XML Native Storage and Query Processing

Ning Zhang
Facebook

M. Tamer Özsu
University of Waterloo, Canada

ABSTRACT

As XML has evolved as a data model for semi-structured data and the de facto standard for data exchange (e.g., Atom, RSS, and XBRL), XML data management has been the subject of extensive research and development in both academia and industry. Among the XML data management issues, storage and query processing are the most critical ones with respect to system performance. Different storage schemes have their own pros and cons. Some storage schemes are more amenable to fast navigation, and some schemes perform better in fragment extraction and document reconstruction. Therefore, based on their own requirements, different systems adopt different storage schemes to tradeoff one set of features over the others.

In this chapter, we review different native storage formats and query processing techniques that have been developed in both academia and industry. Various XML indexing techniques are also presented since they can be treated as specialized storage and query processing tools.

KEYWORDS

DBMS, XML, Native storage formats, Query processing, Indexing

INTRODUCTION

As XML has evolved as a data model for semi-structured data and the *de facto* standard for data exchange, it is widely adopted as the foundation of many data sharing protocols. For example, XBRL and FIXML defines the XML schemas that are used to describe business and financial information; Atom and RSS are simple yet popular XML formats for publishing Weblogs; and customized XML formats are used by more and more system log files. When the sheer volume of XML data increases, storing all these data in the file system is not a viable solution. Furthermore, users often want to query over large volumes of XML data. A customized and non-optimized query processing system would quickly reach its limits. A more scalable and sustainable solution is to load the XML data into a database system that is specifically designed for storing and updating large volumes of data, efficient query processing, and highly concurrent access patterns. In this chapter, we shall introduce some of the database techniques for managing XML data.

There are basically three approaches to storing XML documents in a DBMS: (1) the LOB approach that stores the original XML documents as-is in a LOB (large object) column (Krishnaprasad, Liu, Manikutty, Warner & Arora, 2005; Pal, Cseri, Seeliger, Rys, Schaller, Yu, Tomic, Baras, Berg, Churin & Kogan, 2005), (2) the extended relational approach that shreds XML documents into object-relational (OR) tables and columns (Zhang, Naughton, DeWitt, Luo & Lohman, 2001; Boncz, Grust, van Keulen, Manegold, Rittinger & Teubner, 2006), and (3) the native approach that uses a tree-structured data model, and introduces operators that are optimized for tree navigation, insertion, deletion and update (Fiebig, Helmer, Kanne, Mildenberger, Moerkotte, Schiele, & Westmann, 2002; Nicola, & Van der Linden, 2005; Zhang, Kacholia, & Özsu, 2004). Each approach has its own advantages and disadvantages. For example, the LOB approach is very similar to storing the XML documents in a file system, in that there is minimum transformation from the original format to the storage format. It is the simplest one to implement and support. It provides byte-level fidelity (e.g., it preserves extra white spaces that may be ignored by the OR and the native formats) that could be needed for some digital signature schemes. The LOB approach is also efficient for inserting or extracting the whole documents to or from the database. However it is slow in processing queries due to unavoidable XML parsing at query execution time.

In the extended relational approach, XML documents are converted to object-relational tables, which are stored in relational databases or in object repositories. This approach can be further divided into two categories based on whether or not the XML-to-relational mapping relies on XML Schema. The OR storage format, if designed and mapped correctly, could perform very well in query processing, thanks to many years of research and development in object-relational database systems. However, insertion, fragment extraction, structural update, and document reconstruction require considerable processing in this approach. For schema-based OR storage, applications need to have a well-structured, rigid XML schema whose relational mapping is tuned by a DBA in order to take advantage of this storage model. Loosely structured schemas could lead to unmanageable number of tables and joins. Also, applications requiring schema flexibility and schema evolution are limited by those offered by relational tables and columns. The result is that applications encounter a large gap: if they cannot map well to an object-relational way of life due to tradeoffs mentioned above, they suffer a big drop in performance or capabilities.

Due to these shortcomings of the two approaches, much research has been focusing on the *native* XML storage formats. There is not, and should not be, a single native format for storing XML documents. Native XML storage techniques treat XML trees as first class citizens and develop special purpose storage schemes without relying on the existence of an underlying database system. Since it is designed specifically for XML data model, native XML storage usually provides well-balanced tradeoffs among many criteria. Some storage formats may be designed to focus on one set of criteria, while other formats may emphasize another set. For example, some storage schemes are more amenable to fast navigation, and some schemes perform better in fragment extraction and document reconstruction. Therefore, based on their own requirements, different applications adopt different storage schemes to trade off one set of features over another.

The following are examples of some of the important criteria that many real-world applications consider.

1. Query performance. For many applications, query performance may be the most important criterion. Storage format, or how the tree data model is organized on disk and/or in main memory, is one of the major factors that affect query performance. Other factors include the efficiency of the query processing algorithm and whether there are indexes on the XML data. All of these will be covered in the following sections.
2. Data manipulation performance. This can be further measured by insertion, update, and document or fragment retrievals. Each of these may be an enabling feature for many applications.
3. Space and communication efficiency. XML documents are verbose and contain redundant information. A compact format benefits both the storage space and the network communication cost when they are transferred between different parties.
4. Schema flexibility. XML documents in data exchange often have schemas, which define common structures and validation rules among exchanged data. Many applications may have millions of documents conforming to a large number of slightly different schemas (schema chaos), or these schemas may evolve over time (schema evolution). A native storage format needs to be able to handle both schema chaos and schema evolution efficiently.
5. Miscellaneous features. These include partitioning, amenability to XML indexes, and seamless integration with mid-tier applications such as Java or C#.

Table 1 summarizes the comparisons LOB, OR, and the native XML storage approaches.

