ULoad: Choosing the Right Storage for your XML Application

Andrei Arion[1 2], Véronique Benzaken[2] , Ioana Manolescu[1], Ravi Vijay[1 3]

[1] INRIA Futurs, [2] Univ. Paris XI, [3] IIT Bombay
Andrei.Arion@inria.fr, Veronique.Benzaken@lri.fr, Ioana.Manolescu@inria.fr, Ravi.Vijay@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 [13].

Surprisingly, physical data independence is not yet achieved by XML database management systems (*XDBMS*s, in short). Numerous methods have been proposed for XML storage, labeling and indexing and implemented in various prototypes. However, the data layout resulting from each of these schemes is hard-coded within the query optimizer of the corresponding system. Thus, adding a different type of storage structure (e.g., a new index) requires re-writing the query optimizer, to inform it that a new access path becomes available. This situation prevents XDBMSs from attaining two important features: flexibility and extensibility. By *flexibility*, we mean that widely different storage schemes must be supported, for the varying needs of different workloads and data sets. By *extensibility*, we mean that the XDBMS must adapt gracefully to changes in the workload and/or data set, which naturally require tuning the storage by adding e.g., an index or a materialized view. Such extensibility is a common feature of current relational database management systems (RDBMSs), endowed with automatic index and materialized view selection [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 introduces the XAM formalism. Section 3 describes the ULoad[1] tool functionalities. Section 4 outlines demonstration scenarios, while Section 5 discusses related works.

2 What the DBAs should know about XAMs

In order to interact with ULoad a DBA must have a basic understanding of our storage description language. We present here the information needed for creating and tuning

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

[1]This work is partially funded by the French government research grant ACI TRALALA. The work was performed while Ravi Vijay was visiting INRIA.

a particular storage. We start with a few general considerations and then we illustrate them by an example.

An *XML Access Module* (XAM) corresponds to an XML document fragment stored in a persistent data structure. 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.

A node specification contains an identifier specification $IDSpec$[2], a tag specification $TSpec$, a value specification $VSpec$, and a content specification $CSpec$. By content, we mean the full (serialized) representation of the XML element or attribute.[3]. 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.

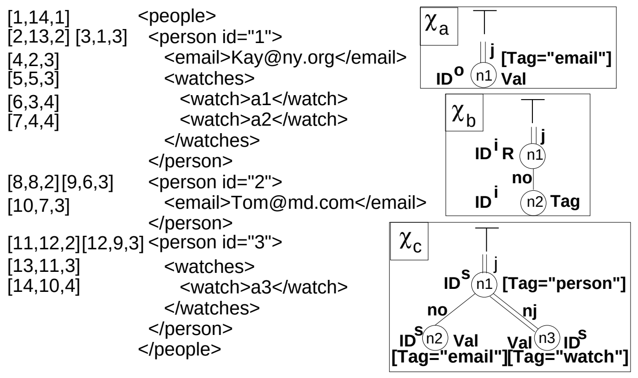
An **R** symbol in an ID specification denotes an *access restriction:* the ID of this XAM node is *required* (must be known) in order to access the data stored in the XAM. This feature is important to model persistent tree storage structures, which enable navigation from a parent node to its children, as in [10, 7]. More generally, **R** symbols allow to model arbitrary XML indexes, on structure and values: key values must be known to perform an index lookup [3].

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 and we omit them for space reasons (for details see [3]).

XAM edges can be either parent-child edges, marked ***/***, from ancestor-descendant edges, marked ***//***. Furthermore, we distinguish *join*, *left outerjoin*, *left semijoin*, *nested join* and *left nested 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**.

The data 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 the stored XML document. XAMs are defined based on a nested relational model [1], but they also support un-nesting, to model e.g., relational storage structure. More details are provided in [3].

Example. Consider the XML snippet and XAMs in Figure 1. 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.



```
[1,14,1]              <people>
[2,13,2] [3,1,3]        <person id="1">
[4,2,3]                   <email>Kay@ny.org</email>
[5,5,3]                   <watches>
[6,3,4]                     <watch>a1</watch>
[7,4,4]                     <watch>a2</watch>
                          </watches>
                        </person>
[8,8,2][9,6,3]          <person id="2">
[10,7,3]                  <email>Tom@md.com</email>
                        </person>
[11,12,2][12,9,3]       <person id="3">
[13,11,3]                 <watches>
[14,10,4]                   <watch>a3</watch>
                          </watches>
                        </person>
                      </people>
```

Figure 1: Sample XML document, and XAMs.

