Path Summaries and Path Partitioning in Modern XML Databases

Andrei Arion
INRIA Futurs–LRI
France
Andrei.Arion@inria.fr

Angela Bonifati
ICAR CNR
Italy
bonifati@icar.cnr.it

Ioana Manolescu
INRIA Futurs–LRI
France
Ioana.Manolescu@inria.fr

Andrea Pugliese
University of Calabria
Italy
apugliese@deis.unical.it

ABSTRACT

We study the applicability of XML path summaries in the context of current-day XML databases. We find that summaries provide an excellent basis for optimizing data access methods, which furthermore mixes very well with path-partitioned stores. We provide practical algorithms for building and exploiting summaries, and prove its benefits through extensive experiments.

1. INTRODUCTION

Path summaries are classical artifacts for semistructured and XML query processing, dating back to 1997 [16]. From path summaries, the concept of path indexes has derived naturally [27]: the IDs of all nodes on a given path are clustered together and used as an index. Paths have also been used as a natural unit for organizing not just the index, but the store itself [7, 9, 19, 38], and as a support for statistics [2, 23].

The state of the art of XML query processing advanced significantly since path summaries were first proposed. Structural element identifiers [3, 28, 31] and structural joins [3, 11, 32] are among the most notable new techniques, enabling efficient processing of XML navigation as required by XPath and XQuery.

In this paper, we make the following contributions to the state of the art on path summaries:

- We study their size and efficient encoding for a variety of XML document, including very "hard" cases for which it has never been considered before. We show summaries are feasible, and useful, even for such extreme cases.
- We describe an efficient method of static query analysis based on the path summary, enabling a query optimizer to smartly select its data access methods. Similar benefits are provided by a schema, however, summaries apply even in the frequent case when schemas are not available [26].
- We show how to use the result of the static analysis to lift one of the outstanding performance hurdles in the processing of physical plans for structural pattern matching, a crucial operation in XPath and XQuery [37]: duplicate elimination.
- We describe time- and space-efficient algorithms, implemented in a freely available library, for building and exploiting summaries. We argue summaries are too useful a technique for a modern XML database system *not* to use it.

Our XSum library is available for download [36]. It has been successfully used to help manage XML materialized views [5], and as a simple GUI in a heterogeneous information retrieval context [1]. We anticipate it will find many other useful applications.

WWW2006, May 22–26, 2006, Edinburgh, UK.

Path summaries have been often investigated in conjunction with path indexes and path-partitioned stores. It is thus legitimate to wonder whether path partitioning is still a valid technique the current XML query processing context ?

With respect to the path partitioning storage approach, our work makes the following contributions:

- We show that path partitioning mixes well with recent, efficient structural join algorithms, and in particular enables very selective data access, when used in conjunction with a path summary.
- A big performance issue, not tackled by earlier path-partitioned stores [19, 25, 38], concerns document reconstruction, complicated by path fragmentation. We show how an existing technique for building new XML results can be adapted to this problem, however, with high memory needs and blocking execution behavior. We propose a new reconstruction technique, and show that it is faster, and most importantly, has an extremely small memory footprint, demonstrating thus the practical effectiveness of path partitioning.

This paper is organized as follows. Section 2 presents path summaries and a generic path-partitioned storage model. Section 3 tackles efficient static query analysis, based on path summaries. Section 4 applies this to efficient query planning, and describes our efficient approach for document reconstruction. Section 5 is our experimental study. We then discuss related works and conclude.

2. PATH SUMMARIES AND PATH PARTITIONING

This section introduces XML summaries, and path partitioning.

2.1 Path summaries

The *path summary* $PS(D)$ of an XML document D is a tree, whose nodes are labeled with element names from the document. The relationship between D and $PS(D)$ can be described based on a function $\phi : D \rightarrow PS(D)$, recursively defined as follows:

1. ϕ maps the root of D into the root of $PS(D)$. The two nodes have the same label.
2. Let $child(n, l)$ be the set of all the l-labeled XML elements in D, children of the XML element n. If $child(n, l)$ is not empty, then $\phi(n)$ has a unique l-labeled child n_l in $PS(D)$, and for each $n_i \in child(n, l)$, $\phi(n_i)$ is n_l.
3. Let $val(n)$ be the set of #PCDATA children of an element $n \in D$. Then, $\phi(n)$ has an unique child n_v labeled $\#text$, and furthermore, for each $n_i \in val(n)$, $\phi(n_i) = n_v$.
4. Let $att(n, a)$ be the value of the attribute named a of element $n \in D$. Then, $\phi(n)$ has an unique child n_a labeled $@a$, and for each $n_i \in att(n, a)$, we have $\phi(n_i) = n_a$.

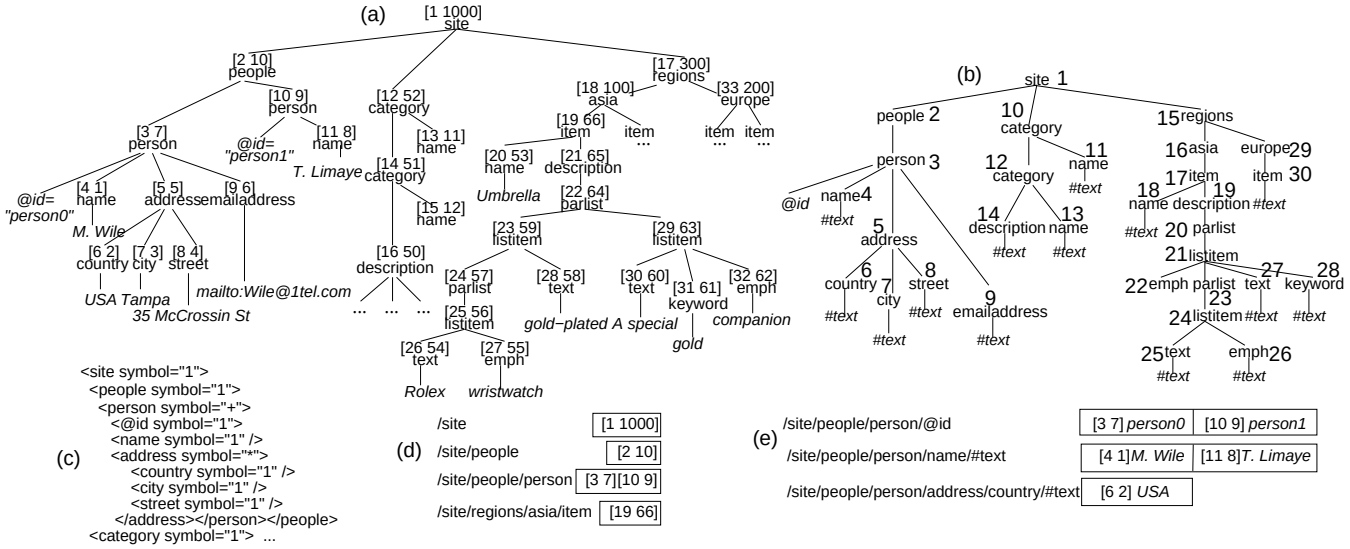


Figure 1: XMark document snippet, its path summary, and some path-partitioned storage structures.