	LOB Storage	**OR Storage**	**Native Storage**
Query	Poor	Excellent	Good/Excellent
DML	Poor/Good	Good/Excellent	Excellent
Document Retrieval	Excellent	Good/Excellent	Excellent
Schema Flexibility	Poor	Good	Excellent
Document fidelity	Excellent	Poor	Good/Excellent
Mid-tier integration	Poor	Poor	Excellent

Table 1. Comparisons between LOB, OR, and Native XML Storages

In the next section, we introduce different native storage formats and compare them in the context of a set of commonly encountered requirements. Some native storage formats, e.g., XMill (Liefke, L., and Suciu, D., 2000) and XGrind (Tolani, P. M., and Haritsa, J. R., 2002) concentrate on only one or a very small subset of the requirements, so we will omit them from this chapter. We then introduce different query processing techniques that have been developed for native XML systems. A large body of such work focuses on answering XPath queries efficiently. Finally, we present various XML indexing techniques for answering path queries. To a certain extent, these XML indexing techniques can be thought of as specialized storage and query processing tools.

NATIVE XML STORAGE METHODS

XML data model treats XML documents as labeled, ordered, unranked trees. Main memory data structure for trees has been studied very well. Different data structures have been proposed for efficient operations such as query, insertion, deletion, and update. However, disk-based data structure for trees had not attracted as much research until the XML data model caught on. The key difference between the main memory data structures and the disk-based data structures are that the former are based on Random Access Machine (RAM) model of computation, in which accessing two different nodes in main memory have the same cost. However, in disk-based data structures, which is based on the paged I/O model, accessing a node in a new page is usually significantly more expensive than accessing a node in the cached page. Therefore many disk-based algorithms are designed to minimize the number of buffer cache misses according to access patterns. The solution to solve these problems is to partition a large tree into smaller trees such that:

1. Each subtree is small enough to be stored entirely into one disk page.
2. Based on some priori knowledge of access patterns, tree nodes that are accessed in consecutive order are clustered as much as possible into one partition.

The second condition is the major difference between different storage formats. For example, suppose nodes n_1 to n_{10} need to be partitioned into two clusters and each of them are free to be in either cluster. If it is known that the access pattern is $n_1, n_2, n_3, \ldots n_{10}$ in that order, then the partitions of $\{n_1, n_2, n_3, n_4, n_5\}$ and $\{n_6, n_7, n_8, n_9, n_{10}\}$ are optimal. However if the access pattern is $n_1, n_2, n_4, n_8, n_9, n_4, n_2, n_5, n_{10}, n_5, n_3, n_6, n_3,$ *and* n_7, then the partitions of $\{n_1, n_2, n_4, n_8, n_9\}$ and $\{n_3, n_5, n_6, n_7, n_{10}\}$ would be optimal. The reason is that each page (containing one partition) is accessed only once, which results in the optimal I/O cost.

It is clear from the above example that the access pattern is a key factor to consider when designing native XML storage formats. For XML applications, query access patterns are path-based pattern matching as defined in W3C XPath (Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., & Siméon, J. 2007) and XQuery (Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J., 2007) languages. There are basically two patterns in accessing XML nodes in a tree: *breadth-first traversal* and *depth-first traversal*. The breadth-first access pattern is usually observed in the XML DOM API, e.g., through the Java getChildNodes method in the Node class. Based on this access pattern, the storage format needs to support fast access to the list of children from its parent node. The depth-first access pattern can be seen in the XML SAX API, where a pair of begin- and end-events are generated for each node visited in document order (coincident with the depth-first traversal of the XML tree). This access pattern requires the storage formats cluster tree nodes based on their document order.

Most native XML storage schemes try to optimize both access patterns, with an emphasis on one of them.

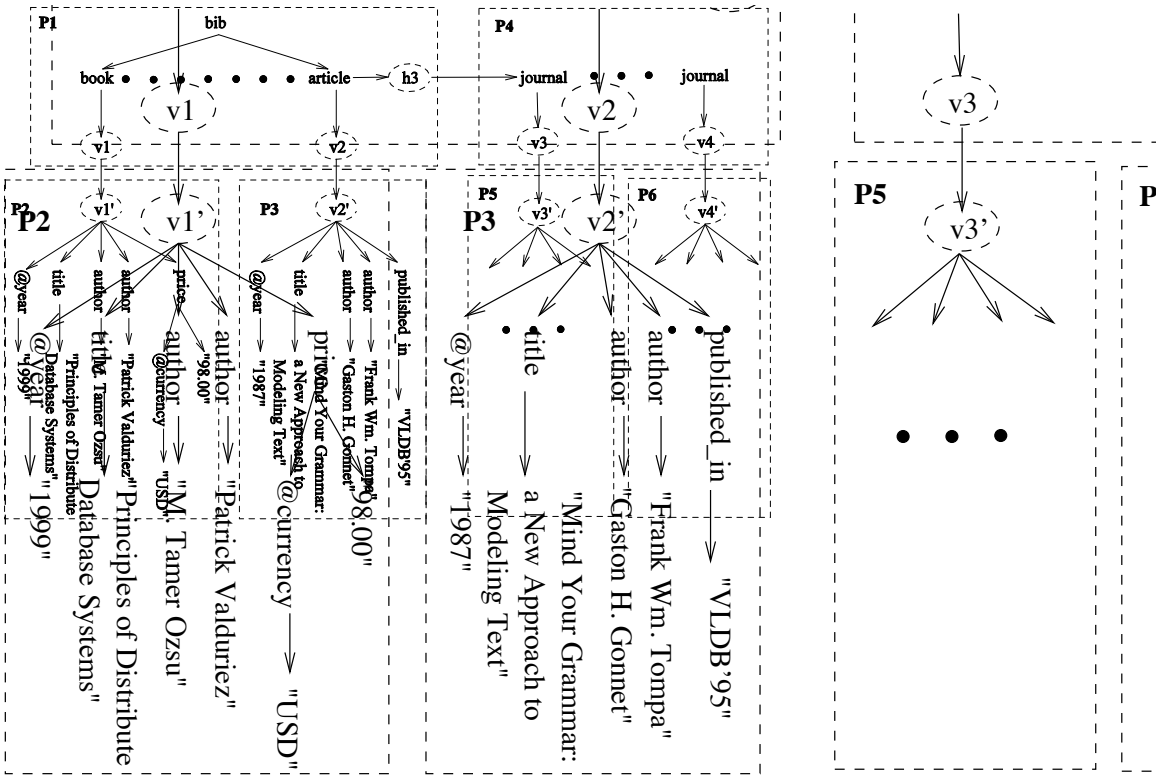

Figure 1. Breadth-first Tree Partitioning (dotted circles are proxy nodes that connect two pages)