The XAM χ_a stores order-preserving IDs, and text values, of all email elements. On the document in Figure 1, assuming integer IDs, χ_a thus stores the tuples:

(ID$_1$=4 Val$_1$="Kay@ny.org")
(ID$_1$=10 Val$_1$="Tom@md.com")

The XAM χ_b stores parent-child IDs, enabling navigation from parent to children elements. On our sample document, some of the tuples stored by χ_b are:

(ID$_1$=1 [(ID$_2$=2 Tag$_2$="person"), (ID$_2$=8 Tag$_2$="person"), (ID$_2$=11 Tag$_2$="person")])
(ID$_1$=2 [(ID$_2$=3 Tag$_2$="@id")]) (ID$_1$=3 [ ]) (ID$_1$=4 [ ])

Notice the nesting of information representing χ_b node n$_2$, inside tuples representing information of the parent node n$_1$. Since the edge n$_1$-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 ID$_1$ attribute must be known, in order to access χ_b tuples.

Finally, the XAM χ_c stores email children and watch descendants of persons having some watch descendants. On our sample document, χ_c stores one nested tuple:

ID$_1$=[2,13,2]   [(ID$_2$=[6,3,4] Val$_2$="a1"),
(ID$_2$=[7,4,4] Val$_2$="a2")]
[(ID$_3$=[4,2,3] Val$_3$="Kay@ny.org")])

Answering queries over XAMs Given a 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 1:
for \$p in //person return <em>{\$p//email}</em>

ULoad will find that q may be answered by: using χ_b (with the root ID$_1$=1) to get the IDs of the root's children; testing the children tag to retain person children; using χ_b again to find person email IDs; finally, using χ_a to obtain the email value. This corresponds to a top-down navigation plan[4]: $\sigma_{Tag_2=email}(\sigma_{Tag_2=person}(\chi_b \bowtie \chi_b) \rhd\!\!\lhd\!\sqsubset \chi_b) \bowtie \chi_a$.

[2]We use **i** for simple IDs, that only uniquely identify elements, **o** for IDs reflecting document order, **s** for structural identifi ers, which allow to infer, by comparing two element IDs, whether one is a parent/ancestor of the other.

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

[4]ULoad uses the structural constraints $\mathcal{C}$, to stop navigation when all email descendants have been found.

χ_c could also be used to answer q, but not alone, because it does not store the emails of users without watch descendants. Thus, ULoad will construct a union plan over χ_c, and a navigation plan like the one above, but restricted to persons without watch descendants.

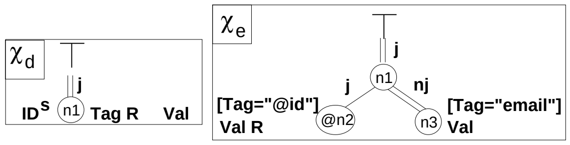


Figure 2: Sample XAMs χ_d and χ_e.

Now, consider the new XAM χ_d shown in Figure 2. For any tag, χ_d returns the structural IDs and values of all elements of that tag, in the style of the tag indexes used in [10, 7]. If χ_d is available, ULoad will answer q using structural join plans on χ_d, as in [12].

Finally, consider the XAM χ_e in Figure 2. It represents an index, which allows to retrieve the email descendants of any element having an @id attribute; the value of the @id attribute is the index key. This illustrates the capacity of XAMs to express generic XML indexes, including structural and content conditions.

Now, let q' be: for \$p in //person return {\$p/@id}.

ULoad notices that @id attributes are not stored, and signals that \$p/@id (and thus, q') cannot be matched on the XAMs in Figure 1. ULoad will suggest a new XAM, storing the @id attribute together with person identifiers.

