World Wide Web (2008) 11:117–151
DOI 10.1007/s11280-007-0036-7

Path Summaries and Path Partitioning in Modern XML Databases

Andrei Arion · Angela Bonifati · Ioana Manolescu · Andrea Pugliese

Received: 6 February 2006 / Revised: 1 June 2007 /
Accepted: 15 June 2007 / Published online: 6 September 2007

Abstract XML path summaries are compact structures representing all the simple parent-child paths of an XML document. Such paths have also been used in many works as a basis for partitioning the document's content in a persistent store, under the form of path indices or path tables. We revisit the notions of path summaries and path-driven storage model in the context of current-day XML databases. This context is characterized by complex queries, typically expressed in an XQuery subset, and by the presence of efficient encoding techniques such as structural node identifiers. We review a path summary's many uses for query optimization, and given them a common basis, namely *relevant paths*. We discuss summary-based tree pattern minimization and present some efficient summary-based minimization heuristics. We consider relevant path computation and provide a time- and memory-efficient computation algorithm. We combine the principle of path partitioning with the presence of structural identifiers in a simple path-partitioned storage model, which allows for selective data access and efficient query plans. This model improves the efficiency of twig query processing up to two orders of magnitude over the similar

A. Arion · I. Manolescu
INRIA Futurs–LRI, Orsay, France
e-mail: Andrei.Arion@inria.fr

A. Bonifati
ICAR CNR, Palermo, Italy
e-mail: bonifati@icar.cnr.it

I. Manolescu (✉)
INRIA Futurs, Gemo group, 4 rue Jacques Monod, ZAC des Vignes,
91893 Orsay Cedex, France
e-mail: Ioana.Manolescu@inria.fr

A. Pugliese
University of Calabria, Rende, Italy
e-mail: apugliese@deis.unical.it

tag-partitioned indexing model. We have implemented the path-partitioned storage model and path summaries in the XQueC compressed database prototype [8]. We present an experimental evaluation of a path summary's practical feasibility and of tree pattern matching in a path-partitioned store.

Keywords path summaries · path partitioning · XML databases

1 Introduction

Path summaries are classical artifacts for semistructured and XML query processing, dating back to 1997 [24]. The path summary of an XML document is a tree, such that every path from the root to some (internal or leaf) node in the document appears exactly once in the summary. The original DataGuide proposal [24] made a clean distinction between the DataGuide (or summary), which only *describes* the structure of the data and can be thought of as a schema extracted from the data, and the *data store* itself.

Path summaries inspired the idea of path indexes [39]: the IDs of all nodes on a given path are clustered together and used as an index. Observe that while the index holds actual data, the summary can be seen as a catalog of this data. The summary is exploited at query compile time for query optimization purposes. The path index may be accessed at query execution time, if the optimizer deems it useful. Subsequent works have used paths as a basis for clustering the storage itself [8, 11, 28, 57] and as a support for statistics [1, 33].

The state of the art of XML query processing advanced significantly since path summaries were first proposed. The current availability of a standard query language for XML, namely XQuery [52], is one significant change. Another important change comes from the current availability of many node identification schemes with interesting properties, allowing, for instance, to determine if two nodes are structurally related just by examining their identifiers. Such identifiers are commonly called *structural IDs* [2, 40, 48], and based on them, efficient structural joins [2, 13, 49] techniques have been developed, enabling efficient processing of XML navigation as required by XPath and XQuery.

We revisit the notions of path summaries and path-driven storage model in the context of current-day XML databases.

The principle of path partitioning, combined with structural identifiers, leads to a simple *path-partitioned storage model*. We had introduced this model in the context of the XQueC compressed XML database [7, 8], but in the present paper we consider its benefits independent of data compression issues.

We review a path summary's many uses for query optimization, such as access method selection, query reformulation, and physical query optimization, and identify the concept of *relevant paths* as a common basis for all these optimizations. We consider in more detail two particular optimization techniques:

- The first technique is *query tree pattern minimization* under the structural constraints encapsulated in a path summary. We outline the differences between this problem and similar schema-based minimization problems addressed by previous works. We provide a simple minimization algorithm which, like previous algorithms, works by erasing nodes from the original pattern. Previous

works left open the question of whether such node-erasing algorithm can find the smallest possible pattern equivalent to the original pattern under the considered constraints [16]. We show that under summary constraints, smaller equivalent patterns may exist than those which can be found by erasing original pattern nodes. This finding opens an interesting direction for future work.

- The second technique involves exploiting relevant paths for *choosing the data access methods* to be used to answer a query. In its most general form, this problem is an instance of query rewriting based on materialized views, under the constraints encapsulated by a summary. This problem has been addressed in a separate work [35] and the resulting algorithms are quite complex. We show that in the particular case of a path-partitioned storage model, one can easily build query plans exhibiting selective data access and low memory needs during processing.

All the above optimizations require the computation of relevant paths for a query, based on a path summary. We describe a time- and memory-efficient relevant path computation algorithm and show it improves over the simple algorithms previously considered.

While a strongly fragmented path-partitioned store leads to good pattern matching performance, it significantly complicates the opposite task which is to construct complex XML subtrees. While existing algorithms can be adapted to the task, they tend to have high memory needs and are blocking. Therefore, we describe a physical operator specifically suited to the path-partitioned model, which is pipelined and thus eliminates these drawbacks.

This paper is organized as follows. Section 2 formally introduces path summaries and path partitioning. Section 3 addresses query optimization based on path summaries, introduces the notion of relevant paths and provides time- and memory-efficient algorithm for computing them. Section 4 outlines query planning based on relevant paths and focuses on the particular case of a path-partitioned store. We have implemented the path-partitioned storage model and path summaries in the XQueC compressed database prototype [8]. Subsequently, to be able to exploit the path summary beyond the XQueC context, we separated our summary implementation in a standalone library [55]. Section 5 studies the practical feasibility of path summaries, the performance of our relevant path computation algorithms, and the performance of tree pattern evaluation on a path-partitioned store. Section 6 compares our work with related works, then we conclude.

2 Path summaries and path partitioning

In this section, we briefly revise the XML data model we consider and outline structural node identifiers, on which we will rely in the sequel. Then, Section 2.2 describes path summaries, while Section 2.3 focuses on the path-partitioned XML storage model.

2.1 Preliminaries

We view an XML document as a tree $(\mathcal{N}, \mathcal{E})$, such that $\mathcal{N} = \mathcal{N}_d \cup \mathcal{N}_e \cup \mathcal{N}_a$, where $\mathcal{N}_d$ is a set consisting of exactly one document node, $\mathcal{N}_e$ is a non-empty set of element

nodes and $\mathcal{N}_a$ is a set of attribute nodes. The set of edges $\mathcal{E}$ respects the restrictions imposed by the XML specification [50]. Thus, the node in $\mathcal{N}_d$ is the tree root and it has exactly one child which belongs to $\mathcal{N}_e$. The parent of any node in $\mathcal{N}_e \cup \mathcal{N}_a$ belongs to $\mathcal{N}_e$. For ease of exposition, in the sequel we will ignore the document node and refer to the unique $\mathcal{N}_e$ child of the document node as the document's root.

All $\mathcal{N}$ nodes are endowed with a unique *identity* and with a *node label*, viewed as a string. Furthermore, all $\mathcal{N}$ nodes have a *value*, which is also a string. The value of an element node is the result of applying the XPath function *text()* on the node [52]. If an element has multiple text descendant nodes, the *text()* function concatenates them, losing the information about their number and relative order in the document. To avoid this, a simple extension to our model consists of making text nodes first-class $\mathcal{N}$ citizens, endowed with identity. To keep our presentation simple, we omit this in the sequel. Figure 1a depicts a sample XML document, which is a simplified instance of an XMark [51] benchmark document.

To represent node identity, XML databases use *persistent identifiers*, which are values uniquely identifying a node within a document (or, more generally, within the whole database). We can view node identifiers (or, in short, IDs) as values obtained by applying an injective function $f : \mathcal{N} \to \mathcal{A}$, where $\mathcal{A}$ is a set of values. We say the identifiers assigned by a given function f are *structural* if, for any nodes n_1, n_2 belonging to the same document, we may decide, by comparing $f(n_1)$ and $f(n_2)$, whether n_1 is a parent/ancestor of n_2 or not. A very popular structural identifier scheme is based on tree traversals, as follows. (1) Traverse the XML tree in pre-order and assign increasing integer labels 1, 2, 3 etc. to each encountered node. We call this integer the pre label. (2) Traverse the XML tree in post-order and similarly assign increasing integer labels to each encountered node. We call this the post label. (3) Assign to each node a depth label corresponding to its depth in the tree. Thus, the depth of the root is 1, the depth label of the root's children nodes is 2 etc. (4) The identifier of each node in the XML document is the triple of its pre, post, depth labels. This scheme was introduced in [2] for XML documents and often used subsequently [17, 26]. For example, Figure 1a depicts (pre,post) IDs above the elements. The depth number is omitted in the figure to avoid clutter.

Many variations on the (pre,post,depth) scheme exist. A simple variant assigns to each element its start and end positions in the document, obtained as follows. All opening and closing tags in the document are numbered in the order in which

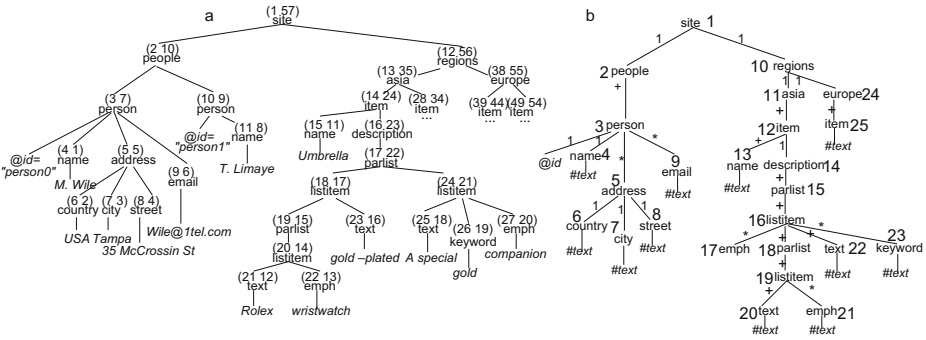


Figure 1 XMark document snippet and its path summary.

they are found in the document 1,2,3 etc. and each element is given as start number the number of its opening tag, and as end number the number of its closing tag. More advanced structural IDs have been proposed, such as DeweyIDs [48] or ORDPATHs [40]. While we use (pre,post,depth) for illustration, the reader is invited to keep in mind that any structural ID scheme can be used.

