XIST: An XML Index Selection Tool

Kanda Runapongsa[1], Jignesh M. Patel[2], Rajesh Bordawekar[3], and Sriram Padmanabhan[4]

[1] Khon Kaen University, Khon Kaen 40002, Thailand, krunapon@kku.ac.th
[2] University of Michigan, Ann Arbor MI 48105, USA, jignesh@eecs.umich.edu
[3] IBM T.J. Watson Research Center, Hawthorne NY 10532, USA, bordaw@us.ibm.com
[4] IBM Sillicon Valley Lab, San Jose CA 95134, USA, srp@us.ibm.com

Abstract. XML indices are essential for efficiently processing XML queries which typically have predicates on both structures and values. Since the number of all possible structural and value indices is large even for a small XML document with a simple structure, XML DBMSs must carefully choose which indices to build. In this paper, we propose a tool, called XIST, that can be used by an XML DBMS as an index selection tool. XIST exploits XML structural information, data statistics, and query workload to select the most beneficial indices. XIST employs a technique that organizes paths that evaluate to the same result into ***structure equivalence groups*** and uses this concept to reduce the number of paths considered as candidates for indexing. XIST selects a set of candidate paths and evaluates the benefit of an index for each candidate path on the basis of performance gains for non-update queries and penalty for update queries. XIST also recognizes that an index on a path can influence the benefit of an index on another path and accounts for such index interactions. We present an experimental evaluation of XIST and current XML index selection techniques, and show that the indices selected by XIST result in greater overall improvements in query response times.

1 Introduction

An XML document is usually modeled as a directed graph in which each edge represents a parent-child relationship and each node corresponds to an element or an attribute. XML processing often involves navigating this graph hierarchy using regular path expressions and selecting those nodes that satisfy certain conditions. A naïve exhaustive traversal of the entire XML data to evaluate path expressions is expensive, particularly in large documents. Structural join algorithms [1,8,27] can improve the evaluation of path expressions, but as in the relational world, join evaluation consumes a large portion of the query evaluation time. Indices on XML data can provide a significant performance improvement for path expressions and predicates that match the index. However, an index degrades the performane of update operations and requires additional disk space. As a result, determining which set of indices to build is a critical administrative task. These considerations for building indices are not new, and have been investigated extensively for relational databases[24,3]. However, index selection for XML databases is more complex due to the flexibility of XML data and the complexity of its structure. The XML model mixes structural tags and values inside data. This extends the domain of indexing to the combination of tag names and element content values. In contrast, relational database systems mostly consider only attribute value domains for indexing. Moreover, the natural emergence of path expression query languages, such as XPath, further suggests the need for *path indices*. Path indices have been proposed in the past for object-oriented databases and even as join indices [23] in relational databases. Unlike relational and object-oriented databases, XML data does not require a schema. Even when XML documents have associated schemas, the schemas can be complex. An XML schema can specify optional, repetitive, recursive elements, and complex element hierarchies with variable depths. In addition, a path expression can also be constrained by the content values of different elements on the path. Hence, selecting indices to speed up the evaluation of XML path expressions is challenging.

This paper describes XIST, a prototype XML index selection tool that uses an integrated cost/benefit-driven approach. The cost models used in this paper are developed for a prototype native XML DBMS that we are building. As in other native XML systems, this system stores XML data as individual nodes [27,17,12,7,14], and also uses stack-based join algorithms [1,8,2] for evaluating path expressions. However, the general framework of XIST can be adapted to systems with other cost models by modifying the cost equations that are presented in this paper.

1.1 Contributions

Our work makes the following contributions:

- We propose a cost-benefit model for computing the effectiveness of a set of XML indices. In this cost-benefit analysis, we account for the index update costs and also consider the interaction effect of an index on the benefit of other indices. By carefully reasoning about index interactions, we can eliminate redundancy computations in the index selection tool.
- When the XML schema is available, XIST uses a concept of *structure equivalence groups*, which results in a dramatic reduction in the number of candidate indices.
- We develop a *flexible* tool that can recommend candidate indices even when only some input sources are available. In particular, the availability of only either the schema or the user workload is sufficient for the tool.
- Our experimental results indicate that XIST can produce index recommendations that are more efficient than those suggested by current index selection techniques. Moreover, the quality of the indices selected by XIST increases as more information and/or more disk space is available.

The remainder of this paper is organized as follows. Section 2 presents data models, assumptions, and terminologies used in this work. In Section 3, we describe the overview of the XIST algorithm. Sections 4, 5, and 6 describe the individual components of XIST in detail. Experimental results are presented in Section 7, and the related work is described in Section 8. Finally, Section 9 contains our concluding remarks and directions for future work.

