

Inside XML Databases

B. Piwowski

November 3, 2005

Sources

The Ronald Bourret web pages about XML databases and articles (see the bibliography part).

This document is a draft and should not be considered to be perfect. It is only published so you have another source of information. USE IT WITH CARE.

1 Introduction

Important concepts that should be known:

- The XML document and the XML data models
- How to specify constraints on structure and content: DTD, schemas, etc.
- How to query XML documents or address part of them (query languages)

The key points:

1. Representation
 - What is the size of a document.
 - How to access a specific element.
 - Search and update.
2. Normalisation and referential integrity
3. Evaluating queries

1.1 What is an XML database?

Following R. Bourret, the first question we might ask is “is XML a database”?

We can define several types of documents: *regular* and *mixed*, also known as data- and document-centric. Regular documents resemble relational data (regular structure and mostly scalar values) while mixed documents have a highly irregular structure (XHTML page, word processing document, ...). Examples of queries for both can be found in .

The properties:

- Round-tripping or document equivalence. A data model implies a set of equivalence in the document sets. For instance, in most XML models
`` and ``
are totally equivalent. The round tripping property is a consequence of the document equivalence:

a perfect round tripping means that the original document, will be exactly the same when retrieved. It depends on the underlying model (see [15] for more information on standard information models). The only model where the round-tripping is perfect is, for evident reasons, text-storage based. In the XML Infoset model, which is the most complete, several informations are lost: the order of declarations within the DTD, the content models of the elements, the document type name, the difference between the two forms of empty elements, comments in the DTD, boundaries of CDATA marked sections, the order of attributes, and the location of declarations. The W3C has proposed the “canonical XML” [5] to specify how to transform a document into a canonical form.

1.2 Properties of XML Data and Queries

From [1], several properties of XML data are distinguished. These properties have a strong impact on storage design:

1. Attributes vs elements
2. Heterogeneity
3. Identity and structure
4. Structure dependance
5. Flexibility in mixed content
6. Presence of a schema. Schemas, in particular if the storage backend is relational, can have a strong influence on the data storage.
7. Null data. In the database world, null data means that the data is not there, either because the data was not available or is not possible. In XML, null can be defined as the absence of an attribute or an element.

2 Storing documents

2.1 Simple strategies

The simplest strategy is obviously to store XML documents in the file system. Some have been especially designed to search within plain XML documents. A slightly more sophisticated option is to store XML documents as BLOBs in a relational database.

2.1.0.1 Filesystem The simplest way to store XML documents; allow a hierarchical grouping of documents into collections, a feature which is often present in native XML databases.

2.1.0.2 BLOB The only interest is that we can use index structures of the database to index parts of documents. A document is stored into a table and “side tables” are used to index some parts of the documents like for example authors of an article (if the article is the XML document).

2.2 XML enabled databases

When a schema is available, it is also possible to use a transformation between an XML schema and a relational (or object) schema. This approach is known as the “XML enabled databases”, where the underlying storage (relational, object, etc.) is the model for data and XML is used as a simple exchange format.

The conversion of an XML schema (XS) into a relational schema (RS) implies the creation of several relations (e.g. one for each element in the DTD). A wrapper is used then used to feed the database from XML documents: for each XML document, a stream of tuples is created. In order to query the database using XML query languages, a query translator module has to be provided: its role is to convert an expression from the XML query language into a SQL expression. As results returned by the SQL expression are only a tuple stream, the output has to be converted to XML.

There are several ways to convert a XML schema into relational schema: a given element can either appear as a relation by itself or be included inside another relation. The latter is called an “inlined” element. Inlining an element avoids unnecessary joins at retrieving time *but* inlining may take more space (this is true if data has to be duplicated).

There are two approaches to this problem: simple heuristics that transform any XML schema into a relational schema or more complex algorithms which take into account the database usage and are cost based. All these approaches revert to the “BLOB” approach if the content of an element is ANY, that is can contain any element or text.

In [16], 3 mapping strategies (basic, shared and hybrid) are proposed. The first part of the processing is common to all these approaches. It creates a graph which is a simplification of the XML schema. Each node of the graph is an element (or an attribute) and edges can be annotated by * (the element appears 0 or more times) or ? (the element appears 0 or 1 times). Then, the following strategies are defined:

Basic creates a relation for *every* element in the DTD and inline every descendant (in the graph) until an edge annotated by * is traversed or the descendant is a part of a loop. The main drawback is that too many relations are created and that information can be spread over several tables.

Shared does create relations for elements with an in-degree greater than one only. Nodes with an in-degree of 1 are inlined, except if they are recursive or accessed through a “*” edge. Less relations are created than in “shared” but also some previously inlined elements are now relations.

Hybrid tries to inline more elements by allowing nodes with an in-degree superior to 1 to be inlined.

