XIST: An XML Index Selection Tool

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

[1] krunapon@kku.ac.th Khon Kaen University, Khon Kaen 40002, Thailand
[2] jignesh@eecs.umich.edu University of Michigan, Ann Arbor MI 48105, USA
[3] {bordaw, srp}@us.ibm.com IBM T.J. Watson Research Center, Hawthorne NY, USA

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 are evaluated to the same result into ***equivalence classes*** 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 on 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 and often require less disk space.

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 naive exhaustive traversal of the entire XML data to evaluate path expressions is expensive, particularly in large documents. Structural join algorithms [1, 7, 29] 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 certain path expressions and predicates since they can directly select the nodes of interest. The choice of indices is one of the most critical administrative decisions for any database system. Building an index can potentially improve the response time of applicable queries but it degrades the performance of updates. Furthermore, in many practical cases, the amount of storage for indices is limited.

These considerations for building indices have been investigated extensively for relational databases [3,26]. 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 suggest the need for *path indices*. Path indices have been proposed in the past for object-oriented databases and even as join indices [25] in relational databases. To the best of our knowledge, the only research that deals with selecting useful path indices to optimize the overall query workload time was proposed by Chawathe et al. [5]. However, this work [5] has its applicability in object-oriented databases, not in XML databases. Unlike relational and object-oriented databases, XML data does not require a schema. Even when XML documents have schemas, the schemas can be complex. An XML schema can specify optional, repetitive, recursive elements, and complex element hierarchies with variable depths. Path queries need to match the schema if it exists. In addition, a path expression can also be constrained by the content values of different elements on the path. Hence, selecting indices to aid the evaluation of such a path expression can be challenging.

Recently, several indexing schemes have been proposed to support complex navigation and selection on XML data [8, 11, 12, 16–18, 20, 21, 23]. While these indexing schemes address at least one XML indexing requirement, they do not fully address the automatic index selection problem in XML databases. The focus of those papers is on the technology of efficient indexing of XML data. In contrast, the focus of our work is on the methods to identify an optimal set of indices assuming path, element, and value indices. In our problem setting, the important issues include the benefit of an index on a path for partially matching a query and the index interaction.

This paper describes XIST, a prototype XML index selection tool using 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 [6, 12, 15, 18, 29], and also uses stack-based join algorithms [1, 2, 7] 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 we use in this paper to accurately model the actual algorithms that are employed in that system.

1.1 Problem Statement

Our goal is to select a set of indices given a combination of a query workload, a schema, and data statistics. We evaluate the usefulness or the *benefit* of an index by comparing the total execution costs for all queries in the workload before and after the index is available. We compare the benefit with the *cost* of updating the index and recommend a set of indices that is the most effective given a constraint on the amount of disk space. To choose a set of indices, we compare the relative *quality* of any two sets of indices. The *total cost* of a set of indices for a given workload is defined as the sum of the execution costs of all queries in the workload. We use the total cost as the quality metric. Given a workload, a set of indices that has the least total cost is called the *optimal set* *of indices*. A smaller total cost for a set of indices indicates a higher quality of the set of indices.

The goal of the index selection tool is to suggest a set of indices that is optimal or as close to optimal as possible within a space constraint, such as a limit on the available disk space for storing indices.

1.2 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 update costs for the index 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 path *equivalence classes*, 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.

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 directed label graph $G_D = (V_D, E_D, root_D, NID_D, label, value)$. Each edge in E_D indicates a parent-child or node-value relationship. An element with content is given a value via the $value$ function. Each element in V_D is labeled with its type name via the $label$ function, and with a unique identifier via the NID_D function. An attribute is treated like an element. Every node is reachable from the element $root_D$.

We encode the nodes of the XML graph using Dietz's numbering scheme [9, 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

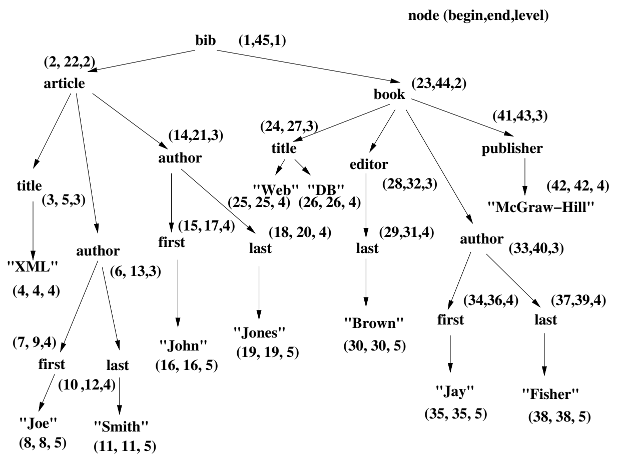

Fig. 1. Sample XML Data Graph

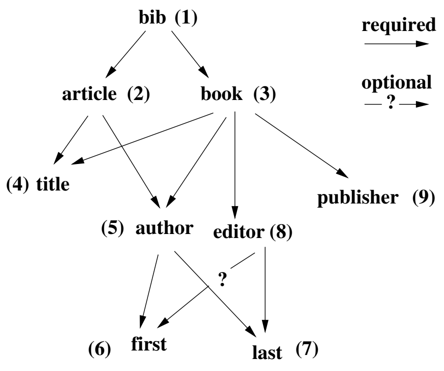

Fig. 2. Sample XML Schema

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 [1, 7, 29] between parents and children or between ancestors and descendants.