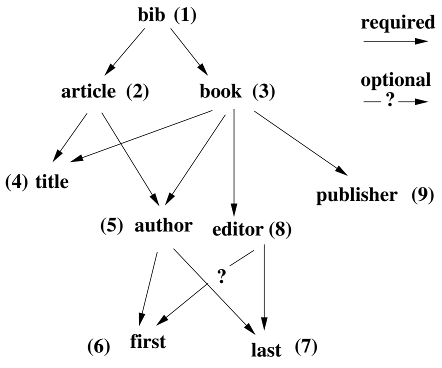

Fig. 1. Sample XML Schema

2 Background

In this section, we describe the XML data models, terminologies, and assumptions that we use in this paper.

2.1 Models of XML Data, Schema, and Queries

We model XML data as a tree. We encode the nodes of the tree using Dietz's numbering scheme [11,10]. Each node is labeled with a pair of numbers representing its positions on preorder and postorder traversals. Using Dietz's proposition, node x is an ancestor of node y if and only if the preorder number of node x is less than that of node y and the postorder number of node x is greater than that of node y. This basic numbering scheme can be extended to include an additional number that encodes the level of the node in the XML data tree. This numbering scheme is used in our cost models when performing the structural joins [27,8,1] between parents and children or between ancestors and descendants. We need the additional level information of a node to differentiate between parent-child and ancestor-descendant relationships.

We model an XML schema as a directed label graph. An XML schema is written in the W3C XML Schema Definition Language which describes and constrains the content of XML documents. It allows the definition of groups of elements and attributes. We can group elements sequentially, or choose some elements to appear and others to disappear, or define an unordered set of elements. An edge between each node in the XML Schema graph here is the edge between a parent element and a child element while child elements are grouped sequentially. The required edge from node A to node B refers that the minimum number of occurrences of B that appear inside node A is at least one; on the other hand, the optional edge refers that the minimum number of occurrences of B that appear inside node A is zero. Figure 1 shows the schema of a sample bibliography database that we use as a running example throughout this paper.

2.2 Terminologies and Assumptions

We now define terminologies for paths and path indices that are used in this paper.

A *label path* p (referred to as a path as well) is a sequence of labels $l_1/l_2/.../l_k$ where the length of the path is k. We assume that the returned result of path p is the ending node of path p. Path p_d is dependent on path p if p is a subpath of p_d. For example, path $l_1/l_2/.../l_k$ is dependent on path l_1/l_2.

A *path index (PI)* on path p is an index that has p as a key and returns the node IDs (NIDs) of the nodes that matched p. (As discussed in Section 2.1, an NID is simply a triplet encoding the begin, end, and level information.)

An *element index (EI)* is a special type of a path index. Since an element "path" consists of only one node, the element index stores only the NIDs of the nodes matched by the element name.

A *candidate path (CP)* is a path on which XIST considers as a candidate for building an index. The corresponding index on the candidate path is called a *candidate path index (CPI)*.

In our work, we consider the following types of indices as candidates: (i) structural indices on individual elements, (ii) structural indices on simple paths as defined above, and (iii) value indices on the content of elements and attribute values. It is possible to extend our models to include other types of indices, such as an index on a twig query, in the future.

3 The XIST Algorithm

In making its recommendations for a set of indices, XIST is designed to work flexibly with the availability of a schema, a workload, and data characteristics of an XML data set. Figure 2 shows an overview of XIST, which consists of four modules that adapt to a given set of input configuration. The first module is the *candidate path selection* module, which eliminates a large number of potentially irrelevant path indices. It uses the following two techniques: (i) If the query workload is available, this module eliminates paths that are not in the query workload, and (ii) If the schema is available, the tool identifies and prunes equivalent paths that can be evaluated using a common index.

To compute the benefits of indices on candidate paths, we use either the *cost-based benefit computation* module or the *heuristic-based benefit computation* module, depending on the availability of data statistics. When data statistics are available, the cost-based benefit computation module is employed. When data statistics are not available, the heuristic-based benefit computation module is operated instead. (The computation carried out by the benefit computation modules may be present in many optimizers, and these modules could be shared by the query optimizer.)

The last module is *configuration enumeration*, which in each iteration chooses an index from the candidate index set that yields the maximum benefit. The configuration enumeration module continues selecting indices until a space constraint, such as a limit on the available disk space, is met.

Figure 3 presents the overview of the XIST algorithm. In the following sections, we describe in detail the steps shown in this figure. The first phase, the selection of candidate paths (CPs), is presented in Section 4. Section 5 discusses the benefit computations for the indices on the selected CPs (CPIs). Finally, Section 6 describes the re-computation for the benefits of CPIs that have not been chosen (line 9).