Breadth-first-based Tree Partitioning

If a storage format favors the breadth-first access pattern, a simple clustering condition needs to be satisfied: grouping all children with their parent in the same disk page. However when a tree is large, it is hard to satisfy this simple condition. The reasons are two-fold: (1) all children should be clustered with their parent in the same page. Since the number of children is unlimited, they are not guaranteed to be stored in the same page; (2) even though all children of a node can be stashed into one page, the clustering condition implies that all its descendants need to be stored into one page as well. These two conditions are unrealistic for most large XML documents. Therefore, a partitioning algorithm based on the bread-first access pattern was proposed to group as many siblings as possible into one page, and if the page overflows, it keeps a "proxy" node (illustrated as in dotted cycles in Figure 1) that act as a linkage to the next page (Helmer, Kanne, Mildenberger, Moerkotte, Schiele, & Westmann, 2002). Figure 1 illustrates the partitioning algorithm on a large tree where the number of children of element bib is too large to fit into one page. If the tree nodes are partitioned based on breadth-first access pattern, as many children as possible should be placed into page P1 with element bib. The proxy node h3 acts as a *horizontal link* to the next sibling of element article. Furthermore, since both child elements book and article have descendants, all their descendants will be stored into separate pages, P2 and P3, where v1 and v2 act as *vertical links* to their children. It is straightforward to see that there is at most one horizontal proxy link for each page.

Even though this storage format is optimized for breadth-first traversal, clever page caching techniques can make it optimal for depth-first traversal as well. Consider Figure 1 as an example. If each page is considered as a node and each (vertical or horizontal) proxy link as an edge, it forms a tree structure, which is called a *page tree*. In the page tree, the edges are ordered so that vertical edges are ordered in the same way as their proxy nodes in the parent page, and the horizontal proxy link, if any, is the last child. Supposing that the elements are visited in depth-first traversal, the access pattern to pages is P1, P2, P1, P3, P4, P5, P4, and P6, which coincide with the depth-first traversal order of the page tree. Therefore, the page caching algorithm can be designed to push every visited page onto a stack during depth-first traversal, and pop it up when it is finished with the traversal (e.g., popping up P2 when P1 is visited the second time). Every page that is popped up from the stack is a candidate for page-out. In this case, depth-first traversal of the XML elements will visit each page exactly once, which is I/O optimal. The memory requirement is the size of the page times the depth of the page tree, i.e., the maximum length of the root-to-leaf path in the page tree.

This native XML storage format can also support efficient data manipulation operations. Inserting a tree node amounts to finding the page where the node should be inserted, and inserting it into the appropriate position. In case of page overflow, a vertical proxy node is created that links to a new page where the new node is inserted. The only caveat of this approach is that fragment retrieval may not be optimal. For example in Figure 1, if the fragment containing the first element book is to be extracted, both P1 and P2 need to be read. In the optimal case, only one page read is necessary.

Depth-first-based Tree Partitioning

Another approach of natively storing trees is to serialize tree nodes in document order, or depth-first order, so that depth-first traversal is optimal (Zhang, Kacholia, & Özsu, 2004). For example, suppose an encoded XML tree is depicted in Figure 2, where elements are encoded as follows:

bib → a, book → b, @year → z, author → c, title → e,
publisher → i, price → j, first → f, last → g

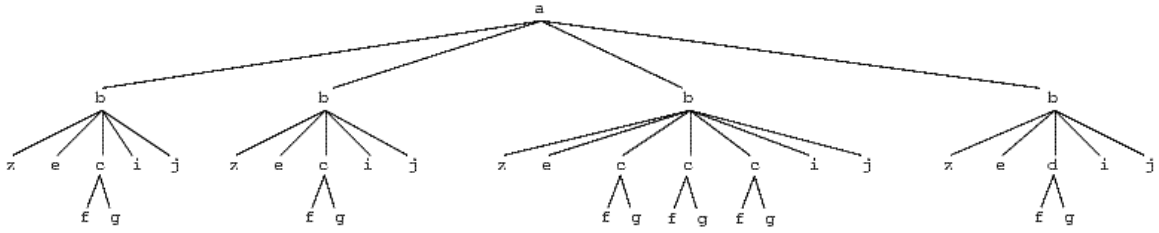

Figure 2. An encoded XML tree

One way to materialize the tree is to store the nodes in pre-order and keep the tree structure by properly inserting pairs of parentheses as the form of balanced parentheses format. For example, (A(B)(C)) is a string representation of the tree that has a root A and two children B and C. The

open parenthesis "(" preceding letter A indicates the beginning of a subtree rooted at A; its corresponding close parenthesis ")" indicates the end of the subtree. It is straightforward that such a string representation contains enough information to reconstruct the tree. However, it is not a succinct representation, because the open parenthesis is redundant in that each node, a character in the string, actually implies an open parenthesis. Therefore, all open parentheses can be safely removed and only closing parentheses are retained as in AB)C)). Figure 3 shows the string representations of the XML tree in Figure 2.

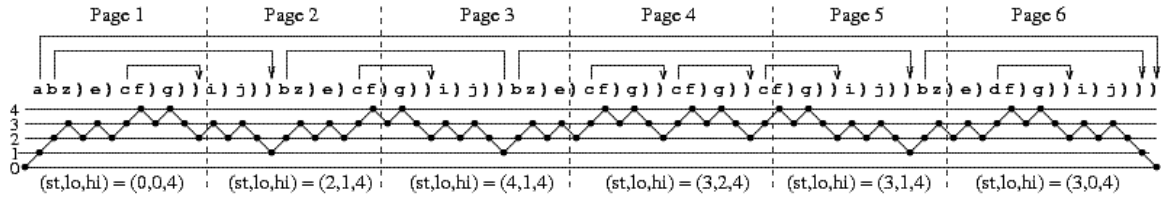

Figure 3. Balanced Parentheses Tree Representation

At the physical storage level, only the string representation in the middle is stored, the pointers above of the string and level property below the string are not. The pointers only serve to easily identify the end of a subtree to the reader, and the levels are used for efficient navigation between siblings. If the string is too long to fit in one page, it can be broken up into substrings at any point and stored in different pages as indicated in Figure 3.