2.2 Path summaries

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

1. ϕ maps the root of D into the root of $S(D)$. 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 $S(D)$ 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 a 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 a unique child n_a labeled $@a$ and for each $n_i \in att(n, a)$, we have $\phi(n_i) = n_a$.

Clearly, ϕ preserves node labels and parent-child relationships.

Paths and path numbers Let *rooted path* denote a path of the form $/l_1/l_2/.../l_k$ going from the root to some (leaf or non-leaf) node of the document. In the sequel, for simplicity, *path* is used to refer to a rooted path, unless otherwise specified. For every path in D, there is exactly one node reachable by the same path in $S(D)$. Conversely, each node in $S(D)$ corresponds to a path in D.

Figure 1b shows the path summary for the XML fragment at its left. We assign an integer number to every summary path; these numbers appear in large font next to the summary nodes, e.g. 1 for /site, 2 for /site/people etc. Thus, there is one number for each summary node and for each path present in the document or summary; we may interchangeably refer to one by means of another. For instance, we may say that the XML node identified by (2,10) in Figure 1a is on path 2 in Figure 1b.

Path summaries as defined above and used in this work correspond to strong DataGuides [24] in the particular case of tree-structured data (recall that DataGuides were originally proposed in the context of graph-structured OEM data [24, 41]).

Similar XML documents may have the same summary. We say a document D *conforms to a summary* S_0, denoted $S_0 \models D$, if $S(D) = S_0$. Thus, a given summary may be used as a repository of structural information concerning several documents in a database.

We add some more structural information to summaries, as follows. Let S be a summary. An *enhanced summary* S' *based on* S is a tree obtained by copying S, and labeling each edge with one symbol among 1, + and *. We say a document D conforms to the enhanced summary S' thus defined if $S \models D$ and furthermore, for any x node in S' and y child of x:

- If the edge $x - y$ is labeled 1, then all D nodes on path x have exactly one child on path y;

- If the edge $x - y$ is labeled +, then all D nodes on path x have at least one child on path y;
- If the edge $x - y$ is labeled *, then no information is available concerning the number of y children of nodes on path x.

Edge labels in enhanced summaries play the role of cardinality constraints and are used for query optimization, as Section 3 will show.

Building and storing summaries For a given document, let N denote its number of element and attribute nodes and $|S|$ the number of nodes in its path summary. In the worst case, $|S| = N$, raising the possibility that summaries may be quite large. In Section 5, we present empirical evidence that in many practical cases, the summary is much smaller than the document. A summary (without edge annotations) is built during a single traversal of the document in $\Theta(N)$ time, using $\Theta(|S|)$ memory [1, 24].

Summaries must be serialized and stored in the XML database's catalog, using some tree serialization format. Let S be a summary and x, y and z be S nodes, such that x is the parent of y and x is an ancestor of z. Recall that x, y and z also designate S label path starting from the root, thus we will use them in the remainder of this section to refer to these paths. Edge annotations can be stored according to one of the following alternatives:

1. A simple way to encode the symbol annotating the edge connecting x and y is to associate this symbol to node y and include it in y's serialization. We call this encoding of edge annotations *direct encoding*. This encoding is simple and compact, yet it has some disadvantages. Consider a document D such that $S \models D$ and let n_x be a D node on path x (for each path in D, there is a node in S on the same path, which we use to designate the path itself). To know how many descendants on path z can n_x have, we need to inspect the annotations of all summary nodes between x and z.
2. Let u, v be two summary nodes. The *up-and-down path from u to v* is the shortest sequence of S edges connecting u to v. If u is an ancestor of v, the up-and-down path from u to v follows the edges going down from u to v. Otherwise, the up-and-down path from u to v goes *up* from u to the closest common ancestor of u and v, then *down* from that ancestor to v (thus the name).
   We define the *1-partition* of a summary S to be a partition of its nodes, such that two S nodes belong to the same partition set *iff* all edges on the up-and-down path between the two nodes are annotated with 1. It can be easily shown that the 1-partition of a summary is unique. Similarly, we define the *+-partition* of S as a partition of the nodes of S, such that two S nodes belong to the same partition set *iff* all edges on the up-and-down path connecting them are annotated with 1 or +. The +-partition of a summary S is also unique. We assign distinct integer numbers to each 1-partition set and similarly distinct numbers to each +-partition set. We annotate every S node with:

   - The number of the 1-partition set this node belongs to, denoted n1 and
   - The number of the +-partition set this node belongs to, denoted n+.

   We serialize each node's n1 and n+ numbers with the node. We call this approach for encoding summary edge annotations *pre-computed encoding*; it simplifies

```
<site symbol="1">
 <people symbol="1">
  <person symbol="+">
   <@id symbol="1"> <name symbol="1" />
   <address symbol="*">
     <country symbol="1" /> <city symbol="1" />
     <street symbol="1" />
   </address>
  </person></people>
 <regions symbol="1">...
```

```
<site n1="1" n+="1">
 <people n1="1" n+="1">
  <person n1="2" n+="1">
   <@id n1="2" n+="1"> <name n1="2" n+="1" />
   <address n1="3" n+="2">
     <country n1="3" n+="2" /> <city n1="3" n+="2"
     <street n1="3" n+="2" />
   </address>
  </person></people>
 <regions n1="1" n+="1">...
```

Figure 2 Sample summary serializations: *at left*, direct encoding; *at right*, pre-computed encoding.

summary-based reasoning in the following sense. Consider, as previously, a summary S, an S node x and let z be a descendent of x in S. Let D be a document such that $S \models D$ and let n_x be a D node on path x. If (and only if) $x.\mathsf{n1}{=}z.\mathsf{n1}$, then n_x has exactly one descendant on path z. Similarly, n_x has at least one descendant on path z if $x.\mathsf{n+}{=}z.\mathsf{n+}$.

Figure 2 exemplifies direct and pre-computed encoding of edge annotations on the summary in Figure 1b. In Figure 2, each summary node is serialized as an XML element. Clearly, other serialization methods may be used.

Direct encoding of edge annotations requires thus 2 bits per summary node if direct encoding is used and $2 \times \lceil log_2(|S|) \rceil$ bits per summary node if pre-computed encoding is used. The presence of edge annotations does not affect the time or space complexity of summary construction; if pre-computed encoding is applied to edge annotations, the time complexity increases by $\Theta(|S|)$ since an extra pass on the summary is needed.

2.3 The path-partitioned storage model

Path summaries do not store XML document contents. However, a simple storage model for XML documents can be devised based on the document's paths and structural identifiers as follows.

Let D be an XML document. Each D node is assigned a structural ID. The *path-partitioned storage model* consists of materializing two sets of storage structures:

- For each path, an *ID path sequence*, which is the sequence of the structural IDs of all D nodes on that path, in document order.
- For each path an *ID and value sequence*, which is the sequence comprising the *(structural ID, value)* pairs obtained from all D nodes on that path, in document order.

The above model is complete, i.e., it stores all the information needed to describe an XML document, in the sense of the data model described in Section 2.1. Figure 3 depicts some of the path-partitioned storage structures resulting from the document in Figure 1.

XML storage structures similar to the path partitioned model have been used in many previous works; we outline the differences between these models and ours in Section 6.

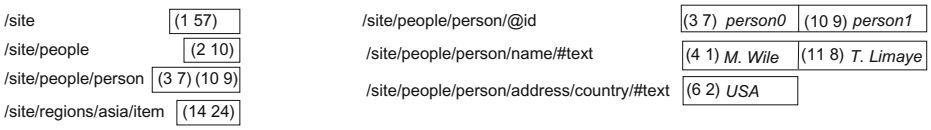


Figure 3 Sample path-partitioned storage structures.

3 Path summary-based XML query optimization

In this section, we consider summary usage for XML query optimization. Section 3.1 briefly describes the query language and query patterns we consider. Section 3.2 outlines the classes of optimizations enabled by summaries and casts them in a common light based on the notion of *relevant paths* for a query pattern. Section 3.3 delves into one particular summary-enabled optimization, namely pattern minimization under constraints. This problem is related to schema-based minimization, but as we show, there are interesting differences. Finally, Section 3.4 addresses the performance issues raised by relevant path sets computation and provides efficient algorithms to this purpose.

3.1 Query language and query patterns

Query language We consider the XQuery subset $\mathcal{Q}$ characterized as follows.

(1) $\text{XPath}^{\{/,//,*,[\,]\}} \subset \mathcal{Q}$, that is, any core XPath [38] belongs to $\mathcal{Q}$. We allow such XPath expressions to end with a call to the function *text()*, returning the text value of the node it is applied on. This represents a subset of XPath's absolute path expressions, whose navigation starts from the document root. Navigation branches enclosed in [ ] may include complex paths and comparisons between a node and a constant c. Predicates connecting two nodes are not allowed in this class; they may be equivalently expressed in XQuery for-where syntax (see below). (2) Let $\$x$ be a variable bound in the query context [52] to a list of XML nodes and p be a core XPath expression. Then, $\$x\ p$ belongs to $\mathcal{Q}$ and represents the path expression p applied with $\$x$'s bindings list as initial context list. This class captures *relative* XPath expressions in the case where the context list is obtained from some variable bindings. We denote the set of expressions (1) and (2) above as $\mathcal{P}$, the set of path expressions. (3) For any two expressions e_1 and $e_2 \in \mathcal{Q}$, their concatenation, denoted e_1, e_2, also belongs to $\mathcal{Q}$. (4) If t is a tag and $e \in \mathcal{Q}$, element constructors of the form $\langle t\rangle\{e\}\langle/t\rangle$ belong to $\mathcal{Q}$. (5) All expressions of the following form belong to $\mathcal{Q}$:

$$\begin{aligned}&\text{for } \$x_1 \text{ in } p_1, \$x_2 \text{ in } p_2, \ldots, \$x_k \text{ in } p_k\\ &\text{where } p_{k+1}\ \theta_1\ p_{k+2} \text{ and } \ldots \text{ and } p_{m-1}\ \theta_l\ p_m\\ &\text{return } q(x_1, x_2, \ldots, x_k)\end{aligned}$$

where $p_1, p_2, \ldots, p_k, p_{k+1}, \ldots, p_m \in \mathcal{P}$, any p_i starts either from the root of some document D or from a variable x_l introduced in the query before p_i, $\theta_1, \ldots, \theta_l$ are some comparators, and $q(x_1, \ldots, x_k) \in \mathcal{Q}$. Observe that a return clause may contain other for-where-return queries, nested and/or concatenated and/or grouped inside constructed elements.

