FIX: Feature-based Indexing Technique for XML Documents

Ning Zhang M. Tamer Özsu Ihab F. Ilyas Ashraf Aboulnaga

University of Waterloo

{nzhang,tozsu,ilyas,ashraf}@uwaterloo.ca

ABSTRACT

Indexing large XML databases is crucial for efficient evaluation of XML twig queries. In this paper, we propose a *feature-based* indexing technique, called FIX, based on spectral graph theory. The basic idea is that for each twig pattern in a collection of XML documents, we calculate a vector of *features* based on its structural properties. These features are used as keys for the patterns and stored in a B^+ tree. Given an XPath query, its feature vector is first calculated and looked up in the index. Then a further refinement phase is performed to fetch the final results. We experimentally study the indexing technique over both synthetic and real data sets. Our experiments show that FIX provides great pruning power and could gain an order of magnitude performance improvement for many XPath queries over existing evaluation techniques.

1. INTRODUCTION

Management of XML data, especially the processing of XPath queries, has been the focus of considerable research and development activity over the past few years. Indexing techniques are crucial for efficiently answering queries in a large database consisting of collections of XML documents. Given a query specified by a path expression, a query processor needs to find the instances of the documents or substructures thereof that satisfy the value and structural constraints specified in the query. Appropriate indexing techniques can significantly improve the performance of this matching operation.

A large body of XML indexing research focuses on *structural indexes* [18, 15, 14, 24]. These techniques cluster XML element nodes based on their structural similarity, with the objective of obtaining better locality and hence better query performance. Depending on different definitions of structural similarity, the types of queries that these indexes can answer varies.

In this paper, we propose a novel indexing technique, FIX, that can answer a large subset of twig queries. We show that FIX is superior to previous techniques in answering these queries, especially on structure-rich XML documents. In addition to the element nodes that are usually handled by structural indexes, FIX treats values (or PCDATA) in the XML documents as special tree nodes after certain manipulation. Hence, FIX is a unified technique that handles both the values and the tree structures.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

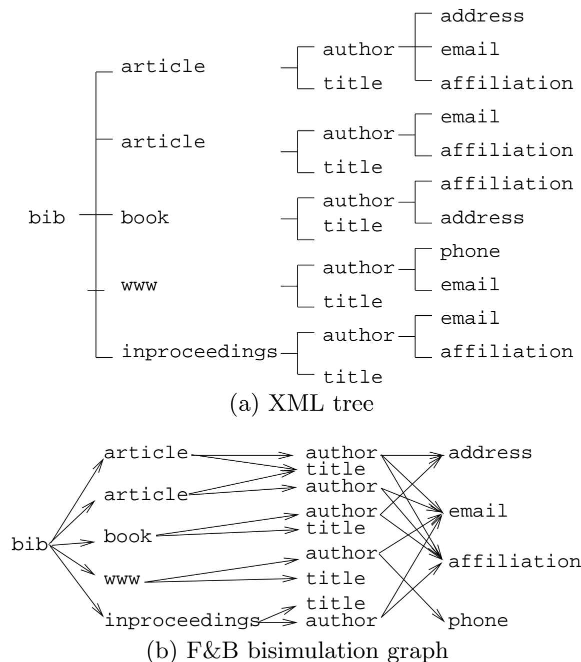
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

1.1 Motivation

Most of the recently proposed structural indexes are based on the idea of clustering similar XML nodes. While structural clustering is effective for data sets that conform to a regular schema (e.g., an `order` always has an `order_id` and `ship_date`), the index could grow remarkably large for structure-rich data sets. The index lookup operator, whose performance is largely dependent on the size of the index, is therefore inefficient on these indexes. To illustrate the problem, Figure 1 shows a bibliography XML document and its clustering index—F&B bisimulation graph. In this data set, all types of publications (`article`, `book`, etc.) have a child element `author`, which may have any combination of subelements `address`, `email`, `phone`, and `affiliation`. Since each `author` element has a different parent or set of children, the `author` elements are incompressible in the F&B bisimulation graph. For a structure-rich data set such as Treebank [1], the F&B bisimulation graph has more than 3×10^5 vertices and 2×10^6 edges. Although particular storage structures are developed to materialize F&B bisimulation graphs on disk (see, e.g., [24]), updating as well as searching in such a large graph could be very expensive. This issue is not specific to the F&B index that structure-rich data generate large graphs, but common to all structural clustering techniques.

The key insight of our proposed index structure, FIX, is to break a large document into small pieces of substructures (which we call *twig patterns*) to achieve high pruning power without searching the whole graph. Our approach is to enumerate all twig patterns in the document and map each of them into a vector of *features* (or *structural characteristics*). The feature vector is a signature of a twig pattern and serves as a key to record the twig pattern in a mature index such as B^+ tree. In the query phase, the features of the query are computed and candidate twig patterns that conform to these features can be quickly retrieved from the index without exploring the whole search space. A following refinement step may be required to obtain the final results.

Using this approach, answering a twig query amounts to looking up a vector key in the B^+ tree. However, two challenges arise: (1) what is the appropriate features of the twig



(a) XML tree

(b) F&B bisimulation graph

Figure 1: An bibliography document and its F&B bisimulation graph

patterns, and (2) how to deal with the fact that the number of twig patterns is exponential to the size of the graph? These two questions are correlated in that if the number of patterns is small, we can index all of them and use their string representations[1] (or the hash codes thereof) as the keys. However, in the general case, when the number of patterns is large, we have to choose a subset of them to index. In this general case, the string representation of a twig pattern is no longer a valid key, since when index lookup for a query pattern fails, we do not know whether the pattern is in fact not in the database, or it is just missed in the index. Accordingly, the pivot of this approach relies on finding the desired features.

We propose a set of features based on spectral graph theory, and prove that they satisfy the *no-false-negative* requirement: by examining only the query and the keys in the index, we are able to fix a *complete* set of candidate twig patterns that may produce results. We do not require *no-false-positives*, i.e., all candidates produce results, since this can be handled by a further refinement step. Therefore, FIX is a pruning index that can be built on top of an existing XPath query processor to achieve better query performance.

1.2 Contributions

In summary, our contributions are as follows:

- We propose a framework based on a novel feature-based index to evaluate a subset of path expressions (defined in Section 2.1). In such a framework, the features are first extracted from the query and they are used as keys to look up candidates in the index. A refinement step is then applied to the candidates to find the final results.
- We propose an index construction algorithm that efficiently enumerates a subset of the twig patterns in an XML tree. The enumerated subset is *complete* in that if the query pattern is not a subpattern of any of the index patterns, it is not a subpattern of the original XML tree either.
- We propose a novel set of features for twig patterns, which are used as keys for the enumerated indexed patterns and query patterns. We show that this set of features are appropriate to be used to prune the index, and they do not introduce false-negatives. To the best of our knowledge, FIX is the first XML indexing technique using feature-based pruning.
- We propose a natural and effective way to index values in FIX. Integrating values into the structural index eliminates the need for two index look-up operations and intersection of the temporary results.
- We experimentally show that in many cases FIX can improve the performance of twig queries by orders of magnitude against the state-of-the-art twig query evaluation operators.

The rest of the paper is organized as follows: in Section 2 we provide the general background of the paper. In Section 3, we introduce how to translate a twig pattern into a matrix and prove certain properties of the eigenvalues of the matrix. In Sections 4 and 5, we present the index construction algorithm and index query algorithm, respectively. In Section 6, we present an experimental study of FIX. We present related work in Section 7 and conclude in Section 8.

2. BACKGROUND

We assume that readers have a basic knowledge of XML data model and XPath path expressions. Chamberlin [7] provides a concise yet comprehensive introduction.

2.1 Twig Queries and Matches

FIX can handle a subset of path expressions called *twig queries*. The term "twig query" is defined slightly differently in different papers in the literature. Our use of the term conforms to the following definition.

Definition 1. A *twig query* is a path expression whose axes could only be /, except for the first axis which could be //. Moreover, there is no KindTest in the expression and no value-based comparisons inside the predicates. A *twig query with value* relaxes the restriction by allowing equality conditions between attribute or element names and atomic values in the branching predicates.