A more flexible mapping is LogoDB [4] and is cost-based. The principle is to first convert the schema into a transitional schema that have the main properties of being close to XML schema and easily convertible to a relational schema. An example is

```
type IMDB = [ Show*, Director*, Actor* ]
type Show = [ @type [ String ], Year, Aka{1,10}, (Movie | TV) ]
...
```

Statistics, collected from instances of the XML documents, are then appended to this schema leading to the **p-schema**.

A set of transformations, which does not modify the underlying document model, are then defined:

- inlining/outlining of an element. For example
type **book** = book [**Title**], type **Title** = title [String]
can be transformed into
type **book** = book [title [**String**]]
- Union factorisation/distribution $a.(b|c) = (a,b|a,c)$ and $a[t1|t2] = a[t1] | a[t2]$
- Repetition merge/split $(a+ = a,a^*)$
- From union to options $(t1|t2) \subset (t1,t2)$. With respect to all other transformations, the semantics here are modified.

This leads to the definition of a space of *alternative representations*. Each point of this space correspond to a different relational representation and is characterised by different query times. The query times define the cost of each representation, and a greedy algorithm is used to find a minimum.

2.3 Native XML database

As defined by the members of the XML:DB mailing list, the general properties of a native XML database are:

- Defines a logical model for an XML document
- Has an XML document as its fundamental unit
- Is not required to have any particular underlying physical storage model

For judging the quality of a naming scheme, the following properties seem well suited [17]:

Expressivity Which decision and reconstruction problems are supported? Decision refers to the process of deciding the relative position of two elements. Reconstruction is the existence of an arithmetic method that permits the computation of the identifier of a node relative to the considered one (e.g. its parent, etc.). Reconstruction can avoid a lookup in the database *and* also can avoid to store additional links between elements (*e.g.* a soft or hard link to the parent, etc.).

Runtime performance What is the performance for decision and reconstruction?

Prediction Is it possible to define a region for a given relation (ie, to predict the region for descendants, etc.)?

Storage consumption What is the size taken to encode XML trees?

Robustness Is it easy to add/remove nodes in the XML tree?

2.3.1 Relationnal storage

A good pointer [1]. Storing a generic XML document, without any schema information, implies the creation of a small quantity of tables in the relationnal storage. Several strategies have been developped so far.

2.3.1.1 Foreign Keys Foreign keys are the simplest way to encode . This way of storing information is fine if the depth of the tree is not too large. Otherwise, evaluating arbitrary paths expression can be prohibitive, since each descendant relation can imply an undefined number of joins. This storage is called KFO (Key, Foreign-key, Order).

2.3.2 Numbering schemes

The first numbering schemes support (almost) only decision.

2.3.2.1 Interval encoding Easy way of defining containment. It is easy to determine the relation between two nodes using this scheme. partitions the space into 4 four regions which are defined as in XPath: ancestor, preceding, descendant, and following nodes. The advantage of the interval encoding is that it is possible to define regions with spare numbers in case of update. Removing a node is also straightforward.

The XISS system proposes an extended preorder numbering scheme. This scheme assigns a pair of numbers $\langle \text{order}, \text{size} \rangle$ to each node, such that: (i) for a tree node y and its parent x , $\text{order}(x) < \text{order}(y)$ and $\text{order}(y) + \text{size}(y) \leq \text{order}(x) + \text{size}(x)$ and (ii) for two sibling nodes x and y , if x is the predecessor of y in preorder traversal, $\text{order}(x) + \text{size}(x) < \text{order}(y)$. While order is assigned according to a pre-order traversal of the node tree, size can be an arbitrary integer larger than the total number of descendants of the current node. The ancestor-descendant relationship between two nodes can be determined by the proposition that for two given nodes x and y , x is an ancestor of y if and only if $\text{order}(x) < \text{order}(y)$ and $\text{order}(x) + \text{size}(x) < \text{order}(y)$.

2.3.2.2 Pre-post A variant of the interval encoding is the pre-post scheme. The pre is the number given to a node is numbered before its children and its next siblings. The post order is the dual of the pre-order: it is the number associated to a node if the children are numbered before the current node. Another way to see this scheme is to imagine that (1) opening/closing tags are numbered separately in document order (2) each node is assigned two different numbers: its opening and closing tag number.