Clearly, ϕ preserves node labels, and parent-child relationships. For every simple path $/l_1/l_2/.../l_k$ in D, there is exactly one node reachable by the same path in $PS(D)$. Conversely, each node in $PS(D)$ corresponds to a simple path in D.

Figure 1(b) shows the path summary for the XML fragment at its left. Path numbers appear in large font next to the summary nodes.

We add to the path summary some more information, conceptually related to schema constraints. More precisely, 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, or *at least one child* on path y, or may lack y children. This information is used for query optimization, as Section 3 will show.

Direct encoding. Let x be the parent of y in the path summary. A simple way to encode the above information is to annotate y, the child node, with: `1` *iff* every node on path x has exactly one child on path y; `+` *iff* every node on path x has at least one child on path y, and some node on path x has several children on path y. This encoding is simple and compact. However, if we need to know how many *descendents* on path z can a node on path x have, we need to inspect the annotations of all summary nodes between x and z.

Pre-computed encoding. Starting from the `1` and `+` labels, we compute more refined information, which is then stored in the summary, while the `1` and `+` labels are discarded.

We identify clusters of summary nodes connected between them only with `1`-labeled edges; such clusters form a *1-partition* of the summary. Every cluster of the 1-partition is assigned a `n1` label, and this label is added to the serialization of every path summary node belonging to that cluster. Then, a node on path x has exactly one descendent on path z *iff* x.`n1`=z.`n1`. We also build a *+-partition* of the summary, aggregating the 1-partition clusters connected among them only by `+` edges, and similarly produce `n+` labels, which allow to decide whether nodes on path x have at least one descendent on path z by checking whether x.`n+`=z.`n+`.

Building and storing summaries. For a given document, let N denote its size, h its height, and $|PS|$ the number of nodes in its path summary. In the worst case, $|PS| = N$, however our analysis in Table 1 demonstrates that this is not the case in practice. The documents in Table 1 are obtained from [34], except for the XMark$n$ documents, which are generated [35] to the size of n MB, and two DBLP snapshots from 2002 and 2005 [15].

A first remark is that for all but the TreeBank document, the summary has at most a few hundreds of nodes, and is 3 to 5 orders of magnitude smaller than the document. A second remark is that as the XMark and DBLP documents grow in size, their respective summaries grow very little. Intuitively, the structural complexity of a document tends to level up, even if more data is added, even for complex documents such as XMark, with 12 levels of nesting, recursion etc. A third remark is that TreeBank, although not the biggest document, has the largest summary (also, the largest we could find for real-life data sets). TreeBank is obtained from natural language, into which tags were inserted to isolate parts of speech. While we believe such documents are rare, robust algorithms for handling such summaries are needed, if path summaries are to be included in XML databases.

A path summary is built during a single traversal of the document, in $O(N)$ time, using $O(|PS|)$ memory [2, 16]. Our implementation gathers `1` and `+` labels during summary construction, and traverses the summary again if the pre-computed encoding is used, making for $O(N + |PS|)$ time and $O(|PS|)$ memory. This linear scaleup is confirmed by the following measures, where the summary building times t are scaled to the time for XMark11:

XMark$n$	XMark2	XMark11	XMark111	XMark233
$n/11$	0.20	1.0	9.98	20.02
$t/t_{11\ \mathrm{Mb}}$	0.32	1.0	8.58	15.84

Once constructed, a summary must be stored for subsequent use. To preserve the summary's internal structure, we will store it as a tree, leading to $O(|PS|)$ space occupancy if the basic encoding is used, and $O(|PS| * log_2|PS|)$ if the pre-computed encoding is used (`n1` and `n+` labels grow in the worst case up to N). We evaluate several summary serialization strategies in Section 5.

2.2 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 [3, 14, 17]. The `pre` number corresponds to the positional number of the element's begin tag, and the `post`

Doc.	Shakespeare	Nasa	Treebank	SwissProt	XMark11	XMark111	XMark233	DBLP (2002)	DBLP (2005)
Size	7.5 MB	24 MB	82 MB	109 MB	11 MB	111 MB	233 Mb	133 MB	280 MB
N	179,690	476,645	2,437,665	2,977,030	206,130	1,666,310	4,103,208	3,736,406	7,123,198
$\|PS\|$	58	24	338,738	117	536	548	548	145	159
$\|PS\|/N$	3.2*10^{-4}	5.0*10^{-5}	1.3*10^{-1}	3.9*10^{-5}	2.4*10^{-3}	3*10^{-4}	1.3*10^{-4}	3.8*10^{-5}	2.2*10^{-5}

Table 1: Sample XML documents and their path summaries.

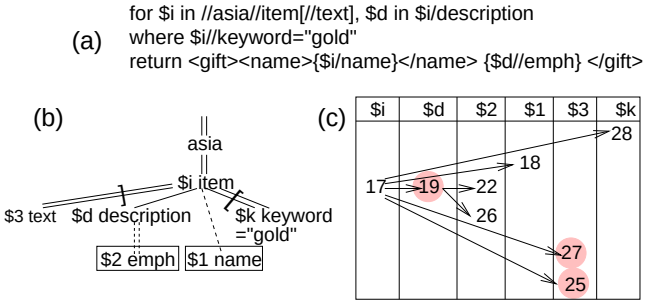


Figure 2: (a): query pattern for the query in Example 1; (b): resulting paths on the document in Figure 1.

number corresponds to the number of its end tag in the document. For example, Figure 1(a) depicts (pre,post) IDs above the elements. The depth number reflects the element's depth in the document tree (omitted in Figure 1 to avoid clutter). Many variations on the (pre,post,depth) scheme exist, and more advanced structural IDs have been proposed, such as DeweyIDs [31] or ORDPATHs [28].While we use (pre,post) for illustration, the reader is invited to keep in mind that any structural ID scheme can be used.

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(d) 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(e) shows some such (ID, value) pair sequences for our sample document.

3. COMPUTING PATHS RELEVANT TO QUERY NODES

An important task of a query optimizer is *access method selection*: given a set of stored data structures (such as base relations, indexes, or materialized views) and a query, find the data structures which may include the data that the query needs to access. An efficient access method selection process requires:

- a store providing selective access methods;
- an optimizer able to correctly identify such methods.

The main observation underlying this work is that path summaries provide very good support for the latter; we explain the principle in Section 3.1 and provide efficient algorithms supporting it in Section 3.2. A path-partitioned storage, moreover, provides robust and selective data access methods (see Section 4).

3.1 The main idea

Given an XQuery query q, the optimizer must identify all data structures containing information about any XML node n that must be accessed by the execution engine when evaluating q. In practice, the goal is to identify structures containing *a tight superset* of the data strictly needed, given that the storage usually does not contain a materialized view for any possible query.

Paths provide a way of specifying quite tight supersets of the nodes that query evaluation needs to visit.