We will focus on the indexing technique for the twig queries first and then extend it to handle twig queries with values in Section 4.6. Section 5 gives the direction on how to extend FIX index to handle //-axes in the middle of the query. Detailed study and experimental evaluation of this extension is a subject of our future work.

A twig query can be thought of as a tree[2] in which each step corresponds to a node in the tree, and the first step is connected to a special *root* node. The axes are translated into edges in the tree. Based on the tree representation, we now define the notion of *existential match* (or simply *match*) between a twig query and an XML tree.

[1] It is well known that a tree can always be translated into a parenthesized string representation (e.g., `(author(phone)(email)))` and vice versa.

[2] We denote the label of node x as $label(x)$.

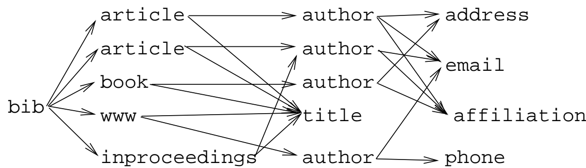


Figure 2: A bisimulation graph of the bibliography document in Figure 1(a)

Definition 2. A twig query Q *matches* an XML tree X if there exists a mapping f from the NameTests of Q to the nodes in X such that the following hold:

- the root of the twig query always matches the document node (parent of the root node in the document).
- for any NameTest $q \in Q$, $label(q) = label(f(q))$.
- if two NameTests u and u' are connected by an axis $\alpha \in \{$"/", "//"$\}$, then $f(u)$ is a parent (or ancestor) of $f(u')$ if $\alpha =$"/" (or "//").

Match does not specify which XML nodes should be returned, therefore it is used for existential testing.

2.2 Bisimulation Graph

Given an XML tree, there is a unique (subject to graph isomorphism) minimum bisimulation graph that captures all structural constraints in the tree. The bisimulation graph defined in this paper is based on the bisimilarity[3] notion defined by Henzinger et al. [12].

Definition 3. Given an XML tree $T(V_t, E_t)$ and a labeled graph $G(V_g, E_g)$, an XML tree node $u \in V_t$ is bisimilar to a vertex $v \in V_g$ ($u \cong v$) if and only if all the following conditions hold:

- u and v have the same label.
- if there is an edge (u, u') in E_t, then there is an edge (v, v') in E_g such that $u' \cong v'$.
- if there is an edge (v, v') in E_g, then there is an edge (u, u') in E_t such that $v' \cong u'$.

Graph G is a bisimulation graph of T if and only if G is the smallest graph such that every vertex in G is bisimilar to a vertex in T.

It is easy to see that the bisimulation graph of a tree is a directed acyclic graph (DAG). Otherwise, if the bisimulation contains a cycle, the tree must also contain a cycle based on the definition.

The bisimulation graph of the XML tree in Figure 1(a) is shown in Figure 2. The difference between the bisimulation graph and the F&B bisimulation graph is that the former requires that two nodes in the XML tree belong to the same equivalence class if their subtrees are structurally equivalent. The bisimulation graph does not require that the two indexing vertices have similar ancestors, but the F&B bisimulation graph does. Consequently, the bisimulation graph clusters the two `author` vertices from `book` and `inproceedings` into one equivalence class.

[3]We use *bisimilarity* to denote the relation between XML nodes and index vertices; and use *bisimulation* graph to denote the resulting index graph after the bisimilarity mappings.

The tree representation of a twig query can always be translated into a bisimulation graph. We call this bisimulation graph the *twig pattern*. Similar to the twig query, we can also define matching twig patterns on a bisimulation graph of an XML tree.

2.3 Matrices and Eigenvalues

An undirected unlabeled graph G with n vertices can always be represented as an $n \times n$ matrix (e.g., adjacency matrix or Laplacian matrix). Given an $n \times n$ matrix $\mathbf{M}$, there exist a column n-vector $\mathbf{v}$ such that

$$\begin{aligned} \mathbf{M} \cdot \mathbf{v} &= \lambda \mathbf{v} \\ \langle \mathbf{v}, \mathbf{v} \rangle &= 1 \end{aligned}$$

where λ is a scalar, and $\langle \mathbf{v}, \mathbf{v} \rangle$ is the inner product of two vectors, which is defined as $\langle \mathbf{v}, \mathbf{v} \rangle = \mathbf{v}^{\mathrm{T}}\mathbf{v} = \sum_{i=1}^{n} v_i * v_i$, for $\mathbf{v} \in \mathbb{R}^{n \times 1}$; or $\sum_{i=1}^{n} \overline{v_i} * v_i$ for $\mathbf{v} \in \mathbb{C}^{n \times 1}$, where $\overline{v_i}$ is the complex conjugate operator. The $\mathbf{v}$ and λ are called the normalized eigenvector (or simply eigenvector) and eigenvalue of $\mathbf{M}$, respectively. The eigenvectors need to be normalized since otherwise there is an infinite number of eigenvalues that are obtained by scaling the eigenvectors. For an $n \times n$ matrix, there are a total of n such eigenvector and eigenvalue pairs, but they may not be distinct. The eigenvalues are usually denoted by $\lambda_1, \ldots, \lambda_n$ ordered by their magnitude in descending order. Throughout the rest of the paper, we denote the maximum and minimum eigenvalues as λ_{max} and λ_{min}, and denote $\lambda(G)$ of graph G as the eigenvalue of the matrix representation of G whenever there is no possibility of confusion.

There is a well-know property about two graphs and their eigenvalues [5], and it is the basis for our structural feature selection.

THEOREM 1. *Let G and H be two undirected unlabeled graphs, and $\mathbf{M}_G$ and $\mathbf{M}_H$ be their adjacency matrices. If H is an induced subgraph of G, then $\lambda_{min}(\mathbf{M}_G) \leq \lambda_{min}(\mathbf{M}_H) \leq \lambda_{max}(\mathbf{M}_H) \leq \lambda_{max}(\mathbf{M}_G)$.*

We will prove in Section 3.3 that a similar theorem also holds for labeled directed graphs after a certain translation from the graph to matrix.

3. FEATURES AND THEIR PROPERTIES

Given a twig pattern (represented as labeled directed graph), we want to identify the distinctive characteristics of the structures contained in it. We call these characteristics *features* of the pattern. Features can be used as a *key* to index and retrieve those instances that match a pattern. In FIX, the features are based on a subset of eigenvalues of the matrix representation of a pattern. Eigenvalues have the desired property that they allow us to prune the index search space without losing any results.

Before introducing how to obtain features, we first set down the foundations of using bisimulation graph as a tool to test the existence (match) of a pattern. This is necessary because bisimulation graphs are the input to calculating the features — eigenvalues. That is, we first prove that the match of a twig pattern on a bisimulation graph is equivalent to the match of its twig query on the XML tree. This means that the bisimulation graph preserves all the structural information required for existential matching. The reason we use twig patterns and bisimulation graphs rather than their

corresponding tree structures is that the trees contain many structural repetitions and are too large to extract features (eigenvalues) from.

In the following subsections, we introduce the translation from a labeled directed graph to a matrix and prove that a similar result to Theorem 1 also holds for the matrix representation, which means that λ_{min} and λ_{max} could be used as a valid tool for pruning. Finally, we will introduce other features that give us additional pruning power.

3.1 Structure Preservation

We first prove that the bisimulation graph preserves all structural information needed for matching, through the following structural preservation theorem.

THEOREM 2. *A twig query Q matches an XML tree X if and only if the twig pattern Q' matches the bisimulation graph X'.*

The proof is quite straightforward after realizing that matching and bisimilarity are homomorphisms on the edge relation. We leave the full proof to the full version of the paper [30] due to space limitations.

This theorem seems contradictory to the fact that the F&B bisimulation graph is the smallest covering index for twig queries [14] and bisimulation graph is smaller than the F&B bisimulation graph. The reason is that here the "structural preservation" is defined for testing pattern existence (the notion of match) and the "covering" in F&B bisimulation is defined in terms of query answering (which needs more information than existential testing). In fact, the bisimulation graph shown in Figure 2 can not answer the query `//inproceedings[author]` since two authors from `inproceedings` and `book` are grouped into one equivalence class. But this graph is sufficient to answer the existence of `authors` under `inproceedings`.