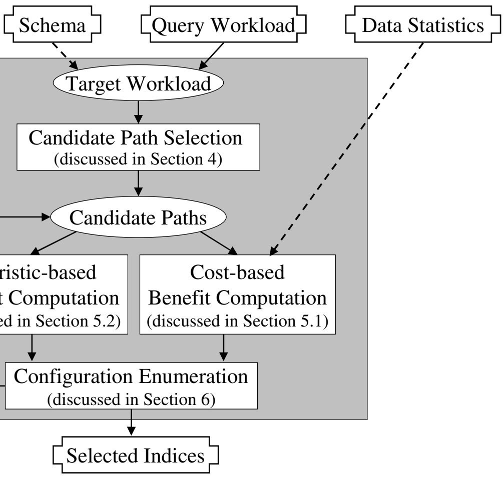

Fig. 2. The XIST Architecture

4 Candidate Path Selection

In this section, we address the important issue of selecting candidate paths (CPs). Since the total number of candidate paths for an XML schema instance can be very large, for efficiency purposes, it is desirable to identify a subset of the candidate paths that can be safely dropped from consideration without reducing the effectiveness of the index selection tool. The candidate path selection module in XIST employs a novel technique to achieve this goal.

Our strategy for reducing the number of CPs is to share an index among multiple paths. Our approach for grouping similar paths involves identifying paths that share the same ending nodes. Since the ending nodes of these paths are the same, the index on a single path of the paths in this group returns the same set of nodes that other paths will (assume that the returned nodes of the path are only the ending nodes of the path).

Definition 1. *A path p_u, $n_1/n_2/.../n_k$, is a unique path if there is one and only one incoming required edge to n_i and there is one and only one incoming required edge to node n_i which must be exclusively from node n_{i-1} for $i = 2$ to k.*

Definition 2. *Path p_1 and path p_2 are in the same structure equivalence group if 1) p_1 and p_2 share the same suffix subpath, 2) the starting node of the shared suffix subpath must have only one incoming edge, and 3) the non-suffix subpath is a unique path.*

We refer to a group of paths that share the same ending nodes as a *Structure Equivalence Group (SEG)*. As an example of an SEG consider the schema shown in Figure 1. A sample SEG in this schema is the set containing the following paths: `bib/book/publisher`, `book/publisher`, and `publisher`. For brevity, we refer to the paths using the concatenation of the first letter of each element on the path (for

```
Inputs: An index space constraint $k$ (which can be the available disk space [default],
or the number of indices), an XML schema, a query workload, and optional data statistics.
The algorithm requires at least the schema or the workload.
Output: A set of recommended path indices, $S$
XIST()
// Phase 1: Candidate Path Selection
1.  use the XML schema or the query workload to compute the target workload $W$.
2.  choose paths and subpaths in the target workload $W$ to form the set of candidate paths ($CPs$).
// Phase 2: Benefit Computation
3.  for each CP $p$, compute the benefit of the corresponding CPI: $I_p$. The benefit is $B(I_p)$
// Phase 3: Configuration Enumeration
4.  initialize the set of selected indices, $S$ to $I_E$ where $I_E$ is the set of element indices.
5.  while ($|S| < k$)
6.      select $p \in CP$ and $I_p \notin S$ such that $B(I_p)$ is the maximum.
7.      $CP = CP - p$
8.      $S = S \cup I_p$
9.      $\exists p \in CP$, recompute the benefits of candidate path indices, $B(I_p)$
10. endwhile
```

Fig. 3. The XIST Algorithm

example, we refer to `bib/book/publisher` as `bbp`). As shown in the schema graph in Figure 1, `bbp` is a unique path as `p` has only one parent, and each of its ancestors also has only one parent. Nodes that match `bbp` are the same as nodes that match the suffix paths of `bbp` which are `bp` and `p`. Thus, these paths are in the same SEG. However, suppose that if `publisher` has two parents: `book` and `article`. Then, `bib/book/publisher`, `book/publisher` and `publisher` do not form a SEG since we cannot assume that `publisher` is the publisher of the book (`book/publisher`). The `publisher` can also be the publisher of the article (`book/article`). On the other hand, `book/publisher` and `/bib/book/publisher` form a SEG. In this SEG, the shared suffix subpath is `book/publisher`

Instead of building indices on each path in an SEG, XIST only builds an index on the shortest path in each SEG. We choose the shortest path because the space and access time of indices in SEGs can often be reduced. This is because the shortest path can simply be a single element. In such an index, we only need to store three integers *(begin, end, level)* per index entry, whereas in indices on longer paths require storing six integers per index entry.

Since structure equivalence groups are determined based only on the XML schema, these groups are valid for all XML documents conforming to the XML schema. The SEGs cannot be determined by using data statistics because statistics do not indicate whether a node is contained in only one element type. Since some elements in XML data can be optional, they may not appear in XML document instances and thus may not appear in data statistics as well.