This numbering scheme can be used to determine windows. Let v and v' be two nodes; v' is a descendant of v iff:

$$\text{pre}(v') > \text{pre}(v) \text{ and } \text{post}(v') < \text{post}(v)$$

This can be rewritten

$$\begin{aligned} \text{pre}(v') &\in]\text{pre}(v), +\infty[\\ \text{post}(v') &\in [0, \text{post}(v)[\end{aligned}$$

It is then tempting to reduce the bounds so to restrict the set of candidates, which can be useful when using certain kind of indexes (like R-trees). First we need to draw a relation between level, pre, post and size:

$$\text{pre}(v) + \text{size}(v) = \text{post}(v) + \text{level}(v)$$

This can be proven by simply stating that

$$\begin{aligned} \text{pre}(v) &= |\text{ancestor}(v)| + |\text{preceding}(v)| \\ \text{post}(v) &= |\text{preceding}(v)| + |\text{descendant}(v)| \end{aligned}$$

The number of ancestors is by definition the level and the number of descendants the size. Let d be a descendant of v and l its last descendant. As $\text{size}(l) = 0$ and $\text{post}(v) \geq \text{post}(l)$, we can easily show that:

$$\text{pre}(v_{\text{last}}) \leq \text{post}(v) + h$$

where h is the maximum level in the database. Such demonstrations can be used to restrict the set of candidates for almost all the axes.

Like in the pre-post scheme, it is possible to leave space for new nodes. For both schemes, it is necessary to add the level in order to have a full support of the child, next and previous sibling axes.

2.3.2.3 Dewey The Dewey Decimal Classification was developed for general knowledge classification. To each node is associated a vector that represents the path from the document's root to the node. The Dewey code thus identifies a node, and encodes both its level and its ancestor identifiers. The drawback of this encoding is the size of the identifier which is both variable and long. With respect to the letter, it is possible to use variable encoding schemes. UTF-8 encoding scheme (see annex ??) was used to efficiently represent a vector.

The Dewey code allows to reconstruct easily the numbering of any related node (parent, child, etc.), even though they might no exist.

2.3.2.4 ORDPATH A variation of the Dewey numbering that allows for unbounded insertions is ORDPATH [13]. The idea is simple: we use a Dewey encoding where only odd numbers have an child-parent relationship. Negative numbers can also be used.

Example:

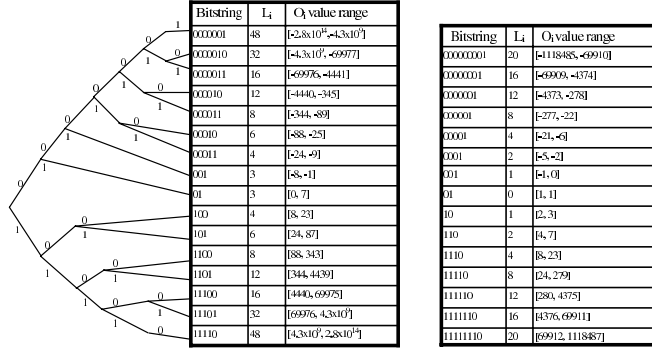


Figure 1: ORDPATH length encoding

- 1.3.1 is the previous sibling of 1.3.2.1, which is in turn the previous sibling of 1.3.2.3.
- 1.3.2.3 is the parent of 1.3.2.3.1

A simple encoding (bit length, number) is then used to encode numbers. This encoding is much more efficient in comparison to the UTF-8 scheme. The actual mapping between the lenght code and the real length in bits can be tuned in order to reduce storage consumption for IDs. The encoding allows to have a simple method for containment decision (substring) and for document order (lexical order).

2.3.2.5 Virtual nodes Virtual nodes are a special case of the Dewey code when the tree has a fixed arity (number of children for a node). Let a be the arity. A node identifier is then computed as for a dewey encoding $(1, i_1, \dots, i_n)$ is:

$$i_1 \times a^{n-1} + \dots + i_{n-1} \times a + i_n + 1$$

To prove that this numbering is valid (two different nodes cannot have the same id) and compact (there is no unassigned numbers).

This gives two easy formulas to compute a child's id and the parent id from a given node id. The problem is that the value space is high. With an arity a and a maximum depth n the maximum number is:

$$\#values = \sum_{i \geq 0}^n a^i = \frac{a^{n+1} - 1}{a - 1} \approx a^n$$

The number of bits required is roughly $n \times \log_2 a$. As an example, a tree of maximum depth 10 and arity 100 needs 9 bytes.

The virtual nodes allows for easy reconstruction (any node id can be reconstructed) and decision but has severe limitations: its storage consumption can be high (though not prohibitive), especially with trees with an important maximum depth.

2.3.2.6 eXist An enhancement of the previous scheme is to allow for different arity at different levels. This approach was taken by eXist; as the virtual nodes, it is a breadth first approach. For each document, a sequence (a_1, \dots, a_d) is associated where a_i is the maximum arity at a given level: a_1 is the number of child nodes for the element root node. The number of elements at a given depth d is then $n_d = a_1 \times \dots \times a_d$. The number of allocated nodes for a depth between 0 and d is $N_d = \sum_{k=0}^n a_k$. We have the following recursive formula:

$$id(1, i_1, \dots, i_d) = (id(1, \dots, i_{d-1}) - N_{d-2} - 1) \times n_d + N_{d-1} + i_d$$

where $id(1, i_1) = 1 + i_1$ and $i_d \leq n_d$ – thus allowing to reconstruct any ancestor and hence any node of the document.