Having the structural preserving property, we can now use the twig pattern and bisimulation graph of an XML document as the subject of querying and indexing instead of twig query and XML tree.

3.2 Anti-symmetric Matrices for Twig Patterns

Given a labeled directed graph (twig pattern), we want to translate it into a matrix such that the matrix preserves as much structural information of the graph as possible. By structural information, we mean the labels of the vertices and the edge relations (here the orientations of the edges are important). Ignoring either of them makes the matrix unrepresentative, and, therefore, reduces the pruning power of any method based on this matrix representation.

To record the vertex label information in the matrix, we assign a distinctive weight to each edge according to the labels of the two incident vertices. This is a one-to-one mapping, therefore it is always possible to translate the weighted directed graph back to the original labeled directed graph.

To preserve the direction information, we represent the directed weighted graph as an *anti-symmetric* matrix (a.k.a. *skew-symmetric* matrix) as follows: we number each vertex v arbitrarily from 1 to n and map it to a dimension in the $n \times n$ matrix $\mathbf{M}$. The reason that we can assign arbitrary numbers (dimensions) to vertices is that any assignment can be permuted to some other assignment (and the permutation results in an isomorphic graph), which is equivalent to permutation of the matrix. It is well known that the eigenvalues of a matrix remain invariant under matrix permutation [10].

If an edge (v_i, v_j) has weight $w_{i,j}$ after the above edge-label-to-integer translation, we assign $\mathbf{M}[i,j] = w_{i,j}$ and $\mathbf{M}[j,i] = -w_{i,j}$. If (v_i, v_j) is not an edge, $\mathbf{M}[i,j] = \mathbf{M}[j,i] = 0$. In this anti-symmetric matrix, the diagonal elements $\mathbf{M}[i,i]$ are always 0 for an acyclic graph. The reason that we put the negative weight at $\mathbf{M}[j,i]$ is that triangle matrices with all $\mathbf{M}[i,i] = 0$ have the same set of eigenvalues $[0, 0, \ldots, 0]$ (matrices having the same eigenvalues are called *isospectral*). A non-zero anti-symmetric matrix is guaranteed to have at least one non-zero eigenvalue [10].

Two anti-symmetric matrices are *isospectral*, if one can be transformed to the other by a non-singular transformation, that is one anti-symmetric matrix can be obtained by multiplying the other anti-symmetric matrix with a non-singular matrix (a matrix that has an inverse). If it is common that two anti-symmetric matrices are isospectral but non-isomorphic, the pruning power will be small. Given that the number of distinct edge label encodings is small in most XML databases and given the requirement of $\mathbf{M}[i,j] = -\mathbf{M}[j,i]$ for anti-symmetric matrices, the probability of two anti-symmetric matrices being isospectral but non-isomorphic is expected to be very small.

As an example, the bisimulation graph in Figure 2 is translated into the following 15×15 matrix (because there are 15 vertices in the graph), where the edges (`bib`, `article`) and (`article`, `author`) are mapped to 2 and 12, respectively:

$$\mathbf{M} = \begin{bmatrix} 0 & 2 & \ldots & 0 \\ -2 & 0 & \ldots & 12 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & -12 & \ldots & 0 \end{bmatrix}$$

3.3 Eigenvalue Containment Property

Given the pairs of λ_{min} and λ_{max} of two anti-symmetric matrices, we prove a similar result to Theorem 1.

THEOREM 3. *Let G and H be two DAGs, and $\mathbf{M}_G$ and $\mathbf{M}_H$ be the anti-symmetric matrix representations of G and H respectively. If H is an induced subgraph of G (which means H is isomorphic to a subgraph of G with the isomorphic mapping f and for every edge (u, v) in H, there is an edge $(f(u), f(v))$ in G such that their weights are the same), then $\lambda_{min}(G) \leq \lambda_{min}(H) \leq \lambda_{max}(H) \leq \lambda_{max}(G)$.*

PROOF. Due to space limitations, we only sketch the proof here and give the full proof in [30].

Since a similar theorem holds for a symmetric matrix (adjacency matrix for undirected graphs), the idea of our proof is to convert the anti-symmetric matrix to a (somewhat) symmetric matrix and use the same proof idea for symmetric matrix in the anti-symmetric case. The rationale of the conversion is based on the fact that the anti-symmetric matrix has some degree of "symmetry" in that $M[i,j]$ and $M[j,i]$ only differ by a negation. In fact, if we multiply the imaginary unit $i = \sqrt{-1}$ with the matrix, we get a *Hermitian* matrix $i\mathbf{M}$, which is a symmetric matrix equivalent in the complex domain $\mathbb{C}^{n \times n}$. What remains is to prove: (1) $\lambda(\mathbf{M}) = -i\lambda(i\mathbf{M})$; and (2) eigenvalue containment property holds for the Hermitian matrix. The first is straightforward by definition of eigenvalues, and the second is very similar to the proof of Theorem 1. □

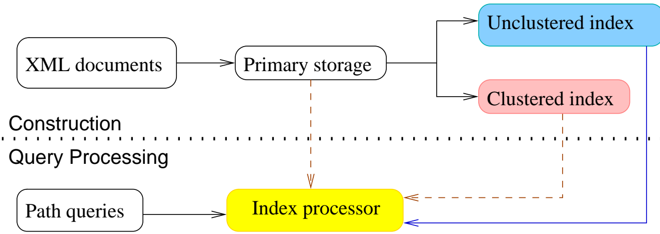


Figure 3: Building and querying indexes

This eigenvalue containment property allows us to choose λ_{min} and λ_{max} as two features to index. Testing for possible matching amounts to checking eigenvalue containment.

Computational Cost: Eigenvalue computation for Hermitian matrix is $O(n^3)$, where n is the number of vertices in the bisimulation graph [19]. Since the twig patterns are usually very small and we break the large bisimulation graph for XML tree into small ones in the index construction step, the real-world computation cost is very efficient—sub-millisecond for a dense 10×10 matrix and sub-second for a dense 300×300 matrix on a PC with Pentium IV 3GHz. Eigenvalue calculation for sparse matrices (which are generated by most bisimulation graphs) should be even more efficient.

3.4 Other Features

In addition to eigenvalues of patterns, there are other possible features that can further increase the pruning power. For example, the root label of the twig pattern or bisimulation graph. It can easily be included in the key to be indexed in the B$^+$ tree. Any bisimulation graph in the index that satisfies the eigenvalue range containment requirement but whose labels do not match with the twig pattern will also be pruned.

Other features may qualify as well, but in this paper we use the set of $\{\lambda_{min}, \lambda_{max}, \text{root label}\}$ as features, and they are the keys of the B$^+$ tree index described in the next section. The pruning criteria is that the indexed eigenvalue range does not contain the query eigenvalue range, or the root labels do not match.

4. INDEX CONSTRUCTION

The overall architecture of constructing and querying the index is depicted in Figure 3. In this section, we concentrate on the construction of FIX and leave the query processing discussion to the next section.

First we give two alternative index types: clustered and unclustered. Then we show their construction algorithm.

4.1 Types of Indexes

As in the relational case, we can build a clustered or unclustered index. Unlike relational databases, the clustered index for FIX incurs storage overhead due to the redundant storage of subelements as explained later. In both cases, the keys of the B$^+$ tree are the features but the "values" are different. In the unclustered index, the values are the references/pointers to the primary data storage (see Figure 4). The advantage of unclustered index is that the primary storage does not need to be changed, and there is very small overhead for building the B$^+$ tree with pointers as the data entries. However, query processing may suffer from the fact that it needs to follow many pointers to perform the query refinement phase, which usually incurs random I/Os.