3 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, backed by a relational, native or mixed 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 XBMS, 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 document D. **(2)** A set of *structural constraints* $\mathcal{C}$ 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 [9] for XML data. A path summary is easy to extract from an XML document, easy to update, compact, and quite expressive [3]. **(3)** An optional query workload $\mathcal{W}$. **(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 the queries in $\mathcal{W}$. This impact is reflected by the optimizer's cost estimation for $\mathcal{W}$, assuming the candidate structure was available. Note that, when actually processing the queries in $\mathcal{W}$, the XDBMS will rely on the same estimation.

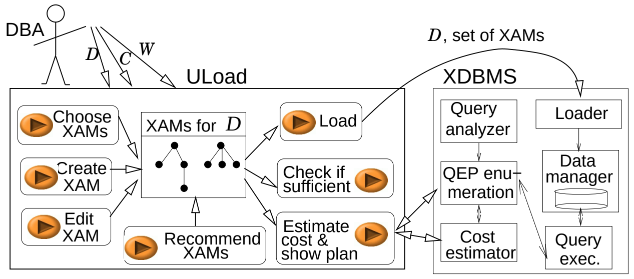


Figure 3: Outline of ULoad's functions.

Based on these inputs, ULoad allows the user to perform several actions, outlined in Figure 3:

- *Choose a (set of) storage and indexing models* from a set of existing ones, such as [5, 8, 14], 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 $\mathcal{W}$. 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 2).
- *Check if a set of XAMs is sufficient to answer $\mathcal{W}$.* ULoad determines this by analyzing the set of XAMs, the structural constraints $\mathcal{C}$ and the workload $\mathcal{W}$. If the XAMs are insufficient, ULoad points out the (parts of) queries that cannot be answered.
- *Estimate the cost of answering $\mathcal{W}$ on a set of XAMs.* ULoad finds all XAM subsets that may jointly be used to answer the $\mathcal{W}$ queries, under the constraints $\mathcal{C}$. If the XDBMS 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.
- *Obtaining 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, $\mathcal{C}$ and $\mathcal{W}$.
- *Loading D* in the XDBMS's store following the XAMs for D, and *run $\mathcal{W}$ queries* using the XDBMS query engine.

In the ULoad box in Figure 3, buttons represent possible tool actions, centered around the set of XAMs for D. An arrow connecting an action to the XAMs shows whether the action produces or uses them.

4 Demonstration scenario

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

First, we assign the XDBMS role (Figure 3) to Postgres. Each XAM is stored in a table; a clustered index is added, if the XAM has **R** fields. ULoad relies on Postgres's loader, query engine and cost estimates.

Second, we pair ULoad with an experimental native XML data store, built on the persistent storage library BerkeleyDB [15]. 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 (relational and native backend storage), 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, 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 views and index storage occupancy.
- Data loading and query processing performance.

5 Comparison with related works

ULoad's genericity allows it to express many existing storage and indexing schemes; ULoad complements their benefits with those of a flexible and extensible storage.

The ULoad approach compares most directly to the Agora [11], Mars [6] and LegoDB [5] projects. Different from these, ULoad:

- is based on a nested (as opposed to relational) algebraic model, better suited to XML querying;
- models important properties of element IDs, with a strong impact on query performance;
- extends the access patterns paradigm to nested data models, thus encompassing complex XML indexes;
- takes explicitly into consideration a generic set of structural constraits and uses these constraints to reason about the equivalence of alternative access paths to the *same* data;
- can combine transparently both nested and flat(relational) storage modules in order to answer a given query (unlike previous works).

XAMs are reminiscent of query pattern formalisms, such as the Abstract Tree Patterns [12]. However, XAMs are focused on storage modelling, as reflected by their ID specifications, and required fields. Also the ULoad approach is reminiscent of clustering strategies in object-oriented systems (eg. [4]).

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 materialized view is not yet clearly defined, since the result of an XQuery is considered a *different (thus, disjoint)* document from its input.

References

[1] S. Abiteboul and N. Bidoit. Non fi rst normal form relations: An algebra allowing data restructuring. *Journal of Computer Systems Science*, 1986.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[3] A. Arion, V. Benzaken, and I. Manolescu. Xml access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[4] Véronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in o_2. In *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects*, pages 403–412, 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 and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[7] 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.

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

[9] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[10] 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.

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

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

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

[14] 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.

[15] The BerkeleyDB library. www.sleepycat.com.