SPHINX: SCHEMA-CONSCIOUS XML INDEXING

Krishna P. Leela   Jayant R. Haritsa

Technical Report
TR-2001-04

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India
`http://dsl.serc.iisc.ernet.in`

SphinX: Schema-conscious XML Indexing

Krishna P. Leela   Jayant R. Haritsa
Database Systems Lab, SERC/CSA
Indian Institute of Science, Bangalore 560012, India

Abstract

User queries on XML documents are typically expressed as regular path expressions. A variety of indexing techniques for efficiently retrieving the results to such queries have been proposed in the recent literature. While these techniques are applicable to documents that are completely schema-less, in practice XML documents often adhere to a schema, such as a DTD. In this paper, we propose SphinX, a new XML indexing scheme that utilizes the schema to significantly enhance the search process. SphinX implements a persistent index structure that seamlessly combines the schema information with standard B-tree technology, resulting in a simple and scalable solution. A performance evaluation over a variety of XML documents, including the Xmark benchmark, indicates significant benefits with regard to both index construction and index access.

1 Introduction

XML (eXtensible Markup Language) [5], by virtue of its self-describing and textual nature, has become extremely popular as a flexible medium of data exchange and storage, especially on the Internet. Correspondingly, there has been significant research activity, in both academia and industry, on the development of languages for specification of user queries on XML document repositories. Many query language have been proposed in the literature, including Lorel [1], XML-QL [10], XPath [8], etc. from which XQuery [7] has emerged as the standard. These proposals have significant differences in their syntax and formulation, but at their core, they all support *regular path expressions* (RPEs), an elegant and powerful mechanism for specifying traversal of graph-based data such as XML. For example, the RPE /bib[//book|article]/author/lastname specifies (in XPath syntax) the retrieval of the lastnames of all authors of books or articles that are reachable from the bib root element.[1]

In order to process RPE queries efficiently, a variety of XML *indexing* techniques have been proposed in the literature. These techniques include the classical Lore [16] and T-index [19], as well as more recent followups such as ToXin [21], XISS [15], Index Fabric [9], APEX [20], F&B-Index [14], Holistic Twig Joins [6], Barashev and Novikov [3] and ViST [25]. The methodology of the majority of these techniques is to first construct a graph-based equivalent of the original XML document, and then to create indexes on this graph representation.

We move on in this paper to considering situations where the XML document that is to be indexed additionally conforms to a *schema*, such as a *Document Type Descriptor* (DTD) [35]. Such situations are quite common

[1]The standard output of XPath expressions is "node-sets" (i.e. a set of node-ids); however, since our focus is on XPath *queries*, we assume that the expected output is the actual data (i.e. the *sub-trees*) associated with the result nodes.

in practice – for example, BioML [28] and MathML [34] are DTDs specified for information exchange by the genomics and mathematics communities, respectively. XML schemas have been used in the prior literature for deriving relational database schemas [24, 4], for query pruning [11] and minimization [26], for gathering document statistics [12], etc. However, using schema information to enhance *indexing efficiency* has not been considered before to the best of our knowledge.[2]

The SphinX Index

We propose here a new indexing mechanism, called **SphinX** (Schema-conscious Path-Hierarchy INdexing of Xml), intended for schema-conforming XML documents. While SphinX can, in principle, be used with any kind of schema (e.g. XML Schema [36]), including those derived *post-facto* from the data (e.g. Data Guide [16]), in this paper we focus specifically on DTDs, which are an extremely popular and common schema representation in real-world applications.

Given an XML document and its associated DTD, stored either on a file system or native XML engine[3], SphinX proceeds as follows: The document is first converted into its equivalent graph representation, called the *Document Graph*. A special feature is that this graph is created with *bi-directional* links in order to facilitate both top-down and bottom-up traversal of the graph. Then, the DTD is also converted into a graph-based representation, called the *Schema Graph*. The leaves of the Schema Graph contain, for all the paths in the document on which indexes have been built, pointers to the roots of B^+*-trees* (hereafter simply referred to as B-trees). Each B-tree indexes directly into the corresponding atomic values in the Document Graph. A pictorial overview of the SphinX system is shown in Figure 1.

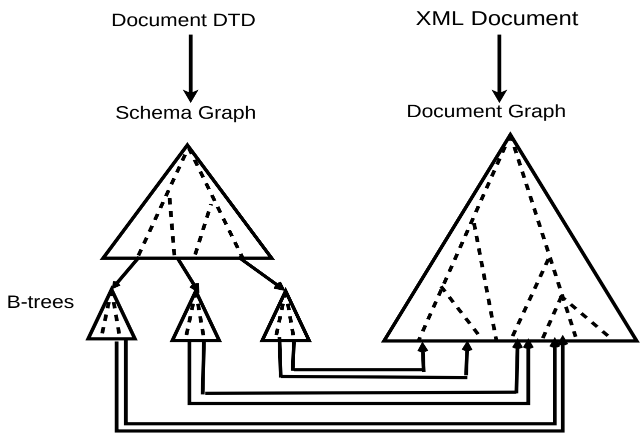


Figure 1: **Overview of SphinX System**

A crucial point to note here is that although the native XML documents may be highly complex and large in size, their associated DTDs are very *compactly* described through the use of regular expressions. For example,

[2]Industrial products such as Tamino [33] and eXcelon do use schemas for a variety of purposes, including determining the storage of XML objects and specifying the indexing properties of individual elements and attributes, but not for supporting indexing per se (personal communication with the technical personnel of these products).

[3]When XML documents are stored on RDBMS backends (e.g. [24, 4]), the standard indexing mechanisms of relational engines have to be used.

the DTD for *Xmark*, the popular XML benchmark [23], is less than 5 KB in size. This compactness is also captured in our Schema Graph representation of the DTD – the Schema Graph takes up only about 50 KB. This means that it can be reasonably assumed that although the Schema Graph is a persistent structure, it can always be loaded in its entirety into memory.

The Schema Graph supports the efficient determination of exactly those paths in the Document Graph that are *relevant* to a given query – this feature is extremely important since path identification forms the basis of answering RPE queries. That is, the Schema Graph ensures the *selection* of only the relevant paths making it both "precise" and "complete". In contrast, indexing techniques on schema-less documents require, from a candidate set of paths, the explicit evaluation and subsequent *rejection* of paths not relevant to the query (i.e. elimination of "false-positives"). The rejection process may require arbitrarily long traversals of the Document Graph, which is typically a large disk-resident structure, making the procedure extremely expensive.

Our choice of B-trees as the core indexing mechanism is because they represent a mature and scalable technology, having been successfully used in the database world for over two decades. This is in contrast to the *specialized* index structures such as "instance functions" [21], and "multi-layered Patricia tries" [9], that form the basis of some of the prior (schemaless) XML indexing techniques. Developing index management strategies for these new kinds of structures is still an open issue, making it difficult to verify the viability of the associated schemes. Therefore, we have specifically chosen to use B-trees in SphinX since all such issues have already been satisfactorily resolved.

In this paper, we present detailed algorithms for SphinX index construction and index access. Our algorithms handle both *simple* RPEs (i.e., fully specified paths) and *general* RPEs (i.e., including wildcards (// and *), alternation (|), and optional (?) operators). A novel feature is that they also efficiently support advanced functionalities such as *range queries*, *recursion*, *IDREFs* and *preservation of XML order*, which are all essential requirements in real-world environments.

Performance Results

To evaluate the effectiveness of our approach, we have conducted an elaborate evaluation of SphinX over a representative set of real and synthetic XML documents, including some generated from Xmark [23]. The documents cover a range of sizes, document characteristics and application domains. An important feature of our experiments is that they include documents large enough that they *cannot be completely stored in memory*. For example, we have included documents whose size is in excess of 1 GB. This situation may be expected to frequently arise in practice since indexing is typically done on large collections of XML documents. The workload to the system consists of a variety of XML queries involving both simple and general regular path expressions.

We consider several metrics including the *index construction time*, the *index space consumption*, and, of course, the *query processing time*. To our knowledge, there do not exist any prior XML indexing systems that utilize schemas in general, and DTDs in particular. Therefore, we have attempted to place the SphinX performance results in perspective by comparing its performance with a representative schemaless approach. Also used as a comparative yardstick is **DGScan** (Document Graph Scan), which represents the performance obtained in the *absence of any index*, necessitating a scan of the Document Graph to answer any query. Our experimental results show that SphinX's schema-consciousness results in significant improvement on all of the metrics.

Contributions

To summarize the contributions of the paper:

- We present, for the first time, a persistent XML indexing mechanism that utilizes schema information to improve the efficiency of the searching process and whose functionalities include handling simple and general RPEs, range queries, recursion, IDREFs, and preservation of XML order.
- We provide a detailed performance evaluation of our indexing mechanism for a variety of simple and general RPE queries over a representative set of large disk-resident XML documents.

1.1 Organization

The remainder of this paper is organized as follows: In Section 2, we present an overview of the SphinX approach. Mechanisms for index construction and index-based query processing are discussed in Section 3. The system architecture and implementation details of SphinX are outlined in Section 4. The performance model and experimental results are highlighted in Sections 5 and 6, respectively. Related work is reviewed in Section 7. Finally, Section 8 summarizes the contributions of the paper and outlines future research avenues.

2 Overview of SphinX

In the most general scenario, the inputs to the SphinX system are sets of DTDs and sets of XML documents, with each document conforming to one among the set of DTDs. For ease of exposition, we will assume in the remainder of this paper that there is a single DTD and a single XML document in the database. Extending the system to a multi-DTD/multi-document scenario is straightforward.

An example DTD of a bibliographic database (on the lines of DBLP [31], the popular computer science bibliography), and an XML document fragment that conforms to this DTD, are shown in Figures 2a and 2b, respectively. The corresponding Schema Graph with leaf-level B-trees and the associated Document Graph, are shown in Figurse 3 and 4, respectively (in Figure 3, the B-trees of the Schema Graph point to the OIDs of the corresponding nodes in the Document Graph). We describe the structures in these figures in more detail below.

