Path Sequence-based XML Query Processing

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

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

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

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

Abstract

We present the path sequence storage model, a new logical model for storing XML documents. This model partitions XML data and content according to the document paths; and uses ordered sequences as logical and physical structures.

Besides being very compact, this model has a strong impact on the efficiency of XQuery evaluation. Its main advantages are: fast and selective data access plans; intelligent combination of such plans, based on a summary of the document paths; and intelligent usage of a concise path summary.

The model is implemented in the GeX[1] system. We describe our physical storage scheme, query processing and optimization techniques. We demonstrate the performance advantages or our approach through a series of experiments.

1 Introduction

XML data management has been a very active research field lately. The XQuery [25] W3C standard for XML querying has almost reached its final state; accordingly, XML database research is concentrating on storage and processing techniques for XQuery evaluation [2, 28, 21, 8, 12, 9, 15, 19, 7]. To illustrate the issues involved in XQuery processing, we use the XMark [24] document snippet depicted in Figure 3(a). The document models an on-line auction site: clients, items for sale, and a classification of items in categories. We use italic fonts for #PCData and attribute values.

Example 1. Consider the query:

```
for     $i in //asia//item, $d in $i/description where $i//keyword="romantic"
return  <gift> <name> {$i/name} </name> {$d//emph} </gift>
```

For every asian item and every description of this item, such that the item has at least one keyword descendent with value "romantic", the query returns a new gift element with the item name, and the emphasized points in its description. Items with multiple "romantic" keywords must produce only one gift element, with the name and emph descendants if any – or an empty gift element otherwise.

A central notion in XQuery evaluation is that of *variable bindings*. The bindings of a query variable v are the elements (duplicate-free in the sense of element identity, and in document order) found by following the path expressions defining v. For example, the bindings for $i consist of the unique, ordered elements found by matching the path expression //asia//item. Similarly, bindings for $d are obtained by following //description from each binding of $i; $i bindings must also be unique, and ordered.

[1] GeX stands for "The Gemo XML engine".

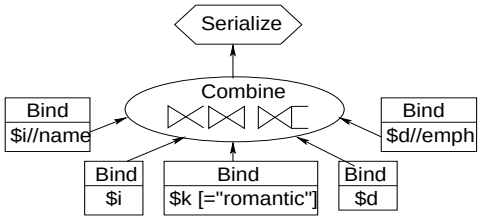


Figure 1: General XQuery evaluation strategy.

	Algorithm loadPathSequences(document d)
1	SEQ=∅; CNT=∅; PS = ∅; IDS=∅
2	n=0; m=−1; $post$=0; p_n=''''; p_m=''''; x_n=0; x_m=0
3	**beginElement**(tag t, attributes $alist$)
4	m=n; p_m=p_n; x_m=x_n
5	n++; p_n=p_n+$''/t''$; IDS.push([n, -1])
6	if the node of p_n is not yet created in PS
7	then create it; let x_n be its number
8	create $SEQ[x_n]$
9	if (m≠0) then PS.markBegin(x_m, x_n)
10	foreach $attr = (an, av)$ in $alist$
11	create the container $p_n/@an$ in CNT if needed
12	append (n, av) to this container
13	**characters**(string $chars$)
14	create the container $p_n/\#text$ in CNT if needed
15	append $(n, chars)$ to this container
16	**endElement**( )
17	$post$++; IDS.top.setPost($post$);
18	append IDS.pop() to $SEQ[x_n]$
19	PS.markEnd(x_n)
20	n=m; p_n=p_m; x_n=x_m

Figure 2: XML document loading algorithm.

A generic query evaluation strategy in the spirit of [9] follows from XQuery semantics [27]. *Bind* all the variables, and the subqueries in the return clause. *Combine* these bindings through joins. Finally, *serialize* the XML results, as dictated by the return clause. While the previous phases manipulate mainly element IDs, this step requires retrieving and tagging XML element contents.

Figure 1 exemplifies this strategy for the query from Example 1. We purposedly left unspecified the details of each step; we will discuss them later. Notice that the joins in Combine pair bindings according to value-based predicates (none in this example), or to *structural relationships* among the element IDs they contain. The latter joins are known as *structural joins* [2, 10, 8, 29]. In Figure 1, within Combine, a structural *join* connects $i and $d bindings; a structural semijoin connects $i and $k, since items with several matching keywords must produce only one result; and structural outerjoins connect $name and emph elements to $i and $d bindings, since a result must be output even for items without name and/or emph descendents.

The performance of XQuery evaluation depends on the efficiency of these three steps: binding, combination, and serialization. The latter can be done efficiently, once the content to be output has been gathered and ordered [20]. Thus, performance is determined by: bindings variables, according to the query predicates, and the relationships between variables; and combining them.

Recent works provide efficient techniques for implementing [2] and ordering [28, 9] structural joins, relying on a relational or tree-structured storage, augmented with B+-tree indexes [12, 9]. These indexes provide ordered access to the identifiers of elements of a given tag. We call this storage and indexing approach *tag partitioning* (**TP**).

In this paper, we propose the new *path sequence* storage model. Our model partitions XML content and structure according to the *data paths*, and stores it in *ordered sequences*. Its main advantage is its support for efficient variable binding, up to several orders of magnitude faster than TP. This is due to the precise structural knowledge of the document, included in the path sequence model. We make the following contributions:

- We describe the logical and physical *path sequence* storage model. This model is more compact than TP, and allows for efficient document loading. The model comprises a document *path summary*, encapsulating compact structural

information.

- We show that based on the path summary, binding variables is much more efficient than when TP is used.
- We extend the iterator execution model, for operators whose output is in document order. Using path summary information, and based on this extension, we derive a family of structural joins algorithms, which use selective predicates on one input to access only the matching part of the other input.
- We show that our storage and query processing model integrates well with XQuery optimization techniques developed for TP [28, 9]. Besides more efficient evaluation options, our model also provides precise structural informations to the optimizer, further improving XQuery performance.

The paper is organized as follows. Section 2 describes the path sequence storage model. We then show how to efficiently answer XQuery queries based on this model. Section 3 discusses the construction of variable binding plans. Section 4 addresses binding combination; the information from the path summary can be used to help an XQuery optimizer. Section 5 presents the approach we take for XML serialization. Section 6 validates the interest of our techniques through extensive experiments. We discuss related work in Section 7, and conclude in Section 8.

2 Path sequence-based storage

In this section, we describe the principles of path sequence-based XML storage, and its implementation options.

2.1 The logical path-sequence storage model

Our model separates the document structure from the document content, and stores each set of similar items in document order. We decompose the document into three distinct structures, that we describe in turn.

The first structure contains a compact representation of the XML tree structure. We assign unique, persistent identifiers to each element in an XML document. We adopt the [pre, post] scheme used in [2, 10, 12]. 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. Using this scheme, an element e_1 is an ancestor of an element e_2 *iff* $n_1.pre < n_2.pre$ and $n_1.post > n_2.post$. For example, Figure 3(a) depicts the [pre, post] ID of each element just above the element.

Furthermore, we *partition the identifiers according to the data path* to which the elements belong. Logically, each partition is a *sequence* of identifiers, ordered by their pre field, which reflects the document order. For example, Figure 3(c) depicts a few path sequences resulting from the sample document in Figure 3(a), for the paths /site, /site/people, /site/regions/asia/item etc.

All IDs in a path sequence appear at the same depth in the document tree, which is the path length. Thus, element depth is concisely stored within the paths.

