Processing XML in Database Systems
Schmidt, A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

XML Data Models and Physical Storage

In this chapter, we go through some of the formal aspects of XML documents; in particular, we look at how XML documents can be stored in database systems and what impact storage has on the way the documents are accessed. We first motivate that different views of XML documents help to make processing transparent depending on the application scenario. For example, if a programmer accesses documents primarily through a parse-tree representation then a tree-based model like the DOM mentioned in previous chapters probably fits his needs best. We then discuss a particular data model, Monet XML, in detail and compare it to other approaches.

3.1 Introduction

XML has penetrated virtually all areas of Internet-related application programming and become the most widely used data exchange framework in a plethora of application areas. It assumes the role of the de facto standard data exchange format in Web database environments. This means that XML documents have to be accessed in a variety of contexts each with its own requirements and idiosyncrasies. On the database side, modelling issues arise from the discrepancy between semi-structured data on the one hand side and fully structured database schemas on the other side. Database researchers have provided valuable insights to bring these two areas together. The solutions proposed include not only XML domain specific developments but also techniques that build on object-oriented and relational database technology.

To make XML the language of all Web databases, performance issues are a crucial point when mission critical solutions are to be deployed for e-business applications. This means that database support for XML processing can only find the widespread use that researchers and businesses anticipate if, on the one hand, storage and retrieval of documents and their contents satisfy the demands of impatient surfers as well as, on the other hand, live up to data interchange challenges and query intensive processing. It is therefore necessary to cross the border between established standards on the declarative and conceptual level like the DOM [W3C98b] and efficient query execution strategies, which may require data structures on the physical level that deviate greatly

\[Parts\ of\ this\ chapter\ were\ published\ in\ [SKWW00a, SKWW00b]\]
from those on the declarative level. Starting from the syntax tree representation of a document, this chapter presents data models that are based on various interpretations of the document tree. For a particular model, Monet XML, we discuss a number of technical issues in detail, while we only have the space for an overview and categorisation of other approaches. All models we present have their inherent advantages and disadvantages. If, for example, a particular model is geared towards efficient navigation through the document tree, then certain types of associative queries may perform badly because they degenerate into random access patterns with high data volume, for which this particular architecture may not be optimal. In this example, the converse also holds: associative mappings, which are aimed at efficient bulk retrieval, tend to bring about rather high document reconstruction costs and thus make navigational access expensive.

Performance however is only one aspect of processing: implementation effort often is as important. Many mappings of XML documents to relational databases are relatively easy to implement on top of existing systems. Additionally, extensible query languages like SQL already exist in these systems and can be adopted for querying the database instances which the mappings create; this way it is not necessary to build a new query processor from scratch. User Defined Functions (UDFs) can help to bridge the gap between the relational model and XML. This way, query results can be converted from tables to documents, thus integrating RDBMSs neatly into the XML world without bringing about the costs of researching and designing an all-XML solution from top to bottom.

To sum up, when an XML solution is to be implemented, there are several aspects to be considered with regards to the data model: Overall performance, when critical, may call for special purpose solutions which require high engineering efforts as long as there are no universally accepted standard application scenarios. The fact however, that XML technology makes communication in and between software systems more robust may make up for the additional costs it causes in one particular component. Implementation effort can vary greatly depending on whether a system is built from scratch or whether an already existing storage engine is extended. Additional aspects concerning the cost of XML systems will be covered in Chapter 5 which presents the XMark benchmark for XML processing systems.

### 3.2 Views of XML Documents

In this section we present the most common views of XML documents. They usually bring about a particular implementation strategy with regards to storage and query processing. In the sequel, we discuss the following three types of models: **associative models**, also known as relational models, define the document structure by the properties of each component and possibly group the components physically by their properties; in **object-based** models the basic entities are the nodes in the parse-tree representation of documents, whereas **native models** emphasise the textual nature of XML. However, we start our overview with the most basic kind of model, which does not rely on database technology.

#### 3.2.1 File System-Like Models

The simplest type of model is file system-like storage, *i.e.*, the main entity is a complete document; internal structure does not play a role. These models may either be imple-
mented on top of real file systems, such as one the filesystems available on UNIX, or inside databases where documents are stored as Binary Large Objects (BLOBs). However, query languages in these models usually do not have access to the internal structure of a document without first constructing a parse tree; this results in two operations, store and recall, which can be supported very efficiently – at the cost however that other operations which require access to the internal structure of documents may become prohibitively expensive. For example, consider a database that consists of documents like those of Figure 2.2; then, selecting documents by keywords that occur enclosed in <author> tags not only requires a scan over the whole database but also the construction of the parse trees of possibly every single document.

### 3.2.2 Associative Models

The basic idea of these models is to associate a set of properties with every node in the syntax tree. The individual properties vary from model to model; however, the set of possible properties is usually limited. This makes it feasible to cluster properties of the same kind and thus optimise access to syntax nodes by their properties. For example, a typical associative query is ‘return the OIDs of all nodes that carry <author> tags.’