Then, for a node $(1, i_1, \dots, i_n)$:

$$id = i_{n-1} + i_n + 1$$

2.3.2.7 Bird It is an alternative to the virtual node scheme and the (pre,size) sheme [17]. Any descendant n' as the property

$$Id(n) < Id(n_c) < Id(n) + w(n) \quad (1)$$

where $w(n)$ is the weight associated to the node n . This is similar to the (pre,size) approach. There are two major extensions:

1. The weight $w(n)$ is easily accessible and stored in a dataguide (a graph that summarises the structure of any XML document). To each node $g(n)$ (or g) in the dataguide is associated a weight $w(g)$ which is shared by all the siblings of the node in the dataguide. Siblings in the dataguide share the same weight.
2. The main id is proportional to the weight:

$$Id(n) \propto w(n) \quad (2)$$

From the two above equations, it is easy to prove that

$$Id(d) - (Id(d) \bmod w(a)) = Id(a)$$

We also know how to construct a child's id:

$$Id(c) = Id(p) + w(c) \times \# \text{ of the child} - Id(p) \bmod w(c)$$

The numbering scheme implies that for each node p in the dataguide and its child p_c , deriving from eq (1):

$$w(p) \geq w(p_c) \times \max_{n|p(n)=p} (\# \text{ children}(n) + 1)$$

which implies a basic recursive strategy to assign weights:

$$w'(p) = \begin{cases} 1 & \text{if no child} \\ \max_{n|p(n)=p} (\# \text{ children}(n) + 1) \times 1, \max_{p_c} w(p_c) & \text{else} \end{cases}$$

and

$$w(p) = \max_s w'(s)$$

where s is either a sibling node of p or a s -sibling (ie, share an ancestor at the s^{th} level). The former is an unbalanced scheme (less expressive) while the latter is a balanced scheme.

The BIRD numbering scheme allows a more efficient storage consumption than the virtual nodes. From a node, one can easily compute the ID of any of its ancestors and of any of its descendants. For DBLP and XMark collections, the BIRD scheme uses around 150 % of the storage path of the pre-order scheme.

2.3.3 Compressed DOM

The compressed DOM storage stores an XML document sequentially. It is a mere transformation of the XML characters and comes from the observation that document structure can be represented more efficiently. In this approach, the document trees are serialized into byte streams. We will study here three different visions of the compressed DOM.

In Xindice [3], documents are stored as a compressed DOM. The principle Xindice uses is deceptively simple here: for every XML document, Xindice will calculate something called the compressed DOM. This is an array of bytes which can be used to reconstruct the complete XML document at any time. An XML document is then stored as a (key, value) pair in the B-Tree, where the key is the name given to the XML document, and the value is the calculated Compressed DOM. For example, an element node will be stored as:

1. Signature (1 byte)
2. A record length (1-4 bytes)
3. Symbol id (2 bytes)
4. Attribute count (1-4 bytes)
5. Content (attributes first)

This scheme is not very flexible with respect to updates. In [11], the Natix storage is described. Trees are stored into one or more physical pages. Each physical page contains a subtree where each node can either be an *internal node*, a *content node* or a *proxy*. The proxy is a pointer to another physical page. This decomposition has an advantage over other serialisations: the semantics (here the structure) of the file is thus closer to the semantics of XML. The insertion algorithm may split records into two separate records in order to add a subtree. In order to split a record (a page), a separator is sought. This separator divides the subtree in tree sets: the preceding (L), the following and descendants (R), and the ancestors (S). The ancestors will be moved into the parent record while the other sets are stored in different record (the constraint is that a record can only contain siblings at the top level). In order to find the separator d , a ratio (L/R) and a tolerance (maximum deviation from the ratio) are used to specify the behaviour of the algorithm. To determine d , the algorithm starts at the subtree's root and recursively descends into the child whose subtree contains the physical "middle" (or the configured split target) of the record. It stops when it reaches a leaf, or when the subtree size in which it is about to descend is smaller than allowed by the split tolerance parameter. In order to force or forbid certain splittings, a "split matrix" may be used in order to modify the cost of separating two nodes (based on their names).

