XML Access Modules: Towards Physical Data Independence in XML Databases

Andrei Arion
INRIA Futurs and Univ. Paris XI, France
Andrei.Arion@inria.fr

Véronique Benzaken
LRI-CNRS, Univ. Paris XI, France
Veronique.Benzaken@lri.fr

Ioana Manolescu
INRIA Futurs, France
Ioana.Manolescu@inria.fr

1. INTRODUCTION

A key factor for the outstanding success of database management systems is *physical data independence*: queries, and application programs, are able to refer to the data at the logical level, ignoring the details on how the data is physically stored and accessed by the system. The corner stone of implementing physical data independence is an *access path selection algorithm*: whenever a disk-resident data item can be accessed in several ways, the access path selection algorithm, which is part of the query optimizer, will identify the possible alternatives, and choose the one likely to provide the best performance for a given query [22].

In the field of XML database management systems (*XDBMSs*, in short), physical data independence remains yet to be achieved. Many XML storage, labeling, and indexing methods have been proposed so far. However, the data layout produced by a given storage scheme is typically hard-coded within the query optimizer of the corresponding system. This situation reduces the XDBMS's flexibility, by locking it within one storage model, while different applications may have different needs. It also raises performance issues. Various workloads and data sets may need adding, e.g., an index or a materialized view; the optimizer should automatically understand how the new persistent structure could be used to answer queries.

We introduce *XML Access Modules (XAMs)*, a step towards physical data independence in XDBMSs. A XAM describes, in an algebraic-style formalism, the information contained in a persistent XML storage structure, which may be a storage module, an index, or a materialized view. The set of XAMs describing the storage is used by the optimizer to build data access plans. Using XAMs, a change to the storage (adding or removing a storage structure) is communicated to the optimizer simply by updating the XAM set.

One of the most useful XAM features is the ability to model indexes whose "keys" and "values" may be complex combinations of XML structure and values. In this respect, XAMs can be seen as a generalization *relational binding patterns* to XML. Relations with binding patterns have been show useful for relational query optimization [21, 11], and these benefits carry over to XAMs.

In the following, Section 2 introduces XAMs, and Section 3 presents its algebraic foundations, with a focus on index support. We briefly discuss related works and perspectives in Section 4.

Informal Proceedings of the Second International Workshop pn *"XQuery Implementation, Experience and Perspectives" (XIME-P)*, in cooperation with ACM PODS/SIGMOD, Baltimore, Maryland, 2004.

2. XML ACCESS MODULES

XAMs are a formalism for representing an XML storage, index or materialized view. This formalism must be general enough to capture many existing (and future !) proposed structure. It must have clearly defined semantics, to allow the optimizer to make informed decisions. Finally, it should be relatively easy to understand and to express.

Enumerating existing XML storage schemes is clearly out of the scope of this work; a classification attempt can be found in [14]. We list here some interesting dimensions of such schemes. The *degree of fragmentation* may go from very low (*blob* storage) to very high (*node-oriented* storage). The *clustering criteria* may be very simple, e.g., cluster nodes by their name [10], or very complex, e.g., cluster nodes connected by a complex path expression [13, 18]. The clustering criteria may be derived from the document's schema, content, a workload, or a combination thereof. Most XML storage structures rely on a *node labeling scheme* which provides persistent identifiers to XML nodes. Such IDs may have interesting properties: they may reflect document order, may allow establishing structural relationships, or may furthermore allow deriving the ID of a node from the ID of e.g., its children. Knowledge about ID properties is crucial for the optimizer in order to exploit them.

Particular applications may not need to use all the above spectrum of choices. However, just like relational database systems, XDBMSs should be able to support different XML data sets and query workloads. Thus, XAMs should capture the above aspects. We describe them next.

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

We now describe XAM specifications, using a grammar-like notation (Figure 1). We use bold font for terminal symbols of the grammar, i.e. constants.

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

A node may be annotated with: an identifier specification $IDSpec$, a tag specification $TSpec$, a value specification $VSpec$, and a content specification $CSpec$.[1] By content, we mean the full (serialized) representation of the XML element or attribute.[2] An ID (resp.