Our second structure stores the contents of XML elements, and values of the attributes. We pair such values to their closest enclosing element identifier (precisely to the pre field of that element ID), and store them in a sequence of [pre,

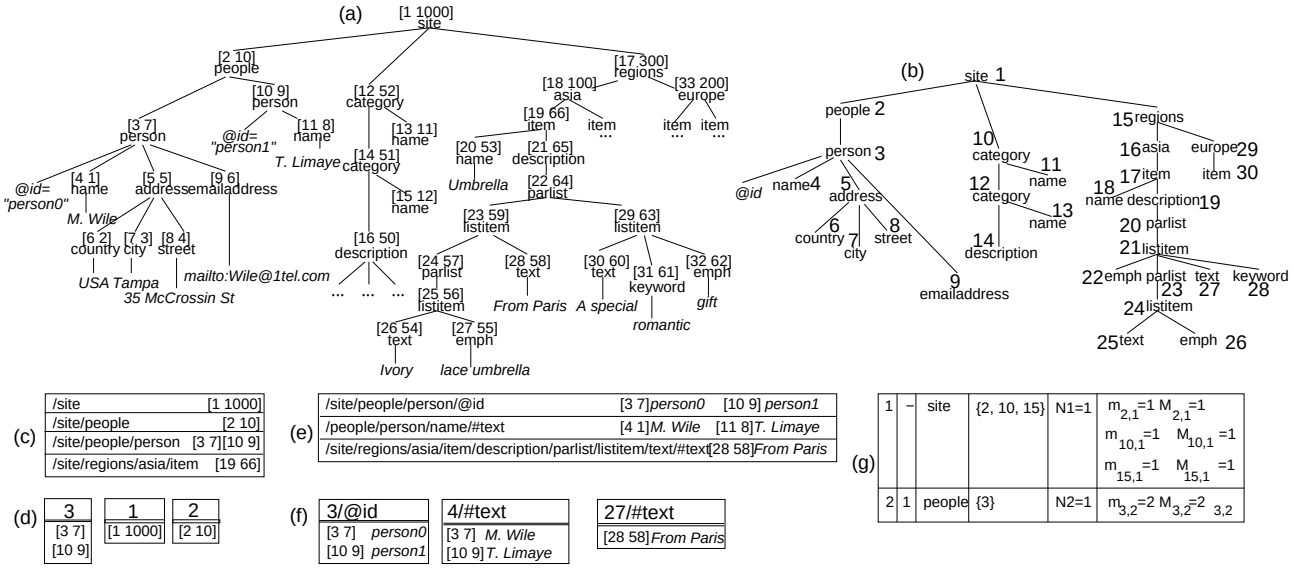


Figure 3: XMark document snippet, its path summary, and some of the resulting storage structures.

value] pairs ordered by pre. We call such a sequence a *container*.[2] Values found in a container are classified as string, integer, or double values; this can be inferred from their format. For example, Figure 3(e) shows some containers for the document in Figure 3(a), for /site/people/person/@id, /site/people/person/name/#text etc.

To deal with XML elements having both textual and element children, we can assign IDs [pre, post] to each text child, as if it were a text element. For example, the fragment <description>New<bold>Nike</bold> sport shoes </description> would yield the IDs: [1, 4] for description, [3, 2] for bold, and [2, 1] and [4, 3] for the two text children of description. For simplicity, in the rest of the paper, we ignore the mixed-content issue.

The above two structures alone can represent a document without any loss. We added a third indexing structure that will prove very useful in query processing. The *path summary* of an XML document is a tree, whose internal nodes correspond to XML elements, and whose leaves correspond to values (text or attributes). For every simple path $/l_1/l_2/.../l_k$ matching one or several (element or value) nodes in the XML document, there is exactly one node reached by the same path in the path summary.

To each node x in the path summary, we assign an *unique integer path number* which characterizes both the node and the path from the summary root to the node x. Thus, each path-driven ID sequence is uniquely associated to a path number. Each container is associated to a pair of a path number, and: either @attrName for attributes, or #text for text content. Figure 3(b) represents the path summary for the XML fragment at its left. Path numbers appear in large fonts next to the summary nodes.

A path summary is different from a dataguide [11]. The former is always a tree, while the latter can be a graph. Another relevant difference is that a dataguide tends to group nodes with the same tag, for example, a single dataguide node would stand for the nodes 17 and 30 in Figure 3. We keep separate information about these element sets.

The path summary encapsulates a set of simple and concise *statistics*. Let x be a node in the summary, on a path

[2]The term is inspired from the XMill project [14].

ending with the tag t, and y be a child of x. We record in the path summary the following statistics:

- N_x: the number of elements found on the path x (the size of the ID sequence corresponding to x).
- $m_{x,y}$, $M_{x,y}$: the minimum, resp. maximum number of y children of the XML elements on the path x.

Notations. For a given document, we denote by N its size, h its height, and N_{PS} the number of nodes in its path summary. We show in Section 6.2.2 that the path summary is very small compared to the document, and thus we keep it in memory at query processing time.

ID sequences, containers, and the path summary together are all the *storage* we materialize from a document, not *indexes* to be added to another persistent storage. We now discuss the loading of a document within this model.

2.2 Document loading algorithm

We load an XML document in a single pass, using an event-based parser [26] which raises events while traversing the document: beginElement on a start tag, characters when an element value is found, and endElement on an end tag.

Figure 2 shows the parser pseudo-code; it gradually construct the ID sequences SEQ, containers CNT, and path summary PS. At any point, the *current* element is the last one for which beginElement was raised; we identify it in Figure 2 by n, and its parent by m. Their respective paths are p_n and p_m, with path numbers x_n and x_m.

On beginElement, we save the current n in m (line 4) and assign a new n number. We push on IDS an incomplete $[n, -1]$ ID, since we don't know its post value yet. If n is the first element encountered on its path, we create the corresponding node in the path summary (lines 6-8). For each node x_m and child x_n, PS stores a temporary value crt_{x_m,x_n} representing, for the most recently encountered element on path p_m, the number of its children on path p_n. The call to PS.markBegin (line 9) increases crt_{x_m,x_n} by 1, and sets to 0 the counters crt_{x_n,x_q}, for all children x_q of x_n. The parser passes the attributes of n as parameters to beginElement, thus we append their values to the proper containers here (lines 10-12).

On characters, we append the value to the correct container (lines 13-15).

On endElement, we compute the post number of n, update n's ID, and pop it from IDS into the right sequence (lines 17-18). The PS.markEnd call (line 19) updates m_{x_n,x_q} and M_{x_n,x_q} using the value crt_{x_n,x_q} if needed.

The algorithm runs in time linear with N, using $O(h+N_{PS})$ memory for the stack and path summary.

2.3 Physical storage for the path sequence model

An essential feature of the path sequence logical model is order in ID sequences, and in containers. Thus, the physical storage structures implementing it must inherently support order. We consider two ordered persistent structures:

B+-trees. This is the option is considered in many works [8, 10, 12, 29], relying on a relational or persistent tree storage. Its advantage is robustness, and good support for updates. Its disadvantage, as we show in Section 6.2.2, is the important bloating factor due to extensive ID indexing.

Persistent sequences. The alternative that we investigate is the usage of *persistent sequences* as basic storage unit. The advantage of this model is its extreme compactness, which leads to reduced memory usage. We found at least one persistent storage system, endowed with locking and transaction functionality, supporting sequences as first-class citizens [6]. A more recent system [13] features a sequence-based storage, largely outperforming an RDBMS in applications where data order is critical (e.g., financial series data). The drawback of sequence-based storage is its poor behavior in the presence of updates.

In this work, we focus on read-only querying. Thus, we adopt persistent sequences as physical storage.

Figure 3(d) represents the physical storage of the ID sequences in Figure 3(b). Items in these sequences have constant length, equal to the ID size. Figure 3(f) depicts the sequences resulting from the containers in Figure 3(c). Container entries have variable length, due to the data values. Thus, we use sequences of variable-length items.

We store the path summary as a sequence of variable-length items, as in Figure 3(g). There is one item per node, comprising: the node number, its parent number, the children numbers, and the statistics previously described. The space occupied by the path summary is linear in N_{PS}.

3 Binding plans on a path sequence storage

In this section, we show how to construct variable binding plans using the path sequence model. Binding plans output *identifiers* of the elements to which the variables are bound.

Notations. A *linear path expression* (***lpe***) is an expression of the form: $(v_i?)\ (/|//)\ l_1\ (/|//)\ l_2\ \ldots (/|//) l_k$, where each l_i is a tag or a $*$, the l_is are connected by $/$ or $//$, $k >= 0$, and v_i is an optional query variable. We refer to k as the *length* of *lpe* P_i. We call an *lpe* of the form $l_1/l_2/\ldots/l_k$ a *simple lpe*. In particular, each node of the path summary corresponds to the simple *lpe* connecting it to the root. We say that two simple *lpes* are related, when the path summary node of one of them is an ancestor of the other's node. The path summary nodes *matching an lpe* P are those obtained by top-down evaluation of the *lpe* against the path summary, considered as a data tree. We also say the simple *lpes* of these nodes *match* P.

