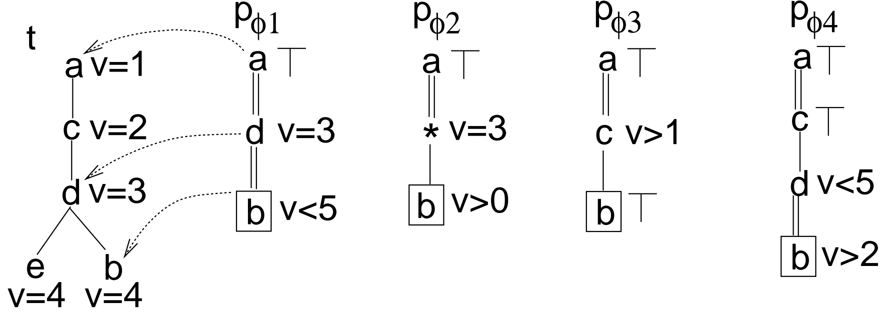



Figure 4.2: Decorated patterns $p_{\phi 1}, p_{\phi 2}, p_{\phi 3}$ and $p_{\phi 4}$.

nodes at the lower end of a dashed edge may lack matches in a data tree, yet matches for the node at the higher end of the optional edge are retained in the pattern's semantics. For example, in Figure 4.3, where $t$ is a data tree (with same-tag nodes numbered to distinguish them), $p_1(t) = \{(c_1, b_2), (c_1, b_3), (c_2, \perp)\}$, where $\perp$ denotes the null constant. Note that $b_2$ lacks a sibling node, yet it appears in $p_1(t)$; and, $c_2$ appears although it has no descendants matching $d$'s subtree.

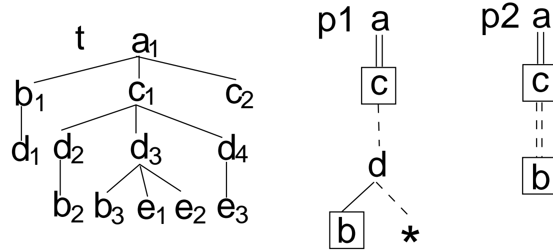



Figure 4.3: Optional patterns example.

Optional pattern embeddings are defined as follows. Let $t$ be a tree and $p$ be a pattern with optional edges. An optional embedding of $p$ in $t$ is a function $e : nodes(p) \rightarrow nodes(t) \cup \{\perp\}$ such that:

1. $e$ maps the root of $p$ into the root of $t$.

2. $\forall\, n \in nodes(p)$, if $e(n) \neq \perp$ and $label(n) \neq *$, then $label(n) = label(e(n))$.

3. $\forall\, n_1, n_2 \in nodes(p)$ such that $n_1$ is the /-parent (respectively, //-parent) of $n_2$:

(a) If the edge $(n_1, n_2)$ is not optional, then $e(n_2)$ is a child (resp. descendant) of $e(n_1)$.

(b) If the edge $(n_1, n_2)$ is optional: $(i)$ If $e(n_1) = \perp$ then $e(n_2) = \perp$. $(ii)$ If $e(n_1) \neq \perp$, let $E'$ be the set of optional embeddings $e'$ from the $p$ subtree rooted at $n_2$, into some $t$ subtree rooted in a child (resp. descendant) of $e(n_1)$. If $E' \neq \emptyset$, then $e(n_2) = e'(n_2)$ for some $e' \in E'$. If $E' = \emptyset$, then $e(n_2) = \perp$.

Conditions 1-3(a) above are those for standard embeddings. Condition 3(b) accounts for the optional pattern edges: we allow $e$ to associate $\perp$ to a node $n_2$ under an optional edge only if no child (or descendant) of $e(n_1)$ could be successfully associated to $n_2$. Based on optional embeddings, optional pattern semantics is defined as for conjunctive patterns.

**Multiple attributes per return node** So far, we have defined pattern semantics as tuples of nodes. A practical view language should allow specifying *what information items does the pattern retain from every return node*. To express this, we define *attribute patterns*, whose nodes may be annotated with up to four attributes:

- *ID* specifies that the pattern contains the node's *identifier*. The identifier is understood as an atomic value, uniquely identifying the node.
- *L* (respectively *V*) specifies that the pattern contains the node's *label* (respectively *value*).
- *C* specifies that the pattern contains the node's *content*, i.e. the subtree rooted at that node, either stored in the view, or as a reference to some repository etc. *Navigation* is possible in a $C$ node attribute, towards the node's descendants.

In Figure 4.4, $p_1$ and $p_2$ are sample attribute patterns.

Attribute pattern semantics is defined as follows. Let $p$ be an attribute pattern, whose return nodes are $(n_1, \ldots, n_k)$, and $t$ be a tree. Let $f_{ID} : nodes(t) \rightarrow \mathcal{A}$ be a labeling function assigning identifiers to $t$ nodes. Then, $p(t, f_{ID})$ is:

$$\{\, tup(n_1, n_1^t) + \ldots + tup(n_k, n_k^t) \mid \exists\, e : p \rightarrow t, e(n_1) = n_1^t, \ldots, e(n_k) = n_k^t \,\}$$

where $+$ stands for tuple concatenation, and $tup(n_i, n_i^t)$ is a tuple having: an attribute $ID_i = f_{ID}(n_i^t)$ if $n_i$ is labeled $ID$; an attribute $L_i = label(n_i^t)$ if $n_i$ is labeled $L$; an

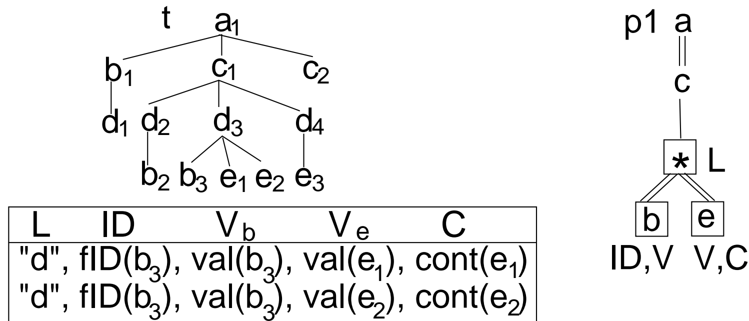



Figure 4.4: Attribute pattern example.

attribute $V_i = value(n_i^t)$ if $n_i$ is labeled $V$; and an attribute $C_i = cont(n_i^t)$ if $n_i$ is labeled $C$. For example, Figure 4.4 depicts $p_1(t, f_{ID})$, for the tree $t$ and some function $f_{ID}$.

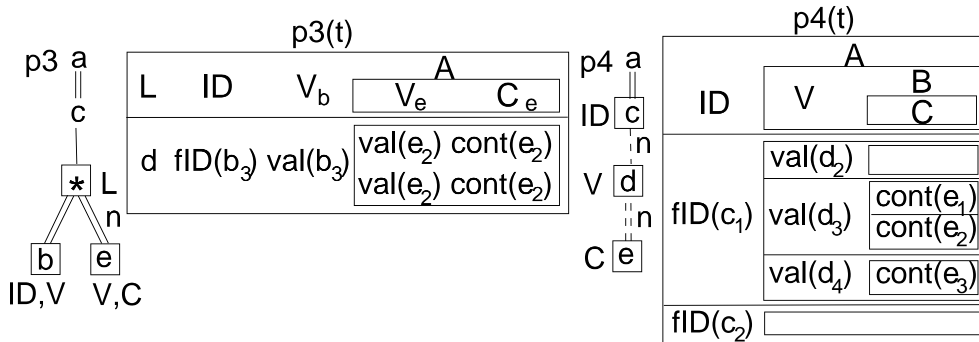



Figure 4.5: Nested patterns and their semantics.

**Nested pattern edges** We extend patterns to distinguish a subset of *nested* edges, marked by an $n$ edge label. For example, pattern $p_3$ in Figure 4.5 is identical to $p_1$ in Figure 4.4 except for the $n$ edge. The semantics of a nested pattern is a nested relation. Let $n_1$ be a pattern node and $n_2$ be a child of $n_1$ connected by a nested edge. Let $n_1^t$ be a data node corresponding to $n_1$ in some data tree. The data extracted from all $n_1^t$ descendants matching $n_2$ appears as a table nested inside the tuple corresponding to $n_1^t$. Figure 4.5 shows $p_3(t)$ for the tree $t$ from Figure 4.4: the attributes $V_e$ and $C_e$ are nested under a single attribute $A$, corresponding to the third return node. Compare this with $p_1(t)$ in Figure 4.4.

## 4.2 Structural summaries

In this section we describe structural XML summaries (also known as path summaries or DataGuides [52]. Our query rewriting approach relies on summaries as a source of structural information concerning the document(s) on which the query is asked. Section 4.2.1 introduces simple structural summaries, while Section 4.2.2 describe summaries enhanced with a form of integrity constraints. Such constraints enable more rewritings.

### 4.2.1 XML path summaries

Path summaries are classical artifacts for semistructured and XML query processing, dating back to 1997 [52]. Path summaries as defined and used in this work correspond to strong DataGuides [52] in the particular case of tree-structured data (recall that DataGuides were originally proposed in the context of graph-structured OEM data [52, 88]). The original DataGuide proposal [52] made a clean distinction between the DataGuide (or summary), which only *describes* the structure of the data and can be thought of as a schema extracted from the data, and the *data store* itself. We preserve this distinction in our work: the dataguides (or path summaries) we use only describe the data instance and do not store data. This is formally described next.

DEFINITION 4.2.1. Path summary. Let $D = (\mathcal{N}, \mathcal{E})$ be an XML document. The *path summary* $S(D)$ is a tree, whose nodes are labeled with element names from the document. The relationship between $D$ and $S(D)$ can be described based on a function $\phi : D \rightarrow S(D)$, recursively defined as follows:

1. $\phi$ maps the root of $D$ into the root of $S(D)$. The two nodes have the same label.

2. Let $child(n, l)$ be the set of all $n$ children labeled $l$, which are also $\mathcal{N}_\rceil$ nodes. If $child(n, l)$ is not empty, then $\phi(n)$ has a unique $l$-labeled child $n_l$ in $S(D)$ and for each $n_i \in child(n, l)$, $\phi(n_i) = n_l$.

3. Let $val(n)$ be the set of text children of an element $n \in D$. Then, $\phi(n)$ has a unique child $n_v$ labeled $\#text$ and furthermore, for each $n_i \in val(n)$, $\phi(n_i) = n_v$.

4. Let $att(n, a)$ be the value of the attribute named $a$ of an element $n \in D$. Then, $\phi(n)$ has a unique child $n_a$ labeled $@a$ and for each $n_i \in att(n, a)$, we have $\phi(n_i) = n_a$.

◁

Definition 4.2.1 deserves some comments. First, observe that $\phi$ preserves node labels and parent-child relationships. Moreover, $\phi$ maps all the nodes from the document $D$ that are on the same *root to node path* to the same summary node.

**Paths and path numbers** Let *rooted path* denote a path of the form $/l_1/l_2/.../l_k$ going from the root to some (leaf or non-leaf) node of the document. In the sequel, for simplicity, *path* is used to refer to a rooted path, unless otherwise specified. For every path in $D$, there is exactly one node reachable by the same path in $S(D)$. Conversely, each node in $S(D)$ corresponds to a path in $D$.

*Notations: paths and summary nodes* Given a summary $S$, the set of $S$ nodes is clearly in bijection with the set of $S$ paths (which is the set of paths in any document conforming to $S$). For ease of explanation, we may refer to a path by its corresponding summary node, or vice-versa.

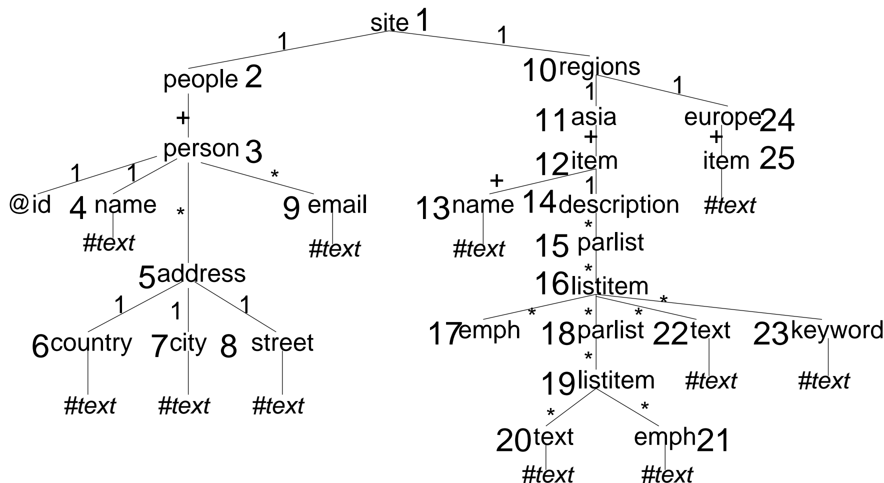



Figure 4.6: The path summary for the XMark document in Figure 1.1

EXAMPLE 4.2.1. Figure 4.6 shows the path summary for the XML fragment from Figure 1.1. We assign an integer number to every summary path; these numbers appear in large font next to the summary nodes, e.g. 1 for /site, 2 for /site/people etc. Thus, there is one number for each summary node and for each path present in the document or summary; we may interchangeably refer to one by means of another. For instance,

we may say that the XML node identified by (2,10) in Figure 1.1 is on path 2 in Figure 4.6(b). ◁

Similar XML documents may have the same summary. This may happen, for instance in the case of documents with identical structure but with diffrent text nodes (think of XML documents describing plane tickets). Observe that text node values are not reflected at all in the summary. Moreover, documents with similar but different structure may also have the same summaries. Consider, for instance, exam attendance lists in XML.

DEFINITION 4.2.2. We say a document $D$ *conforms to a summary* $S_0$, denoted $S_0 \models D$, if $S(D) = S_0$. ◁

Thus, a given summary may be used as a repository of structural information concerning several documents in a database.

**Path summaries and XML storage structures** Path summaries inspired the idea of path indexes [86]: the IDs of all nodes on a given path are clustered together and used as an index. Observe that while the index holds actual data, the summary can be seen as a catalog of this data. The summary can be exploited at query compile time for query optimization purposes. The path index may be accessed at query execution time, if the optimizer deems it useful. As the Chapter 7 will show, subsequent works have used paths as a basis for clustering the storage itself [14, 19, 66, 122] and as a support for statistics [4, 73].

### 4.2.2 Complex summaries

In this thesis we will use a form of path summaries - *enhanced with integrity constraints*. We add some more structural information to summaries, as follows. Let $S$ be a summary.

DEFINITION 4.2.3. Enhanced path summary. An *enhanced summary* $S'$ *based on* $S$ is a tree obtained by copying $S$, and labeling each edge with one symbol among 1, + and *. We say a document $D$ conforms to the enhanced summary $S'$ thus defined if $S \models D$ and furthermore, for any $x$ node in $S'$ and $y$ child of $x$:

- if the edge $x - y$ is labeled 1, then all $D$ nodes on path $x$ have exactly one child on path $y$;

- if the edge $x - y$ is labeled +, then all $D$ nodes on path $x$ have at least one child on path $y$;

$\triangleleft$

In the following, we will term + annotated edges *strong edges* and 1-annotated edges *one-to-one edges*. Figure 4.6 presents an example of a summary whose strong edges are shown in thick lines. We do not introduce a graphical primitive to represent one-to-one edges.

## 4.3 Canonical models for patterns

In this section we introduce the canonical models which are key ingredients for deciding pattern containment. Section 4.3.1 considers the restrictive case of conjunctive tree patterns whereas Section 4.3.2 extends the definition to more complex patterns.

### 4.3.1 Canonical model of a conjunctive pattern

Let $p$ be a conjunctive tree pattern, and $S$ be a summary. Let $e : p \to S$ be an embedding of $p$ in $S$. The *canonical tree derived from* $e$, denoted $t_e$, is obtained as follows:

- For each $n \in p$, $t_e$ contains a distinguished node whose label is that of $e(n)$. When $n$ is a returning node in $p$, we say $e(n)$ is a returning node in $t_e$.
- Let $n \in p$ be a node and $m_1, m_2, \ldots, m_k$ its children. Then, the $t_e$ node corresponding to $e(n)$ has exactly $k$ children, and for $1 \leq i \leq k$, its $i$-th child consists of a parent-child chain of nodes, whose labels are those connecting $e(n)$ to $e(m_i)$ in $S$.

For instance, in Figure 4.7, an embedding $e_1 : p \to S$ maps the upper $*$ in $p$ to the $S$ node numbered 3, and the lower returning $*$ node in $p$ to the $S$ node numbered 5. The tree $t_1$ in Figure 4.7 is the canonical tree derived from $e_1$. Similarly, another embedding $e_2 : p \to S$ associates the upper $*$ node in $p$ to the $S$ node numbered 5, and the lower $*$ node to the $S$ tree numbered 7. The tree $t_2$ in Figure 4.7 is the canonical tree derived from $e_2$. Note that an embedding needs not to be an injective function. The return nodes of $p$ can be embedded into the same $b$ node in $S$, yielding the canonical trees $t_3$ and $t_4$.