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

[2]Clearly, the content of an XML element can always be retrieved

$$NS ::= \top\ N^+ \quad (1)$$

$$N ::= name\ IDSpec?\ TSpec?\ VSpec?\ CSpec? \quad (2)$$

$$IDSpec ::= \textbf{ID}^{\textbf{i}} \mid \textbf{o} \mid \textbf{s} \mid \textbf{p}\ \textbf{(R?)} \quad (3)$$

$$TSpec ::= (\textbf{Tag}(\textbf{R}?)) \mid \textbf{[Tag=}c\textbf{]} \quad (4)$$

$$VSpec ::= (\textbf{Val}(\textbf{R}?)) \mid \textbf{[Val=}c\textbf{]} \quad (5)$$

$$CSpec ::= \textbf{Cont} \quad (6)$$

$$ES ::= E* \quad (7)$$

$$E ::= name_1\ (\textbf{/} \mid \textbf{//})(\textbf{o} \mid \textbf{j} \mid \textbf{s} \mid \textbf{nj} \mid \textbf{no})\ name_2 \quad (8)$$

Figure 1: XAM grammar.

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

Figure 2: Sample XML document.

tag, value, content) specification, attached to a XAM node, denotes the fact that the element/attribute ID (respectively, tag, value, or full textual content) is stored in the XAM.

Node identity is a crucial notion in XQuery processing. Any XML store provides some persistent identifiers; the particular type of IDs used determines the efficiency of matching structural query conditions. The persistent identifiers stored in a XAM are described by the ID specification (line 3); it consists of the symbol **ID**, and one of four symbols, depending on the level of information reflected by the element identifier. We use **i** for simple IDs, for which we only know that they uniquely identify elements. The symbol **o** stands for IDs reflecting document order; simple integer IDs used e.g., in [6, 10, 23] are a typical example. We use **s** to designate structural identifiers, which allow to infer, by comparing two element IDs, whether one is a parent/ancestor the other. They are produced e.g., by the popular (preorder, postorder, depth) labeling schemes based on Dietz's model [8]. We use **p** to designate structural identifiers which allow to *directly derive* the identifier of the parent from that of the child, such as the Dewey scheme in [24], or ORDPATHs [19].

An **R** symbol in an ID specification denotes an *access restriction:* the ID of this XAM node is *required* (must be known) in order to access the data stored in the XAM. This feature is important to model persistent tree storage structures, which enable navigation from a parent node to its children, as in [12, 9]. More generally, **R** symbols allow to model arbitrary XML indexes, on structure and values: key values must be known to perform an index lookup [15].

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

XAM edges can be either parent-child edges, marked **/**, or ancestor-descendent edges, marked **//**. We distinguish *join, left outerjoin, left semijoin, nest join* and *left nest outer join* semantics for the XAM edges, considering the parent node on the left hand). These are marked by the symbols **j**, **o**, **s**, **nj**, respectively **no**. All these joins correspond to structural relationships; nest join variants furthermore allow the construction of complex nested tuples. The operators will be detailed in Section 3.1.

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

from a non-lossy storage, by combining accesses to *several* storage modules. Here, we use Cont only for the storage models able to retrieve it from a *single* persistent data structure.

3. XAM SEMANTICS

The semantics of a XAM χ over a document d is the data contained in a storage module described by χ over document d. It is an instance of a nested relational data model [1, 2], enhanced with order, and further specialized to our setting. This data model features:

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

The value of a tuple attribute can either be of atomic type, or a set/list of tuples; nested tuples are not allowed. Lists (or sets) contain homogeneous tuples; lists of lists are not allowed. Thus, the model allows nesting of tuples and sets/lists, but only in alternation. This model is well-adapted to the hierarchical, ordered structure of XML data, and conceptually close to the XQuery data model [25]. It is also reminiscent of tuple-based XML algebras, as described in [17]. However, XAM semantics adapts it to the needs of storage description, as we explain next.

We define XAM semantics in two stages: first, omitting the **R** annotation (Section 3.1), then including them in Section 3.2. We start by introducing some useful notions.

