ABSTRACT
XML DBMSs require new indexing techniques to efficiently process structural search and full-text search as integrated in XQuery. Much research has been done for indexing XML documents. In this paper, we first survey some of them and suggest a classification scheme. It appears that most techniques are indexing on paths in XML documents and maintain a separated index on values. In some cases, the two indexes are merged and/or tags are encoded. We propose a new method that indexes XML values on ordered trees, i.e., two documents are in the same equivalence class is they have the same tree structure, with identical elements in order and number. The method can be seen as an extension of relational DBMS systems, where schemas are separated from data, and data are manipulated with their associated metadata. We use a simple benchmark to compare our method with two well-known European products. The results show that indexing on full trees leads to smaller index size and achieves 1 to 10 times better query performance in comparison with classical industrial methods that are path-based.

Keywords
Indexing, XML, DBMS, XQuery

1. INTRODUCTION
Since XML has been proposed as a standard exchange format by W3C, many database systems have been developed to store and retrieve XML documents. Most systems are supporting XPath queries and move towards supporting full XQuery, including XPath, FLWR expressions, and full-text search queries. Traditionally, database research distinguishes structure query from content query. Structure queries are dealing with hierarchical traversals of XML trees, i.e., processing the path expression in an XPath query. Content queries are searching for combinations of keywords and values in XML elements, i.e., processing the predicates in an XPath query. In modern XML servers, content search integrated with structure search must be efficiently performed.

The support of structure and content queries require efficient indexing of XML documents. Much research has been carried out to propose and evaluate efficient indexing techniques. Traditionally, document-indexing systems were based on inverted lists giving for each significant keyword the document identifiers with the relative offsets of the keyword instances. With XML most queries are searching for elements rather than for full documents; thus, inverted lists are referencing element identifiers rather than simple document identifiers. However, numerous element identifiers may lead to large index difficult to manage.

Structure query requires an efficient method to determine what nodes are parent or child of a given set of nodes. This can be done by performing structural joins on the edge (node, label, node) relation encoding the document structure [12], by traversing the document composition graph that is maintained when document are inserted [13], [24], [7], or by using clever numbering schemes for node identifiers [10],[21],[22]. Another problem with structure queries is that regular expressions are used extensively. Thus, label paths may be partial, including simple (*) or recursive (/) wildcards. This may lead to recursive structural joins or navigation in entire XML data graphs. Notice that structural joins and numbering schemes may also be used to check ancestor and descendant relationships, thus helping in solving regular expressions.

To overcome these difficulties, structural summaries of XML document collections have been proposed. Dataguides extended with node references were first introduced in Lore [23]. The structural index is in general quite large as every database node is referenced within the index. Several path index structures as T-indexes [24], APEX [6], A(k) indexes [17], D(k) indexes [4] tried to reduce the index size by indexing only useful paths. Such paths can be determined by specific patterns (T-indexes) or by selecting the most frequently used paths in queries (APEX) or by restricting the path length to k [17] in a dynamic way based on the query load [4]. The Fabric index [7] avoids referencing all nodes by encoding paths of terminal nodes (leaves) and storing them in a balanced Patricia trie. However, this approach requires extensions to maintain path order and to perform partial matching of tags.

Numbering schemes have been implemented for example in Xyleme, a successful native XML DBMS [1]. Two schemes are possible: pre and post-order numbering of nodes (a node identifier is a triple pre-traversal order, post-traversal-order, level) and hierarchical addressing of nodes (a node identifier is the compaction of its rank at each level). In both cases, determining if a node is an ancestor or a parent of another is simply done by comparing the identifiers. Xyleme manages a value and structure mixed index giving the node identifier of each label and of each keyword. Xyleme manages a value and structure mixed index giving the node identifier of each label and of each keyword. Xyleme maintains a generalized DTD, i.e., the set of paths with cardinalities. Solving an XPath expression is mostly done through index accesses, i.e., determining for each label and keywords in the XPath the relevant nodes identifiers and
intersecting them. The method is quite efficient, but as with the others, the index references all database nodes. It is managed in main memory, which makes that some applications require large main memory.

A more sophisticated numbering scheme based on positional representation of XML elements has been extensively worked out to answer efficiently queries in which elements are related by a tree structure, namely twig queries. Node identifiers are of the form (DocId, LftPos, RghtPos, LevelNum). Twig queries are decomposed in elementary path expressions that are more or less independently solved by navigations or index traversals. Multiple efficient join algorithms have been proposed to join the intermediate results in an efficient way [3] [33] [2]. Others propose to encode XML data trees and twig queries into sequences to transform the problem in sub-sequence matching [28] [26] [15].

In this paper, we first review the main methods proposed to index XML data. Most of them maintain a path-based structural summary with references to database nodes. In general, two nodes are in the same equivalence class if they have the same incoming path set. Thus, most XML indexing methods can be categorized as path-based indexing approach. We propose a new method called SIOUX founded on a tree-based indexing approach. The method maintains enriched structural summaries without full references to database nodes. The structural summary of a document is called a treeguide. It is an ordered tree; thus, it keeps element order and multiplicity of elements. Two documents are in the same equivalence class if they have the same treeguide, with identical elements in order and number. The treeguide index only keeps document references, not node references. While a path-based index require identifiers at the node level, an ordered tree-based index is rich enough to make possible computing the node identifiers for qualifying path expressions. Furthermore, for twig queries searching for (ordered) tree structure, path-based index leads to many false drops, which is not the case with our method.

The method is centered on a structural index comprising a repository of schemas memorized as XML tree structures (XML documents without leave contents). The repository delivers a schema identifier for a given treeguide. The repository of treeguides is at the heart of the structure index, which keeps for each treeguide the list of document identifiers having this exact schema. The fundamental idea of the method is to index documents by sequences of paths rather than by paths. It results from the observation that path order is generally meaningful (for example, most articles starts with a title and finishes with a conclusion, not the reverse). Moreover, keeping precise structural guides avoid memorizing node identifiers as they can be computed from the treeguide for a query path expression. The method is very effective with documents of irregular schemas. Notice that the idea of using complex structures (i.e., trees or graphs) as index entries has also been exploited for indexing graphs with frequent sub-graphs [32], but it is the first time full graph-indexing is applied in an XML Repository and demonstrated effective in such a context.