We consider a generic tuple-based execution model, following the iterator interface. We denote $op[i]$ the i-th column in the output of *op.* Each column may be either of a simple type like string, integer etc., or of type structural ID.

3.1 Binding one query variable

The first problem we consider is binding a single variable to an *lpe*, such as $x in the query fragment "for $x in /site/people/person/name".

Path partitioning allows immediate access to the bindings: just scan the corresponding ID sequence, numbered 4 in Figure 3. We use **IDScan(x)** iterator, returning the IDs from the path sequence associated to x; IDScan also fills in the depth field in element identifiers, with the length of the simple *lpe* of x. Its output is ordered, and duplicate-free.

Now consider more complex *lpes*, as for example in the fragment "for $x in //parlist//text". We identify the set of path summary nodes matching //parlist//text. We scan all corresponding ID sequences, and at the same time merge them in a

pipelined fashion. The resulting bindings are in document order, and free of duplicates.

In general, we can retrieve the bindings for $x by matching P against the path summary into a set of elementary paths, reading exactly the useful ID sequences, and merging them. No join is required.

Let x_P be the number of nodes in the path summary matching P, N_P be the number of bindings for P, and b be the blocking factor.

Theorem 3.1 *Using path partitioning, the bindings for P can be obtained with an I/O cost of $O(N_P/b + x_P)$, and a CPU cost of $O(N_P * log(x_P))$. The memory occupancy required is $O(N_{PS} + x_P)$.*

The I/O cost is due to scanning exactly the desired binding IDs. Each ID sequence is physically clustered, but all the x_P matching sequences may not be clustered together, thus the x_P extra reads.

The CPU cost corresponds to the merge of x_P ordered ID sequences. The **Merge** operator uses a balanced search tree with x_P leaf nodes. Each ID read from one input is inserted in the search structure in $O(log(x_P))$; to produce an output, the Merge extracts from the search structure the smallest ID it contains, again in $O(log(x_P))$.

The memory occupancy corresponds to holding the path summary, and the Merge's search structure.

To match an *lpe* P against the path summary, we traverse the summary top-down, identifying the tags in P from left to right, and adding nodes found to match the last tag in P to the result set. This can be done in $O(N_{PS})$ time. For example, to bind //asia//item, we start from node 1 with P, and attempt to match its first tag asia. Since the root has a different tag, we propagate the search for P to all children of the root, which propagate it further. On the regions/asia branch, the asia tag is matched, and the search for //item is propagated to its children etc.

Binding with simple predicates. Now consider simple selections on the text value, or attributes of $x, as in:

for $x in //people//person[@id="person0"]

In such cases, we need to access the container for person/@id. We use a **ContScan(*x*)** operator, where x is a container path like 3/@id in Figure 3, returning the ordered [pre,value] container tuples. On top of ContScan, a selection can be applied. If an index on the container exists, we use the **IdxAccess** operator to access only the desired tuples. We furthermore apply a structural join with IDScan(4), to retrieve the complete [pre,post,depth] IDs.

Binding with more complex predicates. This case can be thought of as binding several variables, on some of which there are simple predicates. For example, the following two fragments are equivalent:

for $x in //europe//item[//keyword="romantic"]//price... and

for $x in //europe//item, $y in $x//price where $x//keyword="romantic"...

We address many-variable binding next.

3.2 Binding several related variables

In general, bindings must be computed for several related variables. For example, Figure 4(a) depicts the query variables from Example 1, structured in a *generalized tree pattern (GTP)* [9]. We use simple lines for / relationships, and double lines for //. Dashed lines denote left outerjoin connections between two nodes, considering the parent at left. Edges with

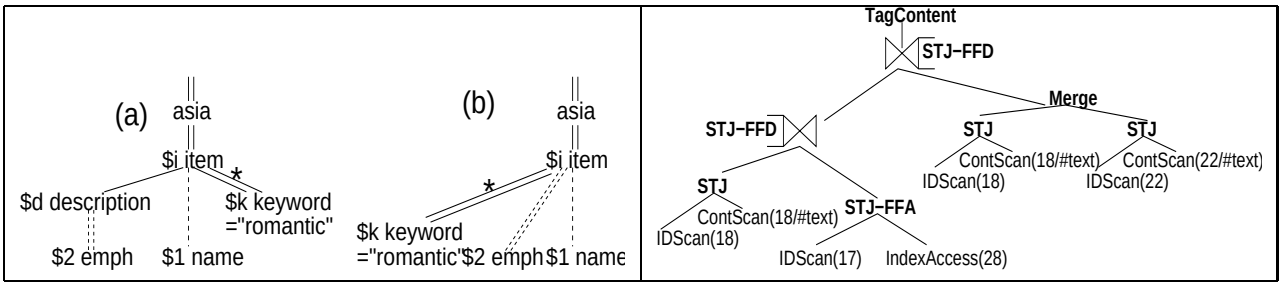


Figure 4: Generalized tree pattern (a) and reduced pattern (b) for the query in Example 1 (left); complete query execution plan to be discussed throughout the paper (right).

a '*' denote left semijoin relationships (also considering the parent at left). We gave the ad-hoc names $1 and $2 to the nodes corresponding to the return clause.

To bind such variables, we proceed in two steps. First, we determine the minimum data sets that need to be accessed to bind the variables, by analyzing relationships among variables, and the path summary. In this step, we may also discover useless variables, or new relationship among them. Second, we construct binding plans, based on the knowledge gathered in the first step.

Variable path inference. We start by computing, for every variable, the set of paths in the path summary to which it may possibly be bound. Throughout this section, let u, v be query variables such that u is a parent of v in the GTP, and s_u, s_v be their respective sets of possible paths.

We compute s_u sets by traversing the GTP and the path summary in parallel, using s_u as a starting point to determine the s_v. For example, for the query in Figure 4 and the path summary in Figure 3, s_i is {/site/regions/asia/item}. From the node in s_i, we match the /description path to find that s_d is {/site/regions/asia/item/description}. Similarly, matching //keyword from s_i yields the set s_k={28} (denoting paths by their numbers in Figure 3(b)). Using s_d and s_i, we find s_1={18} for $1 and s_2={22, 26} for $2.

Semijoin transformation. During path inference, for every edge connecting a parent variable v to a child variable u, and pair of related paths $x \in s_u$, $y \in s_v$, we compute the *maximum path factor* $Mpf_{x,y}$. This factor is the maximum M_{x_i,y_i} found by traversing the path summary from node x to node y. For example, for the edge between $i and $2 in Figure 4(a), we compute the $Mpf_{17,22}$ and $Mpf_{17,26}$; $Mpf_{17,22}$ is the largest among $M_{17,19}$, $M_{19,20}$, $M_{20,21}$, and $M_{21,22}$.

If $Mpf_{x,y} \leq 1$, and if the edge connecting u to v is a semijoin, we transform it into a join edge, since join and semijoin coincide. For example, if $Mpf_{16,28} \leq 1$ in Figure 3, the execution of the query in Example 1 could start with an access to $k, followed immediately by a join with the bindings for $i. Without this optimization, a duplicate-elimination step on $i identifiers would be needed, when combining the bindings.