There are sensible associations other than a node’s tag name. As indicated in Figure 3.1 the tree structure of documents induces certain properties like parent of or child of, which will play an important role later when we discuss Monet XML in detail: every node has an ancestor, even the root element has an ancestor in the sense that it belongs to a document. Another important property is the node’s rank, i.e., a number representing the node’s absolute position in a document or the position relative to its siblings. But we can also define document-dependent properties, e.g., whether a node carries a document-provided ID or to what namespace the node belongs. In general, if a document is accompanied by a DTD, required attributes and children make up for the structured part of the document. However, the detailed lay-out of the properties depends on the actual mapping scheme used for a database instance; in many cases it is influenced by the characteristics of the underlying storage engine and the application scenario.

### 3.2.3 Object-Based Models

*Object-based models* offer a particularly intuitive view on XML documents, especially if the primary access method is navigational access along the parse tree structure. A typical interface was presented in Figure 2.7 in the previous section. The many parse
trees the reader finds in this thesis are all representative for the object-oriented view on XML documents.

There are many flavours of object-oriented models in both the document processing and database community. While the aforementioned DOM specification [W3C98a] is an example of an interface for generic markup languages, [FE01] present a way to access XML data in ODMG and OQL scenarios [Obj02]. [Ren01] makes use of schema information and proposes a way to generate a set of Java class definitions which are then used to access a syntax tree; thus, they open up document semantics to the user of the Java programming language. Object-based models have a rich heritage especially in office document processing. For example, [BRT88] present a data model and algebra to bring as much flexibility as possible to office automation systems, a common theme in document processing (see [BRG88, BR85] and [AH86] for a more theoretical presentation).

### 3.2.4 Native Models

With *native mappings* we describe physical data models that do not fit well into the existing methodologies for describing data deployed in database management systems. Very often, this is because these mappings value the *textual* aspects of XML more than the structural parts.

Most native models draw a great deal of ideas from object-oriented and relational technology. Unfortunately, complete descriptions of the inner workings of most native repositories are not made publicly available and are considered trade secrets; the research community can only make educated guesses from the performance characteristics and physical requirements of the systems. We therefore think that it is sensible to choose a representative example for which a detailed description is available and give a brief overview. In the following, we give a brief outline of the storage engine of the *Natifix* system; an overview of the complete system architecture is given in [KM00a], and a more detailed presentation is available in [KM99].

While the higher-level query languages of *Natifix* conform to the standards presented in the previous chapters, the storage engine features novel ideas which cannot be subsumed by the concepts that we already introduced. The physical storage is page-oriented. Documents are distributed over pages according to their size and additional user- or application-provided fragmentation criteria. Each page contains one or more
(partial) document trees. If a document is to be loaded into the database and does not fit onto a single page, it is split at an edge in the parse tree, turning it into two linked sub-documents. If the database administrator knows the query profile in advance, he can specify additional fragmentation criteria before the bulkload, so that less run-time fragmentation is necessary than would normally occur if the current data lay-out does not match the query profile: If a query binds variables to parts of the documents whose structure is not accessible then the relevant sub-documents are parsed and fragmented at query execution time. Furthermore, the flexible fragmentation criteria allow for domain-specific fine-grained locking and concurrency control mechanisms. The adaptive and hybrid fragmentation strategy, which in principle works without human intervention or preparation, is what made its authors call NATIX a ‘native’ system.

### 3.3 Monet XML

We now discuss a storage mapping that does a complete fragmentation and clustering of the input document. It is named ‘Monet XML’ after our home-grown database engine and primary experimentation platform Monet. The main idea behind the mapping is to break up the parse tree into binary associations; thus it qualifies as an associative model. We will show that this way, associations that are frequently accessed in many queries like parent-child relationships, attributes, or topological orders can be intuitively described, space efficiently stored and effectively queried. In contrast to general graph databases which were extended with XML functionality like Lore [AQM+97], we exploit the basic tree structure of XML elements and incorporate information about the association’s position within the syntax tree relative to the root into our data model. References such as IDREFs that escape the tree structure are taken care of by views on the tree structure. Associations that provide semantically related information are stored together and thus clustered in the binary relations of the database. Along with the decomposition schema we also present a method to translate queries formulated on paths of the syntax tree into expressions of an algebra for vertically fragmented schemas, which is described in detail in [BK99], which is also the primary reference for the Monet Database Kernel.

The approach used in Monet XML is distinguished by two features. Firstly, the decomposition method is independent of the presence of DTDs, but rather explores the structure of the document at parse time. Additionally, schema information is gathered automatically and made available after the decomposition is stored in a database. Secondly, Monet XML tries to minimise the volume of data irrelevant to a query that has to be processed during querying. Storing associations according to their context in the syntax tree provides tables that contain semantically closely related information. As a result, data relevant to a given query can be accessed directly as a separate table, avoiding large and expensive scans over irrelevant data and thus lowering the costs of associative queries with path expressions. Especially the need for hierarchical projections and semijoins vanishes completely.

The high degree of fragmentation introduced by the mapping might fuel reservations since it is likely to incur increased efforts to reconstruct the original document, or parts of it. This is true to the degree that the Monet XML mapping favours associative queries with few predicates and little output volume. During the implementation and evaluation of the mapping, various techniques have been explored to accelerate queries involving reconstruction of (large) parts of the original document, which indeed were disappointing in performance at first. This was successful in the sense that we were able
to improve greatly on the original, naïve algorithms we prototyped [SKWW00a]. Nevertheless, as discussed in Chapter 5, the reconstruction performance of Monet XML is inherently inferior to that of mapping schemes designed with competitive reconstruction behaviour in mind. In general however, as the little quantitative assessment that concludes this section shows, the number of additional joins to recover relationships along which the document was fragmented is fully made up for as they usually involve only little data volume, which indeed was the original design goal.