Figure 1 shows a data graph for a sample bib database using the Dietz numbering scheme. For this example, given the path query `author/last`, we could perform a structural join by looking up the element indexes on `author` and `last`, and verifying parent-child relationships using the numbering scheme. We need the additional level information of a node to differentiate between parent-child and ancestor-descendant relationships.

We also model an XML schema as a directed label graph $G_S = (V_S, E_S, root_S, label, NID_S)$. Each edge in E_S indicates a parent-child relationship. Each node in V_S is labeled with its type name via the $label$ function, and with a unique identifier via the

NID_S function. When applying the NID_S function to a node x, the output is a unique number associated with that node.

Figure 2 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. A *node path* is a sequence of nodes $n_1/n_2/.../n_k$ such that an edge exists between node n_i and node n_{i+1} for $1 \leq i \leq k-1$. A path $l_1/l_2/.../l_k$ matches a node path $n_1/n_2/.../n_k$ if $l_i = label(n_i)$ where $\forall i = 1...k$. A path $l_1/l_2/.../l_k$ matches a node n if $l_1/l_2/.../l_k$ matches a node path ending in n. We refer to node n as the ending node of path $l_1/l_2/.../l_k$. 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.) To allow using the index to evaluate a matching path expression in both forward and backward directions, we assume that the path index structure stores the NIDs of the starting and ending nodes of the paths. For example, the index on `book/editor` stores $\{$[(23,44,2), (28,32,3)]$\}$ where (23,44,2) is the NID of the starting node and (28,32,3) is the NID of the ending node.

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.

Table 1 summarizes the terminology that is used in this paper; some of these terms are described in the following section.

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 3 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

Terms	Explanation
S	Set of indices
W	Target workload
I_p	Index on path p
EQ	Equivalence Class
$B(I_p)$, $B_E(I_p)$, $B_D(I_p)$	Benefits of I_p
F_E, F_D	Benefit computation functions
$U(I_p)$	Update cost of I_p
$C(p, S)$	Cost of evaluating p using S

Table 1. Terminology

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 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.

The XML path index selection problem faces at least three challenges: (i) How to prune out paths that are likely to be unimportant? This is challenging because there may be many distinct paths in XML data sets; (ii) How to efficiently and accurately evaluate the benefits of path indices? This requires an accurate modeling of the costs of the index access methods and the join processing; and (iii) How to efficiently model index interactions and still reduce the number of the index benefit re-computations? This is difficult because the goal is to maximize the overall benefit but do so in a reasonable amount of time.

To address the above problems, the XIST algorithm operates in three phases. The first phase, *candidate path selection*, tackles the problems of choosing candidate paths (CPs). In the second phase, *benefit computation*, the benefits of the candidate paths are evaluated using proposed cost models. If the index interaction were not taken into account, XIST would just choose the top indices with the highest benefits. However, since the index interaction is taken into account, in the third phase, the *configuration enumeration* module chooses the index with the highest benefit in each iteration until the index space constraint is met. In each iteration of this phase, the XIST algorithm recomputes the benefits of only some candidate indices (instead of all candidate indices) to optimize the index selection time.

XIST is designed to adapt to the environment that some inputs may be missing. For instance, if the schema information is not available, XIST skips the schema based candidate path reduction step. If the workload detail is not known, the tool generates a normalized workload distribution using the schema, and assuming that all paths found

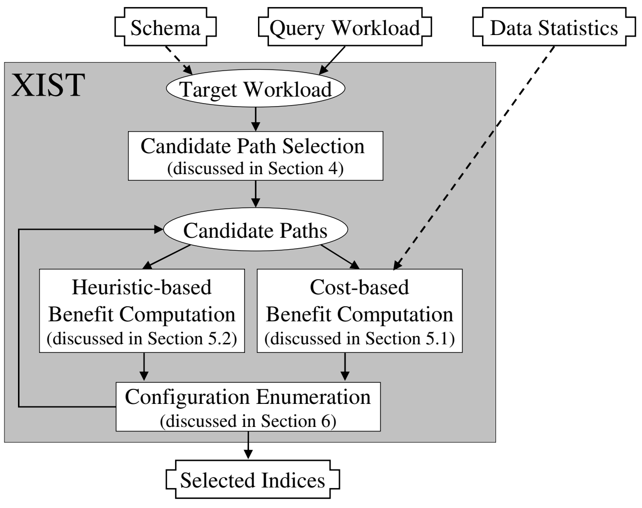

Fig. 3. The XIST Architecture

in the schema are accessed with the same probability. When data statistics are not available, XIST resorts to path length-based heuristics to estimate the benefit of an index.