In addition, the method integrates a value index, which gives for each keyword occurring in a terminal node the node identifier and the relative position in node content. Several forms of node identifiers can be used, as hierarchical addresses or preorder numbers. A well-known technique in information retrieval [11], signature file could replace the value index. The advantage of a signature file is to be more compact than a value index, but the disadvantage is to be less precise and retrieve noise that has to be eliminated by value checking before returning the qualifying elements. We also propose to extend the document structure with a content signature as an additional tag to make treeguides a more differencing structure in case of too many documents of similar structure.

The method is effective not only for searching (ordered) trees and to solve efficiently twig queries, but also for solving XPath expressions. To resolve an XPath expression of the form /Book/Author[contains(Address, "Paris")], the system: (i) search for all trees containing the path /Book/Author/Address, then determines their possible documents and node identifiers through the structural index; (ii) search for all nodes containing "Paris" through the value index; (iii) do the intersection of the two lists of node identifiers to finally return the list of qualifying nodes. It is efficient in general and can be extended with OR, THEN and NOT logical connectors. In addition, if certain lists become too large before intersecting them, they can be ignored and replaced by checking all the documents of the other list. This is the case when all documents have exactly the same tree structure for the structure part of the query, or when all elements contain the searched keywords for the value part of the query. A hybrid benchmark with documents of varying schemas run on three systems demonstrates both the compactness of the index structure and the efficiency of our tree-based indexing scheme.

The remaining of this paper is organized as follows. Section 2 reviews some of the research proposals for indexing XML collections. We give a summary of the most classical techniques and then informally review them at the light of simple evaluation criteria. We also introduce the indexing schemes of two XML database systems: Xyleme and X-Hive, as they are used for benchmarking. We briefly summarize some more advanced techniques, notably those based on approximated dataguides and holistic joins. Next, we describe the SIOUX indexing method founded on tree-based indexing. In section 4, we formulate the algorithms for maintaining the index that have been implemented in our system. In section 5, we propose a simple benchmark for evaluating the new indexing method in comparison of that of Xyleme and X-Hive. The benchmark is run on the three systems and gives rather promising results for the SIOUX method.

2. OVERVIEW OF INDEXING SCHEMES

In this section, we first discuss the main issues to design an indexing scheme. Next, we review some of the most well known schemes and give informal evaluations for each of them.

2.1 Evaluation criteria

As pointed out above, many approaches to XML indexing have been proposed. What are the relevant criteria to consider when selecting one index type? In the sequel, we introduce a few that we believe important and that we shall use to compare proposed methods.
### 2.1 Type of identifiers

All methods require document identifiers and most uses also node identifiers within documents. Node identifiers can be composed of one or more numbers in complement to a document identifier. They can be attributed in such a way that they memorize the parent-child relationships. The disadvantage of not having element identifiers in the index is that post-processing is required to locate document elements after selecting documents through the index. If the number of selected documents is large and if documents are also large, post-processing might be considerable [15]. However, if document are small with complex structures, node identifiers may become costly in place.

#### 2.1.2 Structural Search

Most XPath queries involve the exploration of the descendant (and sometimes the ancestor) of a given set of nodes. A naïve approach is to use join algorithms on the parent-child relation, which is time consuming in general. However, join algorithms can be improved using for example B-tree structures. Another solution is to maintain and navigate the graph structure of the documents. The most sophisticated algorithms are based on element numbering schemes that determine if a node is an ancestor or a descendant of another simply by comparing identifiers.

#### 2.1.3 Keyword Search

Keyword search and its many variations must be efficiently performed, notably to support the CONTAINS predicate of XQuery and the new developments in XQuery Text. In general, keyword search requires a specific index, in its simplest form an inverted list implemented as a B+-tree. Signatures, i.e., Boolean vectors resulting from hashing keywords present in documents or elements, have also been used in traditional information retrieval systems. Inverted lists have been sometimes mixed with structural indexes. To efficiently support XQuery text, the capabilities of prefix, postfix, infix search are important. So is the capability of measuring the distance of two words (in words) in a document. Further, similarity search may require evaluation of semantic proximity of elements. This is achieved in document systems through vector encoding of documents using classical tf-idf (term frequency - inverse document frequency) weight of the important words in a document. Few indexes are supporting similarity search.

#### 2.1.4 Size of indexes

The size of the index is an important criterion. The size of an index depends on the number of entries (e.g., one per document, one per each element, or one per terminal element) but also of the average size of an entry. In general, there is one entry per important word in document content; in addition, certain index requires one entry per name of elements. Graph index may list all paths in documents. For each entry, relevant documents are referenced, with optionally a list of relevant node identifiers per document. Keeping the index small is a good heuristics.

#### 2.1.5 Document Update

The behavior of an index when documents are updated is an important issue. Some types of index are not robust when documents are updated and must be rebuilt, at least for the updated document. Others are incremental and only require deletion and addition of updated elements. In between, solutions have been proposed that are robust for a limited amount of updates per document.

#### 2.1.6 Query Benchmark

Comparing performance of index schemes is always difficult. In general, each proposal presents some experimental results that compare with one or two other approaches by varying some parameters. Traditionally, the collection of the plays of Shakespeare, the XMak benchmark [30], and benchmarks generated from DTDs using the XML Generator from IBM are used. While Shakespeare and XMark are trees, some graph structured XML can be generated using the Flix Markup Language (FlixXML) for movie reviews and the GedML markup language for the genealogical XML data. Most benchmarks specify regular DTDs for XML documents. We are interested in irregular document structures, where few documents share the same DTD. Comparing index size is easy but comparing query performance is a more difficult issue. With XQuery, large panoply of queries can be expressed, including free-text searches [31]. A significant benchmark has to cover most cases without being too complex, which is a difficult compromise.

### 2.2 Indexing Rooted Paths: The Dataguide

Dataguides have been proposed by Goldman & Widom at VLDB97 [13]. They can be perceived as dynamic schemas, but are often extended with targeted node identifiers to serve as index. Such indexes are called targeted dataguide. Dataguide can also helps in query formulation. They are rather concise and accurate structural summaries. Every path in the database has one and only one corresponding path in the dataguide with the same sequence of labels. A rooted label path of a node is a sequence of labels defining a path from the root to the node. To achieve conciseness, a DataGuide describes every unique rooted label path of a collection exactly once. To ensure accuracy a DataGuide encodes no rooted label path that does not appear in the collection. In Lore [23], a dataguide is itself represented as an XML object, but this is an implementation option. Dataguides for a given collection of documents are in general not unique. A strong dataguide is such that any indistinguishable rooted label paths p and p' on the dataguide are also indistinguishable on the database (i.e., p(G)=p'(G) => p(DB) = p'(DB)). A strong dataguide is unique for a given collection of XML documents. Targeted dataguides store for each path node identifiers corresponding to elements having the given path labels. The index is thus an unordered graph with a list of document and node identifier for each path of length 1 to N. Dataguides are difficult to maintain for documents with cycle as paths should be limited in size. For searching on keywords, nothing is provided and an additional value index shall be maintained as in Lore [23]. The main properties of the Targeted Dataguide index are summarized in table 1.

