Path Summaries and Path Partitioning in Modern XML Databases

Andrei Arion*, Angela Bonifati†, Ioana Manolescu* and Andrea Pugliese‡
* INRIA Futurs - LRI France, † ICAR CNR - Italy, ‡ University of Calabria - Italy
andrei.arion@inria.fr, bonifati@icar.cnr.it, ioana.manolescu@inria.fr,
apugliese@deis.unical.it

Categories and Subject Descriptors: H.2.4 [Database Management Systems]: Query processing

General Terms: Algorithms

Keywords: XML, path summaries, path partition, XQuery processing

1. MOTIVATION

The performance of XML query processing in persistent XML databases crucially depends on the chosen data access paths, and on the efficiency of the remaining query processing steps, notably navigation (path) query processing, and result construction (or reconstruction, if the data has been shredded).

In this context, we demonstrate that XML path summaries are useful tools for access path selection, and establish efficient algorithms for building and exploiting them. This leads to very efficient processing when used in conjunction with a path-partitioned store, in particular much better than if tag partitioning is used. Furthermore, we devise an efficient method for complex tree reconstruction, with much lower memory needs than existing alternatives. Our algorithms are implemented in the XSum Java library [6].

2. PATH SUMMARIES AND PATH PARTITIONING

Path summaries The *path summary* $PS(D)$ of an XML document D is a tree, having exactly one node for every path in the document D (see Figure 1). Moreover, for any summary nodes x, y such that y is a child of x, we record on the edge x-y whether every node on path x has *exactly one child* on path y (edge x-y labeled 1), or *at least one child* on path y (edge x-y labeled +).

We have established experimentally that path summaries are generally very small (3 to 6 orders of magnitude smaller than the documents), however, exploiting them may still be challenging, unless carefully designed algorithms are used [2]. A path summary is built in $O(N)$ time, using $O(|PS|)$ memory [3]. Our implementation gathers 1 and + labels during summary construction, in $O(N + |PS|)$ time and $O(|PS|)$ memory.

Path-partitioned storage model

Structural identifiers are assigned to each element in an XML document. A direct comparison of two structural identifiers suffices to decide whether the corresponding elements are structurally related (one is a parent or ancestor of the other) or not. A very popular such scheme consists of assigning (pre,post,depth) numbers to every node [1]. The pre number corresponds to the positional number of the element's begin tag, and the post number corresponds to the number of its end tag in the document.

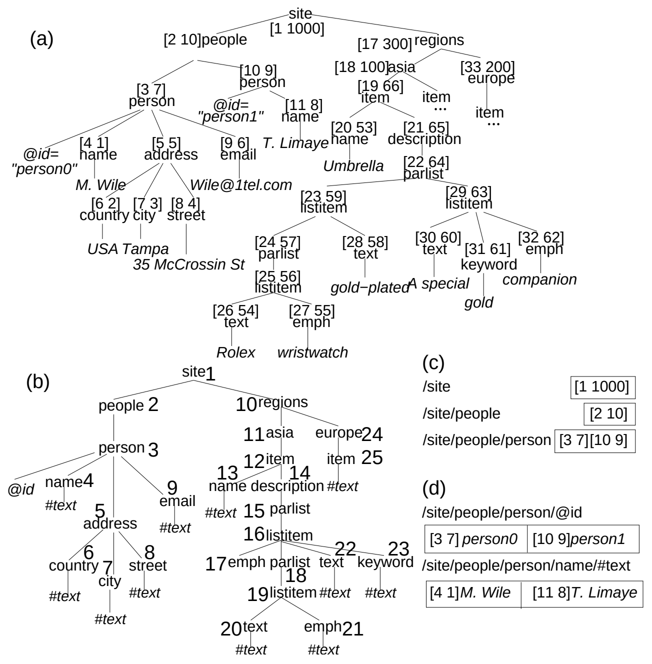

Figure 1: XMark document snippet, its path summary, and some resulting path-partitioned storage structures.

For example, Figure 1(a) depicts (pre,post) IDs next to the elements. The depth number reflects node depth in the document tree (omitted in Figure 1 to avoid clutter).

Based on structural IDs, our first structure contains a compact representation of the XML tree structure. We *partition the identifiers according to the data path* of the elements. For each path, we create an *ID path sequence*, which is the sequence of IDs in document order. Figure 1(c) depicts a few ID path sequences resulting from some paths of the sample document in Figure 1(a). Our second structure stores the contents of XML elements, and values of the attributes. We pair such values to an ID of their closest enclosing element identifier. Figure 1(d) shows some such (ID, value) pair sequences for our sample document.

3. RELEVANT PATH COMPUTATION

The main observation underlying this work is that path summaries provide very good support for access path selection. A path-partitioned storage, moreover, provides robust and selective data access methods (see Section 4).

For instance, for the query //asia//item[description]/name on an XMark [5] document, name elements not belonging to asia/item (such as person names) need not be retrieved. These examples illustrate how ancestor paths, such as //asia, filter descendent paths, separating asian items (relevant) from other items (irrelevant). Descendent paths can also filter ancestor paths.

From an XQuery query, we extract a query pattern, as shown in Figure 2. We distinguish parent-child edges (single lines) from ancestor-descendent ones (double lines). Dashed edges represent optional relationships: the children (resp. descendents) at the lower end of the edge are not required for an element to match the upper end of the edge. Edges crossed by a "[" connect parent nodes with children that must be found in the data, but are not returned by the query, corresponding to navigation steps in path predicates, and in "where" XQuery clauses. We call such nodes *existential*. Boxed nodes are those which must actually be returned by the query. In Figure 2(b), some auxiliary variables $1, $2 and $3 are introduced for the expressions in the return clause, and expressions enclosed in existential brackets [ ].