Figure 4 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).

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 first identifying *a unique path* in an XML schema and then grouping suffix subpaths of the unique path.

Definition 41 *A path p_u, $n_1/n_2/.../n_k$, is a unique path if there is one and only one incoming edge to n_i and there is one and only one incoming edge to node n_{i+1} which must be exclusively from node n_i for $i = 1$ to $k - 1$.*

Since the ending nodes of p_u are the same as the ending nodes of the suffix paths of p_u, the index on p_u returns the same set of nodes that the suffix paths of p_u will (assume that the returned nodes of the path are only the ending nodes of the path).

Definition 42 *Path p_1 and path p_2 are equivalent if p_1 and p_2 share the same suffix subpath and either p_1 is a suffix path of p_2 or p_2 is a suffix path of p_1.*

```
**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.
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. 4. The XIST Algorithm

```
Input: An XML schema graph, $G_s$
Output: Equivalence classes, $EQ$
FindEQs()
1.  for path $p$ that has only one incoming edge
2.      insert $p$ to $EQ(p)$
3.      $n_j$ = the parent of the starting node of $p$
4.      while ($n_j$ has one and only one coming edge)
5.          insert $n_j/p$ to $EQ(p)$.
6.          $p = n_j/p$
7.          $n_j$ = the parent of the starting node of $p$
8.      endwhile
9.  endfor
```

Fig. 5. The Algorithm for Finding Equivalence Classes (EQs)

We refer to a group of equivalent paths as an *Equivalence Class (EQ)*. As an example of an EQ consider the schema shown in Figure 2. A sample EQ 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 example, we refer to `bib/book/publisher` as `bbp`). As shown in the schema graph in Figure 2, `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`.

Instead of building indices on each path in an EQ, XIST only builds an index on the shortest path in each EQ. We choose the shortest path because the space and access time of indices in EQs 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 path equivalent classes are determined based only on the XML schema, these classes are valid for all XML documents conforming to the XML schema. The EQs 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. For example, from the data graph in Figure 1, `first` and `author/first` seem to be equivalent paths since `author/first` seems to be a unique path. However, from the schema graph in Figure 2, `first` is an optional subelement of `editor`, thus there can be an edge coming to `first` from `editor`. Therefore, `author/first` is actually not a unique path, and `first` is thus not equivalent to `author/first`. Therefore, EQs can only be determined by using the schema. Figure 5 presents an algorithm to find the EQs in an XML schema graph.

In the context of XML indices, the equivalence class (EQ) concept is different from the *bisimilarity* [20] and *k-bisimilarity* [17] concepts. Unlike k-bisimilarity, an EQ is defined by an XML schema, not by XML data. Thus, it takes less time to compute EQs than to compute k-bisimilarity since there are usually fewer edges in an XML schema graph than in an XML data graph. Each EQ is a set of paths that lead to the same

```
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. 6. The Index Benefit Computation Algorithm for CPI I_p

destination node; hence it can be represented by the shortest path in the EQ that lead to that node. In each EQ, the starting nodes of the shortest path are $(k_l$-$k_s)$-bisimilar where k_l is the length of the longest path and k_s is the length of the shortest path. For example, for `EQ(p) = {bbp, bp, p}`, nodes matching `p` are 2-bisimilar; `p` is the shortest path ($k_s = 1$), and `bbp` is the longest path ($k_l = 3$).

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 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 6 presents an algorithm for computing the total benefit of I_p, $B(I_p)$.

Figure 6 shows the index benefit computation algorithm which invokes two functions, F_E and F_D, to compute B_E and B_D, respectively. The accuracy of the computed benefits depends on available information. If data statistics are available, XIST computes the benefit by using the cost functions that incorporate the gain achieved via using the index to answer queries and the cost paid for the index update cost, $U(I_p)$. If data statistics are not available, XIST uses heuristics to compute the benefit. The following subsections describe in detail the index benefit computation. We first discuss the cost-based approach and then discuss the heuristic-based approach.

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 We first discuss the cost estimation for retrieving elements and then discuss the cost of evaluating paths with length longer than one. In the following evaluation costs, we assume that element indices and path indices are implemented using a hash index, and that element indices exist.

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,

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

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 [28].