### 2.3 Indexing Template Paths: The T-index

A template-based indexing technique (T-index) has been proposed by [24]. The idea is to build indexes not on all paths, but on selected path templates. The technique consists in grouping database objects into equivalence classes containing objects that are indistinguishable w.r.t. a class of paths defined by a path template. A path template t has the form T1 T2 … Tn where each Ti is either a regular path expression or one of the
following two placeholders $P$ and $F$. A query path $q$ is obtained from $t$ by instantiating each of the $P$ placeholders by some regular path expressions and each of the $F$ placeholders by some formula. T-index indexes all sequences of objects connected by a sequence of path expressions in conformance with the template.

T-indexes are very general, as the template generator cover a wide range of paths, including all paths, paths with given prefix, postfix, etc. There are interesting particular cases. 1-index indexes all objects reachable through an arbitrary path expression $P$ from a root. 2-index is a non-deterministic version of the strong data guide and coincide with it for tree-structured data. 2-index indexes all pairs of objects connected by an arbitrary path expression $P$. While each T-index is designed for a particular class of queries given by one template, it can be used to answer queries of more general forms. Computing this equivalence relation may be expensive (PSPACE complete), so the authors consider finer equivalence classes defined by bisimulation or simulation, which are efficiently computable. A T-index is built from these equivalence classes by constructing a non-deterministic automaton whose states represent the equivalence classes and whose transitions correspond to edges between objects in those classes.

The beauty of T-indexes is the generality and the fact that T-index can be used to help solving a wide variety of queries. A T-index behaves like a concrete view of the database that can be used to reformulate queries and solve them more efficiently. T-index main properties are summarized in table 1.

2.4 Indexing Frequently Used Paths: APEX
APEX is a method to manage adaptive path indexes for XML data [6]. While the strong dataguide maintains all paths from the root, APEX does not keep all but utilizes frequently used paths to improve query performance. APEX maintains in the index all paths of length 1 plus additional required paths, i.e., the set of labels plus some composed paths, those that are frequently queried. APEX assumes the existence of a database maintaining the query workload and determines frequently queried paths by selecting those above a given support.

An APEX index consists of two structures, a graph structure representing the structural summary of the data, and a hash tree structure that associates required paths to nodes of the graph structure. For efficiency, each node of the hash tree is implemented as a hash table on the labels. Terminal nodes refer to a graph structure node. The hash tree is used to find nodes of the structure graph for given label path, also for incremental update. A graph structure node contains the list of node identifiers of the XML documents, similarly to a targeted dataguide node. The APEX graph structure is simpler that the dataguide and accessed through the hash tree. Nice algorithms are proposed to perform incremental updates of the hash tree and graph structure. APEX main properties are summarized in table 1.

2.5 Approximate Structural Summaries
To answer all path queries accurately, dataguides must encode information about long, seldom-queried paths, leading to increased size and complexity. In [17], the authors introduce the $A(k)$-indices, a family of approximate structural summaries which are more compact. They are based on the concept of $k$-bisimilarity, the idea being to group in similar classes nodes that are locally similar, as it often exists in XML databases (i.e., repetitive structures). $A(k)$-indices are space efficient and effectively support path queries, but they can lead to false drops.

The $D(k)$ index [4] is a generalization of the $A(k)$-index that possesses the adaptive ability to adjust its structure according to the current query load. Instead of specifying the same local similarity, $k$, for every equivalence class in the index graph, the $D(k)$-Index uses the most effective local similarities for equivalence classes according to the current query load. A local similarity $k$ is defined for each node in the graph based on the node query load. This dynamic similarity approached is proved to be efficient by the authors who also introduce efficient index updating algorithms.

2.6 Indexing Rooted Paths with Values: The Index Fabric
The index Fabric has been introduced in [7]. It includes both an efficient implementation of the dataguide and a clever extension with values in place of identifiers. To shorten the paths, labels are first encoded with one or more (if required) letter. All paths from the root to a leave containing data are prearranged as a sequence of encoded labels followed by the value as a string. For example, the path /Restaurant/Adress/City[Paris] is encoded as RACParis. To store the encoded strings, the method uses an efficient index for strings, i.e., a Patricia trie. A Patricia trie is a simple form of compressed trie that merges single child nodes with their parents. A balancing mechanism is added to the Patricia tree to guarantee constant access time when searching for paths of length N. Path expressions including predicates on values for elements are performed as string search. Notice that the index does not keep information on non-terminal nodes. It does not manage node identifiers either. Also, it does not keep the order of elements in documents although it stores terminal node values. Maintaining the index Fabric on updates is not an easy task. Table 1 gives an overview of the main properties of the index Fabric.

2.7 Indexing Values and Labels with NID
Several methods have been proposed to replace the structural graph index by a node numbering scheme. Node identifiers (NID) determine elements in documents themselves identified by a document identifier (DID). The NID attribution device aims at replacing structural joins by function computations: $is\_parent(NID1,NID2)$ and $is\_ancestor(NID1,NID2)$ are simple and efficient functions to check the parent and ancestor relationships. We detail some of the proposed schemes below. Some schemes have been proposed to make identifiers stable in presence of updates.

2.7.1 Virtual Node Identifiers
A simple scheme has been proposed in [21]. The document structure is mapped on a k-ary tree. Node identifiers are assigned according to the level-order tree traversal. The parent of node $i$ is $parent(i) = (i-2)/k + 1$ and the $j$-th child of node $i$ is $child(i,j) = k(i-1) + j + 1$, $k$ being the arity of the virtual tree. Among the virtual children of node $i$, be $k(i-1)j+1 \leq j \leq k-1$, only $ri \leq k$ are real nodes. The determination of the real existence of a node can be done in several way, e.g., using a bitmap associated to each document.
Although the method is simple and effective, the determination of the real existence of parent/child relationships and the choice of the arity of the tree are difficulties. Updates require reassigning all node identifiers when documents become larger in some nodes that the arity of the virtual tree. Selecting a large arity implies large identifiers and bitmaps. The authors suggest the possibility to have different arities per level, which is complex.