5 Index Benefit Computation

In this section, we describe the internal benefit models used by the XIST algorithm to compute the benefits of candidate path indices (CPIs). The total benefit of an index

```
Inputs: A set of existing indices S, a target workload W, and a CPI on path $p$, $I_p$
Output: The benefit of $I_p$, $B(I_p)$
ComputeIndexBenefit()
// $F_E$ and $F_D$ are functions for benefit computation
1.  $B_E = 0$
2.  for path $p_e \in EQ(p)$ and $p_e \in W$
3.    $B_e = F_E(p, p_e, S)$
4.    $B_E = B_E + B_e$
5.  endfor
6.  $B_D = 0$
7.  for path $p_d$ with $p$ as a subpath and $p_d \in W$
8.    $B_d = F_D(p, p_d, S)$
9.    $B_D = B_D + B_d$
10. endfor
11. if data statistics are available
12.   $B(I_p) = B_E + B_D - U(I_p)$
13. else
14.   $B(I_p) = B_E + B_D$
```

Fig. 4. The Index Benefit Computation Algorithm for CPI I_p

I_p, $B(I_p)$, is computed as the sum of: (i) B_E, which is the benefit of using I_p for answering queries on the equivalent paths of p (recall that all paths in an EQ share the same path index), and (ii) B_D, which is the benefit of using I_p for answering queries on the dependent paths of p. Figure 4 presents an algorithm for computing the total benefit of I_p, $B(I_p)$.

5.1 Cost-Based Benefit Computation

When data statistics are available, XIST can estimate the cost of evaluating paths more accurately. The collected data statistics consist of a sequence of tuples, each representing a path expression and the cardinality of the node set that matches the path (also called the cardinality of a path expression). XIST uses the path cardinality to predict path evaluation costs. In reality, however, these costs depend largely on the native storage implementation and the optimizer features of an XML engine. To address this issue, we approximate the path evaluation costs via abstract cost models based on our experimental native XML DBMS.

Computing Evaluation Costs. The cost of evaluating a path with the index on the path is estimated to be proportional to the cardinality of the path since the path index is implemented using a hash index and the hash index access cost is proportional to the number of items retrieved from the hash index. Let $C(p_1/p_2, S \cup I_{p_1/p_2})$ be the cost of evaluating p_1/p_2. Then,

$$C(p_1/p_2, S \cup I_{p_1/p_2}) \approx K_I \times (|p_1/p_2|) \quad (1)$$

where K_I is a constant and $|p_1/p_2|$ is the cardinality of the nodes matched by p_1/p_2.

If an index on a path does not exist, XIST splits the path into two subpaths and then recursively evaluates them. When splitting the path, XIST needs to determine the join order of subpaths to minimize the join cost. The chosen pair has the minimal sum of the cardinalities of subpaths. Subpaths are recursively split until they can be answered using existing indices. After subpaths are evaluated, their results are recursively joined to answer the entire path. Finding an optimal join order is not the focus of this paper, but it has been recently proposed [26].

When XIST joins a path of two indexed subpaths, it uses a structural join algorithm [1], which guarantees that the worst case join cost is linear in the sum of sizes of the sorted input lists and the final result list. Let S be the set of indices which exclude the index on p_1/p_2, and $C(p_1/p_2, S)$ be the cost of joining between path p_1 and path p_2. Then,

$$C(p_1/p_2, S) \approx K_J \times (|p_1| + |p_2| + |p_1/p_2|) \tag{2}$$

where K_J is the constant and $|p_i|$ is the estimated cardinality of the nodes that match p_i. The estimated cardinality of the nodes that match the paths are given as an input of the XIST tool (by the XML estimation module in the system).

Since the maintenance cost for an index can be very expensive, XIST also considers the maintenance cost in the index benefit computation. The actual cost for updating a path index is very much dependent on the system implementation details, and different systems are likely to have different costs for index updates. In this paper, for simplicity, we use an update cost model in which the update cost for a given path index is proportional to the the number of entries being updated in the path index. (This cost model can be adapted in a fairly straightforward manner if the cost needs to include a log-based factor, which is a characteristic for tree-based indices.)

Let $U(I_{p_1/p_2})$ be the cost of updating the index on path p_1/p_2, then

$$U(I_{p_1/p_2}) \approx K_U \times (|p_1/p_2|) \tag{3}$$

where K_U is the constant and $|p_1/p_2|$ is the cardinality of the nodes that match p_1/p_2.