Another possibility is to highly compress the structure so it can fit into main memory. In Fuhr et al. [7], a highly compressed index is chosen for a representation called XS tree:

- levels rather than pointers are used for parent-child; level difference is then modified (-1-2) so it lies in \mathbb{N}^* and unary codes are used (ie, huffman encoding supposing that the lower the integer, the higher its frequency).
- tag names are encoded using huffman
- reference is by element number

It is possible to further optimise the storage efficiency by using a DTD (or any other grammar)

2.3.4 How to compute storage consumption for variable schemes

Three things can be computed and depends on the available statistics; they can be predicted by a mathematical model (expectation) or computed experimentaly for a given corpus:

1. The average identifier size (in bits). Usually done through experiments as it is difficult to estimate it.
2. The maximum size (in bits) - ie computing the worst case. For example, for the dewey code it is bounded by the maximum encoding size at each level.
3. The cost of an update (time, I/O operations, etc.)

One can use statistics on XML documents like [12]. For the BIRD numbering scheme, the easiest way to compute the storage consumption is to consider the worst case. etc.

2.4 Isolation (ACID)

We restrict ourselves to the discussion on the isolation property. For atomicity, consistency and durability refer to the current databases techniques as there is nothing really new.

The way to enforce isolation is borrowed from RDBs. We first define atomic operations (read a tuple, read a table, etc.). As the atomic operations from several transactions can be interleaved, we need ensure that the outcome of the transactions would be the same if the transactions be executed sequentially: the execution of the different atomic operations is said to be *serialisable* in this case. Now, how can we know if the outcome would be the same? It will be the same only if we can *safely* swap operations from the different transactions in order to have a sequential execution plan. For example, let us consider two transactions A and B, each operation being composed of 3 operations (A1, A2, A3 and B1, B2, B3). The execution A1 A2 B1 B2 A3 B3 is serialisable if the execution of A3 is not modified by the execution of B1 and B2 (it this case the equivalent execution is A1 A2 A3 B1 B2 B3). See the notion of precedence.

There are three ways to ensure the independance:

predicting must know all the operations of the transactions before scheduling

detecting we abandon a transaction whenever it violates the seriability

avoiding using a lock protocol, we avoid that a transaction violates the seriability by making it wait.

The 2 phase locking protocol (2PL) is an *avoiding* protocol that ensure the seriability: each granule (ie, atomic unit) is either unlocked, or locked in a shared or exclusive mode. The principle is that once a transaction releases a lock, it cannot obtain a new one. A shared lock is granted when a resource has no lock or has only shared locks; an exclusive lock can be granted only when there are no other locks granted. In the “real world”, this can lead to deadlocks.

For XML, the main problem (and the main discussions) are based on the granularity problem. [10] use a DOM model of the XML document (a node has a first/last child, a previous/next sibling; nodes can be queried by id or by name; the text content, the node name and the attributes of a name can be queried) and propose to use three different granularity levels (plus **Doc2PL** whose granularity is the document):

	Granted					
Requested	<i>None</i>	<i>IS</i>	<i>IX</i>	<i>S</i>	<i>SIX</i>	<i>X</i>
<i>IS</i>	+	+	+	+	+	P
<i>IX</i>	+	+	+	P	P	P
<i>S</i>	+	+	P	+	P	P
<i>SIX</i>	+	+	P	P	P	P
<i>X</i>	+	P	P	P	P	P

Table 1: DGLock lock compatibility. + means that the lock is granted, P means that there is a predicate test before.

1. **Node2PL** locks at the parent level: going to the n th child of a node (S) or inserting a child (X). Number of different locks is the number of nodes.
2. **NO2PL** locks nodes whose pointers are (at least conceptually) accessed. For instance, if we insert a child D before the first child of P, then two locks have to be acquired: one for the parent node P (has a pointer to the first child) and the other for P's first child (has a pointer to the previous sibling). Of course, this is true only if the only way to navigate to D is from its parent or its previous sibling. Note that it's not necessary to lock D itself, as there is no way to go to this ndo. Number of different locks: N also.
3. **OO2PL** locks pointers. There are four pointers for each node (first child A, last child Z, preceding L and following R sibling). The compatibility table only forbids to acquire an exclusive lock to an already locked pointer. Number of different locks: $4 * N$

Quick discussion: according to the performances, $\text{Doc2PL} < \text{Node2PL} \sim \text{NO2PL} < \text{OO2PL}$. For the overhead this is the contrary. Exercise: find an example of each operation that can execute concurrently with one protocol but not with the other.

One can have a distinction between structural traversal/modification and content reading/modification. It is possible to do better with a DTD.

2.4.1 Semantic locking

The locking is said to be “semantic” when the application semantics are taken into account while locking.