2.7.2 Pre-Post Order and Tree Level Identifiers in Xyleme

Xyleme is an efficient XML warehouse used in several document-oriented applications. One key feature of Xyleme is its indexing mechanism, which makes possible to process most text-oriented queries without accessing the data. Xyleme stores XML DOM trees in pages using the NATIX (from Mannheim University) repository, which compresses and maps XML trees on one or more pages. Xyleme keeps document ID and element ID. While document IDs are simple references, two forms of element ID are supported:

- The bit-structured scheme gives the structural position of the node by level in the tree. A NID clearly lists all ancestors of the context node.
- The prefix-postfix scheme proposed in [10] keep the preorder and postorder of each node as identifier, generated by a left-deep traversal of the document tree. The ancestor relationship is determined by X ancestor of Y iff pre(X) < pre(Y) and post(X) > post(Y).

Using these types of identifiers, Xyleme maintains an index of labels and values (keywords of contents). To solve an XPath query, the index is accessed with given labels and word predicates, then lists of identifiers are processed to check ancestor/descendant relationships and conjunction/disjunction of conditions. The index is kept in memory and the selected elements can be accessed in parallel by the repositories implemented on multiple computers. Table 1 sums up the main indexing features of Xyleme.

2.7.3 Interval Encoding

A storage efficient technique to identify XML nodes and keep document size is known as interval encoding. It has been used to store efficiently XML documents in relational databases. The general format consists in associating a pair of values (order, size) to each node in the XML graph, such that x is ancestor of y iff order(x) < order(y) ≤ order(x) + size(x) [8].

One way to generate a valid interval encoding is to perform a depth-first traversal of the document tree, using an incrementing counter that assigns the left value to a node when it is first encountered, and the right value when the node is seen for the last time. This is simply a refinement of the Dietz numbering scheme. It can be made resilient for updates by keeping size for the context node.

### Table 1: Overview and synthesis of some indexing methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Dataguid e</th>
<th>T-Index</th>
<th>APEX</th>
<th>FABRIC</th>
<th>Xyleme</th>
<th>X-HIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identiﬁers</td>
<td>DID+NI D</td>
<td>DID+NI D</td>
<td>DID+NI D</td>
<td>DID</td>
<td>DID+NI D</td>
<td>DID+NI D</td>
</tr>
<tr>
<td>Navigation</td>
<td>2 Int</td>
<td>2 Int</td>
<td>2 Int</td>
<td>1 Int</td>
<td>2 Int</td>
<td>3 Int</td>
</tr>
<tr>
<td>Trie Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keywords</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Trie</td>
<td>Index</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>index</td>
<td>index</td>
<td>index</td>
<td>search</td>
<td>search</td>
<td>Index</td>
</tr>
</tbody>
</table>

2.7.4 Relative Region Coordinates

To deal with the content update problem that is not adequately addressed in most numbering proposals, [16] introduces a specific mechanism called relative region coordinates. The idea is to express the coordinate of an XML element in relation to its parent element coordinates: the start and the end of a node are relative to the parent start. Using the relative region coordinate to locate XML elements, we know only where the elements start and end inside the region of their parent elements.

As in index related coordinates are stored together, we need to update only a portion of index file in case of content updating. Also, it is possible to compute the absolute region coordinate of an element by summing up the relative region coordinates of its ancestors and itself. In consequence, the indexing scheme requires update of only a small portion of index file in case of updating.

2.7.5 Others Approaches Using Node Numbering

In addition to node numbering, indexes have been proposed to skip elements that do not participate in any of the results, notably in the case of regular path expressions [22] [5]. Numbering techniques based on interval encoding with (DocId, LeftPos, RightPos, LevelNum) as node identifiers complemented with efficient ordered join algorithms have been exploited for processing efficiently patterns of selection predicates on multiple elements related to a tree structure, namely twig queries [3]. Recently, [15] has proposed merge-based and index-based algorithms to efficiently process twig queries with OR predicates. [9] and [14] have shown that even for simple queries, there are a number of options for how to combine tree-based navigation and structural joins based on information retrieval-style inverted lists, and that by tightly integrating the two types of processing, one can obtain faster query response time. [19] also utilizes navigation based techniques to solve a wider range of queries. More recently, numbering schemes have been used to encode XML data trees and twig queries into sequences [28][26]. Twig patterns are retrieved by performing subsequence matching on the structure-encoded sequences. This yields to query-driven input-probing methods where the structure of the query is probed in a sequence index (e.g., a trie) to retrieve the parts of the XML relevant encoded sequence.

2.8 Overview and Classification

Table 1 gives an overview of the main features of the main analyzed indexing techniques. We add X-Hive [29] as it is an interesting product that we use for benchmarking. Identifiers are in general document identifiers plus node identifiers. Node identifiers can be composed of prefix/postfix or hierarchical addresses as in Xyleme. This avoids structural join and navigation. The index Fabric does not use node identifiers. The size of indexes is generally linear in function of the number of nodes and the number of paths. Two methods are clearly targeted towards solving regular path expressions with wildcards.
The reviewed indexing techniques can be presented in a classification hierarchy as portrayed in Figure 1. The first level classifies methods according to the technique used for resolving hierarchical path expressions. Methods using graph traversal maintains some form of dataguide, i.e., an index of all rooted paths. Fabric used an encoding of all paths stored in a Patricia trie that can be explored using text search. Methods based on a numbering scheme require an additional index containing at least one entry per label (often merged with the value index); this index is the starting point of the search to get lists of identifiers that can be compared. Notice that the reviewed indexing techniques use label and often paths as index entries, which makes index large in general. All index forget about path orders and number of occurrences. Furthermore, most methods except Fabric have difficulties with intensive updates, either to maintain index entries or identifiers.

![Classification of some XML indexing methods](image)

**3. A NEW INDEXING SCHEME BASED ON TREEGUIDE**

We propose a new index scheme coupling a structure index and a value index. The structure index is based on a treeguide, a finer data structure than dataguide: index entries are ordered trees rather than paths. The value index is a conventional inverted list.

**3.1 The Structural Index**

The structural index defines equivalent classes of documents, one class for each XML tree structure existing in the database. The structure is called a treeguide.