A $\mathcal{Q}$ query may perform one or both of the following types of processing: (1) *retrieving*, from the input data, the nodes that should be part of the result; this step may involve complex navigation, evaluation of value predicates etc. and always produces a set of nodes or values from the input document; (2) *constructing* new elements, typically including copies of the elements produced by step (1). Tree patterns are commonly used to represent step (1). In the sequel, we outline the XAM (*XML access modules*) tree pattern formalism we introduced in [4], on which this work relies, and justify why we use it instead of similar formalisms.

XAMs A XAM is a tree pattern of the form $t = (n_t, e_t)$, where n_t is a set of XAM nodes and e_t is a set of XAM edges. We distinguish two types of n_t nodes: element nodes and attribute nodes. The specification of each n_t node consists of: a (possibly empty) set of *predicates*, constraining the XML nodes that may match the XAM node, and a (possibly empty) set of *projected attributes*, specifying which information is retained by the pattern from every XML node matching the XAM node.

The predicates annotating a XAM node can be of the form: (1) $[Tag = c]$, where c is a node label, denoting the fact that only XML nodes having the label c will match the node; (2) $[Val\ \theta\ c]$, when c is a constant and θ is a simple comparison operator such as $=, <, \leq$ etc., denoting the fact that only XML nodes whose value satisfies this predicate will match the node.

Each XAM node can specify between zero and four information items, that the pattern retains from every XML node matching that node. The XAM node labels Tag, Val, ID and $Cont$ denote the fact that the labels (respectively, values, identifiers, or contents) of the XML nodes matching the n_t node are retained (projected) by the XAM. The contents of an XML node denotes the full subtree rooted at that node.

Each XAM edge is specified by a simple (respectively double) line to denote parent-child relationships (respectively ancestor-descendant) relationships. Moreover, each edge is specified by a solid line to denote mandatory children (or descendants), or a dashed line to denote optional children (or descendants). Finally, each edge may be labeled n to denote that matches for the child (resp. descendant) node at the lower end of the edge are nested inside the match for the parent (resp. ancestor) node at the higher end of the edge.

The semantics of a XAM pattern t on a document D, denoted $t(D)$, is a (potentially nested) relation, whose schema is determined by t, and whose content is extracted from D. The semantics is formally defined in [4]. To make this paper self-contained, we illustrate this via three examples in Figure 4, showing the semantics of each pattern on the document in Figure 1 as a table underneath the pattern. Pattern t_1 has only mandatory edges, thus only one person element contributes data to $t_1(D)$. In t_2, the optional edge allows the second person element in Figure 1 to contribute, although it does not have a child with a street descendant. Observe that node 3 is optional with respect to node 2, and node 4 is mandatory with respect to node 3. The missing child leads to a null ($\perp$) in $t_2(D)$. Pattern t_3 has a nested edge, and accordingly the second attribute of $t_3(D)$ is a nested table, containing the values of the text descendants of every /site//asia//listitem element.

XAMs resemble generalized tree patterns (or GTPs) [16], however there are important differences between the two. (1) The main difference is that a GTP for a nested XQuery may consist of several nested blocks, one per for-let-where-return

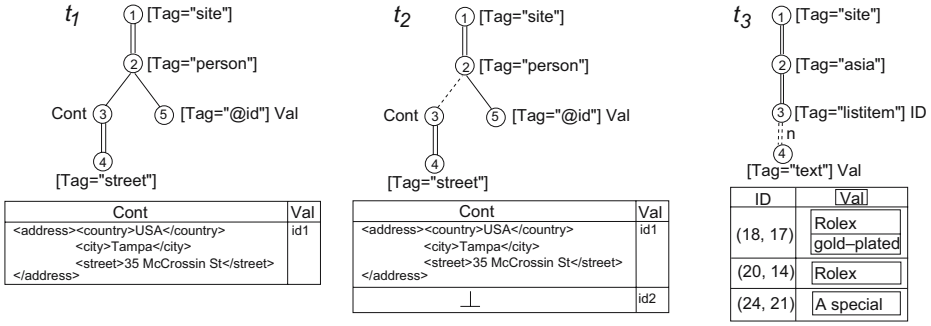


Figure 4 Sample XAM patterns and their semantics on the XML document in Figure 1.

(FLWR) block, whereas a single XAM can capture several nested FLWR blocks. As explained in [6], capturing a larger part of an XQuery with a single pattern is preferable, since it allows to take a bigger advantage of efficient tree pattern evaluation methods (such as e.g. using a materialized view or index matching the pattern). In the context of the present paper, larger patterns are preferable as they provide a larger scope for the static analysis we perform, based on the path summary. (2) GTPs do not specify which information is retained from XML nodes matching pattern nodes, as XAMs do. However, from an optimizer's viewpoint, there is significant difference between a query such as //a/text() which only requires the values (*Val*) of a elements and may be answered e.g. by scanning a node value index, and the query //a, which requires the full element contents (*Cont*). (3) GTPs include universal quantifiers and value joins, while XAM patterns focus only on the structured navigation performed by queries, which can be exploited for summary-based static analysis. Abstract Tree Patterns (APTs) [43] edges generalize GTP (and XAM) edges by specifying how many matches for a child pattern node are allowed for a given match of the parent node (exactly one, at least one, zero or more etc.) We rely on XAMs for this work, mainly due to the differences denoted (1) and (2) above, which still hold between APTs and XAMs.

3.2 Summary-based static analysis on query patterns

In this section, we outline several usages of a path summary as a static analysis tool for query optimization. We consider a summary S (with 1 and + edge annotations) and a query $q \in \mathcal{Q}$ and assume one or several XAM patterns have been extracted from q as described in [6]. Each query pattern is typically analyzed in isolation. Thus, without loss of generality, we consider only one XAM pattern t.

We say t is S-satisfiable if there exists a document D_t such that $S \models D_t$ and $t(D_t) \neq \emptyset$ (where $\emptyset$ denotes an empty table).

Let t be an S-satisfiable pattern. We say a path $p \in S$ is *relevant* for a node $x \in t$ if and only if for some document D such that $S \models D$, and some D node n_x on path p, n_x matches the pattern node x. Observe that the notion of path relevance does not depend on the value predicates of t.

Several paths may be relevant for a given pattern node. For example, Figure 5 shows a sample query, its corresponding pattern, and a table comprising the set of

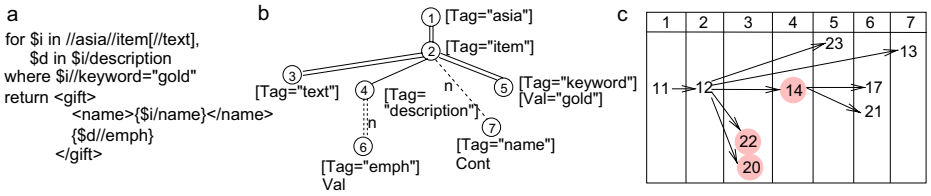


Figure 5 Sample query, resulting pattern, and relevant paths for the pattern and the summary in Figure 1.

relevant paths from the summary depicted in Figure 1 for each pattern node. Ignore for now the shaded bubbles in the Figure; they will be discussed in Section 3.3. Relevant path sets are organized in a tree structure, mirroring the relationships between the nodes to which they are relevant in the pattern.

The S paths relevant for the nodes of a pattern t can be used in many ways:

1. The original DataGuide work [24] exploits them to replace wildcards (node labels or node paths left unspecified by the user) by precise paths.
2. Relevant path sets can be used to perform query pattern minimization under summary constraints, in the style of minimization under integrity constraints [3] or under schema constraints [16]. While the principles are similar, the classes of considered constraints and the minimized patterns attainable, are different. We discuss this in Section 3.3.
3. When summaries are used in conjunction with a path index or a path-partitioned store, the relevant paths for the tree pattern node are used to build data access plans [8, 11, 28, 39, 57].
4. Formalisms like XAM patterns can be used to describe a large variety of the existing proposals for XML stores, indexes, and materialized views, as well as custom (user-defined) materialized views over a given document D [4]. In this context, the (possibly nested) table $t(D)$ is the data actually stored in a persistent structure (e.g. table in a relational store, index in a native system etc.) The relevant path sets of a storage pattern t_s can be used:

   - To infer properties of the storage structure interesting for physical query optimization, as [16] suggests based on schemas;
   - Together with the relevant path sets of a query pattern t_q to determine the usefulness of t_s for the query from which t_q was obtained. This generalizes the analysis performed for path-partitioned stores or indices to storage structures described by arbitrary XAMs.

   In the following section we are going to focus on tree pattern minimization while optimizations along the lines of 3 and 4 above will be discussed in more detail in Section 4.

3.3 Tree pattern minimization under summary constraints

We say two patterns t_1, t_2 are *equivalent under the constraints of S,* denoted $t_1 \equiv_S t_2$, if and only if $t_1(D) = t_2(D)$ for any document D such that $S \models D$. Observe that this

equality only makes sense if t_1 and t_2 have the same nested tables signature (leading to the same number of attributes). More generally, one can only judge whether $t_1 \equiv_S t_2$ for a given isomorphism ϕ between the returned nodes of t_1 (those labeled with ID, Val or $Cont$) and the returned nodes of t_2.

Let t be a pattern and t' be a pattern obtained from t by erasing one t node and re-connecting the remaining nodes among themselves. If $t' \equiv_S t$, we say t' is an *S-contraction* of t[1]. A pattern t is said to be *minimal under S-contraction* if and only if no pattern t' obtained from t via S-contraction satisfies $t \equiv_S t'$.

The process of *minimizing a pattern t by S-contraction* consists of finding all the patterns t', minimal under S-contraction, which can be derived from t. Observe that several such patterns t' may exist. For instance, in Figure 6, the patterns t'_1 and t'_2 are the result of minimizing the pattern t by S-contraction. No pattern obtained from t'_1 or t'_2 by S-contraction is still equivalent to t. (Notice that a pattern of the form $//a//d//e$, obtained by erasing one node from t'_1 or t'_2, is no longer equivalent to t, since it also returns e nodes on the path $/a/d/e$, which do not belong to t).