Using Cost Models for Computing Benefits. Now we describe how the cost models are used to compute the total benefit of an index when data statistics are available. The benefit function $F_E(p, p_e, S)$ is the function to compute the benefit of using I_p to completely evaluate a path in the structure equivalence group of p (p_e), assuming the set of indices S exists. The benefit function $F_D(p, p_d, S)$ is the function to compute the benefit of using I_p to partially answer a dependent path of p (p_d), assuming the set of indices S exists.

$$\boxed{\begin{array}{lcl} F_E(p, p_e, S) & = & C(p_e, S) - C(p, S \cup I_p) \\ F_D(p, p_d, S) & = & C(p_d, S) - C(p_d, S \cup I_p) \end{array}}$$

Fig. 5. F_E and F_D for I_p (with Statistics)

5.2 Heuristic-Based Benefit Computation

When data statistics are not available, XIST estimates the benefit of the index by using the lengths of queries and the length of the candidate path (CP). The benefit of a candidate path index (CPI) is estimated based on: a) the number of joins required to answer queries with and without the CPI, and b) the use of a CPI to completely or partially evaluate a query.

In the following paragraphs, we use the following notations: p is a CP, I_p is a CPI, p_e is an equivalent path of p (a path that I_p can completely answer), and p_d is a dependent path of p (a path that I_p can partially answer).

We first consider the benefit of I_p when it can completely answer a query. This benefit is computed by the F_E function, which estimates the number of joins needed as the length of the shortest unindexed subpath of p. Let $L(p)$ be the length of path p, S be the set of existing indices, and $L^{'}(p, S)$ be the length of the shortest unindexed subpath in p. $L^{'}(p, S)$ is the difference between the length of p and that of the longest indexed subpath of p.

Next, we consider the benefit of I_p when it can partially answer a query. This benefit is computed by the F_D function. Like F_E, F_D estimates the number of joins needed to answer the query. However, in this case, the number of joins needed is more than just the length of an unindexed subpath of the query. The closer the length of p to the length of unindexed subpath of the query, the higher benefit of I_p is. We use the difference between the length of p and that of the query as the number of the joins that the index cannot answer. The benefit functions F_E and F_D are shown in Figure 6.

$$\boxed{\begin{array}{lcl} F_E(p, p_e, S) & = & L^{'}(p_e, S) \\ F_D(p, p_d, S) & = & L^{'}(p_d, S) - (L(p_d) - L(p)) \\ & = & L^{'}(p_d, S) - L(p_d) + L(p) \end{array}}$$

Fig. 6. F_E and F_D for I_p (Without Statistics)

6 Configuration Enumeration

After the benefit of each CPI is computed using F_E and F_D in the index benefit algorithm (Figure 4), the first two phases of the XIST algorithm (Figure 3) are completed. In the third phase, XIST first selects the CPI with the highest benefit to the set of chosen indices S. Since XIST takes the index interaction into account, it needs to recompute the benefits of CPIs that have not been chosen. The key idea in efficiently recomputing the benefits of CPIs is to recompute only the benefits of the indices on paths that are affected by the chosen indices. A naïve algorithm would recompute the benefit of each CPI that has not been selected. In contrast, XIST employs a more efficient strategy no matter whether it uses the heuristic-based benefit computation or cost-based benefit computation. The strategy is briefly described below.

XIST considers three types of paths that are affected by a selected index on path p: (a) subqueries that have not been evaluated and that contain p as a subpath, (b) paths that are subpaths of p, and (c) paths that are not subpaths of p but are subpaths of paths in (a).

Using these relationships between selected paths and other unselected paths, we can reduce the number of benefit re-computations for the unselected indices. We need to re-compute the benefits of unselected indices because the benefits of these indices depend on the existence of selected indices. The situation in which the benefits of one index depends on the existence of other indices is called *index interaction*.

If we did not find such relationships between the selected indices and the unselected indices, we could spend an excessive amount of time in computing the benefits of many remaining unselected indices.

7 Experimental Evaluation

In this section, we present the results from an extensive experimental evaluation of XIST, and compare it with current index selection techniques.

7.1 Experimental Setup

The XIST tool that we implemented is a stand-alone C++ application. It uses the Apache Xerces C++ version 2.0 [20] to parse an XML schema. It also implements the selection and benefit evaluation of candidate indices, and the configuration enumeration. We then used the indices recommended by the XIST toolkit as an input to an native XML DBMS that we are currently developing. This system implements stack-based structural join algorithms [1]. It uses B+tree to implement the value index, and uses a hash indexing mechanism to implement the path indices. It evaluates XML queries as follows: if a path query matches an indexed pathname, the nodes matching the path are retrieved from the path index. If there is no match, the DBMS uses the structural join algorithm [1] to join indexed subpaths. Queries on long paths are evaluated using a pipeline of structural join operators. The operators are ordered by the estimated cardinality of each join result, with the pair resulting in the smallest intermediate result being scheduled first. A query with a value-based predicate is executed by evaluating the value predicate first.

In all our experiments, the DBMS was configured to use a 32 MB buffer pool. All experiments were performed on an 1.70 GHz Intel Xeon processor, running Debian Linux version 2.4.13.