For pattern node, we compute a *minimal set of relevant paths*. A path p is relevant for node n *iff*: (i) the last tag in p agrees with the tag of n (which may also be *); (ii) p satisfies the structural conditions imposed by the n's ancestors, and (iii) p has descendents paths in the path summary, matching all non-optional descendents of the node. Relevant path sets are organized in a tree, mirroring the relationships between their corresponding nodes.

The paths relevant to nodes of the pattern in Figure 2(b) appear in Figure 2(c). The path 14 for the variable $d has no impact on the query result, because: (i) $d is not required to compute the query result; (ii) it follows from the path summary that every element on path 12 (relevant for $i) has exactly one child on path 14 (relevant to $d). Thus, query evaluation does not need to find bindings for $d, but only $i and $2, and combine them. We say 14 is *useless*.

A path summary may guarantee that every XML element on path 12 has at least one descendent on path 22, if all paths between 12 and 22 are annotated 1 or +. In this case, 22 is a *trivial* path for the existential node $3. If the annotations between 12 and 20 are also 1 or +, path 20 is also trivial. The execution engine does not need to check, on the stored data, which elements on path 12 have descendents on paths 20 and 22: we know they all do. Thus, paths 20 and 22 are discarded from the set of $3.

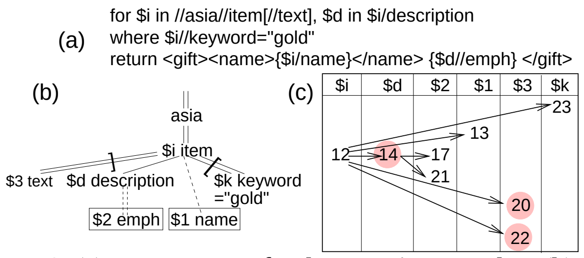

Figure 2: (a): query pattern for the query in Example 1; (b): resulting paths on the document in Figure 1.

XSum [6] provides an efficient algorithm [2] computing minimal path sets in $O(|PS| * |q|^2)$ time, using $O(|PS| * |q|)$ memory.

4. QUERY PROCESSING

Access path selection Given an XML fragmentation model and a path summary, the following generic XML access path selection strategy applies: (i) Compute relevant paths for query pattern nodes. (ii) Compute associated paths for data in every storage structure (table, view, index etc.) (iii) Choose, for every query pattern node, a storage structure whose associated paths form a (tight) superset of the node's relevant paths. In the case of a path-partitioned store, the query plan resulting from this access path selection method is exemplified in Figure 3 for the query in Figure 2. IDs(n) designates an access to the sequence of structural IDs on path n, while IDAndVal(n) accesses the (ID, value) pairs where IDs identify elements on path n, and values are text children of such elements. The left semi-join (⋉) and the left outer-joins (⟕) are *structural*, i.e. they combine inputs based on parent-child or ancestor-descendent relationships between the IDs they contain. This QEP takes good advantage of relevant paths to access only a very small subset of the data present in an XMark document.

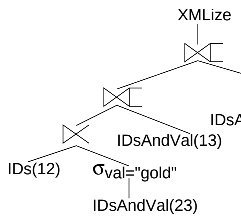

Figure 3: Complete QEP for the query in Figure 2.

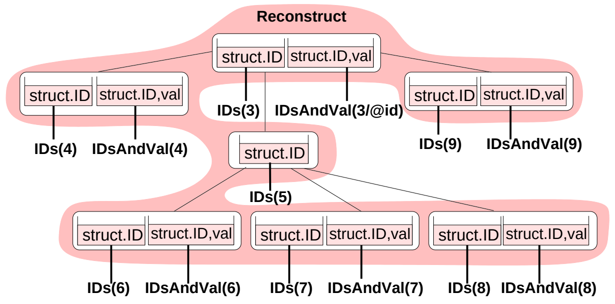

Figure 4: Reconstruct plan for //person on XMark data.

Element (re)construction The biggest performance issues using a path-partitioned store concern the task of reconstructing complex XML subtrees, since the data has been aggressively partitioned.

A first approach is to adapt the SortedOuterUnion [4] method for exporting relational data in XML, to a path-partitioned setting with structural IDs; this is illustrated in Figure 3. In the worst case, the complexity of this method is $O(N * h/B)$ I/O complexity, where B is the blocking factor, and its time complexity is $O(N * h)$.

Based on a path summary and a path-partitioned store, we devised a more efficient approach, using the physical operator *Reconstruct*. This operator reads in parallel the ordered sequences of structural IDs and (ID, value) pairs from all the paths to recombine, and produces textual output in which XML markup (tags) and values are concatenated in the right order [2] (see Figure 2). Interestingly, *Reconstruct does not build intermediary results*, thus it has a smaller memory footprint, namely $O(n)$, where n is the number of paths from which data is combined. For large documents, $n \ll N * h/B$, thus the Reconstruct is particularly competitive.

Our experimental validation, as well as a full comparison with related works on XML query processing, can be found in [2].

5. REFERENCES

[1] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.
[2] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases (full version). www-rocq.inria.fr/˜manolesc, 2006.
[3] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, Athens, Greece, 1997.
[4] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.
[5] The XMark benchmark. www.xml-benchmark.org, 2002.
[6] The XSum library. www-rocq.inria.fr/gemo/XSum, 2005.