1. INTRODUCTION

A key factor for the outstanding success of database management systems is that they provide 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 cornerstone 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 [23].

Surprisingly, physical data independence is not yet achieved by XML database management systems (XDBMSs, in short). Numerous methods have been proposed for XML storage [4, 7, 11, 14, 18, 24], labeling [5, 20] and indexing [6, 15, 19, 26] and implemented in various prototypes. However, the data layout resulting from each of these schemes is hardcoded within the query optimizer of the corresponding system. Thus, adding a different type of storage structure (e.g., a new index) requires rewriting the query optimizer, to inform it that a new access path becomes available. This situation prevents XDBMSs from attaining two important features: flexibility and extensibility. By flexibility, we mean that widely different storage schemes must be supported, for the varying needs of different workloads and data sets. By extensibility, we mean that the XDBMS must adapt gracefully to changes in the workload and/or data set, which naturally require tuning the storage by adding e.g., an index or a materialized view. Such extensibility is a common feature of today’s relational database management systems (RDBMSs), which currently include automatic index and materialized view selection algorithms [2].

We demonstrate ULoad, an XML storage tuning tool, which is a step towards achieving physical data independence for XML. ULoad is meant to help the database administrator (DBA) in choosing the persistent XML storage and indexing modules best suited for a given dataset, and workload, thus achieving our flexibility and extensibility requirements.

Given a document to store, and a query workload, ULoad: (i) allows the DBA to choose, customize, and apply some storage and indexing models, picked among a large set of existing ones; (ii) lets the DBA define her own specialized persistent structures; (iii) presents to the DBA the query execution plans (QEPs) for the workload queries, on the storage model chosen, and their costs; and (iv) may propose a storage adapted to the data and workload, based on a cost-driven search.

At the core of ULoad lies a novel algebraic formalism (with a simple graphical representation), called XML Access Modules (XAMs), describing the information contained in a persistent XML storage structure. XAMs are generic enough to capture many existing storage and indexing schemes, and they have several other innovative features. First and foremost, a set of XAMs is a high-level description of how a document is stored. Based on this description, with clear algebraic foundations, ULoad provides an access path selection algorithm, identifying the storage structures that can be used to answer a given query. When changing a document’s storage, we only need to update the set of XAMs describing it; no change to the optimizer’s code is needed. Second, XAMs capture important properties of persistent XML identifiers, with a crucial impact on the efficiency of XML query and update processing. Finally, XAMs provide an accurate model for XML indexes, since they allow specifying the fields whose values have to be known (that is, the index key), in order to access the index data.

This document is structured as follows. Section 2 describes the ULoad tool functionalities. Section 3 introduces the XAM formalism. Section 4 outlines demonstration scenarios, while Section 5 discusses related works.

2. ULOAD TOOL OUTLINE

ULoad recommends, or assists the DBA in choosing or defining, a persistent storage scheme adapted to a particular application. ULoad does not store XML data; rather, it works in conjunction with an XDBMS, which may be backed by both a relational or a native XML store (see Section 4). ULoad checks and guarantees that the storage chosen is both sufficient, and efficient for the application needs. Once the DBA is satisfied with a given storage, ULoad emits a set of loading directives to the XDBMS, effectively materializing this storage. ULoad offers the choice among a wide variety of existing storage and indexing strategies, as well as the ability to define custom storage structures and indexes.

ULoad needs several inputs. (1) A set of documents to store; in this paper, for simplicity, we consider a single doc-
A set of structural constraints $C$ describing $D$'s structure. In general, such constraints may come from a DTD or XML Schema. To handle XML documents with or without a schema, ULoad considers structural constraints under the form of a path summary, equivalent to a Dataguide [12] for XML data. A path summary is easy to extract from an XML document to update compact, and quite expressive [3]. We plan to extend ULoad to XML type descriptions in our future work. (4) Optionally, access to the XDBMS's cost estimator. ULoad needs to estimate the impact of a candidate storage structure, on the performance of the processing of queries in $W$. This impact is reflected by the optimizer's cost estimation for $W$, assuming the candidate structure was available. Note that, when actually processing the queries in $W$, the XDBMS will rely on the same estimation.