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 equivalence class of p, 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. 7. F_E and F_D for I_p (with Statistics)

The benefit functions F_E and F_D, which are shown in Figure 7, are derived from the difference between the cost of evaluating a path before and after a candidate index is available. In this figure, I_p represents the candidate index on path p. p_e is the path that is in the same equivalence class as p, whereas p_d is a path that is not in the same equivalence class as p, but contains p as a subpath. S is used to represent the set of current selected indices.

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 sections, 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. For example, to compute the number of joins needed to evaluate `book/author(ba)`, we need to find $L^{'}$(`ba`, S). Initially, S contains only element indices, thus the longest indexed subpath of `ba` is the index on `book(b)` or the index on `author(a)`. Therefore, $L^{'}(\texttt{ba}, S) = L(\texttt{ba}) - L(\texttt{b}) = 2 - 1 = 1$. The number of joins required to answer `ba` is one. That is, $F_E(\texttt{ba}, \texttt{ba}, S) = 1$.

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. For example, the index on `ba` can be used to answer `baf`, but only partially. Initially, only element indices exist, the length of an unindexed subpath of `baf` is $L^{'}(\texttt{baf}, S) = L(\texttt{baf}) - L(\texttt{b}) = 3 - 1 = 2$. Since the index on `ba` cannot completely answer `baf`, the benefit of the index needs to be reduced by the cost of the join between `ba` and `f`. This join cost is estimated as the difference between the length of `baf` and that of `ba`. Thus, the benefit of the index for query `baf` becomes $L'(\texttt{baf}, S) - (L(\texttt{baf}) - L(\texttt{ba})) = L'(\texttt{baf}, S) - L(\texttt{baf}) + L(\texttt{ba}) = 2 - 3 + 2 = 1$. That is, $F_D(\texttt{ba}, \texttt{baf}, S) = 1$.

The benefit functions F_E and F_D are shown in Figure 8. In this figure, $L^{'}(p, S)$ represents the length of the shortest unindexed subpath of path p, and $L(p)$ represents the length of the path p.

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

Fig. 8. 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 6), the first two phases of the XIST algorithm (Figure 4) 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 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). We call these categories type-A, type-B, and type-C respectively.

As an example of indices in each of these types, consider the following five candidate indices:

```
book/author (ba),
```

book/author/first (baf),
author/first (af),
author/last (al), and
article/author/last (aal).

Also assume that the queries left to be evaluated are the following paths: `baf` and `af`. Path `baf` is a type-A path with respect to path `ba` because path `baf` contains `ba` as a subpath. On the other hand, path `ba` is a type-B path for path `baf` since path `ba` is a subpath of path `baf`. Path `af` is a type-C path for path `ba` because path `af` is not a subpath of path `ba`, but `af` is a subpath of `baf` which contains `ba`.

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 a lot of time in computing the benefits of many remaining unselected indices. For example, suppose that the index on `ba` yields the maximum benefit out of all candidate indices. Assume that the candidate indices include the paths `baf`, `af`, `al`, and `aal` Assume also that the query workload consists of these following paths: `baf` and `af`. Then, the index on `ba` is chosen first. The naïve approach would then recompute the benefits of the remaining four candidate indices. XIST chooses to recompute only the benefits of the indices on `baf` (type-A) and on `af` (type-C). The index on `ba` affects neither the benefit of the index on `al` nor the benefit of the index on `aal`. XIST uses this property and does not recompute the benefits of these 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 [22] 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 [19], Mondial [27], Shakespeare Plays [14], and XMark benchmark [24]. 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. This set is then partitioned according to the path query length, generating subsets with path queries of equal lengths. These subsets are further partitioned according to whether the query has a value-based predicate. Ten queries are then randomly picked from the subsets using the following criteria. First, a query length between two and the maximum length of the paths in the data set is chosen randomly. Then, with a probability of 0.5, the chosen path is appended with a value-based predicate.

As an example, using this generation method, some of the queries on the Plays data set are shown below:

```
FM/P
/PLAY/ACT/EPILOGUE/SPEECH[SPEAKER="KING"]
/PLAY/INDUCT/SPEECH/SPEAKER
PROLOGUE/SPEECH[SPEAKER="Chorus"]
```

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 equivalence class (EQ) in reducing the number of candidate paths. Next, we present the experimental validation of the cost model used for benefit analysis. Then, we compare the performance of XIST with the performance of a number of alternative index selection schemes. We also show the impact of the different types of inputs (namely query workload, XML schema and statistics) on the behavior of the XIST toolkit. Finally, we analyze the performance of all the index selection schemes when the workload changes over time.

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 Path Equivalence Classes To assess the effectiveness of path equivalence class, we measure the number of paths and the number of path equivalence classes in each data set. Note that paths in an equivalence class are represented by a single unique path which is the smallest path pointing to the same destination node. Therefore, the number of equivalence classes denotes the number of such unique paths.