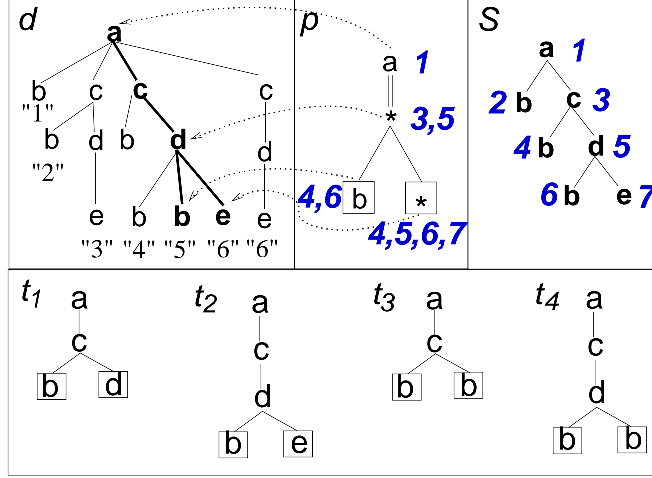



Figure 4.7: Sample document $d$,a conjunctive pattern $p$, the path summary $S$ of $d$ and the canonical model of $p$.

Let the return nodes in $p$ be $n_1^p, \ldots, n_k^p$. Then for every tree $t_e \in mod_S(p)$ corresponding to an embedding $e$, the tuple $(e(n_1^p), \ldots, e(n_k^p))$ is called *the return tuple of* $t_e$. Note that two different trees $t_1, t_2 \in mod_S(p)$ may have the same return tuples.

The $S$-*canonical model of* $p$, denoted $mod_S(p)$, is the set of the canonical trees obtained from all possible embeddings of $p$ in $S$. Clearly, for any canonical tree $t_e$, $S \models t_e$.

Observe that two distinct embeddings may yield the same canonical tree. For instance, let $p'$ be the pattern $/a//*//e$ where $b$ is the returning node, and consider the following two embeddings of $p$ in the summary $S$ in Figure 4.7:

- $e'_1$ maps the $*$ node of $p'$ to the $S$ node numbered 3;
- $e'_2$ maps the $*$ node of $p'$ to the $S$ node numbered 5.

The canonical trees derived from $e'_1$ and $e'_2$ coincide. When defining $S$, we consider it duplicate-free.

In Figure 4.7, for the represented pattern $p$ and summary $S$, we have $mod_S(p) = \{t_1, t_2, t_3, t_4\}$.

In the following, we use the term *subtree* in the following sense. We say a tree $t'$ is a

subtree of the tree $t$ if ($i$) $t'$ and $t$ have the same root, ($ii$) the nodes of $t'$ are a subset of the nodes of $t$ and ($iii$) the edges of $t'$ are a subset of the edges of $t$.

PROPOSITION 4.3.1. Let $t$ be a tree and $S$ be a summary such that $S \models t$, $p$ be a $k$-ary conjunctive pattern, and $\{n_1^t, \ldots, n_k^t\} \subseteq nodes(t)$.
$(n_1^t, \ldots, n_k^t) \in p(t) \Leftrightarrow \exists\ t_e \in mod_S(p)$ such that:

1. $t$ has a subtree isomorphic to $t_e$. For simplicity, we shall simply say $t_e$ is a subtree of $t$.
2. For every $0 \leq i \leq k$, node $n_i^t$ is on path $n_i^S$, where $n_i^S$ is the $i$-th return node of $t_e$.

$\triangleleft$

*Proof:*
$\Leftarrow$: Let $e : p \rightarrow S$ be one of the embeddings associated to $t_e$ (recall that several such embeddings may exist). We define $e' : p \rightarrow t$ as follows: for every $n \in p$, $e'(n) = e(n)$, which is safe since $e(p) \subseteq nodes(t_e) \subseteq nodes(t)$. Clearly, $e'$ is an embedding, and $e(n_i^p) = n_i^t$ for every $0 \leq i \leq k$, thus $(n_1^t, \ldots, n_k^t) \in p(t)$.

$\Rightarrow$: By definition, if $(n_1^t, \ldots, n_k^t) \in p(t)$, there exists an embedding $e : p \rightarrow t$, such that $e(n_i^p) = n_i^t$ for every $0 \leq i \leq k$. We denote by $e_S : p \rightarrow S$ the embedding obtained from $e$, by setting $e_S(n)$ to be the path of $e(n)$ for each node $n$ of $p$. Let $t_e$ be the $mod_S(p)$ tree corresponding to $e_S$. We show that $t_e$ is a subtree of $t$.

Let $n$ be a $t_e$ node such that $n = e_S(n_p)$ for some $n_p \in p$. Then, $n$ is the path of $e(n_p)$, and since $e$ is an embedding of $p$ in $t$, then $n$ belongs to $t$. Thus, all the images of $p$ nodes through $e_S$ belong to $t$.

Now consider a $t_e$ node $n$, and let us prove that its children also belong to $t$. Let $m$ be a direct child of $n$. Then, by definition of $t_e$, $m$ participates in a chain of nodes connecting $e_S(n_p)$ to $e_S(m_p)$, for some $m_p$ child of $n_p$ in $p$. By definition of $e_S$, $e_S(m_p)$ is the path of $e(m_p) \in t$, thus the chain of nodes between $e(n_p)$ and $e(m_p)$ belongs to $t$, thus all edges and nodes between these two nodes (including $m$) belong to $t$. Thus, $t_e$ is a subtree of $t$.

To see that for each $i$, $n_i^d$ is on path $n_i^e$, observe that $n_i^d$ is $e(n_i^p)$ for some returning node $n_i^p$ of $p$, and furthermore $e_S(n_i^p)$ is the path of $n_i^d$ and is also $n_i^e$.

For example, in Figure 4.7, bold lines and node names trace a $d$ subtree isomorphic to $t_2 \in mod_S(p)$ (recall $t_2$ from Figure 4.7). For the sample document and pattern, the

thick-lined subtree is the one Proposition 4.3.1 requires in order for the boxed nodes in $d$ to belong to $p(d)$.

A pattern $p$ is said $S$-*unsatisfiable* if for any document $d$ such that $S \models d$, $p(d) = \emptyset$. The above proposition provides a convenient means to test satisfiability: $p$ is $S$-satisfiable iff $mod_S(p) \neq \emptyset$.

DEFINITION 4.3.1. Let $S$ be a summary, $p$ be a pattern, and $n$ a node in $p$. The set of *paths associated to* $n$ (or the *path annotation* of $n$) consists of those $S$ nodes $s_n$, such that for some embedding $e : p \to S$, $e(n) = s_n$. ◁

At right in Figure 4.7, the pattern $p$ is repeated, showing next to each node (in italic font) the paths associated to that node.

**Maximum size of the canonical model** In general $|mod_S(p)| \leq |S| \times |p|$. The Figure 4.8 illustrates the worst case in which $|mod_S(p)| = |S| \times |p|$. Indeed for every return node from the pattern $p$ the canonical model contains a chain of length $|S|$ that equals the dataguide size. However in practice, as we will see in Section 4.6 the canonical model is very small.

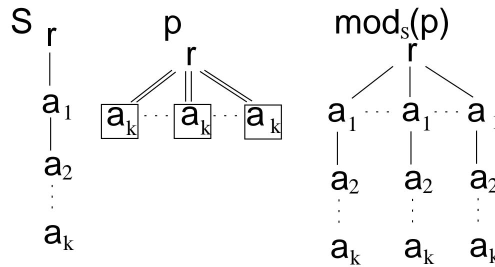



Figure 4.8: Maximum size of the canonical model

### 4.3.2 Canonical model of complex patterns

**Canonical model of a decorated pattern** Given a summary $S$, the $S$ canonical model $mod_S(p_\phi)$ of a decorated pattern $p_\phi$, is obtained from $mod_S(p)$ (where $p$ is the pattern obtained by erasing $p_\phi$'s formulas) by decorating, in every tree $t_e \in mod_S(p)$ corresponding to an embedding $e$: ($i$) each node $s = e(n)$, for some $n \in nodes(p)$,

with the formula $\phi_n(v)$ from $p_\phi$, $(ii)$ all other nodes with $T$. For example, in Figure 4.9, $mod_S(p_{\phi 1}) = \{t_{\phi 1}\}$, $mod_S(p_{\phi 2}) = \{t'_{\phi 2}, t''_{\phi 2}\}$, $mod_S(p_{\phi 3}) = \{t_{\phi 3}\}$ and $mod_S(p_{\phi 4}) = \{t_{\phi 4}\}$.

Note that two pattern nodes $n_x, n_y$, decorated with different (or even contradictory) formulas $\phi_x(v)$, $\phi_y(v)$ may be mapped by an embedding $e$ to the same summary node $s$. In this case, the canonical model tree corresponding to $e$ must contain different nodes for $e(n_x)$ and $e(n_y)$, each labeled with its respective formula. Thus, canonical model trees in the presence of predicates are no longer strictly speaking $S$ subtrees. However, their size remains moderate, since the $p$ subtrees corresponding to $n_x$, respectively, $n_y$ will each be reflected once in the canonical tree, under $e(n_x)$, respectively, $e(n_y)$. For simplicity, we will continue to assume here that canonical model trees are $S$ subtrees.

Let $t_\phi$ be a decorated tree, $p_\phi$ a $k$-ary decorated pattern and $S$ a summary. A characterization of the tuples in $p_\phi(t_\phi)$ derives directly from Proposition 4.3.1, considering decorated patterns and trees.

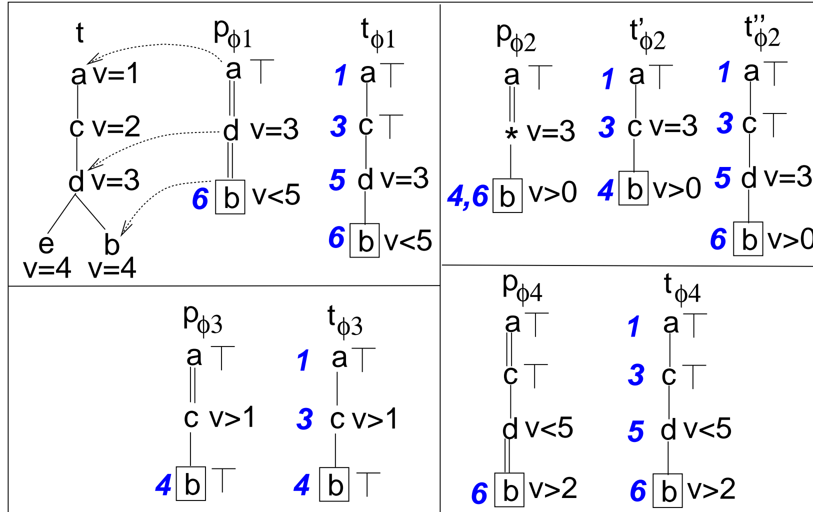



Figure 4.9: Decorated patterns $p_{\phi 1}, p_{\phi 2}, p_{\phi 3}$ and $p_{\phi 4}$, their canonical models, and a decorated embedding.

**Canonical model of a pattern with optional edges** Given a summary $S$ and an optional pattern $p$, $mod_S(p)$ is obtained as follows:

- Let $E$ be the set of optional $p$ edges. Let $p_0$ be the strict pattern obtained from $p$ by making all edges non-optional.

- For every $t_e \in mod_S(p_0)$ and set of edges $F \subseteq E$, let $t_{e,F}$ be the tree obtained from $t_e$ by erasing all subtrees rooted in a node at the lower end of a $F$ edge. If $p(t_{e,F}) \neq \emptyset$, add $t_{e,F}$ to $mod_S(p)$.

For example, in Figure 4.10, let $p_0$ be the strict pattern corresponding to $p_1$ (not shown in the figure), then $mod_S(p_0) = \{t_1\}$. Applying the definition above, we obtain: $t_1$ when $F$; $t_2$ when $F$ contains the edge under the $d$ node; $t_3$ when $F$ contains the edge under the $c$ node, or when $F$ contains both optional edges. Thus, $mod_S(p_1) = \{t_1, t_2, t_3\}$.

As described above, the canonical model of an optional pattern may be exponentially larger than the simple one. In practice, however, this is not the case, as Section 4.6 shows.

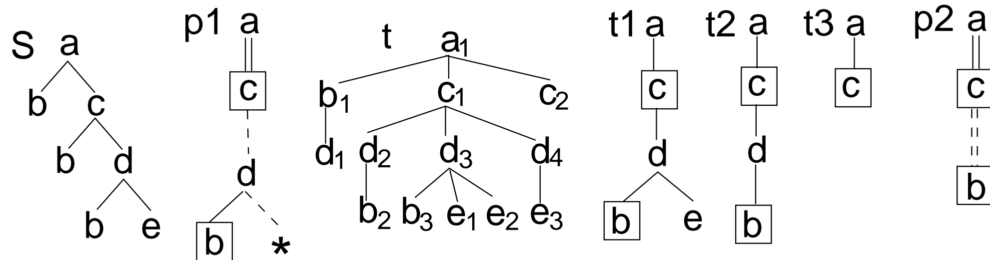



Figure 4.10: Optional patterns example.

**Canonical model of an attribute pattern** The $S$-canonical model of an attribute pattern is defined just like for regular ones. Attribute pattern containment is characterized by the same conditions as for simple patterns, *and* requires that corresponding return nodes be annotated with exactly the same attributes. In Figure 4.11, $p_1 \subseteq_S p_2$.

**Canonical model for nested patterns** The $S$-canonical model of an nested pattern is defined just like for regular ones.

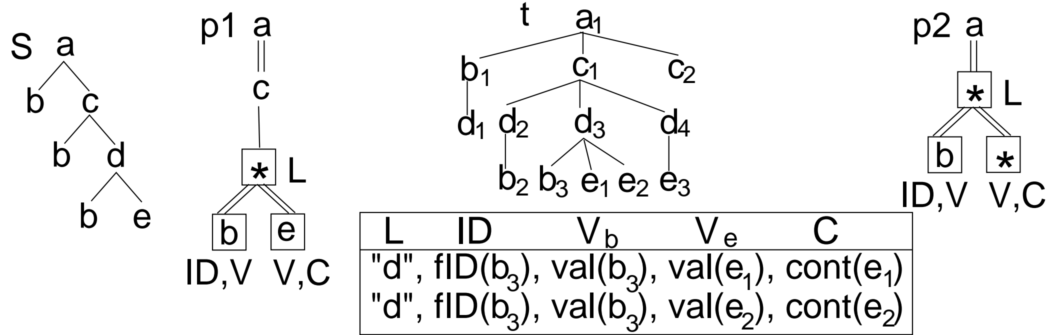



Figure 4.11: Attribute pattern example.

## 4.4 Pattern containment

We start by defining pattern containment under summary constraints:

DEFINITION 4.4.1. Let $p, p'$ be two tree patterns, and $S$ be a summary. We say $p$ *is $S$-contained in* $p'$, denoted $p \subseteq_S p'$, *iff* for any $t$ such that $S \models t$, $p(t) \subseteq p'(t)$. $\triangleleft$

We define *$S$-equivalence* as two-way containment, and denote it $\equiv_S$. When $S$ is known, we simply call it equivalence.

### 4.4.1 Conjunctive patterns

A practical method for deciding containment is stated in the following proposition:

PROPOSITION 4.4.1. Let $p, p'$ be two conjunctive $k$-ary tree patterns and $S$ a summary. The following are equivalent:

1. $p \subseteq_S p'$
2. $\forall\ t_p \in mod_S(p)\ \exists\ t_{p'} \in mod_S(p')$ such that $(i)$ $t_{p'}$ is a subtree of $t_p$ and $(ii)$ $t_p, t_{p'}$ have the same return nodes.
3. $\forall\ t_p \in mod_S(p)$ whose return nodes are $(n_1^t, \ldots, n_k^t)$, we have $(n_1^t, \ldots, n_k^t) \in p'(t_p)$.

$\triangleleft$

*Proof:*
In order to prove the equivalences, note that by definition, $p \subseteq_S p'$ is equivalent to: $\forall\, t$ such that $S \models t$, and nodes $n_1^t, \ldots, n_k^t$ of $t$:

$$(n_1^t, \ldots, n_k^t) \in p(t) \Rightarrow (n_1^t, \ldots, n_k^t) \in p'(t).$$

For any such $t$ and $n_1^t, \ldots, n_k^t$, let $n_1^S, \ldots, n_k^S$ be the $S$ nodes corresponding to the paths of $n_1^t, \ldots$, respectively $n_k^t$ in $t$. Then, $p \subseteq_S p'$ is equivalent to:

$$(*)\ \ \forall\, t \text{ such that } S \models t,\ \{n_1^t, \ldots, n_k^t\} \text{ nodes of } t,\ S_1 \Rightarrow S_2$$

where $S_1$ is:

$\exists\ t_e \in mod_S(p)$ such that $t_e$ is a subtree of $t$ and $(n_1^S, \ldots, n_k^S)$ are the return nodes of $t_e$

and $S_2$ is:

$\exists\ t_{e'} \in mod_S(p')$ such that $t_{e'}$ is a subtree of $t$ and $(n_1^S, \ldots, n_k^S)$ are the return nodes of $t_e'$

(1) $\Rightarrow$ (2): if $p \subseteq_S p'$, let the role of $t$ in $(*)$ be successively played by all $t_e \in mod_S(p)$ (clearly, $S \models t_e$). Each such $t_e$ naturally contains a subtree (namely, itself) satisfying $S_1$ above, and since $S_1 \Rightarrow S_2$, $t_e$ must also contain a subtree $t_{e'} \in mod_S(p')$ with the same return nodes as $t_e$.

(2) $\Rightarrow$ (1): let $t$ be a tree and $(n_1^t, \ldots, n_k^t) \in p(t)$. By Proposition 4.3.1, $t$ contains a subtree $t_e \in mod_S(p)$, such that the return nodes of $t$ are those of $t_e$, namely $(n_1^t, \ldots, n_k^t)$. By (2), $t_e$ contains a subtree $t_{e'} \in mod_S(p')$ with the same return nodes, and $t_{e'}$ is a subtree of $t$, thus (again by Proposition 4.3.1) $(n_1^t, \ldots, n_k^t) \in p'(t)$.