For instance, for the query //asia//item[description]/name, given the summary in Figure 1, elements on paths 17 must be returned, therefore data from paths 18 to 28 may need to be retrieved. Query evaluation does not need to inspect elements from other paths. For instance, paths 4 and 11 are not relevant for the query, even though they correspond to name elements; similarly, item elements on path 30 are not relevant. These examples illustrate how ancestor paths, such as //asia (16) filter descendent paths, separating 17 (relevant) from 30 (irrelevant). Descendent paths can also filter ancestor paths. For instance, DBLP contains article, journal, book elements etc. The query //*[inproceedings] must access /dblp/article elements, but it does not need to access /dblp/journal or /dblp/book elements, since they never have inproceedings children.

Let us consider the process of gathering, based on a path summary, the relevant data paths for a query. We consider the downward, conjunctive XQuery subset from [13]. Every query yields a query pattern in the style of [13]. Figure 2 depicts an XQuery query (a), and its pattern (b). 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 every node in the pattern, 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 structure, mirroring the relationships between the nodes to which they are relevant in the pattern.

The paths relevant to nodes of the pattern in Figure 2(b) appear in Figure 2(c). The paths surrounded by grey dots are relevant, but not part of the minimal relevant sets, since they are either *useless "for" variable paths*, or *trivial existential node paths*.

Useless "for" variable path. The path 19 for the variable $d, although it satisfies condition 1, has no impact on the query result, on a document described by the path summary in Figure 1. This is because: (i) $d is not required to compute the query result; (ii) it follows from the path summary that every element on path

17 (relevant for $i) has exactly one child on path 19 (relevant to $d). This can be seen by checking that 19 is annotated with a 1 symbol. Thus, query evaluation does not need to find bindings for $d. Instead, it suffices to bind $i and $2 to the correct paths and combine them, shortcircuiting the binding of $d.

In general, a path p_x relevant for a "for" variable $x is useless as soon as the following two conditions are met:

1. $x, or path expressions starting from $x, do not appear in a "return" clause.
2. If $x has a parent $y in the query pattern, let p_y be the path relevant for $y, ancestor of p_x. Then, all summary nodes on the path from some child of p_y, down to p_x, must be annotated with the symbol 1. If, on the contrary, $x does not have a parent in the query pattern, then all nodes from the root of the path summary to p_x must be annotated with 1.

Such a useless path p_x is erased from its path set. If $x had a parent $y in the pattern, then there exists a path p_y, ancestor of p_x, relevant for $y. If $x has some child $z in the pattern, in the final solution, an arrow will point directly from p_y to the paths relevant for p_z, shortcircuiting p_x. In Figure 2, once 19 is found useless, 17 will point directly to the paths 22 and 26 in the relevant set for $2.

Trivial existential node paths. The path summary in Figure 1 guarantees that every XML element on path 17 has at least one descendent on path 27. This is shown by the 1 or + annotations on all paths between 17 and 27. In this case, we say 27 is a trivial path for the existential node $3. If the annotations between 17 and 25 are also 1 or +, path 25 is also trivial. The execution engine does not need to check, on the actual data, which elements on path 17 actually have descendents on paths 25 and 27: we know they all do. Thus, paths 25 and 27 are discarded from the set of **$3**.

In general, let p_x be a path relevant for an existential node $x; this node must have had a parent or ancestor $y in the pattern, such that the edge going down from $y, on the path connecting $y to $x, was marked by a "[". There must be a path p_y relevant for $y, such that p_y is an ancestor of p_x. We say p_x is a trivial path if the following conditions hold:

1. All summary nodes between y and x are annotated with either 1 or +.
2. All paths descendent of p_x, and relevant for nodes below $x in the query pattern, are trivial.
3. No value predicate is applied on $x or its descendents.

After pruning out useless and trivial paths, nodes left without any relevant path are eliminated; the connecteed paths of the remaining nodes are returned. For the query pattern in Figure 2(b), this yields exactly the result in Figure 2(c) from which the grey-dotted paths, and their pattern nodes, have been erased.

3.2 Computing relevant paths

Having defined minimal sets of relevant paths, the question is how to efficiently compute them. This problem has not been tackled before. Moreover, trivial algorithms (such as string matching of paths) do not apply, due to the complex tree structure of query patterns, and to the tree-structured connections between relevant paths. Such methods cannot minimize path sets, either.

A straightforward method is a recursive parallel traversal of PS and q, checking ancestor conditions for a path to be relevant for a pattern node during the descent in the traversal. When a path p satisfies the ancestor conditions for a pattern node n, the summary subtree rooted in p_n is checked for descendent paths corresponding to the required children of n in the pattern. This has the drawback

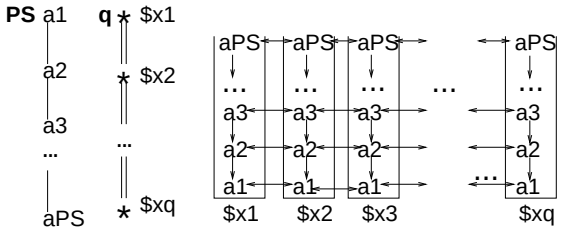


Figure 3: Sample query pattern and relevant path sets.

of visiting a summary node more than once. For instance, consider the query //asia//parlist//listitem: on the summary in Figure 1, the subtree rooted at path 24 will be traversed once to check descendents of path 20, and once to check descendents of the path 23.

A more efficient method consists of performing a single traversal of the summary, and collecting potentially relevant paths, which satisfy the ancestor path constraints, but not necessarily (yet) the descendent path constraints. When the summary subtree rooted at a potentially relevant path has been fully explored, we check if the required descendent paths have been found during the exploration. Summary node annotations are also collected during the same traversal, to enable identification of useless and trivial paths.

The total size of the relevant path sets may be quite important, as illustrated in Figure 3. Here, any subset of $|q|$ nodes of PS contains one path relevant for every node of q, leading to a cumulated size of $|q|! * (|PS| - |q|)!/|PS|!$ relevant paths. This is problematic with large summaries: relevant path identification is just an optimization step, and should not consume too much memory, especially in a multi-user, multi-document database. Therefore, a compact encoding of relevant path sets is needed.

The single-traversal algorithm described above may run on an in-memory de-serialized summary. A more efficient alternative is to traverse the summary in streaming fashion, using only $O(h)$ memory to store the state of the traversal. The algorithm we propose to that effect is shown in Algorithm 1; it runs in two phases.

Phase 1 (finding relevant paths) performs a streaming traversal of the summary, and applies Algorithm 2 whenever entering a summary node, and Algorithm 3 when leaving the node. Algorithm 1 uses one stack for every pattern node, denoted $stack(n)$. Potentially relevant paths are gathered in stacks, and eliminated when they are found irrelevant, useless or trivial. An entry in $stacks(n)$ consists of:

- A *path* (in fact, the path number).
- A *parent* pointer to an entry in the stack of n's parent, if n has a parent in the pattern, and *null* otherwise.
- A *selfparent* pointer. This points to a previous entry on the same stack, if that entry's path number is an ancestor of this one's, or *null* if such an ancestor does not exist at the time when the entry has been pushed. Self-pointers allow to compactly encode relevant path sets.
- An *open* flag. This is set to *true* when the entry is pushed, and to *false* when all descendents of p have been read from the path summary. Notice that we cannot afford to pop the entry altogether when it is no longer open, since we may need it for further checks in Algorithm 3 (see below).
- A set of *children* pointers to entries in n's children's stacks.