7.2 Data Sets and Queries

We used the following four commonly used XML data sets: DBLP [18], Mondial [25], Shakespeare Plays [13], and XMark benchmark [22]. For each data set, we generated a workload of ten queries. These queries were generated using a query generator which takes the set of all distinct paths in the input XML documents as input. The detail of the generation method can be found in the full-length version of the paper [21].

As an example, using the generation method, some of the queries on the Plays data set are as follow: `FM/P` and `/PLAY/ACT/EPILOGUE/SPEECH[SPEAKER="KING"]`.

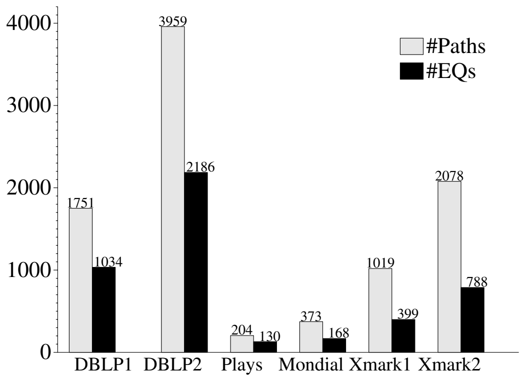

Fig. 7. Number of Paths and Structure Equivalence Groups

7.3 Experimental Results

We now present experimental results that evaluate various aspects of the XIST toolkit. First, we demonstrate the effectiveness of the path structure equivalence group (SEG) in reducing the number of candidate paths. Then, we compare the performance of XIST with the performance of a number of alternative index selection schemes. We also performed the experiments to access the impact of the different types of inputs (namely query workload, XML schema and statistics) on the behavior of the XIST toolkit. In addition, we also analyzed the performance of all the index selection schemes when the workload changes over time. However, due to the space limitation, only partial experimental results of the impact of different types of inputs are presented here; more experimental results are presented in the full-length version of this paper [21].

The execution time numbers presented or analyzed in this paper are cold numbers, i.e., the queries do not benefit from having any pages cached in the buffer pool from a previous run of the system.

Effectiveness of Structure Equivalence Groups. Paths in an structure equivalence group are represented by a single unique path which is the smallest path pointing to the same destination node. Therefore, the number of structure equivalence groups denotes the number of such unique paths.

Figure 7 plots the number of paths and the number of structure equivalence groups for all the data sets used in this experimental evaluation. In this figure, DBLP1 and XMark1 represent those paths from DBLP and XMark with lengths up to five, and DBLP2 and XMark2 represent those paths with lengths up to ten. As shown in Figure 7, the number of structure equivalence groups is fewer than the number of total paths by 35%-60%. *This result validates our hypothesis that the number of candidate paths can be reduced significantly using the XML schema to exploit structural similarities.*

Comparison of Different Indexing Schemes. We compare the performance of the following sets of indices: indices on elements (*Elem*), indices on paths with length up to two (*SP*), indices suggested by XIST (*XIST*), and indices on the full path query

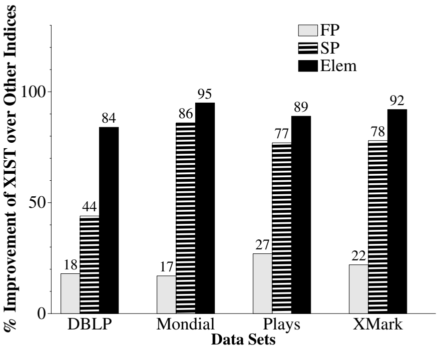

Fig. 8. Performance Improvement of *XIST*

definitions (*FP*). The *Elem* index selection strategy is interesting because it is a minimal set of indices to answer any path query. *SP* is a set of indices on short paths. *XIST* is a set of indices chosen by the XIST tool with given all input information (schema, statistics, and query workload). *FP* is a set of indices that requires no join when evaluating paths without value-based predicates.

All indexing schemes (*Elem*, *SP*, *XIST*, and *FP*) only build indices on paths without value-based predicates. To evaluate paths with value-based predicates, a join operation is used between the nodes returned from indices and the nodes that match the value predicates. We choose to separate the value indices from a path indices to avoid having an excessive number of indices – one for each possible different value-predicate. In our experimental setup, all indexing schemes share the same value indices to evaluate paths with value-based predicates.