Join branch elimination. Similarly, we compute the *minimum path factor* $mpf_{x,y}$ for each related pair of paths x and y, as the minimum m_{x_i,y_i} along the path from x to y. If the edge between u and v is a join, $mpf_{x,y} = 1$ and $Mpf_{x,y} = 1$ for all related paths x, y, and there is no predicate directly on v, then we eliminate v from the GTP, and "glue" all descendents of v to u directly. For example, on the GTP in Figure 4(a), if all asian items have exactly one description, the variable $d is useless and is eliminated, leading to the reduced GTP in Figure 4(b).

```
   Algorithm traverse(GTPNodes u,x,
                      PSNodes pu, px, pv)
1  foreach GTP node v_i child of u
2    nextX=x; nextPX=px
3    if (the tag of pv matches the tag of v_i)
4      if (v_i is a variable node)
5        mark(x, v_i, px, pv)
6        upd(mpf_{px,pv}, Mpf_{px,pv}, m_{pu,pv}, M_{pu,pv})
7        nextX=v_i; nextPX=pv
8      foreach PSNode pw child of pv
9        traverse(v_i, nextX, pv, nextPX, pw)
10   if (the edge u-v_i is ancestor-descendent)
11     foreach PSNode pw child of pv
12       traverse(u, x, pv, px, pw)
```

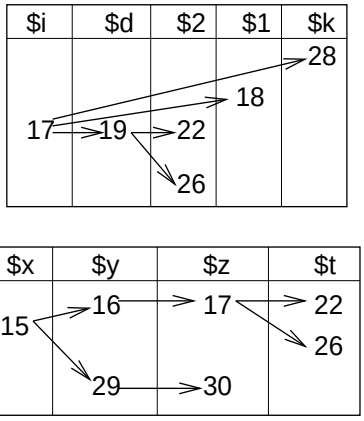


Figure 5: The path inference algorithm, and sample results.

Variable path pruning. Next, we use each set s_v to prune useless paths from s_u, where u is a variable parent of v in the GTP. For example, consider the query: for $p in //parlist, $k in $p//text return $k.

Path inference in this case yields s_p={20, 23}, and s_k={26}. Since we know by the path summary that 26 is a descendent of 20, but not of 23, we eliminate the useless path 23 from s_p, which becomes {20}. Such pruning applies only when in the GTP, x and y are connected by continuous edges only. It does not apply to children connected to their parents by dashed edges, since these children are not required by the query. Thus, even ancestors that do not have them must produce results.

The same pruning may find that a query has no answer, due to an empty path set for a variable connected through joins and semijoins only to the root.

GTP enrichment. Finally, new structural relationships among variables may be discovered. For example, consider the fragment "for $i in //item, $p in $i//parlist, $t in $i//text". After path inference and pruning, all paths of $t are descendents of some path of $p. This translates into a new join edge from $p to $t, providing new possibilities of binding combination (the GTP node set does not change).

The path inference algorithm. The algorithm is sketched in Figure 5. It consists of a recursive traversal function traverse that is invoked on a path summary node pv, given: its parent summary node pu; the last GTP node u that has been matched to an ancestor of pv; the last GTP *variable* node x that has been mached to an ancestor of pv[3]; and the path px where this x was matched.

The invocation of traverse on pv aims to match pv to one of the (GTP) children of u. For each such child v_i, we check if the tag of pv matches that of v_i. If, furthermore, v_i is a variable, then we mark the path we found for it, given that its parent variable x was matched to the path px (line 5). Furthermore, we update $mpf_{px,pv}$ and $Mpf_{px,pv}$ with the m and M statistics stored for pu and pv (line 6).

Next, whether v_i is a variable or not, we propagate the matching further by trying to match the children of v_i (in the GTP) to the children of pv (in the path summary), through recursive calls (lines 8-9). For this call, the last variable matched is x (that may have changed when matching v_i), and the last GTP node matched is v_i.

Finally, if the edge beween u and v_i is ancestor-descendent, we also traverse pv's children with the same parameters

[3]Not all GTP nodes are variables; for example, the asia node in Figure 4 is not a variable, yet it has to be matched before its descendents.

as for pv. This ensures that // GTP edges are correctly matched at any depth in the path summary.

The worst-case complexity of this algorithm is $O(N_{PS}^{N_{GTP}})$, where N_{GTP} is the size of GTP; this is consistent with e.g. [18]. The worst case arises, for example, if the summary degenerates into a list of N_{PS} nodes labeled a, and the GTP is a list of N_{GTP} //-connected nodes, labeled a. However, path summaries for real data sets are quite different, therefore, this cost is affordable in practice. Furthermore, N_{GTP} is rarely very large; and if all variables of the query do not have a common ancestor, we have to solve several smaller, unrelated, matching problems.

The inference process outputs the path lists found for each variable, grouped hierarchically following the variables in the GTP. In Figure 5, at the top right, we depicted the result for the query in Example 1. Each path found for a given variable points to the corresponding paths found for its children variables in the GTP. (Arrows preserve all edge semantics from the GTP; arrows leaving from the same path of a variable, and reaching different paths of another variable, have *or* semantics.) Arrows are also annotated with the mpf and Mpf computed factors.

Paths without required descendents are also pruned based on this result. For example, in Figure 5 at the bottom right, we have depicted the paths resulting for the variables in: "$x in //regions, $y in $x/*, $z in $y/item, $t in $z//emph" on the path summary in Figure 3. Realizing that path 30 for $z has no descendent path for $t, we prune 30, and then 29.

Semi-join transformation and join branch elimination also apply (and may reduce) on the result of path pruning. Finally, to infer new edges in the GTP, we analyze the surviving variable paths. This process may be piggybacked on traverse, using a matrix of N_{PS}^2 bits in which we mark which path is descendent of which other. The complexity of these steps is still dominated by the path inference itself.

Binding plan construction. Using the path sets that result from the above steps, we construct individual binding plans, as described in Section 3.1. For the query in Example 1, this yields: IDScan(17) for $i; Join(IDScan(28), Filter(ContScan(28/#text)) for $k; IDScan(19) for $1; Merge(IDScan(22), IDScan(26)) for $2 ($d has been eliminated). These plans are selective, as they only access IDs whose paths allow them to contribute to the query result.

Metadata and statistics for the binding plans. Let $|op|$ denote the cardinality estimate for operator op. If $op[i]$ is of type ID, we denote by $pList(op, i)$ the set of path numbers on which IDs from $op[i]$ may be found.

For IDScan(x) plans, $pList(op)$={x}, $|op|=N_x$. For plans of the form Merge(IDScan(x_1),..., IDScan(x_k)), $pList(op)$={x_1,x_2,...x_k}, we estimate $|op|$ as $\sum_{x \in pList(op)} N_x$. For Filter or IdxAccess plans, predicate selectivity estimation are needed to estimate $|op|$. Value statistics on containers can be used to this purpose.

4 Combining binding plans

Binding plans are combined bottom-up through joins and structural joins, as in [28, 9]. We now focus just on efficient techniques enabled by the path sequence model, which can be profitably integrated to the combination step.

Section 4.1 derives from the path-based model an order-preserving condition for structural joins. Section 4.2 describes more complex algorithms, that skip useless parts of the join inputs, enabled by our ordered path-partitioned storage. Section 4.3 discusses the impact of our techniques on join ordering, and choice of physical operator.

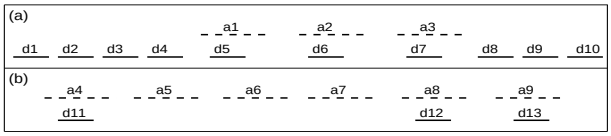


Figure 6: Structural joins when inputs can be skipped.

```
    Algorithm STJ-FastForward(anc, desc)
1   a = anc.getID(); d = desc.getID()
2   while (! eof(anc) or ! eof(desc)) do
3       if (a is an ancestor/parent of d)
4       then output [a, d]; d = desc.nextFF-pre(a.pre)
5       else if (d before a)
6           then d = desc.nextFF-pre(a.pre)
7           else // a before d
8               a = anc.nextFF-post(d.post)
9   endwhile
```

Figure 7: The STJ-FastForward (STJ-FF) algorithm.

4.1 Order-preserving structural join

The *StackTreeDesc (STD)* and *StackTreeAnc (STA)* algorithms [2] directly apply in our context. They pair inputs by structural relationships among structural [pre, depth, post] element IDs they contain. STA and STD require inputs sorted in the document order of the IDs to join; STA preserves the order of the ancestor IDs, and STD the order of the descendent IDs. Both algorithms employ a stack; STA needs some other data structures. Both work in pipeline.