Figure 3 outlines the content of all stacks after relevant path sets have been computed for q. Horizontal arrows between stack entries represent *parent* end *children* pointers; downward vertical arrows represent *selfparent* pointers, which we explain shortly.

Algorithm 1: Finding minimal relevant path sets

```
   Input  : query pattern q
   Output: the minimal set of relevant paths paths(n) for
           each pattern node n
   /*          Phase 1: finding relevant paths          */
   /* Create one stack for each pattern node:           */
1  foreach pattern node n do
2      stacks(n) ← new stack
3  currentPath ← 0
4  Traverse the path summary in depth-first order:
5  foreach node n visited for the first time do
6      Run algorithm beginSummaryNode
7  foreach node n whose exploration is finished do
8      Run algorithm endSummaryNode
   /*          Phase 2: minimizing relevant path sets   */
9  foreach node n in q do
10     foreach stack entry se in stacks(n) do
11         if n is existential and
12         all1or+(se.parent.path, se.path) then
13             se is trivial. Erase se and its descendants from
               the stack.
14         if n is a "for" var. and n and its desc. are not
           boxed and all1(se.parent.path,se.path) then
15             se is useless. Erase se from stacks(n) and
16             connect se's parent to se's children, if any
17     paths(n)← paths in all remaining entries in stacks(n)
```

Algorithm 2: **beginSummaryNode**

```
   Input: current path summary node labeled t
   /* Uses the shared variables currentPath, stacks     */
1  currentPath ++;
   /* Look for pattern query nodes which t may match:   */
2  foreach pattern node n s.t. t matches n's label do
       /* Check if the current path is found in the correct
          context wrt n:                                 */
3      if (1) n is the topmost node in q, or (2) n has a parent
       node n', stacks(n') is not empty, and stacks(n').top is
       open then
4          if the level of currentPath agrees with the edge
           above n, and with the level of stacks(n').top then
               /* The current path may be relevant for n, so
                  create a candidate entry for stacks(n):  */
5              stack entry se ← new entry(currentPath)
6              se.parent ← stacks(n').top
7              if stacks(n) is not empty and stacks(n).top is
               open then
8                  se.selfParent ← stacks(n).top
9              else
10                 se.selfParent ← null
11             se.open ← true
12             stacks(n).push(se)
```

In Algorithm **beginSummaryNode**, when a summary node (say p) labeled t starts, we need to identify pattern query nodes n for which p may be relevant. A first necessary condition concerns the final tag in p: it must be t or $*$ in order to match a t-labeled query node. A second necessary condition concerns the *context* in which p is encountered: at the time when traversal enters p, there must be an open, potentially relevant path for n's parent is an ancestor of p. This can be checked by verifying that there is an entry on the stack of n', and that this entry is *open*. If n is the top node in the pattern, if it should be a direct child of the root, then so should p.If both conditions are met, an entry is created for p, and connected to its parent entry (lines 5-6).

The *selfparent* pointers, set at the lines 7-10 of Algorithm 2, allow sharing children pointers among entries nodes in the same stack. For instance, in the relevant node sets in Figure 3, node `a1` in the stack of `$x1` only points to `a1` in the stack of `$x2`, even though it should point also to nodes `a2`, `a3`, ..., `aPS` in the stack of `$x2`, given that these paths are also in descendent-or-self relationships with `a1`. The information that these paths are children of the `a1` entry in the stack of `$x1` is implicitly encoded by the *selfparent* pointers of nodes further up in the `$x1` stack: if path `a3` is a descendent of the `a2` entry in this stack, the `a3` is implicitly a descendent of the `a1` entry also.

This stack encoding via *selfparent* is inspired from the Holistic Twig Join [11]. The differences are: (i) we use it when performing a single streaming traversal over the summary, as opposed to joining separate disk-resident ID collections; (ii) we use it on the summary, at a smaller scale, not on the data itself. However, as we show in Section 5, this encoding significantly reduces space consumption in the presence of large summaries. This is important, since real-life systems are not willing to spend significant resources for optimization. In Figure 3, based on *selfparent*, the relevant paths are encoded in only $O(|q| * |PS|)$. Our experimental evaluation in Section 5 shows that this upper bound is very relaxed.

In line 11 of Algorithm 2, the new entry se is marked as *open*, to signal that subsequent matches for children of n are welcome, and pushed in the stack.

Algorithm **endSummaryNode**, before finishing the exploration of a summary node p, checks and may decide to erase the stack entries generated from p. A stack entry is built with p for a node n, when p has all the required *ancestors*. However, **endSummaryNode** still has to check whether p had all the required *descendents*. Entry se must have at least one *child* pointer towards the stacks of all required children of n; otherwise, se is not relevant and is discarded. In this case, its descendent entries in other stacks are also discarded, if these entries are not indirectly connected (via a *selfparent* pointer) to an ancestor of se. If they are, then we connect them directly to *se.selfparent*, and discard only se (lines 4-9).

The successive calls to **beginPathSummaryNode** and **endPathSummaryNode** lead to entries being pushed on the stacks of each query node. Some of these entries left on the stacks may be trivial or useless; we were not able to discard them earlier, because they served as "witnesses" that validate their parent entries (check performed by Algorithm 3).

Phase 2 (minimizing relevant path sets) in Algorithm 1 goes over the relevant sets and prunes out the trivial and useless entries. The predicate **all1(p_x, p_y)** returns true if all nodes between p_x and p_y in the path summary are annotated with `1`. Similarly, **all1or+** checks if the symbols are either `1` or `+`. Useless entries are "short-circuited", just like Algorithm 3 did for irrelevant entries. At the end of this phase, the entries left on the stack are the minimal relevant path set for the respective node.

Evaluating **all1** and **all1or+** takes constant time if the precomputed encoding is used (Section 2). With the basic encoding, Phase 2 is actually a second summary traversal (although for readability, Algorithm 1 does not show it this way). For every p_x and p_y such that Phase 2 requires evaluating **all1(p_x, p_y)** and

Algorithm 3: **endSummaryNode**

```
Input: current path (node in the path summary),
       labeled t
/* Uses the shared variables currentPath, stack */
1  foreach query pattern node n s.t. stacks(n) contains an
   entry se for currentPath do
     /* Check if currentPath has descendents in the
        stacks of non-optional n children:          */
2    foreach non-optional child n' of n do
3      if se has no children in stacks(n') then
4        if se.ownParent ≠ null then
5          connect se children to se.ownParent
6          pop se from stacks(n)
7        else
8          pop se from stacks(n)
9          pop all se descendent entries from
           their stack
10   se.open ← false
```

all1or+(p_x, p_y), the second summary traversal verifies the annotations of paths from p_x to p_y, using constant memory.

Overall time and space complexity. The time complexity of Algorithm 1 depends linearly on $|PS|$. For each path, some operations are performed for each query pattern node for which the path may be relevant. In the worst case, this means a factor of $|q|$. The most expensive among these operations, is checking that an entry had at least one child in a set of stacks. If we cluster an entry's children by their stack, this takes at most $|q|$ steps. Putting these together, we obtain $O(|PS| * |q|^2)$ time complexity. The space complexity in $O(|PS| * |q|)$ for encoding the path sets.