2.1 Document Graph

The Document Graph is a *rooted node-labeled* graph where the elements of the XML document form the nodes, the edges represent parent-child relationships, and the atomic values (i.e., PCDATA) form the leaves. The graph is constructed by parsing the source XML document in a depth-first manner. The label *&i* (where i is an integer) associated with each node represents the OID of the node.

Attributes, as shown in the example, are represented similarly to elements, and form additional nodes in the graph, hanging off their associated parent elements. All IDREFs point to their associated IDs, an example

```
<!ELEMENT  bib  (book, article)* >
<!ELEMENT  book  (author+, title, publisher) >
<!ATTLIST   book  RefID  IDREF  #REQUIRED >
<!ELEMENT  article (author+, title, conference?) >
<!ELEMENT  title  CDATA>
<!ATTLIST   article  ID  ID #REQUIRED>
<!ELEMENT  article  conference  CDATA >
<!ELEMENT  publisher  (name, address*) >
<!ELEMENT  name  CDATA>
<!ELEMENT  address  CDATA>
<!ELEMENT  author  (firstname?, lastname)>
<!ELEMENT  firstname  CDATA>
<!ELEMENT  lastname  CDATA>
```

(a)

```
<bib>
  <book  RefID="1997">
    <title>Elements of ML programming</title>
    <author>  <lastname>Ullman </lastname>  </author>
    <publisher> <name>Prentice Hall   </name>
     <address>New Jersey 07458<address> </publisher>
  </book>
  <book  RefID="2001">
     <title>Foundation for Object/Relational Databases</title>
     <author><lastname>Darwen </lastname> </author>
     <publisher><name> Addison−Wesley </name></publisher>
  </book>
  <article  ID="2001">
      <author> <lastname>Ullman</lastname>
                  <firstname> Jeffrey D.</firstname>
      </author>
      <title> Querying Websites Using Compact Skeleton </title>
      <conference> ICDT </conference>
  </article>
</bib>
```

(b)

Figure 2: Example DTD and XML fragment

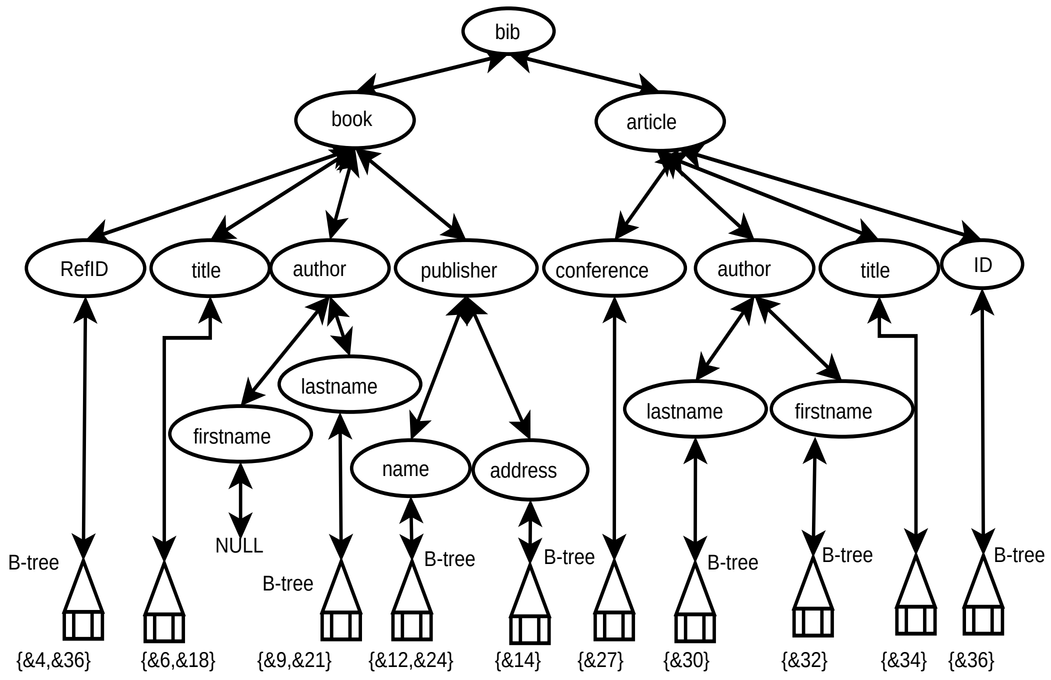


Figure 3: Example SphinX Index

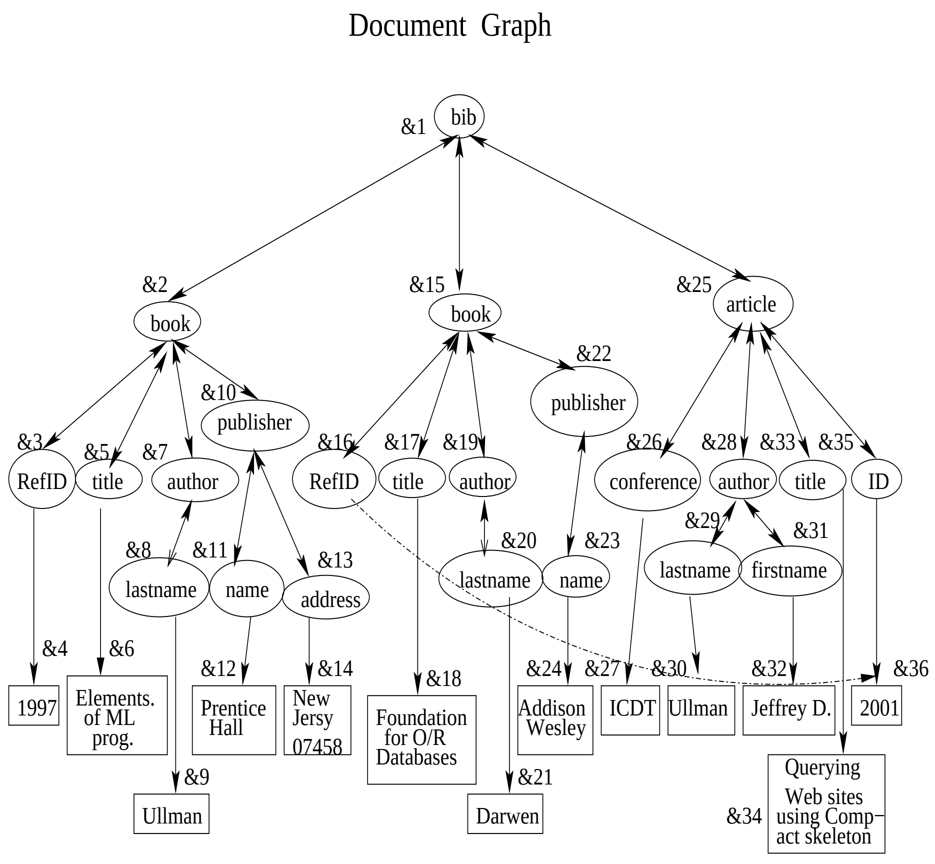


Figure 4: Example Document Graph

of which is shown in the dot-dash line emanating from node *&16* in Figure 4. The *order* of the elements in the original document, an important property in XML, is *preserved* in the Document Graph by organizing the sub-elements of each element in the graph in document occurrence order.

Another important feature of the Document Graph is that it is *bi-directional*, that is, every parent node stores all its child node numbers and vice versa. This permits easy movement through the graph for both top-down and bottom-up traversals.

Finally, note that the size of the Document Graph is a function of the size of the XML document since it explicitly represents every XML fragment appearing in the document.

2.1.1 Storage of Document Graph

To reduce the traversal times of the Document Graph, which in turn affect the query processing times, the graph is stored such that meta-data and data are separated from each other. Specifically, the nodes (elements/attributes) are clustered in one location, the children in another, and the actual leaf data in yet another location. Since each cluster is composed of fixed-size records, given a node number N (the unique identifier assigned by SphinX during the depth-first traversal), the associated page identifier and the location in the page are easily computed as follows:

$$Page\ Number = N\ /\ (PageSize/Recordsize)$$
$$Index\ in\ the\ Page = N\ \%\ (PageSize/Recordsize)$$

This approach significantly reduces the number of disk accesses since sequential access of the cluster is not necessary. Note that these formulas can continue to be used for in-place updates to element or attribute values, as well as for XML fragment appends and deletions – it is only when there are fragment inserts at random locations in the document does it fail – however, such operations are comparatively rare in practice. The complete update algorithm for the Document Graph and SphinX index structures is given later in Figure 11.

Note that in the situation wherein the backend is not a vanilla file system, but a native XML database engine (e.g. NatiX [29], Timber [30]) – in this case, the engine's built-in functionality for storage and traversal of graphs could be used for processing the Document Graph.

2.1.2 Handling IDs and IDREFs

Typically, the Document Graph will be a tree, but in general it can be a graph through the presence of IDs and IDREFs, which allow for nodes to refer to any other node in the tree. We now explain how IDs and IDREFs are processed while generating the Document Graph.

Our technique, drawing from standard compiler methods [2], is based on maintaining a hash-table of nodes constructed on their IDs. With this structure, any IDREF to a previously defined ID (i.e. a *backward* reference) can be immediately resolved by looking up the ID in the hash-table. However, *forward* references, where an IDREF appears prior to the associated ID cannot be handled in the same manner. For this situation, we create a list of IDREFs for each ID that has been referenced but not yet encountered. The list of IDREFs is "backpatched" when the ID is eventually encountered and inserted into the hash-table.