This storage format is very efficient for depth-first traversal: it is only necessary to sequentially read the string from the beginning to the end, and there is no need to cache any page other than the current page. Both I/O complexity and caching size are optimal. Also practically, this storage format is more desirable than the breadth-first tree partitioning approach introduced previously. The reasons are twofold: (1) this storage format can easily take advantage of existing prefetching mechanism from the I/O component of DBMS. There is no pointer (proxy node) to follow when deciding which page to prefetch; and (2) this storage format is more CPU-cache friendly simply because nodes are ordered in depth-first order and the next node to be visited is mostly likely already loaded into the CPU cache. On the other hand, the breadth-first tree partitioning approach may see much more cache misses since the next visited node may be on the child page, and when that page is finished, the parent page is brought up again (e.g., P1, P2, P1, P3, etc. in Figure 1). . So this behavior of juggling between the parent page and children pages causes a lot of CPU cache misses.

The downside of this depth-first-based tree partitioning approach is its inefficiency in breadth-first traversal, e.g., traversing from one node to its next sibling. To solve this problem, one can store the begin and end positions of each element into an auxiliary data structure that serves as an index to find the end of a subtree. To find the next sibling of an element, what is needed is to find the end of the subtree and the next element is its next sibling. If the next character is a ')', then there is no next sibling. This method is simple and can reduce the index size by including only "large" elements in the auxiliary structure. Small elements are more likely to be within the same page so they do not save much in terms of I/O cost. This optimization is very useful during the path expression evaluation introduced in next section, where the XML tree is mostly visited in depth-first pattern and can jump to the next sibling during the traversal. However, maintaining the

begin and end positions is an extra cost during inserting and deleting nodes. One insertion in the middle of a page may cause many updates of the begin and end positions of nodes after the inserted node in the page.

Another solution to trade off the navigation and maintenance costs is to keep some extra information in the page header. The most useful information for locating children, siblings and parent is the node level information, i.e., the depth of the node from the root. For example, assuming the level of the root is 1, the level information for each node is represented by a point in the 2-D space under the string representation in Figure 3 (the *x*-axis represents nodes and the *y*-axis represents level). For each page, an extra tuple (st, lo, hi) is stored, where st is the level of the last node in the previous page (st is 0 for the first page), lo and hi are the minimum and maximum levels of all nodes in that page, respectively. If all the page headers are preloaded into main memory, they can serve as a light-weight index to guess which page contains the following sibling or parent of the current node. The idea is based on the fact that if the current node U with level L has a following sibling, the page that contains this following sibling must have a character“)” with level L − 1 (this is the closing parenthesis corresponding to U). If L −1 is not in the range [lo, hi] of a page, it is clear that this page should not be loaded. This could greatly reduce the number of page I/Os for finding the following sibling of large elements. As an example, suppose the search is to find the following sibling of element b in page 3. Since the level of b is 2, the level of its corresponding end-subtree character ‘)’ is 1. Since the [lo, hi] ranges of page 4 and 5 are [2, 4], and [1, 4] respectively, page 4 can be skipped and page 5 can be read to locate the node whose level is 1. The next character is the following sibling of b. This approach may not be as efficient as the pointer-based approach for tree navigation, but it has lower maintenance cost, since the extra cost of inserting a node may just be updating the page header in some rare cases if the minimum or maximum level statistics is changed.

QUERY PROCESSING TECHNIQUES

Path expressions in XPath or XQuery are an important focus of XML query processing. A path expression defines a tree pattern that is matched against the XML tree and specifies nodes that need to be returned as results. As an example, the path expression //book[author/last = "Stevens"][price < 100] finds all books written by Stevens with the book price less than 100. Path expressions have three types of constraints: the tag name constraints, the structural relationship constraints, and the value constraints. They correspond to the name tests, axes, and value comparisons in the path expression, respectively. A path expression can be modeled as a tree, called a *pattern tree*, which captures all three types of constraints.

There are basically two types of path query processing techniques: query-driven and data-driven. In the query-driven navigational approach (Brantner, Helmer, Kanne, & Moerkotte, 2005), each location step in the path expression is translated into a transition from one set of XML tree nodes to another set. To be more specific, a query execution engine translates a path query into a native XML algebraic expression. Each location step (e.g., //book, book/author, or author/last) in the path expression is translated into an Unnest-Map operator that effectively replaces an input list with an output list satisfying structural relationship specified by the axis. A path expression is then translated into a chain of Unnest-Map operators. Specifically, a Unnest-

Map operator defined for a child-axis takes a list of nodes and produces a list of child nodes. The breadth-first tree partitioning based storage format is most suitable for such operations, since child nodes are clustered and can be reached directly in one page or accessed through proxy links in other pages. However, it has large memory requirement since it needs to cache the intermediate results for each location step.

In the data-driven approach (Barton, Charles, Goyal, Raghavachari, Fontoura, & Josifovski, 2003; Zhang, Kacholia, & Özsu, 2004), the query is translated into an automaton and the data tree is traversed in document order according to the current state of the automaton. This approach is also referred as the streaming evaluation approach since the automaton is executed on top of the stream of input SAX events and there is no need to preprocess the XML document. This data-driven query processing approach works best with the depth-first based tree partitioning storage format. A subset of path expression is identified as a Next-of-Kin (NoK) expression when all axes are child-axis except the axis of the pattern tree root. For example, the path expression //book[author/last = "Stevens"][price < 100] is a NoK expression. A NoK expression can be evaluated by a single scan of the input stream with constant memory requirement. A general path expression containing descendant axes does not have this nice property. Algorithm 1 is the pseudocode for the NoK pattern matching.