Figure 9 plots the number of paths and the number of equivalence classes 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 9, the number

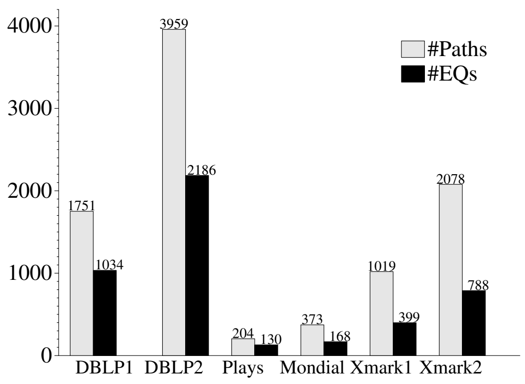

Fig. 9. Number of Paths and Equivalence Classes

Path Index	$\|p\|$	Time (ms.)	K_I
FM/P	148	5	0.0338
SPEECH/SPEAKER	31028	1127	0.0363
SPEECH/LINE	107833	3872	0.0359

Table 2. Path Index Access Cost

of equivalence classes 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.*

Experimental Validation of the Cost Models We validate the cost models presented in Section 5.1 using one of the data sets: the Shakespeare Plays. The results presented in Table 2 indicate a linear relationship between the path cardinality and the path index access time. This linear relationship can be expressed as:

$$T(I_p) = K_I * |p| \tag{4}$$

where $T(I_p)$ is the time taken to retrieve results using an index on p and $|p|$ is the number of nodes matching p. From path cardinalities and times shown in Table 2 and using Equation 4, the value of K_I is approximately 0.04.

We also found that the join cost was proportional to the sum of the sizes of the sorted input lists and the size of the output list. This linear relationship can be expressed as:

$$T(I_{p_1/p_2}) = K_J * (|p_1| + |p_2| + |p_1/p_2|) \tag{5}$$

Using the path cardinalities and times shown in Table 3 and Equation 5, we determine that K_J is approximately 0.09 for the structural join algorithm used in our experiments.

In the remaining experiments we use the values of K_I and K_J that are determined by this experiment.

Comparison of Different Indexing Schemes We then compare the performance of the following sets of indices: indices on elements (*Elem*), indices on paths with length

Path (p_1/p_2)	$\lvert p_1 \rvert$	$\lvert p_2 \rvert$	$\lvert p_1/p_2 \rvert$	Time (ms.)	K_J
FM/P	37	148	148	30	0.09
SPEECH/SPEAKER	31028	31081	31081	9121	0.10
SPEECH/LINE	31028	107833	107833	22789	0.09

Table 3. Path Join Cost

up to two (*SP*), indices suggested by XIST (*XIST*), and indices on the full path query definitions (*FP*). The *Elem* index selection strategy is interesting because it is a minimal set of indices to answer any path query. The *SP* follows the index selection technique recently proposed by Kaushik et al. [16], which prefers to index on short paths over long paths. *FP* is a set of indices that requires no join when evaluating paths without value-based predicates.

The XIST toolkit is provided with information about an XML schema, data statistics, and a query workload. When the workload information is available, all indexing schemes built indices on only elements and/or paths that appear in the workload. When there is no workload information, all indexing schemes, except *FP*, generate indices by using the schema information. In this case, the candidate paths are paths found using the schema.

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 10 shows the performance improvement of *XIST* over other indices for the four experimental data sets. In this figure, the improvement is measured as:

$$\frac{T(I) - T(XIST)}{T(I)}$$

where $T(I)$ is the execution time with the use of the index set *I* for evaluating all queries in the workload.

The results shown in Figure 10 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 path equivalence classes (EQs) while evaluating path queries. In many cases, long path queries 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 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. Table 4 presents the sizes of data sets and indices for all data sets.

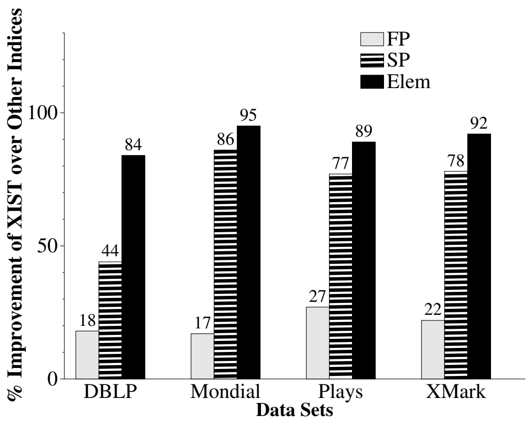

Fig. 10. Performance Improvement of *XIST*

Impact of Input Information on XIST In this section, we present results investigating the behavior of XIST for different combinations of input information. We compared the execution times when using indices suggested by XIST with indices selected by other index selection strategies. When the workload information is available, we only show the performance of *FP* and *XIST* since it is much better than the performance of *Elem* and *SP*. When the workload information is not available, we show the performance of all indexing schemes, except *FP*, since *FP* cannot be generated without the workload information.