4. QUERY PLANNING AND PROCESSING BASED ON RELEVANT PATH SETS

We have shown how to obtain for every query pattern node n, a set of relevant paths *paths(n)*.

No matter which particular fragmentation model is used in the store, it is also possible to compute the paths associated to every storage structure, view, or index. For example, assume a simple collection of structural identifiers for all elements in the document, such as the Element table in [38] or the basic table considered in [32]: the path set associated to such a structure includes all PS paths. As another example, consider an index grouping structural IDs by the element tags, as in [20] or the LIndex in [25]: the path set associated to every index entry includes all paths ending in a given tag. A path index such as PIndex [25] or a path-partitioned store [9] provides access to data from one path at a time.

Based on this observation, we recommend the following simple access path selection strategy:

- Compute relevant paths for query pattern nodes.
- Compute associated paths for data in every storage structure (table, view, index etc.)
- Choose, for every query pattern node, a storage structure whose associated paths form a (tight) superset of the node's relevant paths.

This general strategy can be fitted to many different storage models. Section 4.1 explores it for the particular case of a path-partitioned storage model. Section 4.2 and 4.3 show how path information may simplify the physical algorithms needed for structural join processing, respectively, complex output construction.

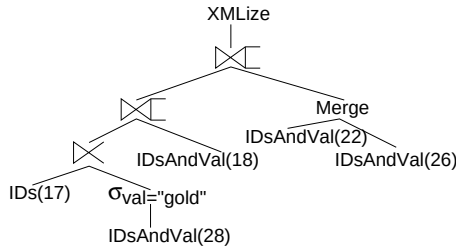


Figure 4: Complete QEP for the query in Figure 2.

4.1 Constructing query plans on a path-partitioned store

With a path-partitioned store, IDs and/or values from every path are individually accessible. In this case, the general access method selection approach becomes: (i) construct access plans for every query pattern node, by merging the corresponding ID or value sequences (recall the logical storage model from Figure 1); (ii) combine such access plans as required by the query, via structural joins, semijoins, and outerjoins. To build a complete query plan (QEP), the remaining steps are: (iii) for every relevant path p_{ret} of an expression appearing in a "return" clause, reconstruct the subtrees rooted on path p_{ret}; (iv) re-assemble the output subtrees in the new elements returned by the query. For example, Figure 4 depicts a QEP for the sample query from Figure 2. In this QEP, 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 ($\ltimes$) and the left outer-joins ($\ltimes\!\!\sqsubset$) are *structural*, i.e. they combine inputs based on parent-child or ancestor-descendent relationships between the IDs they contain. Many efficient algorithms for structural join exist [3, 11]; we are only concerned here with the logical operator.

The plan in Figure 4 is directly derived from the relevant path sets, shown at the end of Section 3.1, and the query itself. The selection σ has been taken from the query, while the Merge fuses the information from the two relevant paths for $2 (emph elements). The final XMLize operator assembles the pieces of data in a result. Section 4.3 studies this in more details.

The QEP in Figure 4 takes good advantage of relevant paths to access only a very small subset of the data present in an XMark document. For instance, only asian item data is read (not from all items), and only the useful fragments of this data; for instance, voluminous item descriptions do not need to be read.

Clearly, more elaborate index structures such as the F&B index [21] may lead to accessing even less data, providing e.g. direct access only to those items (path 17) that have keywords (path 28). However, even an F&B index cannot provide item IDs with their *optional* emph children (paths 22 and 26), because we are interested also in items that *do not* have such children, while the F&B index and its variants correspond to *conjunctive* paths only (all items are required). Furthermore, the selection σ still needs to be applied on on keywords, and path indexes cannot help here.

The conclusion we draw is: while ellaborate structure indexing schemes can cover more complex path patterns, the simplicity and robustness of a simple path-driven access to IDs, combined with the ability to do *most of query processing just by efficient structural ID joins*, provide good support for efficient query evaluation in current-day XML database engines.

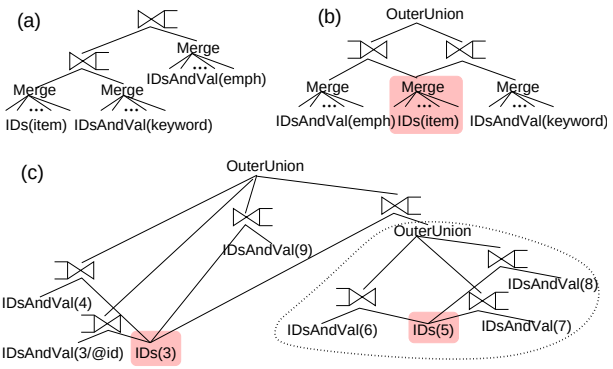


Figure 5: Sample outer-union QEPs with structural joins.

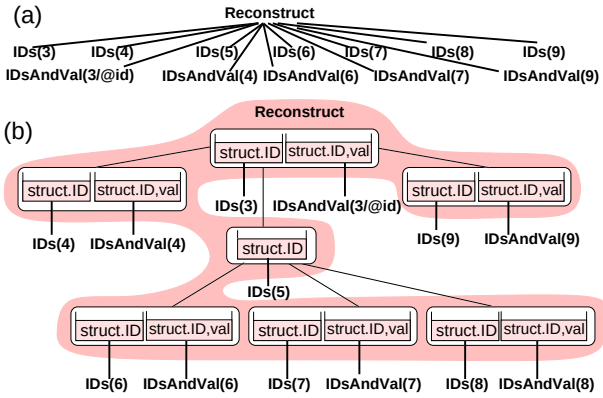


Figure 6: Reconstruct plan for //person on XMark data.

4.2 Using path annotations for efficient structural joins plans

Path expressions used in XPath and XQuery need to return duplicate-free lists of nodes, in specific orders. Structural joins, in contrast, may introduce spurious duplicates, whose elimination is expensive. Schema-based techniques have been developed to decide when duplicate elimination is unnecessary [18]. However, schemas are often unavailable [26]. In this section, we show that even in that case, path summaries enable similar reasoning.

Let op_1, op_2 be two operators, such that $op_1.X$ and $op_2.Y$ contain structural identifiers. The outputs of op_1 and op_2 are ordered by the document order reflected by X, resp. Y, and are assumed duplicate-free. Assume we need to find the $op_2.Y$ IDs that are descendents of some $op_1.X$ IDs. If an ID y_0 from $op_2.Y$ has two different ancestors in $op_1.X$, the result of the structural join $op_1 \bowtie op_2$ will contain y_0 twice, and (depending on the physical algorithm employed [3]) the output may not be in document order, requring explicit Sort and duplicate-elimination.

Path annotations provide a sufficient condition which ensures every $op_2.Y$ ID has at most one ancestor in $op_1.X$: *for any two possible paths p_1, p_2 for element IDs in $op_1.X$, p_1 is not an ancestor p_2*. Then, duplicate elimination and sort may be skipped, reducing the cost of the physical plan. For example, none of the structural joins in Figure 4 require ordering or duplicate elimination.