(2) $\Leftrightarrow$ (3) follows directly from Proposition 4.3.1.

Proposition 4.4.1 gives an algorithm for testing $p \subseteq_S p'$: compute $mod_S(p)$, then test that $(n_1^S, \ldots, n_k^S) \in p'(t_e)$ for every $t_e \in mod_S(p)$, where $(n_1^S, \ldots, n_k^S)$ are the return nodes of $p$. The complexity of this algorithm is $O(|mod_S(p)| \times |S| \times |p| \times |p'|)$, since each $mod_S(p)$ tree has at most $|S| \times |p|$ nodes., and $p'(t_e)$ can be computed in $|t_e| \times |p'|$ [54]. In the worst case, $|mod_S(p)|$ is $|S|^{|p|}$. This occurs when any $p$ node

matches any $S$ node, e.g. if all $p$ nodes are labeled $*$, and $p$ consists of only the root and $//$ children. For practical queries, however, $|mod_S(p)|$ is much smaller, as Section 4.6 shows.

A simple extension of Proposition 4.4.1 addresses containment for unions of patterns:

PROPOSITION 4.4.2. Let $p, p'_1, \ldots, p'_m$ be $k$-ary conjunctive patterns and $S$ be a summary. Then, $p \subseteq_S (p'_1 \cup \ldots \cup p'_m) \Leftrightarrow$ for every $t_e \in mod_S(p)$ such that $(n_1, \ldots, n_k)$ are the return nodes of $t_e$, there exists some $1 \leq i \leq m$ such that $(n_1, \ldots, n_k) \in p'_i(t_e)$. $\triangleleft$

### 4.4.2 Decorated patterns

A characterization of $S$-containment among decorated patterns can be similarly obtained from Proposition 4.4.1. Considering two decorated patterns $p_\phi, p'_\phi$ and a summary $S$, condition 3 from Proposition 4.4.1 is replaced by: $\forall t_{p_\phi} \in mod_S(p_\phi)$ such that the return nodes of $t_{p_\phi}$ are $(n_1, \ldots, n_k)$, we have $(n_1, \ldots, n_k) \in p'_\phi(t_{p_\phi})$. For example, in Figure 4.9, $p_{\phi 1} \subseteq_S p_{\phi 2}$. Characterizing the situations where $p_\phi \subseteq_S p_{\phi 1} \cup \ldots \cup p_{\phi n}$ requires some auxiliary notations. Let $t_e$ be a tree from the $S$-canonical model of some decorated pattern. We denote the nodes of $t_e$ (which are also $S$ nodes) by $s_{i_1}, \ldots, s_{i_m}$, where $1 \leq i_1, \ldots, i_m \leq |S|$, and $m$ is the size of $t_e$. For instance ,the nodes of $t_{\phi 1}$ in Figure 4.9 can be identified by $s_1, s_3, s_5$ and $s_6$, given the $S$ node numbers in Figure 4.7. We denote by $\phi_{t_e}(v_1, \ldots, v_{|S|})$ the conjunction of the formulas attached to all $t_e$ nodes, with the convention that each $v_{i_j}$ is the variable corresponding to the node $s_{i_j}$:

$$\phi_{t_e}(v_1, \ldots, v_{|S|}) = \phi_{s_{i_1}}(v_{i_1}) \wedge \ldots \wedge \phi_{s_{i_m}}(v_{i_m}) \mid nodes(t_e) = \{s_{i_1}, \ldots, s_{i_m}\}$$

For instance, $\phi_{t_{\phi 1}}(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ in Figure 4.9 is: $(v_3 = 3) \wedge (v_6 < 5)$.

We have $p_\phi \subseteq_S p_{\phi 1} \cup \ldots \cup p_{\phi n}$ $iff$:

1. For every $t_e \in mod_S(p_\phi)$ such that $(n_1, \ldots, n_k)$ are the return nodes of $t_e$, there exists some $i$, $1 \leq i \leq n$, such that $(n_1, \ldots, n_k) \in p_{\phi i}(t_e)$.

2. For every $t_e \in mod_S(p_\phi)$, let $f(t_e)$ be the set of patterns $p_{\phi i}$ which make condition 1. true. Let $g(t_e)$ be the set of trees from $mod_S(p)$, with $p \in f(t_e)$, having the same return nodes as $t_e$. Then:

$$\phi_{t_e}(v_1, \ldots, v_{|S|}) \Rightarrow$$
$$\bigvee_{t'_e \in g(t_e)} (\phi_{t'_e}(v_1, \ldots, v_{|S|}))$$

Intuitively, condition 2 ensures that the value conditions attached to the nodes of $p_\phi$ are stricter than the disjunction of the $p_{\phi 1}, \ldots, p_{\phi m}$ conditions. The complexity of condition 2 is $N^{|S|}$, where $N$ is the number of constants used in value comparison. In practice, we expect $N$ to be small. Moreover, the formulas typically carry over much less than $S$ variables (as in the above example). Restricting the value predicates to equalities (drastically) reduces the complexity.

We illustrate this criteria by deciding whether $p_{\phi_2} \subseteq_S p_{\phi 1} \cup p_{\phi 3} \cup p_{\phi 4}$, for the patterns in Figure 4.9. We have $mod_S(p_2) = \{t'_{\phi 2}, t''_{\phi 2}\}$. We obtain (omitting the variables $(v_1, \ldots, v_7)$ from all formulas for brevity):

- $\phi_{t'_{\phi 2}}$ is $(v_3 = 3) \wedge (v_4 > 0)$, $f(t'_{\phi 2}) = \{p_{\phi 3}\}$, $g(t'_{\phi 2}) = \{t_{\phi 3}\}$, and $\phi_{t_{\phi 3}}$ is $(v_3 > 1)$. Thus, $\phi_{t'_{\phi 2}} \Rightarrow \phi_{t_{\phi 3}}$.
- $\phi_{t''_{\phi 2}}$ is $(v_5 = 3) \wedge (v_6 > 0)$, $f(t''_{\phi 2}) = \{p_{\phi 1}, p_{\phi 4}\}$, $g(t''_{\phi 2}) = \{t_{\phi 1}, t_{\phi 4}\}$, $\phi_{t_{\phi 1}}$ is $(v_5 = 3) \wedge (v_6 < 5)$, and $\phi_{t_{\phi 4}}$ is $(v_5 < 5) \wedge (v_6 > 2)$. Thus, $\phi_{t''_{\phi 2}} \Rightarrow \phi_{t_{\phi 1}} \vee \phi_{t_{\phi 4}}$.

Thus, we conclude $p_{\phi 2} \subseteq_S p_{\phi 1} \cup p_{\phi 3} \cup p_{\phi 4}$.

### 4.4.3 Optional pattern edges

Containment for (unions of) optional patterns is determined based on canonical models similarly as for the conjunctive patterns. For example, in Figure 4.10, we have $p_1 \subseteq_S p_2$.

### 4.4.4 Attribute patterns

Attribute pattern containment is characterized as follows:

PROPOSITION 4.4.3. Let $p_{1,a}, p_{2,a}$ be two attribute patterns, whose return nodes are $(n^1_1, \ldots, n^1_k)$, respectively $(n^2_1, \ldots, n^2_k)$, and $S$ be a summary. We have $p_{1,a} \subseteq_S p_{2,a}$ iff:

1. For every $i$, $1 \leq i \leq k$, node $n^1_i$ is labeled $ID$ (respectively, $V$, $L$, $C$) iff node $n^2_i$ is labeled $ID$ (respectively, $V$, $L$, $C$).
2. Let $p_2$ be the simple pattern obtained from $p_{2,a}$. For every $t_e \in mod_S(p_{1,a})$, whose return nodes are $(n^t_1, \ldots, n^t_k)$, we have $(n^t_1, \ldots, n^t_k) \in p_2(t_e)$.

◁

In Figure 4.11, $p_1 \subseteq_S p_2$. Containment of unions of attribute patterns may be characterized by extending Proposition 4.4.2 with a condition similar to 1 above.

### 4.4.5 Nested patterns

Let $p_{n,1}, p_{n,2}$ be two nested patterns whose return nodes are $(n_1^1, \ldots, n_k^1)$, respectively, $(n_1^2, \ldots, n_k^2)$, and $S$ be a summary. For each $n_i^1$ and embedding $e : p_{n,1} \to S$, the *nesting sequence* of $n_i^1$ and $e$, denoted $ns(n_i^1, e)$, is the sequence of $S$ nodes $p'$ such that: $(i)$ for some $n'$ ancestor of $n_i^1$, $e(n') = p'$; $(ii)$ the edge going down from $n'$ towards $n_i^1$ is nested. Clearly, the length of the nesting sequence $ns(n_i^1, e)$ for any $e$ is the number of $n$ edges above $n_i^1$ in $p_{n,1}$, and we denote it $|ns(n_i^1)|$. For every $n_i^2$ and $e' : p_{n,2} \to S$, the nesting sequence $ns(n_i^2, e')$ is similarly defined.

PROPOSITION 4.4.4. Let $p_{n,1}, p_{n,2}$ be two nested patterns and $S$ a summary as above. $p_{n,1} \subseteq_S p_{n,2}$ iff:

1. Let $p_1$ and $p_2$ be the unnested patterns obtained from $p_{n,1}$ and $p_{n,2}$. Then, $p_1 \subseteq_S p_2$.
2. For every $1 \leq i \leq k$, the following conditions hold:
   (a) $|ns(n_i^1)| = |ns(n_i^2)|$.
   (b) for every embedding $e : p_{n,1} \to S$, there exists an embedding $e' : p_{n,2} \to S$ with the same return nodes as $e$, such that $ns(n_i^1, e) = ns(n_i^2, e')$.

◁

Intuitively, condition 1 ensures that the tuples in $p_1$ are also in $p_2$, abstraction being made from their nesting. Condition 2(a) requires the same nested signature for $p_1$ and $p_2$, while 2(b) imposes that nesting be applied "under the same nodes" in both patterns.

Condition 2(b) can be safely relaxed, in the presence of another class of integrity constraints. Assume a distinguished subset of $S$ edges are *one-to-one* (Section 4.2.2). Then, nesting data under an $s_1$ node has the same effect as nesting it under its $s_2$ child. Taking into account such information, the equality in condition 2(b) is replaced by: $ns(n_i^1, e)$ and $ns(n_i^2, e')$ are connected by one-to-one edges only.

Nested edges combine naturally with the other pattern extensions we presented. For example, Figure 4.5 shows the pattern $p_4$ with two nested, optional edges, and $p_4(t)$ for the tree $t$ in Figure 4.10. Note the empty tables resulting from the combination of missing attributes and nested edges.

## 4.5 Pattern minimization under summary constraints

We conclude our discussion of XAM containment with an incursion in a related topic, namely tree pattern minimization. While this incursion is brief, it enables an interesting comparison (Section 6) of tree pattern minimization under different types of constraints.

Let $t$ be a pattern and $t'$ be a pattern obtained from $t$ by erasing one $t$ node and re-connecting the remaining nodes among themselves. If $t' \equiv_S t$, we say $t'$ is an $S$-*contraction* of $t$[1]. A pattern $t$ is said to be *minimal under $S$-contraction* if and only if no pattern $t'$ obtained from $t$ via $S$-contraction satisfies $t \equiv_S t'$.

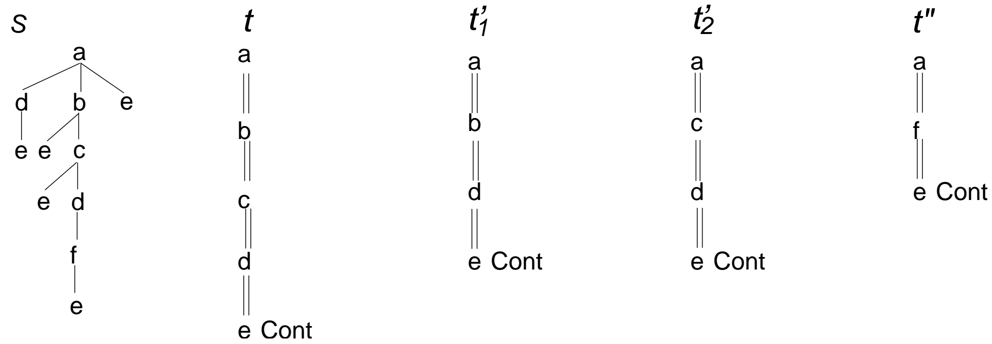



Figure 4.12: Sample summary, a pattern $t$ and some smaller, equivalent patterns.

The process of *minimizing a pattern t by $S$-contraction* consists of finding all the patterns $t'$, minimal under $S$-contraction, which can be derived from $t$. Observe that several such patterns $t'$ may exist. For instance, in Figure 4.12, the patterns $t'_1$ and $t'_2$ are the result of minimizing the pattern $t$ by $S$-contraction. No pattern obtained from $t'_1$ or $t'_2$ by $S$-contraction is still equivalent to $t$. (Notice that a pattern of the form $//a//d//e$, obtained by erasing one node from $t'_1$ or $t'_2$, is no longer equivalent to $t$, since it also returns $e$ nodes on the path $/a/d/e$, which do not belong to $t$).

[1]Observe that $t' \equiv_S t$ requires that no return node of $t$ has been erased.

It turns out that $S$-contraction does not always yield the smallest possible patterns $S$-equivalent to $t$ (where pattern $p_1$ is smaller than $p_2$ if $p_1$ has fewer nodes than $p_2$). For instance, in Figure 4.12, the pattern $t''$ is smaller than both $t'_1$ and $t'_2$, yet $t \equiv_S t''$. The intuitive reason is that the summary brings in *more* nodes than are available in the original pattern. The process of *minimizing a pattern $t$ under $S$ constraints* consists of finding all patterns $t'$ that are $S$-equivalent to $t$, and such that no pattern $t''$ smaller than $t'$ is still $S$-equivalent to $t$. This may yield smaller patterns than those found by $S$-contraction only.

In general, there may be more than one such smallest equivalent pattern. For instance, if we modify the summary in Figure 4.12 to add a $g$ node between the $f$ node and its child $e$, the pattern $t'''$ corresponding to the query $//a//g//e$ is also $S$-equivalent to $t$, smaller than $t'_1$ and $t'_2$, and $t'''$ has the same size as $t''$.

No pattern smaller than $t''$ and still $S$-equivalent to $t$ can be found.

## 4.6 Experimental evaluation of XAM containment

The XAM containment algorithm presented in this work has been implemented in the ULoad prototype [13]. We do not compare the performance of our algorithm with that of existing rewriting algorithms [18, 79, 38, 120] as the type of structural constraints that we are using and which determines the space of rewritings is significantly different (see Section 6). Instead, in this section we present the containment times for standard data sets (DBLP and XMark) using the XMark benchmark queries as well as randomly generated query patterns.

We report on measures performed on a laptop with an Intel 2 GHz CPU and 1 GB RAM, running Linux Gentoo, and using JDK 1.5.0. We denote by XMark$n$ an XMark [115] document of $n$ MB.

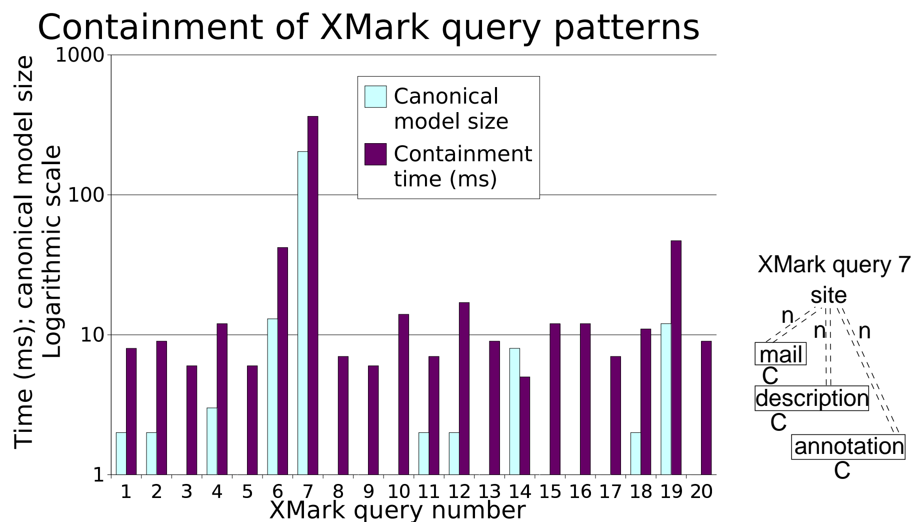
| *Doc* | *Shakespeare* | *Nasa* | *SwissProt* | *XMark11* | *XMark111* | *XMark233* | *DBLP02* | *DBLP05* |
|---|---|---|---|---|---|---|---|---|
| *Size* | 7.5MB | 24MB | 109MB | 11MB | 111MB | 233MB | 133 | 280MB |
| *N* | 179690 | 476645 | 2977030 | 206130 | 1666310 | 4103208 | 3736406 | 7123198 |
| *\|S\|* | 58 | 111 | 264 | 536 | 548 | 548 | 145 | 159 |
| *Ns(N1)* | 40 (23) | 80 (64) | 167(145) | 188(153) | 188(153) | 188(153) | 43(34) | 47 (39) |

Figure 4.13: Sample XML documents and their summaries..