2.2 Schema Graph

The Schema Graph is a *rooted, node-labeled* graph, wherein each node represents either an element or an attribute that is present in the DTD, and the edges represent element nesting. The graph is constructed by parsing the DTD in a depth-first manner. It is also *bi-directional* to facilitate easy traversal. In the process of transformation from the DTD to this graph-equivalent, all element cardinality constraints such as ?, $+$, and $*$ that may be present in the DTD are ignored and the associated element is treated as equivalent to a single element. That is, a?, a+, and a*, are all replaced by a simple a. Further, the alternation operator | is replaced by the more general conjunction. That is, (a | b) is replaced by (a,b). These transformations guarantee that the Schema Graph is finite in size and can be constructed in time proportional to the number of elements present in the DTD. Note that while this is a relaxed (lossy) transformation, the relaxation is only in precision and not in *completeness* – that is, all documents adhering to the original DTD are also valid against the corresponding Schema Graph.[4]

Each leaf of the Schema Graph contains either a pointer to a B-tree or a *NULL* value. The B-tree pointers are present for those paths in the graph on which indexes have been built, whereas non-indexed paths end in *NULL* pointers. In Figure 3, for example, indexes have been built on all the paths in the DTD except /bib/book/author/firstname. As an optimization, it is possible to represent in the graph only those paths that

[4]This relaxation in precision with respect to the DTD is unrelated to the preciseness and completeness of the *indexing process* mentioned in the Introduction.

have had an index built on them, and dynamically update the graph whenever an index is built on a new path. However, since as mentioned earlier, the size of the Schema Graph is rather small, it appears cheaper to pre-construct the *entire graph* and terminate currently non-indexed paths with *NULL* pointers.

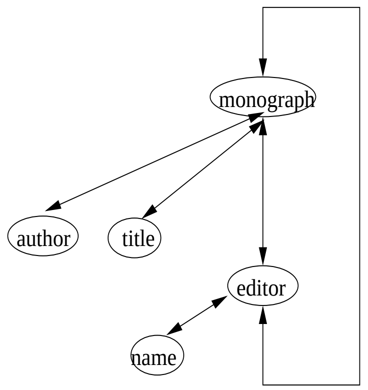
Finally, note that although the construction of the Schema Graph is superficially similar to that of the Document Graph, its size is related to the complexity of the DTD, *not* the XML document.

2.2.1 Handling Recursive DTDs

The Schema Graph will typically have a tree structure, as in our example. However, *recursion* in the DTD results in a *graph* structure with parent-child links going in both directions among a set of nodes. A recursive DTD fragment with mutually recursive editor and monograph elements [24] and the corresponding Schema Graph are shown in Figures 5a and 5b, respectively.

```
<!ELEMENT monograph ( title, author, editor)>
<!ELEMENT editor monograph>
<!ATTLIST editor name CDATA #REQUIRED>
```

(a)



(b)

Figure 5: Schema Graph for Recursive DTD

2.3 B-trees

Each B-tree is a path-specific index that is built on the atomic values appearing at the tail-end of the occurrences of that path in the Document Graph. The tree can be built on any path terminating in an attribute or PCDATA element. For example, the B-tree constructed on /bib/book/RefID path in the Schema Graph in Figure 3 is built on all the atomic year values (e.g. 1997, 2001) that appear as the values of the /bib/book/RefID path in the Document Graph. B-trees constructed over an element or attribute that occurs in a recursive loop have an additional attribute for every data value, specifying the level at which the data occurred in the Document Graph. For example, in Figure 5b, the B-tree that would be constructed over author would include this additional attribute.

Since DTDs represent the entire data in XML documents as character strings, all B-trees in SphinX are built using string-type keys, unless the data type is specified by the user. However, if instead of simple DTDs, we had *type-based* schemas such as the more recent XML Schema [36], then the B-trees could have type-specific keys. For example, the B-tree for /bib/book/ID would have integer keys, rather than string keys.

In our current implementation, the building of each B-tree index entails a separate scan of the XML document. In principle, however, it is possible to simultaneously construct multiple indexes in the same scan.

2.4 Choice of Indexes

Maintaining indexes on *every* path in the Schema Graph may turn out to be extremely expensive. In practice, only those paths that appear frequently in the query workload need to have indexes built on them. While there has been some recent work on *automated* XML index selection [22], we do not address this issue here and simply assume that suitable index choices have already been made by the database administrator.

3 Query Processing using SphinX

We now move on to discussing how query processing is implemented in SphinX. As mentioned earlier, queries on XML documents have at their core, *regular path expressions* (RPEs), which describe how the document must be traversed. For example, consider the following query Q_0 on the example document in Figure 4:

Q_0: /bib/article[@ID[text()= "2001"]]//author

The intention of query Q_0 is to list the names of all authors who contributed to the article with ID 2001. Note that the // indicates that this information should be retrieved no matter where the author element may be nested in the Document Graph. Further, note that this query relates *sibling* nodes (i.e. ID and author) in the Document Graph, which implies that answering the query could potentially require *backtracking* to the common parent (article) during the query processing.

The straightforward option to answer the above query is to perform a breadth-first traversal of the *entire* Document Graph and return results for all those paths satisfying the regular expression in the query. However, this complete traversal would be extremely expensive for non-trivial-sized XML documents, and therefore there is a need for indexes such as SphinX to speed up the traversal.

```
Query_Processing
Input : Regular Path Expression Query $Q$.
Output: Answer set corresponding to the RPE Query $Q$.
    // Extract the Precise paths from the $RPEs$ of query $Q$
1. For each of the $RPE[i]$ in $Q$
    // Obtain the paths from Schema Graph using Index_Path_Extraction
2.      ($Precise\_Path, B\text{-}treeRoot : B_R$) = Index_Path_Extraction ($RPE[i]$);
    // Obtain $Node\_Set[i]$ using Node_Identification
3. For each of the $Precise\_Paths$ with Predicates
    // RPE involving no Search Key takes a NULL value for Search Key
    // RecursiveLevel is the exact number of recursive unrollings in a recursive query
    // zero otherwise
4.      $Node\_Set[i]$ = Node_Identification ($B_R, SearchKey, PredicateQuery,$
                                              $RecursiveLevel$)
5. Operate Bottom-Up Query plan and evaluate the result.
```

Figure 6: Query Processing in Sphinx

3.1 Processing Steps for General RPEs

Before describing how the above query is handled in SphinX, we first outline the general mechanics of our technique. Given a set of general RPEs in the query, the high-level algorithm for processing the query is presented in Figure 6. Specifically, the query is processed in three distinct steps: *Index Path Extraction*, *Node Identification*, and *Document Graph Traversal*, described in the following subsections.

3.1.1 Index Path Extraction

The Index Path Extraction algorithm, given in Figures 7 and 8, converts the general RPEs present in the query into a set of simple paths (i.e., fully-specified paths), and identifies from the Schema Graph which of these paths have indexes. The output of the algorithm is the set of simple paths that appear in the RPEs as well as the pointers to the roots of the B-trees that are associated with each of these simple paths.

```
Index_Path_Extraction
Input: Schema Graph $SG$, RPE: $(S_1, E_1, S_2, E_2, \ldots S_N, E_N)$
    where S(Axes) = { //, / } and E = Elements including '*'s.
Output: Precise paths corresponding to the RPE, B-tree root pointers with each of the precise paths, if any.
1. Stack of structures consisting of nodes and their corresponding Paths:
        A. $ResultNodeSet = \{ 0 \}$ // Root node        B. $Node\_Set = \Phi$
    $Path = NULL$, and $Path\_Tmp = NULL$
    // Find the absolute path and nodes corresponding to $E_1$ in $SG$ if $S_1$ is a '//'
2. If ($S_1$ == '//') // If initial axes is '//' get all nodes corresponding to '//$E_1$'
3.      $Path, Initial\_Nodes$ = Find_Path $(ResultNodeSet, Path, (S_1, E_1))$
4. Else $Path, Initial\_Nodes = E_1, 0$
5. $Current\_Level = 1$
6. For each node in $Initial\_Nodes$ do
    // Find all paths by Find_Path using the initial nodes as starting pointers
7.      Find_Path $(Initial\_Nodes.Pop, Path, (S_2, E_2, S_3, \ldots S_N, E_N),$
            $Current\_Level)$
```