Algorithm 1. NoK Pattern Matching

```
NoK-Main
1.  R := empty set;
2.  if Root-Lookup(pnode, snode) then
3.      NPM(pnode, snode, R);
4.  endif
5.  return R;

Root-Lookup (pnode, snode)
1.   if label(pnode) != label(snode) then
2.       return FALSE;
3.   endif
4.   while pnode is not the root of pattern tree do
5.       pnode := pnode.parent;
6.       snode := Parent(snode);
7.       Root-Lookup(pnode, snode);
8.   endwhile
9.  return TRUE;

NPM
1.  if proot is the returning node then
2.     construct or update a nlist for the candidate result;
3.     append snode to R;
4.  endif
5.  S := all frontier children of proot;
6.  u := First-Child(snode);
```

```
7.  repeat
8.     for each s in S that matches u with both tag name and value constraints do
9.           b := NPM(s,u,R);
10.           if b = TRUE then
11.                 S := S - {s};
12.                 delete s and its incident arcs from the pattern tree;
13.                  insert new frontiers caused by deleting s;
14.           endif
15.    endfor
16.  u := Following-Sibling (u);
17. until u = NIL or S = empty set
18. if S != empty set and proot is a returning node then
19.    remove all matches to proot in R;
20.     return FALSE;
21. endif
22. return TRUE;
```

This algorithm first finds the XML tree nodes (*snode*) that match with the root of the pattern tree (*pnode*), and invokes NoK pattern matching (*NPM*) from there. The results are put in the list *R*. The idea of the *NPM* is to maintain a set of frontier children for each pattern tree node. Whenever a pattern tree node is matched, all its children in the pattern tree are called frontier children and they are the candidates to be matched for the next incoming XML tree node in depth-first traversal. This procedure is a recursive function and traverses the XML tree in a depth-first fashion through the two function calls *First-Child* and *Following-Sibling*. Therefore, depth-first-based clustering storage formats works best with this algorithm.

STRUCTURAL XML INDEXES

There are many approaches to indexing XML documents. An XML document can be shredded into relational tables and regular B-tree indexes can be created on their columns (Zhang, C., Naughton, J., DeWitt, D., Luo, Q., & Lohman, G., 2001 ; Li, Q., & Moon, B., 2001), or it can be represented as a sequence and the index evaluation algorithm is reduced to string pattern matching (Wang, H., Park, S., Fan, W., & Yu, P., 2003; Rao, P., & Moon, B., 2004). Another large body of research focuses on *structural* XML indexes, where XML tree nodes are clustered based on their *structural similarity*. To some extent, an XML index can be thought of as a special XML storage format in that it also clusters tree nodes based on certain criteria. The differences are that XML indexes are solely created for query performance and storage formats are designed for many purposes and need to trade off between different requirements. We will focus on the structural XML indexes in this section.

Different notions of structural similarity result in different structural indexes. There are basically five notions of structural similarity: tag name, rooted path, bisimulation graph, F&B bisimulation graph, and feature vector. In the next sections, we shall introduce indexes based on these notions of structural similarity.

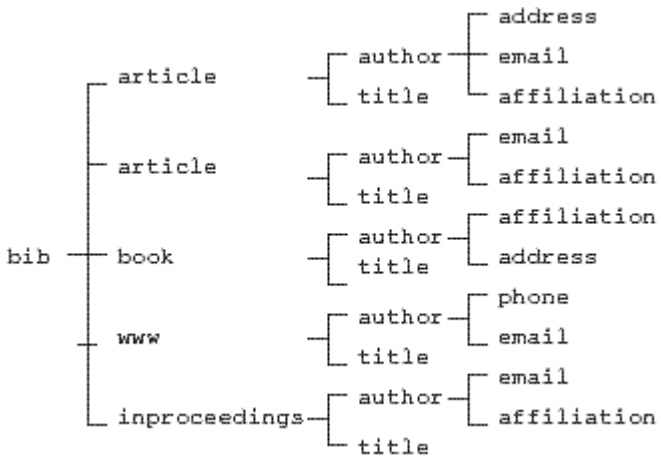

Figure 4. An Example XML Tree

Tag Name Index

Under the simplest similarity notion, two nodes are similar if and only if they have the same tag name. Indexes based on this similarity notion are called tag-name indexes. The tag-name index can be easily implemented by a regular B+ tree, where the keys are the tag names and the values are the pointers to the storage that store the elements with the tag name. Elements clustered based on tag names are also used as input to many join-based path expression evaluation algorithms (Zhang, Naughton, DeWitt, Luo, & Lohman, 2001; Al-Khalifa, Jagadish, Koudas, Patel, Srivastava, & Wu, 2002; Bruno, Koudas, & Srivastava, 2002). Therefore, this evaluation strategy can be thought of as an index-based join.

Consider the XML tree in Figure 4. There are 11 distinct tag-names in the document, each of which corresponds to one key in the tag name index. Given a tag name, it is easy to find all elements with that tag name. Therefore, the tag name index is sufficiently expressive to answer path queries of length one such as //author. Since it is not suitable for answering queries of length greater than one, it has very limited usage in practice.

Path Index

The rooted path of a node is the sequence of nodes starting from the document root to this node. If two nodes have the same rooted path, they are said to be similar in terms of their rooted path. In Figure 4, there are 22 distinct rooted paths, e.g., (bib), (bib, article), (bib, article, author), (bib, book, author), etc. Each rooted path can be assigned a distinct path ID, which serves as the key to the B+ tree. The value of the path index is the pointer to the element storage with this rooted path. For example, the two email elements under article subtrees have the same rooted path, so they can be reached by looking up in the path index with their path ID. Both DataGuide (Goldman & Widom, 1997) and 1-index (Milo & Suciu, 1999) fall into this category.

It is straightforward to see that the path index is a refinement of the tag-name index, in that all nodes under the same clusters in the path index are also under the same cluster in the tag-name

index, but not vice versa. For example, the email element under the www subtree does not have the same rooted path as the two email elements under the article subtree. Because of this, the path index can answer queries involving a simple path expression having only child-axes, e.g., /bib/article/author. Even though the path index is a refinement of the tag-name index, it cannot answer the query //author since the key to the path index is the ID rather than the real path. One way to solve this is to use the reversed rooted path as the key. For example, the keys include (email, author, article, bib), (author, www, bib) etc. When the nodes are clustered based on the reversed rooted path, elements with the same tag name will be implicitly clustered as well. Therefore, this path index can also be used to answer queries such //author.