To start with, we gather some statistics on summaries of several documents[2]. In Table 4.13,$N$ is the total number of elements in the original XML document, $n_s$ is the number of strong edges, and $n_1$ the number of one-to-one edges; such edges are quite frequent, thus many integrity constraints can be exploited by rewriting. Figure 4.13 demonstrates that summaries are quite small, and change little as the document grows: from XMark11 to XMark232, the summary only grows by 10%, and similarly for the DBLP data. Intuitively, the complexity of a data set levels off at some point. Thus, while summaries may have to be updated (in linear time [52]), the updates are likely to be modest.

To test containment, we first extracted the patterns of the 20 XMark [115] queries, and tested the containment of each pattern in itself under the constraints of the largest XMark summary (548 nodes). Figure 4.14 (top) shows the canonical model size, and containment time.

[2]All documents, patterns and summaries used in this section are available at [110].



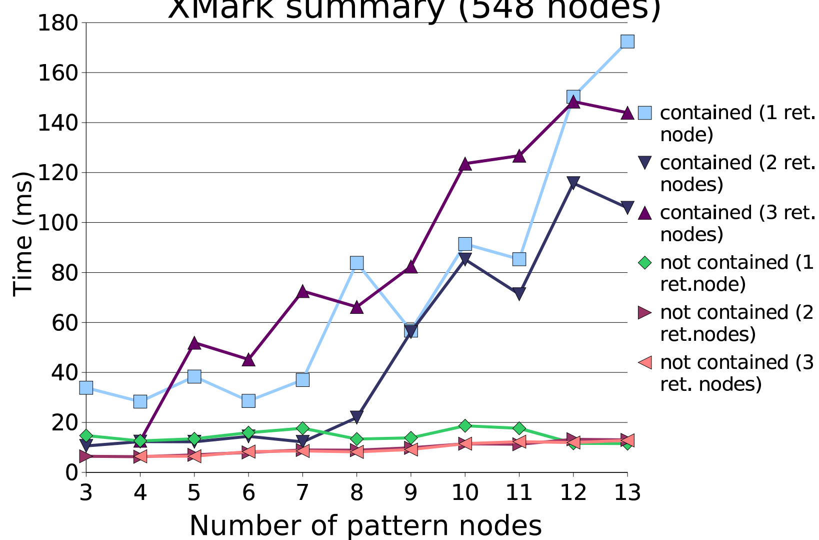




Figure 4.14: XMark pattern containment.

Note that $|mod_S(p)|$ is small, much less than the theoretical bound of $|S|^{|p|}$. The $S$-model of query 7 (shown at top right in Figure 4.14) has 204 trees, due to the lack of structural relationships between the query variables. Such queries are not likely to be frequent in practice.

We also generated synthetic, satisfiable patterns of $3 - 13$ nodes, based on the 548-nodes XMark summary. Pattern node fanout is $f = 3$. Nodes were labeled $*$ with probability $0.1$, and with a value predicate of the form $v = c$ with probability $0.2$. We used 10 different values. Edges are labeled $//$ with probability $0.5$, and are optional with probability $0.5$. For this measure, we turned off edge nesting, since randomly generated patterns with nested edges easily disagree on their nesting sequences, thus containment fails, and nesting does not significantly change the complexity (Section 4.4.5).

For each $n$, we generated 3 sets of 40 patterns, having $r$=1, 2, resp. 3 return nodes; we fixed the labels of the return nodes to $item$, $name$, and $initial$, to avoid patterns returning unrelated nodes. For every $n$, every $r$, and every $i = 1, \ldots, 40$, we tested $p_{n,i,r} \subseteq_S p_{n,j,r}$ with $j = i, \ldots, 40$, and averaged the containment time over 780 executions. Figure 4.14 shows the result, separating positive from negative cases. The latter are faster because the algorithm exits as soon as one canonical model tree contradicts the containment condition, thus $mod_S(p)$ needs not be fully built. Successful test time grows with $n$, but remains moderate. The curves are quite irregular, since $|mod_S(p)|$ varies a lot among patterns, and is difficult to control.

We repeated the measure with patterns generated on the DBLP'05 summary. The containment times presented in 4.15 are 4 times smaller than for XMark. This is because the XMark summary contains many nodes named *bold*, *emph* etc., thus our pattern generator includes them often in the patterns, leading to large canonical models. A query using three *bold* elements, however, is not very realistic. Such formatting tags are less frequent in DBLP's summary, making DBLP synthetic patterns closer to real-life queries. We also tested patterns with 50%, and with 0% optional edges, and found optional edges slow containment by a factor of 2 compared to the conjunctive case. The impact is much smaller than the predicted exponential worst case (Section 4.3.2), demonstrating the algorithm's robustness.

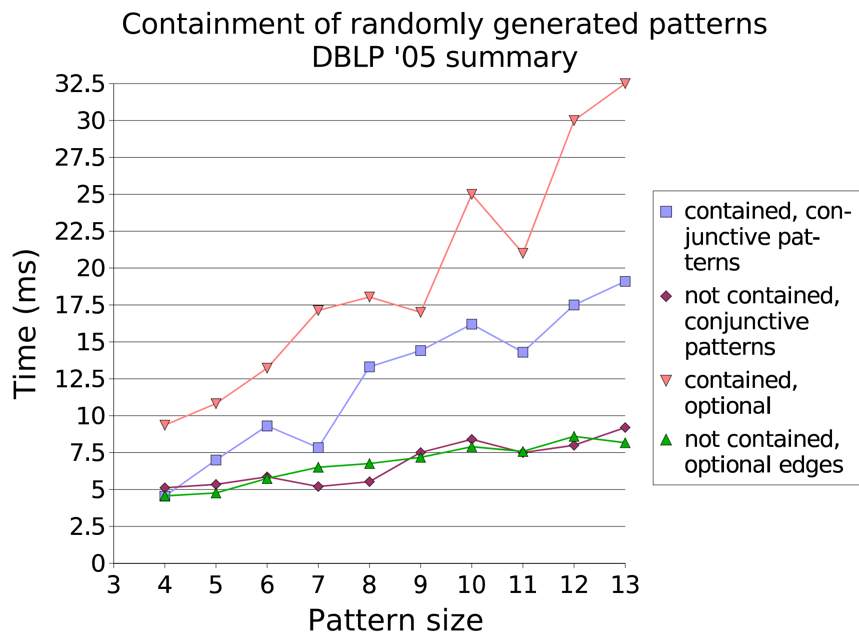



Figure 4.15: DBLP pattern containment.

# Chapter 5

# Rewriting XQuery queries using XAM views

## 5.1 Overview

Our global approach for rewriting XQuery queries can be described with the help of the schema in Figure 5.1. The first step consists of translating the user query $Q$ into an algebraic expression. In the most general case, this algebraic expression features un XML construction operator(XMLize), applied over a value-join expression which combines several query tree patterns ($XQ_1 \ldots XQ_n$ in the figure). This first step has been detailed in Chapter 3.

In the second step, each query tree pattern (or query XAM) $XQ_i$ is rewritten individually, based on a set of materialized views described by XAMs. The particular rewriting algorithm we describe uses a summary (introduced in Section 4.2.1), and relies on the summary-based containment algorithm described in Chapter 4. Observe that this step (tree pattern rewriting) can be performed based on other types of constraints, or even in their absence. (We do not investigate these possibilities in this thesis.) The result of rewriting one query XAM is a set of algebraic expressions (or logical plans) whose base nested relations are materialized views[1]. Thus, each rewriting $EQ_i^j$ of a query XAM is a logical plan, equivalent (under the summary constraints) to $XQ_i$.

In the final step, we compute rewritings that are S-equivalent to the query $Q$ by choosing one rewriting for each of the query patterns $XQ_i$ and replacing $XQ_i$ with the respective rewriting in $Q$'s algebraic expression. Thus, complete rewritings for $Q$

[1]The leafs of the logical plan are $Scan$ operators which were defined in Section 1.2.2

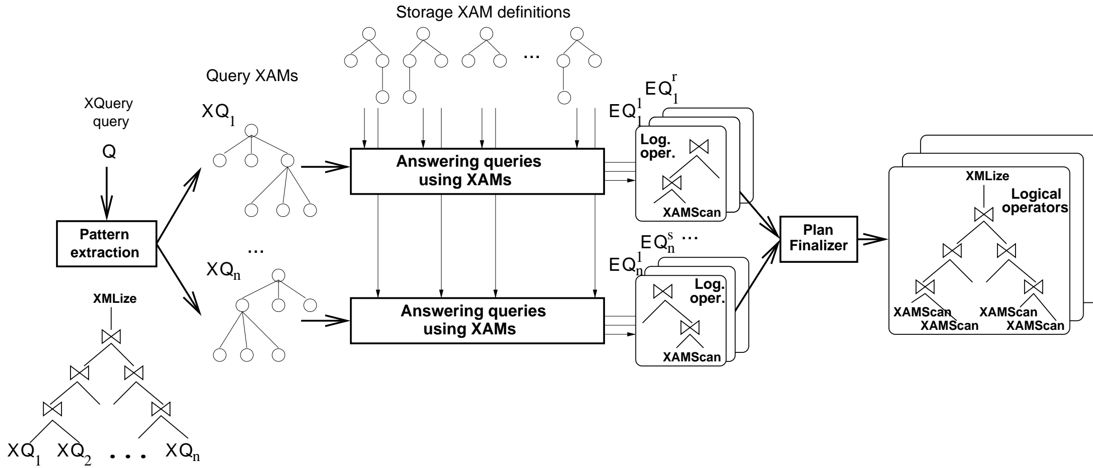



Figure 5.1: Query processing outline in ULoad.

are algebraic plans S-equivalent to Q, built over a subset of the available materialized views.

We end our overview with two observations. First, assuming that the second step, namely the rewriting of each query tree pattern based on tree pattern views is complete (which is indeed the case), it can be shown that our overal query rewriting approach is complete as well. In other words, rewriting $Q$ in a "piecemeal" fashion, one tree pattern at a time, does not lead to missing solutions. The intuition for this is that since value joins are not part of the view language, one does not miss solutions by rewriting tree patterns in separation.

Second, observe that our approach produces "total" query rewritings, that is, rewritings which do not assume some "base XML store" is available in order to evaluate parts of the query. Our view is that the store itself can be described by XAMs, as it is frequently the case (Section 2.3). Therefore, the rewriting algorithm is able to exploit together base storage structures, indices or materialized views, uniformly described as storage XAMs (which, for brevity, we term "materialized view XAMs" in this chapter).

## 5.2 Motivating example

As an example illustrating our rewriting approach, consider the following XQuery:

```
for $x in document("XMark.xml")//item[//mail] return
⟨res⟩ {$x/name/text(),
        for $y in $x//listitem return
        ⟨key⟩ {$y//keyword} ⟨/key⟩} ⟨/res⟩
```

Figure 5.2(a) shows a simplified XMark document fragment. At the right of each node's label, we show the node's identifier, e.g. $n_1$, $n_2$ etc. Figure 5.2(b) shows the structural summary of the document in Figure 5.2(a). We rewrite queries based on XAM-described materialized views. Figure 5.2(c) depicts the definitions of views $V_1$ and $V_2$, and the result obtained by evaluating the views over the sample document above.

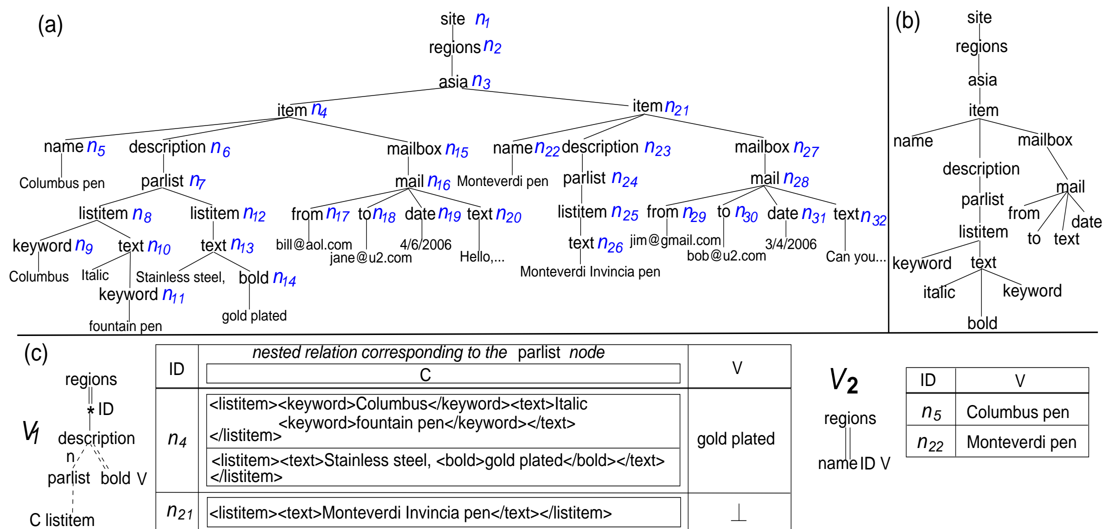



Figure 5.2: (a) XMark document fragment, (b) its structural summary and (c) two materialized views.

Rewriting can benefit from knowledge of the structure of the document and of the structural IDs as can be seen from our example.

*Summary-based rewriting* Consider the following rewriting opportunities that are enabled by the structural summary. First, although the tree pattern of $V_1$ does not explicitly indicate that $V_1$ stores data from ⟨item⟩ nodes, $V_1$ is useful if the structural summary in Figure 5.2(b) guarantees that all children of ⟨region⟩ that have ⟨description⟩ children are labeled item.

Second, in the absence of structural summaries, evaluation of the $y//keyword path of the query is impossible since neither $V_1$ nor $V_2$ store data from keyword nodes. However, if the structural summary implies that all //region//item//keyword nodes are

descendants of some //region//item/description/parlist/listitem, we can extract the keyword elements by navigating inside the content of ⟨listitem⟩ nodes, stored in the $A.C$ attribute of $V_1$.

Third, $V_1$ stores //region//*/description/parlist/listitem elements, while the query requires all ⟨listitem⟩ descendants of //regions//item. $V_1$'s data is sufficient for the query, if the summary ensures that //regions//item//listitem and //regions//*/description/parlist/listitem deliver the same data.

*Summary-based optimization* The rewritten query can be more efficient if it utilizes the knowledge of the structural summary. For example, $V_1$ may store some tuples that should not contribute to the query, namely from ⟨item⟩ nodes lacking ⟨mail⟩ descendants. In this case, using $V_1$ to rewrite our sample query requires checking for the presence of ⟨mail⟩ descendants in the $C$ attribute of each $V_1$ tuple. If all ⟨item⟩ nodes have ⟨mail⟩ descendants, $V_1$ only stores useful data, and can be used directly.

The above requires using structural information about the document and/or integrity constraints, which may come from a DTD or XML Schema, or from other structural XML summaries, such as Dataguides [52]. The XMark DTD [115] can be used for such reasoning, however, it does not allow deciding that //regions//item//listitem and //regions//*/description/parlist/listitem bind to the same data. The reason is that ⟨parlist⟩ and ⟨listitem⟩ elements are recursive in the DTD, and recursion depth is unbound by DTDs or XML Schemas. While recursion is frequent in XML, it rarely unfolds at important depths [84]. A Dataguide is more precise, as it only accounts for the paths occurring in the data; it also offers some protection against a lax DTD which "hides" interesting data regularity properties.

*Rewriting with rich XAM patterns* In addition to structural summaries, we also make use of the rich features of the XAMs, such as nesting and optionality. For example, in $V_1$, ⟨listitem⟩ elements are optional, that is, $V_1$ (also) stores data from ⟨item⟩ elements without ⟨listitem⟩ descendants. This fits well the query, which must indeed produce output even for such ⟨item⟩ elements. The nesting of ⟨listitem⟩ elements under their ⟨item⟩ ancestor is also favorable to the query, which must output such ⟨listitem⟩ nodes grouped in a single ⟨res⟩ node. Thus, the single view $V_1$ may be used to rewrite *across nested FLWR blocks*.

*Exploiting ID properties* Maintaining structural IDs enables opportunities for reassembling fragments of the input as needed. For example, data from ⟨name⟩ nodes can only be found in $V_2$. $V_1$ and $V_2$ have no common node, so they cannot be simply joined. If, however, the identifiers stored in the views are structural. $V_1$ and $V_2$ can be combined by a *structural join* [7] on their attributes $V_1.ID$ and $V_2.ID$. Furthermore, some ID schemes also allow inferring an element's ID from the ID of one of its chil-

dren [90, 107]. Assuming $V_1$ stored the ID of ⟨parlist⟩ nodes, we could derive from it the ID of their parent ⟨description⟩ nodes, and use it in other rewritings. Realizing the rewriting opportunities requires ID property information attached to the views, and reasoning on these properties during query rewriting. Observe that $V_1$ and $V_2$, together, contain all the data needed to build the query result *only if* the stored IDs are structural.

## 5.3 Summary-based rewriting of conjunctive patterns

In this section, we focus on the task of rewriting simple, conjunctive XAMs under summary constraints. Let $p_1, \ldots, p_n$ and $q$ be some patterns and $S$ be a summary. The rewritings we consider are logical algebraic expressions (or simply plans) built with the patterns $p_i$, and with the help of the algebra described in Section 1.2.2. The main operators used are: $\cup$ (duplicate-preserving union), $\bowtie_=$ (join pairing input tuples which contain exactly the same node), $\bowtie_\prec$ and $\bowtie_{\prec\prec}$ (structural joins, pairing input tuple whenever nodes from the left input are parents/ancestors of nodes from the right input), and $\pi$ (duplicate-preserving projection)[2].

