Schema-conscious XML indexing

Krishna P. Leela¹, Jayant R. Haritsa*

Database Systems Lab, SERC/CSA, Indian Institute of Science, Bangalore 560012, India

Received 4 May 2003; received in revised form 31 October 2005; accepted 31 October 2005

Recommended by Y. Ioannidis

Abstract

User queries on extensible markup language (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 document type descriptor (DTD). In this paper, we propose schema-conscious path-hierarchy indexing of XML (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.

© 2005 Elsevier B.V. All rights reserved.

Keywords: XML; Index; DTD; B-tree

1. Introduction

Extensible markup language (XML) [1], 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 languages have been proposed in the literature, including Lorel [2], XML-QL [3], XPath [4], etc. from which XQuery [5] 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.²

In order to process RPE queries efficiently, a variety of XML indexing techniques have been designed. These techniques can be broadly classified into two categories: content-based and schema-based. Content-based techniques, such as XMark [6], focus on optimizing the retrieval of specific data content without considering the underlying schema. In contrast, schema-based techniques, such as SphinX, leverage the schema information to enhance query processing. In this paper, we propose SphinX, a schema-conscious path-hierarchy indexing scheme that seamlessly combines the schema information with standard B-tree technology, resulting in a simple and scalable solution.

1Currently with Yahoo! India.
2The 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 subtrees) associated with the result nodes.
proposed in the literature. These techniques include the classical Lore [6] and T-index [7], as well as more recent followups such as ToXin [8], XISS [9], Index Fabric [10], APEX [11], F&B-Index [12], Holistic Twig Joins [13], Barashev and Novikov [14] and ViST [15]. 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) [16]. Such situations are quite common in practice—for example, BioML [17] and MathML [18] 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 [19,20], for query pruning [21] and minimization [22], for gathering document statistics [23], etc. However, using schema information to enhance indexing efficiency has not been considered before to the best of our knowledge.3

1.1. 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 [25]), including those derived post-facto from the data (e.g. Data Guide [6]), 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, 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 Fig. 1.

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, the DTD for Xmark, the popular XML benchmark [26], 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

---

3Industrial products such as Tamino [24] and eXcelon [37] 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).

4When XML documents are stored on RDBMS backends (e.g. [19,20]), the standard indexing mechanisms of relational engines have to be used.
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” [8], and “multi-layered Patricia tries” [10], 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.

1.2. 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 [26]. 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 DGS

1.3. 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.

1.4. 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.

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 multiDTD/multi-document scenario is straightforward.
An example DTD of a bibliographic database (on the lines of DBLP [27], the popular computer science bibliography), and an XML document fragment that conforms to this DTD, are shown in Figs. 2a and b, respectively. The corresponding schema graph with leaf-level B-trees and the associated document graph, are shown in Figs. 3 and 4, respectively (in Fig. 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 of which is shown in the dot-dash line emanating from node &l6 in Fig. 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

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

Fig. 2. Example DTD and XML fragment.
Fig. 3. Example SphinX index.

Fig. 4. Example document graph.
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:

\[
\text{Page number} = N/(\text{pagesize}/\text{recordsize}),
\]

\[
\text{Index in the page} = N\%/(\text{pagesize}/\text{recordsize}).
\]

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 Fig. 11.

Note that in the situation wherein the backend is not a vanilla file system, but a native XML database engine (e.g. NatiX [28], Timber [29])—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 [30], 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, \( (a \mid b) \) 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.\(^5\)

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 Fig. 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 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

\(^5\)This relaxation in precision with respect to the DTD is unrelated to the preciseness and completeness of the indexing process mentioned in the Introduction.
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 [19] and the corresponding schema graph are shown in Figs. 5a and b, respectively.

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 Fig. 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 Fig. 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 [25], 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 [31], 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, RPEs, which describe how the document must be traversed. For example, consider the following query $Q_0$ on the example document in Fig. 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.

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 Fig. 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 Figs. 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
Find Path (ResultNodeSet, Path, RPE(S₁, E₁, S₂, E₂, S₃, ..., Sₙ, Eₙ), Level)
1. If (RPE == NULL)
   Path.Count++; return
3. Else If (S₁ == ‘?’) // Child Axes
   Node = ResultNodeSet.Pop, Level++
5. If (E₁ == ‘*’)
   // ResultNodeSet consists of all nodes under ‘Node’ as the element is ‘*’
   Clean(Node.Set[Path.Count])
   // 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
   Node.Set[Path.Count].Path & Path
   // Find_Path is called recursively
9. Find Path (ResultNodeSet, Path, (S₂, E₂, S₃, ..., Sₙ, Eₙ), Level)
10. Else If it is a specific element(E₁), find E₁ child pointer of ‘Node’
11. Clean(Node.Set[Path.Count])
   Node.Set[Path.Count].Path & Path
14. Find Path (ResultNodeSet, Path, (S₂, E₂, S₃, ..., Sₙ, Eₙ), Level)
15. Else If (S₁ == ‘?’) // Descendant Axes
17. If (E₁ == ‘*’)
18. Level = Current.Level, Min.Ht = 0
   // if a ‘?’ precedes ‘*’ in the RPE, continue till all ‘*’s end
19. While (E₁ == ‘*’)
20. Skip (E₁.Min.Ht); Min.Ht++; Level++
21. If (Find Level (E₁LEVEL-MIN.HT, E₁.MIN.HT+1) == Level)
22. Clean(Node.Set[Path.Count])
   // Find all descendants which end at E₁.MIN.HT+1 and start at ‘Node’
   Node.Set[Path.Count].Path & Path
   // Find.Path is called recursively from current set of nodes
25. Find Path (ResultNodeSet, Path, (S₁.MIN.HT+1, E₁.MIN.HT+1, ..., Sₙ, Eₙ), Level)
26. Else return
27. Else // Find descendants of ‘Node’ ending with E₁
28. Clean(Node.Set[Path.Count])
   Node.Set[Path.Count].Path & Path
31. Find Path (ResultNodeSet, Path, (S₂, E₂, S₃, ..., Sₙ, Eₙ), Level)