Bisimulation Graph

One of the major limitations of the above two indexes is that they cannot answer path queries with predicates. The reason is that their similarity notion is based on the current node label (tag-name index) or the label of its ancestors (rooted path). For path expressions with predicates, it is necessary to find nodes whose children or descendant match a subtree. For example, the path expression //author[address][email] finds all author elements that have both address and email as children. To be able to find all such author elements, the index needs to cluster the nodes based on their set of children. The bisimulation graph was proposed to solve this problem (Henzinger, Henzinger & Kopke, 1995).

The bisimulation graph of a node can be derived from the subtree rooted at the node. Given an XML tree, there is a unique (subject to graph isomorphism) minimum bisimulation graph that captures all structural constraints in the tree. The definition of bisimulation graph is as follows:

Definition 1 (Bisimulation Graph):

Given an XML tree $T(N, A)$, where N is the set of nodes and A is set of tree edges, and a labeled graph $G(V, E)$, where V is the set of vertices and E is a set of edges, an XML tree node u in N is (forward) bisimilar to a vertex v in V (denoted as $u \sim v$) if and only if all the following conditions hold:

1. u and v have the same label.
2. if there is an edge (u, u') in A, then there is an edge (v, v') in E such that $u' \sim v'$.
3. if there is an edge (v, v') in E, then there is an edge (u, u') in A such that $v' \sim u'$.

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 node in T.

It is easy to see that every node in the XML tree T is bisimilar to itself. Furthermore, if there are two leaf nodes in T whose labels are the same, these two nodes can be merged together and the resulting vertex is, by definition, still bisimilar to the two nodes. It can also be proven 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. In fact the bisimulation graph of a DAG is a DAG.

The above properties imply that the bisimulation graph can be constructed in a bottom up fashion: merging leaf nodes with the same labels first and then keep merging upper level nodes

that have the same set of children. Note that when merging non-leaf nodes, two nodes are said to have the same set of children if and only if each child can find another child in the other set such that they recursively have the same set of grandchildren. This satisfies the definition of bisimulation. For example, the bisimulation graph of the XML tree in Figure 4 is depicted in Figure 5 by merging tree nodes from bottom up.

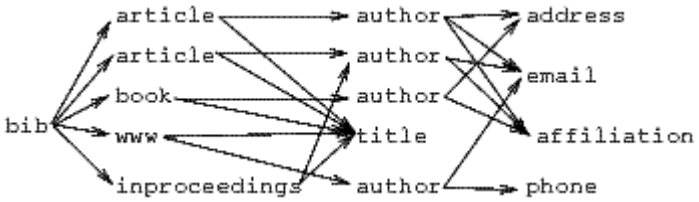

Figure 5. Bisimulation graph of the XML tree in Figure 4.

After the bisimulation graph is constructed, each vertex in the bisimulation graph corresponds to a set of nodes in the XML tree. This actually defines an equivalent class of tree nodes: if two tree nodes are bisimilar, they should be merged in the bisimulation graph construction phase. Therefore, each equivalent class is a cluster of tree nodes (which is called *bisimulation cluster*) that are represented by the vertex in the bisimulation graph. It is also easy to prove that for a predicated path query such as //author[address][email], if any node in the bisimulation cluster satisfies this path expression, then all nodes in that cluster do. Therefore, the bisimulation graph can be thought of as a structural index.

Since this index is a graph rather than the key-value pairs as in the tag-name and path indexes, it cannot be stored using the existing disk-based data structures such as the B+ tree. Fortunately since the bisimulation graph is usually much smaller than the XML tree, it is usually fully loaded into the main memory and then fed as an input to the tree pattern matching algorithm. The tree pattern matching algorithm is very similar to the one introduced in the previous section. However, the input event generation is different: previously the XML tree was traversed in depth-first order during which the begin and end events were generated for each element. If the input is the bisimulation graph, the DAG needs to be traversed in depth-first order, during which the begin and end events are generated when a vertex is visited the first and last time, respectively.

F&B Bisimulation Graph

As mentioned in the previous subsection, the bisimulation graph can answer predicated path queries starting with a descendant-axis with only one step in the non-predicate path. For example, it can answer queries such as //author[address][email] but not queries such as /bib/article/author[address][email]. The reason is that the bisimulation graph clusters tree nodes only based on their subtrees, without considering their ancestors. Therefore, all author nodes that have the two children address and email will be clustered together, even though some of them do not have an ancestor element article.

To solve this problem, the F&B bisimulation graph was proposed to be the combination of the bisimulation graph and the path index (Kaushik, Bohannon, Naughton & Korth, 2002). Two nodes are F&B bisimular if and only if they have the same rooted path and they are bisimilar. Its formal definition is as follows:

Definition 2 (Backward Bisimulation Graph)

A node *u* is defined as backward bisimilar to *v* if and only if all the following conditions hold:

1. *u* and *v* have the same label.
2. *parent*(*u*) is backward bisimilar to *parent*(*v*).

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

By this definition, it is easy to see that the path index of an XML tree is a backward bisimulation graph of the tree.

Definition 3 (F&B Bisimulation Graph):

Given an XML tree *T*(*N*, *A*) and a labeled graph *G*(*V*, *E*), an XML tree node *u* in *N* is F&B bisimilar to a vertex *v* in *V* (denoted as $u \approx v$) if and only if *u* is forward and backward bisimilar to *v*.

Graph *G* is an F&B bisimulation graph of *T* if and only if *G* is the smallest graph such that every vertex in *G* is F&B bisimilar to a node in *T*.

An F&B bisimulation graph can be constructed by clustering tree nodes that are F&B bisimilar. The construction can be extended from the construction of the bisimulation graph: if two vertices in the bisimulation graph have two different rooted paths, they will be separated into two vertices in the F&B bisimulation graph.