Observe that allowing unions in rewritings leads to finding some rewritings for cases where no rewriting could be found without them. For instance, in Figure 5.4, the only rewriting of $p_1$ based on $q$ and $p_3$ is $q \cup p_3$. This contrasts with traditional conjunctive query rewriting based on conjunctive views [74], and is due to the summary constraints.

A set of plans is said *redundant* if it contains two plans $e_1, e_2$ returning the same data on any XML document (thus, regardless of any summary constraints). For instance, for any pattern $p$, if $e_1 = \pi_{n_1}(p)$ and $e_2 = \pi_{n_1}(\pi_{n_1,n_2}(p))$, the set $\{e_1, e_2\}$ is clearly redundant. We are not interested in redundant plans, since our focus is on rewriting under summary constraints. Furthermore, $e_1$ is typically preferrable to $e_2$ in the example above, since $e_1$ is more concise. More generally, among all plans $e$ that are $S$-equivalent to $q$, and also equivalent among themselves independently of $S$, we are interested in finding a *minimal* plan, i.e., one having the smallest number of operators (there may be several minimal plans, which we regard as equally interesting).

Our rewriting problem can thus be formally stated as: find a maximal, non-redundant set of plans $e$ over $p_1, \ldots, p_n$, such that each plan $e$ in the set satisfies $e \equiv_S q$.

Our query rewriting follows the general "generate-and-test" approach: produce candi-

[2] Other operators ($\sigma$, nesting and unnesting, and XPath navigation) will be introduced for more complex patterns in Section 5.4.

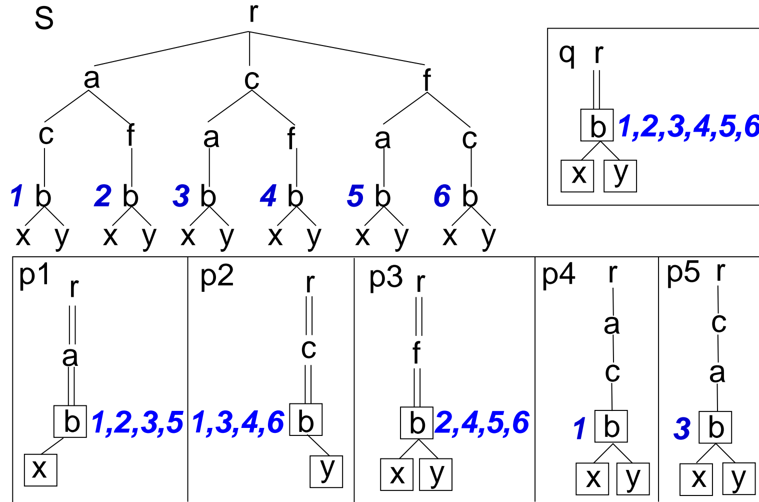



Figure 5.3: Summary $S$, query $q$ and patterns.

date rewritings, and test their equivalence to the query. We first consider testing.

We need to test the $S$-equivalence between a plan $e$ and a query pattern $q$. The usage of different formalisms for $e$ and $q$ has its advantages: tree patterns are suited for queries, while an algebraic rewriting language is easy to translate in executable plans. However, testing $S$-equivalence is more natural on patterns. Therefore, for each algebraic rewriting $e$, the rewriting algorithm builds an $S$-equivalent pattern $p_e$, and it is $p_e$ that will be tested for equivalence to $q$. However, all plans do not have an $S$-equivalent pattern ! For example, in Figure 5.3, no pattern is $S$-equivalent to $p_1 \bowtie_{b=b} p_2$. The intuition is that we can't decide whether $a$ should be an ancestor, or a descendant of $c$ in the hypothetic equivalent pattern. Fortunately, it can be shown that any plan is $S$-equivalent to some *union* of patterns. This is discussed in detail in Section 1.2.2. For example, $p_1 \bowtie_{b=b} p_2 \equiv_S (p_4 \cup p_5)$ in Figure 5.3. Thus, to test whether $e \equiv_S q$, we can rely on our algorithms for testing the $S$-equivalence of $q$ with (a union of) patterns corresponding to $e$. The containment tests may be expensive, thus the importance of a rewriting generation strategy that does not produce many unsuccessful ones.

Let us now consider possible generation strategies. Conjunctive query rewriting, in the relational setting [74] as well as in more recent XML-oriented incarnations [38], follows a "bucket" approach, collecting all possible rewritings for every query atom (or node), and combining such partial rewritings into increasingly larger candidate rewritings.

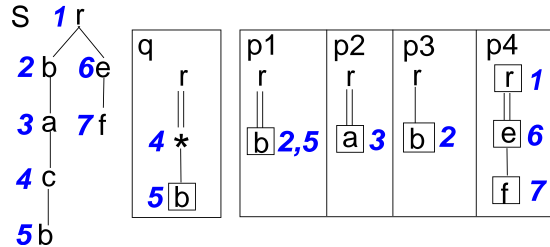



Figure 5.4: Pattern join configuration.

Bucket-style generation of rewritings is not adapted in the presence of summary constraints. First, rewriting must consider also views that do not fit in any bucket, i.e. do not cover any query atom. For instance, in Figure 5.4, $q$ asks for $b$ elements at least two levels below the root, while $p_1$ provides all $b$ elements, including some not in $q$. The pattern $p_2$ does not cover any $q$ nodes, yet $(p_2 \bowtie_{a \prec\!\!\prec b} p_1) \equiv_S q$. Second, a rewriting may fail to cover some query nodes, yet be equivalent to the query. For instance, consider a summary $S = r(a(b))$, the query $q = /r//a//b$, and the pattern $p_1 = /r//b$. Clearly, $p_1 \equiv_S q$, yet $p_1$ lacks an $a$ node (implicitly present above $b$, due to the $S$ constraints).

As a consequence, our basic generation approach should not start with buckets, but proceed in an inflationary manner, combining views into increasingly larger plans.

**Characterization of the search space** We have a query pattern $Q$, whose number of nodes we denote as $n(Q)$. $Q$ has $n(Q) - 1$ edges.

We want to cover $Q$ with some query plan $P$, which is a join plan over XAMs. We may count the size of $P$ in the number of XAMs it involves, and denote this as $|P|$. Covering $Q$ means covering all the atoms of $Q$, that is: finding data for the return nodes, while making sure that all conditions that constrain these return nodes are respected. These conditions are materialized by the edges which connect them with the rest of the XAM – ultimately, all XAM edges.

How much effort do we need to do to enforce one such condition, one edge in $Q$ ?

- In the best case, nothing (this is the case when one XAM alone perfectly fits $Q$). Thus, the lower bound for $|P|$ is 1.

- In the worst case, a chain of joins whose maximal height is bounded by $|S|$, the size of the path summary.

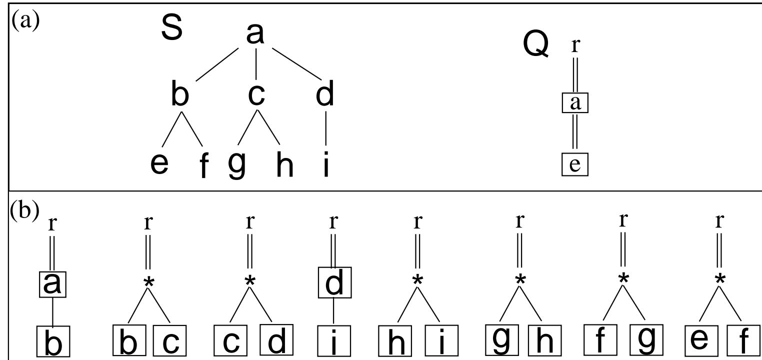



Figure 5.5: (a) A summary S and a query to be evaluated on S. (b) A view set that produces the bigest number of joins in the rewritting of Q on the summary S

This means $1 \leq |P| \leq (n(Q)-1)*|S|$. No plan bigger than $(n(Q)-1)*|S|$ needs to be explored An example of the worst case in wich $|P| = (n(Q)-1)*|S|$ is displayed in Figure 5.5. Trying to rewrite the query Q on the summary S using the view set present in Figure 5.5(b) we need to build a plan that uses all the views in the set in order to gather the $a$ elements paired with their $e$'s. [3]

The size of the query plan search space is: the number of intermediary plans we need to build, to attain at most size $k = (n(Q)-1)*|S|$. Think of such plans as being $m$-way joins over XAMs, where $m \leq k$. If we fix $m$ XAMs to join and we fix the corresponding join predicates, we only need to build one such join plan (not explore various alternative join trees that apply the same predicates on the same XAMs, in some different order and join paranthesizing).

*Summary-based pruning* A summary enables restricting the set of views initially considered for rewriting, without losing solutions. Assume that for a view $p_i$, for any $n_p \in nodes(p_i) \setminus root(p_i)$ and $x$ associated path of $n_p$, and for any $n_q \in nodes(q) \setminus root(q)$ and $y$ associated path of $n_q$, $x \neq y$, $x$ is neither an ancestor nor a descendant of $y$. Then, the rewriting algorithm can ignore $p_i$. The intuition is $p_i$'s data belongs to different parts of the document than those needed by the query. An example is pattern $p_4$ for the rewriting of $q$ in Figure 5.4.

The summary can also be used to restrict the set of intermediary rewritings. The rewriting algorithm manipulates $(e,p)$ (plan, pattern) pairs, where $e \equiv_S p$. Consider

[3]This is just an upper bound. In practice, if one edge actually needed $|S|$ structural joins on top of one another, then another edge above or below this one cannot require $|S|$ edges again (since the total height of the join tree is $|S|$). If the query is a vertical chain of nodes, $|P|$ is bounded by $|S|$.

**Algorithm 2**: Summary-based pattern rewriting

```
   Input  : summary S, patterns p_1, ..., p_n, q
   Output: rewritings of q using p_1, ..., p_n
1  M_0 ← {(p_i, p_i) | 1 ≤ i ≤ n}; M ← M_0
2  repeat
3      foreach (e_i, p_i) ∈ M, (e_j, p_j) ∈ M_0 do
4          foreach possible way of joining e_i and e_j using ⋈_{=id}, ⋈_≺, ⋈_≺≺ do
5              (e, p) ← (e_i, p_i) ⋈ (e_j, p_j)
6              if p ≠ p_i and p ≠ p_j then
7                  if p ≡_S q then
8                      output l
9                  else
10                     if |e| ≤ |q| × |S| then
11                         M ← M ∪ {(e, p)}
12 until M is stationary
13 foreach minimal N ⊆ M s.t. ∪_{(e,p)∈N} p ≡_S q do
14     output ∪_{(e,p)∈N} p
```

two pairs $(e_x, p_x)$ and $(e_y, p_y)$, and a possible join result $(e_z, p_z) = (e_x, p_x) \bowtie (e_y, p_y)$. ($i$) If $p_z$ is $S$-unsatisfiable, we can discard $(e_z, p_z)$. ($ii$) If the produced pattern (or pattern union) $p_z$ coincides with $p_x$, we may discard the partial rewriting $(e_z, p_z)$, since any complete rewriting $e'$ based on $e_z$ is non-minimal ($e_z$ can be replaced with $e_x$, which is smaller). The role of $S$ here is to prune non-minimal plans.

*Summary-based reduction of containment tests* Structural summaries also allow reducing the number of containment tests performed during rewriting. Since containment is only defined on same-arity patterns, prior to testing whether $p \subseteq_S q$, one must identify $k$ return nodes of $p$, where $k$ is the arity of $q$, extract from $p$ a pattern $p'$ returning those $k$ nodes, then test if $p' \subseteq_S q$. If $p$'s arity is smaller than $k$, clearly $p \not\equiv_S q$. Otherwise, there are many ways of choosing $k$ return nodes of $p$, which may lead to a large number of containment tests. However, if $p \subseteq_S q$, then for every return node $n_i$ of $p$ and corresponding return node $m_i$ of $q$, the $S$ paths associated to $n_i$ must be a subset of the $S$ paths associated to $m_i$. We only test containment for those choices of $k$ nodes of $p$ satisfying this path condition.

*Rewriting algorithm* Algorithm 2 outlines the rewriting process discussed above. $M_0$ is the set of initial (plan, pattern) pairs, and $M$ is the set of intermediary rewritings.

Initial view pruning is applied prior to step 1, while partial rewritings are pruned in step 6 (for conciseness, some pruning steps are omitted). Lines 13-14 compute union plans. The algorithm's soundness is guaranteed by the equivalence test (line 7). The algorithm is complete due to its exhaustive search. The search space is finite thanks to the summary: it can be shown that all patterns above a certain size (thus their equivalent plans) have an equivalent smaller pattern (thus plan). The complexity is determined by the size of the search space, multiplied by the complexity of an equivalence test. The search space size is in $O(2^{C_{|p|}^{|q|}})$, where $|p| = \Sigma_{i=1,\ldots,n}|nodes(p_i)|$ and $|q| = |nodes(q)|$ (the formula assumes that every $p_i$ node, $1 \leq i \leq n$, can be used to rewrite every $q$ node).

## 5.4 Extending rewriting

The algorithm presented in the previous section is appropriate for simple, unnested conjunctive XAMs which only return node IDs, and relies on a simple summary. Considering the full XAM language and enhanced summaries has two types of consequences. First, the equivalence (thus, the two-way containment) test performed at line 7 in Algorithm 2 must conside the appropriate XAM canonical model. Second, a set of changes are needed in the rewriting algorithm. We detail them in this section.

*Decorated patterns* entail the following adaptation of Algorithm 2. Whenever a join plan of the form $l_1 \bowtie_{n_1=n_2} l_2$ is considered (line 5), the plan is only built if $\phi_{n_1}(v) \wedge \phi_{n_2}(v) \neq F$, in which case, the node(s) corresponding to $n_1$ and $n_2$ in the resulting equivalent pattern(s) are decorated with $\phi_{n_1}(v) \wedge \phi_{n_2}(v)$.

*Optional patterns* can be handled directly.

*Attribute patterns* require a set of adaptations. First, some selection ($\sigma$) operators may be needed to ensure no plan is missed, as follows. Let $p$ be a pattern corresponding to a rewriting and $n$ be a $p$ node. At lines 7 and 13 of the algorithm 2, we may want to test containment between $q$ (the target pattern) and (a union involving) $p$. Let $n_q$ be the $q$ node associated to $n$ for the containment test.

- If $n$ is labeled $*$ and stores the attribute $L$ (label), and $n_q$ is labeled $l \in \mathcal{L}$, then we add to the plan associated to $p$ the selection $\sigma_{n.L=l}$.
- If $n$ is decorated with the formula $\phi_n(v) = T$ and stores the attribute $V$ (value), and $n_q$ is decorated with the formula $\phi_{n_q}(v)$, then we add to the plan associated to $p$ the selection $\sigma_{\phi_{n_q}(v)}$.

Second, prior to Algorithm 2, we *unfold* all $C$ attributes in the query and view patterns:

- Assume the node $n$ in pattern $p$ has only one associated path $s \in S$. To unfold $n.C$, we erase $C$ and add to $n$ a child subtree identical to the $S$ subtree rooted in $s$, in which all edges are parent-child and optional, and all nodes are labeled with their label from $S$, and with the $V$ attribute.
- If $n$ has several associated paths $s_1, \ldots, s_l$, then $(i)$ decompose $p$ into a union of disjoint patterns such that $n$ has a single associated path in each such pattern and $(ii)$ unfold $n.C$ in each of the resulting patterns, as above.

Before evaluating a rewriting plan, the nodes introduced by unfolding must be extracted from the $C$ attribute actually stored by the ancestor $n$. This is achieved by XPath navigation on $n.C$. Rewritings in this case will use a $nav_q$ operator, where $q$ is an XPath expression using the child and descendant axes, applying on the $C$ attribute. Navigation-based rewriting is studied in [18, 79, 120].

A view pre-processing step may be enabled by the properties of the ID function $f_{ID}$ employed in the view. For some ID functions, e.g. ORDPATHs [90] or Dewey IDs [107], $f_{ID}(n)$ can be derived by a simple computation on $f_{ID}(n')$, where $n'$ is a child of $n$. If such IDs are used in a view, let $n_1 \in p_i$ be a node annotated with $ID$, and $n_2$ be its parent. Assume $n_1$ is annotated with the paths $s_1^1, \ldots, s_k^1$, and $n_2$ with the paths $s_1^2, \ldots, s_l^2$. If the depth difference between any $s_i^1$ and $s_j^2$ (such that $s_j^2$ is an ancestor of $s_i^1$) is a constant $c$ (in other words, such pairs of paths are all at the same "vertical distance"), we may compute the ID of $n_2$ by $c$ successive parent ID computation steps, starting from the values of $n_1.ID$.

Based on this observation, we add to $n_2$ a "virtual" ID attribute annotation, which the rewriting algorithm can use as if it was originally there. This process can be repeated, if $n_2$'s parent paths are "at the same distance" from $n_2$'s paths etc. Prior to evaluating a rewriting plan which uses virtual IDs, such IDs are computed by a special operator $nav_{f_{ID}}$ which computes node IDs from the IDs of its descendants.

*Nested patterns* entail the following adaptations:

First, Algorithm 2 may build, beside structural join plans (line 5), plans involving *nested structural joins*, which can be seen as simple joins followed by a grouping on the outer relation attributes. Intuitively, if a structural join combines two patterns in a large one by a new unnested edge, a nested structural join creates a new nested edge.