3.4 Data Model and Algebra

The most common view on raw XML documents, i.e., we don’t have domain knowledge about the content, is that through syntax trees. With string and int denoting sets of character strings and integers and oid being the set of unique object identifiers, we can define an XML document formally:

**Definition 3.** An XML document \( d = (V, E, r, \text{label}_E, \text{label}_A, \text{rank}) \) is a rooted tree with nodes \( V \) and edges \( E \subseteq V \times V \) and a distinguished node \( r \in V \), the root node. The function \( \text{label}_E : V \rightarrow \text{string} \) assigns labels to nodes, i.e., elements; \( \text{label}_A : V \rightarrow \text{string} \rightarrow \text{string} \) assigns pairs of strings, attributes and their values, to nodes. Character Data (CDATA) are modelled as a special ‘string’ attribute of \( \text{CDATA} \) nodes, \( \text{rank} : V \rightarrow \text{int} \) establishes a ranking between sibling nodes. For elements without any attributes \( \text{label}_A \) maps to the empty set.

In comparison to Definition 1 this definition is geared towards physical data models and query execution by, for example, making rank information explicit in the model itself. Figure 2.1 showed an XML document which describes a fragment of a bibliography; the corresponding syntax tree for the first two entries is displayed in Figure 3.4. The representation is largely self-explanatory, the \( o_i \) denote object identifiers (OIDs) whose assignment is arbitrary, e.g., depth-first traversal order as in the figure. We apply the common simplification not to differentiate between PCDATA and CDATA, nor do we take rich datatypes into account. In contrast to many object-based models, we assign OIDs only to nodes representing XML elements and not to those representing XML attributes.

There are other data models such as [ABS99] where XML syntax trees are laid out differently: whereas our data model is a node-centric model, i.e., \( \text{label}_E \) and \( \text{label}_A \) are functions with domain \( V \), [ABS99] chose to centre their definitions around the edge set \( E \). We deviated from this view for a number of reasons. Firstly, using \( V \) as the primary point of reference enables smoother transitions between the object-based and associative view of documents; this, secondly, again results in cleaner code when the concepts are implemented and different views have to be supported. Lastly, it makes it more natural to define additional functions with semantics like ‘this node carries an attribute of type \( x \)’, which are important in both associative and object-based data mappings.

3.4.1 Preliminaries

Before we discuss techniques how to store a syntax graph as a database instance, we introduce the concepts of associations and path summaries. They identify spots of interest and constitute the basis for the Monet XML Model. In the sequel, a centred
dot (·) serves as a placeholder for an element of a union or sum type that can be derived from the context.

**Definition 4.** A pair $A(o, ·) \in \text{oid} \times (\text{oid} \cup \text{int} \cup \text{string})$ is called an *association*. $A$ is also called the *type* of the association and can be thought of as a label attached to $(o, ·)$.

The different data types used for associations describe different parts of the tree: associations of type $\text{oid} \times \text{oid}$ represent edges, i.e., parent-child relationships. Attribute
values (including character data, which are represented by vertices with label 'string', that start from 'cdata' labelled nodes – this is possible since cdata nodes by definition can’t carry other attributes) are modelled by associations of type oid × string, while associations of type oid × int are used to preserve the topology of a document in the database, i.e., the rank information.

**Definition 5.** A *path* in the syntax tree is a sequence of vertex or edge labels starting at the root. For a node o in the syntax tree, we use path(o) to denote the path to o.

As an example, consider the node with OID o₃ in Figure 3.4; its path is bibliography/article/author. The corresponding character data string “Ben Bit” has path bibliography/article/author/cdata[string], where ‘a/b’ denotes that a is the parent of b and a[c] denotes that c is an attribute of a. Note that there is one special attribute, the rank, i.e., the relative or absolute position of a syntax node in the textual representation of the document; we write p[rank] to denote the path describing the rank, which is a numerical value. Ranks were left out in Figure 3.4 to keep the presentation smooth and since they are implicit between sibling nodes; for example, node o₁₂ has rank bibliography/article/author/rank = 3 assuming that the first sibling o₈ carries the rank 1.

So paths describe the schematic position of the element in the graph relative to the root node; we also call path(o) the type of the association ⟨, o⟩. The set of all paths in a document is called the document’s *path summary*. Note that there is a mapping from the set of OIDs to the path summary that determines the type of each association. Therefore we can write ⟨, o⟩ as an abbreviation for A⟨, o⟩ since A = path(o), i.e., A is implied. Later we need some tools to manipulate paths: if p is a path and e is an element name then p/e denotes the concatenation of p and e; likewise, if a is an attribute name then p[a] is the concatenation of p and a. We can also define functions that usually operate on lists for paths: if p = l₁/.../lₙ₋₁/lₙ is a path, then last(p) = lₙ and second-last(p) = lₙ₋₁, no matter whether lₙ is an attribute or an element. Note that only the last member of a path last(p) can be an attribute; all other members are elements since XML attributes are strings without structure or sub-elements.

**Definition 6.** A *regular path expression* is a path expression extended with the symbol ‘/’ which denotes not only a parent-child relationship (like ‘/’) but the more general ancestor relationship.