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

DEFINITION 3.1 (TAG-DERIVED COLLECTION). Let t be an element name and d be an XML document. We define the *tag-derived collection (set/list) of* t as a set/list of tuples R_t(ID:ID, Val: $\mathcal{A}$, Tag: String, Cont: String):

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

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

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

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

Similarly, the collection $R_t^@$ reflects all attributes nodes labeled t, and $R_*^@$ reflects all attribute nodes. □

R_{book}

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

$R^{@}_{year}$

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

R_{title}

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

Figure 3: Tag-derived lists on the document in Figure 2.

As an example, Figure 3 shows the tag-derived lists $R_{book}(d)$, $R_{title}(d)$ and $R^{@}_{year}(d)$, where d is the sample document in Figure 2. For simplicity, we will only use the attribute names ID, Val, Tag and Cont in association with the above types, and omit the atomic attribute types.

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

DEFINITION 3.2 (STRUCTURAL JOINS). Let R and S be tuple sets, and $R.x$ and $S.y$ be attributes of type ID. For a given tuple $t_R \in R$, let $child(t_R.x, S.y)$ be the set of tuples in S whose y attribute is a child of $t_R.x$.

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

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

where $\|$ stands for tuple concatenation.

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

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

The parent-child structural outerjoin of R and S, $R \bowtie\!\!\!\!\subset^{pc} S$, is:

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

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

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

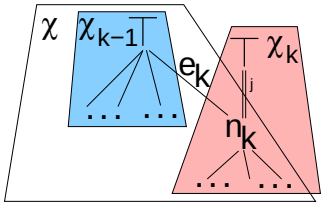

Figure 4: Generic XAM.

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

DEFINITION 3.3 (NEST STRUCTURAL JOINS). Let R and S be two set of tuples, and $R.x$ and $S.y$ be two attributes of type ID. The nest parent-child structural join of R and S, denoted as $R \bowtie^{pc}_{n} S$, is:

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

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

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

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

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

Nest structural joins have been mentioned in [20]; we formalized them here within our data model to make this paper self-contained.

3.1 Semantics of a XAM without access restrictions

In this section, we focus on a XAM χ without **R** annotations. Without loss of generality, we assume χ to be ordered.

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

DEFINITION 3.4 ($\top$ SEMANTICS). Let χ consist of the single node $\top$. In this case, we have:

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

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

DEFINITION 3.5 (TWO-NODE XAM SEMANTICS). Let χ be a XAM consisting of a node $\top$ connected to a node n_1 by an edge labeled with // and **j**. The semantics of χ over a document d is:

1. If n_1 is an element node, with a $TagSpec$ of the form **[Tag=*t*]**, then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R_t(d)))$.
2. If n_1 is an element node with a different $TagSpec$ (or none), then, $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R_*(d)))$.

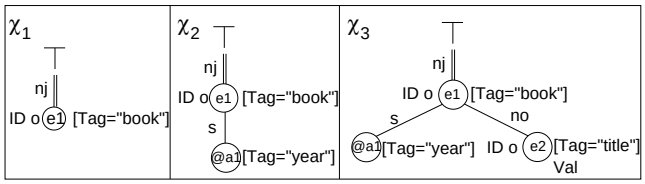

Figure 5: Sample XAMs to illustrate XAM semantics.

3. If n_1 is an attribute node, with a $TagSpec$ of the form **[Tag=t]**, then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R^{@}_t(d)))$.

4. If n_1 is an attribute node with a different $TagSpec$ (or none), then $[\![\chi]\!]_d = \Pi_\chi(\sigma_\chi(R^{@}_*(d)))$.

In the above formula:

1. σ_χ is a selection on the conjunction of all predicates of the form **Val=**c that appear in the value specifications of χ nodes. σ_χ checks each such predicate against the respective Val attributes.

2. Π_χ is a projection which: (i) eliminates the root attribute, (ii) for every non-$\top$ node in χ, retains the ID (respectively, the Val, Tag, Cont attribute) only if the node has an ID specification of the form **ID** (respectively, value specification of the form **Val**, tag specification of the form **Tag**, and content specification of the form **Cont**) and (iii) eliminates duplicate tuples.

□

Now, consider a larger XAM, such as χ in Figure 4. In this figure, χ_{k-1} is the same as χ but for the rightmost subtree, rooted in node n_k. Furthermore, χ_k is obtained by adding a $\top$ node on top of n_k, connected to n_k by a descendent edge annotated j (join).