**Definition 1:** *Treeguide.* Let \( x \) be an XML document. The treeguide \( T(x) \) is the ordered tree that: (i) includes exactly every instance of the label paths in \( x \) and no more. (ii) Generates the XML document structure when traversed in preorder.

Notice that a treeguide differs from a dataguide as multiple paths and elements in different orders are distinguished; also, it is not an index but a document structure, with no reference to data. For simplicity, we consider only trees although the definition could be extended to graphs with cycles (for encoding references). Attributes in documents are processed as elements, but the order of attributes is irrelevant. To deal with attributes as elements, we include them as the first children of the node they are attached to and we order them by alphabetical order.

**Definition 2:** *Treeguide Index.* Let \( C \) be a collection of XML documents \( x_1, x_2, \ldots, x_n \). A treeguide index simply gives for each different treeguide of the document collection the list of documents \( x_i \) having this treeguide.

A treeguide index shall implement efficiently the following functions:

- Retrieve a given treeguide if it exists.
- Retrieve all documents of a given treeguide.
- Add a given document \( x_{n+1} \) in the collection and maintain the index.
- Delete a given document \( x_i \) from the collection and maintain the index.
- Find all documents satisfying a given XPath in the collection.
- Allow easy identification of document nodes to join with value index on identifiers.

We discuss in the next section possible implementations of treeguide index and describe our. It is important to notice that a treeguide index defines equivalent classes of documents per schema, i.e., \( x_i \equiv x_j \) iff \( TG(x_i) = TG(x_j) \). Unlike current indexing schemes, we maintain an index with one entry per treeguide and not per path. Also, the treeguide index references objects and not elements. However, as treeguides distinguish orders and duplicates, the index is able to deliver the hierarchical address (i.e., 1.2.5 for node 1, child 2, child 5) of any node satisfying a path. Treeguides are richer than dataguides and avoid storing node identifiers within the structural index, which saves in index size (see evaluation).

In the case of a homogeneous collection of objects having an identical treeguide, our structure index will have only one entry giving the list of objects. To avoid such situations with non-discriminating structure indexes, if an entry of the index is too large, we propose to add signatures of objects to further distinguish the equivalent objects; thus, \( x_i \equiv x_j \) iff \( TGG(x_i) = TGG(x_j) \), where \( TGG \) stands for Signed Treeguide. Let us recall that a signature of an element is the binary encoding of some hash function on \( [0,N] \) applied to each keywords of the element. Thus, a signed treeguide can be seen as a treeguide plus an empty tag whose label is the signature.

**Definition 3:** *Signed Treeguide Index.* Let \( C \) be a collection of XML documents \( x_1, x_2, \ldots, x_n \). A signed treeguide index gives for each different signed treeguide of the document collection the list of documents \( x_i \) having this signed treeguide.

The generation of the signature tag can be seen as a pre-processing step adding a signature labeled tag to the document. It can also be integrated dynamically to the signed treeguide index functions: signatures can be generated to split...
dynamically index entries that are too large. Notice also that signature can be used to focus on path queries with keywords predicates (i.e., CONTAINS(., “keywords”)): only entries with relevant signatures for the given keywords have to be retrieved. We will not develop further this possibility to distinguish treeguides with signatures, but this can be useful in case of many documents of similar structure to develop some extensible partitioning through treeguides.

3.2 The Value Index
To accelerate value search (exact match, less or greater predicates, keywords, phrase match, etc.), we use a classical inverted list scheme. For each keyword appearing as element or attribute value, the value index gives the document id, the identifier of the element in the document treeguide, and the offset of the keyword in the element. The value index could be organized as a B-tree or a hash table; we implement the last solution.

Definition 4: Value Index. The value index is a mapping giving for each keyword the list of document identifiers with the node identifiers and relative addresses of elements containing the keyword.

An alternative possible organization for the value search accelerator consists in using a signature file. The storage utilization of a signature file is better than that of an inverted list. Each content-enclosing element or attribute (i.e., leaf node) of an XML file can be signed. A signature file can be represented as table SIGN (DocId, EleId, signature), where DocId is a document identifier, EleId an element identifier, and signature is computed by setting to 1 only the bits whose ranks result from the hash function applied to an element keyword. Giving a list of keywords, the hash function determines a signature s. All couples <DocId, EleId> whose associated signatures include s are potential candidates for answering the value search. While the value index determines exactly the matching elements, the signature file determines only possible matching elements. One or the other structure can be used in conjunction with our treeguide index, which is a nice feature. For simplicity, we implement the value index as explained below.

4. MAINTAINING TREEGUIDE INDEX
As explained above, the treeguide index keeps the mapping between treeguides and documents. The structure has to be efficient, compact, and extensible (e.g., to support signature). We divide the mapping in two steps: (i) mapping Treeguides (TG) to treeguide identifiers; (ii) mapping treeguide identifiers (TGI) to XML objects.

4.1 Mapping treeguides to treeguide identifiers
All treeguides are integrated in a union tree as exemplified in Figure 2. This global treeguide structure (GTG) is optimized. It is a graph in which: (i) A node represents an XML element or attribute of one or more treeguides. (ii) An edge represents either a parent / child (down edge) or a preceding / following sibling relationship (next edge) in at least one treeguide. To reduce the size of the union graph, the lists of nodes at a given level in the union tree are merged together in sequence order in such a way that the total list be minimum in number of entries.

Thus, all nodes at a given level are linked together, but identifiers distinguish treeguides as explained below.

Nodes are labelled with the corresponding element name or attribute name prefixed by @. To be able to retrieve a treeguide, nodes belonging to a treeguide are marked with the corresponding treeguide identifier. In other words, the set of treeguide identifiers a node belongs to is added to its label (e.g., book: 1,2). The order of element nodes in each treeguide is preserved at each level. The GTG can be updated by inserting nodes at existing levels or at a new bottom level.

4.2 Mapping treeguide identifiers to XML objects
We use a simple indexed table MAP_TGI to map a treeguide identifier to an XML object. The entry number in the table corresponds to the TGI, TGI are integers allocated from 1 to N; when a TG is deleted, the TGI is marked free for latter allocation to a new treeguide. The table is composed of two columns as shown in Figure 3. The first contains the number of objects with the given treeguide, the second is a pointer to the first bucket containing the XML object identifiers. The table is maintained in main memory. A swapping mechanism is implemented to write it on disk when required. Identifier buckets are of fixed size and simply organized in a chained list.