Figure 7: Algorithm Index Path Extraction

The conversion technique is the following: The RPE is evaluated against the Schema Graph utilizing a Breadth-First Search traversal, similar to a language evaluated against a Deterministic Finite Automaton. For RPEs ending with a PCDATA element or attribute, this algorithm returns the set of simple paths that appear in the RPEs with the PCDATA element or attribute as the leaf, along with the B-tree root pointer corresponding to that path. For RPEs ending with a non-PCDATA element, this algorithm returns the set of simple paths that appear in the RPEs with the nearest descendant PCDATA element of the non-PCDATA element assuming the role of the leaf, along with the B-tree root pointer corresponding to that path. If there are multiple nearest descendant PCDATA elements for a non-PCDATA element, the PCDATA element with the smallest height w.r.t. a leaf is returned. We discuss later in Section 3.3.4, as to why it is sufficient to consider just *one* of the PCDATA descendants for a non-predicate query.

```
**Find_Path** $(ResultNodeSet, Path, RPE(S_1, E_1, S_2, E_2, S_3, \ldots S_N, E_N), Level)$
1. **If** $(RPE == NULL)$
2.     $Path\_Count$++; **return**
3. **Else If** ($S_1$ == '/') // Child Axes
4.     $Node = ResultNodeSet.Pop$ , $Level$++
5.     **If** ($E_1$ == '*')
         // ResultNodeSet consists of all nodes under 'Node' as the element is '*'
6.         $Clean(Node\_Set[Path\_Count])$
7.         $Node\_Set[Path\_Count]$ = *All_Children* $(Node)$
         // Store the nodes and their appropriate paths in the ResultNodeSet where
         // Paths are obtained by concatenating the present absolute path with and the
         // relative path of the node obtained by *All_Children*
8.         $ResultNodeSet.Push()$ = $Node\_Set[Path\_Count].Nodes,$
                   $Node\_Set[Path\_Count].Path$ & Path
           // Find_Path is called recursively
9.         **Find_Path** $(ResultNodeSet, Path, (S_2, E_2, S_3, \ldots S_N, E_N), Level)$
10.    **Else** // If it is a specific element($E_1$), find $E_1$ child pointer of 'Node'
11.        $Clean(Node\_Set[Path\_Count])$
12.        $Node\_Set[Path\_Count]$ = *Find_Children* $(Node, E_1)$;
13.        $ResultNodeSet.Push()$ = $Node\_Set[Path\_Count].Nodes,$
                   $Node\_Set[Path\_Count].Path$ & Path
14.        **Find_Path** $(ResultNodeSet, Path, (S_2, E_2, S_3, \ldots S_N, E_N), Level)$
15. **Else If** ($S_1$ == '//') // Descendant Axes
16.    $Node = ResultNodeSet.Pop$ , $Level$++
17.    **If** ($E_1$ == '*')
18.        $Level = Current\_Level, Min\_Ht = 0$
           // if a '//' precedes '*' in the RPE, continue till all '*'s end
19.        **While** ($E_{Min\_Ht}$ != '*')
20.            *Skip* ($E_{Min\_Ht}$); $Min\_Ht$++; $Level$++
21.            **If** ( *Find_Level* $(E_{Level-Min\_Ht}, E_{Min\_Ht+1})$ == $Level$)
22.                $Clean(Node\_Set[Path\_Count])$
                   // Find all descendants which end at $E_{Min\_Ht+1}$ and start at 'Node'
23.                $Node\_Set[Path\_Count]$ = *Find_Descendants* $(Node, E_{Min\_Ht+1})$;
24.                $ResultNodeSet.Push()$ = $Node\_Set[Path\_Count].Nodes,$
                           $Node\_Set[Path\_Count].Path$ & Path
                   // Find_Path is called recursively from current set of nodes
25.                **Find_Path** $(ResultNodeSet, Path, (S_{Min\_Ht+1},$
                           $E_{Min\_Ht+1}, \ldots S_N, E_N), Level)$
26.            **Else return**
27.    **Else** // Find descendants of 'Node' ending with $E_1$
28.        $Clean(Node\_Set[Path\_Count])$
29.        $Node\_Set[Path\_Count]$ = *Find_Descendants* $(Node, E_1)$;
30.        $ResultNodeSet.Push()$ = $Node\_Set[Path\_Count].Nodes,$
                   $Node\_Set[Path\_Count].Path$ & Path
31.        **Find_Path** $(ResultNodeSet, Path, (S_2, E_2, S_3, \ldots S_N, E_N), Level)$
```

Figure 8: Algorithm Find_Path of Index Path Extraction

3.1.2 Node Identification

The Node Identifi cation algorithm, shown in Figure 9, takes as inputs the roots of the B-trees output from the previous step, as well as the associated search keys derived from the selection predicates of the user query. It searches the B-trees for these keys and, for the successful searches, outputs the associated pointers to nodes in the Document Graph. Note that the search key could be an exact-match or a range-match predicate. For non-predicate queries, as no search keys are involved, this algorithm returns one of the data pointers of the B-tree (corresponding to the root pointer previously returned by the Index Path Extraction algorithm).

```
Node_Identification
Input : B-tree root pointer $B_R$, Search Key $K$, Query Type $QT$, Recursive Level $RL$.
Output: Search Key associated pointers to nodes in the Document Graph, if $QT$ is a predicate
query. One of the data pointers of the B-tree, if $QT$ is a non-predicate query.
1. $NodeSet = \Phi$
     // If the query is a predicate query then the 'NodeSet' is just a B-tree search on 'K'
2. If ($QT$ == Predicate Query)
3.       $NodeSet$ = B-tree_Search ($B_R$, $K$, $RL$);
4. Else // Non-Predicate RPE. So returns only one data pointer of Document Graph
5.       $NodeSet$ = B-tree_Search ($B_R$, NULL, $RL$);
```

Figure 9: Algorithm Node Identifi cation

3.1.3 Document Graph Traversal

In the last stage of query processing, starting with the pointers to the relevant nodes in the Document Graph, the graph is traversed and the results are output. Note that since we have constructed the Document Graph to be bi-directional, the graph can be traversed equally easily bottom-up or top-down. In particular, we have two functions, *traverseDown* and *traverseUp*, which support these traversals.

3.2 Example of Index Access

We now illustrate the above indexing process for query Q_0 (/bib/article [@ID [text()= "2001"]]//author). The Index Path Extraction step is fi rst used to extract the precise paths from the query RPEs, which for Q_0 evaluates to /bib/article/ID for /bib/article/ID, and to /bib/article/author for /bib/article//author. Then, "joining" the resulting paths of the two regular path expressions it gets the relevant simple paths as /bib/article/ID for /bib/article/ID, and /bib/article/author for /bib/article
//author. The B-tree constructed over /bib/article/ID is then used in the Node Identifi cation step for predicate fi ltering, which results in only OID *&35*. Finally, the traversal to locate the fi nal results is done using the Document Graph Traversal step, starting from OID *&35*.

Consider a situation where there were a thousand entries for ID "2001" with regard to books, and only one with regard to articles. With a traditional schemaless indexing system, all thousand-and-one entries would have had to be followed. In marked contrast, SphinX by virtue of its Schema Graph, would follow only the sole productive entry, highlighting the utility of schema-consciousness.

For queries with *multiple* predicates, the most-selective predicate (i.e. the one estimated to have fewest result pointers from the B-tree) is selected and traversed to the common parent of all the predicates and the answer node. Note that making these estimations to choose the most selective predicate is the function of the Query Optimizer of the XML data management system, whereas our focus here is on the indexing module.

3.3 Complex Scenarios

The above example illustrated the SphinX approach for generic queries with general RPEs. We now explain how SphinX handles a variety of more complicated scenarios that arise in practice.

3.3.1 Recursive DTD Queries

We first explain how index path extraction is done for queries on documents conforming to *recursive* DTDs. Consider the query /monograph[//author[text() = ”Ullman”]]//title on the recursive DTD introduced in the previous section (see Figure 5a), which lists the titles of every monograph that has Ullman as an author. Here, /monograph//author is a general RPE matching a monograph node under the root node and then any descendant author element. The pattern is equivalent to the union of the following infinite sequence of patterns:

```
<monograph> <title> $t </>
          <author> Ullman </> </>
<monograph> <editor> <monograph> <title> $t </>
          <author> Ullman </> </> </> </>
  ...
```

The complication here is that when SphinX’s first step, Index Path Extraction, is executed for the general RPEs /monograph//author and /monograph//title, it is *not* possible to enumerate the simple paths since they are infinitely many. However, we can circumvent the problem by enumerating only the *first* “unrolling” of the recursion. The (pseudo) simple paths in this case are /monograph[editor/monograph]//author and /monograph[editor/monograph]//title, respectively. We also obtain the B-tree pointer for /monograph//author since it involves a key search predicate (“Ullman”). Then, in Step 2, Node Identification, the B-tree is searched to output pointers to all the leaf nodes in the Document Graph that contain “Ullman”. Finally, in the last step, Document Graph Traversal, we traverse the XML document graph using the node pointers obtained in Step 2, and search for the path /monograph[editor/monograph]//title.

A slightly modified strategy is followed for queries involving an *exact* number of recursive unrollings, as in the query /monograph/editor/monograph/author which is concerned only with authors at depth of two in the recursion. In this case, the additional *level* information which is present in the B-trees for recursive elements and attributes (refer Section 2.3) is utilized to filter out the correct results.

3.3.2 Queries with IDs and IDREFs

We next move on to describing how SphinX evaluates queries involving IDs and IDREFs. Since our Document Graph is constructed such that all IDREFs referencing an ID point to the physical disk pointer of the element containing ID, we simply traverse this physical pointer to evaluate the query. To make this concrete, assume that elements *person* and *article* have attributes *id* and *author* that are ID and IDREFs, respectively:

```
<!ATTLIST person id ID #REQUIRED>
```

```
<!ATTLIST article author IDREFs #REQUIRED>
```

and that the corresponding XML document has the following fragment:

```
<person id="007">
  <firstname>James</>
  <lastname>Bond</>
</person>
<person id="008">
  ...
</person>
<article author="007 008">
  ...
</article>
```

Now consider the query where the user wants to find the lastnames of all the authors of articles whose author attribute value is "007". This query first searches the B-tree to find all the pointers to author attributes in the Document Graph whose IDREF attribute value is "007". After finding the pointers to the author attributes in the Document Graph, we now traverse the Document Graph from person node using the physical pointer from author (as author is an IDREF, the value "007" will be pointing to the person with id "007"), returning the lastname.

3.3.3 Preserving XML Order

XML documents are, by definition, ordered and it is necessary for the indexing scheme to ensure that this order is reflected in the query processing as well. We now illustrate with the query /bib/article[5]/title how SphinX handles order semantics. This query lists the title of the fifth article appearing in the bibliography. Since there is no value predicate on title, SphinX follows one of the pointers of the B-tree, constructed over the path /bib/article/title, to the Document Graph. Recall, as mentioned in Section 2.1, that the graph is constructed such that the sub-elements of each element are organized in occurrence order. This means that the position of article will be stored in its parent. Therefore, traversing to the parent of article, which in this case is bib, allows us to find the node pointer of the fifth article among its children, which when followed results in title being output.

3.3.4 Handling Non-Predicate Queries

All the queries discussed so far have been predicate-based – we now move on to describing how SphinX evaluates non-predicate queries. Non-predicate queries can be classified into two categories: *Leaf*, where the query termination is on a leaf element, and *Internal*, where the query is on an internal node element.

The first category, Leaf, can be easily answered by merely scanning the B-tree constructed over the PCDATA element or attribute. For the second category, Internal, however, a different strategy has to be followed. To aid in the description of this strategy, we introduce ORACLE, an optimal, but practically infeasible, index structure, which needs to merely traverse the Document Graph to output the elements, as it "magically" knows the pointers to the elements in the Document Graph. Note that this is the least amount of work *any* indexing method will need to perform. To make this clear, consider the query /bib/book on the example document shown in Figure 4, whose goal is to return all the book elements. Since book is a internal element, we need to traverse in a depth-first manner to output all the descendants of book. ORACLE, magically knowing all the pointers to book (i.e. OIDs *&2* and *&15*) in the document graph, needs to just traverse all the descendants and output them in that fashion.