As shown in Figure 1, ULoad allows the user to:

1. Choose a (set of) storage and indexing models from a set of existing ones, such as [4, 6, 11, 14, 24, 26], according to which $D$ will be stored. Different parts of $D$ may be stored according to different models; indices can be selectively built to support $W$. Or, the user may define her own persistent data structures (in the style of materialized views) using a graphical language. The result of this stage is a set of XAMs for $D$; each of them describes a persistent data structure storing some part of $D$ (see Section 3).

2. Check if a set of XAMs is sufficient to answer $W$. ULoad determines this by analyzing the set of XAMs, the structural constraints $C$ and the workload $W$. If the XAMs are insufficient, ULoad points out the (parts of) queries that cannot be answered.

3. Find out the cost of answering $W$ on a set of XAMs. ULoad finds all XAM subsets that may jointly be used to answer each query from $W$, under the constraints $C$. If the XML DBMS cost estimator is available, ULoad calls it to assess the quality of the query plans the DBMS would generate on such storage. Otherwise, ULoad computes its own query plan, applies simple heuristics such as selection pushing and join reordering, and uses its own cost estimations.

4. Obtain a recommended set of XAMs. The DBA may want to get a baseline recommendation, which she can then tune. To that purpose, ULoad applies some efficient heuristics to pick a set of XAMs providing reasonable performance for $D$, $C$ and $W$.

5. Load $D$ in the XDBMS's store following the XAMs for $D$, and run $W$ queries using the XDBMS query engine.

Inside the ULoad box in Figure 1, buttons represent the possible actions of the tool, centered around the set of XAMs for $D$. An arrow connecting an action to the XAMs shows whether the action produces or uses them.

3. XML ACCESS MODULES

An XML Access Module (XAM) corresponds to an XML document fragment stored in a persistent data structure. Formally, a XAM is described as 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.

![Figure 1: Outline of ULoad's functions.](image)

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

$$NS := \top \cdot N^+$$

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

$$\text{IDSpec ::= ID|o|s|p (R)}$$

$$\text{TSpec ::= (Tag (R)| [Tag = c]}$$

$$\text{VSpec ::= (Val (R)| [Val = c]}$$

$$\text{CSpec ::= Cont}$$

$$\text{ES ::= E s}$$

Any XAM specification contains a special node $\top$, corresponding to the document root (that is, the ancestor of all element and attributes in a document). The other nodes represent XML elements or attributes. We associate a name to each XAM node representing elements or attributes, by convention, names starting with $\emptyset$ are used for XAM nodes representing XML attributes.

A node can be annotated by zero or more among: an identifier specification $\text{IDSpec}$, a tag specification $\text{TSpec}$, a value specification $\text{VSpec}$, and a content specification $\text{CSpec}$. By content, we mean the full (serialized) representation of the XML element or attribute. An ID (resp. tag, value, content) specification, attached to a XAM node, denotes the fact that the element/attribute ID (respectively, tag, value, or full textual content) is stored in the XAM.

Node identity is a crucial notion in XQuery processing. Any XML store provides some persistent identifiers; the particular type of IDs used determines the efficiency of 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 [7, 11, 24] are a typical example. We use $s$ to

1 Strictly speaking, attribute nodes can be uniquely identified by the ID of their parent element and the attribute name. We use explicit attribute IDs for simplicity.

2 Clearly, the content of an XML element can always be retrieved from a non-losing storage, by combining accesses to several storage modules structures. In contrast, we use Cont only for the storage models able to retrieve it from a single persistent data structure.
Figure 2: Sample XML document, and XAMs.

designate structural identifiers, which allow to infer, by comparing two element IDs, whether one is a parent/ancestor to the other. They are produced e.g., by the popular (preorder, postorder, depth) labeling schemes based on Dietz’s model [9]. 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 [25], or ORDPATHs [20].