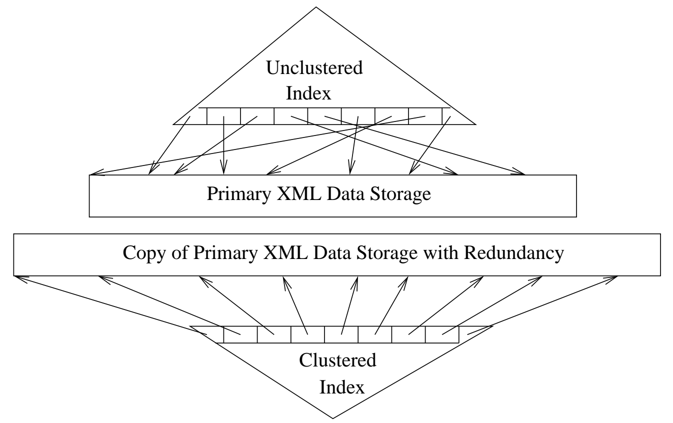


Figure 4: Clustered and Unclustered FIX Indexes

On the other hand, we can build a clustered index by copying the contents in the primary storage pointed by the pointers and store them sequentially according to their feature keys (see Figure 4). This is different from the relational case since we cannot reorder data units in place. The reason is that the data units in the XML case are subtrees and one may contain another as a descendant. Therefore, in order to make the value sorted in the same order as the keys, the clustered index has to copy each subtree to another storage, which may incur large space overhead. Therefore, there is a tradeoff between the storage overhead and performance in the query refinement step.

Unclustered indexes are easier to build and they are the only choice if data has to be ordered on other criteria. They may be useful when the selectivity of the typical queries is high so that few pointers are produced as candidates. On the other hand, clustered index could provide better performance because the I/O are essentially sequential. In the case where the database consists of a large collection of relatively small documents and each of them are inserted into the database as an entry, the clustered index may be the right choice because we can reorder the documents so that their order coincides with the order of their feature keys. Furthermore, there is no redundancy in the storage since every document is treated as a unit. Therefore, the clustered index does not need to keep a copy of the primary storage and incurs no space overhead.

4.2 Index Construction for Collections of Documents

The index construction algorithm takes a collection of XML documents as input, and inserts them into a B$^+$ tree index. The algorithm works in two phases: in the first phase, it generates *indexable units* that are small enough to efficiently extract features from. An indexable unit could be a small document in the collection, or a substructure of a large document. In the second phase, the features of the indexable units are computed and inserted into the B$^+$ tree.

The index construction procedure is codified as the method

CONSTRUCT-INDEX in Algorithm 1, where input C is a collection of XML documents (possibly singleton), L is the depth limit, and I is a B^+ tree that holds the index entries. The depth limit is a parameter for a document being qualified as an indexable unit. In the following subsection, we first introduce how to index an indexable unit, and we introduce how to handle large document in the subsequent subsection.

4.3 Construction of an Index Entry for a Small Document

Each small document whose depth is no larger than the depth limit (an application-dependent threshold) is treated as a unit and converted into a bisimulation graph, which, in turn, is translated into an anti-symmetric matrix. For each of these matrices, we calculate the eigenvalues and use the λ_{max} and λ_{min} together with the root label of the document as the key to be inserted into the B^+ tree. The entry inserted into the B^+ tree is the document itself if we want to build a clustered index, or the pointer to the primary storage for an unclustered index. This process is codified in the CONSTRUCT-ENTRIES method in Algorithm 1. The third parameter of the methods has to be set to 0 indicating that the document does not need to be partitioned.

In the input to the CONSTRUCT-ENTRIES method, X is the input event stream, and H is a B^+ tree index. Parameter L is the pattern depth limit and is set to 0 in this case. The variable G (line 1) is of type BisimGraph, which is a data structure that contains two substructures: the root of the bisimulation graph and a mapping from a *signature* to a vertex in the bisimulation graph. It also maintains the maximum depth of the bisimulation graph. The signature is a data structure that uniquely identifies a vertex. It consists of the vertex label and a set of child vertices. Two XML nodes are in the same equivalence class (bisimulation vertex) if and only if their signatures, namely, labels and children are the same by the definition of bisimilarity.

CONSTRUCT-ENTRIES works in a SAX-like event-driven paradigm: Whenever an open event (corresponding to encountering an open tag when parsing the XML document) is received, a new signature is created and initialized with its label and an empty set of child vertices (line 5). The pair of signature and pointer to the primary storage corresponding to the event is pushed onto a stack *PathStack* (line 6). This pair is popped whenever the corresponding closing event (corresponding to a closing tag) is received (line 8). Since at this time, all children (and their descendants) corresponding to the current event have been visited and their corresponding bisimulation vertices are recorded in the signature that is popped from the stack (line 17), we need to look up the mapping maintained in G to see if the signature already exists (line 9). If it is not in the mapping, then we need to create a new bisimulation vertex u and insert all bisimulation vertices maintained in the signature into u's children list, and then record the new mapping from the signature to u in G (lines 11–13). If the signature is already in the bisimulation graph, we only have to release the memory acquired for the signature. If the *PathStack* is not yet empty (which means the whole tree has not been traversed), we need to update the children list of u's parent in the *PathStack* (lines 16–17); otherwise, we set u as the root of graph G and call BTREE-INSERT to update the database. $G.dep$ is the maximum depth of the bisimulation

Algorithm 1 Constructing FIX for a Collection of Documents

```
CONSTRUCT-INDEX(C : Collection, L : int, I : BTree)
1  for each XML document d ∈ C
2    do if the depth of d ≤ L then
3           CONSTRUCT-ENTRIES(I, d, 0);
4       else CONSTRUCT-ENTRIES(I, d, L);

CONSTRUCT-ENTRIES(H : BTree, X : EventStream, L : int)
 1  G ← empty graph;     ▷ G is of type BisimGraph
 2  PathStack ← empty stack;
 3  while X generates more event x
 4    do if x is an open event then
 5              sig ← ⟨x.label, ∅⟩;    ▷ c_set initialized to ∅
 6              PathStack.push(⟨sig, x.start_ptr⟩);
 7        elseif x is a closing event then
 8              ⟨sig, start_ptr⟩ ← PathStack.pop();
 9              u ← lookup sig in G;
10              if sig is not in G then
11                    create vertex u with label x.label;
12                    create edge (u, v_i) for each v_i ∈ sig.c_set;
13                    create mapping sig ⇒ u in G;
14              else release sig;
15              if PathStack is not empty then
16                   p_sig ← PathStack.top().first;
17                   p_sig.c_set ← p_sig.c_set ∪ {u}
18              else G.root ← u;
19                   if L = 0 then
20                         BTREE-INSERT(H, u, G.dep, start_ptr);
21              if L > 0 then
22                    GEN-SUBPATTERN(H, v, L, start_ptr);

BTREE-INSERT(H : BTree, u : BisimVertex, L : int, ptr : StoragePointer)
1  if u.eigs is not set then
2         convert u into matrix M up to depth L;
3         ⟨λ_max, λ_min⟩ ← EIG-PAIR(M);
4         u.eigs ← ⟨λ_max, λ_min⟩;
5  k ← ⟨u.eigs, u.label⟩;
6  if H is a clustered index then
7         v ← pattern instance from the primary storage
              following ptr;
8         insert v in H with key k;
9  else   insert ptr in H with key k;

GEN-SUBPATTERN(H : BTree, v : BisimVertex, L : int, ptr : StoragePointer)
1  if v.eigs is set then
2         BTREE-INSERT(H, v, 0, ptr);
3  else   Tr ← BISIM-TRAVELER(v, L, ptr);
4         CONSTRUCT-ENTRIES(H, Tr, 0);
```

graph, which indicates that the whole graph should be indexed.

The BTREE-INSERT method is fairly straightforward: it first checks whether or not the bisimulation vertex is associated with a pair $\langle \lambda_{max}, \lambda_{min} \rangle$. If not, it converts the graph into an anti-symmetric matrix under the depth limitation, calculates the eigenvalue thereof, and associates the $\langle \lambda_{max}, \lambda_{min} \rangle$ pair with u (lines 2–4). Then it uses the pair and the root label as a key and inserts the pointer in the B^+ tree for the unclustered index. If the index is a clustered index, we need to retrieve the XML documents from the primary storage and store them as values of the B^+ tree.