DGLock [8] locks at a dataguide level (ie, not on physical atomic units): a granule is a node in the dataguide with an associated predicate. This work takes over the idea of granular locking on directed acyclic graphs. All the way from the root to the node is locked (either X or S, depending on whether it is an update or not). For instance, a read on `/store/auction[price>5]` will requires IS lock on `/store`, `/store/auction` and an S lock on `/store/auction/price` with a predicate “`. > 5`”. There are also locks which can be acquired for a node and all its descendants (IS and IX). There is also a special lock (SIX) for shared lock with intention to acquire an exclusive lock below. This gives a complex compatibility table: for instance, a IS lock is always granted but when there is an X lock.

1. Examples: T1 queries `/store/auction[price>5]/description` and while T2 update `/store/auction[price<.5]` to set price to 4. They can execute independently. If executing T3 `/store/auction[price>3]/description` concurrently with T2 = problem.

```

<element name="courses">
  <complexType><sequence>
    <element ref="course" maxOccurs="unbounded"/>
  </sequence></complexType>
  <key name="courseKey">
    <selector xpath="./course"/>
    <field xpath="@id"/>
  </key>
</element>
<element name="students">
  <complexType><sequence>
    <element ref="student" maxOccurs="unbounded">
  </sequence></complexType>
  <keyref name="studentCourse" refer="courseKey">
    <selector xpath="./student"/>
    <field xpath="course"/>
  </keyref>
</element>

```

Figure 2: Example of an integrity reference in XML Schema (excerpt)

2.5 Normalisation and referential integrity

2.5.1 Referential integrity

2.5.2 Normalisation

A good introduction to the 5NF is given in [?]. Normalisation refers to the process of designing a database schema such as a given piece of data is represented only once. The normalisation goal is to eliminate (or at least minimise) the redundancy in the data. The normalisation takes place at the schema level and has a direct effect on the data. Besides reducing redundancy, it helps to:

1. Reduce ambiguity in data expression
2. Facilitate preservation of data consistency
3. Allow for rational maintenance of data

In relational database, normalisation is related to the concepts of normal forms. In the context of XML retrieval, it is tempting to extend the concepts [14] as data redundancy can appear as well.

First normal form The first normal form is just for the data being in relational format; it would be a well formed XML document.

Second normal form The second normal form is violated when a nonkey field is a fact about a subset of a key. Example: in the relation (Time, Street, # of buses, # holes) the # holes is a fact about the street (but does not depend on time). With functional dependencies, a relation is 2NF if for every FD $X \rightarrow A$ where A is not part of the key, X is not a strict subset of a key.

Third normal form The third normal form is violated when a nonkey field is a fact about another nonkey field. For example, in (Employee, department, location) location is a fact about the department. With functional dependencies, a relation is 3NF if for every FD $X \rightarrow A$, X is a superkey or A is a part of a key.

```

<!ELEMENT courses (course*)>
<!ELEMENT course (title, taken_by)>
<!ATTLIST course cno CDATA #required>
<!ELEMENT title (#PCDATA)>
<!ELEMENT taken_by (student*)>
<!ELEMENT student (name, grade)>
<!ATTLIST student sno CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT grade (#PCDATA)>

```

Figure 3: An example DTD

Boyce-Codd Normal Form (BCNF) The Boyce-Codd normal form is even more restrictive: all fields are dependent on the key. With functional dependencies, a relation is BCNF if for every FD $X \rightarrow A$, X is a key.

Fifth Normal Form Deals with multivalued facts. Not so important in XML since the one-to-many relationship can be easily used.

2.5.2.1 Extension to XML In order to extend functional dependencies, the following steps are necessary [2]¹:

1. Introduce functional dependencies within XML documents.

How is a tuple defined in RDB? It is simply a function from an attribute name to a value in a given domain. In order to achieve that goal, we can flatten the representation of an XML document. A tree is a set of tree-tuples and each tree tuple can be easily defined as the mapping of a subtree of the XML document into a tree tuple. The subtree is chosen so that every simple path (s-path) /A/B etc. returns one node at most. Equality can be defined on these subtrees (when the order between nodes is dropped) when there is an exact match (ie, same node or same value if a #PCDATA or an attribute).

A functional dependency is then defined between two sets of s-paths, S and T , if for any tree tuple such that all the subtrees defined by the s-paths in the left part of the FD are equal, then so are the subtrees of the s-paths of the right part.

In Figure 3, we could define the following FDs:

- A course is determined its id:
/courses/course/@cno -> /courses/course
- Two students from the same course cannot have the same id:
/courses/course, /courses/course/taken_by/student/@sno -> /courses/course/taken_by/student
- A student is determined by its id:
/courses/course/taken_by/student/@sno -> /courses/course/taken_by/student

There is also a notion of **transitive closure** $(D, \Sigma)^+$ like in relational databases.