Notice that path partitioning (Section 2.2) leads to structures which naturally fulfill this requirement. If all element IDs are stored together [32, 38], or partitioned by the tags [20], this is not the case. For instance, if all IDs of parlist elements are stored together in the document in Figure 1, this includes elements from paths 20 and 23, and 23 is a descendent of 20. Thus, when evaluating e.g. //parlist//listitem, some listitem IDs will get multiplied by their ancestors, requiring a duplicate elimination.

4.3 Reconstructing XML elements

The biggest performance issues regarding a path-partitioned store are connected to the task of reconstructing complex XML subtrees, since the data has been partitioned vertically. In this section, we study algorithms for gathering and gluing together data from multiple paths when building XML output.

A first approach is to adapt the SortedOuterUnion [30] method for exporting relational data in XML, to a path-partitioned setting with structural IDs. The plan in Figure 4 does just this: the components of the result (name and emph elements) are gathered via two successive structural outerjoins. In general, the plan may be more complex. For instance, consider the query:

for $x in //item return <res> {$x//keyword} {$x//emph} </res>

The plan in Figure 5(a) cannot be used for this query, because it introduces multi-valued dependencies [33]: it multiplies all emph elements by all their keyword cousins, while the query asks the for keyword and emph descendents of a given item to be concatened (not joined among themselves). The plan in Figure 5(b) solves this problem, however, it requires materializing the item identifiers (highlighted in grey), to feed them as inputs in two separate joins.

If the materialization is done on disk, it breaks the execution pipeline, and slows down the evaluation. If it is done in memory, the execution will likely be faster, but complex plans end up requiring more and more materialization. For instance, the simple query //person leads to the plan in Figure 5(c), where the IDs on both paths 3 (person) and 5 (address) need to be materialized to avoid erroneous multiplication of their descendants by successive joins. The sub-plan surrounded by a dotted line reconstructs address elements, based on city, country and street. The complete plan puts back together all components of person.

The I/O complexity of this method is driven by the number of intermediary materialization steps and the size of the materialized results. Elements from some path x must be materialized, as soon as they must be combined with multiple children, and at least one of children paths y of x is not annotated with 1 (thus, elements on path x may have zero or more children on path y). Some IDs are be materialized multiple times, after joins with descendent IDs at increasing nesting level. For instance, in Figure 5, person IDs are materialized once, and then a second time after being joined with address IDs. In the worst case, assuming IDs on all paths in the subtree to be reconstructed must be materialized on disk, this leads to $O(N * h/B)$ I/O complexity, where B is the blocking factor. If in-memory materialization is used, the memory consumption is in $O(N * h)$. The time complexity is also $O(N * h)$.

To reduce the space requirements, we devise a second method, named Reconstruct. The idea is to read in parallel the ordered sequences of structural IDs and (ID, value) pairs from all the paths to recombine, and to produce directly textual output in which XML markup (tags) and values taken from the inputs, are concatenated in the right order. The Reconstruct takes this order information:

- *From the path summary*: children elements must be nested inside parent elements. Thus, a <person> tag must be output (and a person ID read from IDs(3)) before the <name> child of that person, and a </name> tag must be output (thus, all values from IDsAndVal(4) must have been read and copied) before the </person> tag can be output.
- *From the structural IDs themselves*: after an opening <person> tag, the first child of person to be reconstructed in the output comes from the path n, such that next structural ID in the stream IDs(n) is the smallest among all structural

ID streams corresponding to children of person elements.

Figure 6(a) outlines a Reconstruct-based plan, and Figure 6(b) zooms in into the Reconstruct itself (the shaded area). Reconstruct uses one buffer slot to store the current structural ID, and the current (ID, value) pair, from every path which contributes some data to the output. The IDs are used to dictate output order, as explained above; the values are actually output, properly nested into markup. The buffers are connected by thin lines; their interconnections repeat exactly the path summary tree rooted at person in Figure 1.

A big advantage of Reconstruct is that *it does not build intermediary results*, thus it has a smaller memory footprint than the SortedOuterUnion approach. Contrast the QEPs in Figure 5(b) and Figure 6(b): the former needs to build address elements separatedly, while the latter combines all pieces of content directly. A second advantage is that the Reconstruct is pipelined, unlike the SortedOuterUnion, which materializes person and address IDs.

The Reconstruct has $O(N)$ time complexity. It needs one buffer page to read from every path which contributs some data to the output, thus it has $O(n)$ memory needs, where n is the number of paths from which data is combined; especially for large documents, $n \ll N * h/B$, thus the Reconstruct is much more memory-efficient than the SortedOuterUnion approach.

5. EXPERIMENTAL EVALUATION

We have implemented path summaries within the XQueC path-partitioned system [6, 7], and as an independent library [36]. This section describes our experience with building and exploiting summaries, alone or in conjunction with a path-partitioned store.

We use the documents from Table 1, ranging from 7.5 MB to 280 MB, with relatively simple (Shakespeare) to extremely complex (TreeBank) structure. Experiments are carried on a Latitude D800 laptop, with a 1.4 GHz processor, 1 GB RAM, running RedHat 9.0. We use XQueC's path-partitioned storage system [6], developed based on the popular persistent storage library BerkeleyDB from www.sleepycat.com. The store uses B+-trees, and provides efficient access to the IDs, or (ID,val) pairs, from a given path, in document order. All our development is Java-based; we use the Java HotSpot VM 1.5.0. All times are averaged over 5 runs.

5.1 Path summary size and serialization

Summary sizes, in terms of nodes, have been listed in Table 1. We now consider the sizes attained by serialized stored summaries. Two choices must be made: (i) XML or binary serialization, (ii) direct or precomputed encoding of parent-child cardinalities (Section 2), for a total of four options. XML serialization is useful since summaries may be easily inspected by the user, e.g. in a browser. Summary nodes are serialized as elements, and their annotations as attributes with 1-character names. Binary serialization yields more compact summaries; summary node names are dictionary-encoded, summary nodes and their labels are encoded at byte level. Precomputed serialization is more verbose than the direct one, since n1 and n+ labels may occupy more than 1 and + labels.

Table 2 shows the *smallest* serialized summary sizes (binary with direct encoding). Properly encoded, information-rich summaries are much smaller than the document: 2 to 6 orders of magnitude smaller, even for the large TreeBank summary (recall Table 1).

We also measured XML-based summary encodings for the documents in Table 2, and found they are 2 to 5 times larger than the direct binary one. We also measured the size of the binary precomputed summaries, and found it always within a factor of 1.5 of the direct binary one, which is quite compact.

5.2 Relevant path computation

Doc.	Shakespeare	XMark11	XMark233
Size (MB)	7.5	11	233
XML pre-comp (KB)	0.68	4.85	4.95
XML pre-comp / size	$8*10^{-5}$	$4*10^{-4}$	$2*10^{-5}$
Doc.	SwissProt	DBLP 2005	TreeBank
Size (MB)	109	280	82
XML pre-comp (KB)	3.11	1.62	2318.01
XML pre-comp / size	$2*10^{-5}$	$5*10^{-6}$	$3*10^{-2}$

Table 2: Serialized summary sizes (binary, direct encoding).

query no.	1	2	3	4	5	6	7	8	9	10
time (ms)	14	14	14	15	14	14	14	29	46	29
query no.	11	12	13	14	15	16	17	18	19	20
time (ms)	28	28	14	14	15	16	15	14	15	14

Table 3: Computing relevant paths for the XMark queries.