Complexity: CONSTRUCT-ENTRIES is a single-pass algorithm that reads each incoming event once. For each closing event, the algorithm searches the bisimulation graph for signature, which could be $O(1)$ using an efficient hashing

method. Therefore, the CPU cost of the construction algorithm is $O(n+m)$, where n is the number of events generated from the input event stream (in case of XML SAX-event stream, it is the number of XML elements in the whole collection), and m is the number of vertices in the bisimulation graph.

The major cost of Algorithm 1 is the I/O cost, which depends on the number of B^+ tree insertions and number of reads from the primary storage. In the unclustered case, the number of B^+ tree insertions is the same as the number of documents in the collection since we generate only one bisimulation graph for each document. In the clustered case, the B^+ tree I/O is the same as the unclustered case but there is additional I/O cost for retrieving documents from the primary storage, which is proportional to the number of documents in the collection as well. In summary, the I/O cost is $O(N)$ where N is the number of documents in the collection.

4.4 Construct Entries for a Large Document

The bisimulation graph of a large document could be very large. Furthermore, no substructures in the large document can be pruned if it is indexed as one entry. Therefore, we need to enumerate subpatterns inside the document tree and populate the instances into the B^+ tree. If the database consists of multiple large documents, we need to enumerate subpatterns for each of them.

First, we need to restrict the subpattern size before enumerating its instances in the XML tree. Based on the same idea of local similarity in prior works [15, 8], we limit the depth of subpatterns to a small number k (k-patterns). With this construct, however, the index loses some expressive power: it can only answer a twig pattern up to depth k. The tradeoff between expressive power and efficiency is common [15] and does not invalidate the benefit of building the index. It is easy for the query optimizer to test whether a twig query is covered by an index.

The method for index construction with limited pattern depth is the CONSTRUCT-ENTRIES method in Algorithm 1, with a positive argument L as the depth limit. The CONSTRUCT-ENTRIES needs to call GEN-SUBPATTERN to enumerate subpatterns given the root of the subpattern and depth L. The GEN-SUBPATTERN method is based on the idea that we can create a bisimulation graph "traveler" (BISIM-TRAVELER) that traverses the bisimulation graph in depth-first order within the depth limit L. During the traversal, it generates an open event when traversing to another vertex, or a closing event when it finishes traversing the subtree of the node or when it traverses to a depth of L. This stream of events can, in turn, be fed to the CONSTRUCT-ENTRIES method. The depth limit in the call to CONSTRUCT-ENTRIES is set to 0 whenever we need to index the whole subpattern. The method will generate a new bisimulation graph that is a subgraph of the original one, and store it into the B^+ tree as described in Section 4.3. To guarantee that the subpattern enumeration process is performed only once for each bisimulation vertex, we also associate the bisimulation vertex with the $\langle \lambda_{max}, \lambda_{min} \rangle$ pair of the subpattern, indicating that this vertex has already been enumerated and the eigenvalues are calculated (line 1).

The reason that we need to go all the way to define a traveler and call CONSTRUCT-ENTRIES again instead of using the subgraph beginning at the current vertex v is that the subgraph itself usually is *not* a bisimulation graph. The limit on the depth causes the subgraph to contain some repetitions such that the subgraph is no longer a bisimulation graph. For example, in Figure 2, the subgraph of depth 2 rooted at `bib` is not a bisimulation graph since `article` is repeated twice.

We use the following theorem to derive the cost of the enumeration algorithm and to prove the completeness of the index. Detailed proof is left to the full version [30].

THEOREM 4. *For an index with positive depth limit, the number of subpattern instances that are enumerated by the function* CONSTRUCT-INDEX *in Algorithm 1 is exactly the same as the number of elements in the document.*

Complexity: The CPU cost of building the index with positive depth limit is the same as the cost for building the index on the collection of small documents, except that there is the additional cost for enumerating subpatterns. For each vertex in the bisimulation graph, the subpattern rooted at this vertex is enumerated once, therefore the additional CPU cost is the same as the number of vertices in the bisimulation graph. Therefore, the CPU cost is $O(n+m)$ where n is the number of XML elements and m is the number of vertices in the bisimulation graph.

The I/O cost is dependent on the number of pattern instances generated, i.e., number of elements in the XML document. For each pattern instance, there is a B^+ tree insertion operation, and for clustered index, there is an additional read operation in primary storage. Therefore, the I/O cost is $O(n)$, where n is the number of XML elements.

4.5 Completeness of Index Construction

We show that the index constructed in the previous subsections is *complete* for any k-pattern query, if the depth limit of the index is at least k.

THEOREM 5. *If the index is built with depth limit at least k (in the case where depth limit is 0 for collection of small documents, k is the maximum depth of the all documents in the collection), a k-pattern is not contained in the XML document, if it is not contained in the index.*

The proof is quite straightforward and given in the full version [30].

4.6 Supporting Value Equality Predicates

FIX supports value-based equality predicates such as the query `//article[author = "John Smith"]/title`. We note that the PCDATA in the XML documents, as well as the atomic value "John Smith" in the query, can be thought of as "labels" of the text nodes, which are children of element nodes. However, we cannot directly use the values in the same way as we use the element node labels in indexing and querying. The reason is that the bisimulation graph is converted to a matrix by mapping an edge (identified by the labels of the two incident vertices) to an integer. If the domain of one of the vertex labels is infinite, the edge will be mapped to an infinite domain as well, making the matrix computation impractical.

To solve this problem, we map/hash the PCDATA or atomic value to an integer in a small range $(\alpha, \alpha+\beta]$ that does not overlap with the alphabet of element names Σ, where $\alpha > \sigma$ ($\forall \sigma \in \Sigma$), and β is a small integer parameter.

After the mapping, we treat the hashed integer as the *label* of a value node, then the FIX index can be constructed based on the new document tree with value nodes. It is straightforward to see that after the value-to-label mapping, all the properties (including the completeness) still holds for the index with value nodes. Therefore, FIX index uniformly supports structure and value matching.

It may be necessary to carefully choose the β value to tradeoff between query processing time and size of the index. With a large β, the values can be mapped to a large domain, and the bisimulation graph is large. Since the substructures are enumerated for each vertex in the bisimulation graph, there will be many substructures enumerated and inserted into the B^+ tree. This will result in a much larger B^+ tree compared to the B^+ tree containing only structures. On the other hand, with a small β, the B^+ tree will be small, but many different values will be hashed into the same label. This will introduce more false-positives because of the collisions in hashing. How to choose a proper β for a given data set is an interesting problem left for future work.

5. QUERY PROCESSING AND OPTIMIZATION USING FIX

Using FIX for query processing has two steps: the *pruning* phase prunes the input and produces candidate results, and the *refinement* phase takes the candidate results and validates them using a query processor.

Given a twig query of depth k, it is relatively straightforward to process a query using FIX (Algorithm 2): we need to first check whether the index covers the twig query by comparing the depth limit of the index and the depth of the twig query. If it does, the query tree is converted into a bisimulation graph (twig pattern), then the pattern is converted into an anti-symmetric matrix, and the λ_{max} and λ_{min} are computed. This pair of eigenvalues and the root label of the twig pattern are used as a key to perform range query in the index. For each candidate returned by the range query, the refinement query processor is invoked to get the final results. Before the query processor takes over, we need to replace the leading //-axis with /-axis. This is because any descendants of the root of an indexed pattern instance are also indexed. They will be visited eventually if they are returned by the index as candidates. For value predicates, it is straightforward to see that they can be answered without false-negatives.

For a general path expression that contains //-axes in the middle, we can decompose the query tree into multiple twig queries that are connected by //-edges. For example, the query `//open_auction[.//bidder[name][email]]/price` can be decomposed into two sub-queries: `//open_auction/price` and `//bidder[name][email]`. If the database consists of small documents and the depth limit is set to unlimited, the document whose $[\lambda_{min}, \lambda_{max}]$ range contains the $[\lambda_{min}, \lambda_{max}]$ ranges of both twig queries should be returned as candidates. If the index is built with a non-zero depth limit on large documents, only pattern instances that contain the top sub-twig query (`//open_auction/price` in the above example) are returned as candidates, otherwise even if the candidate may match the bottom sub-twig query (`//bidder[name][email]` in the above example), the top sub-twig query will not be matched thus the whole query is not matched. In this case, the bottom sub-twig query does not provide any pruning.