Fig. 8. Algorithm find_path of index path extraction.

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.

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.

3.1.2. Node identification

The node identification algorithm, shown in Fig. 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).

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 first 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 identification step for predicate filtering, which results in only OID &435. Finally, the traversal to locate the final 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 in-
troduced in the previous section (see Fig. 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:

\[
\text{<monograph> <title> $t$ </title>}
\]
\[
\text{<author> Ullman </author> <title> $t$ </title>}
\]
\[
\text{<monograph> <editor>}
\]
\[
\text{<author> $t$ </author> <title> $t$ </title>}
\]
\[
\text{...}
\]

The complication here is that when SphinX’s first
step, index path extraction, is executed for the
general RPEs /monograph[/author and /mono-
graph[/title, it is not possible to enumerate the
simple paths since they are infinitely many. How-
ever, we can circumvent the problem by enumerat-
ing 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:

\[
\text{<!ATTLIST person id ID #REQUIRED>}
\]
\[
\text{<!ATTLIST article author IDREF #REQUIRED>}
\]

and that the corresponding XML document has the
following fragment:

\[
\text{<person id='007'>}
\]
\[
\text{<firstname> James </firstname>}
\]
\[
\text{<lastname> Bond </lastname>}
\]
\[
\text{</person>}
\]
\[
\text{<person id='008'>}
\]
\[
\text{...}
\]
\[
\text{</person>}
\]
\[
\text{<article author ='007 008'>}
\]
\[
\text{...}
\]
\[
\text{</article>}
\]

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
the 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 titles 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 \(/\text{bib/book}\) on the example document shown in Fig. 4, whose goal is to return all the book elements. Since book is an 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 12 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 \(/\text{bib/book}\), 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 \(/\text{bib/book/author}\), \(/\text{bib/book/publisher}\), 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 \(/\text{bib/book}\), 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 schema graph, and then to traverse the B-tree and document graph. For example, consider this query with leading wildcards: \(/\text{author}\). When it is evaluated on the document shown in Fig. 2, SphinX first goes through the schema graph and generates simple path expressions for \(/\text{author}\) which evaluate to \(/\text{bib/book/author}\) and \(/\text{bib/article/author}\). 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 [6] and the more recent ToXin [8], would have processed some sample queries.

Lore makes use of structures called dataguides [32] 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) [33] 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 Fig. 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 [8]. 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 Fig. 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,\(^6\) then SphinX traverses the document graph using the latter entry-point rather than the former.

4. The SphinX system

A pictorial version of the SphinX architecture is shown in Fig. 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.

\(^6\)The node is G itself, if G is a leaf node.
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 Fig. 11.

5. Performance model

In this section, we describe the experimental setup used to profile 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 [26], 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 also evaluated the performance of a representative schemaless indexing technique, namely Lore, whose Version 5.0 executable is publicly available [34]. Also we have compared with DGScan (refer to Section 4), 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 MB 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

Fig. 10. Architecture of SphinX.

![Fig. 11. Updating sphinx index structures.](image-url)
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 [35], Conference [27], Journal [27], Ham-Radio [38], and the remaining two are synthetic documents generated from the Xmark benchmark [26]. 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 field indicates 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.

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 [35]. 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 [27].

**Ham-Radio:** This document is obtained from the publicly available FCC Ham Radio database [38] 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.