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 [13, 10]. More generally, R symbols allow to model XML indexes: index keys have to be known, in order to perform an index lookup.

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).

Concerning XAM edges, we distinguish parent-child edges, marked as /, from ancestor-descendant edges, marked as //. Furthermore, we distinguish join, left outerjoin, and left semijoin semantics for the XAM edges, considering the parent node on the left hand; these are marked by the symbols j, o, respectively s.

The data from D stored by a XAM is a set (or list) of nested tuples, whose schema follows the XAM specification, and whose content is derived from D. XAMs are defined based on a nested relational model [1], suited to the nature of XML; however, they also support explicit unnesting, to model e.g., relational storage. We illustrate this here by examples; for a formal definition, see [3].

Examples. Consider the XML snippet and XAMs in Figure 2. The XAMs are depicted under a tree graphical form, actually used in the ULoad GUI, following the XAM tree structure. On the left side of begin tags, we show (preorder, postorder, depth) identifiers for the element and its attributes. The preorder number reflects the element position in the document; we will use it as a simple order-preserving ID, when needed. For simplicity, we assume all XAMs ordered.

The XAM \( \chi_d \) stores order-preserving IDs, and text values, of all element objects. On the document in Figure 2, assuming integer IDs, \( \chi_d \) thus stores the tuples:

\[
(ID_1 = 4 \quad Val_1 = "Kay@ny.org") \quad (ID_1 = 10 \quad Val_1 = "Tom@md.com")
\]

The XAM \( \chi_b \) stores parent-child IDs, enabling navigation from a parent element to its element children. On the document in Figure 2, some of the tuples stored by \( \chi_b \) are:

\[
(ID_1 = 1 \quad \{ ID_2 = 2 \quad Tag_2 = "person" \}, \quad ID_2 = 8 \quad Tag_2 = "person" \}, \quad ID_2 = 11 \quad Tag_2 = "person" \})
\]

Notice the nesting of information representing \( \chi_b \) node \( n_2 \), inside tuples representing information of the parent node \( n_1 \). Since the edge \( n_1 \rightarrow n_2 \) has outerjoin semantics, childless nodes (such as the email element numbered 4) appear with an empty list of child tuples. The value of the ID1 attribute must be known, in order to access \( \chi_b \) tuples.

Finally, the XAM \( \chi_e \) stores email children and watch dependents of persons having at least a watch dependent. On the document in Figure 2, \( \chi_e \) stores a single nested tuple:

\[
(ID_1 = [2, 3, 2] \quad \{ ID_2 = [6, 3, 4] \quad Val_2 = "a1" \}, \quad ID_2 = [7, 4, 4] \quad Val_2 = "a2" \}) \quad \{ ID_2 = [4, 2, 3] \quad Val_2 = "Kay@ny.org" \})
\]

Answering queries over XAMs. For a given query and a set of XAMs, ULoad identifies all XAM combinations that may be used to answer the query. For example, consider the query q on the document in Figure 2:

\[
\text{for } \text{sp} \text{ in } \text{/person return } <\text{em}><\text{sp}/email</text/em>>
\]

ULoad will find that q may be answered by: using \( \chi_d \) (with the root ID=1) to get the IDs of the root’s children; testing the children tag to retain person children; using \( \chi_b \) again to find person email IDs; finally, using \( \chi_e \) to obtain the email value. This corresponds to a top-down navigation plan \( ^* \tau_{sp} = \text{email} (\sigma_{\text{Tag}} \text{= email} (\chi_b \bowtie \chi_b) \bowtie \chi_e) \bowtie \chi_e \).

\( \chi_e \) could also be used to answer q, but not alone, because it does not store the emails of users without watch descendents. Thus, ULoad will construct a union plan over \( \chi_e \), and a navigation plan similar to the one above, but restricted to person elements without watch descendents.

Now, let \( \text{q’} \) be: for \( \text{sp} \text{ in } \text{/person return } \{ \text{sp}/\text{id} \} \).