TKn: //S/VP/(NP/PP)^n/NP **T0**: //A **T1**: //NP **T2**: //NNP
T3: //WHADVP **T4**: //NP//NNP **T5**: //S[NPP][_COMMA_]/PP
T6: //ADJP/PP/NP **T7**: /FILE/EMPTY/S[VP/S]/NP/VP

Table 4: XPath renditions of query patterns on TreeBank data.

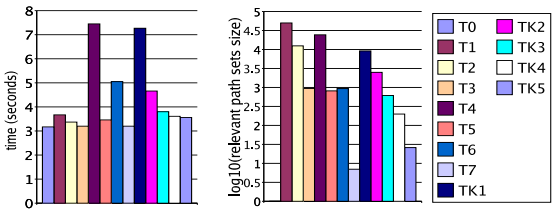


Figure 7: Relevant path computation times on TreeBank (left) and resulting relevant path set size (right, log scale).

We now study the performance of the relevant path set computation algorithm from Section 3.2. We use the XMark233 summary as representative of the moderate-sized ones, and Treebank as the largest (see Table 2). Table 3 shows the time needed to compute the relevant path sets for the 20 queries of the XMark benchmark [35], on the XMark111 summary, serialized in binary format with precomputed information. The query patterns have between 5 and 18 nodes. Path computation is very fast, and takes less than 50 ms, demonstrating its scalability with complex queries.

We now measure the impact of the serialization format on the relevant path computation time. The following table shows this time for the XMark queries 1 and 9, for which Table 3 has shown path computation is fastest, resp. slowest (times in milliseconds):

query no.	XML dir.	XML pre-cp.	bin. dir.	bin. pre-cp
1	73.0	37.0	22.3	14.2
9	255.7	133.6	98.6	46.4

Path computation on an XML-ized summary is about 4-5 times slower than on the binary format, reflecting the impact of the time to read the summary itself. Also, the running time on a precomputed summary is about half of the running time on a direct-encoded one. This is because with direct encoding, path set minimization requires a second summary traversal, as explained in Section 3. The space saving of the binary, direct encoding over the binary pre-computed encoding (less than 50%) is overcome by the penalty direct encoding brings during relevant path sets computations. We thus conclude the *binary, pre-computed encoding* offers the best time-space compromise, and will focus on this only from now on. If the optimizer caches query plans, however, the binary direct encoding may be preferrable.

We now consider the TreeBank summary (in binary pre-

computed encoding), and a set of query patterns, shown in Table 4 as XPath queries for simplicity (however, unlike XPath, we compute relevant path sets for *all* query nodes). Treebank tags denote parts of speech, such as S for sentence, VP for verb phrase, NP for noun phrase etc. TKn denotes a parameterized family of queries taken from [12], where the steps /NP/PP are repeated n times.

Figure 7 (left) shows the times to compute the relevant paths for these queries. Due to the very large summary (2.3 MB), the times are measured in seconds, two orders of magnitude above those we registered for XMark. Queries T0 to T3 search for a single tag. The time for T0 is spent traversing the summary only, since the tag A is not present in the summary,[1] thus no stack entries are built. The other times can be decomposed into: the constant summary traversal time, equal to the time for T0; and the time needed to build, check, and prune stack entries.

T1 takes slightly more than T2, which takes more than T3, which is very close to T0. The reason can be seen by considering at right in Figure 7 the respective number of paths: T1 results in much more paths (about 50.000) than T2 (about 10.000) or T3 (about 1.000). More relevant paths means more entries to handle.

The time for T4 is the highest, since there are many relevant paths for both nodes. Furthermore, an entry is created for all NP summary nodes, but many such entries are discarded due to the lack of NNP descendents. T5, T6 and T7 are some larger queries; T6 creates some ADJ entries which are discarded later, thus its relatively higher time. The times for TKn queries *decreases as* n increases, a tendency correlated with the number of resulting paths, at right in Figure 7. Large n values mean more and more selective queries, thus entries in the stacks of nodes towards the beginning of the query (S, VP) will be pruned due to their lack of required descendents (NP and PP in the last positions in the query).

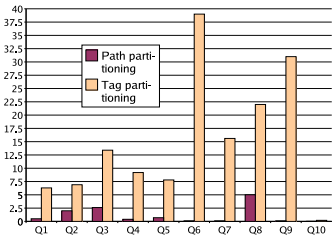
The *selfparent* encoding proved very useful for queries like T4. For this query, we counted more than 75.000 relevant path pairs (one path for NP, one for NPP), while with the *selfparent* encoding, only 24.000 stack entries are used. This demonstrates the interest of *selfparent* pointers in cases where there are many relevant paths, due to a large summary and/or to * query nodes.

5.3 Variable binding with path partitioning

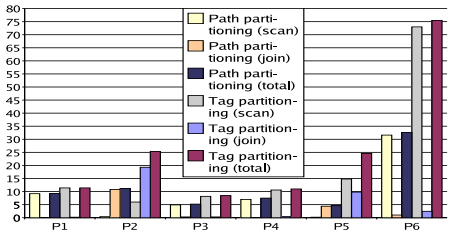
We measured the time needed to *bind* variables to element IDs on our path partitioned store. We identify relevant paths based on the summary, read the ID sequences for relevant paths, and perform structural joins if needed. For comparison, we also implemented a similar store, but where IDs are partitioned *by their tags*, not by their paths, as in [17, 20]. On both stores, the StackTreeDesc [3] structural algorithm was used to combine structural IDs.

Figure 8 shows the execution times for 10 XPath queries (Q6 on SwissProt, Q9 and Q10 on DBLP, the others on a 116 MB XMark document), and 6 tree patterns (P1 to P6 on the 116 MB XMark). In Figure 8, path partitioning takes advantage of the relevant path computation to achieve tremendous performance advantages (up to a factor of 400 !) over tag partitioning. This is because often, many paths in a document end in the same tag, yet only a few of these paths are relevant to a query, and our relevant path computation algorithm identifies them precisely. For the patterns P1 to P6, we split the binding time in ID scan, and ID structural join. We see that the performance gain of path partitioning comes from its reduced scan time. For all these queries, the relevant path computation time was less than 20 ms, thus 2 to 6 orders of magnitude less than execution time. This confirms the interest of the access method selection algorithm enabled by the summary.

[1] A tag dictionary at the beginning of the summary allows detecting erroneous tags directly. We disabled this feature for this measure.