It turns out that S-contraction does not always yield the smallest possible patterns S-equivalent to t (where pattern p_1 is smaller than p_2 if p_1 has fewer nodes than p_2). For instance, in Figure 6, the pattern t'' is smaller than both t'_1 and t'_2, yet $t \equiv_S t''$. The intuitive reason is that the summary brings in *more* nodes than are available in the original pattern. The process of *minimizing a pattern t under S constraints* consists of finding all patterns t' that are S-equivalent to t, and such that no pattern t'' smaller than t' is still S-equivalent to t. This may yield smaller patterns than those found by S-contraction only.

In general, there may be more than one such smallest equivalent pattern. For instance, if we modify the summary in Figure 6 to add a g node between the f node and its child e, the pattern t''' corresponding to the query $//a//g//e$ is also S-equivalent to t, smaller than t'_1 and t'_2, and t''' has the same size as t''.

No pattern smaller than t'' and still S-equivalent to t can be found.

To keep the example simple, in Figure 6 we assumed no edges are annotated + or 1. Clearly, such annotations also provide opportunities for minimization, as we will show next.

Let us compare minimization under S constraints with minimization considered in previous works.

Tree pattern minimization (without constraints) yields a unique minimal pattern and can be performed in polynomial time in the size of the pattern [3]. In the presence of constraints of the form "every a element has a b child (or descendant)", minimization remains polynomial in the size of the pattern and a unique minimal pattern exists [3]. A polynomial algorithm for minimization under child and descendant constraints is also outlined in [16], and it yields a unique solution.

A different class of constraints introduced in [16] has the form "between every a element and its c descendant, there must be a b element". Minimization under such constraints alone leads in polynomial time to a unique solution [16]. Minimization under such constraints *and* child and descendant constraints is shown to lead, in some cases, to several equal-size patterns [16]; the minimization algorithm of [16] only

[1] Observe that $t' \equiv_S t$ requires that no returned node of t has been erased.

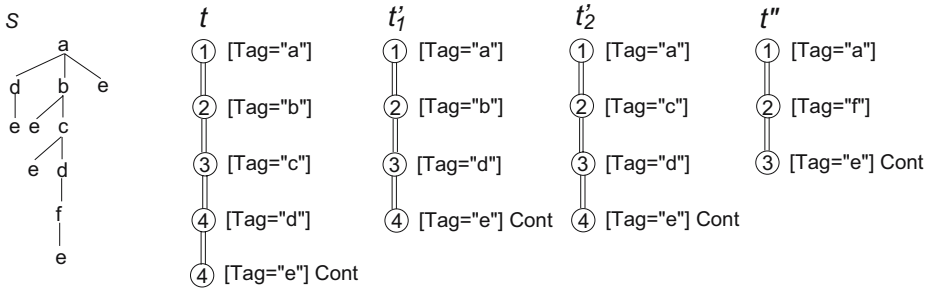


Figure 6 Sample summary, a pattern t and some smaller, equivalent patterns.

applies contraction (erases nodes from the initial pattern). The question whether a pattern smaller than these exists is left open [16].

The differences between such minimization algorithms and the ones enabled by a summary are well illustrated by Figure 6. This example shows that minimization under S constraints, whether by S-contraction or in the general case, does *not* lead to a unique solution. The question left open in [16] is answered by t'' in Figure 6: smaller patterns than can be found by contraction (as described in [16]) do exist, and there may be several such patterns of equal size.

It is also worth pointing out that summary constraints have finer granularity than child and descendant constraints at tag level, considered in [3, 16]. Consider e.g. a constraint like "between every b element and its d descendant, there must be a c element", implied by the summary in Figure 6, and which could be exploited by the algorithm in [16]. If we add a d child to the b summary node, the constraint no longer holds, thus the algorithm of [16] cannot even find t'_1. However, t'_1 would still be an S-equivalent rewriting of t under the constraints of the modified S.

We have thus identified two minimization problems under summary constraints, which, to our knowledge, have not been considered in previous works: minimization by contraction and general minimization (which may bring in nodes not present in the original pattern, such as the t'' node labeled f in Figure 6).

A simple algorithm for minimization by S-contraction consists of trying to erase pattern nodes (that are not annotated with $Val\,\theta\,c$ or with Val, ID nor $Cont$), and checking if the pattern thus obtained is still equivalent to the original one. An algorithm for deciding pattern equivalence under summary constraints is provided in [35]. However, performing a large number of equivalence tests may be considered expensive, making some faster techniques desirable. Pattern minimization under S constraints in the general case is an area where we expect to work in the future.

We now show how two simple minimization techniques (using *useless paths* and *trivial existential node paths*) , initially proposed in [16] based on constraints at tag level, can be exploited to reduce pattern size under summary constraints. These techniques are interesting as they can be very efficiently applied based on relevant path sets (for which, in turn, we developed efficient computation algorithms). Thus, *some* minimization is applied at a very low price, piggy-backed on the efficient process of relevant path computation (which has many other useful uses).

We recall to the reader's attention the pattern and its relevant paths depicted in Figure 5.

Useless paths The path 14, relevant for the pattern node 4, has no impact on the query result, on a document conforming to the path summary in Figure 1. This is because: (1) the pattern node 4 is not annotated with *Val* or *Cont*, thus its data is not returned by the query; (2) it follows from the path summary that every element on path 12 (relevant for the pattern node 2) has exactly one child on path 14. This can be seen by checking on the summary annotations (recall Figure 1). Thus, query evaluation does not need to find bindings for the variable $d (to which the pattern node 4 corresponds).

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

1. n_x is not annotated with *Val* or *Cont*;
2. If n_x has a parent n_y in the query pattern, let p_y be the path relevant for n_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, n_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. In Figure 5, once 14 is found useless, 12 will point directly to the paths 17 and 21 in the relevant set for the pattern node 6.

Trivial existential node paths We say a pattern node is *existential* if neither this node nor its descendants in the pattern are annotated with *Val*, *Cont*, or a value predicate.

The path summary in Figure 1 guarantees that every XML element on path 12 has at least one descendant on path 22. This is shown by the 1 or + annotations on all paths between 12 and 22. In this case, we say 22 is a trivial path for the existential node 3. If the annotations between 12 and 20 are also 1 or +, path 20 is also trivial. The execution engine does not need to check, on the actual data, which elements on path 12 actually have descendants on paths 20 and 22: we know they all do. Thus, paths 20 and 22 are discarded from the set of pattern node 3.

In general, let p_x be a path relevant for an existential node n_x; this node must have a closest non-existential ancestor n_y in the pattern. There must be a path p_y relevant for n_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. For all paths p_y as described above, all summary nodes between p_y and p_x are annotated with either 1 or +.
2. All paths descendant of p_x, and relevant for nodes below n_x in the query pattern, are trivial.
3. No value predicate is applied on n_x or its descendants.

A trivial path as p_x above is eliminated from the relevant path set of node n_x.

For the query pattern in Figure 5, the trivial and useless paths are those shaded in grey.

Pattern contraction based on path pruning After pruning out useless and trivial paths, nodes left without any relevant path are eliminated from the pattern. For the

query pattern in Figure 5b, this yields exactly the result in Figure 5c from which the grey-dotted paths, and their pattern nodes, have been erased.

Observe that the relevant path sets of a pattern node n may be diminished, yet not empty. In this case, n remains in the pattern, and its reduced path set may be exploited by the optimizer to construct a data access plan for n that reads less data than if the original path set had been kept. The ability of the optimizer to do so depends on the storage model chosen; in particular, the path-partitioned storage model does allow such optimization. This will be discussed in more detail in Section 4.

3.4 Computing relevant paths

Having defined relevant paths, the question is how to efficiently compute them. Given a summary S and a pattern p, we have defined relevant paths based on the set of documents conforming to S. The basic observation underlying summary-based optimization since the DataGuide work [24] is that relevant paths can be computed by *evaluating the tree pattern over the path summary* (patterns like GTPs [16] or XAMs [4] require simple modifications to handle optional edges).

A straightforward method is a recursive parallel traversal of S and q, checking ancestor conditions for a path to be relevant for a pattern node during the descent in the traversal [1, 24]. When a path p satisfies the ancestor conditions for a pattern node n, the summary subtree rooted in p_n is checked for descendant paths corresponding to the required children of n in the pattern. This simple method is suboptimal in terms of *running time*, since it may traverse 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 19 will be traversed once to check descendants of path 15, and once to check descendants of the path 18.

A more efficient method consists of performing a single traversal of the summary and collecting potentially relevant paths that satisfy the ancestor path constraints, but not necessarily (yet) the descendant path constraints. When the summary subtree rooted at a potentially relevant path has been fully explored, we check if the required descendant 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. This algorithm may run: (1) on the in-memory summary, which has the drawback of requiring $\Theta(|S|)$ memory space; (2) more efficiently, traversing the summary in streaming fashion, using only $O(h)$ memory (where h is the summary height) to store the state of the traversal.

A remaining problem of such a time-efficient method concerns the total size of the relevant path sets. Figure 7 illustrates this on a summary S of size s, and a pattern q having k nodes, all of which are annotated Val. All pattern node labels are unspecified. A straightforward evaluation of q over S, as envisioned in [1, 24], computes a set of k-tuples of the form $t = (p_1, p_2, \ldots, p_k)$, such that for any $1 \leq j \leq k$, p_j is a relevant path for q's node j, and for any $1 \leq j \leq (k-1)$, p_j is an ancestor of p_{j+1}. For the pattern q in Figure 7, there will be $k! \times (s-k)!/s!$ such tuples (assuming $s \geq k$, which we expect is the frequent case). Some of these tuples are shown close to q, at its right, in Figure 7. In general, ancestor-descendant edges and unspecified pattern node labels may lead to large sets of relevant path tuples.