We now illustrate for the above query how SphinX evaluates internal non-predicate queries, and where it differs from ORACLE. SphinX uses a B-tree, built on one of the descendants of element /bib/book, say title. We follow one of the B-tree data pointers to the Document Graph and traverse bottom-up, storing all the elements and attributes along the path in a buffer. After reaching the book element, we traverse top-down along other descendant paths of book, such as /bib/book/author, /bib/book/publisher, etc. As explained earlier in Section 2.1, every parent node stores all the node pointers of its children and vice-versa. Therefore, when the first book element is reached, we can identify all its parent nodes, using which all other book nodes are traversed. Specifically, in the above example query, we identify the node number of bib, as it is the only parent. Thus, SphinX compared to ORACLE, incurs the extra effort of finding a B-tree data pointer for each of the B-trees returned by the Index Path Extraction step, and then traversing the distinct parents of the queried internal element.

The extra computation for preserving output order involves sorting, but duplicate elimination is not necessary as no node is traversed twice – the parents of the internal nodes of interest do not contain duplicates. We use a hash-table-based sorting scheme, wherein all the internal nodes are hashed. A list is made of the node identifiers obtained from the parents of the internal nodes and this list is then sorted. Finally, the list is processed in sequential order and the results are output by using the node identifiers to directly index into the hash-table.

For example, when processing the above-mentioned example query (/bib/book), the extra effort incurred is finding a B-tree data pointer constructed over title; traversing the distinct parents of book, which happens to be only bib; hashing all the book pointers; sorting the list of node numbers corresponding to book obtained from its parents (bib); and, finally, outputting the node contents in list order. Note that if order is not specified in the query, this hash-based sorting scheme becomes unnecessary.

3.3.5 Handling Queries with Leading Wildcards

Evaluating queries with leading wildcards requires SphinX to first generate simple path expressions for the general RPE from the SchemaGraph, and then to traverse the B-tree and Document Graph. For example, consider this query with leading wildcards: //author. When it is evaluated on the document shown in Figure 2, SphinX first goes through the SchemaGraph and generates simple path expressions for //author which evaluate to /bib/book/author and /bib/article/author. SphinX then traverses through one of the B-trees of each path, reaches the parents of the associated termination nodes, and then prints the authors in document order.

3.3.6 Handling Aggregate Queries

Aggregate queries return values such as Count, Max, Min, etc. SphinX evaluates aggregate queries directly using the B-trees if the aggregate function is applied on a leaf node. On the other hand, if the aggregate function is applied on an internal node, the query is evaluated similar to the way a normal internal-node query is evaluated, and then the aggregate function is applied on the result.

3.4 Comparison with Schemaless Approaches

We now illustrate how the presence of the DTD schema results in important indexing-related advantages by explaining how two representative schemaless approaches, namely, the classical Lore [16] and the more recent ToXin [21], would have processed some sample queries.

Lore makes use of structures called *dataguides* [13] to describe the "schema" of stored XML data. Dataguides are a concise and accurate summary of the path structure of a semi-structured database. For selection predicates, Lore uses a B-tree based *value index (Vindex)* [18] that takes a label, operator and value, and returns the set of OIDs of objects that satisfy the given value constraint and have the specified incoming label. Note that the index is based only on the *last* node label in a path to an object. Therefore, if a *Vindex* is built on ID in Figure 3, it constructs a B-tree that stores all the values of ID with their OIDs occurring on *all* paths. This means that for evaluating the Q_0 query mentioned at the beginning of this section, the *Vindex* of Lore will return the child of OID *&35* along the path /bib/article/ID. Also, from the Document Graph, the child of this ID object is referenced by the path /bib/book/RefID. Because the DataGuide algorithm constructs each target set, we can, using this feature, compute the intersection between the set of OIDs returned by the *Vindex* and the target set of /bib/article/ID, which results in the child of OID *&35*. Next, we examine all the objects returned by the intersection to identify the author objects whose article ID is "2001". Since the child of OID *&35* is also referenced by the path /bib/book/RefID, the cost of the indexing process is increased due to the expensive examination of this unproductive path.

The above problem persists even if we consider a more recent strategy such as ToXin[21]. For example, consider the following query which lists all the titles authored by "Ullman":

/bib[//author/lastname[text()= "Ullman"]]//conference

For this query, ToXin first goes through a "pre-selection" stage for computing the set of nodes reachable by paths matching /bib//author/lastname. In this case, ToXin follows all the paths over the index schema that match the RPE /bib//author/lastname, which for Figure 3 correspond to /bib/book/author/lastname and /bib/article/author/lastname. Then, it identifies the *instance tables* corresponding to the last node in each path – instance tables are 2-column tables that keep track, through a "key, foreign-key" association, of the parent-child relationship between nodes in the Document Graph. A union of the child columns of these tables is computed, which corresponds to all the lastname OIDs. The second "selection" stage performs the value selection of "Ullman" over all these OIDs by evaluating the value associated with each OID using the *value tables* – these are 2-column tables that store "OID, value" associations for the leaf nodes in the Document Graph. This will return the OIDS *&8* and *&29*. Then, in the last "post-selection" stage, the navigation up, and subsequently down, from these starting nodes to generate the conference names is carried out. Note that although the query processing procedures are different between Lore and Toxin, the net effect of pursuing unproductive paths is common to both the techniques.

3.5 Integration with Query Optimizer

As a final point, we now discuss how SphinX can be integrated with a query optimizer for efficient query evaluation. Consider a query of the form /A/B[text()="XML"]/C/D/E/F/G where it is required to find all the

G's that have a B ancestor containing the text "XML". If the query optimizer estimates that the number of data elements obtained by searching for "XML" in the B-tree over node B is large compared to the total number of data elements in the B-tree constructed over the leaf node nearest to G[5], then SphinX traverses the document graph using the latter entry-point rather than the former.

4 The SphinX System

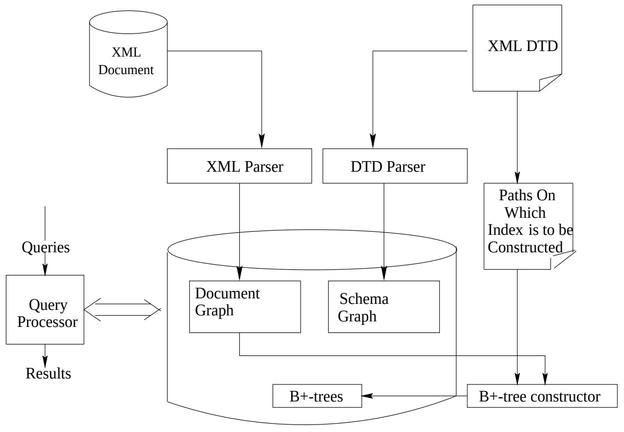


Figure 10: Architecture of SphinX

A pictorial version of the SphinX architecture is shown in Figure 10. Here, the *XML parser* parses the XML document and constructs the node-labeled Document Graph representing the data contained in the XML document. The *DTD parser* constructs the Schema Graph from the document DTD, and initializes all the leaf pointers to *NULL*. As and when it is decided to build an index on a given path, the *B-tree constructor* is invoked to scan the Document Graph and construct the index on the desired path. When the index construction is complete, the leaf pointer of the corresponding path in the Schema Graph is updated to point to the root of the newly-constructed B-tree.

We have implemented a prototype of SphinX, as per the above architecture, for indexing DTD-conforming XML documents. All programs, including the XML document parser to produce the Document Graph, the DTD parser to produce the Schema Graph, as well as the B-tree constructors, have been completely written in the C programming language, totally running to about 3000 lines of code.

As index maintenance involves updates as one of the key operations, we present an algorithm for updating SphinX index structures in Figure 11.

5 Performance Model

In this section, we describe the experimental setup used to profi le SphinX's performance. We evaluated SphinX on a representative set of real and synthetic XML documents, including one generated from Xmark, the popular XML benchmark [23], for a variety of RPE queries. As mentioned earlier, we are not aware of any other schema-conscious indexing strategies. Therefore, solely to put SphinX's performance into perspective, we have

[5]The node is G itself, if G is a leaf node.

```
Index_Update
Input : 1. Query proposing updation(deletion) or
        2. XML Fragment to be inserted
Output: Updated Sphinx Index Structures.
1. If Query is $Update(Deletion)$
      // Query Processing routine is called with (Query - Query Update portion) as input
      // which returns the nodes that needs to be updated
2.       $Update\_Nodes$ = Query_Processing (Query)
3.       Update the nodes of the Document Graph in $Update\_Nodes$ and store
            the old values in $Old\_Values$
4.       If $Update\_Path$ consists of a $Predicate\_Path$
5.             ($Precise\_Path, B\text{-}treeRoot : B_R$) = Index_Path_Extraction ($Update\_Path$)
            // Update the B-tree where all $Old\_Values$ will be replaced by $K2$
6.             $NodeSet$ = B-tree_Update ($B_R$, $Old\_Values$, $K2$, $RL$)
7. If Query is $Insert$
      // Insertion means the XML Fragment or document is adjoined to the present
      // set of document(s)
8.       Update the Document Graph.
            As the Document Graph is maintained to incorporate insertions, new buckets
            are created for inserting the parent and children pointers.
         // Update the B-tree information for the XML fragment
9.       For each $Path$ in the XML Fragment
10.            ($Path, B\text{-}treeRoot : B_R$) = Index_Path_Extraction ($Path$)
                // Insert all the new values in the B-tree
11.             B-tree_Insert ($B_R$, $New\_Values$, $RL$)
```

Figure 11: Updating Sphinx Index Structures

also evaluated the performance of a representative schemaless indexing technique, namely Lore, whose Version 5.0 executable is publicly available [27]. Also we have compared with DGScan (refer to Section 1), which represents the performance in the absence of any index, necessitating a scan of the Document Graph to answer any query.