Simplifying assumption: IDs present. We start by assuming that n_k has an ID specification of the form **ID**.

DEFINITION 3.6 (SEMANTICS OF A XAM WITH IDS). Let χ be a XAM, χ_{k-1} and χ_k be the XAMs derived from χ as in Figure 4. The semantics of χ over a document d, denoted $[\![\chi]\!]_d$, is:

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

In the above, σ_χ and Π_χ are defined as in the previous definition, while $\circ$ stands for:

- structural join if e_k is labeled **j**
- structural semijoin if e_k is labeled **s**
- structural outerjoin if e_k if e_k is labeled **o**
- nest structural join if e_k is labeled **nj**
- nest-structural outerjoin if e_k is labeled **no**

The structural relation tested by the above structural (possibly nest) join, outerjoin or semijoin depends on the edge e_k: it is parent-child if e_k is labeled /, and ancestor-descendent if it is labeled //. □

This definition constructs a structural join tree isomorphic to the XAM tree itself; structural joins are paranthesized bottom-up.

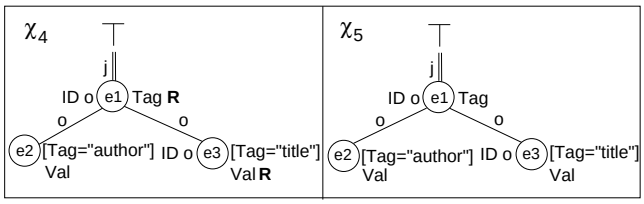

Figure 6: Sample XAM with access restrictions.

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

DEFINITION 3.7 (SEMANTICS OF A XAM IN GENERAL). Let χ be a XAM and χ' be the XAM obtained from χ by adding ID specifications to χ's node n_k as above. The semantics of χ is:

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

where the semantics of Π_χ is specified in Definition 3.5. □

For example, consider the XAMs in Figure 5. Let d be the XML document in Figure 2, where node numbers are used as order-preserving IDs. By Definition 3.5, we obtain:

$[\![\chi_1]\!]_d$= [e1(ID=2, Tag="book"), e1(ID=7, Tag="book")]

In the above, and in the sequel, XAM node names appear explicitly in every tuple, to facilitate reading.
We obtain $[\![\chi_2]\!]_d$ by a structural semijoin on $[\![\chi_1]\!]_d$ and $R^{@}_{year}$:

$[\![\chi_2]\!]_d$= [e2(ID=2, Tag="book")]

Only the first tuple from $[\![\chi_1]\!]_d$ contributed to $[\![\chi_2]\!]_d$, since only the first book had a match in $R^{@}_{year}$.
Applying again Definition 3.6 on $[\![\chi_2]\!]_d$, we obtain:

$[\![\chi_3]\!]_d$= [e1(ID=2,Tag="book",
e2[(ID=4,Tag="title",Val="Data on the Web")])]

3.2 Semantics of a XAM with access restrictions

We now extend the XAM semantics to account for the **R** (required) marker. Intuitively, values for the required fields have to be known, to be able to access the data stored by the XAM.

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

For instance, consider the XAM χ_4 in Figure 6. χ_4 contains information about elements having "title" and "author" sub-elements. However, in order to access this information, one must provide the tag of the elements corresponding to e_1, and the title associated to these elements. A typical storage structure modeled by χ_4 would be an index on publications, with a composite index key consisting of the publication type and title (the required attributes in χ_4).

For instance, consider the following binding tuple for χ_4:

t_{B1}= e1(Tag="book", e3[(Val="Data on the Web")])

The information content of χ_4 on the document d from Figure 2, with the bindings list $[t_{B1}]$, is:

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

input : $t(a_1, a_2, \ldots, a_k)$, where $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ are marked R;
binding tuple $t_B(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$
output: $t \cap b$