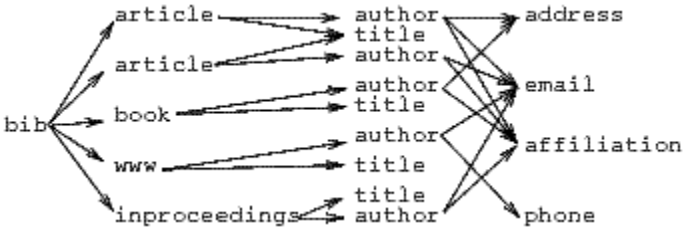

Figure 6. F&B bisimulation graph of XML tree in Figure 4.

It is easy to see that the F&B bisimulation graph is a further refinement of both the path index and the bisimulation graph in that for any node cluster n1 under F&B bisimulation graph there is a node cluster n2 under path index or bisimulation graph such that n1 is the subset of n2. This means that the bisimulation graph can answer more path queries than the path index and the bisimulation graph. Figure 6 is an example of the F&B bisimulation graph of XML tree shown in Figure 4. Note that there are five authors in the F&B bisimulation graph, even though there are only four of them having different subtree structure (verify that the author under the second article has the same subtree structure as the author under inproceedings) as shown in Figure 4.

Because of this finer granularity of clustering, the F&B bisimulation graph can answer query //article/author[email][affiliation].

It is also easy to prove that the F&B bisimulation graph is also a DAG. Therefore, given a path expression, the evaluation algorithm will do a pattern matching on the DAG in the same way as to the bisimulation graph. Since F&B bisimulation graph is a refinement of the bisimulation graph, it is usually larger than the bisimulation graph. For some complex data set, the F&B bisimulation graph could be too large to be fit into main memory. Therefore disk-based storage formats are proposed for F&B bisimulation graph.

To store the F&B bisimulation graph on disk, it is necessary to materialize it into a list of vertices, each of which keeps the pointers to its children. With carefully designed physical storage format, the F&B bisimulation graph can be used to efficiently answer a large set of queries including all queries that can be answered by all the structural indexes mentioned before. This is based on the observation that the F&B bisimulation graph is a refinement of the bisimulation graph and the path index, and both of which are also refinements of the tag-name index. The refinement relationships between these structural indexes form a lattice shown in Figure 7.

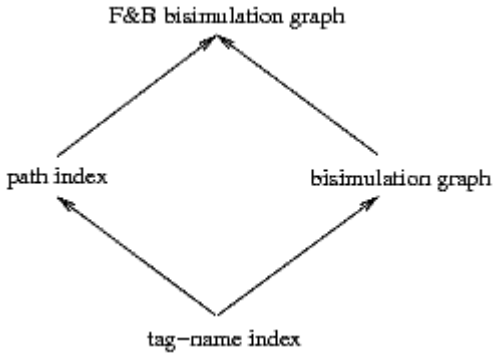

Figure 7. Relationship between different structural indexes.

Based on this observation, when the vertices in the F&B bisimulation graph are stored, further clustering is possible so that the resulting storage format can also be treated as the path index and the tag-name index (Wang, Wang, Lu, Jiang, Lin, & Li, 2005). The clustering criteria are as follows:

1. All XML tree nodes that are F&B bisimilar are clustered into chunks.
2. All chunks having the same path ID are clustered into fragments.
3. All fragments having the same tag names are clustered into tapes.

These three clustering criteria have different granularity, with the chunk being the finest and the tape being the coarsest. It is easy to see that all nodes in the same type have the same tag name and all nodes with the same tag name are clusterd into one tape. Therefore, the tape has the same clustering criterion as the tag-name index, so answering queries such as //author amounts to finding the tape who is clustered based on tag name “author”. Similarly the fragment has the same clustering criterion as the path index. Therefore, it can be used to answer queries that are

suitable for the path indexes. Finally, the chunks are used to answer queries that can only be answered by the F&B bisimulation graph.

Feature-based Index

As seen from the previous subsection, F&B bisimulation graph is the finest granular index and it can answer the largest set of queries. However, this also gives rise to the problem that the F&B bisimulation graph could be very large. In fact, it could grow as large as more than two million vertices for a medium sized (81MB) complex data set Zhang, Özsu, Ilyas & Aboulnaga, 2006. Even with the help of the disk-based clustering format introduced in the previous subsection, answering a twig query such as //author[email][address] will need to perform an expensive tree pattern matching for each vertex in the tape corresponding to tag name author. It is necessary to match against every vertex in the tape because of the need to find all author vertices whose children contains both email and address as a subset.

The feature-based index (Zhang, Özsu, Ilyas & Aboulnaga, 2006) was proposed to solve this problem by computing a list of features (or distinctive properties) of these bisimulation graphs, and use these features as filters to the tree pattern matching. To some extent, this feature-based index is similar to hash-based index in that it has pruning power to eliminate a set of inputs that are guaranteed to be non-match, and then use the more expensive matching algorithm to further check the remaining set.

The general framework of the feature-based index is as follows:

1. At the index creation phase:
    a. Compute the bisimulation graph of the XML tree.
    b. Enumerate all subgraphs of depth k (indexable units) in the data bisimulation graph, and compute their features. The reason to put a limit on the depth is that otherwise there are too many indexable units.
    c. Insert indexable units based on their distinctive features into a B+ tree.
2. At the query execution time:
    a. Compute the bisimulation graph of the query tree.
    b. If the depth of the bisimulation graph is not greater than k, then calculate the features of query bisimulation graph; otherwise, the index cannot be used.
    c. Use the query features to filter out indexed units.
    d. Perform tree pattern matching for the remaining bisimulation vertex.

The key idea of the feature-based index is to come up with the most distinctive features that have the maximum pruning power. To answer a twig query, the features need to be able to distinguish different bisimulation vertices having a certain tag name, and the capability to find subgraphs that are supergraphs of the query graph. The first is trivial since the tag name of the bisimulation graph serves the purpose. The second requirement needs some graph theory properties. In fact the following theorem lays the foundation of the feature-based indexes.

Theorem 1:

Given two graphs G and H, let l_{min} and l_{max} denote the minimum and maximum eigenvalues of the matrix representation of a graph, respectively. Then the following property holds: if H is an induced subgraph of G, then $l_{min}(G) \leq l_{min}(H) \leq l_{max}(H) \leq l_{max}(G)$.