Figure 2: Example of GTG

To extract a TG of given identifier TGI from the GTG, a simple procedure consists in traversing the tree and extracting all nodes of identifier TGI with the corresponding edges. The advantage of the proposed structure is to make possible the factorisation of nodes at the same level in all TG. If a node exists for a given TG at a given level, the cost to represent it for another TG is an identifier (an integer indeed). The GTG is thus a structure whose number of nodes in worst case is the number of element and attribute pathnames in the database. Thus this is similar to a structural guide. Notice that if a TGI appears in a node, all its ancestors shall contain this TGI as a TG containing a path contains also all its sub-paths.

Figure 3: Example of MAP_TGI
Figure 3: The mapping of treeguide identifiers to matching objects

4.3 Indexing values
Each value of an atomic type (e.g., numeric, date, url) or each significant keyword for a text type is indexed through an inverted list. Significant keywords in texts are simply determined from a list of excluded words (e.g., stop words). The inverted list is organized as a hash table mapping each constant value to a list of doublet <document identifier, node identifier>. The node identifier is a number identifying the node in the treeguide (we use a pre-order traversal to identify nodes but other numbering schemes are possible). The offset of the word could be additionally memorized for answering queries of type "word followed by word" or "phrase search", as required in XQuery Text.

An interesting alternative would be to use extensible hashing on keys <constant, node identifier> to store and retrieve the tuples <constant, document identifier, node identifier, offset> in a file. The advantage would be to access directly to elements of searched path for a given keyword. This would be efficient to solve queries such as /Book/Author[contains(Address, "Paris")]: the GTG and its associated MAP_TGI will determine objects with structure /Book/Author/Address and compute the Address node identifier; the hashing will determine objects with value "Paris" for the relevant node identifier.

Let us point out that node identifiers are simple integers obtained by traversing the treeguide in some fixed order (e.g., preorder). From the GTG, the TG of an object of given identifier can be extracted as explained above. Thus, a parallel traversal of the TG and the object is possible to determine the node content. The fundamental of our method is to manipulate an object always in association with its schema (a tree), as well as in traditional relational DBMSs (a table structure). Thus, node identifiers can be derived from treeguides and encoded as compact integers.

4.4 Searching a Treeguide
When inserting a new XML object in the database, it is required to determine if the associated treeguide already exists in the GTG. If it exists, the treeguide identifier is simply returned; the constant values of the leaves are indexed and the object can be inserted. If it does not exist, the new treeguide must be inserted in the GTG with a new TG1, and the document reference is inserted in the MAP_TGI table.

The search problem of a TG in a GTG can be seen as an extension of the well know "tree pattern matching" problem [18] [27]. Notice that this algorithm is also useful for solving (ordered) tree pattern queries. The general problem can be formulated as follows: given two labeled ordered trees, the pattern and the target, find all occurrences of the pattern in the target. In our case, the pattern is TG and the target GTG. In addition, several constraints have to be fulfilled: (i) Nodes are marked with identifiers so that we only accept patterns whose nodes are all marked with a common identifier. (ii) The matching nodes have to be of same depth. (iii) The pattern has to be strictly included, without additional leaves.

In the problem of ordered tree inclusion two ordered labelled trees P and T are given, and the pattern tree P matches the target tree T at a node x, if there exists a one-to-one map f from the nodes of P to the nodes of T which preserves the labels, the ancestor relation and the left-to-right ordering of the nodes.

More precisely, the problem can be formalized as follows; we have to find a one-to-one map f_i from the nodes of TG to the nodes of GTG preserving:

1. The root: The root of TG maps to the root of GTG.
2. The labels with a common identifier i: ∀ v ∈ TG, v and f_i(v) have the same label with identifier i among their set of identifiers.
3. The left-to-right ordering of the nodes: ∀ v_1, v_2 ∈ TG, v_1 is left of v_2 ⇔ f_i(v_1) is left of f_i(v_2)
4. The depth of the trees: ∀ v_1, v_2 ∈ TG, v_1 is at the same depth that v_2 ⇔ f_i(v_1) is at the same depth that f_i(v_2)
5. The covering of all nodes in GTG: ∀ v ∈ GTG, f_i⁻¹(v) ∈ TG.

The algorithm to check the existence of a treeguide in the GTG is derived from the tree pattern-matching algorithm of [27] to our problem by supporting the additional constraints. The algorithmic cost is at worst in O(s+g) where s is the number of nodes of TG, g is the number of nodes of GTG. This cost is clearly less than this of well-known "tree pattern matching" algorithms (e.g., cost is of order s^g in [18]). This is due to the fact that we are in a particular case with the supplementary constraints described above.

The algorithm is illustrated in Figure 4. It portrays a simple GTG and three treeguides denoted TG1, TG2 and TG3. Let us exemplify the algorithm with TG1. The algorithm starts at the root and tries to match the trees level by level. At level 1, TG.A matches with TG1.A; thus, candidates treeguides are GoodTG= {1,3,4}. TG.D does not match with B:2;3; thus, treeguides {2,3} cannot be candidates (BadTG= {2,3}).

TG.D matches with TG1.D, thus GoodTG={1,3,4} ∩ {4} = {4} is the only remaining candidate, which is selected as not in BadTG. TG2 is not included in GTG as there is no C node at level 1 in GTG. TG3 is not included too as B is present (thus GoodTG= {1,2,3,4}) but A is not present, neither D, neither the second occurrence of A; thus BadTG= {2,3}) which yields that 2 and 3 are excluded.

![Figure 4: Searching a TG in a GTG](image-url)
4.5 Inserting a New Treeguide
We now present the insertion of a new treeguide TG in a global treeguide GTG. The global treeguide does not contain TG. The insertion proceeds recursively level by level. The problem is to find the best matching sub-sequence for each level, so as to limit the size of the GTG. The insertion algorithm is given in annex. We describe the principles with a simple example below.

Let S be a sequence of tags to insert at level i. For example, consider the insertion of TG2 in GTG of Figure 4; at level 1, S= (A—B—C). For each tag in S, the algorithm searches for all positions where it appears if any. For example, A appears in 1 and 4; B in 2; C nowhere. Next, all possible placements for matching letters are checked. The best placement is the one with maximum matches; in case of several with same score, the first is simply selected. For example, A can be placed in 1 and B in 2, C being anywhere after 2, say 3; the score of this placement is 2. A can also be placed in 4, B in 5, and C in 6; the score of this placement is 1; thus the first placement with A in 1, B in 2 and C in 3 is selected. Finally, all tags in the selected placement have to be marked with the new TG identifier.