To avoid intermediary sorting steps, it would be interesting to have a structural join operator preserving *both* ancestor and descendent ID order; however, this is not always possible. For example, let op_1 output successively the IDs [23,59], [25,56], and let op_2 return successively [26,54], [26,58]. The join result may be ordered in either order:

- ([**23**,59], [26,54]); ([**23**,59], [28,58]); ([**25**,56], [26,54])
- ([23,59], [**26**,54]); ([25,56], [**26**,54]); ([23,59], [**28**,58])

A structural join can preserve both input orders iff the IDs in the ancestor are free of ancestor-descendent pairs [2, 9]. This condition is data-dependent, and thus impractical to check. We provide a sufficient, easier to check condition:

Theorem 4.1 *Let op_1 and op_2 be two operators, such that $op_1[i]$ and $op_2[j]$ contain element IDs. If the paths in $pList(op_1, i)$ are all pairwise unrelated, then a structural join matching ancestor IDs in $op_1[i]$ with descendent IDs in $op_2[j]$ can preserve both input orders.*

Given a set $pList$ of size k, the condition can be checked on the path summary in $O(k * h)$: from each node in $pList$, we navigate upwards to the root; if we encounter another node from $pList$, then the paths are not pairwise unrelated. Thus, when anc paths are unrelated, the (simpler) STD algorithm can be used with the knowledge that it preserves both ancestor and descendent order.

4.2 The STD-FastForward family of algorithms

A structural join may avoid reading parts of the inputs [29, 10, 8, 21]. Figure 6 depicts two such examples. Element IDs are shown as horizontal [pre,post] segments, dashed for IDs of the anc operator, and solid for IDs of the $desc$ operator. An anc ID a is an ancestor of a $desc$ ID d iff the segment of d is under the segment of a.

In Figure 6(a), the first descendent of a_1 is d_5. Thus, $d_1 - d_4$ could be skipped, since they do not produce join results. Instead, the join should advance $desc$ to the first d that may be a descendent of a_1, that is, such that d.pre>a_1.pre. Since the $desc$ tuples are sorted on pre, it is possible to advance $desc$ according to this order.

In Figure 6(b), after the pair a_4, d_{11}, the join could skip $a_5 - a_7$, and advance anc to the first a that could be an ancestor of d_{12}, that is, such that a.post>d_{12}.post. This requires skipping on post, but the inputs are ordered on pre; the skip is only possible if the two orders coincide. From the pre and post semantics, it is easy to see that these orders coincide in an ID set iff it has no ancestor-descendent pairs.

A sufficient condition applies: Given an operator *op* such that *op*$[i]$ is of type ID, if $pList(op, i)$ are pairwise unrelated, then the pre and post orders in *op*$[i]$ coincide.

Thus, we design the *STJ-FastForward* (**STJ-FF**) structural join algorithm; it is derived from StackTreeDesc, although for readability we omitted the stack manipulation details. STJ-FF skips inputs both based on anc.post and $desc$.pre. STJ-FF is depicted in Figure 7. It relies on two new methods provided by anc and $desc$:

- *nextFF-pre(col, int n)* positions an operator *op* to the first output tuple having *op*$[col]$.pre$\geq n$. If such a tuple does not exist, nextFF-pre returns false.
- *nextFF-post(col, int n)* positions an operator *op* to the first output tuple having *op*$[col]$.post$\geq n$. If such a tuple does not exist, nextFF-pre returns false.

Using these methods, a parent operator commands its children to produce tuples with certain values only. They can only be called on operators *op* whose output is sorted on *op*$[col]$.pre; nextFF-post further requires that the paths in $pList(op, col)$ be pairwise unrelated.

IDScan and ContainerScan operators implement the nextFF methods by binary search over the sequence. The Merge, and the STJs, implement it combining calls to the nextFF methods of their child operators. An STJ supports: (i) nextFF-pre on both the ancestor and descendent ID columns; (ii) nextFF-post on the ancestor and descendent ID columns, *if* those paths are unrelated. For example, let *op*=STJ(IDScan(17), Merge(IDScan(21), IDScan(24))) in Figure 3. *op* supports: nextFF-pre on columns 0 and 1, and nextFF-post on 0 only, since 21 and 24 are related.

In Figure 6(a), after the initial skip to d_5, the region d_5, d_6, d_7 contains useful IDs. Thus, after finding d_5, a binary search to find d_6 would make useless effort. Therefore, we implement skipping "optimistically" in the IDScan and ContScan operators: the next available value is examined, and only if it doesn't satisfy the skip condition, binary search is performed. This reduces skipping overhead when matching IDs are grouped, as we illustrate in Section 6.3.

Skipping in structural joins may lead to thrashing [29]. For skipping to be profitable, a certain threshold frequency of useless IDs must be reached, starting from which skipping allows to scan less data. If a block is read for only one useful ID, skipping the comparisons for the other ones will not bring noticeable benefits (Section 6.3).

The situations in Figure 6 do not always occur simultaneously. For example, consider "for $x in //person[id="person0"], $y in $x/name". When joining bindings for $x and $y, there will be many useless $y bindings, and a single (useful) one for $x. Thus, we specialize STJ-FF into: **STJ-FFA**, which only skips anc tuples, and advances $desc$ through next() calls only; and **STJ-FFD**, which only skips $desc$ tuples, and advances anc through next(). We demonstrate their interest in Section 6.3.

4.3 Optimizing with structural joins and path sequences

To evaluate a given structural join, the optimizer examines the operands' $pLists$, and whenever ancestor $pLists$ are unrelated, it knows that both ordered will be respected. For example, *all* structural joins derived from the XMark [24] queries feature unrelated anc paths.

The optimizer also needs to decide when to use skipping structural joins as those described in Section 4.2. For this, cardinality estimates on binding plans, and structural join plans, are needed. In Section 3.2, we have shown how to estimate the cardinality of individual binding plans. The estimation techniques described in [23] can be used to determine the structural join plans size, based on the binding plan statistics. In [23], histograms are built on structural IDs of elements satisfying a given set of *elementary predicates*, such as: having a given tag, or text value. Our framework provides more precise elementary predicates, since we know from which *path* a given ID comes; similarly, value statistics are built on path-partitioned containers. For example, for the query: "for $x in //person[@id="person1"], $y in $x//age", STJ-FFA is a good join candidate, since $x is only bound to a single person.

To chose a join order, our optimizer relies on I/O cost and cardinality estimates for data access plans and join plans, and on the interesting orders required and preserved by each operator, to avoid sorting steps whenever possible.

5 Serializing XML results

We now describe our strategy for constructing new XML results, and/or for retrieving full XML elements from the database. Since the path partitioned model is very fragmented, those two tasks are exactly the same: any result element must be (re)constructed piece by piece. For example, consider the query in Example 1, after the path inference and simplifications described in Section 3.2. The query is internally brought to the form:

for $i in //asia//item where $i//keyword="romantic"

return <gift> <name>$i/name/text()</name> $reconst$($i//emph,{22,26}) </gift>

where $reconst$ is an internal reconstruction macro. This macro is unfolded into a GTP fragment as follows: the path summary subtrees rooted at 22 and 26 are traversed entirely top-down, and outerjoin edges are added for all the data elements contained. Finally, a Merge operator is added, since there are two different paths.

Figure 4(right) depicts the resulting complete QEPs for this query. It starts by joining $i bindings with bindings for the "romantic" keywords; STJ-FFA skips useless item IDs. This is followed by two successive pipelined outerjoins for $i/name and $i//emph. Both use STJ-FFD, since there are relatively few ancestors and many descendents. The **TagContent** operator serializes XML contents out of a stream of tuples sorted and structured following the lines of [20].

In general, let x be a node in the path summary, I_n be the size of the path summary subtree rooted at x, and C_n be the number of attribute and content containers attached to these I_n nodes. Reconstructing elements on path x takes $I_n + C_n$ structural joins: one join to connect every node to its parent, and one to connect the containers. The leaves of the reconstruction plan are IDScans and ContScans. For a very large number of outerjoins branches, materializing the result of the for-where branches, such as the STJ-FFA in Figure 4(right), and performing the outerjoins separately may be faster, since it prevents the outerjoins from building gradually very large intermediary results. The overhead is in keeping

intermediary results (ID tuples) in memory, which is in general affordable.