While such computations apply on summaries which are typically much smaller than the database, some combinations of summaries and queries can still lead to large

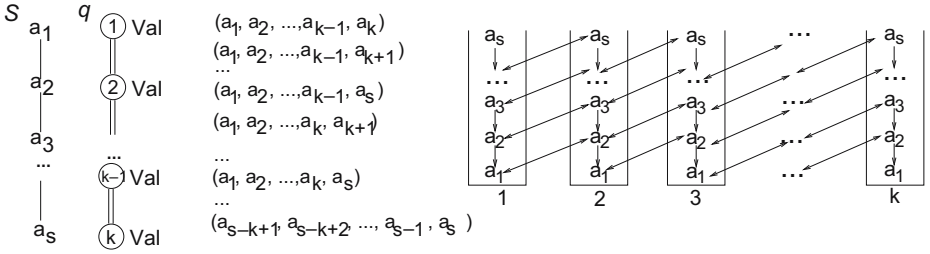


Figure 7 Sample query pattern and relevant path sets: at the center, relevant path tuples; at right, paths encoded on stacks (see Algorithm 1).

memory needs. Relevant path identification is just one among many steps needed during query optimization, and therefore it should be frugal with both running time and needed memory, especially in a multi-user, multi-document database. Therefore, a compact encoding of relevant path sets is needed.

Algorithm 1 shows how to compute relevant paths in a single streaming summary traversal, encoding the answers in a space-efficient way to avoid an explosion in answer size. Algorithm 1 runs in two phases.

Phase 1 (finding relevant paths) traverses 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 *stacks*(*n*). Potentially