Second, prior to the containment tests, we may adapt the nesting path(s) of some nodes in the patterns produced by the rewritings. Let $(l, r)$ be a plan-pattern pair pro-

duced by the rewriting. $(i)$ If $r$ has a nesting step absent from the corresponding $q$ node, we eliminate it by applying an *unnest* operator on $l$. $(ii)$ If a $q$ node has a nesting step absent from the nesting sequence of the corresponding $r$ node, if this $r$ node has an $ID$ attribute, we can produce the required nesting by a *group-by ID* operator on $l$; otherwise, this nesting step cannot be obtained, and containment fails.

## 5.5 Building equivalent plan-pattern pairs

This section describes our approach for building algebraic plans together with equivalent tree patterns throughout the optimization process. Section 5.5.1 define the problem, while Section 5.5.2 details the most technically involved step in this process: synthesizing tree patterns equivalent to a given complex query plan.

### 5.5.1 Examples and problem statement

As shown in Algorithm 2(line 5), during the rewriting of a query XAM, we combine algebraic plans via joins. However, each of the plans had an equivalent tree pattern, and we still must construct a pattern equivalent to the new join plan. These patterns are needed in order to test when a given plan is equivalent to the query XAM. As we will see, in some cases no equivalent pattern exists, but a union of patterns can be found such that this union is equivalent to the newly constructed plan. We discuss some illustrative examples next.

Figure 5.6 depicts a summary $S$, whose $b$ nodes are numbered for convenience, and a set of patterns. Next to each node labeled $b$ in these patterns, we show its path annotation[4] against $S$ as a set of integers, corresponding to the respective $b$ nodes in $S$. We assume patterns $p_1, p_2, p_4$ and $p_7$ describe the data stored in the materialized views $v_1, v_2, v_4$ and $v_7$, thus the rewriting algorithm starts with the (plan, pattern) pairs $(v_1, p_1)$, $(v_2, p_2)$, $(v_4, p_4)$ and $(v_7, p_7)$.

Let us consider combining $(v_1, p_1)$ and $(v_2, p_2)$. The pattern equivalent to $v_1 \bowtie_{bID=bID} v_2$ would return the IDs of $b$ elements with an $a$ ancestor, together with their $x$ and $y$ children's ID. The resulting equivalent pattern turns out to be $p_3$ in Figure 5.6.

Now consider combining $(v_1, p_1)$ with $(v_4, p_4)$. In the pattern $S$-equivalent to $v_1 \bowtie_{bID=bID} v_4$, the $b$ node should have both an $a$ and a $c$ ancestor. Moreover, in this pattern, the $b$ node's annotation should be the intersection of the $b$ nodes' annotations in

[4]We recall the definition of path annotations from Definition 4.3.1

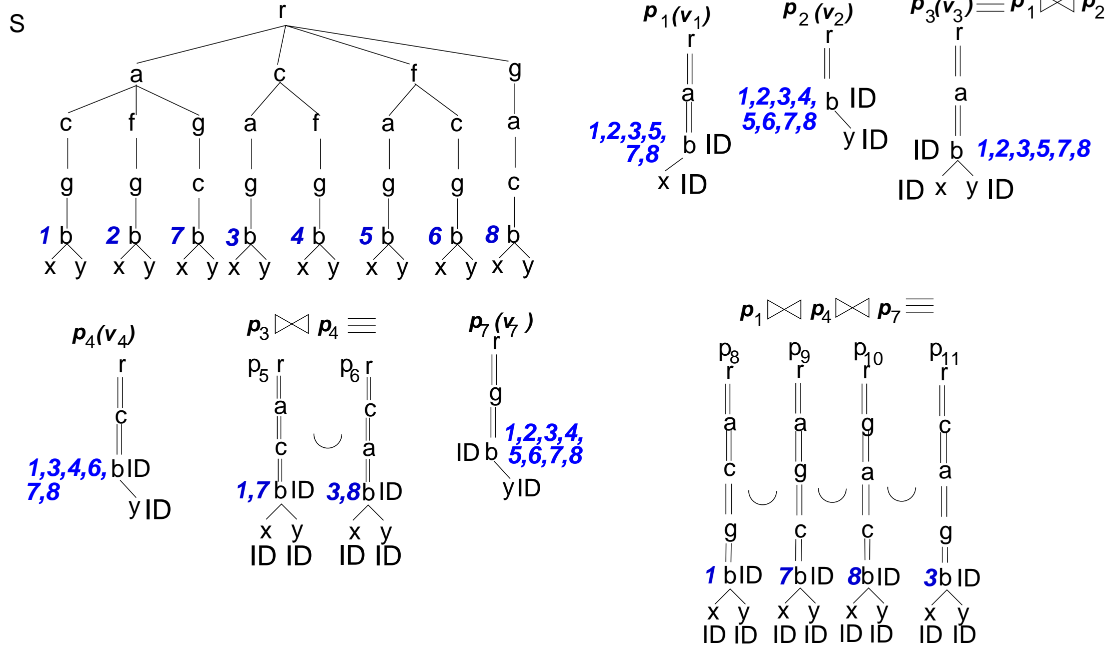



Figure 5.6: Sample summary $S$, patterns representing stored structures $(p_1, p_2, p_4, p_7)$ and patterns obtained by successive joins $(p_3, p_5, p_6, p_8, p_9, p_{10}, p_{11})$.

$p_1$ and $p_4$, that is, the path set $\{1, 3, 7, 8\}$. From the structure of $S$, it follows that $a$ is an ancestor of $c$ on paths 1 and 7, while $c$ is an ancestor of $a$ on paths 3 and 8. Thus, there is no way to place $a$ and $c$ as ancestors of $b$ in a hypothetical pattern $S$-equivalent to $v_1 \bowtie_{bID=bID} v_4$. However, if we relax the problem by also considering *pattern unions*, then a solution can be found. For instance, in Figure 5.6, $v_1 \bowtie_{bID=bID} v_4 \equiv_S p_5 \cup p_6$.

Now consider the join plan $(v_1 \bowtie_{bID=bID} v_4) \bowtie_{bID=bID} v_7$. By a similar reasoning carrying over the $a$, $c$ and $g$ ancestors of the $b$ node, no single pattern is $S$-equivalent to this plan, however the union $p_8 \cup p_9 \cup p_{10} \cup p_{11}$ is equivalent to this plan.

It can be shown that any algebraic plan built with $\bowtie_=$, $\bowtie_\prec$, $\bowtie_{\prec\prec}$, and $\pi$ on top of some patterns $p_1, \ldots, p_n$ is $S$-equivalent to a union of patterns. While the patterns in Figure 5.6 only feature non-optional, non-nested edges, this result carries over for general-case XAMs.

### 5.5.2 Computing the pattern equivalent to a join plan

Algorithm 3 shows how to combine two (plan,pattern) pairs via a join on the plans, into a resulting (plan,pattern) pair. Its input are two patterns $p_1$ and $p_2$, and its output is a pattern $p$ which is $S$-equivalent to $p_1 \bowtie_{a_k.ID=b_l.ID} p_2$. We call this algorithm *pattern zipping*, due to the way it handles pattern nodes (see below). Observe that we may sometimes need to zip *unions of patterns*, not simple patterns; for simplicity we focus on zipping two patterns for now, and will extend this later to pattern unions.

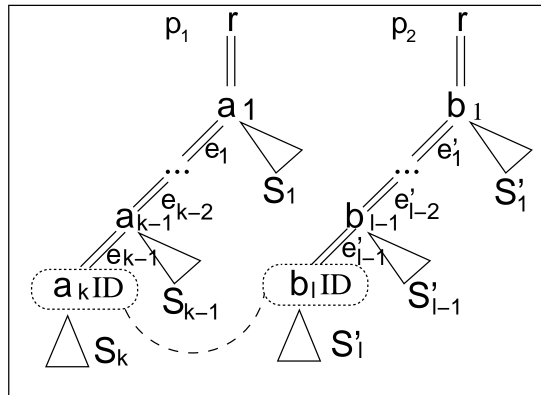



Figure 5.7: Zipping patterns $p_1$ and $p_2$ on $a_k.ID = b_l.ID$.

The zipping algorithm can be traced by considering Figure 5.7. Here, $p_1$ and $p_2$ are shown distinguishing the ancestors of $a_k$ in $p_1$ with the $a_1, a_2, \ldots, a_{k-1}$ labels, and similarly the ancestors of $b_l$ in $p_2$. The edges on the path from $p_1$'s root to $a_k$ are labeled $e_1, e_2, \ldots, e_{k-1}$, while the remaining children of each $a_i$ node are grouped in a forest denoted $S_i$. Similar $e'$ edge labels and $S'$ forests are outlined in $p_2$.

Algorithm 3 is called with a focus on two specific nodes $a_i$ and $b_j$ from $p_1$, respectively $p_2$. Initially, $i = k$ and $j = l$; in subsequent recursive calls, $a_i$ and $a_j$ will move up from $a_k$, respectively $b_l$, towards the pattern roots. It is due to this gradually moving pair of nodes that we call the algorithm "zipping".

We assume $a_k$ and $b_l$ have compatible labels (either the same label, or one label is $*$), compatible conditions on their values, if any, and the intersection of their path annotations is non-empty (otherwise, the join result is empty).

**Algorithm 3**: Compute the equivalent pattern corresponding to the join of $p_1$ and $p_2$ on $a_k.ID = b_l.ID$

**Input** : Pattern $p_1$, pattern $p_2$, pattern node $a_i$, pattern node $b_j$, pattern node $a_k$, pattern node $b_l$

**Output**: A set of patterns $\{r_1, r_2, \ldots, r_m\}$ such that $r_1 \cup r_2 \cup \ldots \cup r_m \equiv_S p_1 \bowtie_{a_i.ID=b_j.ID} p_2$

1 **if** $a_i = a_k$ *and* $a_j = b_l$ **then**
2 copy $a_k$ and its subtree in $p$; add a copy of $S'_j$ as $a_k$ children in $p$; $\alpha(a_k, p) \leftarrow \alpha(a_k, p_1) \cap \alpha(b_l, p_2)$;
3 *propagatePathComputations(*$p, a_k$*)*;
4 return computeEquivPattern($p_1$, $p_2$, $a_{k-1}$, $b_{l-1}$, $a_k$, $b_l$);
5 **else**
6 $P_1 \leftarrow$ all paths from $\alpha(a_i, p_1)$ that are ancestors of a path from $\alpha(b_j, p_2)$;
7 $P_2 \leftarrow$ all paths from $\alpha(b_j, p_2)$ that are ancestors of a path from $\alpha(a_i, p_1)$;
8 $P_3 \leftarrow \alpha(a_i, p_1) \cap \alpha(b_j, p_2)$;
9 **if** $P_1 \neq \emptyset$ **then**
10 copy $p_1$ into a new pattern $p$;
11 insert $p_2.b_j$ between $p.a_i$ and $p.a_{i+1}$ connected with $p.a_i$ by the edge $e_{i-1}$ and with $p.a_{i+1}$ by the edge $e'_{j-1}$;
12 copy $S_j$ trees under $p.a_i$;
13 $\alpha(a_i, p) \leftarrow P_1$;
14 *propagatePathComputations*$(p, a_i)$;
15 $res_1 \leftarrow$ *computeEquivPattern*$(p, p_2, a_i, b_{j-1}, a_k, b_l)$;
16 **if** $P_2 \neq \emptyset$ **then**
17 copy $p_2$ into a new pattern $p$;
18 insert $p_1.a_i$ between $p.b_j$ and $p.b_{j+1}$, connected with $p.b_j$ by the edge $e_{j-1}$ and with $p.b_{j+1}$ by the edge $e'_{i-1}$;
19 $\alpha(b_j, p) \leftarrow P_2$;
20 *propagatePathComputations*$(p, b_j)$;
21 $res_2 \leftarrow$ *computeEquivPattern*$(p, p_2, a_{i-1}, b_j, a_k, b_l)$;
22 **if** $P_3 \neq \emptyset$ **then**
23 copy $S_j$ as children of $a_i$ in $p_1$;
24 $\alpha(p, a_i) \leftarrow P_3$;
25 *propagatePathComputations*$(p_1, a_i)$;
26 $res_3 \leftarrow$ *computeEquivPattern*$(p, p_2, a_{i-1}, b_{j-1}, a_k, b_l)$;
27 **return** $res_1 \cup res_2 \cup res_3$;

At the first invocation, $a_i = a_k$ and $b_j = b_l$, and Algorithm 3 (lines 1-4) computes the subtree of the node which, in the resulting pattern $p$, corresponds to the unification of nodes $a_k$ and $b_l$. We will call this node $a_k$ also in $p$. The $p$ subtree rooted in $a_k$ is built as follows:

- The children of $b_l$ (the $S'_l$ forest in Figure 5.7) are copied as children of $a_k$ in $p$. Thus, the children of $a_k$ in $p$ are the subtrees from forests $S_k$ and $S'_l$.
- The path annotation for $a_k$ in $p$ is computed as the intersection of the paths annotations of $a_k$ in $p_1$ and of $b_l$ in $p_2$.
- The path annotations of $a_k$'s descendants in $p$ are modified to reflect $a_k$'s path annotation. We call this *downward path propagation from* $a_k$ *in* $S_k$. During downward propagation, for every descendant $x$ of $a_k$ in $S_k$, the path annotation of $x$ is restricted to those paths which are descendants of some path now annotating $a_k$. If for a non-optional descendant $x$, the path annotation has become $\emptyset$, then $p$ (and the join) have empty result.
  We have still not determined the shape of $p$ *above* the node $a_k$; this is more delicate and may lead to pattern unions, as illustrated by $v_1 \bowtie_{bID=bID} v_4$ in Figure 5.6. In preparation of that step, the path annotation of node $a_{k-1}$ in $p_1$ is restricted to those paths which have a descendant in the path annotation of $a_k$ in $p$. Then, downward path propagation is applied from $a_{k-1}$ into $S_{k-1}$, subtree of $p_1$ (see Figure 5.7). The process is repeated on $a_{k-2}$, ..., $a_1$. Similarly, the annotations of $b_{l-1}$ are restricted, then propagated into $S'_{l-1}$, and so on until $b_1$.

The part of the resulting pattern *above* node $a_k$ is computed by lines 5-26 in Algorithm 3.

For the current $a_i$ and $b_j$ nodes, we compute three sets as follows: $P_1$ holds the paths from the annotation of $a_i$ that are ancestors of a path from the annotation of $b_j$. Symmetrically, $P_2$ holds the paths from $\alpha(p_2, b_j)$ that are ancestors of a path from $\alpha(p_1, a_j)$. Finally, $P_3$ gathers the paths belonging both to the path annotation of $a_i$ in $p_1$, and to the annotation of $b_j$ in $p_2$.

If $P_1$ is not empty, this means that for some documents conforming to $S$, some XML nodes matching $a_i$ are ancestors of some nodes matching $b_j$. Therefore, in the pattern $p$ equivalent to $p_1 \bowtie_{a_k.ID=b_l.ID} p_2$ (or in one of the patterns appearing in the equivalent pattern union), node $a_i$ may appear *as an ancestor* of node $b_j$. In this case, we insert $b_j$ in $p$ *between* $a_i$ and $a_{i+1}$. Of course, $b_j$ preserves its children subtrees. Having decided on the relative order of $a_i$ and $b_j$ on the path from $a_k$ to the root of the resulting pattern, we move one step upwards, and call the algorithm again, to decide

the relative positions of nodes $a_i$ and $b_{j-1}$. In this case, zipping advances one level up, both to $a_{i-1}$ and to $b_{j-1}$. The resulting pattern(s) are gathered in $res_1$.

Similarly, if $P_2$ is not empty, then node $a_i$ may appear underneath $b_j$ in $p$ (or one of the patterns in the equivalent pattern union, if a single pattern cannot be found). In a symmetric manner, $a_i$ is inserted between $b_j$ and $b_{j+1}$ and the algorithm is called again to place $a_{i-1}$ and $b_j$. The resulting patterns are gathered in $res_2$.

Finally, if $P_3$ is not empty, the nodes $a_i$ and $b_j$ may be unified in a single one in $p$ (or one of the patterns in the equivalent pattern union). The resulting patterns are gathered in $res_3$.

The patterns in $res_1 \cup res_2 \cup res_3$ make up the result of the algorithm.

We can follow examples of Algorithm 3's execution on the patterns in Figure 5.6.

- When computing $p_1 \bowtie_{b.ID=b.ID} p_2$ (recall the patterns in Figure 5.6), the zip algorithm will first gather the $x$ and $y$ children of the $b$ node in the result. Then, the zip algorithm is called again on the $a$ node from $p_1$ and the $r$ node from $p_2$. We have $\alpha(p_1, a) = \{/r/a, /r/c/a, /r/g/a, /r/f/a\}$ and $\alpha(p_2, r) = \{/r\}$, thus $P_1 = \emptyset$, $P_2 = \alpha(p_2, b)$ and $P_3 = \emptyset$. Thus, $a$ is inserted under $r$ in the resulting pattern, and the zip algorithm is reinvoked on the $r$ node in $p_1$ and the $r$ node in $p_2$, which unifies them (lines 22-26) and the resulting pattern is $p_3$.
- When computing $p_1 \bowtie_{b.ID=b.ID} p_4$, after unifying the $b$ nodes and cumulating their children, the zip algorithm is called on the $a$ node from $p_1$ and the $c$ node from $p_2$. Here, $\alpha(p_1, a) = \{/r/a, /r/c/a, /r/g/a, /r/f/a\}$, $\alpha(p_2, c) = \{/r/a/c, /r/c, /r/f/c, /r/g/c\}$, leading to: $P_1 = \{/r/a\}$, $P_2 = \{/r/c\}$, $P_3 = \emptyset$. This leads to the creation of two patterns, one (the future $p_6$) with the $a$ node above the $c$ node due to $P_1$, the other (the future $p_7$) with the $c$ node above the $a$ node. Each of these patterns is completed by an extra invocation of the zip algorithm.