For example, in Figure 3.4 the regular path expression bibliography/cdata denotes the set of all cdata nodes, which is {o₄, o₆, o₉, o₁₁, o₁₃, o₁₅}.

### 3.4.2 The Monet XML Model

As we pointed out in the beginning, the question central to querying XML documents is how to store the syntax tree as a database instance that provides efficient retrieval capabilities. Given Definition 3 the tree could be stored using a single database table for the parent-child relations (similar to the work [vZAW99]), another one for the elements’ labels and so on. Though space efficient, such a decomposition makes querying expensive by enforcing scans over large amounts of data irrelevant to a path expression being evaluated, since structurally unrelated data are possibly stored in the same tables. Even if the query consists of a few joins only, large data volumes may have to be processed; [FK99] present an extended discussion of storage schemes of this kind.
Definition 7. Given an XML document \( d \), the Monet transform is a quadruple \( M_t(d) = (E, A, T) \) where:

- \( E \) is the set of binary relations that contain all associations between nodes;
- \( A \) is the set of binary relations that contain all associations between nodes and their attribute values, including character data;
- \( T \) is the set of binary relations that contains all associations of the same type in the same binary relation. A relation that contains the tuple \( (\cdot, \cdot) \) is named \( path(\cdot) \), and conversely, the information that a tuple contains is stored in exactly one relation so that no information is duplicated. This idea results in the following definition:

We pursue a rather different approach using the structures defined above, i.e., storing all associations of the same type in the same binary relation. A relation that contains the tuple \( (\cdot, \cdot) \) is named \( path(\cdot) \), and conversely, the information that a tuple contains is stored in exactly one relation so that no information is duplicated. This idea results in the following definition:

\[
\begin{align*}
\text{bibliography/article} &= \{(01, 02), (01, 07)\}, \\
\text{bibliography/article/author} &= \{(02, 03), (07, 010), (07, 012)\}, \\
\text{bibliography/article/author/cdata} &= \{(03, 04), (010, 011), (012, 013)\}, \\
\text{bibliography/article/author/cdata[string]} &= \{(04, "Ben Bit"), (011, "Bob Byte"), (013, "Ken Key")\}, \\
\text{bibliography/article/title} &= \{(02, 05), (07, 014)\}, \\
\text{bibliography/article/title/cdata} &= \{(05, 06), (014, 015)\}, \\
\text{bibliography/article/title/cdata[string]} &= \{(06, "How to Hack"), (015, "Hacking & RSI")\}, \\
\text{bibliography/article/editor} &= \{(07, 08)\}, \\
\text{bibliography/article/editor/cdata} &= \{(08, 09)\}, \\
\text{bibliography/article/editor/cdata[string]} &= \{(09, "Ed Ito")\}, \\
\text{bibliography/article[key]} &= \{(02, "BB88"), (07, "BK99")\}
\end{align*}
\]
XM L Data Models and Physical Storage

4 2 2

Data Models and Physical Storage

TT is the set of binary relations that contain all pairs of nodes and their rank;

r remains the root of the document.

Encoding the path to a component into the name of the relation achieves a significantly higher degree of semantic fragmentation for non-trivially structured documents than would be implied by a strategy that follows the fragmentation criteria of plain data guides as implemented in the Lore system and proposed by [GW97]. In other words, we use path to group semantically related associations into the same relation. As a direct consequence of the decomposition schema, we do not need to introduce novel features on the storage level to cope with irregularities induced by the semi-structured nature of XML, which are typically taken care of by NULLs or overflow tables [DFS99]. Moreover, it should be noted, that the complete decomposition can be implemented so that it is linear in the size of the document with respect to time complexity of the algorithm. Concerning memory requirements, it is in O(h), h being the maximum depth of the syntax tree, in addition to the space the binary relations in the database storage engine occupy, i.e., it is not necessary to materialise the complete syntax tree. Table 3.4 shows the Monet transform of the example document. A more detailed implementation-centric discussion of the mapping is presented in the following chapter. The following proposition shows that the Monet mapping does not drop information that is present in the source document.

Proposition 8. The above mapping is lossless, i.e., for an XML document d there exists an inverse mapping $M'_{-1}$ such that $d$ and $M'_{-1}(M_t(d))$ are isomorphic.

We present a straightforward proof of this proposition by giving the definitions of $M'_{-1}$ and $M_t$.

Constructive Proof of Proposition 8. Definition 7 introduces the Monet transform $M_t(d) = (r, E, A, T)$ of a document d. For a document d the sets E, A and T are computed as follows:

for elements:

$$E = \bigcup_{(o_1, o_2) \in X_E} \text{path}(o_1) / s(o_1, o_2),$$

for attributes (including text):

$$A = \bigcup_{(o_1, s_1, s_2) \in \text{label}_A} \text{path}(o_1) / [s_1](o_1, s_2),$$

for ranking integers:

$$T = \bigcup_{(o_1, o_2) \in \text{rank}} \text{path}(o_1) / [\text{rank}](o_1, o_2),$$

where E and \text{label}_E are combined into one set

$$X_E = \{(o_1, o_2, \text{label}_E(o_2))(o_1, o_2) \in E\},$$

\text{label}_A is interpreted as a set $\subseteq \text{oid} \times \text{string} \times \text{string}$ as well as $\text{rank} \subseteq \text{oid} \times \text{int}$.