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: */
foreach *pattern node* n **do**
  stacks(n) ← new stack
currentPath ← 0
Traverse the path summary in depth-first order:
foreach *node* n *visited for the first time* **do**
  Run algorithm **beginSummaryNode**
foreach *node* n *whose exploration is finished* **do**
  Run algorithm **endSummaryNode**
/* **Phase 2: minimizing relevant path sets** */
foreach *node* n *in* q **do**
  foreach *stack entry* se *in stacks(n)* **do**
    if n *is existential and*
    all1or+*(se.parent.path, se.path)* **then**
      se is trivial. Erase se and its descendants from the stack.
    if n *is a "for" var. and* n *and its desc. are not boxed and* ***allI****(se.parent.path,se.path)* **then**
      se is useless. Erase se from *stacks(n) and*
      connect se's parent to se's children, if any
  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                                   */
currentPath ++;
/* Look for pattern query nodes which t may match:                                 */
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:            */
    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
        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): */
            stack entry se ← new entry(currentPath)
            se.parent ← stacks(n').top
            if stacks(n) is not empty and stacks(n).top is open then
                se.selfParent ← stacks(n).top
            else
                se.selfParent ← null
            se.open ← true
            stacks(n).push(se)
```

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 descendants 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.

Algorithm 3: **endSummaryNode**

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

Figure 7 (right) shows the content of all stacks after relevant path sets have been computed for the pattern q in the same figure. Bidirectional arrows between stacks represent *parent* and *children* pointers; vertical arrows between nodes in the same stack represent *selfparent* pointers, which we explain shortly.

In Algorithm *beginSummaryNode*, when a summary node (say p) labeled t starts, we need to identify pattern nodes n for which p may be relevant. The final tag in p must be t or $*$ in order to match a t-labeled pattern node. Moreover, at the time when traversal enters p, there must be an open, potentially relevant path for n's parent which 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 node entries in the same stack. For instance, in the relevant path sets in Figure 7, node a_1 in the stack of pattern node 1 only points to a_2 in the stack of pattern node 2, even though it should point also to nodes $a_3, \ldots, a_s$ in the stack of pattern node 2, given that these paths are also descendants of a_1. The information that these paths are children of the a_1 entry in the stack 1 is implicitly encoded by the *selfparent* pointers of nodes further up in the stack 1: if path a_3 is a descendant of the a_2 entry in this stack, then a_3 is implicitly a descendant of the a_1 entry also.

This stack encoding via *selfparent* guarantees relevant paths are encoded in only $O(|q| \times |S|)$ space. Our experimental evaluation in Section 5 shows that this upper bound is very relaxed. The encoding is inspired from the Holistic Twig Join [13]. The differences are: (1) we use it when performing a single streaming traversal over the summary, as opposed to joining separate disk-resident ID collections; (2) 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 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 *descendants*. 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 descendant 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 pre-computed encoding is used (Section 2). With the basic encoding, Phase 2 traverses the summary again (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 *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 $|S|$. 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 has $\Theta(|q|)$ time complexity. Putting these together, we obtain $\Theta(|S| \times |q|^2)$ time complexity. The space complexity in $O(|S| \times |q|)$ for encoding the path sets. In practice, path sets are much smaller, as Section 6 shows.

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)*. A large family of existing XML storage, indexing and materialized view proposals can be described by XAM patterns [4]. This family includes, e.g., the relational tables described in [23], the different storage strategies of [47], tag indexes [29, 36], path indexes [11, 36], and the materialized views exploited in [10, 56].

- Consider an index grouping IDs by the element tags, as in [29] or the LIndex in [36]. A single-node pattern can be drawn for each tag t in the document, labeled with $[Tag = t]$ and with ID.
- A path index such as PIndex [36] or a path-partitioned store [11] provides access to data from one path at a time.
- A view expressed in core XPath as in [56] also yields a XAM pattern.

Let *storage structure* designate any among: a storage structure (e.g. a table in [23, 47]), an index (e.g. tag index [29]), or a materialized view (e.g. an XPath view [56]). Let p_s be a XAM pattern describing such a storage structure, and consider a summary S and a document D such that $S \models D$.

As explained before, a query $q \in \mathcal{Q}$ yields one or more XAM patterns p_q. The task of *access method selection* when processing q consists of choosing among all the available storage structures, the ones to use to answer q. This implies (1) finding all such suitable structures, and (2) choosing among the possible alternatives, presumably with the help of some cost model.

Consider one query pattern p_q and a p_q node n_q labeled with *Val* or *Cont* (intuitively, n_q represents some data that the query must return). A storage structure described by a pattern p_s can be used to provide some of the data required by n_q if and only if p_s has a node n_s, such that the relevant paths of n_q and the (descendants of) relevant paths of n_s have a non-empty intersection.

The relevant paths of n_s may be a superset of n_q's paths (e.g. if the query is $/a/b$, p_s is a materialized view of the form $//b$, and the summary implies b elements occur on paths $/a/b$ and $/a/c/b$). The opposite can also arise, i.e. the relevant paths of n_q are a superset of n_s's paths. To see why we compare the paths of n_q with descendants of n_s's relevant paths observe that if n_s is labeled *Cont*, one can extract from the storage structure t_s all descendants of elements matching n_s by XPath navigation.

In the most general case, enumerating the ways in which the data required by a query pattern p_q can be computed out of the storage patterns $p_s^1, p_s^2, \ldots, p_s^n$ amounts to query rewriting under summary constraints and is addressed in [35]. Section 4.1 considers a particular case where this rewriting problem is very simple, namely when a path-partitioned storage model is used, and shows it can lead to time- and memory-efficient query execution plans. Such plans are enabled by the high fragmentation degree of the path partitioned model, which understandably raises problems when complex XML subtrees must be returned. Section 4.2 discusses efficient algorithms for this purpose.

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: (1) construct access plans for every query pattern node, by merging the corresponding ID or value sequences (recall the logical storage model from Figure 3); (2) 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: (3) for every relevant path p_{ret} of an expression appearing in a "return" clause, reconstruct the subtrees rooted on path p_{ret}; (4) re-assemble the output subtrees in the new elements returned by the query. For example, Figure 8 depicts a QEP for the sample query from Figure 5. In this QEP, IDs(n) designates an access to the sequence of structural IDs on path n, while IDsAndVal(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 right outer-joins ($\ltimes\!\sqsubset$) are *structural*, i.e. they combine inputs based on parent-child or ancestor-descendant relationships between the IDs they contain. Many efficient algorithms for structural join exist [2, 13]; we are only concerned here with the logical operator.

The plan in Figure 8 is directly derived from the relevant path sets from Figure 5c and the query. The selection σ has been taken from the query, while the Merge fuses the information from the two relevant paths for $2 (emph element). The final

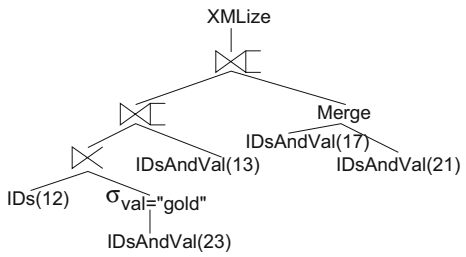


Figure 8 Complete QEP for the query in Figure 5.

XMLize operator assembles the pieces of data in a result. Section 4.2 studies this in more detail.

We do not delve into more plan construction detail, as the process is similar to the one described in [16]. The difference is that QEPs built on a path-partitioned store benefit from relevant paths to access only a very small subset of the data. For instance, with path partitioning only asian item IDs are read, whereas in [16] (which uses a tag index) all item IDs need to be read. In the query planning approach of [43], which builds on [16], data structures manipulated during query processing include (pointers to) subtrees in a persistent tree store. The path partitioned model does not include a persistent tree, and the corresponding QEPs only manipulate IDs until the last stage in the plan (which reconstructs full trees). The advantage of manipulating IDs only is to avoid scanning from the disk useless sub-elements. For instance, our approach does not need to read item description elements, which are quite large. The disadvantage of not having a persistent tree lies in the difficulty to assemble complex XML elements in the output (addressed in Section 4.2).

Thus, the path partitioned storage model, coupled with a summary and with relevant path computation on query patterns, leads to efficient query plans. These plans access less data than if the common tag-based index is used [16, 22, 29, 43], manipulate small intermediary results (mostly IDs), and (like the plans built in [16, 43]) can scale up well in the input data size, due to the usage of efficient structural join operators [2]. More specialized indexes or views can clearly be added on top of a path-partitioned store; we simply aim here at highlighting the opportunities of the basic model.

Physical optimizations We end this section by showing how physical optimization techniques previously presented [16, 42] are directly supported by relevant path sets.

Path expressions used in XPath and XQuery need to return duplicate-free lists of nodes. Let op_1, op_2 be two operators, such that $op_1.X$ and $op_2.Y$ contain structural IDs. 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 descendants 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 $op_1 \rtimes \lhd op_2$ will contain y_0 twice. If the X column was not needed after the join, a duplicate elimination operator is required on the Y column. Moreover, the join result order depends on the physical algorithm employed [2], and may or may not fit the order requirements for the other query operators (e.g. other structural joins). This may bring the need for explicit Sort operators subsequently. Consider the particular case when $op_1.X$ contains IDs of elements of a single tag a and $op_2.Y$ contains IDs of elements of a single tag b. In this case, it has been shown [16] that if the schema implies that a elements cannot have a descendants, no duplicate elimination or sort is needed, as the join output is duplicate-free and follows both $op_1.X$ and $op_2.Y$ orders.

Relevant path sets allow generalizing this observation. Consider, as before, a structural join of the form $op_1 \bowtie_{X\, anc\, Y} op_2$, where the join checks whether IDs from $op_1.X$ correspond to ancestors of elements in $op_2.Y$. If, for any two possible paths p_1, p_2 of element whose IDs are in $op_1.X$, p_1 is not an ancestor p_2, then the output does not contain duplicate values in the X or Y columns, and its order respects both the X and Y input orders. In particular, if op_1 is IDs(p) for some path p, this condition is satisfied, unlike the case when all element IDs are stored together [49, 57]

or partitioned by the tags [29]. This order descriptor of the storage can be given as input to an optimization algorithm handling complex orders as described in [42].

Finally, group-by and duplicate elimination avoidance techniques have been proposed based on schema-derived cardinality constraints of the form "every *a* element has at most one *b* descendant" [16]. Relevant path sets enable the same class of optimization at the finer granularity of paths.

Observe that relevant paths may enable some more optimizations than a schema does, in the case where the schemas are loosely specified. For instance, the DBLP DTD (http://www.informatik.uni-trier.de/~ley/) does not state that a phdthesis has exactly one author child (which for obvious reasons is always the case), whereas the DBLP summary does capture this. Moreover, associating relevant paths to patterns allows such optimizations in the general case, even if the pattern is loosely specified. For instance, given the query /dblp/*[school="U.Florida"]/author, relevant path computation shows that the * node can only be labeled masterthesis or phdthesis (since these are the only DBLP publications having school children elements), and that such elements have only one author child.

4.2 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 [47] method for exporting relational data in XML to a path-partitioned setting with structural IDs. The plan in Figure 8 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 9a cannot be used for this query, because it introduces multi-valued dependencies [21]: it multiplies all emph elements by all their keyword cousins, while the query asks for the keyword and emph descendants of a given item to be concatenated (not joined among themselves). The plan in Figure 9b 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 9c, 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 intermediate materialization steps and the size of the materialized results. Elements on path x must be materialized if they must be combined with multiple children, and some child path y of x is not annotated with 1. Some IDs are materialized multiple times, after joins

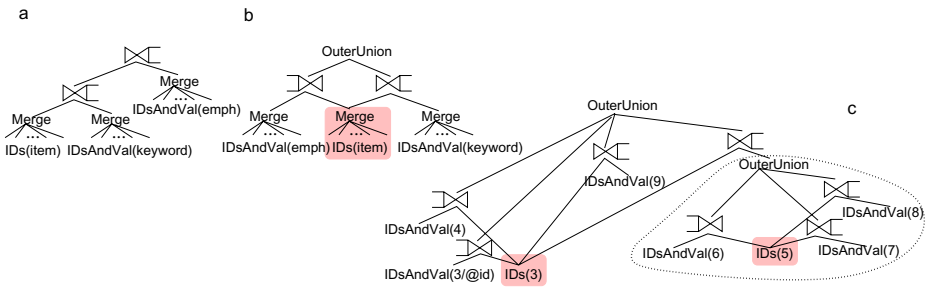


Figure 9 Sample outer-union QEPs with structural joins.

with descendant IDs at increasing nesting level. For instance, in Figure 9, 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 $O(N \times h/B)$ I/O complexity, where B is the blocking factor and h the document height. If in-memory materialization is used, the memory consumption is in $O(N \times h)$. The time complexity is also $O(N \times h)$.