**Xmark100 MB and Xmark1 GB:** These synthetic documents were generated from Xmark using its xmlgen data generator [26]. 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. 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 specifies 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 specified (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 [4] 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 Xmark1 GB, and Queries 8, 9, and 10, which are purely general search queries, are on Xmark100 MB. The queries can also be classified 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:

\[SQ_1: \text{/Shakespeare/Play[Title[text()=``The Tragedy of Antony and Cleopatra'']]/Persona/Persona}\]
\[SQ_2: \text{/conference/proceedings[year[text()=``1998'']]/title}\]
\[SQ_3: \text{/conference/in-proceedings[year[text()=``1995'']]/booktitle}\]
\[SQ_4: \text{/journal/article}\]
\[SQ_5: \text{/journal/article/author}\]
\[SQ_6: \text{/FMDataBase/FCCAmRadio/Address}\]
\[SQ_7: \text{/site/open_auctions/open_auction/type}\]

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

\[GQ_1: \text{//[Title[text()=``The Tragedy of Antony and Cleopatra'']}//Persona/Persona}\]
\[GQ_2: \text{//proceedings[year[text()=``1998'']]/title}\]
\[GQ_3: \text{//[year[text()=``1995'']//booktitle}\]
\[GQ_4: \text{//article}\]
\[GQ_5: \text{//journal//author}\]
\[GQ_6: \text{//FCCAMRadio//Address}\]
\[GQ_7: \text{//open_auction//type}\]
\[GQ_8: \text{//site/people/person[/address/zipcode > 4 and \text{/address/zipcode < 20]/country}\]
\[GQ_9: \text{//site/regions/item[1][quantity=1]/location}\]
\[GQ_{10}: \text{//site/categories/category//description//parlist/listitem\text{/parlist}\]

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.

6.1. Index construction performance

The index sizes 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 significant 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 efficiently, SphinX improves upon Lore by 2–4 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

<table>
<thead>
<tr>
<th>Document</th>
<th>DGScan (MB)</th>
<th>Lore (MB)</th>
<th>SphinX (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shakespeare</td>
<td>15</td>
<td>35</td>
<td>16</td>
</tr>
<tr>
<td>Conference</td>
<td>85</td>
<td>226</td>
<td>98</td>
</tr>
<tr>
<td>Journal</td>
<td>63</td>
<td>230</td>
<td>70</td>
</tr>
<tr>
<td>Ham-Radio</td>
<td>834</td>
<td>*</td>
<td>855</td>
</tr>
<tr>
<td>Xmark100 MB</td>
<td>190</td>
<td>377</td>
<td>193</td>
</tr>
<tr>
<td>Xmark1 GB</td>
<td>1900</td>
<td>*</td>
<td>1904</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document</th>
<th>DGScan (s)</th>
<th>Lore (s)</th>
<th>SphinX (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shakespeare</td>
<td>14</td>
<td>41</td>
<td>16</td>
</tr>
<tr>
<td>Conference</td>
<td>94</td>
<td>1433</td>
<td>122</td>
</tr>
<tr>
<td>Journal</td>
<td>73</td>
<td>969</td>
<td>95</td>
</tr>
<tr>
<td>Ham-Radio</td>
<td>1220</td>
<td>*</td>
<td>1305</td>
</tr>
<tr>
<td>Xmark100 MB</td>
<td>135</td>
<td>5760</td>
<td>144</td>
</tr>
<tr>
<td>Xmark1 GB</td>
<td>1930</td>
<td>*</td>
<td>2002</td>
</tr>
</tbody>
</table>
documents, Ham-Radio and Xmark, the Lore system failed to complete, signified 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:

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

\[
\text{IndexSize} = (N_e + N_a)K_1 + (N_e * F * K_2) + (N_f * K_3)
\]

where the first 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:

\[
\text{IndexCost} = (N_e * F + N_a + N_f)/\text{PageSize},
\]

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 formula for index size and index cost shows that they approximately hold for the data sets considered.

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 \((\text{Number of elements in the result set}/\text{Total number of such elements in the Document})\), 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 \(GQ_8, GQ_9, \text{and } GQ_{10}\) were run on the Xmark100 MB data set instead of Xmark1 GB, as Lore failed to build the index for Xmark1 GB.

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 \textit{Pindex}, which for a path \( p \) returns the set of objects \( O \) reachable via \( p \). The set of paths that are indexed by \textit{Pindex} should begin at named objects and “contain no regular expressions” and it cannot be used for all queries and path expressions [33]. Lore uses another physical query operator, called \textit{Scan}, for non-predicate queries, whose approach is similar to \textit{pointer-chasing} in object-oriented systems [36].

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 gains over schema-less 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 [6], T-index [7], ToXin [8], XISS [9], Index-Fabric [10], APEX [11], Forward-and-Backward-Index [12], Holistic Twig Join [13], Barashev and
Novikov [14], and ViST [15]. In this section, we briefly overview this previous work.

The earliest system that we are aware of is Lore [6], 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 [7] 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 [8]. 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 [9]. 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 [10], 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 [32] whereas refined paths can support queries that have wildcards, alternates, and different constants. The APEX index [11] 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 [12], 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 [13]. 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 [14] proposed new index structures suitable to support path expressions. The indexing structures considered in their work are B-trees and Patricia tries. Finally, ViST [15], 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 [8] and multi-layered Patricia tries [10]), 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 [26] 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
Our experimental results, over a variety of XML documents and RPE queries, show that SphinX performs significantly better compared to schema-less 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 Swarna-jayanti Fellowship from the Department of Science & Technology, Govt. of India.

References


[18] http://www.w3.org/Math


[27] http://www.informatik.uni-trier.de/~ley/db


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


[37] http://www.excelon.com

[38] ftp://ftp.ictcompress.com/pub/xmltextfiles