Due to space limitations, in this paper we only present the results for DBLP and XMark. 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.

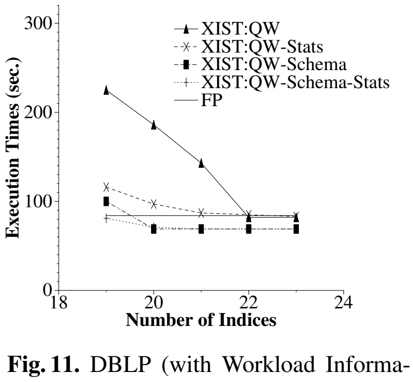

Fig. 11. DBLP (with Workload Information)

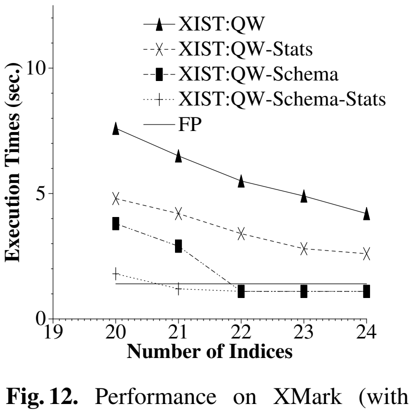

Fig. 12. Performance on XMark (with Workload Information)

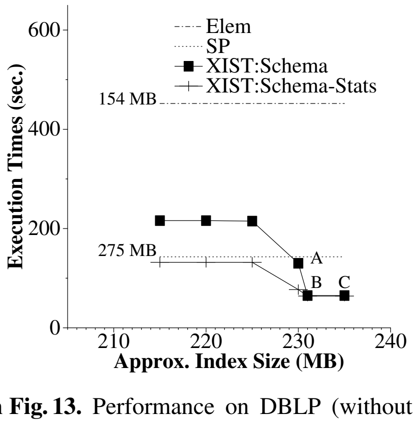

Fig. 13. Performance on DBLP (without Workload Information)

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

Table 4. Sizes of Data Sets and Indices

In the following subsections, we first present experimental results evaluating the index selection strategies when workload information is available. Then, we present results for the case when workload information is not available.

Evaluation when Workload Information is Available

In this section, we compare the performance of the index selection method when the workload information is available. The results for this experiment are presented in Figures 11 and 12. In these figures, the X-axis plots the number of indices that XIST recommends. For this experiment, we use this metric instead of the default metric which is the available disk space (the input parameter k in Figure 4) because it allows us to better evaluate the incremental additional benefit of selecting an additional index.

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 11-12, the number of *FP* indices is used to assign the initial number of indices that the XIST tool generates.

The cost-based benefit evaluation estimates the benefits of indices more accurately than the heuristic-based benefit evaluation. As shown in Figures 11-12, the execution times of *XIST* with `QW-Stats` (`QW-Schema-Stats`) are usually smaller than the execution times of *XIST* with `QW` (`QW-Schema`) at each point of index space. However, as the number of indices increases, the discrepancy becomes smaller because the number of allowed indices is large enough to cover most indices that are critical to reduce the overall workload evaluation time.

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 12. In this graph, comparing the execution times of *XIST* with `QW`, the gap of the execution times between 20 and 21 indices is smaller than the gap of the execution times between 21 and 22 indices. This indicates that with the heuristic-based benefit function, it is possible for a less useful index to be chosen before a more useful index. On the other hand, in the graph of the execution times of *XIST* with `QW-Stats`, the gaps of the execution times for each index increment gradually decrease. This indicates that the chosen index is more useful than other remaining candidate indices when the cost-based benefit function is employed.

Another important observation is that when the number of allowed indices is sufficiently large, the performance of *XIST* with `QW-Schema` and with `QW-Schema-Stats` are about the same, and are better than the performance of *FP*. This indicates that for sufficiently large index space, the input information consisting of the workload and the schema is adequate to achieve the good performance.

Evaluation when Workload Information is not Available

In this section, we compare the performance of the index selection methods when the workload information is *not* available. The results for this experiment are presented in Figures 13 and 14. In these figures, the X-axis plots the approximate total sizes of all the indices in bytes (the default input parameter k in Figure 4). We use this metric because a large incremental number of indices is required to decrease the execution time when the workload information is not available. Often an index selection strategy will prefer not to build an index even when there might be room to fit the index on disk (as the index may have no additional benefit). In these figures, the index sizes for *SP* and *Elem* methods are shown explicitly.