The cost of FIX index processing consists of three parts: CPU cost of converting a twig query into its bisimulation graph, converting the graph into a matrix, and computing the eigenvalues of the matrix. The cost of the first two components is $O(m)$ each, where m is the size of the query, and the eigenvalue computation is $O(m'^3)$, where m' is the size of the bisimulation graph and $m' \leq m$. For a reasonable sized query, these costs are negligible. The I/O cost includes searching the B^+ tree and retrieving the document from B^+ tree (for the clustered index) or from the primary storage (for the unclustered index). The cost of searching the B^+ tree is well studied and the missing part in cost estimation is the number of candidate results. This can be estimated if we have further knowledge (e.g., histograms on λ_{max}, λ_{min}, and root labels of pattern instances). A good practice is to build a histogram on the primary sorting key (e.g., λ_{max}) in the B^+ tree. The rest of the cost is that of refinement of the candidate results. Although the number of candidates may be the same, clustered and unclustered index may have much different cost due to the difference between sequential and random I/O.

Algorithm 2 Index Query Processing

Index-Processor(Q : TwigQuery, Idx : FIX)

1 check the Idx depth limit is no shorter than Q's depth;
2 $Q' \leftarrow$ Convert-to-Bisim-Graph(Q);
3 $\mathbf{M} \leftarrow$ Convert-to-Matrix(Q');
4 $\langle \lambda_{max}, \lambda_{min} \rangle \leftarrow$ Eig-Pair(M);
5 $k \leftarrow \langle \lambda_{max}, \lambda_{min}, \text{root label of } Q' \rangle$;
6 $C \leftarrow Idx.search(k)$;
7 **if** Idx has non-zero depth limit
8 replace the leading //-axis with /-axis from Q;
9 **for** each $c \in C$
10 **do if** Idx is clustered **then**
11 run refinement query processor with Q on c;
12 **else** run refinement query processor with Q following the pointer c;

6. EXPERIMENTAL EVALUATION

In this section, we first evaluate the performance of structural FIX index according to three implementation independent metrics, as well as the actual run-time speedup against the state-of-the-art evaluation techniques. Then we evaluate the integrated value and structural index. While the wall clock time speedup is the "net effect" of the benefit of using FIX index over a specific algorithm implementation, implementation independent metrics reveal more insights into the design decisions of the FIX index and provide a general guideline of how much improvement the FIX index can achieve for any implementation.

The FIX index is implemented in C++ and uses Berkeley DB for the B^+ tree implementation. We choose the NoK processor [29] to perform the refinement step. The NoK processor is an efficient navigational operator that evaluates a path expressions that only contains child and following-sibling axes. To compare with the unclustered FIX index, we extend the implementation of NoK operator with support for //-axes. To compare with the clustered FIX index, we choose the disk-based F&B index [24], whose implementation is obtained from the authors. The disk-based F&B index has been reported to have superior performance over a number of other indexes, so we select it as a representative state-of-the-art clustering index. All the tests are conducted

data sets	size	# elements	ICT	|UIdx|	|CIdx|
TCMD	27.9 MB	115306	17.8 sec	0.2 MB	6.1 MB
DBLP	169 MB	4022548	32.5 sec.	2 MB	77.9 MB
XMark	116 MB	1666315	86 sec.	5.6 MB	143.3 MB
Treebank	86 MB	2437666	375 sec.	37.3 MB	310.6 MB

Table 1: Characteristics of experimental data sets, the total construction times for both unclustered and clustered indexes (ICT), and the sizes of the unclustered index (UIdx) and clustered index (CIdx)

on a PC with Pentium IV 3GHz CPU and 1GB memory running Windows XP.

6.1 Test Data Sets and Index Construction

We tested both synthetic and real data sets. For the category of large collection of small documents, we use XBench [27] TCMD (text-centric multi-document) data set, which models the real world text-centric XML data sets such Reuters news corpus and the Springer digital library. This data set contains 2,607 documents with various sizes from 1KB to 130KB. The document structures have small variations, e.g., an `article` element may or may not have a `keywords` subelement. Since all documents in the collection are small, we do not enumerate substructures in each document when constructing the index, i.e., the depth limit parameter in Algorithm 1 is set to zero.

We also tested FIX with non-zero depth limit on large XML documents such as DBLP [16], XMark [21] with scale factor 1, and Treebank [1]. They are chosen because of their different structural characteristics. The structure in DBLP is very regular and the tree is shallow, so the same structure is repeated many times, making each structural pattern less selective. The XMark data set is structure-rich, fairly deep and very flat (fan-out of the bisimulation graph is large), therefore, the structures are less repetitive. The Treebank data set represents highly recursive documents. It is deeper but less flat than XMark; the structures are very selective.

The basic statistics, index construction time, and index sizes for these data sets are listed in Table 1. The index we constructed for XBench TCMD data has no depth limit, and the indexes for the other data sets are constructed by enumerating subpatterns of depth limit 6. The construction times for indexes with smaller depth limits are slightly faster. This depth limit is chosen so that the index can cover fairly complex twig queries. Depending on the complexity of bisimulation graph of the document and the depth limit, the enumerated subpattern could be too large for calculating eigenvalues (e.g., number of edges is larger than 3000). In this case, we do not calculate the eigenvalues but use an artificial $[\lambda_{min}, \lambda_{max}]$ range of $[0, \infty]$ to guarantee that the instances of this subpattern will always be returned as a candidate result. This may lose pruning power, but fortunately, there are very few such cases in all the test data sets for reasonable depth limit of 6 (1 for DBLP, 11 for XMark, and 1 for Treebank). As seen from Table 1, the indexes for the data sets can be constructed very efficiently except perhaps for the Treebank data set. Nevertheless, considering the complexity of Treebank and comparing to the F&B index [24], whose construction time is several hours on the same data set, the FIX construction time is quite reasonable. In all the data sets, the unclustered index is smaller than the F&B index but the clustered index is larger.

query	*sel*	*pp*	*fpr*
`TCMD_hi`	79.31%	26.12%	71.99%
`TCMD_md`	49.23%	5.62%	46.21%
`TCMD_lo`	16.85%	0.35%	16.29%
`DBLP_hi`	99.97%	99.79%	84.91%
`DBLP_md`	72.59%	70.85%	5.91%
`DBLP_lo`	47.36%	47.35%	0.002%
`XMark_hi`	99.96%	99.87%	75.13%
`XMark_md`	99.10%	98.71%	30.14%
`XMark_lo`	98.89%	98.43%	30.01%
`TrBnk_hi`	99.97%	95.37%	99.45%
`TrBnk_md`	99.81%	85.97%	98.67%
`TrBnk_lo`	97.48%	95.36%	45.79%

Table 2: Implementation-independent metrics for representative queries for each data sets in each category

6.2 Performance Evaluation Based on Implementation-independent Metrics

We define three metrics to evaluate the effectiveness of FIX: pruning power $pp = 1 - cdt\,/\,ent$, selectivity $sel = 1 - rst\,/\,ent$, and false-positive ratio $fpr = 1 - rst\,/\,cdt$, where *cdt* is the number of entries returned by the index as candidate results, *ent* is number of all entries in the index, and *rst* is the number of entries that actually produce at least one final result. For the index with depth limit 0 on a large collection of small documents, *pp* is the ratio of number of documents pruned by the index over the total number of documents in the collection. For the index with non-zero depth limit k, since each element corresponds to an entry in the index (the subtree of depth k starting from that element), *pp* is the ratio of remaining elements as the result of pruning over the total number of elements.