2. Definition of (a) normal form(s) that disallow FDs that cause redundancy.

For Σ compatible with D , (D, Σ) is in XNF, if for every **non-trivial** FD φ from $(D, \Sigma)^+$ of the form $S \rightarrow p/\# \text{ PCDATA}$ or $S \rightarrow p/@\text{att}$, then $(D, \Sigma) \models S \rightarrow p$ is such as $T \models \varphi$.

¹we follow their proposition in the actual examples

We can see that a modified version of the third FD implies that the DTD is not in XNF:

/courses/course/taken_by/student/@sno -> /courses/course/taken_by/student/name/#PCDATA
and /courses/course/taken_by/student/@sno -> /courses/course/taken_by/student/name does not hold.

This can be fixed by changing the schema.

3. Automatic conversion of a grammar into its normal form (in RDB, there is an algorithm for 3FN).
Lossless decomposition. Not to be covered.

3 Retrieving documents

Three parts in this section:

- How do we represent the result of an XQuery
- How to compute certain physical operators
- How to accelerate with indexes

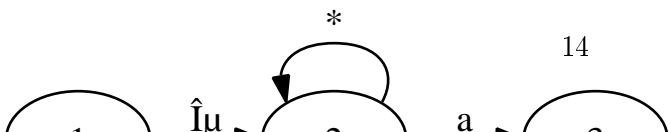
3.1 Answering XQuery

3.1.1 Automata

Automatas: they are event driven (useful for streams) but are not as easily optimisable as an algebra. An example of automaton applied to XPath processing is the NFA (Non-deterministic Finite Automata). The idea is simple: an automaton can recognise regular languages, that is simple symbol strings. An element within an XML document can be seen as the succession of element names from the root element to the element. For instance, a paragraph (P) in a section (S) in an article (A) is represented as the string ASP.

If we want to evaluate the expression //paragraph, we can then build an automaton that recognise any string composed of a zero-or-more sequence of symbols and finished by the symbol paragraph. An NFA for this XPath will have three state (1. start, 2. any, 3. end). The transition from 1 to 2 is ϵ , the transition from 2 to 2 is "*" (any symbol from the alphabet), and the transition from 2 to 3 is "paragraph".

For each //a, we have this automata:



3.1.2 Algebra

The purpose of an algebra is to

Most of the approaches used in query evaluation are based on the algebra. As in relational databases, we can distinguish two types of algebra: a physical algebra and a logical algebra.

3.1.2.1 MonetDB The MonetDB algebra [9].

3.1.2.2 Proximal nodes

3.2 Algorithms

The algorithms below focus mainly on the problem of evaluating a path step (staircase join) or several path steps (Twig Queries). The main idea of these two algorithms are the same as the merge join of relational databases.

3.2.1 Twig queries: PathStack

Instead of computing each step in an XPath expression, it is also possible to group them into an Twig Query (also known as pattern queries).

The most famous algorithm is the TwigStack which is a generalisation of PathStack for twig queries. Twig stack can be even improved using specific index structures like XB-trees, leading to the TwigStackXB algorithm [6].

Twig stack ensures that before selecting an element from a stream

1. The element has a descendant in each of its child streams
2. The property is verified recursively

3.2.2 Staircase join

Handles all axes **but** is not global as TwigStack. However, it is better than TwigStack because we don't need all the couples everytime (ex. for //A//B//C, we don't care about all the couple (a,b) - we just need the set of b that are a descendant of at least one A).

How: elements are ordered in document order and with a (pre,size) scheme or equivalent. Three strategies:

1. Pruning: removes any element from the context set that is not important for the result
2. Partitioning: consider only non-overlapping parts for any two component nodes (the idea is to avoid extra work and duplicates).
3. Skipping: useful to skip a set of elements from B if we have a bound on the next valid one them.

For some algorithms, you may need a stack (for example, for the child axis).

3.2.3 Other “tricks”

Independent loops (XQuery)

Existential operators

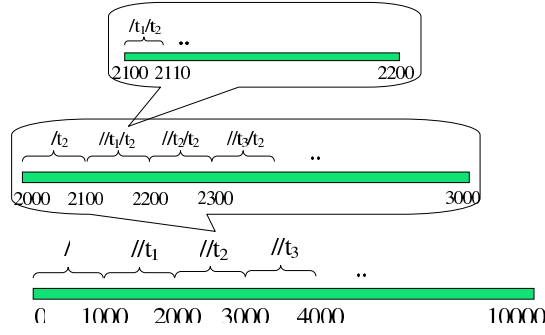


Figure 4: P-Labeling

3.2.4 Evaluating query plans

Not to be covered. Deals with the problem of the estimation of selectivity (of an XPath expression for example)

3.3 Indexes

Specific algorithms, often based on specific data structures (the index), are used to accelerate query processing in databases in general and XML databases in particular. In this section, we describe some of the optimisation techniques used in XML Query evaluation.