4.6 Deleting a Treeguide
A treeguide is deleted when no XML document conforms anymore to its structure. The deletion is triggered when the count of objects with the given treeguide becomes 0 (Card= 0 in MAP_TGI table, see Figure 3). To remove the TG in GTG, we remove the TGI from any node containing it. If a node contains no more TGI, the node is removed. This requires a traversal of the GTG tree.

4.7 Resolving an XPath Expression
In this section, we introduce the XPath query-processing algorithm using the treeguide index and the value index. For simplicity and to state clearly the part of XQuery expression that our system currently supports, we consider queries of the form:

\[ \text{FOR } S^V \text{ in } <\text{forest}> \text{ WHERE } (S^V/XPath1)^* \text{ RETURN } (S^V/XPath2)^* \text{ formally } \{ XPath2^* \mid XPath1^* \}. \]

XPath is filtering expressions and XPath2 are projection expressions. XPath can be as general as defined in W3C XPath version 2. For simplicity and shortness, we consider only conjunctive queries but disjunctions could be supported easily in WHERE clauses. The process can be divided in three steps: (i) Determine the TGI that verifies the structural constraints. (ii) Compute the relevant element references. (iii) Extract the response elements in main memory.

4.7.1 Determination of relevant treeguides
This step consists in extracting from all regular path expressions in the query the label paths that are mandatory for the query. From the set of label paths, a simple search in the GTG of all TGs including these label paths returns the relevant treeguide identifiers. For example, let us consider the query:

\[ \text{FOR } S^B \text{ in collection("Books") } \]

\[ \text{WHERE } S^B/Book/Author[contains(.,"John") \text{ and } } \]
\[ /\text{Address/City}= "Paris" \text{ and } \]
\[ ../Category="Computer"]//Company. \text{RETURN } S^B/Book/Author/@speciality \]

This could be rewritten in XPath:

\[ /\text{Book/Author[contains(.,"John") \text{ and } } \]
\[ /\text{Address/City}= "Paris" \text{ and } \]
\[ /\text{Book/Category="Computer"]}[.//\text{Company}]//@\text{speciality} \]

The required label paths are:

\[ \{ /\text{Book/Author}@\text{speciality} \mid /\text{Book/Author}, /\text{Book/Author/Address/City, /Book/Category, /Book/Author/}
\]
\[ /\text{Book/Author/Company} \}. \]

The algorithm for determining the relevant TGIs simply searches each path expression in the GTG. For each, it collects all the instances and returns the list of all TGIs corresponding to the end-path nodes. Wildcards (*, //) have to be included in the search for retrieving children or descendants of a given name. Expanded path expressions in which wildcards are replaced by effective label paths are inserted in the set of required label paths in place of path expressions with wildcards, so as to memorize effective paths. Positional predicates ([]) are also included in label paths. To search a label path in the GTG, we use an extension of the tree pattern-matching algorithm [18]. The precise algorithm is described in annex. It is different from the TG searching algorithm described above as all relevant paths have to be retrieved and all nodes of retrieved treeguide need not be covered. As all path expressions have to be satisfied, we do the intersection of all TGI lists to determine the relevant treeguides and TGIs.

4.7.2 Computation of relevant element references
This step first transforms the set of relevant TGIs in document identifiers plus node identifiers; second, it applies the value index to determine precisely the document nodes to retrieve. Going to the MAP_TGI table, we first determine a set of relevant references of documents with the required TG. Each label path is then encoded as a node identifier by prefix traversal of the TG (a hierarchical address could also be used but would be less compact, e.g., 4 for /Book/Author and 17 for /Book/Author/Address/City in place of 1.3 and 1.3.5.2). Finally, we obtain a set of document identifiers with a set of identifiers of elements in these documents. This is the result of the structural search. If the result is empty, the query has an empty answer.

Otherwise, the value search is used to restrict the list of results, if any. Using the value predicate in the query (e.g., 4="John" and 17="Paris"), we access the value index. For each value (e.g., John, Paris), we read the corresponding entry and filter with the node identifier. Doing the intersection of all sub-entries, we obtain a set of document identifiers with a set of hierarchical addresses of elements in these documents. This gives the result of the value search.
The two sets of document identifiers are intersected, which yields the final result of this step, with the associated expanded path expressions and hierarchical addresses. Notice that some optimizations are possible:

(i) A threshold can be set for the length of any list of document identifiers: if it is too long, it can be replaced by the overflow list (meaning all documents qualify), which will not be taken into account when doing intersection of lists.

(ii) Intersection of document lists can be ordered so as to start with the most selective one. The size of the value index entries can be used to estimate the most effective order of intersections.

(iii) In case of too long lists of document identifiers, signed treeguide could be used and the signature could be checked for determining document relevance according to the value predicates.

4.7.3 Extraction of the relevant elements

The last step consists in loading in main memory the response elements whose precise address (document identifier and element identifier) has been retrieved in the previous step. If certain predicates have not been checked in the value search, a final loading of the necessary element value and a final check are required. The addresses of the elements are included in the result list of the previous step. Notice that the only required portion of documents to access as results refer precisely to those elements. This requires an object manager capable of accessing directly an element in a document, which is done through an element index managed for each document.

4.8 Resolving Ordered Tree Pattern Queries

A tree pattern query or twig query searches for a labeled tree in XML documents. It is essentially a complex selection predicate on both structure and content of an XML document. The example given above (sub-section 4.7) is essentially a twig query searching for the labelled trees represented in figure 5, assuming that we know that addresses do not contain the company tag. Most proposed algorithms to solve twig queries works with AND and sometimes with OR connectors for sibling nodes [15].

As shown above, we can handle such queries by searching for the covering paths, then union or intersect the retrieved identifiers. We can also search directly for all treeguides satisfying the relevant tree patterns in the GTG and intersect the retrieved identifiers (e.g., see Figure 5). The search is simply another application of a simple variation of the search algorithm given in annex. In general, with AND connector linking Path and Path2, both order Path1, Path2 and Path2, Path1 have to be considered. Notice that we can efficiently handle THEN connector, as treeguides are ordered trees. For example, we could handle efficiently the query in slightly extended XQuery:

FOR $B$ in collection("Books")

WHERE $B$/Book/Author[contains(.,"John") and /Address/City="Paris" then ...
//Category="Computer"] then //Company.