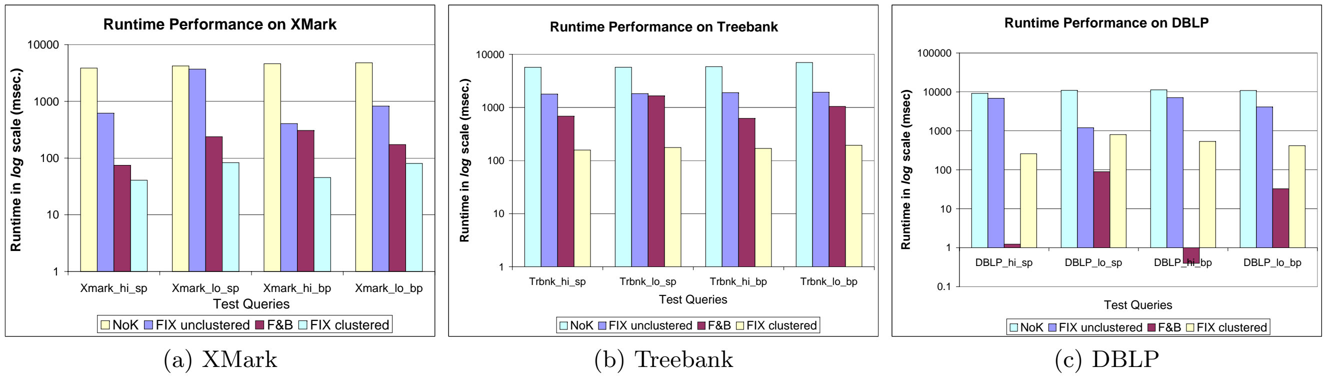
In order to evaluate the real effectiveness of the index, the pruning power metric should be combined with the *selectivity* of the query. The low pruning power of a query does not mean that the index is ineffective if the selectivity is also low (i.e., the query is not selective). The only bad case is when the selectivity is high but the pruning power is low.

The metric *fpr* is another indicator of the effectiveness of the pruning of the FIX index against the "perfect" index, which produces no false-positives.

For each data set, we randomly generate 1000 test queries and select representative queries based on their selectivities: low, medium, and high. However, depending on the characteristics of the data sets (i.e., the distribution of the substructures), these queries may not cover all 3 selectivity criteria. For example, since each document in the XBench TCMD have very similar structure, the queries are more likely to fall into the category of low selectivity. On the other hand, XMark and Treebank data sets are structure-rich, thus almost all queries fall into the high selectivity category. For these cases, we select the representative queries given below with *relatively* high or low selectivity[4].

```
TCMD_hi : /article/epilog[acknowledgements]/references/a_id
TCMD_md : /article/prolog[keywords]/authors/author/contact[phone]
TCMD_lo : /article[epilog]/prolog/authors/author
DBLP_hi : //proceedings[booktitle]/title[sup][i]
DBLP_md : //article[number]/author
```

[4]We eliminated queries that have selectivity 0 and 1 since they do not give us much information about the index.



(a) XMark (b) Treebank (c) DBLP

Figure 6: Runtime comparisons on XMark, Treebank, and DBLP

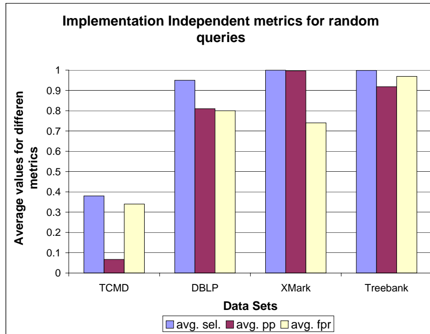


Figure 5: Average selectivity, pruning power, and false-positive ratio of 1000 random queries on different data sets

```
DBLP_lo : //inproceedings[url]/title
XMark_hi: //category/description[parlist]/parlist/listitem/text
XMark_md: //closed_auction/annotation/description/text
XMark_lo: //open_auction[seller]/annotation/description/text
TrBnk_hi: //EMPTY/S/NP[PP]/NP
TrBnk_md: //S[VP]/NP/NP/PP/NP
TrBnk_lo: //EMPTY/S[VP]/NP
```

The selectivity, pruning power, and false-positive ratios for these queries are listed in Table 2. For low selectivity queries (e.g., `TCMD_lo`), FIX does not show strong pruning power. However, since only about 16% of the returned candidates are false positives, the index still performs well in that most of the remaining candidates produce final results. On the other hand, for highly selective queries, such as (almost) all XMark and Treebank queries, FIX prunes very well, very close to the selectivity. This means that the features used in FIX reflect the intrinsic structural characteristics of the patterns. However, the false-positive ratios for queries in this category could also be high (e.g., `TrBnk_hi` and `TrBnk_md`). This suggests that there may be other features that are unique in this data set that are missed in our index, which will be considered in our future work. For the queries in the medium category, the effectiveness of FIX varies. The pruning powers of FIX on some queries (e.g., `DBLP_md` and `XMark_md`) are very close to their selectivities, and the false-positive ratios are reasonable. On the other hand, some queries have poor pruning power (e.g., `TCMD_md`) or the false-positive ratio is high (e.g., `TrBnk_lo`). This case represents the grey area that is hard to estimate the cost.

The average of the three metrics over the random 1000 queries for each data set is shown in Figure 5. As seen from this figure, the average pruning power is very close to the selectivity for XMark and Treebank, but there are about 32% and 14% differences for TCMD and DBLP, respectively. One of the reasons for this is that, as indicated earlier, unlike XMark and Treebank, XBench TCMD and DBLP are not structure-rich. *Structural* indexes that cluster based on structures are not likely to be effective anyway. In the Section 6.4, we shall show that the integrated structural and value index can improve the pruning power as well as the query processing time.

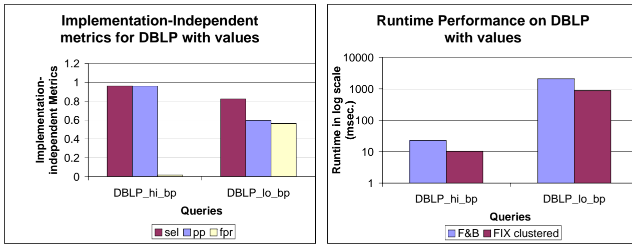
6.3 Run-time Performance Evaluation

We tested the running time speedup for using FIX indexes. Although clustered index (such as F&B index and clustered FIX index) are more efficient in query processing, they are less efficient in result subtree construction (due to the loss of document order). Furthermore, clustering criteria may conflict with other sorting criteria, making the unclustered FIX index or the original storage preserving document order (such as the one in [29]) preferable. To compare similar techniques based on the same criteria, we focus on the run-time performance of unclustered FIX index with the NoK navigational operator without index support, and compare the clustered FIX index with clustered F&B index that has been shown to perform better than other indexes [24].

To be able to benchmark different types of queries, we look at both simple path (sp) and branching path (bp) queries. Together with the selectivity dimension, low (lo) and high (hi) selectivity, there are four test queries for each data sets: $\{\texttt{hi}, \texttt{lo}\} \times \{\texttt{sp}, \texttt{bp}\}$. The test queries are listed as follows:

```
XMark_hi_sp: //item/mailbox/mail/text/emph/keyword
XMark_lo_sp: //description/parlist/listitem
XMark_hi_bp: //item[name]/mailbox/mail[to]/text[bold]/emph/bold
XMark_lo_bp: //item[payment][quantity][shipping][mailbox/mail
             /text]/description/parlist
Trbnk_hi_sp: //EMPTY/S/NP/NP/PP
Trbnk_lo_sp: //EMPTY/S/VP
Trbnk_hi_bp: //EMPTY/S/NP[PP]/NP
Trbnk_lo_bp: //EMPTY/S[VP]/NP
DBLP_hi_sp : //inproceedings/title/i
DBLP_lo_sp : //dblp/inproceedings/author
DBLP_hi_bp : //inproceedings[url]/title[sub][i]
DBLP_lo_bp : //article[number]/author
```

Figure 6 depicts the speedup of the FIX indexes to the existing techniques in logarithmic scale. As shown in Figures 6(a) and 6(b), FIX unclustered and clustered indexes



(a) Implementation-independent Metrics

(b) Runtime Speedup

Figure 7: DBLP with values

performs considerably better than the NoK or F&B indexes, respectively. However, on the more regular and simple data set DBLP (Figure 6(c)), although FIX unclustered index still outperforms NoK, F&B index outperforms FIX clustered index, particularly in the cases of queries with high selectivity. The reason is that the structure of DBLP data set is very regular and shallow. The whole F&B index for DBLP is only 180 KB, and could easily fit into main memory due to the caching mechanism of F&B index implementation. We conjecture, however, that queries on simple data sets usually involve value constraints. For such a general path expression, the majority of processing time is spent on the value-predicate evaluation. Therefore, we also test the index performance with value constraints.