Q1 //europe//item/descr
Q2 //regions//item//descr
Q3 //europe//parlist//bold
Q4 //europe//parlist//listitem
Q5 //item//descr//keyword
Q6 //Entry//METAL//Descr
Q7 //categ//listitem//text
Q8 //parlist//listitem//text
Q9 //dblp//book//author
Q10 //dblp//book//title



P1 for $x in //person, $y in $x//name, $z in $x//watch
P2 for $x in //asia, $y in $x//item, $z in $y//description
P3 for $x in //item, $y in $x//description, $z in $x//parlist
P4 for $x in //item, $y in $x//parlist, $z in $x//keyword
P5 for $x in //categories, $y in $x//parlist, $z in $x//text
P6 for $x in //article, $y in $x//title, $z in $x//year

Figure 8: Binding variables with path- and tag-partitioning.

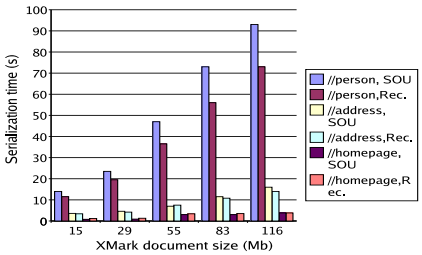


Figure 9: SortedOuterUnion and Reconstruct performance.

Impact of path minimization. Relevant path computation finds that the second tag in Q1-Q5 is useless (Section 3), thus IDs for those tags are not read, in the measures in Figure 8. Turning minimization off increased the running time by 15% to 45%.

5.4 Reconstructing path-partitioned data

We tested the performance of the two document reconstruction methods described in Section 4.3, on our path-partitioned store. Figure 9 shows the time to build the full serialized result of //person, //address, //homepage, on XMark documents of increasing sizes. The sorted outer union (denoted SOU in Figure 9) materialized intermediary results in memory. On the XMark116 document, //person outputs about 15 MB of result. As predicted in Section 4.3, both methods scale up linearly. The Reconstruct is noticeably faster when building complex elements such as address and person. Furthermore, as explained in Section 4.3, it uses much less memory, making it interesting for a multi-user, multi-query setting.

5.5 Conclusions of the experiments

Our experiments have shown that path summaries can be serialized very compactly; the binary encoded approach yields the best trade-off between compactness and relevant path computation performance. Our path computation algorithm has robust performance, and produces intelligently-encoded results, even for very complex summaries. Path partitioning takes maximum advantage of summaries; used in conjunction with structural identifiers and efficient structural joins, it provides for very selective access methods. Scalable reconstruction methods make path partitioning an interesting idea in the context of current-day XML databases.

6. RELATED WORK

Path summaries and path partitioning are not new [2, 9, 16, 19, 23, 27, 38]. Other more elaborate structure indexes have been proposed [21, 27], however, they are very complex to build and to maintain, and thus are built for a few selected paths only. Complex, richer XML summaries have also been used for data statistics;they tend to grow large, thus only very small subsets are kept [29]. This is appropriate for cardinality estimation, yet inadequate for access method selection, since some structure information is lost.

Path information has been used recently for XPath materialized view-based rewriting [8] and for access method selection [5, 10]. Our work is complementary in what concerns the path summary, since we formalized and presented efficient algorithms for exploiting summaries, which could be integrated with these works. As we have demonstrated, some documents yield large summaries, whose exploitation may raise performance problems, therefore, we have provided an efficient relevant path computation algorithm, which furthermore performs some interesting query minimizations. The only previous path summary exploitation algorithm concerns simple linear XPath path queries only [2], and it does not perform any minimization. With respect to [5, 10], in this work we focused on formalizing relevant path computation, and showing its benefits in the particular context of a path-partitioned store.

Many works target specifically query minimization, sometimes based on constraints, e.g. [4, 22]. We show how summaries can be used as practical structures encapsulating constraints, even when a schema is unavailable (which is often the case [26] !). Schema-independent minimization techniques are orthogonal to our work.

With respect to path partitioning, we considered the task of retrieving IDs satisfying given path constraints as in [8, 10, 27] and shown that structural IDs and joins efficiently combine with path information. Differently from [8, 10, 27] which assume available a persistent tree structure, we also considered the hard task of rebuilding XML subtrees from a path-partitioned store. We studied an extension of an existing method, and proposed a new one, faster and with much lower memory needs.

The starting point of this work is the XQueC compressed XML prototype [6, 7]. The contributions of this paper on building and exploiting summaries for optimization have a different scope. An early version of this work has been presented in an informal setting, within the French database community only [24].

7. REFERENCES

[1] S. Abiteboul, I. Manolescu, B. Nguyen, and N. Preda. A test platform for the INEX heterogeneous track. In *INEX*, 2004.

[2] A. Aboulnaga, A. R. Alamendeen, and J.F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[3] 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.

[4] S. Amer-Yahia, S. Cho, and L. Lakshmanan. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[5] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the right storage for your XML application. In *VLDB*, 2005.

[6] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing queries to compressed XML data (demo). In *VLDB*, 2003.

[7] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed XML data. In *EDBT*, 2004.

[8] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[9] D. Barbosa, A. Barta, A. Mendelzon, and G. Mihaila. The Toronto XML engine. *WIIW Workhshop*, 2001.

[10] A. Barta, M. Consens, and A. Mendelzon. Benefits of path summaries in an XML query optimizer supporting multiple access methods. In *VLDB*, 2005.

[11] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[12] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, 2003.

[13] Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.

[14] S. Chien, Z. Vagena, D. Zhang, and V. Tsotras. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[15] DBLP. http://www.informatik.uni-trier.de/~ley/.

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

[17] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A.N. Rao, F. Tian, S. Viglas, Y. Wang, J.F. Naughton, and D.J. DeWitt. Mixed mode XML query processing. In *VLDB*, 2003.

[18] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in XPath. In *DBPL*, 2003.

[19] H.Jiang, H.Lu, W.Wang, and J.Yu. Path materialization revisited: an efficient XML storage model. In *AICE*, 2001.

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

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

[22] L. Lakshmanan, G. Ramesh, H. Wang, and Z. Zhao. On testing satisfiability of tree pattern queries. In *VLDB*, 2004.

[23] M. Lee, H. Li, W. Hsu, and B. Ooi. A statistical approach for XML query size estimation. In *DataX workshop*, 2004.

[24] I. Manolescu, A. Arion, A. Bonifati, and A. Pugliese. Path sequence-based XML query processing. *Journées des Bases de Données Avancées* (informal proceedings only), 2004.

[25] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. TR, 1998.

[26] L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *Proc. of the Int. WWW Conf.*, 2003.

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

[28] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

[29] N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD*, 2002.

[30] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[31] 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.

[32] J. Teubner, T. Grust, and M. van Keulen. Bridging the GAP between relational and native XML storage with staircase join. In *VLDB*, 2003.

[33] J. Ullman. *Principles of Database and Knowledge-base Systems*. Computer Science Press, 1989.

[34] University of Washington's XML repository. www.cs.washington.edu/research/xml/datasets, 2004.

[35] The XMark benchmark. www.xml-benchmark.org, 2002.

[36] XSum. www-rocq.inria.fr/gemo/XSum, 2005.

[37] The XQuery 1.0 language. www.w3.org/XML/Query.

[38] M. Yoshikawa, T. Amagasa, T. Uemura, and S. Shimura. XRel: A path-based approach to storage and retrieval of XML documents using RDBMSs. *ACM TOIT*, 2001.