RETURN $B$/Book/Author/@speciality

could be solved by looking for the tree pattern given in figure 5(a), with relevant order.

5. Performance Evaluation

To demonstrate the validity of the SIOUX method, we compare our system with two industrial systems, namely Xyleme [1] and X-Hive [29] that implement efficient path-based indexing. The three systems were set on a Pentium Centrino 1,6 GHz with 1 Giga of RAM under Linux. Measures concern mainly the query execution time and the volume of the structure index that is in the core of our approach.

As most proposed benchmark have fixed collection schema, we develop our own mini-benchmark with some varying schemas. We consider three data sets whose characteristics are given in table 2. We used the ToXgene tool to generate XML documents varying on characteristics described by the four columns of table 2. The A set holds 300 documents with the same structure. The B and C sets experiment with a larger variety of schemas (22 various structures for the C set) as indicated. Set B and C contain two totally different subsets of schema. In each subset, schema varies on a few tags (in general the number of author occurrences).

<table>
<thead>
<tr>
<th>Table 2: The data sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of documents</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>Set A</td>
</tr>
<tr>
<td>Set B</td>
</tr>
<tr>
<td>Set C</td>
</tr>
</tbody>
</table>

In general, due to the opacity of the implementation of industrial systems, it is difficult to compare structure index sizes. However, as the results below show, the index size necessary to index one document proves that our solution is economic in disk storage. The sizes found with industrial products are of others order of magnitude; this is probably because they use hash tables or other block allocation techniques. Thus, surprising sizes are given in table 3.

Measures of Xyleme are captured directly using a DBMS API. From the results, we infer that the important size of indexes and the decreasing cost for indexing each document are probably due to the use of a hashing organization for the index. To index only one document, we measure an initial index size of 141176 bytes, which suggests an allocation of fixed blocks for the index.

For X-Hive, no API is available to get the size of the index structure. To get results, we measure the disk memory delta...
when creating the index and also the main memory delta. We observe no variation on disk, but an important memory variation. As we can observe in table 4, this variation is proportional to the number of documents indexed.

The results are quite encouraging for Sioux. They demonstrate: (1) For SIOUX, an average cost of 11.29 bytes to index each document that contains an average of 12 elements. Thus, the cost of one element is less than 1 byte. (2) Using the GTG structure, indexing 300 documents (Set A) of identical schemas cost almost no more than indexing one document. At the opposite, Xyleme and Xhive index all the nodes, which also explain the gap with our approach. (3) Considering the index size for only one document, table 4 (last line), shows that the cost for the three index are quite similar. (4) For the three data sets, it appears that the index size is not proportional neither to the number of documents, nor to the number of different treeguides, nor to the number of nodes in the data set. This requires further investigations, but could come from hashing or paging schemes used for storing indexes. For the worst case, another measure on a set of documents with totally different structures shows that the size is close to a 1-Index method [24].

We experiment query processing with four typical queries:

- Query on structure. It projects the documents on an XPath specified.
- Query on attribute. It is a projection on an attribute fully specified.
- Query on text value. It is a selection on a textual element containing a given value.

The queries with associated results are given in table 4.

<table>
<thead>
<tr>
<th>XQuery</th>
<th>Xyleme</th>
<th>Xhive</th>
<th>Sioux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure query</td>
<td>for $col in collection(&quot;catalog&quot;), $b in $col/catalog/book return $b</td>
<td>A: 36.2</td>
<td>A: 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B: 76.2</td>
<td>B: 12.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C: 150</td>
<td>C: 390.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B: 17.3</td>
<td>B: 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C: 31.7</td>
<td>C: 23.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B: 15.5</td>
<td>B: 20.5</td>
</tr>
</tbody>
</table>

The first three queries focus on the structure of documents, and the last query focuses on the structure and on a constant value contained in one of the elements. As with the B and C sets, documents have different schemas, the queries extracts a subset of the documents that constitute the solution. The queries have been run for the three systems on a same machine ten times; average execution times are given in ms.

For the first three queries and for a given dataset, the SIOUX execution times are roughly the same. This can be explained by the fact that the XPath is processed on the GTG structure to find directly the relevant documents. All the documents with the same structure are processed only one time, in contrast with the other approaches. Finer measures of our method show that more than 95% of the time is mainly used for the extraction of the XML returned fragments from the originally stored documents (projection). For the last case, the constant value is very selective for the three systems. We assume that the value index access and the result construction times are quite similar. That may mean that we should optimize the value index and result construction algorithms. In case of larger collection of documents, the reduced size of our index should make SIOUX behave well. In summary, this simple benchmark demonstrates that our approach achieve equal or better performance for simple XPath queries, with a factor sometimes up to 10. It should be even better for tree pattern queries.

6. CONCLUSION

In this paper, we survey the multiple methods proposed for indexing XML. Most methods indexes on document paths with node identifiers. Then, we introduce SIOUX, a new indexing method that we implemented in a native XML repository. SIOUX indexes documents with ordered structural trees. It separates the structure index and the value index, but unifies both for computing node identifiers satisfying a given predicate. The key of SIOUX is an efficient ordered tree-matching algorithm based on an optimized structure for maintaining the union schema of all documents, with associated references to documents.

A simple but covering various types of query benchmark has been run to demonstrate that SIOUX is more efficient for structure query than two existing products. For value query, it is equivalent with the current implementation. Works remain to be done to optimize projections and value search in our system. The SIOUX implementation should also be extended to solve tree pattern queries, where it should brings its full power of indexing by trees. We plan to do an extended benchmark with complex tree pattern queries and larger collections of deeper documents. But we consider the current benchmark as a good validation of our approach in comparison with industrial products.
Although XML indexing is a well-visited topic with a lot of contributions, we believe that there is still room in this domain, notably for supporting complex queries including text queries and for extending the index structures to parallel systems and peer-to-peer query processing. We believe also that there exists a large discrepancy between published methods with often convincing performance evaluations on paper and real commercial systems as we saw them during our benchmark. This is a problem for research, which should devise methods easy to implement, as it is the case for SIOUX.

7. REFERENCES


[22] Quanzhong Li, Bongki Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB 2001: 361-370


[29] Xmark: http://monetdb.cwi.nl/xml/


[31] Xifeng Yan, Philip Yu, Jiawei Han: Graph Indexing: A Frequent Structure-based Approach, SIGMOD 2004: 335-346.