Figure 8 shows the performance improvement of *XIST* over other indices for the four experimental data sets. The results shown in Figure 8 illustrate that *XIST* consistently outperforms all other index selection methods for all the data sets. *XIST* is better than *Elem* and *SP* because *XIST* requires fewer joins for evaluating the queries. *XIST* performs better than *FP* largely because the use of structure equivalence groups (SEGs) while evaluating path queries. In many cases, long path queries without value predicates are equivalent to queries on a single element. In such cases, if the size of the element index is smaller than the size of the path index, XIST recommends using the element index to retrieve answer. On the other hand, *FP* needs to access the larger path index. Another reason for the improved performance with *XIST* is that for some data sets the total size of *XIST* indices (including element indices and path indices) is smaller than that of *FP* indices (including element indices and path indices). The total size of *XIST* indices is smaller because it shares a single index among the equivalent paths. Note that the equivalent paths cannot be determined when using *FP* since *FP* does not take a schema as an input information. *FP* takes only query workload as an input information. Table 1 presents the sizes of data sets and indices for all data sets.

Table 1. Sizes of Data Sets and Indices

Data Set	Size	Index Size (MB)			
	(MB)	*Elem*	*SP*	*FP*	*XIST*
DBLP	117	91	117	110	101
Mondial	2	2	3	3	3
Plays	8	80	86	85	81
XMark	11	27	27	27	27

7.4 Impact of Input Information on XIST

In this section, we compared the execution times when using indices suggested by XIST with indices selected by other index selection strategies.

Due to space limitations, in this paper we only present the results for DBLP and XMark when the workload information is available. DBLP represents a shallow data set (short paths), and XMark represents a deep data set (long paths). The experimental results for these two data sets are representative of the results for the other two data sets.

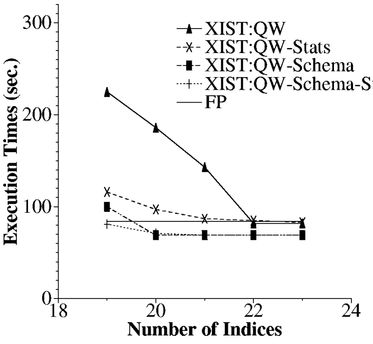

Fig. 9. Performance on DBLP

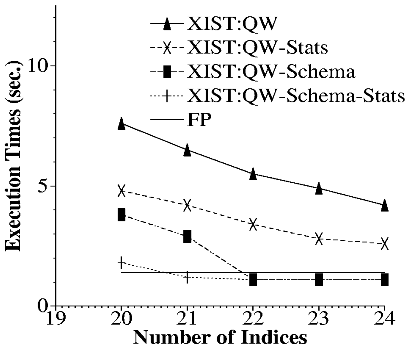

Fig. 10. Performance on Xmark

When the workload information is available, *FP* and *XIST* exploit the information to build indices that can cover most of the queries. *FP* indices cover all paths without value-based predicates in the workload. Thus, its index selection is close to optimal (without any join). When using *FP*, the only joins that the database needs are the joins between the returned nodes from the path index and the nodes that satisfy the value predicates. In Figures 9-10, the number of *FP* indices is used to assign the initial number of indices that the XIST tool generates.

As opposed to the heuristic-based benefit function, the cost-based benefit function guarantees that the more useful indices are chosen before the less useful indices. The execution times of *XIST* with `QW-Stats` (`QW-Schema-Stats`) gradually decrease as opposed to the execution times of *XIST* with `QW` (`QW-Schema`). This is particularly noticeable in Figure 10.

8 Related Work

In the 1-index [19], data nodes that are bisimilar from the root node stored in the same node in the index graph. The size of the 1-index can be very large compared to the data size, thus A(k)-index [16] has been proposed to make a trade off between the index performance and the index size. While k-bisimilarity [16] is determined by using XML data, the SEGs in this paper are determined by using an XML schema. Recently, D(k)-index [6], which is also based on the concept of bisimilarity, has been proposed as an adaptive structural summary. Like XIST, D(k) also takes the query workload as an input. However, XIST also takes the XML schema into account while D(k) does not. Although both k-bisimilarity and SEGs group paths that lead to the nodes with the same label, SEGs group paths in an XML schema but k-bisimilarity group paths in XML data.

Chung et al. have proposed APEX [9], an adaptive path index for XML documents. Like APEX, XIST exploits the query workload to find indices that are most likely to be useful. On the other hand, APEX does not distinguish the benefit of indices on two paths with same frequencies, but XIST does. In addition, APEX does not exploit data statistics and XML schema in index selection as opposed to XIST.

Recently, Kaushik et al. have proposed F&B-indexes that use the structural features of the input XML documents [15]. F&B indexes are forward-and-backward indices for answering branching path queries. Some heuristics in choosing indices, such as prioritizing short path indices over long path indices are proposed [15]. On the other hand, XIST takes many additional parameters, such as the information from a schema or a query workload.

Many commercial relational database systems employ index selection features in their query optimizers. IBM's DB2 Advisor [24] recommends candidate indices based on the analysis of workload of SQL queries and models the index selection problem as a variation of the knapsack problem. The Microsoft SQL Server [3,4] uses simpler single-column indices in an iterative manner to recommend multi-column indices. XIST groups a set of paths (a set of multiple-columns) that can share the index.