6.4 Performance of Value Indexes

We choose $\beta = 3$ when building the value index to balance the query performance and the space overhead. Since DBLP is the only real data set (the PCDATA in other data sets are all randomly generated), and since queries with value-predicates are all branching paths, we only tested the branching paths with high selectivity and low selectivity on the DBLP data set. The test queries are listed as follows:

```
DBLP_vl_hi: //proceedings[publisher="Springer"][title]
DBLP_vl_hi: //inproceedings[year="1998"][title]/author
```

Figure 7(a) shows the implementation-independent metrics. For low selective queries, the FIX index with values performs comparable to the FIX index with no values as far as the implementation-independent metrics are concerned. For high selective queries, however, FIX index with values demonstrates a significant improvement over the pure structural index, with the selectivity and pruning power almost identical, and false-positive ratio (*fpr*) around 1.7%. Figure 7(b) shows the runtime speedups compared to F&B index. The FIX index with values outperforms F&B index on both queries by more than a factor of 2. However, FIX index with values does not come for free, the construction time and memory requirement are much higher than the pure structural index (around a factor of 30 and 10 with $\beta = 10$, respectively). We believe, with careful tuning of the β value, one can achieve the balance between the cost associated with the index construction and the savings for the query processing.

7. RELATED WORK

A wide variety of join-based, navigational, and hybrid XPath processing techniques have been proposed [28, 3, 6, 11, 29, 17]. Much research has focused on indexing techniques to improve these existing query processors. Among these techniques, TJFast [17] proposes extended Dewey ID to encode input elements for the holistic twig join; and XB-tree [6], XR-tree [13], iTwigStack [9], and ToXin [4] are proposed to prune the input lists to the holistic twig join operator. FIX also follows the pruning approach, but it is not designed to work for a particular operator, and can be coupled with any path processing operator that can perform query refinement.

A parallel line of research focuses on the clustering index [18, 15, 14, 24]. The common theme of these clustering techniques is that they are all based on some variant of *simulation/bisimulation* graph of the XML data tree. Depending on different definition of the (bi)simulation, the computational complexity, space complexity and type of queries that can be answered are different. For example, if the bisimulation graph is defined using the similarity or bisimilarity relation [12], only simple linear path queries can be answered without looking at the original data and performing refinement; while indexes constructed using the forward and backward (F&B) bisimilarity relation [2] can answer all twig (branching) path queries, thus bearing the name of *covering index*. For a non-covering index, a further refinement step may be required to search for the final results from the candidate results. FIX uses the bisimulation graph as the basis for representing structural information in the XML tree that is good enough to answer existential match. FIX does not use the bisimulation graph itself is as an index, but uses the structural information extracted from the bisimulation graph. By separating a large bisimulation graph into smaller ones, we can quickly find a substructure as the candidate of a pattern without traversing the whole graph.

ViST [23] and PRIX [20] are two other XML indexing techniques that fall into one category: they both break the XML document into structure-encoded strings and store them into a conventional index such as B^+ tree. These strings can also be considered as features. However, unlike FIX, these indexes need special operators for refinement.

Eigenvalues and spectral graph theory have many applications in other areas of computer science. Our initial idea was inspired by the work in Computer Vision, where spectra of shock graphs are used to index visual objects [22]. The shock graph is a unlabeled directed graph to represent the abstract of objects. They use the full set of eigenvalues as features to approximate query processing, but did not make use or prove the $[\lambda_{min}, \lambda_{max}]$ property to for substructure queries (thus did not prove the property). There are also related work in the area of data mining, in which a large collection of graphs are indexed by identifying "features" — frequent substructures [25, 26]. Their features are combinatorial in that features are compared by subgraph isomorphism.

8. CONCLUSION

As more and more document are stored in XML databases, an index is needed to quickly retrieve a subset of candidates to do further refinement. Depending on the characteristics of the data sets, a value-based index or a structural index or both are appropriate for certain queries. In this paper, we have proposed the feature-based index FIX for indexing substructures as well as values in a document or collection of documents. Our approach, to the best of our knowledge, is the first XML indexing technique to take the substructures

and values as a whole object and compute its distinctive features. Unlike many other indexing techniques, FIX can be combined with an XPath query processor with litter or no change in its implementation. We have successfully applied FIX as a pruning index for an existing highly optimized navigational operator, resulting in orders of magnitude speedup in running time.

In addition to the navigational operator, we plan to apply FIX to other query operators and evaluate the performance of other ways of incorporating values into the index. We also plan to investigate the use of R-tree or other high-dimensional indexing trees to gain further pruning power. Finally, we are also interested in finding more features to use in index and finding out what features are most effective for a particular type of data sets.

Acknowledgements We thank Justin W.L. Wan of the University of Waterloo for the fruitful discussion that leads to the proof of Theorem 3. We are also grateful to Wei Wang and Hongzhi Wang for kindly providing us the source and binary code of [24] for performance comparisons.

9. REFERENCES

[1] University of Washington XML Data Repository. Available at `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[3] 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'02*, pages 141–152, 2002.

[4] A. Barta, M. P. Consens, and A. O. Mendelzon. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *VLDB'05*, pages 133–144, 2005.

[5] B. Bollobás. *Modern Graph Theory*. Springer, 1998.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD'02*, pages 310–322, 2002.

[7] D. Chamberlin. XQuery: An XML Query Language. *IBM Systems J.*, 41(40):597–615, 2002.

[8] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An Adaptive Structural Summary for Graph-Structured Data. In *SIGMOD'03*, pages 134–144, 2003.

[9] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Index Techniques. In *SIGMOD'05*, pages 455–466, 2005.

[10] D. M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs: Theory and Application*. Academic Press, 1979.

[11] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed Mode XML Query Processing. In *VLDB'03*, pages 225–236, 2003.

[12] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulation on Finite and Infinite Graphs. In *FOCS'95*, pages 453–462, 1995.

[13] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *VLDB'03*, pages 273–284, 2003.

[14] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexing for Branching Path Queries. In *SIGMOD'02*, pages 133–144, 2002.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *ICDE'02*, pages 129–140, 2002.

[16] M. Ley. DBLP XML Repository. Avaiable at `http://dblp.uni-trier.de/xml`; accessed April 2004.

[17] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *VLDB'05*, pages 193–204, 2005.

[18] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. 7th Int. Conf. on Database Theory*, pages 277–295, 1999.

[19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[20] P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *ICDE'04*, pages 288–300, 2004.

[21] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[22] A. Shokoufandeh, D. Macrini, S. Dickinson, K. Siddiqi, and S. W. Zucker. Indexing Hierarchical Structures Using Graph Spectra. *IEEE PAMI*, 27(7):1125–1140, 2005.

[23] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *SIGMOD'03*, pages 110–121, 2003.

[24] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li. Efficient Processing of XML Path Queries Using the Disk-based FB Index. In *VLDB'05*, pages 145–156, 2005.

[25] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD'04*, pages 335–346, 2004.

[26] X. Yan, P. S. Yu, and J. Han. Substructure Similarity Search in Graph Databases. In *SIGMOD'05*, pages 766–777, 2005.

[27] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *ICDE'04*, pages 621–633, 2004.

[28] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD'01*, pages 425–436, 2001.

[29] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE'04*, pages 54 – 65, 2004.

[30] N. Zhang, M. T. Özsu, I. F. Ilyas, and A. Aboulnaga. FIX: Feature-based Indexing Technique for XML Documents. Technical Report CS-2006-07, University of Waterloo, 2006. Available at `http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-07.pdf`.