In Figure 13, after point A, when we increased the index space available to build indices for XIST, the size of *XIST* grew at a small rate. The performance of *XIST* is optimal at point B. At point C, XIST has additional index space, but XIST still suggests the same set of indices as the set of indices at point B. In other words, the XIST toolkit chooses only useful indices and these indices take only part of an available index space.

As shown in Figures 13-14, *XIST* and *SP* always outperform *Elem*. Compared to *SP*, *XIST* is more desirable since it allows the system to make the tradeoff between the performance and the size of indices. As the space available for building indices increases, *XIST* starts outperforming *SP*. As shown in these figures, *XIST* indices not only result in better performance, but are often smaller in size than the indices selected by *SP*.

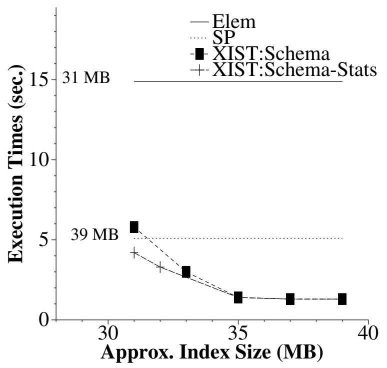

Fig. 14. Performance on XMark (without Workload Information)

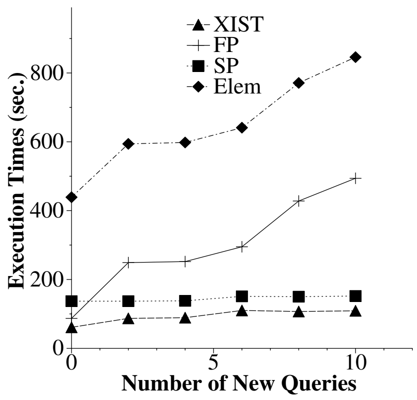

Fig. 15. Performance on DBLP (with Changing Workloads)

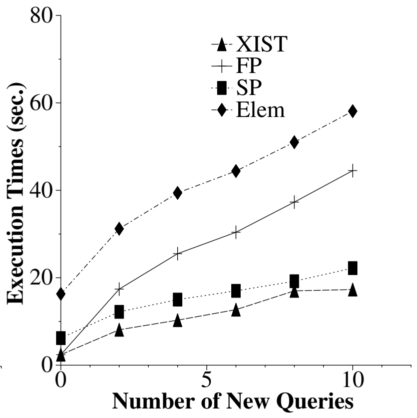

Fig. 16. Performance on XMark (with Changing Workloads)

Impact of Changing Workloads on Performance In the last experiment (corresponding to Figures 13-14), we demonstrated that XIST consistently selects better indices than the other methods even when the user workload is not static. In many situations, a user workload may be available initially, but this workload may change over times as new queries are gradually added. Thus, an index selection tool must gracefully deal with the changing workload environment. In this section, we consider this case of changing workloads.

For this experiment, the initial workload has ten queries. Then, we introduce ten additional random queries to the workload, introducing two queries at at time. Consequently, we have five workloads in addition to the initial training workload. This strategy of using increasing number of queries in the workload allows us to evaluate the effectiveness of the index selection strategies as the workload gradually changes.

For this experiment, we assume that the schema information, data statistics, and the information of the training workload are available as inputs to all indexing schemes. The *Elem* index selection strategy selects all element indices, including the elements that do not appear in the training workload. The *SP* indices include all possible indices on paths with length two, based on the schema. The *FP* indices are indices built on paths that appear in the initial training workload. XIST handles the environment with changing workloads by picking x% of indices based on the initial query workload, and the remaining 100-x% of indices selected assuming that the query workload is not available. In our experiments, x is set to 50.

For this experiment, each index selection strategy recommends indices based on the initial training workload, and these indices are then used for the remaining workloads.

Figures 15 and 16 show the results of this experiment. In these figures, the Y-axis represents the total execution times of executing all queries in the workload, and the X-axis represents the number of new queries in each workload. When the number of new queries is zero, there are ten queries in the workload. When the number of new queries is ten, there are 20 queries in the workload.

As opposed to *FP* and *Elem*, *XIST* and *SP* scale well as new queries are introduced to the workload. As shown in Figures 15 and 16, the query execution times for *FP* and *Elem* grow rapidly as the workload changes. The reason for this rapid increase in execution times is because the evaluation of the new queries requires many additional joins. On the other hand, the execution times when using the *XIST* and *SP* indices only increase gradually as new queries are added to the workload. *XIST* and *SP* perform better because these two techniques identify a common set of paths, which are not too specific and can hence be used to efficiently evaluate the new queries which may have some common subexpressions with the initial training workload.

8 Related Work

Related work in index selection for XML documents spans many areas. This section overviews related work in structural summary generation for semistructured documents, path index design, and index selection.