To see that this mapping is lossless we give the inverse mapping. Given an instance of the Monet XML model $M_t(d)$ we can reconstruct the original rooted tree $d = (V, E, r, \text{label}_E, \text{label}_A, \text{rank})$ in the following way (R denotes the set of all relations in a database instance):


3.4 Data Model and Algebra

1. \( V = \{ o_i | (\exists R \in R)(\exists o_j \in \text{oid}) : R(o_i, o_j) \} \),

2. \( E = \{ (o_i, o_j) | (\exists R \in R) : R(o_i, o_j) \} \),

3. \( r' = r \),

4. \( \text{label}_E = \{ (o_i, o_j) | (\exists R \in R) (\exists o_j \in \text{oid}) (\exists s \in \text{string}) : R(o_i, o_j) \land \text{second-last}(R) = s \} \),

5. \( \text{label}_A = \{ (o_i, s_1, s_2) | (\exists A \in A) : A(o_i, s_2) \land \text{last}(A) = s_1 \} \),

6. \( \text{rank} = \{ (o, i) | (\exists T \in T) : T(o, i) \} \).

This concludes the proof.

3.4.3 Object-Based Perspective

In addition to the relational perspective we adhered to so far, the Monet transform also enables an object-based perspective, where an object is seen as a node in the syntax tree, which is often more intuitive to the user and is adopted by standards like the DOM [W3C98b]. Particularly in high-level query languages, approaches that bear strong similarities with object-based techniques have emerged. Given the Monet transform, we have the necessary tools at hand to logically reconcile the relational perspective with the object-based view.

It is natural to re-assemble an object with OID \( o \) from those associations whose first component is \( o \): e.g., the node with OID \( o_2 \) is easily converted into \( \text{object}(o_2) = \{ \text{key}(o_2, "BB88"), \text{author}(o_2, o_3), \text{title}(o_2, o_5) \} \), an instance of a suitably defined class \( \text{article} \) with members \( \text{key}, \text{author} \) and \( \text{title} \). However, XML is regarded as an incarnation of the semi-structured paradigm. One consequence of this is that we cannot expect all instances of one type to share the same structure. In the example, the second publication has an \( \text{editor} \) element whereas the first does not. We therefore distinguish between two kinds of associations: \( \text{(strong)} \) associations and \( \text{weak} \) associations. Strong associations constitute the structured part of XML – they are present in every instance of a type; weak associations account for the semi-structured part: they may or may not appear in a given instance. Objects \( o_2 \) and \( o_7 \) reflect this: \( o_7 \) has an \( \text{editor} \) member whereas \( o_2 \) has not. So \( \text{editor} \) is a weak association of \( \text{article} \) objects. We can now define the following:

**Definition 9.** An \( \text{object} \) corresponding to a node \( o \) in the syntax tree is a set of strong and weak associations \( \{ A_1(o, o_1), A_2(o, o_2), \ldots \} \).

In the presence of a DTD, strong and weak associations can be defined slightly differently by looking at the right-hand side of an element declaration: Kleene stars, Kleene pluses, question marks and alternatives can be seen as indicators that the respective element is a weak association. We again note that although it is possible to navigate through a document in an object-based manner, Monet XML does not support this well. For example, if we want to reconstruct an object \( o \) of type \( \text{path}(o) \) we have to check all strong and weak associations of \( o \) for the presence of an attribute of \( o \), thus, being required to do much more work than necessary in an object-based system which provides a data type association list for attribute storage; of course, it is difficult to optimise such a data type for associative retrieval. The next question we address directly arises from the modelling of objects: How can we re-formulate queries from an object-based setting to queries in relational Monet XML?
3.4.4 Execution Model and Algebra

The unified view provided by the Monet XML model extends directly to querying. For the relational layer, a multitude of operators implementing the relational algebra, including specialties intrinsic to vertical fragmented schemas, have been proposed in the literature. Hence, we omit a discussion of technical issues concerning bare, relational query processing in the context of vertical fragmentation and refer the interested reader to [BK99] for a comprehensive overview.

More interesting is the actual translation of an OQL-like query to match the facilities of the underlying query execution engine. We only outline the translation by an example query. The process bears strong resemblance to mapping techniques developed to implement object-based query interfaces on relational databases; thus, we can resort to the wealth of techniques developed in that field. See [BC00] for a comparative analysis of different query languages for XML.

Consider the following query which selects those of Ben Bit’s publications whose titles contain the word ‘Hack’; the semantics of the statements are similar to [BT99]:

```sql
select p
from bibliography/article p,
p/author/cdata[string] a,
p/title/cdata[string] t
where a = "Ben Bit"
and t like "Hack";
```

The query consists of two blocks, a specification of the variables involved, which translates to computing the proper binary relations, and constraints that filter the tuples of interest. For resolving path expressions, we need to distinguish two types of variables in the from clause: variables that specify sets, \( p \) in the example represents a set of OIDs, and variables, which specify associations (here of type \( \text{oid} \times \text{string} \)), \( a \) and \( t \).