Despite the high degree of storage fragmentation, reconstruction is quite performant: (i) Each leaf operator reads data on the correct path only; in the absence of selection predicates, the data read is exactly the data that must be returned. If predicates apply, STJ-FF joins may skip useless descendents. (ii) All joins preserve both orders (no sort).

6 Experimental validation

In this section, we present the implementation of the techniques previously presented within the GeX prototype (Section 6.1). We assess the efficiency of our storage system for various XQuery processing tasks: storage conciseness and binding plans (Section 6.2), binding plan combination (Section 6.3), and serialization (Section 6.4).

6.1 Hardware and software environment

We implemented the path sequence model in the GeX system. GeX is entirely implemented in Java; it consists of about 150 classes, or 40.000 lines of code. The GeX persistent storage is built on the BerkeleyDB [6] database library. We used constant-size sequences for storing IDs, and variable-size ones for containers and the path summary. When container indexes were needed, we built them using BerkeleyDB's B+-trees.

Within the query engine, IDScan, ContScan, and IdxAccess operators interact directly with the BerkeleyDB storage. The optimizer is similar to the one described in [9]; however, it constructs binding plans as described in Section 3 by reasoning on the path summary, and uses the structural join operators described in Section 4.1, as explained in Section 4.3.

We ran GeX on a DELL D800 computer, with a Pentium III at 1.4 GHz and 1Gb of RAM, running RedHat Linux 9.0. All times are averaged over 20 hot runs. We report results on: an **XMark** document of 118 Mb; the **DBLP** data set of 133 Mb; and the **SwissProt** dataset of 114 Mb, available at www.cs.washington.edu/research/xmldatasets.

6.2 Data storage and binding plans

6.2.1 Comparison baseline: tag partitioning

We compare our approach with the state-of-the-art XML storage model, based on partitioning, and structural identifiers. Within this model, XML documents are stored as persistent trees or relational tables. Redundant B+-tree indexes are added to the storage, indexing all element IDs by a (tag+pre) key. These indexes give access to the list of all IDs of a certain tag. We call this approach *tag partitioning* (**TP**); it is used in [9, 12, 29]. We refer to our approach as *path sequences* (**PS**).

In TP systems, index-based query processing is much faster than using the persistent tree or non-indexed tables. Thus, we compare our query processing techniques with the techniques from [2, 28, 9] based on the tag indexes. To that purpose, we implemented TP also, in two variants: using BerkeleyDB's B+-trees, and by tag-based sequences. The latter can be seen as merging all sequences (or containers) ending in the same tag, in a single sequence.

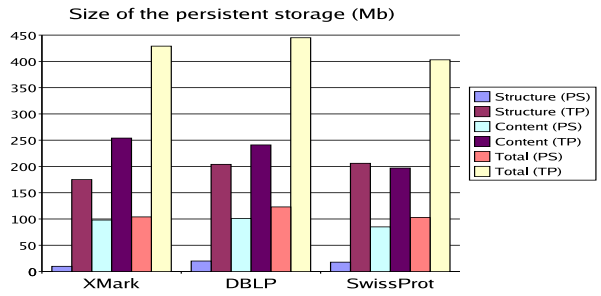


Figure 8: Space occupancy comparison for PS and TP.

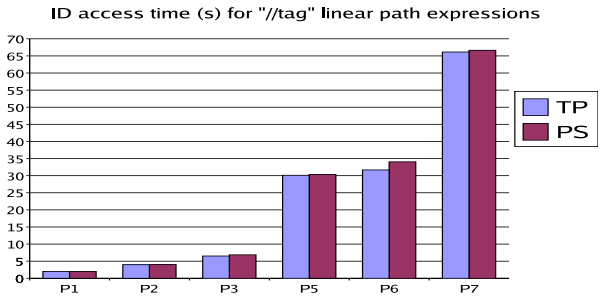


Figure 9: PS merging overhead when binding one variable.

6.2.2 Space occupancy

Figure 8 compares the space occupancy for PS, and TP using B+-trees as in [9, 12, 29]. For XMark, document structure stored with PS takes about 10% of the document size; with TP, it takes 20 times more. This is due to the presence of tags in the B+-tree keys (with PS, they are only stored in the path summary), and to the intermediary index pages. Overall, the document size is multiplied by 3.5-4 by when stored with TP, close to the factor of 4 reported in [9]; with PS, the storage size is slightly smaller than the document.

The path summary occupies 26 Kb for XMark, 6 Kb for DBLP, 10 Kb for SwissProt; N_{PS} is respectively 514, 122, and 117. This validates the assumption that it can be kept in memory when processing queries.

Having demonstrated sequence conciseness, in the following, we report on PS *and* TP performance on a persistent sequence-based storage. We have experimented query processing on TP-based B+-trees also, using a specific set of IDScan and ContScan operators; the overall observations are the same.

6.2.3 Variable binding performance

In this section, we compare TP and PS binding performance. Without loss of generality, we mainly consider *lpes* where tags are connected by //; the / case is similar.

The case most favorable to TP is a variable bound to a *lpe* of the form //tag: PS merges several IDScans (Section 3.1), while TP only looks up the IDs of the given tag. When only one path summary node matches //tag, PS and TP coincide.

Figure 9 compares the execution time for binding plans when the TP and PS plans differ. The *lpes* used are: P_1=//item, P_2=//description, P_3=//bold, P_4=//category on XMark; P_5=//title, P_7=//author on DBLP; and P_6=//Descr on SwissProt. The cardinalities of the binding plans range from 2.000 (P_1) to 720.000 (P_7). The PS overhead, due to scanning and merging different sequences, is negligible. This is because: (i) the Merge is pipelined, CPU-only, and efficient due to the ordered tree of inputs it maintains; (ii) IDs are tightly packed by PS in sequences and read sequentially, thus no thrashing occurs; (ii) the Merge fan-out is of the order of hundreds, making it possible to keep in memory one block from each sequence to be merged.

Binding a variable to longer *lpes*. In such cases, TP is handicapped by its unselective data access. For example, let P be /site/people/person/name: TP only allows to access all name IDs. Most of such IDs are *not* on the required path; they may be on paths 11, 18 etc. (Figure 3), leading to useless data scans. To separate just the person names, we must access

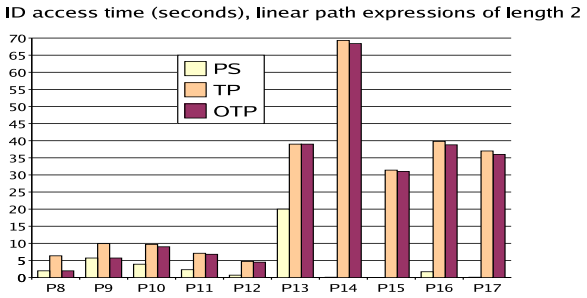


Figure 10: Binding one variable to *lpe*s of length 2.

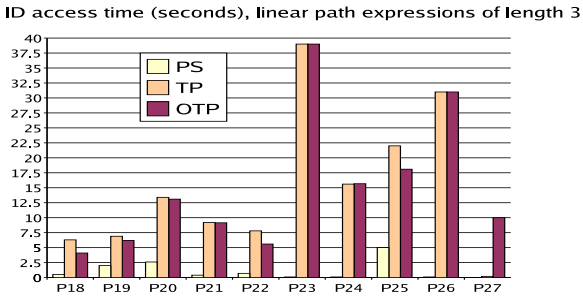


Figure 11: Binding one variable to *lpe*s of length 3.

all person IDs (thus read even more data !), and apply a structural join between the two. Furthermore, we must read all site and people IDs, and successively join with them also, to separate just the proper names.

In general, to bind a variable to a *lpe* of length k, TP must scan the IDs of all tags involved, and compute an k-way join. PS just scans the result. (although we only need the IDs of the elements of tag l_k). Contrast this with theorem **??**.