To reduce the space requirements, we devise a physical operator specialized for the path-partitioned store, named Reconstruct. It reads in parallel the ordered sequences of structural IDs and (ID, value) pairs from all the paths to recombine, and produces 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 the next structural ID in the stream IDs(n) is the smallest among all structural ID streams corresponding to children of person elements.

Figure 10a outlines a Reconstruct-based plan, and Figure 10b 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 intermediate results*. Thus, it has a smaller memory footprint than the SortedOuterUnion approach. Contrast the QEPs in Figures 9b and 10b: the former needs to build address elements separately, 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.

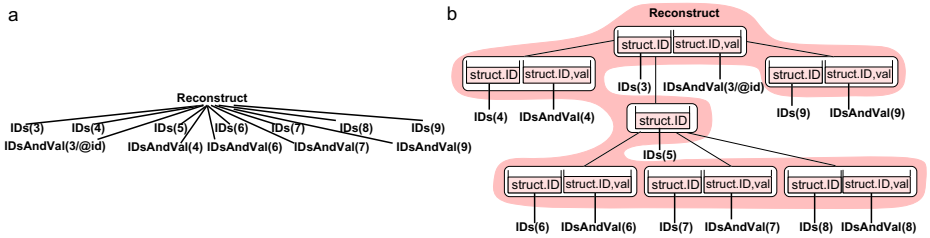


Figure 10 Reconstruct plans for //person on XMark data.

The Reconstruct has $O(N)$ time complexity. It needs one buffer page to read from every path which contributes 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 and the path-partitioned storage model within the XQueC system [7, 8]. Subsequently, we isolated XQueC's path summary construction and manipulation algorithms in a standalone Java library called XSum (available for download [55]) in order to use it in other projects. This section describes our experience exploiting summaries, alone or in conjunction with a path-partitioned store.

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 [7], 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 Summary size

In this section, we study summary sizes for a variety of XML documents: some are obtained from [54], to which we add a set of XMark$n$ documents produced by the XMark data generator [51] to the size of n MB, and two DBLP snapshots from 2002 and 2005 (http://www.informatik.uni-trier.de/~ley/). Table 1 shows the document sizes, number of document nodes N, number of summary nodes $|S|$, and the ratio $|S|/N$.

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. 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 out as more data is added, even for complex documents such as XMark, with 12 levels of nesting, recursion etc. 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

Table 1 Sample XML documents and their path summaries.

Doc.	UW Course	Shakespeare	Nasa	Treebank	SwissProt
Size	3 MB	7.5 MB	24 MB	82 MB	109 MB
N	84,051	179,690	476,645	2,437,665	2,977,030
$\|S\|$	18	58	24	338,738	117
$\|S\|/N$	2.1×10^{-4}	3.2×10^{-4}	5.0×10^{-5}	1.3×10^{-1}	3.9×10^{-5}
Doc.	XMark11	XMark111	XMark233	DBLP (2002)	DBLP (2005)
Size	11 MB	111 MB	233 Mb	133 MB	280 MB
N	206,130	1,666,310	4,103,208	3,736,406	7,123,198
$\|S\|$	536	548	548	145	159
$\|S\|/N$	2.4×10^{-3}	3×10^{-4}	1.3×10^{-4}	3.8×10^{-5}	2.2×10^{-5}

natural language text, 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.

We now consider the sizes attained by serialized stored summaries. Two choices must be made: (1) XML or binary serialization, and (2) 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. Pre-computed 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 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 pre-computed summaries and found it always within a factor of 1.5 of the direct binary one, which is quite compact.

A path summary is built during a single traversal of the document, using $O(|S|)$ memory [1, 24]. We gather `1` and `+` labels during summary construction and traverse the summary again if the pre-computed encoding is used, making for $\Theta(N + |S|)$ time and $\Theta(|S|)$ memory. As a simple indication, the time to build and serialize the summary of the 280 MB DBLP document (in XML format) is 45 s.

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

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

Table 3 Computing relevant paths for the XMark queries.

Query and time										
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

5.2 Relevant path computation

We now study the performance of the relevant path set computation algorithm from Section 3.4. Computing the relevant paths for a tree pattern is part of the query optimization stage and it only involves the summary (no access to the persistent repository takes place).

The setting for these measures is the following. Summaries for the XMark111 and Treebank document have been previously computed and serialized in binary format, using the pre-computed encoding. We use the XMark111 summary as representative of the moderate-sized ones and Treebank as the largest (see Table 2). For the measure, the file containing the serialized summary is opened and traversed by an event-based parser we wrote. The parser computes the relevant paths as described in the algorithm from Section 3.4. We measure the time taken by this traversal and computation.

Table 3 shows the relevant path computation time on patterns resulting from the 20 queries of the XMark benchmark [51], on the XMark111 summary. 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. Table 4 shows this time for the XMark queries 1 and 9, for which Table 3 has shown path computation is fastest, resp. slowest. Path computation on an XML-ized summary is about four to five times slower than on the binary format, reflecting the impact of the time to read the summary itself. The running time on a pre-computed 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 one only from now on. If the optimizer caches query plans, however, the binary direct encoding may be preferable.

Table 4 Impact of summary serialization format on relevant path computation time.

Query no.	XML dir. (ms)	XML pre-cp. (ms)	bin. dir. (ms)	bin. pre-cp (ms)
1	73.0	37.0	22.3	14.2
9	255.7	133.6	98.6	46.4

Table 5 XPath renditions of query patterns on TreeBank data.

Renditions of query patterns
TK*n*: //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

We now consider the TreeBank summary (in binary pre-computed encoding) and a set of query patterns, shown in Table 5 as XPath queries for simplicity (however, 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. TK*n* denotes a parameterized family of queries taken from [14], where the steps /NP/PP are repeated *n* times. Figure 11 (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,[2] 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 the number of resulting paths at the right in Figure 11: T1 yields many more paths (about 50,000) than T2 (about 10,000) or T3 (about 1,000). Each relevant path is a stack entry 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 descendants. 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 TK*n* queries *decreases as n* increases, a tendency correlated with the number of resulting paths, at the right in Figure 11. 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 descendants (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 value of *selfparent* pointers in cases where there are many relevant paths, due to a large summary and/or * query nodes.

5.3 Tree pattern evaluation based on a path-partitioned store

We measured the time needed to evaluate some tree patterns (that is, find the structural ID tuples corresponding to their matching elements) on a path partitioned

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

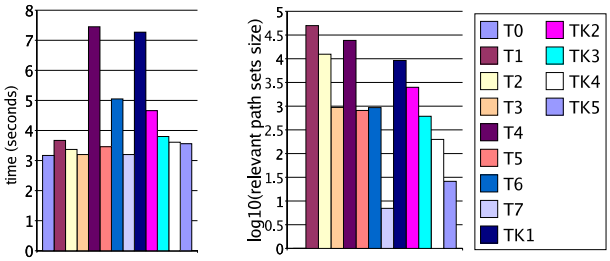
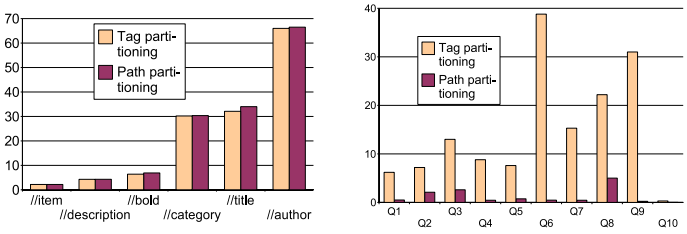


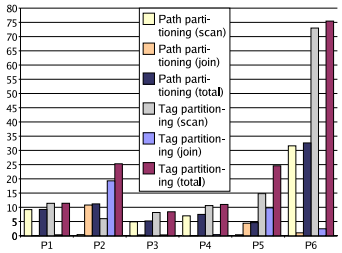
Figure 11 Relevant path computation times on TreeBank (*left*) and resulting relevant path set size (*right*, log scale).

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 in XQueC a similar store, but where IDs are partitioned *by their tags*, not by their paths, as in [26, 29]. On both stores, the StackTreeDesc [2] structural algorithm was used to combine structural IDs.

We start by considering the simplest case when looking for the IDs of all nodes of a given tag. This experiment quantifies the overhead of path partitioning in the case where the access is not performed by the path. At the top left in Figure 12, we show the execution times for finding the IDs of the elements corresponding to the queries //item, //description, //bold, //category on the 111 MB XMark document, as well as the queries //title and //author on the 128 MB DBLP document. We have chosen these queries because these tags occur on up to a hundred different paths in the respective



Q1	//europe//item/descr	Q6	//Entry//METAL//Descr
Q2	//regions//item//descr	Q7	//categ//listitem//text
Q3	//europe//parlist//bold	Q8	//parlist//listitem//text
Q4	//europe//parlist//listitem	Q9	//dblp//book//author
Q5	//item//descr//keyword	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 12 Tree pattern evaluation with path and tag partitioning.

documents. The execution times at the top left of Figure 12 show that the overhead of path partitioning, that is, the effort to merge the ID sequences corresponding to all paths, is quite small.

In general, the effort to produce one output tuple when merging k sequences is of the order of $\lceil \log_2(k) \rceil$. This effort may remain moderate even for large k values. Thus, we believe path partitioning does not incur a strong overhead over tag partitioning, even in the case when all elements of the same tag are required together. However, excessive fragmentation incurred by path partitioning may raise problems of a different nature. When loading the TreeBank document in our path-partitioned store, the number of storage structures filled in by loading (equal to the summary size, see Table 1) outgrew the default number of files one can simultaneously open (1,024 on our system). Since we implemented each storage structure as a BerkeleyDB database hosted in a separated file, this posed problems. We implemented an open file counter that closed and re-opened files to stay within reasonable limits, but the more general lesson is that very large numbers of paths may require some careful engineering of the store.

The other graphs in Figure 12 show the execution times for 10 XPath queries (Q6 on SwissProt, Q9 and Q10 on DBLP, the others on a 111 MB XMark document), and 6 tree patterns (P1 to P6 on the 111 MB XMark). In Figure 12, path partitioning achieves important performance improvements (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, confirming the advantage of path-based indexing over tag-based indexing.

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 12. Turning minimization off increased the running time by 15% to 45%.

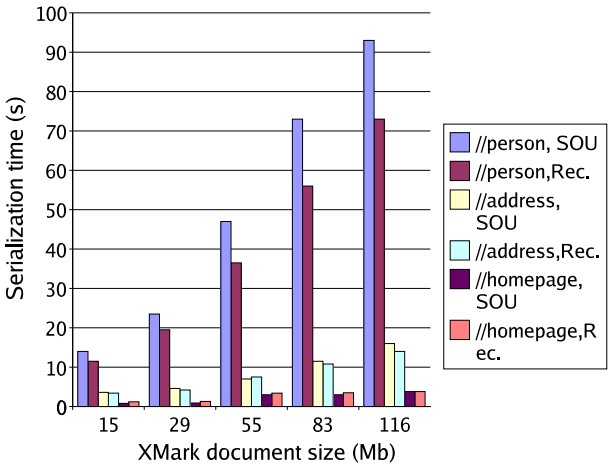


Figure 13 SortedOuterUnion and Reconstruct performance.

Table 6 Sample summary-based query unfolding.

Query	
document("xmark.xml")//person	for $x1 in document("xmark.xml")/site return <site> { for $x2 in $x1/people return <people> for $x3 in $x2/person return $x3 </people> } </site>

5.4 Reconstructing path-partitioned data

We tested the performance of the two document reconstruction methods described in Section 4.2 on our path-partitioned store. Figure 13 shows the time to build the full serialized result of //person, //address and //homepage on XMark documents of increasing sizes. The sorted outer union (denoted SOU in Figure 13) materialized intermediate results in memory. On the XMark111 document, //person outputs about 15 MB of result. As predicted in Section 4.2, both methods scale up linearly. The Reconstruct is noticeably faster when building complex elements such as address and person. Furthermore, as explained in Section 4.2, 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

The idea of summarizing semi-structured data can be traced back to the work on representative objects [41]. Representative objects are meant as a help in formulating queries, and they allow detecting empty-result queries without accessing the repository. Path summaries have been used as a basis for optimization in many works [12, 39]. In [12], the authors propose a tree pattern pruning operator whose task is similar to computing relevant paths. However, the authors only consider XPath (conjunctive) patterns and do not consider summary-based pattern minimization. Optimizations proposed in [12], concerning ordering the structural joins needed to process a conjunctive tree pattern, can easily be adapted to the path-partitioned storage model.

Two kinds of optimizations were considered in [24, 39]. (1) Summaries were used to make queries more specific by replacing node label or path wildcards by precise paths. This is an important optimization in that context since path query evaluation was based on navigation in the data graph, with quite poor access locality and thus poor performance. Precise navigation meant fewer data blocks to visit. In our context, the presence of structural identifiers allows evaluating path queries without navigating. Some recent works have shown that navigation and structural joins combined achieve the best performance [26]. While the basic path-partitioned model does not provide a persistent tree to navigate in, the two are not fundamentally incompatible. (2) Path indices were used to evaluate parts of the query's navigation. Our data access method selection approach (Section 4) follows directly this idea, with the difference that we combine such data access plans by more scalable structural joins, unavailable at the time of [39]. More generally, we have pointed at data access method selection based on relevant paths, when the storage structures are described by arbitrary patterns.

Storage models organized based on element paths have been used in [11, 28, 57]. These works exploited more or less explicitly a path summary as a key to the storage structures. In [28, 57], paths are stored as string attributes in a relational table, and relevant path computation is performed by string pattern matching. While this is appropriate for linear path queries, more processing is required for tree pattern queries, and for complex patterns involving optional nodes. The work described in [12], discussed above, is the most recent follow-up on [11] regarding summary usage.

The work on Dataguides [24] advocates using summaries as a basis for path indexing. Index structures more complex than path indexes have been proposed e.g. in [30]. A complete F&B index [30] is more complex to build and to maintain than a path index. Therefore the authors provided alternative schemes building only partial indices [30, 31, 46]. In contrast, for us the path-partitioned set of IDs is the storage itself, thus it has to be complete. Complex F&B indices can be added to the path-partitioned store. If they are restricted to forward navigation and do not use ID-IDREF connections, they may be described by XAMs and exploited by access method selection as outlined in Section 4. If backward navigation is used, relevant path computation becomes more complex, but is still possible.