1 $t_{RES} \leftarrow 0_t$
/* t_{RES} is a tuple having t's type, atomic attributes set to $\perp$, and collection attributes set to empty collections */
2 **foreach** a_{i_j} *in* $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ **do**
/* check if t matches the binding in b */
3 **if** a_{i_j} *is of an atomic type* **then**
4 **if** $t.a_{i_j} = b.a_{i_j}$ **then**
5 $t_{RES}.a_{i_j} \leftarrow t.a_{i_j}$
6 **else**
7 return [ ]
8 **else**
/*a_{i_j}'s type is a list of tuples */
9 $t_{RES}.a_{i_j} \leftarrow \bigsqcup_{t' \in t.a_{i_j}, t'' \in b.a_{i_j}} t' \cap t''$
10 **if** $t_{RES}.a_{i_j} = [\,]$ **then**
11 return [ ]
12 **foreach** *attribute* $a_j, 1 \leq j \leq k, a_j \notin a_{i_1}, \ldots, a_{i_k}$ **do**
13 $t_{RES}.a_j \leftarrow t.a_j$
14 return $[t_{RES}]$

e1(ID=2, Tag="book",
e2[(Val="Abiteboul"), (Val="Suciu")],
e3[(ID=4, Val="Data on the Web")])

Now consider another binding tuple t_{B2} for χ_7:

t_{B2}=e1(Tag="article", e3[(Val="Data on the Web")])

The information content of χ_4 on document d with the binding list $[t_{B2}]$ is empty, since d does not contain any article called "Data on the Web".

Let t_{B3} be the binding tuple:

t_{B3}=e1(Tag="book", e3[(Val="The Syntactic Web")])

The information content of χ_4 on document d, with the bindings $[t_{B1}, t_{B3}]$ is:

[e1(ID=2, Tag="book",
e2[(Val="Abiteboul"), (Val="Suciu")],
e3[(ID=4, Val="Data on the Web")]),
e1(ID=7, Tag="book",
e2[(Val="Tom Lerners-Bee")],
e3[(ID=12, Val="The Web: next generation")])]

To formalize the above, we need the notion of *tuple intersection*. Let t and b be two tuples such that the signature of b is a projection on the signature of t. Then, $t \cap b$ represents the data accessible from t *given* b; this data can consist of zero or one tuple, containing (possibly part of) the data from t.[3]

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

We illustrate nested tuple intersection with an example.

Consider the following tuple t and binding tuple b_1:

t=e1(ID=2, Tag="book",
e2[(Val="Abiteboul"), (Val="Suciu")],
e3[(ID=4, Val="Data on the Web")])
b_1=e1(ID=2, e2[(Val="Suciu"), (Val="Buneman")])

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

t_{RES}=e1(ID=$\perp$, Tag=$\perp$, e2[ ], e3[ ])

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

t_{RES}=e1(ID=2, Tag=$\perp$, e2[ ], e3[ ])

and then successively into:

t_{RES}=e1(ID=2, Tag=$\perp$,
e2[(Val="Abiteboul") $\cap$ (Val="Buneman") $\bigsqcup$
(Val="Suciu") $\cap$ (Val="Buneman") $\bigsqcup$
(Val="Abiteboul") $\cap$ (Val="Suciu") $\bigsqcup$
(Val="Suciu") $\cap$ (Val="Suciu")],
e3[ ]),

t_{RES}=e1(ID=2, Tag=$\perp$, e2[(Val="Suciu")], e3[ ])

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

t_{RES}=e1(ID=2, Tag="book", e2[(Val="Suciu")],
e3[(ID=4, Val="Data on the Web")])

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

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

DEFINITION 3.8 (RESTRICTED XAM SEMANTICS). Let χ be a XAM with some required attributes, and χ^0 be a XAM obtained from χ by erasing all **R** markers. Let B be a list of binding tuples for χ. The semantics of χ over a document d, with bindings B, is defined as:

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

□

The following example illustrates restricted XAM semantics.