3.3.1 Values

3.3.2 Structural

Element

blah

Parent-child

blah

XB-Tree

An XB-tree [6] is a variant of a balanced tree (B-Tree) and is designed for indexing the positional representation (DocId, LeftPos, RightPos, LevelNum). The nodes are stored by LeftPos and each page corresponds to a segment.

P-Labeling

P-labelling (Figure 4) is used in order to find efficiently paths of the type $//a/b/c$ or $/a/b/c/...$. Let call P the set of paths. They define an order on paths: $p_1 \subset p_2$ iff the nodes returned by the evaluation of p_1 is included in the set of nodes returned by p_2 : this is denoted

$$\llbracket p_1 \rrbracket \subset \llbracket p_2 \rrbracket$$

For example, $\llbracket //a/b \rrbracket \subset \llbracket //b \rrbracket$. It is easy to prove that the sets of results of an XPath of P are either included or completely disjoint. That is

$$\llbracket p_1 \rrbracket \subset \llbracket p_2 \rrbracket \text{ or } \llbracket p_1 \rrbracket \supset \llbracket p_2 \rrbracket \text{ or } \llbracket p_1 \rrbracket \cap \llbracket p_2 \rrbracket = \emptyset$$

3.4 Other

3.4.1 Path summaries

Can be seen as an *a posteriori* schema. The origins can be found in Dataguides [?] which were proposed in the Lore framework. A dataguide is a summary of the structure of all the stored XML documents and is constructed and updated each time a document is inserted or modified. For XML documents, a dataguide is a forest of trees such that a simple XPath expression of the type `/a/b/c` (only child axis and element name node test) yields one and only one node. Formally, a dataguide is valid for a database if there exists a mapping m of nodes in documents to nodes in the dataguide such as:

1. if x has parent y then $m(x)$ is the parent of $m(y)$
2. x has no parent $\Leftrightarrow m(x)$ has no parent.
3. If x and y have the same parent and the same local part (label, tag name), $m(x) = m(y)$

Dataguides can also be useful when helping the user to write its query, by checking if paths are valid. A recent article is [?]. Path summaries can also be used for path expansion.

3.4.2 Signatures

associate a signature

References

- [1] Sihem Amer-Yahia. Storage techniques and mapping schemas for xml. Technical report, AT&T Labs Research, 2003.
- [2] Marcelo Arenas and Leonid Libkin. A normal form for xml documents. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 85–96, New York, NY, USA, 2002. ACM Press.
- [3] James Bates and Kevin O'Neill. Xindice 1.1 internals guide. <http://xml.apache.org/xindice/dev/guide-internals.html>, 2003. Version 193056.
- [4] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Legodb: Customizing relational storage for xml documents. In *VLDB*, pages 1091–1094, 2002.
- [5] John Boyer. Canonical xml version 1.0. <http://www.w3.org/TR/2001/REC-xml-c14n-20010315>, March 2001.
- [6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 310–321. ACM, 2002.
- [7] Norbert Fuhr and Norbert Gövert. Index compression vs. retrieval time of inverted files for XML documents. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management*, pages 662–664, McLean, Virginia, USA, November 2002. ACM.
- [8] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. Xmltm: efficient transaction management for xml documents. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 142–152, New York, NY, USA, 2002. ACM Press.

- [9] Torsten Grust and Jens Teubner. Relational algebra: Mother tongue—xquery: Fluent. In *Proceedings of the first Twente Data Management Workshop on XML Database and Information Retrieval*, Enschede, The Netherlands, 2004.
- [10] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Evaluating lock-based protocols for co-operation on xml documents. *ACM SIGMOD Record*, 33(1):58–63, 2004.
- [11] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. Lehrstuhl für praktische Informatik III 8, Universität Mannheim, June 1999.
- [12] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The xml web: a first study. In *WWW*, pages 500–510, 2003.
- [13] Patrick E. O’Neil, Elizabeth J. O’Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *SIGMOD Conference*, pages 903–908. ACM, 2004.
- [14] Will Provost. Normalizing xml. On the web, two parts: <http://www.xml.com/pub/a/2002/11/13/normalizing.html> and <http://www.xml.com/pub/a/2002/12/04/norma> 2002.
- [15] Airi Salminen and Frank Wm. Tompa. Requirements for xml document database systems. In *ACM Symposium on Document Engineering*, pages 85–94. ACM, 2001.
- [16] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.
- [17] Felix Weigel, Klaus U. Schulz, and Holger Meuss. The bird numbering scheme for xml and tree databases - deciding and reconstructing tree relations using efficient arithmetic operations. In Stéphane Bressan, Stefano Ceri, Ela Hunt, Zachary G. Ives, Zohra Bellahsene, Michael Rys, and Rainer Unland, editors, *XSym*, volume 3671 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2005.