Our work is closest to the index selection schemes proposed by Chawathe et al. [5] for object oriented databases. Both the index selection schemes [5] and XIST find the index interaction through the relationships between subpath indices and queries. However, XIST exploits the structural information to reduce the number of candidate indices and optimize the query processing of XML queries while [5] only looks at the query workload to choose candidate indices for evaluating object-oriented queries.

9 Conclusions

In this paper, we have described XIST, an XML index selection tool, which recommends a set of path indices given a combination of a query workload, a schema, and data statistics. By exploiting structural summaries from schema descriptions, the number of candidate indices can be substantially reduced for most XML data sets and workloads. XIST incorporates a robust benefit analysis technique using cost models or a simplified heuristic. It also models the ability of an index to effectively evaluate sub-paths of a path expression. Our experimental evaluation demonstrates that the indices selected by XIST perform better compared to existing methods. In our experimental evaluation, we had to tailor the cost model used in XIST to accurately model the techniques that are implemented in our native XML system. However, we believe that the general framework of XIST, with its use of structure equivalence groups and efficient benefit recomputation methods, can be adapted for use with other DBMSs with different implementation and query evaluation algorithms. To adapt this general framework to other systems, accurate cost models equations are required that account for the system-specific details. Within the scope of this paper, we have chosen to focus on the general framework and algorithms of an XML index selection tool, and have demonstrated that its effectiveness for our native XML DBMS.

In the future, we plan on extending XIST to include additional types of path indices, such as indices on regular path expressions and on twig queries.

References

1. S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, San Jose, CA, February 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD*, pages 310–321, Madison, Wisconsin, June 2002.
3. S. Chaudhuri and V. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, pages 146–155, Athens, Greece, September 1997.
4. S. Chaudhuri and V. Narasayya. Auto Admin "What-If" Index Analysis Utitlity. In *SIGMOD*, pages 367–378, Seattle, Washington, June 1998.
5. S. Chawathe, M. Chen, and P. S. Yu. On Index Selection Schemes for Nested Object Hierarchies. In *VLDB*, pages 331–341, Santiago, Chile, September 1994.
6. Q. Chen, A. Lim, and K. W. Ong. D(K)-Index:An Adaptive Structural Summary for Graph-Structured Data. In *SIGMOD*, pages 134–144, San Diego, CA, June 2003.
7. S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient Complex Query Support for Multiversion XML Documents. In *EDBT*, pages 161–178, Prague, Czech Republic, 2002.
8. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, pages 263–274, Hong Kong, China, 2002.
9. C. Chung, J. Min, and K. Shim. APEX:An Adaptive Path Index for XML Data. In *SIGMOD*, pages 121–132, Madison, WI, June 2002.
10. P. Dietz and D. Sleator. Two Algorithms for Maintaining Order in a List. In *Proc. 19th Annual ACM Symp. on Theory of Computing (STOC'87)*, pages 365–372, San Francisco, California, 1987.
11. P. F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the Fourtheenth Annual ACM Symposium of Theory of Computing*, pages 122–127, San Francisco, California, May 1982.
12. T. Grust. Accelerating XPath Location Steps. In *SIGMOD*, pages 109–120, Madison, Wisconsin, 2002.
13. J. Bosak. The Plays of Shakespeare in XML. `http://metalab.unc.edu/bosak/xml/eg/shaks200.zip`.
14. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, N. Niwatwattana, D. Srivastava, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.
15. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Expressions. In *SIGMOD*, pages 133–144, Madison, WI, May 2002.

16. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *ICDE*, pages 129–140, San Jose, CA, February 2002.
17. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, pages 361–370, Roma, Italy, September 2001.
18. M. Ley. The DBLP Bibliography Server. `http://dblp.uni-trier.de/xml/`.
19. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of the International Conference on Database Teorey*, pages 277–295, Jerusalem, Israel, January 1999.
20. T. A. X. Project. Xerces C++ Parser. `http://xml.apache.org/xerces-c/index.html`.
21. K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan. XIST:An XML Index Selection Tool. `http://gear.kku.ac.th/~krunapon/research/xist.pdf`.
22. A. Schmidt, F. Wass, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. An XML Benchmark Project. Technical report, CWI, Amsterdam, The Netherlands, 2001. `http://monetdb.cwi.nl/xml/index.html`.
23. P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.
24. G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 Advisor:An Optimizer Smart Enough to Recommend its Own Indexes. In *ICDE*, pages 101–110, 2000.
25. W. May. The Mondial Database in XML. `http://www.informatik.uni-freiburg.de/~may/Mondial/`.
26. Y.Wu, J. Patel, and H. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, pages 443–454, Bangalore, India, 2003.
27. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Managment Systems. In *SIGMOD*, Santa Barbara, California, May 2001.