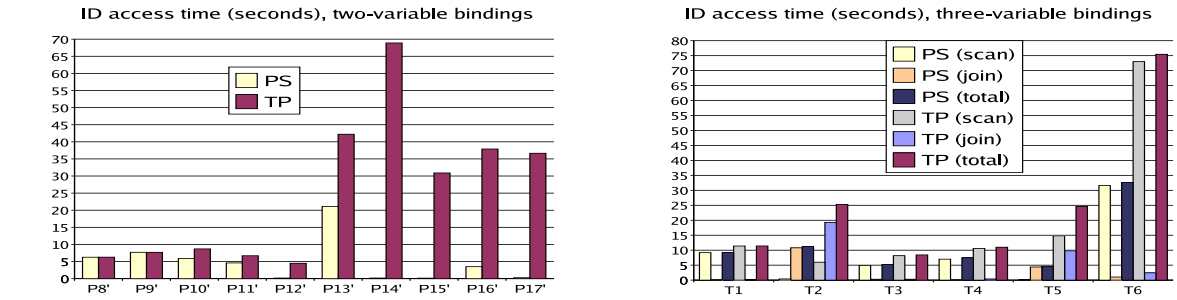
Now consider the query "for $x in //parlist//text". TP must join all parlist IDs with all text IDs; but a new complication arises. A text ID may be a descendent of two parlist elements, as is the case for the element [26,54] in Figure 3. Such text elements will appear twice in the join result. Thus, a final duplicate elimination (*dup-elim*) is required.

Optimized TP. To give TP a chance, we use the PS path summary to eliminate useless data scans, joins, and dup-elims from TP binding plans.

Let P be a linear path expression of length k, $P = l_1//\ldots//l_{i-1}//l_i//l_{i+1}//\ldots//l_k$, s_P be the set of path summary nodes matching P, and s'_P be the set of nodes matching $P' = l_1//\ldots//l_{i-1}//l_{i+1}//\ldots//l_k$ (l_i has been skipped). If $s'_P = s_P$, then P' is equivalent to P; reading and joining the IDs with the tag l_i can be avoided, reducing the binding cost. In general, more than one such tag l_i can be eliminated; for example, from //site//people//person//name we can eliminate site, and person. There may be several minimal-length *lpe*s equivalent to P; both //people//name and //person//name are equivalent to the previous path, and cannot be further reduced. Once s_P and s'_P are known, we can decide if $s'_P = s_P$ in $O(N_{PS} * h)$ (look for a node labeled l_i between l_{i-1} and l_{i+1}, in all s'_P paths). To find the minimal equivalent path(s), we test s''_P against s_P for all subsequences P'' of P. We call *optimized TP* (**OTP**) the most efficient binding plan derived from all the resulting minimal sequences. This binding plan only applies the final dup-elim if it is necessary: that is, if two paths in s_P are related. We now compare the binding performance of: PS, TP, and OTP. TP and OTP use the best structural join ordering.

Figure 10 considers *lpe*s of length 2: P_8=//description//parlist, P_9=//parlist//listitem, P_{10}=//item//keyword, P_{11}=//person//name, P_{12} = //category//name on XMark; P_{13}=//article//title, P_{14}=//book//author and P_{15}=//book//title on DBLP; P_{16}=//METAL/Descr and P_{17}=//LIPID//Descr on SwissProt. For P_8 and P_9, PS and OTP coincide; TP performs one extra join and a dup-elim. From P_{10} to P_{17}, OTP only spares the dup-elim, which is cheap since the join result order was favorable. The performance difference between PS and OTP reflects the proportion of useless IDs scanned by OTP due to the tag-partitioned storage. This proportion reaches 430 for P_{14}; PS is very fast for P_{14}, P_{15} and P_{17}.

In Figure 11 we measured *lpe*s of length 3 (not listed for space reasons). (O)TP performance is determined by the data



T_1	$x in //person, $y in $x//name, $z in $x//watch	T_4	$x in //item, $y in $x//parlist, $z in $x//keyword
T_2	$x in //asia, $y in $x//item, $z in $x//description	T_5	$x in //categories, $y in $x//listitem, $z in $x//text
T_3	$x in //item, $y in $x//description, $z in $x//parlist	T_6	$x in //article, $y in $x//title, $z in $x//year

Figure 12: Binding 2- and 3-variable patterns.

scans; PS speed-up is of up to several hundreds.

Overall, the TP handicap grows with the length of the *lpe*, as this translates to potentially more useless ID scan.

Path summary processing. We have measured the time to apply the path inference, simplification, and GTP enrichment, as described in Section 3.2, on GTPs of up to 10 variables, derived from XMark queries. We found this time to be negligible; the longest value was 0.145 s.

6.3 Combining variable bindings

In this section, we study binding performance for GTPs of two or more variables. Figure 12 compares TP and PS performance when binding two, respectively, three related variables. Differently from the previous section, in these cases, TP cannot benefit from any optimization, since the scans and the structural join are required, in order to get binding pairs. Furthermore, PS must also scan many inputs and join them. Also, in Figure 12, PS and TS use the best possible join order.

In Figure 12 (left), we consider again the paths P_8 to P_{17} that we used in Figure 10, but this time we replace "for $x in //tag$_1$//tag$_2$" with "for $x in //tag$_1$, $y in $x//tag$_2$". For example, in Figure 12(top), P_8' stands for the two-variable pattern "for $x in //description, $y in $x//parlist". We notice that for P_8' and P_9', PS and TP are very close, since: TP makes no useless reads (all parlist elements are under description elements, and similarly for P_9'), and the cost is dominated by data access. For P_{10}' - P_{17}', the unselective scans of TP translate directly into huge performance differences with PS; the curves mirror quite closely those from Figure 10 (top). The biggest time we measured for the join in itself was 4.2 seconds, for P_{17}'.

In Figure 12 (right), we consider a set of three-variable patterns, T_1-T_5 on XMark and T_6 on DBLP, shown just underneath the corresponding graph.

While quite small, these patterns illustrate well the performance issues involved. We have chosen just 1-tag *lpes* to connect $x to $y and $z, to pick the best possible case for TP: if longer paths connect variables, TP will do useless scans and joins, as explained in the previous section.

Overall, scan, join, and total time are still smaller for PS than for TP. For T_1, T_3, T_4 and T_6, the join cost is negligible for both, compared to the data scan cost. In these cases, simply using a PS storage provides for faster binding. For T_2 and T_5, the joins by themselves take a more important part; even with the best ordering, the first join creates many intermediary results, on which the second join spends more time. This problem can be alleviated using holistic twig joins [8], which

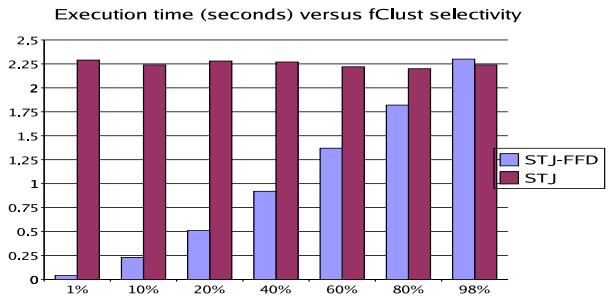


Figure 13: Descendent skipping (f_{clust}).

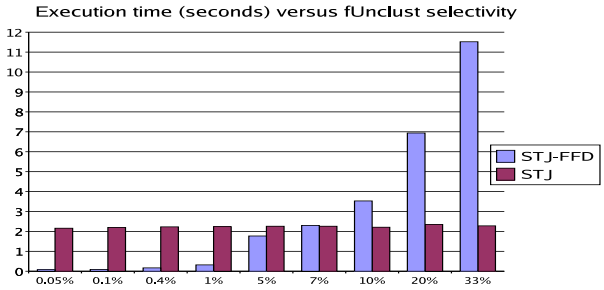


Figure 14: Descendent skipping ($f_{unclust}$).