Dataguides [11] provide concise and accurate summaries of all paths originating from the root node in a semistructured document. While Dataguides assume schema-less data and represent only paths appear in a semistructured document, XIST EQs represent structural summaries of documents using XML schema information.

In [20], Milo and Suciu describe T-indexes, a generalized path index structure for semistructured documents. A particular T-index is associated with a set of paths that match a path template. Their approach uses bisimulation relations to efficiently group together nodes that are indistinguishable w.r.t to the given template into path equivalence classes. If two nodes are bisimilar, they have the same node label and their parents also share the same label. In the 1-index [20], 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 [17] has been proposed to make a trade off between the index performance and the index size. While k-bisimilarity [17] is determined by using XML data, the EQs in this paper are determined by using an XML schema. Although both k-bisimilarity and EQs group paths that lead to the nodes with the same label, EQs group paths in an XML schema but k-bisimilarity group paths in XML data.

Chung et al. have proposed APEX [8], an adaptive path index for XML documents. The main contributions of APEX are the use of data-mining techniques to identify frequently used subpaths, and the implementation of index structures that enable incrementally updates to match the workload variations. 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 [16]. 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 [16]. On the other hand, XIST takes many additional parameters, e.g., not only the path length, to assess the usefulness of the indices. Furthermore, XIST exploits information from a schema or a query workload while the index selection techniques [16] do not take advantage of this information.

Many commercial relational database systems employ index selection features in their query optimizers. For example, IBM's DB2 Universal Database (UDB) uses DB2 Advisor [26], which 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. That is, the indices on fewer columns are considered before indices on more number of columns. If we applied this concept in the index selection for XML query processing, we would miss an index on a long path that is more useful than indices on short paths. XIST does not take this approach, and it also groups a set of paths (a set of multiple-columns) that can share the index. In addition, unlike the DB2 Advisor and the index selection tool for Microsoft SQL Server, XIST can still suggest indices even when there is no query workload information.

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. Using subpaths to answer queries over longer paths has also been used in other database domains, such as in the OLAP optimization that uses aggregated summary tables to answer higher dimensional queries [13]. The key difference between [5] and XIST is that 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

We describe XIST 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 for effectively processing sub-paths of a path expression. Our experimental evaluation on standard XML data sets demonstrated that the indices selected by XIST perform better and also have a smaller size compared to current techniques. In addition, XIST can suggest a useful set of indices in various environments, such as when the workload changes.

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 equivalence classes and efficient benefit recomputation methods, can be adapted for use with other DBMSs with different implementation details and query algorithms. To adapt this general framework to other systems, accurate cost models equations are required that account for the system-specific details. Index selection techniques, like query optimization techniques, rely heavily on the accurate methods of estimating system costs, which are often interesting research topics by themselves. 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 more types of path indices, such as indices on regular path expressions and on twig queries. We also plan on exploring the use of equivalence classes to assist in schema-driven XML query optimization.

References

1. S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. 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. Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, and Donghui Zhang. Efficient Complex Query Support for Multiversion XML Documents. In *EDBT*, pages 161–178, Prague, Czech Republic, 2002.
7. Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, pages 263–274, Hong Kong, China, 2002.
8. C. Chung, J. Min, and K. Shim. APEX:An Adaptive Path Index for XML Data. In *SIGMOD*, pages 121–132, Madison, WI, June 2002.
9. 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.
10. P.F. Dietz and D.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. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, Jerusalem, Israel, August 1997.
12. T. Grust. Accelerating XPath Location Steps. In *SIGMOD*, pages 109–120, Madison, Wisconsin.
13. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD*, pages 205–216, Montreal, CA, June 1996.
14. J. Bosak. The Plays of Shakespeare in XML. `http://metalab.unc.edu/bosak/xml/eg/shaks200.zip`.
15. 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.
16. 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.
17. 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.
18. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, pages 361–370, Roma, Italy, September 2001.
19. M. Ley. The DBLP Bibliography Server. `http://dblp.uni-trier.de/xml/`.
20. 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.
21. L. Poola and J. Haritsa. Sphinx: Schema-conscious Xml Indexing. Technical report, Dept. of Computer Science and Automation, Indian Institute of Science, November 2001.
22. The Apache XML Project. Xerces C++ Parser. `http://xml.apache.org/xerces-c/index.html`.
23. F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *International Workshop on the Web and Databases*, pages 49–54, Santa Barbara, CA, May 2001.
24. A.R. Schmidt, F. Wass, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse. An XML Benchmark Project. Technical report, CWI, Amsterdam, The Netherlands, 2001. `http://monetdb.cwi.nl/xml/index.html`.
25. P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.
26. 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.

27. W. May. The Mondial Database in XML. http://www.informatik.uni-freiburg.de/~may/Mondial/.
28. Y.Wu, J.M. Patel, and H.V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, pages 443–454, Bangalore, India, 2003.
29. 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.