PROPOSITION 5.5.1. The pattern union computed by Algorithm 4 is equivalent, under the path summary constraints, to the plan $p_1 \bowtie_{(a_k.ID=b_l.ID)} p_2$. $\triangleleft$

The proof of this proposition relies on XAM containment algorithms under summary constraints (Chapter 4).

Several simple extensions increase the generality of the zipping algorithm and we briefly present them now:

First, ID joins can be directly extended to ID-based semijoins, nested joins, (nested) outerjoins etc. We only need to use the proper join operator in the algebraic plan, and assign to the edges connecting the $b_j$'s children to their parents in the resulting pattern the correct annotation. Thus, optional edges reflect outerjoins, while nested edges reflect nested joins.

Second, the ID-based join can be replaced in a straightforward manner with a structural join, requiring that node $a_k$ from $p_1$ be an ancestor/parent of node $b_l$ from $p_2$. In this case, $a_k$ and $b_j$ are not unified as in lines 1-5, rather, the algorithm proceeds directly to zip $a_k$ and $b_{j-1}$, since we know $a_k$ must appear above $b_j$ in the resulting pattern(s). Similar extensions deal with structural semijoins and outerjoins, and nested structural joins.

Finally, the zipping algorithm as defined above may only zip one pattern with another, however it can produce a union of patterns. The simple extension presented in Algorithm 4 handles the general case, when we need to zip one pattern union with another such union.

---

**Algorithm 4**: ZipUnion

**Input** : Pattern union $p_1^1 \cup p_1^2 \cup \ldots \cup p_1^m$, pattern union $p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n$, pattern node $a_k$, pattern node $b_l$

**Output**: Pattern union $p^1 \cup p^2 \cup \ldots \cup p^r$ such that $p^1 \cup p^2 \cup \ldots \cup p^r \equiv_S (p_1^1 \cup p_1^2 \cup \ldots \cup p_1^m) \bowtie_{a_k.ID=b_l.ID} (p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n)$

1 $res \leftarrow \emptyset$

2 **foreach** *pattern* $p_1 \in \{p_1^1, p_1^2, \ldots, p_1^m\}$ **do**

3     **foreach** *pattern* $p_2 \in \{p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n\}$ **do**

4         $res \leftarrow res \cup$ computeEquivPattern($p_1$,$p_2$,$a_k$, $b_l$,$a_k$, $b_l$)

5 return $res$

---

## 5.6 Experimental evaluation of XAM rewriting

We rewrite the query patterns extracted from the XMark queries [115]. The view pattern set is initialized with 2-node views, one node labeled with the XMark root tag, and the other labeled with each XMark tag, and storing $ID$, $V$, to ensure *some* rewritings exist. Experimenting with various synthetic views, we noticed that large synthetic view patterns did not significantly increased the number of rewritings found, because the risk that the view has little, if any, in common with the query increases with

the view size. The presence of random value predicates in views had the same effect. Therefore, we then added 100 random 3-nodes view patterns based on the XMark233 summary, with 50% optional edges, such that a node stores a (*structural*) $ID$ and $V$ with a probability $0.75$. Figure 5.8 shows for each query: the time to prepare the rewriting and prune the views as described in Section 4.4, the time elapsed until the first equivalent rewriting is found (this includes the setup time), and the total rewriting time. The first rewriting is found quite fast. This is useful since in the presence of many rewritings, the rewriting process may be stopped early. Also, view pruning was very efficient: of the 183 initial views, on average only 57% were kept.

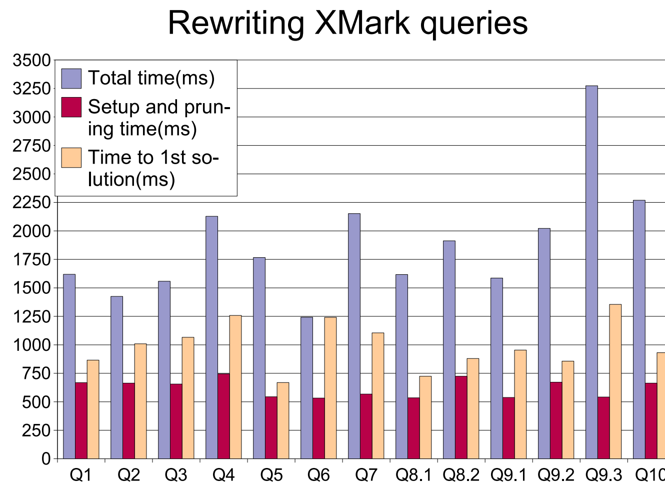



Figure 5.8: XMark query rewriting

*Experiment conclusions* Pattern containment performance closely tracks the canonical model size for positive tests; negative tests perform much faster. Containment performance scales up with the summary and pattern size. Rewriting performance depends on the views and number of solutions; a first rewriting is identified fast.

### Plan combination performance

In this section, we show that computing equivalent patterns has a very moderate impact on the total effort spent optimizing a query by constructing algebraic plans *only*.

To that effect, we used the largest summary we could find for an XMark document, namely the one with 548 nodes. We used the query patterns extracted from the XMark

queries 1 to 10 as the target patterns. The view pattern set is initialized with 2-nodes views of the form $/site//X$, for each distinct tag $X$ appearing in an XMark document. Each such view stores the $ID$ and $V$ (value) of the $X$-labeled nodes; the purpose of these initial views is to ensure that *some* rewritings will be found for every query.

Experimenting with various synthetic views, we noticed that large synthetic view patterns did not significantly increase the number of rewritings found, because the risk that the view has little, if any, in common with the query increases with the view size. Therefore, we added to the initial set of views 100 randomly-generated view patterns, based on the XMark summary. Each such view has 3 nodes (plus the root). $50\%$ of the edges in these views are optional. $75\%$ of the view nodes store IDs and values. We assume all IDs enable structural joins. No value predicates were added.

The graph at the left in Figure 5.9 shows the times in milliseconds to rewrite the XMark query patterns using all the 183 views described above. Multiple query patterns correspond to query 8 and 9, since the query performs a value join over several independent tree patterns. For each query, we show the time needed to compute the equivalent patterns, as a part of the total rewriting time. At the right, we show the number of intermediary (plan,pattern) pairs that had to be constructed in the rewriting process for each query.

We remark that the time needeed to construct an equivalent pattern is moderate (at most $20\%$ of the total time). The optimization process overall performs reasonably fast when considering the number of (plan, pattern) pairs explored. As previously mentioned, practical rewriting or optimization algorithms typically stop before exploring the full search space, while our tests explore it all.

We conclude that our algorithm for developing patterns in parallel with the development of algebraic plans is practically feasible and affordable in real-life XML optimizers.

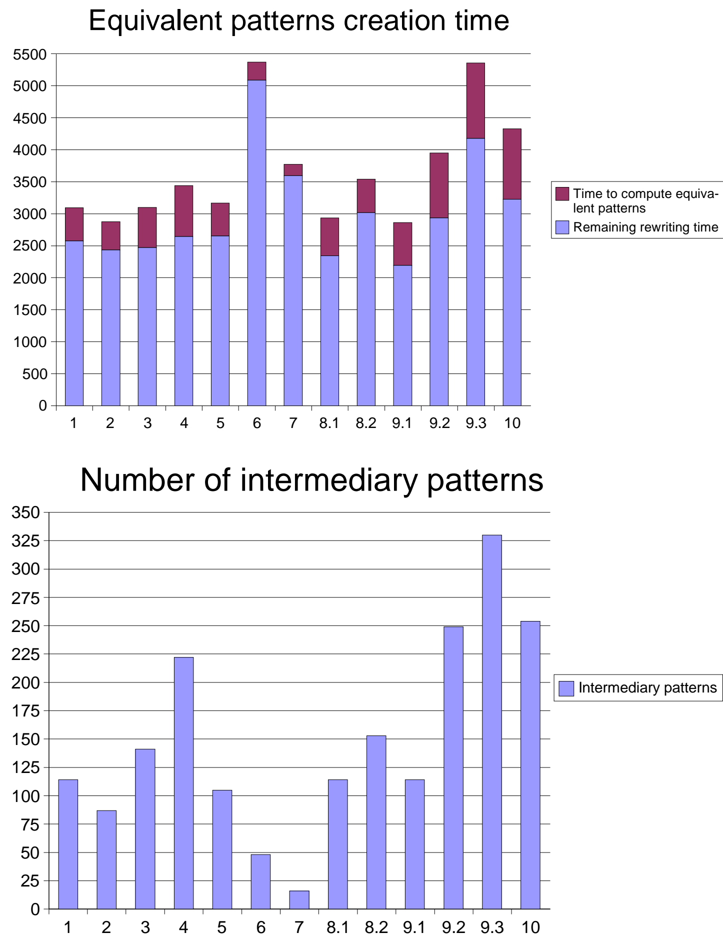



Figure 5.9: Query rewriting and the impact of equivalent pattern computation for XMark queries.

# Chapter 6

# Related works

The general approach of this thesis is similar to GMAP [109] which used a declarative language (the spj-queries) to describe the contents of storage structures in relational databases. Simillarly, at the core of our approach lies the XAM tree pattern language and we have presented a complete framework that includes algorithms for the algebraic translations of XQuery queries as well as for XAM containment and rewriting under constraints.

For each component of our framework, we discuss next the main related works. In Section 6.1 we recall the benefits of our XAM language wrt. to previous works. In Section 6.2 we compare our logical algebra with other XML algebras. In Section 6.3 we review in context our pattern extraction algorithm whereas in Section 6.4 we outline the previous containment and rewriting approaches. Finnaly, in Section 6.5 we briefly discuss some techniques of pattern minimization in the presence of structural constraints.

## 6.1 Materialized views for XML

XAMs are reminiscent of query pattern formalisms, such as the Abstract Tree Patterns [93] or of clustering strategies in object-oriented systems e.g. [20]. However, XAMs are originally focused on storage modelling, as reflected by their ID specifications, and required fields. This approach compares most directly to the Agora [80], Mars [42], LegoDB [24] and ShreX [10] projects. The XAM model departs from these previous approaches in that:

- it is based on a nested (as opposed to relational) algebraic model, which makes

it an interesting candidate for XML querying;

- it models important properties of element IDs, allowing to use and evoluate between diffrent ID labeling schemas, with different algebraic properties, possibly allowing different evalutation strategies with a strong impact on query performance;
- it provides an accurate model for XML indexes, since it allows to specify the fields whose values have to be known (that is, the index key), in order to access the index data.
- it doesn't rely on the presence of an XML schema or a DTD. Instead, we use an aposteriori schema that is extracted from the data.

## 6.2 Algebras for XML

At the logical level, many algebras have been devised in recent years ranging from tuples to nested tuples to trees [27, 33, 35, 65, 93, 100]. For the purpose of this work, we have used an algebra (Section 1.2.2) borrowed from [82] based on an effort to clarify and unify existing proposals for XML algebras. We discuss here the diferences with the other existing algebras.

**Tree algebras for XML** The TAX [65] algebra is based on a simple ordered tree model, comprising nodes and values. The notion of ID used in TAX is weaker than XQuery's notion considered here; for instance, a deep copy of an element node in TAX would have the same ID as the original node. The algebra manipulates sets of trees (with extensions to bags). TAX *selection* is similar to *nav*, applying a tree pattern on a set of input trees. TAX tree patterns are similar to ours, but do not contain optional edges. TAX selections produce trees, isomorphic to the pattern tree, whereas *nav* produces tuples. Though TAX does not explicitly use tuples, it implicitly does, since all produced trees have an"homogenous" skeleton on which variable bindings are attached. Note that the usage of such intermediary trees is incompatible with XQuery's strong notion of IDs, since a node in an intermediary tree would end up having two parents, one in the original document and another in the result tree. XML result construction is handled by grouping and updating result to align it with the desired return template (returned nodes do not have fresh identity as XQuery requires).

**YAT and Xstasy** The YAT [35] algebra was devised for unordered data, and is more limited *wrt* navigation. Xstasy [102] derives from YAT and is closer to XQuery semantics. Xstasy is based on ordered trees, endowed with XQuery-style identifiers; it uses

sets, and nested tuples internally, wrapped as trees (see comment on TAX's internal trees). Xstasy assumes available a global node ordering function, which is used by a *Sort* when ordered results are needed. The *path* operator corresponds to *nav*; it uses tree patterns with compulsory or optional edges. *path* may capture in a variable either *every* matching node, as in our algebra, or *all* nodes; our algebra achieves the latter by grouping on top of *nav*, or via nested structural joins. The *return* operator is similar to our *XMLize*; it assigns fresh identities to returned nodes, in keeping with XQuery semantics.

Improving over TAX, generalized tree patterns [33] have introduced optional edges into structural navigation patterns, and argued for the usefulness of outer structural joins. The more recent logical tree classes [93] demonstrated the interest of structure-preserving pattern matching, and introduced nest structural joins to support this.

**Tuple based algebras for XML** Many of the works originating from the database community based XQuery processing on tuple-based algebras [106]. The trend is hardly surprising: In the past, tuple-based algebras delivered great benefits to relational query processing but also provided a solid base for the processing of nested OQL data and OQL queries. Tuple-based algebras for XQuery carry over to XQuery key query processing benefits such as performing joins using set-at-a-time operations. Galax, the XQuery reference implementation use a strongly typed tuple-based algebra [106].

**Purely relational algebras for XML** MonetDB system [25] propose the fully relational evaluation of XQuery expressions. MonetDB uses relational encodings of sequences and ordered, unranked trees and compiles XQuery expressions into a relational algebra [56].

## 6.3 Extracting tree patterns from XQuery

Several previous works have considered the extraction of tree patterns from XQuery queries, mostly in order to translate the query to some algebraic representation based on tree patterns. We have also proposed such an algorithm in Chapter 3. In this section we compare our approch with pre-existing ones.

Several view-based XML query rewriting frameworks [22, 120] concentrate on XPath views, storing data for one pattern node only (since XPath queries have one return node), and lacking optional edges. Similar indexes are described in [37, 68]. In Fig. 6.1, the only XPath views are $V_1$-$V_7$, which represent the largest XPath patterns that one can derive from the query in Fig. 6.1; they store $Cont$ for all nodes which must be returned (such as the $c$, $e$ and $h$ nodes), and $ID$ for all nodes whose values

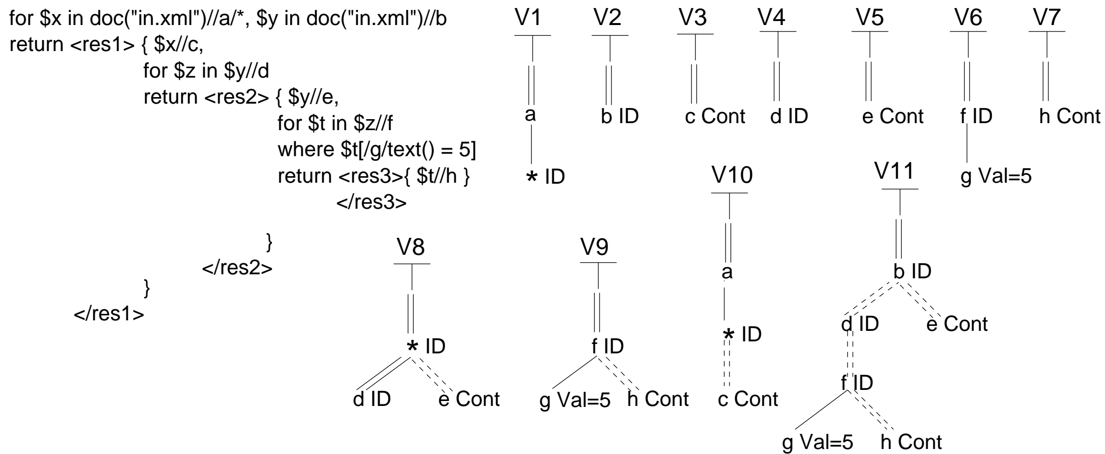



Figure 6.1: Sample XQuery query and corresponding tree patterns/views.

or content are not needed for the query, but which must be traversed by the query (such as the $a/*$, $b$, $d$ nodes etc.). In this case, the only way to answer the query is to perform five joins and a cartesian product (the latter due to the fact that $\$x$ and $\$y$ are not connected in any way) to connect the data from $V_1$-$V_7$. This approach has some disadvantages. First, it needs an important amount of computations, and second, it may lead to reading from disk more data than needed (for instance, $V_7$ contains all $h$ elements, while the query only needs those $h$ elements under //$b$//$d$//$f$).

The algorithms of [33, 93] extract patterns storing information from several nodes, and having optional edges. However, these patterns are not allowed to span across nested for-where-return expressions. In Fig. 6.1, this approach would extract the patterns $V_2, V_{10}, V_8$ and $V_9$, thus the query can be rewritten by joining the corresponding views. This still requires three joins, and may lead to read data from many elements not useful to the query.

## 6.4 Containment and rewriting under integrity constraints

The problem of query containment and rewriting has received a lot of attention because it plays a crucial role in query optimization, data integration and physical data independence.

Many works studied this problem for relation databses. In System R [104] providing an *access path selection algorithm*: whenever a disk-resident data item can be accessed in several ways, the access path selection algorithm, which is part of the query

optimizer, will identify the possible alternatives, and choose the one likely to provide the best performance for a given query.