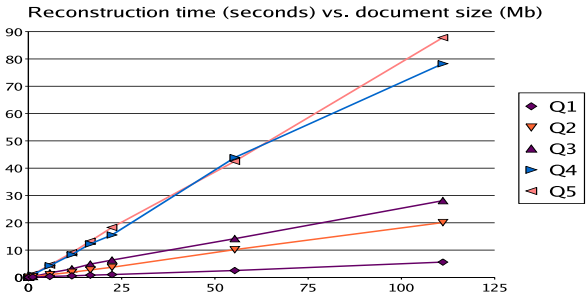


Figure 15: Performance of reconstruction queries.

reduce intermediary results in large structural join trees; however, holistic joins were proposed based on a TP storage, and they do not reduce data scans. Thus, the selective PS data access can be fruitfully combined with the efficiency of TP to improve overall query performance.

Impact of predicates. In the presence of predicates, binding a GTP can read *less data* than binding all variables separately, since a selective access to one of them can be used to skip inputs from the other ones (Section 4.2).

To assess skipping benefits, we use two controllable filter predicates. $F_{clust}(k)$ erases the first $(100-k)\%$ of its input, and lets all the last $k\%$ survive. On the contrary, $f_{unclust}(k)$ lets $k\%$ of its input tuples survive, but they are spread out evenly over the input, e.g., if k is 10, then 1 in every 10 successive tuples survives.

We first take op_1 as f_{clust}(IDScan(4), k) and op_2 be IDScan(5); this is equivalent to "for $x in //person[@id≥$\lambda$], $y in $x/name", where λ is (1-k/100)*N_4 (Figure 13). This is the optimal case for STJ-FFD: the join jumps over the first 100-$k\%$ tuples of op_2 in a single step. Furthermore, due to its "optimistic" skipping (Section 4.2), STJ-FFD never skips over the remaining op_2 tuples, since they all match op_1 IDs. The gains of STJ-FFD over STJ are directly proportional to the skipped input size (100-k)*$|op_2|$.

We now take op_1 as $f_{unclust}$(IDScan(4), k) and op_2 as before, equivalent to "for $x in //person[(@id mod 100)≥k], $y in $x/name" (Figure 14). This is the worst case for STJ-FFD: after every matching pair, it makes a skip, in $O(log(|op_2|))$. For very small selectivities, STJ-FFD keeps its advantages; starting from $k = 6$, the overhead of skipping makes STJ-FFD worse than STJ. We recorded very similar results for STJ-FFA, on many similar plans.

The heuristic we derive for the general case is: use skipping joins (STJ-FFA or STJ-FFD) when one input's estimated size is very small, e.g., for point queries.

6.4 XML data serialization

PS storage exhibits a high degree of "vertical" fragmentation: all IDs are separated from all their children IDs, and from their contents. To show that this fragmentation does not bring a bottleneck for many-join queries, we measured several *full*

reconstruction queries: from the designated elements, return all descendent elements and their contents (we are no longer just finding bindings !) Figure 15 shows the complete execution times, including data serialization and XML tagging, for the following *lpes*: Q_1=//person//name, Q_2=//person//profile, Q_3=//asia//item, Q_4=//closed_auction, and Q_5=//person. We adopted the partial materialization approach described in Section 5, since in these cases there are as many outerjoins as nodes in the path summary subtree designated by Q_i (from 1 for Q_1, to 59 for Q_4).

Reconstruction time scales up linearly with the document (and thus, the output) size. Reconstruction is quite efficient (about 1 minute for results of size several Mb. Holistic twig joins [8] could be employed to further reduce the time.

6.5 Conclusions

The PS storage model is very compact, since it has practically no overhead; it supports variable binding up to several orders of magnitude faster than TP, due to its precise structural knowledge. The ordered PS model enables skipping useles data, at the level of joins, but also larger plans. The target of PS optimizations (selective data access) and techniques like [8] (minimizing intermediary QEP results) are complementary, thus these optimizations can be fruitfully combined to improve query performance.

7 Related work

Several XML storage models have been proposed over the years [3]. Our work is placed in the context of XQuery processing based on structural joins. The work described in [1] also uses structural identifiers, but on a different storage, where (using computer clusters) all data is guaranteed to be in memory; in our approach, data is (selectively) read from disk, still providing good performance. The closest related works have been done within the Timber and Niagara systems [2, 28, 8, 9, 12], and are based on tag partitioning. A path-partitioned storage provides much better performance for variable binding plans, as explained in Section 6.2.1.

Our join branch elimination (Section 3.2) is an instance of constraint-based minimization [4]; $m_{x,y}$ and $M_{x,y}$ act as child constraints, while $mpf_{x,y}$ play the role of descendent constraints. More recent works studied XPath containment and simplification using schema information, such as a DTD or an XMLSchema [17, 18, 22]. While our path summary resembles a schema, there are some informations that a schema provides, while a path summary doesn't, for example, disjoint choices within an element's structure (an element can *either* have an a, *or* a b child). Thus, schema-driven techniques could be applied before/during our path inference process. The path summary is easy to construct, concise, and very useful for XQuery processing, especially for variable binding. Interestingly, 40% of the XML documents found on the Web have a DTD [16], and about 1% have an XML Schema; without schemas, path summaries still apply.

Techniques for input skipping in structural joins on pre were described using B+-tree indexes [29, 10], chained stacks [8], and spatial indexing methods [21]. Our STJ-FF avoids the same comparisons as in [29, 10], without a storage overhead, as we incorporate skipping into the execution model directly. Our first contribution in terms of skipping useless ID comparisons is path inference, which allows to construct joins on IDs from related paths only. This optimization can be composed with those in [21, 8]. The second contribution is skipping on post, under path-related conditions.

GeX started as an XML querying and compression engine [5]; the work reported here is independent of compression.

8 Conclusion and future work

We have presented the path sequence storage model, a compact and fragmented model for XML storage. The path sequence model improves over similar systems by preserving precise path information, in the storage as well as under the form of a path summary that is extensively used for query optimization. Our model is fully implemented within the GeX system; we validated its performance advantages through a series of experiments.

In the future, we envision extending GeX with support for updates, and schemas. For what concerns updates, using the path inference algorithm described in Section 3.2 on an XML declarative update statement, it is possible to infer if the update will create new paths, or move nodes from some paths to others. Thus, although the path summary is derived from the data, it is possible to maintain it through document updates. With respect to schemas, we plan to find a simple compromise between schema language conciseness, and difficulty of query minimization, based on the results of [18, 17].

Acknowledgements. The authors are extremely grateful to Alberto Lerner, for many useful comments on this paper.

References

[1] Vincent Aguilera, Frederic Boiscuvier, Sophie Cluet, and Bruno Koechlin. Pattern tree matching for XML queries. Gemo Technical Report number 211, 2002.

[2] 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.

[3] S. Amer-Yahia. Storing techniques and mapping schemas for XML. Submitted for publication, 2003.

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

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

[6] The BerkeleyDB database library. http://www.sleepycat.com.

[7] Omar Boucelma, Mehdi Essid, Zoe Lacroix, Julien Vinel, Jean-Yves Garinet, and Abdelkader Betari. VirGIS: Mediation for geographical information systems. In *ICDE*, 2004.

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

[9] 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.

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

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

[12] 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.

[13] A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, 2003.

[14] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *SIGMOD*, 2000.

[15] A. Marian and J. Simeon. Projecting XML documents. In *VLDB*, 2003.

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

[17] J. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, 2003.

[18] F. Neven and T. Schwentick. XPath containment in the presence of disjunctions, DTDs, and variables. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, 2003.

[19] Tuyet Tram Dang Ngoc and Georges Gardarin. Evaluating XQuery in a full-XML mediation architecture. In *Proc. Journées Bases de Données Avancées*, 2003.

[20] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *VLDB*, Cairo, Egypt, 2000.

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

[22] P. Wood. Containment for XPath fragments under DTD constraints. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, 2003.

[23] Y. Wu, J.M. Patel, and H.V. Jagadish. Estimating answer sizes for XML queries. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 2002.

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

[25] The XQuery language. www.w3.org/TR/xquery, 2003.

[26] The SAX project. www.saxproject.org.

[27] XQuery 1.0 formal semantics. www.w3.org/TR/2004/WD-xquery-semantics.

[28] H. Jagadish Y. Wu, J. Patel. Structural join order selection for XML query optimization. In *ICDE*, 2003.

[29] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in RDBMSs. In *SIGMOD*, 2001.