Path indexing schemes such as APEX [18], the $D(k)$ index [46], and multiresolution indexing [27] only materialize a path in the index if the path appears in the workload. In particular, multiresolution indexing allows different degrees of detail to co-exist in the index. This allows the index to closely track the workload query needs.

Besides indexing, summaries have also been used as a support for cardinality statistics about an XML tree [20, 45]. The statistic-annotated summary of an XML tree is built to fit the space budget that the user is willing to pay for the precision of its estimations. This support for statistics is significantly more complex to build than a simple path summary. However, a simple summary (as considered in this work), when used as a support for path cardinalities, is likely to yield poor-quality cardinality estimations, thus the need for more elaborate schemes [45].

An interesting class of compressed structures is described in [14] and is used as a basis for query processing. This approach compresses the XML structure tree into a compact DAG, associating to each DAG node the set of corresponding XML

element nodes. We have performed measures (omitted for brevity) showing that the path summary is generally smaller, in some cases by two orders of magnitude, than this DAG. This is explained by the fact that for two XML nodes to correspond to a single summary node, a path summary only requires that their incoming paths be the same. In contrast, the DAG summary introduced in [14] also requires that the tree structure found below the two nodes be similar. The difference increases in the presence of recursive, variable and repeated structure. We thus argue that path summaries are generally much more robust and therefore of practical interest.

In [15], the authors propose a specialized query processing framework based on the summaries described in [14]. The authors present an approach for handling DAG-compressed structures throughout the query processing steps, which reduces the risk that the unfolded compressed structure would outgrow the available memory. In contrast, we make the point that path summaries can be added with minimal effort into existing XQuery processing systems, and that they marry well with efficient techniques such as structural identifiers and structural joins.

Path information has been used recently for XPath materialized view-based rewriting [10] and for access method selection [5, 12]. The summary-based rewriting algorithm based on materialized views we described in [35] generalizes these approaches. In the present work, we aimed at casting summary-enabled optimization techniques, including, but not limited to access method selection, in a single framework, and highlight the advantages of the simple path-partitioned storage model for XML query processing.

The only previous relevant path computation algorithm we could find concerns simple linear path queries only [1, 24]. It works in memory in a top-down manner and does not perform any path minimization. The algorithm we have described bears similarities with existing stack-based tree pattern matching algorithms [13]. Its time complexity is not surprising, since it follows the results of [25]. Its advantage is to require very little memory, which is a desirable feature given that relevant paths have to be computed during query optimization, with a small memory budget.

Many works target specifically query minimization, sometimes based on constraints, e.g. [3, 16, 19, 32]. We have outlined the differences between our work and existing works on minimization under constraints in Section 3.3. Constraints can be obtained from an (a priori) XML Schema or from a summary extracted from the data. A first big advantage of summary-based minimization (and optimization in general) is that it can apply even when schemas are unavailable (a frequent case [37]) and can use information that even an available schema doesn't provide, as our example at the end of Section 3.3 shows. Constraint-independent minimization techniques such as [19] are orthogonal to our work and can be combined.

With respect to path partitioning, we considered the task of retrieving IDs satisfying given path constraints as in [10, 12, 39] and show that structural IDs and joins efficiently combine with path information. Differently from [10, 12, 39] which assume available a persistent tree structure, we also considered the difficult task of re-building 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.

Complex, richer XML summaries have also been used for data statistics;they tend to grow large, thus only limited-size subsets are kept [44]. Since path indices represent the store itself in our context, we must keep it complete.

The starting point of this work is the XQueC compressed XML prototype [7, 8]. 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 [34]. A 2-pages poster based on this work has been recently published [9].

7 Conclusion and perspectives

We have described a practical approach for building and exploiting path summaries as metadata in a persistent XML repository, i.e., information about the structure encountered in the XML document. We have shown how summaries can be combined with path partitioning to achieve efficient, selective data access, a plus for processing queries with complex navigation requirements.

Our own experience developing the summary was first included in our XQueC [7, 8] XML compression project. Subsequently, we isolated it out of the XQueC prototype and found it useful in some applications which we briefly describe below. Our summary library is freely available [55].

Apprehending varied-structure data sources In the framework of the INEX[3] collaborative effort, we concentrated on designing an integrated conceptual model out of heterogeneously-structured bibliographic data sources. As a side effect of building summaries, XSum also generates image files of such summaries [55]. We used this feature to get acquainted with the sources and visualize their structure. This is in keeping with the initial Dataguide philosophy of using summaries for exploring data sets [24].

Physical data independence We developed a materialized view management tool for XQuery, called ULoad [5]. This tool includes a query rewriting module based on views, which naturally leads to containment and equivalence problems. ULoad judges containment and equivalence *under summary constraints*, thus exploiting summaries and path annotations.

Query unfolding An ongoing work in the Gemo group requires a specific form of query unfolding. As soon as an XQuery returns some elements found by some unspecified navigation path in the input document (that is, using the descendant axis), the query must be rewritten so that it returns *all elements on the path from the document root to the returned node*, not just the returned node as regular XPath semantics requires. For instance, the query //person in an XMark document must be transformed into the query in Table 6. This work is still ongoing.

Perspectives Our ongoing work focuses on adding to the XSum library a version of the containment and equivalence algorithms implemented in ULoad. We are also considering the joint usage of summary and schema information for XML

[3]INEX stands for Initiative for the Evaluation of XML Information Retrieval; see http://inex.is.informatik.uni-duisburg.de.

tree pattern query rewriting and containment; we anticipate that this combined usage provides increased information and thus more opportunities for optimization.

We are also currently extending ULoad to support XQuery updates; accordingly, we expect to implement summary maintenance under data modifications in XSum. It is to be noted that summary maintenance has very low complexity, using our notion of summary [24], thus we do not expect this to raise difficult issues.

Acknowledgements The authors are grateful to Christoph Koch for providing us with his XML compressor code [14], and to Pierre Senellart for sharing with us his query unfolding application.

References

1. Aboulnaga, A., Alamendeen, A.R., Naughton, J.F.: Estimating the selectivity of XML path expressions for internet scale applications. In: VLDB (2001)
2. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE (2002)
3. Amer-Yahia, S., Cho, S., Lakshmanan, L.: Minimization of tree pattern queries. In: SIGMOD (2001)
4. Arion, A., Benzaken, V., Manolescu, I.: XML Access Modules: Towards Physical Data Independence in XML Databases. XIME-P Workshop (2005)
5. Arion, A., Benzaken, V., Manolescu, I., Vijay, R.: ULoad: choosing the right storage for your XML application. In: VLDB (2005)
6. Arion, A., Benzaken, V., Manolescu, I., Vijay, R.: Algebra-based tree pattern extraction in XQuery. In: FQAS Conference (2006)
7. Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: XQueC: Pushing queries to compressed XML data (demo). In: VLDB (2003)
8. Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: Efficient query evaluation over compressed XML data. In: EDBT (2004)
9. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern XML databases (poster). In: WWW (2006)
10. Balmin, A., Ozcan, F., Beyer, K., Cochrane, R., Pirahesh, H.: A framework for using materialized XPath views in XML query processing. In: VLDB (2004)
11. Barbosa, D., Barta, A., Mendelzon, A., Mihaila, G.: The Toronto XML engine. In: WIIW Workhshop (2001)
12. Barta, A., Consens, M., Mendelzon, A.: Benefits of path summaries in an XML query optimizer supporting multiple access methods. In: VLDB (2005)
13. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD (2002)
14. Buneman, P., Grohe, M., Koch, C.: Path queries on compressed XML. In: VLDB (2003)
15. Buneman P., Choi B., Fan, W., Hutchison, R., Mann, R., Viglas, S.: Vectorizing and querying large XML repositories. In: ICDE, pp. 261–272 (2005)
16. Chen, Z., Jagadish, H.V., Lakshmanan, L., Paparizos, S.: From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In: VLDB (2003)
17. Chien, S., Vagena, Z., Zhang, D., Tsotras, V.: Efficient structural joins on indexed XML documents. In: VLDB (2002)
18. Chung, C.W., Min, J.K., Shim, K.: APEX: an adaptive path index for XML data. In: SIGMOD (2002)
19. Deutsch, A., Papakonstantinou, Y., Xu, Y.: The NEXT logical framework for XQuery. In: VLDB, pp. 168–179 (2004)
20. Drukh, N., Polyzotis, N., Garofalakis, M.N., Matias, Y.: Fractional XSKETCH synopses for XML databases. In: XSym (2004)
21. Fagin, R.: Multivalued dependencies and a new normal form for relational databases. ACM Trans. Database Syst. **2**(3), 262–278 (1977)
22. Fiebig, T., Helmer, S., Kanne, C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system. VLDB J. **11**(4), 292–314 (2002)

23. Florescu, D., Kossmann, D.: Storing and querying XML data using an RDMBS. In: IEEE D. Eng. Bull (1999)
24. Goldman, R., Widom, J.: Dataguides: enabling query formulation and optimization in semistructured databases. In: VLDB. Athens, Greece (1997)
25. Gottlob, G., Koch, C., Pichler, R.: The complexity of XPath query evaluation. In: PODS (2003)
26. Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A.N., Tian, F., Viglas, S., Wang, Y., Naughton, J.F., DeWitt, D.J.: Mixed mode XML query processing. In: VLDB (2003)
27. He, H., Yang, J.: Multiresolution Indexing of XML for Frequent Queries. In: ICDE (2004)
28. Jiang, H., Lu, H., Wang, W., Yu, J.: Path materialization revisited: an efficient XML storage model. In: AICE (2001)
29. Jagadish, H.V., Al-Khalifa, S., Chapman, A., Lakshmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: Timber: a native XML database. VLDB J. **11**(4), (2002)
30. Kaushik, R., Bohannon, P., Naughton, J., Korth, H.: Covering indexes for branching path queries. In: SIGMOD (2002)
31. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: ICDE (2002)
32. Lakshmanan, L., Ramesh, G., Wang, H., Zhao, Z.: On testing satisfiability of tree pattern queries. In: VLDB (2004)
33. Lee, M., Li, H., Hsu, W., Ooi, B.: A statistical approach for XML query size estimation. In: DataX workshop (2004)
34. Manolescu, I., Arion, A., Bonifati, A., Pugliese, A.: Un modèle de stockage xml basé sur les séquences. Ing. Syst. Inf. **2**(10), 9–37 (2005)
35. Manolescu, I., Benzaken, V., Arion, A., Papakonstantinou, Y.: Structured materialized views for XML queries. INRIA Tech. Report No. 1233, Available at http://hal.inria.fr.
36. McHugh, J., Widom, J., Abiteboul, S., Luo, Q., Rajaraman, A.: Indexing semistructured data. Technical Report (1998)
37. Mignet, L., Barbosa, D., Veltri, P.: The XML web: a first study. In: WWW Conference (2003)
38. Miklau, G., Suciu, D.: Containment and equivalence for an xpath fragment. In: PODS, pp. 65–76 (2002)
39. Milo, T., Suciu, D.: Index structures for path expressions. In: ICDT (1999)
40. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: SIGMOD (2004)
41. Nestorov, S., Ullman, J.D., Wiener, J.L., Chawathe, S.S.: Representative objects: concise represenations of semistructured, hierarchical data. In: ICDE (1997)
42. Paparizos, S., Jagadish, H.V.: Pattern tree algebras: sets or sequences? In: VLDB (2005)
43. Paparizos, S., Wu, Y., Lakshmanan, L., Jagadish, H.: Tree logical classes for the efficient evaluation of XQuery. In: SIGMOD (2004)
44. Polyzotis, N., Garofalakis, M.N.: Statistical synopses for graph-structured XML databases. In: SIGMOD (2002)
45. Polyzotis, N., Garofalakis, M.N.: Structure and value synopses for xml data graphs. In: VLDB (2002)
46. Qun, C., Lim, A., Ong, K.W.: D(k)-Index: an adaptive structural summary for graph-structured data. In: SIGMOD (2003)
47. Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C., Funderburk, J.: Querying XML views of relational data. In: VLDB (2001)
48. Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered XML using a relational database system. In: SIGMOD (2002)
49. Teubner, J., Grust, T.,van Keulen, M.: Bridging the GAP between relational and native XML storage with staircase join. In: VLDB (2003)
50. W3: The extensible markup language (XML). www.w3.org/TR/XML (2006)
51. Schmidt, A.: The XMark benchmark. www.xml-benchmark.org (2002)
52. Marchiori, M.: The XQuery 1.0 language. www.w3.org/XML/Query (2000)
53. Ullman, J.: Principles of database and knowledge-base systems. Computer Science Press (1989)
54. University of Washington's XML repository www.cs.washington.edu/research/xmldatasets (2004)
55. XSum: www-rocq.inria.fr/gemo/XSum (2005)
56. Xu, W., Ozsoyoglu, M.: Rewriting XPath queries using materialized views. In: VLDB (2005)
57. Yoshikawa, M., Amagasa, T., Uemura, T., Shimura, S.: XRel: a path-based approach to storage and retrieval of XML documents using RDBMSs. In: ACM TOIT (2001)