We collapse each path expression that is not available in the database by joining the binary relations along the path specification. This establishes an association between the first and last element of the path. Finally, we take the intersection of the elements specified. Matching the variables against the running example, the from clause specifies the following elements:

\[
p = \{o2, o7\},
\]

\[
\text{assoc}(p/a) = \{(o2, "Ben Bit"), (o7, "Bob Byte"),
\]

\[
(o7, "Ken Key"))
\]

\[
\text{assoc}(p/t) = \{(o2, "How To Hack"),
\]

\[
(o7, "Hacking & RSI")
\]

Queries containing regular path expressions, i.e., paths with wildcards, directly benefit from the availability of the path summary. Standard methods for the evaluation of regular expressions can be applied to the path summary and enable the immediate determination of the candidate relations. We will cover the evaluation of regular path expressions in the following chapter in greater detail.

The evaluation of the where clause is not of particular interest in this context. Though processing of binary tables differs from the conventional relational model in several aspects, but these differences have no direct impact on our method; we again refer to [BK99] for details.
3.4.5 Optimisation with DTDs

As XML documents are not required to conform to DTDs we generally do not assume that they do. However, in this section we show that our data model is flexible enough to take advantage of additional domain-knowledge in the form of DTDs or XML Schema specifications. Again, the first-class paths in Monet XML are the focal feature necessary for a seamless integration. We present an approach that is similar in spirit to [STH+99].

For motivation, consider again the example document in Figure 2.1 and 3.4, and also suppose we are given the DTD in Figure 3.5. The DTD *inter alia* says that each publication may only have a single title element. Given this rule, we can collapse each path from the publication nodes to the character data of title elements without losing information; thus,

{ bibliography/article/title[(o₂, o₅), (o₇, o₁₄)],
  bibliography/article/title/cdata[(o₅, o₆), (o₁₄, o₁₅)],
  bibliography/article/title/cdata[string][(o₆, "How to Hack"),
  (o₁₅, "Hacking & RSI")]
}

may be reduced to

{ bibliography/article/title[(o₂, o₅), (o₇, o₁₄)],
  bibliography/article/title[string][(o₅, "How to Hack"),
  (o₁₄, "Hacking & RSI")].

That is, we take advantage of DTDs by identifying and subsequently collapsing 1:1 relationships to reduce storage requirements and the number of joins in query processing. The result of hierarchically joining the associations takes the place of the original data. Some of these 1:1 relationships can be inferred from a DTD, others require domain-specific knowledge: our common sense knowledge of bibliographies tells us that in bibliographies the only elements whose order is important are author and editor elements. Thus, we may, on the one hand, drop all rank relations that do not belong to author or editor tags and furthermore reduce the before mentioned path to:

{ bibliography/article/title[(o₂, “How to Hack”), (o₇, “Hacking & RSI”)] }.

The detailed algorithm now looks as follows: We go through the path summary and, for every path \( p_1 / \ldots / p_n / l/r, n \geq 0 \) with children, and the corresponding rule \( l \rightarrow R \), where \( r \) occurs in \( R \), in the DTD we do the following:
Table 3.1: Q1 through Q10 in [SYU99]

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>/PLAY</td>
</tr>
<tr>
<td>Q2</td>
<td>/PLAY/ACT</td>
</tr>
<tr>
<td>Q3</td>
<td>/PLAY/ACT[index()=2]</td>
</tr>
<tr>
<td>Q4</td>
<td>/PLAY/ACT[-3]</td>
</tr>
<tr>
<td>Q5</td>
<td>/PLAY/ACT/TITLE</td>
</tr>
<tr>
<td>Q6</td>
<td>//SCENE/TITLE</td>
</tr>
<tr>
<td>Q7</td>
<td>/PLAY/ACT/TITLE</td>
</tr>
<tr>
<td>Q8</td>
<td>//ACT/TITLE</td>
</tr>
<tr>
<td>Q9</td>
<td>/PLAY/ACT/SCENE/SPEECH[SPEAKER='CURIO']</td>
</tr>
<tr>
<td>Q10</td>
<td>//ACT/[SPEECH/SPEAKER='CURIO']</td>
</tr>
</tbody>
</table>

1. If $R$ contains alternatives, Kleene stars or Kleene pluses then no optimisation is possible because ranking information would (possibly) get lost.

2. Otherwise there is redundancy, and we can materialise the joins of $p_1/\ldots/p_n/l$ with all the children of $p_1/\ldots/p_n/l/r$, mark $p_1/\ldots/p_n/l/r$ as virtual and remove from the database $p_1/\ldots/p_n/l/r$ and its children.

Note that we apply this optimisation technique not to the DTDs themselves, although this would be possible too, to derive a storage schema but rather simplify the paths present in the actual document instance. We do not derive a storage schema from the DTDs.

Coming back to the issue of implementation effort we addressed in the previous chapter: it should be assessed whether these kinds of optimisations justify the increased complexity in the query processor. Depending on the input documents we observed gains in query execution speeds between 2% and 25% for the queries listed in the following section and, naturally, decreases in the database sizes. However the complexity and therefore maintainability of the path evaluation engine increased considerably.

A final remark on the effect of fragmentation with a 'real world' DTD. Figure A.1 in the Appendix shows the graph structure implied by the element relationships of the DTD of the XMark benchmark document (see Chapter 5 and [SKF'00]). The effect of straightening out the graph structure into a tree structure by making use of the path summary of the database instance in displayed in Figure A.2 in the Appendix. The simplicity of the tree structure in the latter figure has a direct impact on query performance, which is improved on by having simpler selection predicates and smaller data volumes, and the software complexity of the query processor, which is reduced considerably.