Our experiments were performed on a generic P-III machine running Redhat Linux 7.1, configured with 512 MBytes of memory and 36 GB of local SCSI disk storage. The data and index are both stored on the disk. In the remainder of this section, we describe the XML documents and the queries used in our experiments.

5.1 XML Document Set

We present results here for six documents, four of which are real-world documents (Shakespeare [32], Conference [31], Journal [31], Ham-Radio, and the remaining two are synthetic documents generated from the Xmark benchmark [23]. The characteristics of these documents are summarized in Table 1, where the remaining column titles have the following meaning: The *Size* field refers to the total disk space occupied by the document; the *Records* field indicates the number of top-level records in the document; the *Depth* field indicates the maximum level of nesting; the *Elems* and *Attrs* fields indicate the number of elements and attributes in the DTD, respectively; the *TotalElems* and *TotalAttrs* fields indicate the number of element and attribute occurrences in the document, respectively; and the *Recursion* and *IDREFs* fields indicate whether the DTD and the associated document include this feature. The values in Table 1 show that the documents in the testbed cover a range of size and complexity characteristics.

Document	Type	Size(MB)	Records	Depth	Elems	Attrs	Total Elem	Total Attr	Recursion	IDREFs
Shakespeare	Real	7.4	35	6	22	0	179619	0	Yes	No
Conference	Real	40	104K	3	25	2	1029494	159928	No	No
Journal	Real	27	76K	3	15	2	804176	89567	No	No
HAM-RADIO	Real	361	700K	4	24	0	14117198	0	No	No
Xmark100MB	Synthetic	112	1	8	77	16	1666315	381878	Yes	Yes
Xmark1GB	Synthetic	1126	1	8	77	16	16703249	3829772	Yes	Yes

Table 1: **Document Statistics**

A brief description of the XML document contents, which arise from a variety of application domains, are given below:

Shakespeare: This document is the publicly available XML version of the plays of Shakespeare [32]. Many of the element values in this document are long textual passages; it also features recursion.

Conference, Journal: These documents represent the conference and journal entries, respectively, from the DBLP archive [31].

Ham-Radio: This document is obtained from the publicly available FCC Ham Radio database of the US Government's Federal Communications Commission. It has the highest percentage of meta-data content (approximately 70%) among the set of XML documents considered here.

Xmark100MB and Xmark1GB: These synthetic documents were generated from Xmark using its *xmlgen* data generator [23]. Xmlgen produces XML documents modeling an auction website, a typical e-commerce application. The documents are "broad and tall" – that is, the equivalent document tree features both large fanouts and deep nesting in the element structures (the Schema Graph for the Xmark DTD is shown in Appendix A). Many of the element values are long textual passages, similar to the Shakespeare document. Further, this document features both recursion and IDREFs.

5.2 XML Queries

We classify queries into *simple searches* and *general searches*, based on whether they involve simple or general RPEs. As mentioned earlier, in a simple search, the user fully specifi es the structure of the data that is to be searched (e.g. /bib/book/author/firstname), whereas in a general search, the structure of the data is only partially specifi ed (e.g. /bib//firstname).

We ran both simple search and general search queries on all the documents – these queries are given below (in the Xpath [8] format). The SQ queries refer to the simple search queries, whereas the GQ queries are the general search queries. Further, Query 1 is on Shakespeare, Queries 2 and 3 are on Conference, Queries 4 and 5 are on Journal, Query 6 is on Ham-Radio, Query 7 is on Xmark1GB, and Queries 8, 9, and 10, which are purely general search queries, are on Xmark100MB. The queries can also be classifi ed based on whether they are predicate queries or non-predicate queries: Queries 1, 2, 3, 8, and 9 represent predicate queries, while Queries 4, 5, 6, 7, and 10 are non-predicate queries.

The simple RPE queries used in our experiments are given below:

SQ_1: /Shakespeare/Play[Title[text() = ”The Tragedy of Antony and Cleopatra”]]/Personae/Persona

SQ_2: /conference/proceedings[year[text() = ”1998”]]/title

SQ_3: /conference/inproceedings[year[text() = ”1995”]]/booktitle

SQ_4: /journal/article

SQ_5: /journal/article/author

SQ_6: /FMDataBase/FCCAmRadio/Address

SQ_7: /site/open_auctions/open_auction/type

The general RPE queries used in our experiments are given below:

GQ_1: //[Title[text() = ”The Tragedy of Antony and Cleopatra”]]//Personae/Persona

GQ_2: //proceedings[year[text() = ”1998”]]/title

GQ_3: //[year[text() = ”1995”]]//booktitle

GQ_4: //article

GQ_5: /journal//author

GQ_6: //FCCAMRadio//Address

GQ_7: //open_auction/type

GQ_8: /site/people/person[/address/zipcode $>$ 4 and /address/zipcode $<$ 20]/country

GQ_9: /site/regions//item[1][quantity = 1]/location

GQ_{10}: /site/categories/category/description/parlist/listitem/parlist

5.3 Performance Metrics

We compare the indexing techniques on two classes of metrics: *construction metrics* and *query metrics*. The construction metrics, **Index Size** and **Index Creation Time** evaluate the space occupied and time taken to build by the index structure respectively. There is a single query metric, **Query Response Time**, which evaluates the total time for index access and result retrieval.

6 Experimental Results

We conducted a variety of experiments evaluating SphinX on the performance framework (documents, queries, and metrics) described in the previous section. The results of these experiments are discussed here. We first present the results for the index construction metrics, followed by the query metric.

Document	DGScan	Lore	SphinX
Shakespeare	15M	35M	16M
Conference	85M	226M	98M
Journal	63M	230M	70M
HAM-RADIO	834M	*	855M
Xmark100MB	190M	377M	193M
Xmark1GB	1900M	*	1904M

Table 2: **Index Sizes**

Document	DGScan	Lore	SphinX
Shakespeare	14s	41s	16s
Conference	94s	1433s	122s
Journal	73s	969s	95s
HAM-RADIO	1220s	*	1305s
Xmark100MB	135s	5760s	144s
Xmark1GB	1930s	*	2002s

Table 3: **Index Creation Time**

6.1 Index Construction Performance

The index size and creation times for SphinX and the benchmark algorithms (Lore and DGScan) are shown in Tables 2 and 3, respectively. The times and sizes for DGScan include only the Document Graph, as it works in the absence of any index. For SphinX, the sizes and times include construction of the Document Graph, Schema Graph, and B-trees on elements and attributes on which queries are evaluated. The sizes and times mentioned for Lore include the Lindex, Pindex(Data Guide), Vindex, and Tindex indexes. The Vindex and Tindex are created on only elements and attributes on which queries are evaluated.

The reason the index sizes show a signifi cant increase from the original document sizes is because the entire XML document is represented in the Document Graph. Secondly, by constructing the Document Graph and Schema Graph effi ciently, SphinX improves upon Lore by 2 to 4 times with respect to space and about an order of magnitude with respect to time. Further, as the document size increases, the performance gap between SphinX and Lore increases, and in fact, for the two largest documents, Ham-Radio and Xmark, the Lore system failed to complete, signifi ed by the "*" entry. Finally, in absolute terms, the SphinX index creation time appears to be satisfactory in that the 1.1 GB benchmark document is processed in about half an hour, translating to a processing speed of close to 1 MB per second (recall that these numbers are obtained with vanilla hardware).

6.1.1 Construction Space and Time Estimators

We now give a simple analytical formula to estimate the size and construction time of the SphinX index, given an arbitrary XML document whose characteristics are described by the following parameters:

N_e	:	Number of elements in the XML document
N_a	:	Number of attributes in the XML document
N_f	:	Number of PCDATA elements in the XML document
F	:	Average fanout of an element in the XML document
K_1	:	Size of a node(element or attribute) in the Document Graph
K_2	:	Size of an edge(element or attribute pointer) in the Document Graph
K_3	:	Average size of PCDATA in the Document Graph

In terms of the above parameters, the total index size is given by

$$IndexSize = (N_e + N_a)\, K_1 + (N_e * F * K_2) + (N_f * K_3)$$

where the fi rst term in the equation represents the space taken by all the nodes (elements and attributes) in the Document Graph, the second term represents the space consumed by the edges of the Document Graph, and the third term represents the space consumed by the leaf nodes of the Document Graph.

Moving on to the cost of constructing the Document Graph, the number of disk accesses taken when constructing the Document Graph can be estimated as follows:

$$IndexCost = (N_e * F + N_a + N_f) \ / \ PageSize$$

where the second and third terms represent the number of disk accesses for all the attributes and PCDATA elements in the document, which need to be written only once. As the parent needs to be modified for all its children, the number of disk accesses potentially increases by a factor of F for all elements. Empirical evaluation on the above formulae for index size and index cost shows that they approximately hold for the datasets considered.