Consider the XAM χ_4 in Figure 6. Erasing all its **R** marks leads to the XAM χ_5 shown next to it. Let d be the document in Figure 2. By Definition 3.7, we have:

$[\![\chi_5]\!]_d$=[e1(ID=2, Tag="book", e2[(Val="Data on the Web")],
e3[(ID=5,Val="Abiteboul"), (ID=6,Val="Suciu")]),
e1(ID=7, Tag="book", e2[(Val="The Syntactic Web")],
e3[(ID=9, Val="Tom Lerners-Bee")]),
e1(ID=10, Tag="phDThesis",
e2[(Val="The Web: next generation")],
e3[(ID=13, Val="Jim Smith")])]

Denoting the three tuples above as t_1, t_2 and t_3, we have $[\![\chi_5]\!]_d = [t_1, t_2, t_3]$. Let B be the following bindings for χ_4:

B=[e1(Tag="book", e3[(Val="Data on the Web")]),
e1(Tag="book", e3[(Val="The Syntactic Web")])]
=$[b_1, b_2]$

Applying Definition 3.8, we obtain:

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

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

4. RELATED WORKS AND PERSPECTIVES

We have presented XML access modules, a formalism with clean algebraic foundations for describing XML storage structures. XAMs are reminiscent of query pattern formalisms, such as the Abstract Tree Patterns [20] or of clustering strategies in object-oriented systems (eg. [4]). However, XAMs are focused on storage modelling, as reflected by their ID specifications, and required fields. . This approach compares most directly to the Agora [16], Mars [7], LegoDB [5] and ShreX [3] projects. We present a formalism with well-defined semantics which, which departs from these previous approaches in that:

- it is based on a nested (as opposed to relational) algebraic model, better suited to XML querying;
- it models important properties of element IDs, with a strong impact on query performance;
- it provides an accurate model for XML indexes, since it allows to specify the fields whose values have to be known (that is, the index key), in order to access the index data.
- it extends the access patterns paradigm to nested data models, thus encompassing complex XML indexes;

One may wonder why we do not describe storage structures by XQuery queries, and apply view-based query rewriting. The main reason is that the notion of XQuery materialized view is not yet clearly defined, since the result of an XQuery is considered a *different (thus, disjoint)* document from its input. Also, features such as interesting ID properties and required fields are not easy to express via XQuery.

How to use XAMs ? Given an XQuery query, the optimizer has to find which XAMs, and in which combination, provide the data that is needed by the query. Notice that XAMs alone cannot capture all the operations that the query may perform: for instance, joins, re-structuring, new element construction etc. have to be performed on top of the accesses to XAM data. This is rightfully so, since XAMs only provide data access, and do not cater to other transformation that the query may apply.

The process of XAM selection for a given query should be driven by a set of constraints describing the input document. Indeed, a XAM containing all title elements can be used to answer a query for //book/title only if we know that no other titles appear in the document. Several classes of constraints can be considered; DTDs or XML Schema constraints are just one possible example.

We are currently devising an algorithm for selecting and combining meaningful XAMs for a query, in the simple case in which constraints are encapsulated in a DataGuide. In a nutshell, a DataGuide can be seen as a compact representation of a set of path constraints which the document satisfies. Our algorithm finds useful XAMs for a query, by unfolding the query and the XAMs based on the path constraints, and then identifying portions of the XAMs that provide data needed by the query. This algorithm, in the particular setting of path constraints, is a close relative of the well-known bucket algorithm from the Information Manifold, enhanced with proper treatment of the **R** annotations. We plan to extend this to other types of constraints, and in particular, to CDuce types (see www.cduce.org).

5. REFERENCES

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *JCSS*, 1986.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Amer-Yahia and Y. Kotidis. Web-services architectures for efficient XML data exchange. In *ICDE*, 2004.

[4] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In *4th Int'l Workshop on Persistent Objects*, 1990.

[5] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[6] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.

[7] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[8] P. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, 1982.

[9] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[10] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. In *IEEE Data Eng. Bull.*, 1999.

[11] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

[12] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB J.*, 11(4), 2002.

[13] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[14] I. Manolescu. XML Query Processing: Storage and Query Model Interplay. EDBT Summer School 2004.

[15] I. Manolescu, V. Benzaken, and A. Arion. XML access modules. Technical report, 2005.

[16] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.

[17] I. Manolescu and Y. Papakonstantinou. An unified tuple-based algebra for XQuery. Available at www-rocq.inria.fr/~manolesc/PAPERS/algebra.pdf.

[18] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[19] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.

[20] S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, 2004.

[21] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[22] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in relational database systems. In *SIGMOD*, 1979.

[23] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

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

[25] XML Query Data Model. http://www.w3.org/TR/query-datamodel, 2005.