### 3.5 Quantitative Assessment

In this section we, give performance impressions which show that our technique scales with respect to size of the resulting database, as well as how they behave in querying and browsing a database. As application domains we chose readily available XML document collections: the ACM SIGMOD Anthology [Ley99], Webster’s Dictionary [Dyc00], and Shakespeare’s plays [Bos00].
## 3.5 Quantitative Assessment

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>1.2</td>
<td>5.6</td>
<td>6.8</td>
<td>8.0</td>
<td>4.4</td>
<td>4.9</td>
<td>5.0</td>
<td>5.0</td>
<td>8.8</td>
<td>12.7</td>
</tr>
<tr>
<td>2A</td>
<td>150</td>
<td>180</td>
<td>180</td>
<td>190</td>
<td>340</td>
<td>350</td>
<td>370</td>
<td>1300</td>
<td>1040</td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td>–</td>
<td>4.4</td>
<td>5.6</td>
<td>6.8</td>
<td>3.2</td>
<td>3.7</td>
<td>3.8</td>
<td>3.8</td>
<td>7.6</td>
<td>11.5</td>
</tr>
<tr>
<td>2B</td>
<td>–</td>
<td>30</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>190</td>
<td>200</td>
<td>220</td>
<td>1150</td>
<td>890</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of response times in ms for query set given in [SYU99] and replicated in Table 3.1. The first and the third line feature Monet XML’s execution times; the second and the fourth those reported in [SYU99].

![Graph showing response time vs. database size](image)

**Figure 3.6: Scaling of document**

We implemented Monet XML within the Monet database server, as described in [BK99]. The measurements were carried out on a Silicon Graphics 1400 Server with 1 GB main memory, running at 550 MHz. For comparisons with related work, we used a Sun UltraSPARC-IIi with 360 MHz clock speed and 256 MB main memory running SunOS 5.8.

### Database Size

The resulting sizes of the decomposition scheme are a critical issue. Theoretically, the size of the path summary can be linear in the size of the document as the worst case – if the document is completely un-structured. However, in practical applications, we typically find large structured portions within each document so that the size of the path summary and therefore the number of relations remains small. Table 3.3 shows the

<table>
<thead>
<tr>
<th>Documents</th>
<th>size in XML</th>
<th>size in Monet XML</th>
<th>#Tables</th>
<th>Loading</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM Anthology</td>
<td>46.6 MB</td>
<td>44.2 MB</td>
<td>187</td>
<td>30.4 s</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>7.9 MB</td>
<td>8.2 MB</td>
<td>95</td>
<td>4.5 s</td>
</tr>
<tr>
<td>Webster</td>
<td>56.1 MB</td>
<td>95.6 MB</td>
<td>2587</td>
<td>56.6 s</td>
</tr>
</tbody>
</table>

Table 3.3: Sizes of document collections in XML and Monet XML format.
Figure 3.7: Cumulative distribution of the relation sizes of the Webster Dictionary and Shakespeare's Plays
database sizes for our examples in comparison with the size of the original XML code. The third column contains the number of tables, i.e., the size of the path summary. The last column shows the total time needed to parse, decompose and store the documents.

It leaps out that the Monet XML version of the ACM Anthology is of smaller size than the original document once the bulkload has been committed. This reduction is due to the ‘automatic’ compression inherent in the Monet transform, where tag names are stored only once as meta information, and the removal of redundantly occurring character data done automatically in the Monet kernel. For example there are only few different publishers compared to the number of entries in general. In the decomposition, full entries of these fields can be replaced with references. We can expect similar effects to occur with other decomposition schemas such as the ones cited in the introduction to this chapter.

Figure 3.7 exhibits some characteristics of real-world XML documents: on display are the cumulative distribution of decreasingly sorted relation sizes of the Monet transform. For orientation, 95% lines indicate where most of the data are clustered. It turns out that only a small portion of the database tends to contain most of the information; the distribution of the histogram of the relation sizes itself is double-exponential.

**Scaling**

In order to inspect the scaling behaviour of our technique we varied the size of the underlying document. In doing so, we took care to maintain the ratio of different elements and attributes of the original document. We scaled the ACM Anthology from 30 to $3 \cdot 10^6$ publications which corresponds to XML source sizes between 10 KB and 1GB. The database sizes and the insertion times scaled linear in the size of the XML document.

**Querying**

To test for query performance under scaling we ran four queries consisting of path expressions of length one through four for various sizes of the Anthology. As Figure 3.6 shows, the response times for each query, given as a function of the size of the document, is linear in the size of the database. Only for small sizes of the database, the response time is dominated by the overhead of the database system. Notice that both axes are logarithmic.

At the time we conducted these experiments, only a few of the performance analyses published offered the possibility to reproduce and compare results, which made meaningful comparison difficult at that time. The results we used to compare Monet XML against were reported in [SYU99] who implemented their algorithms as a front-end to Postgres. In [SYU99], the authors propose a set of ten queries using Shakespeare’s plays [Bos00] as an application domain. We refer to their approach as SYU in the following. In Table 3.2 we contrasted response times of Monet XML with SYU obtained from experiments on the above mentioned Sun Workstation.