Document	**Query#**	**Selectivity**	**DGScan**	**Lore**		**SphinX**	
				CPU	Disk	CPU	Disk
Shakespeare	SQ_1	29/1031 = 0.03	4.1s	3.0s	0.3s	0.6s	0.1s
Conference	SQ_2	20/425 = 0.05	216.8s	*	*	0.4s	0.1s
Conference	SQ_3	738/104186 = 0.007	258.9s	*	*	5.1s	1.7s
Journal	SQ_4	76095/76095 = 1.0	412.2s	98.6s	73.8s	30.8s	20s
Journal	SQ_5	145036/145036 = 1.0	381.5s	45.0s	75.2s	2.9s	5.3s
HAM-RADIO	SQ_6	705008/705008 = 1.0	932.8s	*	*	71.8s	121s
Xmark1GB	SQ_7	217500/217500 = 1.0	724.4s	*	*	8.1s	12.1s

Table 4: Query Performance for Simple RPEs

Document	**Query#**	**Selectivity**	**DGScan**	**Lore**		**SphinX**	
				CPU	Disk	CPU	Disk
Shakespeare	GQ_1	29/1031 = 0.03	7.0s	3.4s	2.0s	0.6s	0.2s
Conference	GQ_2	20/425 = 0.05	243.4s	*	*	0.7s	0.4s
Conference	GQ_3	738/104186 = 0.007	262.3s	*	*	5.3s	2.5s
Journal	GQ_4	76095/76095 = 1.0	483.2s	100.2s	77.8s	32.1s	20s
Journal	GQ_5	145036/145036 = 1.0	458.5s	47.3s	78.3s	3.2s	5.5s
HAM-RADIO	GQ_6	705008/705008 = 1.0	1583.9s	*	*	83s	121.8
Xmark1GB	GQ_7	217500/217500 = 1.0	2812.1s	*	*	8.5s	14.4s
Xmark100MB	GQ_8	6617/12716 = 0.52	437.8s	8s	5.3s	2.6s	2.2s
Xmark100MB	GQ_9	21370/21751 = 0.98	425.6s	25.2s	14.0s	0.8s	0.4s
Xmark100MB	GQ_{10}	17/17 = 1.0	412.3s	5.2s	1.6s	0.4s	0.1s

Table 5: Query Performance for General RPEs

6.2 Query Performance

Moving on to the response time metric, the performance results of SphinX for the simple and general query sets are shown in Tables 4 and 5, respectively. The times are broken up into CPU and disk components. Further, the query selectivities, computed as *(Number of elements in the result set / Total number of such elements in the Document),* are also shown in the tables.

As mentioned in the Introduction, the semantics we associate with XPath queries is that the actual results, and not just the node-ids, need to be provided to the user. Therefore, the query run times shown here include the time for *displaying* the results on the standard output – this is the reason that the absolute performance numbers are higher than might be initially expected for such query expressions. Note that GQ_8, GQ_9, and GQ_{10} were run on the Xmark100MB dataset instead of Xmark1GB, as Lore failed to build the index for Xmark1GB.

We see in these tables that firstly, DGScan performs extremely poorly, since it has no index and needs to traverse the document in a depth-first manner to answer the query. DGScan for Simple RPEs can prune some of the paths from the Document Graph because the information regarding the level and parent is available for every element or attribute in the RPE. For example, for query SQ_1, it is not necessary for DGScan to traverse children of Shakespeare other than Play and also not necessary to search for Play at levels other than two in the Document Graph. However, for General RPEs, as wildcards are involved, DGScan cannot prune many of the unproductive paths due to level and parent information not being available for most of the elements and attributes in the RPE.

In contrast, while the schemaless Lore does improve on DGScan, as shown in Tables 4 and 5, the margins of improvement are not as large as those of SphinX. Although Lore's query evaluation for the queries listed in Section 5.2 is not mentioned explicitly, the performance gap can be attributed to some of the following reasons: Lore uses the *Vindex* or *Tindex* for predicate queries to find all OIDs that satisfy the value predicate in the query. After finding the OIDs of all the atomic objects it traverses the Document Graph to evaluate the query which may include unproductive paths. For non-predicate queries, Lore uses the *Pindex*, which for a path p returns the set of objects O reachable via p. The set of paths that are indexed by *Pindex* should begin at named objects and "contain no regular expressions" and it cannot be used for all queries and path expressions [18]. Lore uses another physical query operator, called *Scan*, for non-predicate queries, whose approach is similar to *pointer-chasing* in object-oriented systems [17].

Moving on to SphinX, we see that it offers a substantial improvement over DGScan and Lore, typically by an order of magnitude on DGScan. The reason for the improved performance is directly due to SphinX's schema-consciousness, which allows it to ensure that only productive paths are followed. That is, SphinX first finds all the productive paths using the Schema Graph, and then uses the B-trees to find all the pointers to the nodes in the Document Graph that satisfy the predicate. As shown in Tables 4 and 5, the performance gain for predicate and non-predicate queries is due to the efficient pruning of paths. For non-predicate queries, as explained in Section 3.3.4, the overhead compared to ORACLE, the impractical optimal algorithm, is marginal, thus resulting in good performance gain over schemaless approaches.

Overall, the above results show how SphinX effectively utilizes the schema knowledge to achieve very significant gains in query processing times.

7 Related Work

As mentioned in the Introduction, a variety of XML indexing techniques have been proposed in the literature over the last few years. These include Lore [16], T-index [19], ToXin [21], XISS [15], IndexFabric [9], APEX [20], Forward-and-Backward-Index [14], Holistic Twig Join [6], Barashev and Novikov [3], and ViST [25]. In this section, we briefly overview this previous work.

The earliest system that we are aware of is Lore [16], which uses structures called *dataguides* to describe the "schema" of the stored XML data. Dataguides are a concise and accurate summary of the path structure of a semi-structured database. In Lore, there are no less than four indexes: one for paths, one for values, one for strings, and one for backward traversal through the Document Graph.

Subsequently, the T-index was proposed in [19] for indexing a specific set of path expressions. The most basic type of T-index, called a 1-index, attempts to describe all paths along the data graph using a finite state automaton, where each state includes a number of elements, and each extent contains all the data graph nodes in a particular equivalence class – an equivalence class is a set of nodes that are reachable via the same specific set of paths.

ToXin, an in-memory index that supports both backward and forward navigation of the XML graph to answer regular path queries was proposed in [21]. ToXin consists of two different types of index structures: the *value index* and the *path index*. The path index is composed of the index tree, which is a minimal dataguide, and a set of instance functions, one for each edge in the index tree. Each instance function keeps track of the parent-child relationship between the pair of nodes that defines each XML element. Another index, called XISS, based on a numbering scheme for elements was proposed in [15]. Multiple indexes including an *element_index*, *attribute_index*, *structure_index*, as well as *name_indexes* and *value_tables* have been used in this approach. The proposed join algorithms can process regular path expression queries without traversing the hierarchy of XML data.

Index Fabric, proposed in [9], provides efficiency and flexibility using Patricia tries. This indexing mechanism utilizes the aggressive key compression inherent in Patricia tries to index a large number of strings in a compact and efficient structure. This work considers two paths: Raw paths and Refined paths. Raw paths are conceptually similar to dataguides [13] whereas refined paths can support queries that have wild cards, alternates, and different constants. The APEX index [20] retains only frequently used paths to speed up query processing. APEX also has the desirable property that it can be updated incrementally to match the changes in query workloads. The Forward and Backward-index (F&B-Index), proposed in [14], can be viewed as a covering index for branching path expression queries. It is also shown that for a large and natural class of indexes, it is the smallest index that can cover all branching path expression queries.

A Holistic Twig Join approach is proposed for matching XML twig patterns in [6]. Here, a chain of linked stacks are used to compactly represent partial results for root-to-leaf query paths, which are then composed to obtain matches for the twig pattern. Holistic twig join algorithms run the risk of having to process many false positives. Barashev and Novikov [3] proposed new index structures suitable to support path expressions. The indexing structures considered in their work are B-trees and Patricia tries. Finally, ViST [25], which takes a similar approach to SphinX, provides a unified index on both content and structure of the XML documents. It uses tree structures as the basic unit of query execution in order to avoid expensive join operations and relies solely on B+-tree indexes.

The paramount difference between our work and this prior art is, of course, that we consider *schema-based* XML indexing in general, and DTD-based indexing in particular. In contrast, the prior art is applicable to documents that come in an "as is" condition, without any auxiliary schema-related meta-data. Another important distinction is that while specialized index structures (e.g. instance functions [21] and multi-layered Patricia tries [9]), form the basis of some of these techniques, SphinX only uses the standard B-trees, which represent a mature and scalable technology. Moreover, SphinX has been evaluated on documents generated from Xmark [23] which, as mentioned earlier, produces complex "broad and tall" documents. Finally, while some of the techniques do not provide persistent disk-resident indexes, others fail to fully support functionalities such as range queries, value predicate queries, recursion, IDREFS and preservation of XML order, which are essential requirements in practice.

We now illustrate the advantages of using "pre-defined schemas", such as DTDs, as compared to "post-facto" schemas derived from the data, such as DataGuides. The structures extracted from data can make the index access slower and run into problems like the following: (a) In Lore, creating a DataGuide over a database is equivalent to conversion of an NFA to a DFA. In the worst case, conversion of a graph-structured database may require time and space exponential in the number of objects and edges in the database. This has been verified in our experimental results, where we were not able to create indexes for some of the datasets; (b) IndexFabric is not efficient for regular path expressions which involve wild cards. Further, since Patricia trees are lossy in their compression, it is always necessary to go through the base data to eliminate false positives; (c) XISS is unable to preserve order of the XML document; (d) The Barashev and Novikov approach is likely to grow large, as the structure of the document grows, similar to Lore.

Overall, SphinX's contribution lies mainly in the query processing strategy; in particular, it applies even if the schema graph is obtained from a different source, as opposed to a DTD. For example, if we have a DataGuide, the leaves of the DataGuide can be made to point to B-trees constructed on the leaf nodes and then SphinX's query processing strategy can be used to process queries. Note that such portability is not straightforward for index structures that, unlike SphinX, do not differentiate between structure and content.

Finally, the recently proposed XIST index selection tool [22] can be used to identify the set of paths on which SphinX indexes should be constructed.

8 Conclusions and Future Work

We have developed SphinX, a new persistent mechanism for indexing XML documents. Our approach exploits, for the first time to our knowledge, schema over the XML document to efficiently process regular path expression queries. We utilize the schema knowledge to convert general RPEs into an equivalent set of simple RPEs, and thereby ensure that only productive paths are followed in the Document Graph. This is in contrast to previous techniques which, due to being schema-indifferent, could run the risk of following unproductive paths. Further, our use of standard B-trees for indexing from the Schema Graph to the Document Graph ensures that the implementation is robust and that the system is scalable. Our approach supports a host of advanced features such as IDREFS, recursion, order maintenance, and range queries, many of which have not been addressed by previous XML indexing schemes, but are important features in practice. Finally, while we have focused on DTDs as the schema source in this paper, the SphinX technique could be extended to alternative schema sources as well.