ULoad notices that \( \text{id} \) attributes are not stored, and signals that \( \text{sp}/\text{id} \) (and thus, \( \text{q’} \)) cannot be matched on the XAMs in Figure 2. ULoad will suggest a new XAM, storing the \( \text{id} \) attribute together with person identifiers; this will enable answering \( \text{q’} \).

Figure 3: Sample XAMs \( \chi_d \) and \( \chi_e \).

Now, consider the new XAM \( \chi_d \) shown in Figure 3. For any tag, \( \chi_d \) returns the structural IDs and values of all elements of that tag, in the style of the tag indexes of Timber [13] and Natix [10]. If \( \chi_d \) is available, ULoad will answer \( \text{q} \) using structural join plans on \( \chi_d \), as in [21].

ULoad uses the structural constraints \( \mathcal{C} \), to stop navigation when all email descendents have been found.
As a last example, consider the XAM $\chi_e$ in Figure 3. It represents an index, which allows to retrieve the email de-scendants of any element having an $\Phi d$ attribute; the value of the $\Phi d$ attribute is the index key. This illustrates the ca-pacity of XAMs to express generic XML indexes, including structural and content conditions.

4. DEMONSTRATION SCENARIO

The concepts behind ULoad, in particular the XAM for-malism, apply both to relational storage models, and to na-tive ones. To demonstrate this, and to explore the trade-offs between robust relationa1 stores, and more flexible, but less mature, native ones, we will show ULoad in two settings.

First, we assign the XDBMS role (Figure 1) to Postgres. Each XAM is stored in a table; a clustered index is added, if the XAM has $R$ fields. ULoad takes advantage of Postgres’s data loader, query engine and cost estimates.

Second, we pair ULoad with an experimental native XML data store, built on the persistent storage library BerkeleyDB [27]. In this setting, ULoad uses its own simple cost model, mainly accounting for the number of accesses to BerkeleyDB disk-resident structures.

In both these settings, we plan to demonstrate:

- How to specify XAMs in our graphical language. To demonstrate XAM expressive power, we will show how existing storage and indexing schemes, e.g. [4, 11, 24, 15], can be automatically compiled into XAM sets.
- The data required by an XQuery query over a document $D$; ULoad extracts this under a tree form, similar to a XAM. We show how ULoad uses this form to highlight the parts of the query for which the storage is insufficient (if any).
- The alternative access paths identified by ULoad for each query in the workload, and the resulting QEPs with their cost estimates.
- ULoad’s algorithm for choosing the views and indexes to materialize over a storage model. This algorithm aims at a trade-off between the performance of queries in $\mathcal{W}$, and the storage space taken by views and indexes.
- The performance of data loading and query processing.

5. COMPARISON WITH RELATED WORKS

ULoad’s genericity allows it to express many existing stor-age and indexing schemes; ULoad complements their ben-eﬁts with those of a ﬂexible and extensible storage. The ULoad approach compares most directly to the Agora [17] and Mars [8] projects. Different from these, ULoad: (i) is based on a nested (as opposed to relational) model, better suited to XML querying; (ii) models important properties of element IDs, with a strong impact on query performance; (ii) extends the access patterns paradigm [16, 22] to nested data models, thus encompassing complex XML indexes.

XAMs are reminiscent of query pattern formalisms; the closest one is the Abstract Tree Pattern [21]. However, XAMs are focused on storage modelling, as reﬂected by their ID speciﬁcations, and required ﬁelds. One may wonder why we do not describe storage structures by XQuery queries, and apply view-based query rewriting. One reason is that crucial features of an XML storage, such as persistent IDs and their properties, are not explicitly present in XQuery (nor in XML itself!). Second, the notion of XQuery ma-terialized view is not yet formally deﬁned, since the result of an XQuery is in principle a different (thus, disjoint) document from its input. For more information on XAMs, and a detailed comparison with existing storage models, see [3].

6. REFERENCES