The figures display a substantial difference in response time showing that Monet XML outruns the competitor by up to two orders of magnitude (rows 1A, 2A). The times for SYU include a translation of XQL to SQL that is handled outside the database server. To allow for this difference, we additionally computed the response times relative to query 1 for both systems separately, assuming that preprocessing costs have a constant contribution. These figures exhibit actual query processing time only (rows
1B,2B). Monet XML shows an increase of processing time by less than 12 ms whereas SYU is up to 1150 ms slower than its fastest response time.

An analysis of the figures exhibits the advantages of the Monet model. While SYU store basically all data on a single heap and have to scan these data repeatedly, the Monet transform yields substantially smaller data volumes. In some extreme cases, the query result is directly available in Monet XML without any processing and only needs to be traversed and output. Another noticeable difference concerns the complexity of queries: the straightforward semantics of the Monet XML model result in relatively simple queries; conversely, the compiled SQL statements that SYU present are quite complex. Figure 3.8 illustrates this for Q2.

The comparison with Lore [MAG+97b] exhibited essentially the same trends on small document instances. However, we were not able to bulkload and query larger documents like the ACM Anthology as Lore requested more than the available 1 GB main memory. In contrast, using Monet XML we engineered a system functionally equivalent to the Web-accessible DBLP server [Ley99] that operated in less than 130 MB of main memory.

**Browsing a Database**

Our final experiments aim at assessing the systems' capabilities with respect to browsing. As an example consider a typical query as it is run on the ACM Anthology server several thousand times a day: 'Retrieve all conference publications for a given author.' Clearly, the size of the output may vary drastically and it is of particular interest for a browsing session that response times are kept low independent of the size of the answer.

Figure 3.9 shows both the total response time including textual rendering and response time of the repository. As expected, the time for rendering the output increases significantly yet is linear in the result size. However, the response time of the repository increases at a significantly lower rate. This is due to the reconstruction of the associations in the form of joins rather than chasing individual chains of pointers. Even for authors with a large number of publications the overall response time is well under one tenth of a second, which makes interactive browsing affordable. Also note that the
lower line in Figure 3.9 could also be interpreted as the cost of constructing a view while the upper line additionally includes rendering the view to textual XML.

The results presented demonstrate the performance potential of our approach which deploys full vertical fragmentation. The overall low response times show that reducing the data volume involved in isolated database operations at the expense of additional joins pays off very well in our scenario – not only in terms of overall performance but also when scaling is an issue.

### 3.6 Conclusion

In this section we first briefly summarise the impact that the choice of data model has on the overall architecture of a software system. In Monet XML, we assumed that object identifiers are only used to identify the object and do not carry additional semantics. The main reason for this was that we did not want to interfere with Monet's transaction system. We could, however, have chosen to encode the rank of a node into its OID, which would have forced us to lock large parts of a database if we want to update a document. Similar reasoning holds for other semantics like type information.

We have already seen that the data volume produced by different mappings during query execution can greatly impact performance. Furthermore the complexity of the mapping can have repercussions on the engineering complexity of the query processor and optimiser. A related point is that not every mapping can be bulkloaded on-line; for some strategies off-line preprocessing of, for example, a DTD is necessary.

So we presented a data model for efficient processing of XML documents. Our experiences show that it is worth taking the plunge and fully decompose XML documents into binary associations. The experimental results obtained with a prototype implementation based on Monet underline the viability of our approach: the effort to reduce data volume quickly pays off as gains in efficiency. Overall, our approach combines the elegance of clear semantics with a highly efficient execution model by means of a simple and effective mapping between XML documents and a relational schema.
3.7 Bibliographical Remarks

There exist a number of different approaches to viewing structured documents. A relatively early model, Groves, although never really formalised to the best of our knowledge, bears strong similarities to the association model that Monet XML deploys. Although the etymology is unknown, is a posteriori interpreted as ‘Graph Representation of Property Values.’ Informally, see [Cov01] for an overview of groves for practitioners. The site also contains interesting information on the history of marking up data structures in documents. According to [F.00] groves were introduced (1) to guarantee implementation independence of higher order concepts, (2) to provide a formal data model for SGML/XML, (3) to facilitate standard addressing and a simple query language, and, (4) to ensure scalability of application by abstracting from the syntax tree view of documents. Groves play an important role in defining views of documents and specifying semantics [Pre99]. For direct use in databases however they seem too general in their definition and too specific in their application.

By now, a plethora of literature is available now on architecture for processing semi-structured data and XML documents. [AQM+97] present the database system Lore which aims at semi-structured data in general and XML in particular; a theoretical basis for querying general graph structures is presented in [BDHS96]. The first work the author is aware of on processing XML data in relational systems is [FK99] who present an analysis of many different mapping strategies. [DFS99] present a method for extracting the regular parts of documents and storing them in tables; irregularities are stored in what they call overflow tables. The XML extensions of Lore are described in [GW97]. Unlike Lore, [KM00b, STH+99] and the aforementioned [SYU99] all make use of either relational or object-relational technology to store documents. The semantic homogeneity of Monet can also be characterised by the notion of competing non-terminals [Mur00], which it is designed to avoid.

Parts of this chapter were published in [SKWW00a] and later in [SKWW00b], of which it is an extended version.