Our experimental results, over a variety of XML documents and RPE queries, show that SphinX performs significantly better compared to schemaless approaches such as DGScan and Lore with respect to both index construction as well as index access. The performance improvements range from a few times to an order of magnitude, and generally increase with increasing document size.

In closing, we have attempted to describe in this paper an "industrial-strength" schema-conscious XML index with good performance characteristics.

Acknowledgements

This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener, “The Lorel query language for semistructured data”, *Intl. Journal on Digital Libraries*, 1(1), April 1997.

[2] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] D. Barashev and B. Novikov, “Indexing XML to support Path Expressions”, *Proc. of ADBIS Conf.*, September 2002.

[4] P. Bohannon, J. Freire, P. Roy and J. Simeon, “From XML schema to relations: A cost-based approach to XML storage”, *Proc. of 18th IEEE Intl. Conf. on Data Engineering*, June 2002.

[5] T. Bray, J. Paoli and C. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0”, February 1998, *http://www.w3.org/TR/1998/REC-xml-19980219*.

[6] N. Bruno, N. Koudas and D. Srivastava, “Holistic Twig Joins: Optimal XML Pattern Matching”, *Proc. of ACM SIGMOD Conf.*, June 2002.

[7] D. Chamberlin, D. Florescu, J. Robie, J. Simon and M. Stefanescu, “XQuery 1.0: An XML Query Language”, *W*3C Working Draft, World Wide Web Consortium, September 2005, *http://www.w3.org/TR/xquery/*.

[8] J. Clark and S. DeRose, “XML Path Language (XPath) Version 1.0”, *W3C Recommendation*, World Wide Web Consortium, November 1999, *http://www.w3.org/TR/xpath*.

[9] B. Cooper, N. Sample, M. Franklin, G. Hjaltason and M. Shadmon, “A Fast Index for Semistructured Data”, *Proc. of 27th Intl. Conf. on Very Large Data Bases*, August 2001.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, “A Query Language for XML”, *Proc. of 8th Intl. World Wide Web Conf.*, May 1999.

[11] M. Fernandez and D. Suciu, “Optimizing Regular Path Expressions Using Graph Schemas”, *Proc. of 14th IEEE Intl. Conf. on Data Engineering*, February 1998.

[12] J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Simeon, “StatiX: Making XML Count”, *Proc. of ACM SIGMOD Conf.*, June 2002.

[13] R. Goldman and J. Widom, “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases”, *Proc. of 23rd Intl. Conf. on Very Large Data Bases*, August 1997.

[14] R. Kaushik, P. Bohannon, J. Naughton and H. Korth, “Covering Indexes for Branching Path Queries”, *Proc. of ACM SIGMOD Conf.*, June 2002.

[15] Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions”, *Proc. of 27th Intl. Conf. on Very Large Data Bases*, August 2001.

[16] J. McHugh, S. Abiteboul, R. Goldman, D. Quass and J. Widom, “Lore: A Database Management System for Semistructured Data”, *ACM SIGMOD Record*, September 1997.

[17] J. McHugh and J. Widom, “Query Optimization for XML”, *Proc. of 25th Intl. Conf. on Very Large Data Bases*, September 1999.

[18] J. McHugh, J. Widom, S. Abiteboul, Q. Luo and A. Rajaraman, “Indexing Semistructured Data”, *ftp://db.stanford.edu/pub/papers/semiindexing98.ps*.

[19] T. Milo and D. Suciu, “Index Structures for Path Expressions”, *Proc. of Intl. Conf. on Database Theory*, January 1999.

[20] J. Min, C. Chung and K. Shim, “APEX: An Adaptive Path Index for XML data”, *Proc. of ACM SIGMOD Conf.*, June 2002.

[21] F. Rizzolo, “ToXin: An Indexing Scheme for XML Data”, *MSc Thesis*, Dept. of Computer Science, Univ. of Toronto, Canada, January 2001.

[22] K. Runapongsa, J. Patel, R. Bordawekar, and S. Padmanabhan, “XIST: An XML Index Selection Tool”, *Proc. of 2nd XML Database Symposium*, August 2004.

[23] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey and R.Busse, “The XML Benchmark Project”, April 2001, *http://monetdb.cwi.nl/xml/Benchmark/benchmark.html*.

[24] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt and J. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities”, *Proc. of 25th Intl. Conf. on Very Large Data Bases*, September 1999.

[25] H. Wang, S. Park, W. Fan and P. Yu, “ViST: A Dynamic Index Method for Querying XML Data by Tree Structures”, *Proc. of ACM SIGMOD Conf.*, June 2003.

[26] P. Wood, “Minimizing Simple XPath Expressions”, *Proc. of 4th Intl. Workshop on the Web and Databases*, May 2001.

[27] *http://www-db.stanford.edu/lore/release*

[28] *http://www.bioml.com*

[29] *http://www.dataexmachina.de/natix.html*

[30] *http://www.eecs.umich.edu/db/timber/*

[31] *http://www.informatik.uni-trier.de/∼ley/db*

[32] *http://www.oasis-open.org/cover/bosakShakespeare200.html*

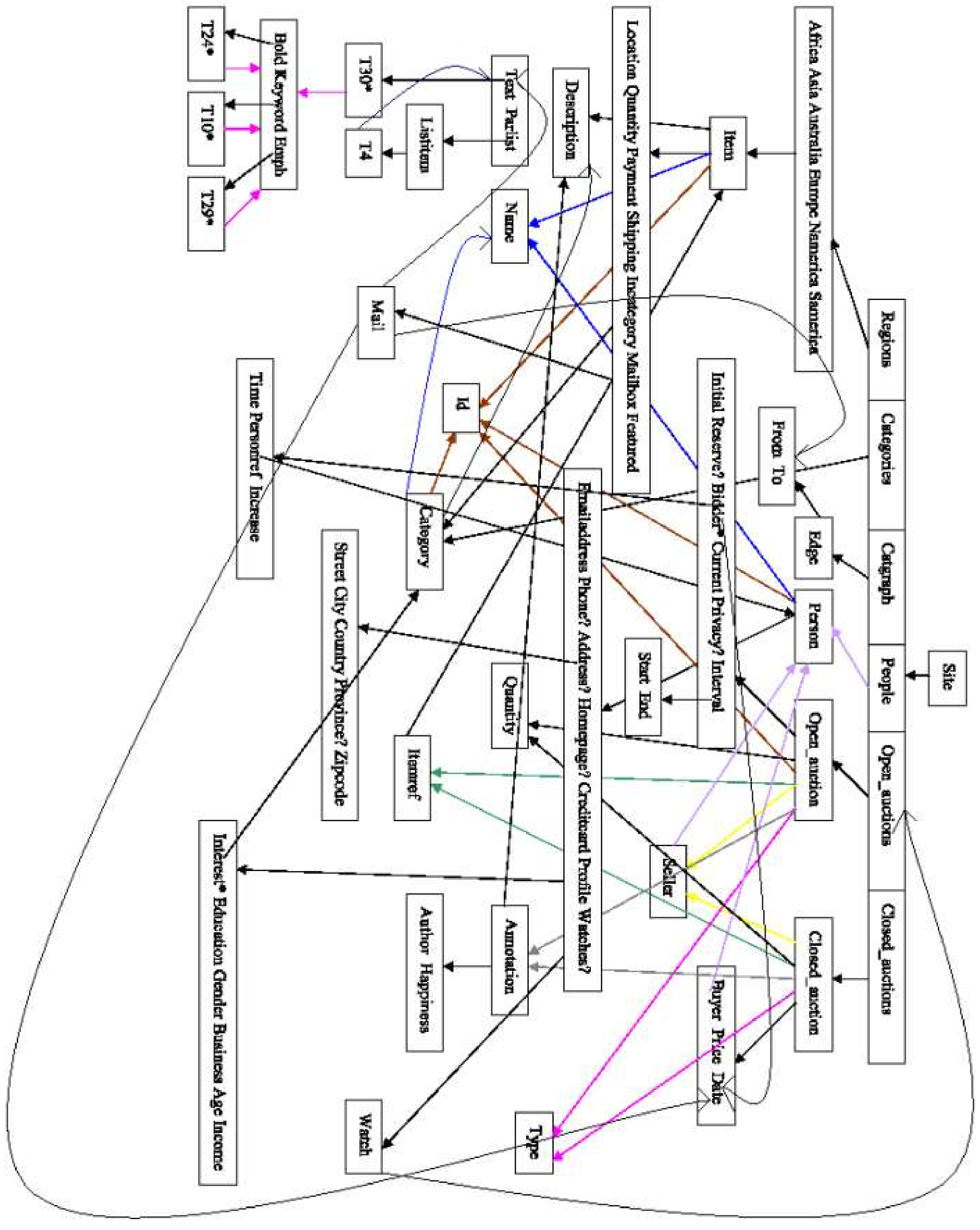
[33] *http://www1.softwareag.com/corporate/products/tamino/default.asp*

[34] *http://www.w3.org/Math*

[35] *http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm*

[36] *http://www.w3.org/XML/Schema*

A Xmark DTD


Site
Regions
Categories
Catgraph
People
Open_auctions
Closed_auctions
Africa Asia Australia Europe Namerica Samerica
Item
Location Quantity Payment Shipping Incategory Mailbox Featured
Description
Text Parlist
Listitem
T30*
T4
Bold Keyword Emph
T24*
T10*
T29*
Name
Mail
Id
From To
Edge
Person
Initial Reserve? Bidder* Current Privacy? Interval
Time Personref Increase
Category
Emailaddress Phone? Address? Homepage? Creditcard Profile Watches?
Street City Country Province? Zipcode
Start End
Quantity
Itemref
Open_auction
Seller
Closed_auction
Interest* Education Gender Business Age Income
Author Happiness
Annotation
Buyer Price Date
Watch
Type