The bucket algorithm was introduce for query rewriting in a data integration context [75, 95]. This algorithm creates for each subgoal in the query a bucket that contains all the views that are relevant to that subgoal. Then, rewritings for the whole query are created by combining views from every bucket. As we have shown in Section 5.3 bucket-style generation of rewriting cannot be used in our rewriting problem because it can miss solutions.

Containment and rewriting for semistructured queries have received significant attention in the literature [50, 85, 91], in particular under schema and other constraints [41, 42, 89, 114].

We studied pattern containment in the presence of path summary constraints. A summary limits tree depth (and guarantees finite algebraic rewriting), while a (recursive) schema does not. In practical documents, recursion is present, but not very deep [84], making summaries an interesting rewriting tool. More generally, schemas and summaries enable different (partially overlapping) sets of rewritings. Summary constraints are related to path constraints [29], and to the constraints used for query minimization in [9]. However, summaries describe *all* possible paths in the document, unlike the constraints of [9]. Our containment decision algorithm is related to the basic containment algorithm of [85], enhanced to benefit from summary constraints. (In the absence of a summary, our algorithm would use the canonical model of [85].). Moreover, the techniques employed in [85] to speed up containment test could also be applied to our setting.

Containment of nested XQueries has been studied in [45], based on a model without node identity, unlike our model.

Query rewriting based on XPath materialized views is addressed in [18, 79, 72, 120]. Our work differs in several important respects.

- The materialized views we consider include optional nodes, allowing them to closely fit the data needs of XQuery queries. For instance, consider the query:

  *for* \$*x in doc("XMark.xml")//item*
  *return* *<result>*{\$*x//keyword*}*</result>*

  The approach of [18, 79, 120] would navigate inside the view */site//item* to answer. Given this view, our approach would do the same, but our view language allows specifying a much smaller view, storing the (possibly empty) nested list

of keyword values for each item element, i.e. the query's data needs and nothing more. Observe that an XPath view */site//item//keyword* cannot be used, since ⟨*result*⟩ elements must be produced even for items lacking keywords.

- The views we consider store *nested tuples*, allowing rewriting of queries with complex return clauses. Consider the query:

  *for* $x *in doc("XMark.xml")//item*
  *return* <*result*>⟨*k*⟩{$*x//keyword*}⟨*/k*⟩
  ⟨*p*⟩{$*x//price*}⟨*/p*⟩<*/result*>

  No XPath view can fit tightly this query, whereas a view with two nested outer-join edges going from an em item node to the *keyword* and *price* nodes directly matches the query.

- Our views allow specifying interesting storage features, such as structural IDs, which increase the set of possible rewritings by allowing structural view joins. The rewritings considered in [18, 79] are limited to applying XPath navigation.

Rewriting of XQuery queries using nested XQuery views is addressed in [32, 38]. Our approach is different due to the presence of constraints (which leads to a different containment algorithm), and structural node identifiers. While using XQuery to define views seems tempting, there are some shortcomings, both noted in [38]:

- If an XQuery view builds new elements including elements from the input, the identity of the input elements is lost (element constructors have COPY semantics). XQuery-based rewritings are thus correct as long as node identity is not an issue. We explicitly model the IDs frequently present in the store, allowing their usage in the rewriting.

- Consider the XQuery materialized view :

  *for* $x *in doc("XMark.xml")//item*
  *return* <*result*>{$*x/name/text()*}
  {$*x//keyword*}<*/result*>

  One cannot answer //item//mail//keyword based on this view, because each item may have zero or more keywords both in its description and in the associated mailbox, and they are impossible to separate. In our approach, a view with two nested optional edges would allow answering this query.

## 6.5 Constraint-based tree pattern minimization

Let us compare minimization under $S$ constraints with minimization considered in previous works. Tree pattern minimization (without constraints) yields a unique minimal pattern and can be performed in polynomial time in the size of the pattern [8]. In the presence of constraints of the form "every $a$ element has a $b$ child (or descendant)", minimization remains polynomial in the size of the pattern and a unique minimal pattern exists [8]. A polynomial algorithm for minimization under child and descendant constraints is also outlined in [33], and it yields a unique solution.

A different class of constraints introduced in [33] has the form "between every $a$ element and its $c$ descendant, there must be a $b$ element". Minimization under such constraints alone leads in polynomial time to a unique solution [33]. Minimization under such constraints *and* child and descendant constraints is shown to lead, in some cases, to several equal-size patterns [33]; the minimization algorithm introduced in that work only applies contraction (erases nodes from the initial pattern). The question whether a pattern smaller than these exists is left open [33].

The differences between such minimization algorithms and the ones enabled by a summary are well illustrated by Figure 4.12. This example shows that minimization under $S$ constraints, whether by $S$-contraction or in the general case, does *not* lead to a unique solution. The question left open in [33] is answered by $t''$ in Figure 4.12: smaller patterns than can be found by contraction (as described in [33]) do exist, and there may be several such patterns of equal size.

It is also worth pointing out that summary constraints have finer granularity than child and descendant constraints at tag level, considered in [8, 33]. Consider e.g. a constraint like "between every $b$ element and its $d$ descendant, there must be a $c$ element", implied by the summary in Figure 4.12, and which could be exploited by the algorithm in [33]. If we add a $d$ child to the $b$ summary node, the constraint no longer holds, thus the algorithm of [33] cannot even find $t'_1$. However, $t'_1$ would still be an $S$-equivalent rewriting of $t$ under the constraints of the modified $S$.

A simple algorithm for minimization by $S$-contraction consists of trying to erase pattern nodes (that are not annotated with $Val\,\theta\,c$ or with $Val$, $ID$ nor $Cont$), and checking if the pattern thus obtained is still equivalent to the original one. However, performing a large number of equivalence tests may be considered expensive, making some faster techniques desirable. In [15] we present an efficient heuristic for pattern minimization which may not always find the minimal pattern, but finds a smaller one efficiently.

# Chapter 7

# Conclusion and perspectives

In this thesis we have presented an approach for achieving physical data independence in XML databases. This approach consists in using a rich tree pattern language to uniformly describe different XML storage structures, indices or materialized views and then to rewrite XQuery queries over these views.

The novelty of our work resides in several independent aspects.

- The expressive power of our view language goes beyond previous XPath-based proposals, and comes close to the needs of XQuery queries. When evaluated on a document, a materialized view yields a collection of (possibly nested) tuples, whose attributes can be simple values, element IDs, XML fragments, or other tuple collections.

  We consider views with optional and nested edges, as they correspond to the semantics of nested XQuery queries. Tuple-based views are useful because they preserve the relationships between different nodes in a document. For instance, one can store an XML element with its (possibly empty) nested collection of descendants of a given tag and its persistent identifier assigned by the system.

  Moreover, when a view stores persistent IDs for some nodes, our view language allows specifying interesting properties of the IDs, which enlarge the space of possible rewritings

- We provide an algorithm based on an algebraic framework that extracts XAM patterns from XQuery queries. The patterns we identify may span over nested XQuery blocks, enabling the query optimizer to use more complex views than in the previous approaches.

- Our work also addresses the problem of XML query rewriting and containment under path summary constraints. The path summary turns out to be a crucial instrument for efficiently finding some interesting equivalent rewritings

Future research directions include the incremental view maintenance in the presence of updates and the materialized view selection of XAMs. We briefly describe them next.

# Future works

**XAM materialized view selection** A very interesting problem on which we are curently working on is the materialized view selection problem. We are interested in devising an automatic XAM selection algorithm that starting from a query workload would propose the set of views that are optimal.

More formally, we consider a set of XQuery queries $Q_1, Q_2, \ldots, Q_n$ and a set of *weights* associated to the queries, namely $w_1, w_2, \ldots, w_n$. The goal is to find a set of XAMs to materialize in order to minimize $\Sigma_{i=1,2,\ldots,n}(w_i \times cost_{\mathcal{M}}(Q_i))$. In this formula, $cost_{\mathcal{M}}(Q_i)$ is the *cost* associated by the cost model $\mathcal{M}$ to the query $Q_i$. More precisely, let $V_1, V_2, \ldots, V_k$ be a possible set of views to materialize. We aim at minimizing $\Sigma_{i=1,2,\ldots,n}(w_i \times cost_{\mathcal{M}}(Q_i)|_{(V_1,V_2,\ldots,V_k)})$. Here, the expression $cost_{\mathcal{M}}(Q_i)|_{(V_1,V_2,\ldots,V_k)}$ denotes the cost of evaluating $Q_i$ assuming the views $V_1, V_2, \ldots, V_k$ are available.

The problem obviously depends on the choice of a cost model $\mathcal{M}$. We are exploring several candidate models that consider the computational costs associated to answering the queries $Q_i$ and the space consumption associated to materializing a view.

**Incremental XAM maintenance in the presence of updates** We are also interested in the incremental propagation of updates to XML materialized views.

Let consider an XML document $D$, and a set of materialized views described by the XAMs $X_1, X_2, \ldots, X_n$. For each view $X_i$, let $V_i$ be its stored content, $V_i = X_i(D)$.

Let $U$ be an update directive, and let $D'$ be the document obtained by applying $U$ on $D$, that is, $D' = U(D)$. Propagating the update $U$ to the materialized view $V_i = X_i(D)$ means replacing $V_i$ by $V_i' = X_i(D')$.

The obvious way of obtaining $V_i'$ is to update the document itself and compute from scratch $V_i' = X_i(D')$. However, this is computationally expensive. The alternative which we explore is to compute an *update script* $\Delta$ to be applied to $V$ in order to

directly turn it into $V_i'$: $V_i' = \Delta(V_i)$.

# References

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer Systems Science*, 1986.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[4] A. Aboulnaga, A. R. Alamendeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[5] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[6] Vincent Aguilera, Frederic Boiscuvier, Sophie Cluet, and Bruno Koechlin. Pattern tree matching for XML queries. Gemo Technical Report number 211, 2002.

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

[8] S. Amer-Yahia, S. Cho, and L. Lakshmanan et al. Minimization of tree pattern queries. In *SIGMOD*, 2001.

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

[10] S. Amer-Yahia and Y. Kotidis. Web-services architectures for efficient XML data exchange. In *ICDE*, 2004.

[11] A. Arion, V. Benzaken, and I. Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. In *XIMEP Workshop*, 2005.

[12] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *Flexible Query Answering Systems (FQAS)*, 2006.

[13] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the right storage for your XML application (demo). In *VLDB*, 2005.

[14] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed XML data. In *EDBT*, 2004.

[15] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. *WWW Journal*, 2007.

[16] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for xml queries. In *VLDB*, pages 87–98, 2007.

[17] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path summaries and path partitioning in modern XML databases. In *WWW*, pages 1077–1078, 2006.

[18] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[19] D. Barbosa, A. Barta, A. Mendelzon, and G. Mihaila. The Toronto XML engine. *WIIW Workhshop*, 2001.

[20] Véronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in $O_2$. In *4th Int'l Workshop on Persistent Objects*, pages 403–412. Morgan Kaufmann, 1990.

[21] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowl. Data Eng.*, 1(2):196–214, 1989.

[22] K. Beyer, F. Ozcan, S. Saiprasad, and B. Van der Linden. DB2/XML: designing for evolution. In *SIGMOD*, 2005.

[23] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. Legodb: Customizing relational storage for xml documents. In *VLDB*, 2002.

[24] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[25] P. Boncz, T. Grust, M.van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.

[26] Peter A. Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, K. Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane. In *EDBT*, pages 1190–1193, 2006.

[27] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. In *ICDE*, 2005.

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

[29] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. *ACM Trans. Comput. Log*, 4(4), 2003.

[30] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516, Montreal, Canada, 1996.

[31] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proceedings of the International Workshop on the Web and Databases, Houston, USA*, 2000.

[32] L. Chen and E. Rundensteiner. XCache: XQuery-based caching system. In *WebDB*, 2002.

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

[34] S. Chien, Z. Vagena, D. Zhang, and V. Tsotras. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[35] V. Christophides, S. Cluet, and J. Simeon. Semistructured and structured integration reconciled: Yat += efficient query processing. Technical report, available at http://www.rocq.inria.fr/verso/Jerome.Simeon/YAT.

[36] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an adaptive path index for XML data. In *SIGMOD*, 2002.

[37] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.

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

[39] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.

[40] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.

[41] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In *KRDB Workshop*, 2001.

[42] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[43] Yanlei Diao, Daniela Florescu, Donald Kossmann, Michael J. Carey, and Michael J. Franklin. Implementing memoization in a streaming xquery processor. In *XSym*, pages 35–50, 2004.

[44] P. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, 1982.

[45] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB*, 2004.

[46] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[47] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *VLDB*, 2003.

[48] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. In *IEEE Data Engineering Bulletin*, 1999.

[49] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. In *Proc. of the Int. WWW Conf.*, pages 119–135, 2000.

[50] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, 1998.

[51] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. In *Technical Report, INRIA*, volume 3680, 1999.

[52] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, Athens, Greece, 1997.

[53] R. Goldman and J. Widom. Approximate DataGuides. Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, 1999.

[54] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS*, 2003.

[55] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *TODS*, 29, 2004.

[56] Torsten Grust and Jens Teubner. Relational algebra: Mother tongue - xquery: Fluent. In *TDM*, pages 9–16, 2004.

[57] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents, 2003.

[58] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[59] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode XML query processing. In *VLDB*, 2003.

[60] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs, 2003. In ICDE.

[61] *IN*itiative for the *E*valuation of *X*ML retrieval. inex.is.informatik.uni-duisburg.de:2004, 2004.

[62] Information technology - database languages - SQL part 14: XML-related specifications (SQL/XML). 2004.

[63] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB J.*, 11(4), 2002.

[64] H. V. Jagadish, Shurug Al-Khalifa, Laks Lakshmanan, Andrew Nierman, Stylianos Paparizos, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. Timber: A native XML database. In *SIGMOD*, page 672, 2003.

[65] H. V. Jagadish, L. V. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *DBPL*, 2001.

[66] H. Jiang, H. Lu, W. Wang, and J. Xu. XParent: An efficient RDBMS-based XML database system. In *ICDE*, 2002.

[67] C. C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, 2000.

[68] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[69] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[70] Alfons Kemper and Guido Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[71] D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *ICDE*, pages 313–320, 2001.

[72] L. Lakshmanan, H. Wang, and Z. Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[73] M. Lee, H. Li, W. Hsu, and B. Ooi. A statistical approach for XML query size estimation. In *DataX workshop*, 2004.

[74] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[75] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[76] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[77] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[78] Library Of Congress, http://www.loc.gov.

[79] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[80] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.

[81] I. Manolescu, D. Florescu, D. Kossmann, D. Olteanu, and F. Xhumari. Agora: Living with XML and relational. In *VLDB*, 2000. Software demonstration.

[82] I. Manolescu and Y. Papakonstantinou. An unified tuple-based algebra for XQuery. Available at www-rocq.inria.fr/˜manolesc/PAPERS/algebra.pdf.

[83] Ioana Manolescu, Andrei Arion, Angela Bonifati, and Andrea Pugliese. Path sequence-based XML query processing. In *BDA*, 2004. (Informal proceedings).

[84] L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *Proc. of the Int. WWW Conf.*, 2003.

[85] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[86] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[87] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.

[88] S. Nestorov, J. D. Ullman, J. L. Wiener, and S. S. Chawathe. Representative objects: Concise represenations of semistructured,hierarchical data. In *ICDE*, 1997.

[89] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[90] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

[91] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, 1999.

[92] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, Taipei, Taiwan, 1995.

[93] S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, 2004.

[94] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for XML twigs. In *ICDE*, 2004.

[95] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[96] The Qexo system. http://www.gnu.org/software/qexo/.

[97] The QizX system. www.xfra.net/qizxopen/.

[98] C. Qun, A. Lim, and K. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[99] Praveen Rao and Bongki Moon. PRIX: Indexing And Querying XML Using Prufer Sequences, In ICDE, 2004.

[100] C. Ré, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.

[101] A. Sahuguet. The Kweelt system. http://sourceforge.net/projects/kweelt.

[102] Carlo Sartiani. A query algebra for xml p2p databases. In *EDBT Workshops*, pages 637–648, 2006.

[103] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Relational storage and retrieval of XML documents. In *WebDB*, 2000.

[104] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in relational database systems. In *SIGMOD*, Boston, MA, USA, 1979.

[105] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[106] J. Simeon. The Galax system. http://www.galaxquery.org.

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

[108] J. Teubner, T. Grust, and M. van Keulen. Bridging the GAP between relational and native XML storage with staircase join. In *VLDB*, 2003.

[109] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *VLDB*, pages 367–378, Santiago, Chile, 1994.

[110] ULoad web site. gemo.futurs.inria.fr/projects/XAM.

[111] A. Vyas, M. Fernández, and J. Siméon. The simplest XML storage manager ever. In *XIME-P*, 2004.

[112] H. Wang and X. Meng. The Sequencing of Tree Structures for XML Indexing, 2005.

[113] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[114] P. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

[115] The XMark benchmark. www.xml-benchmark.org, 2002.

[116] XML Specification. http://www.w3.org/XML/, 1998.

[117] Document Object Model. http://www.w3.org/DOM/.

[118] XQuery 1.0 formal semantics. www.w3.org/TR/2004/WD-xquery-semantics.

[119] XML Query Data Model. http://www.w3.org/TR/query-datamodel, 2005.

[120] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[121] The XQuery 1.0 language. www.w3.org/XML/Query.

[122] M. Yoshikawa, T. Amagasa, T. Uemura, and S. Shimura. XRel: A path-based approach to storage and retrieval of XML documents using RDBMSs. *ACM TOIT*, 2001.

[123] Ning Zhang, Varun Krasholia, and Tamer Oszu. A Succint Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE*, 2004.