This theorem indicates that the minimum and maximum eigenvalues of all the indexable units can be extracted as the features, and inserted, together with the root tag name, as keys into the B+ tree. Given a query bisimulation graph H, the minimum and maximum eigenvalues can be computed and searched in the B+ tree to find all graphs G such that all the following conditions hold:

1. $root\_label(G) = root\_label(H)$
2. $l_{min}(G) \leq l_{min}(H)$
3. $l_{max}(H) \leq l_{max}(G)$

Since the feature-based index indexes all subgraphs in the bisimulation graphs, it can answer all types of queries that a bisimulation graph can answer. Furthermore, since the features are all numbers, the feature-based index leverages the mature disk-base data structure such as B+ tree in query, creation as well as update. Another advantage of the feature-based index is that that it is very flexible to be extended to support a larger set of queries. For example, if the feature set includes the rooted path ID, then it can answer the query set covered by the F&B bisimulation graph.

CONCLUSION

In this chapter, we introduce state-of-the-art native XML storage, query processing and indexing techniques. These three techniques are closely related in that the design of the storage formats is affected by the access pattern of the XML tree nodes, which can be observed through XML query processing. The design of XML indexes is also dependent on the query processing techniques, but it focuses on how to efficiently execute a subset of queries, without considering a whole spectrum of requirements (e.g., DML, fragment retrieval etc.) that the storage formats have to.

REFERENCES

Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M., Srivastava, D., & Wu, Y. (2002). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering* (pp. 141-152). San Jose, California: IEEE Computer Society.

Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., & Josifovski, V. (2003). Streaming XPath Processing with Forward and Backward Axes. *In Proceedings of*

the 19th International Conference on Data Engineering (pp. 455-466). Banglore, India: IEEE Computer Society.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., & Siméon, J. (2007). XML Path Language (XPath) 2.0. W3C Recommendation, 23 January 2007. Available at http://www.w3.org/TR/xpath20/.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J. (2007). XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. Available at http://www.w3.org/TR/xquery/.

Brantner, M., Helmer, S., Kanne, C. C., & Moerkotte, G. (2005). Full-fledged Algebraic XPath Processing in Natix. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE 2005 (pp. 705-716). Tokyo, Japan: IEEE Computer Society.

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., & Teubner, J. (2006). MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 479-490). Chicago, Illinois: ACM.

Bruno, N., Koudas, N., & Srivastava, D. (2002). Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 310-322). Madison, Wisconsin: ACM.

Fiebig, T., Helmer, S., Kanne, C. C., Mildenberger, J., Moerkotte, G., Schiele, R., & Westmann, T. (2002). Anatomy of a Native XML Base Management System. VLDB Journal, 11-4, 292—314.

Goldman, R., & Widom, J. (1997). DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases* (pp. 436-445). Athens, Greece: ACM.

Henzinger, M. R., Henzinger, T. A., & Kopke, P. W. (1995). Computing Simulation on Finite and Infinite Graphs. In *Proceedings of 36th Annual Symposium on Foundations of Computer Science* (pp. 453-462). Milwaukee, Wisconsin: IEEE Computer Society.

Kaushik, R., Bohannon, P., Naughton, J. F., & Korth, H. F. (2002). Covering Indexing for Branching Path Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 133-144). Madison, Wisconsin: ACM.

Krishnaprasad, M., Liu, Z. H., Manikutty, A., Warner, J. W., & Arora, V. (2005). Towards an industrial strength SQL/XML Infrastructure. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE 2005 (pp. 991-1000). Tokyo, Japan: IEEE Computer Society.

Li, Q., & Moon, B. (2001). Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 26st International Conference on Very Large Data Bases* (pp. 361–370). Roma, Italy: ACM.

Liefke, H., & Suciu, D. (2000). XMill: An Efficient Compressor for XML Data. In Proceedings of *the ACM SIGMOD International Conference on Management of Data* (pp. 153-164). Dallas, Texas: ACM.

Milo, T., & Suciu, D. (1999). Index Structures for Path Expressions. In Proceedings of 7th International Conference on Database Theory (pp. 277-295). Jerusalem, Israel: Springer.

Nicola, M., & Van der Linden, B. (2005). Native XML Support in DB2 Universal Database. In *Proceedings of the 31st International Conference on Very Large Data Bases* (pp. 1164–1174). Trondheim, Norway: ACM.

Pal, S., Cseri, I., Seeliger, O., Rys, M., Schaller, G., Yu, W., Tomic, D., Baras, A., Berg, B., Churin, D., & Kogan, E. (2005). XQuery Implementation in a Relational Database System. In *Proceedings of the 31st International Conference on Very Large Data Bases* (pp. 1175–1186). Trondheim, Norway: ACM.

Rao, P., & Moon, B. (2004). PRIX: Indexing And Querying XML Using Prufer Sequences. In *Proceedings of the 20th International Conference on Data Engineering* (pp. 288-300). Boston, Massachusetts: IEEE Computer Society.

Schkolnick, M. (1977). A Clustering Algorithm for Hierarchical Structures. ACM Transactions on Database Systems (TODS), 2-1, 27-44.

Tolani, P. M., & Haritsa, J. R. (2002). XGRIND: A Query-friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering* (pp. 225-234). San Jose, California: IEEE Computer Society.

Wang, H., Park, S, Fan, W., & Yu, P. (2003). ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In Proceedings of *the ACM SIGMOD International Conference on Management of Data* (pp. 110-121). San Diego, California: ACM.

Wang, W., Wang, H., Lu, H., Jiang, H., Lin, X., & Li, J. (2005). Efficient Processing of XML Path Queries Using the Disk-based FB Index. In *Proceedings of the 31st International Conference on Very Large Data Bases* (pp. 145–156). Trondheim, Norway: ACM.

Zhang, C., Naughton, J. F., DeWitt, D. J., Luo, Q. & Lohman, G. M. (2001). On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (pp. 425-436). Santa Barbara, California: ACM.

Zhang, N., Kacholia, V., & Özsu, M. T. (2004). A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proceedings of the 20th International Conference on Data Engineering* (pp. 54-65). Boston, Massachusetts: IEEE Computer Society.

Zhang, N., Özsu, M. T., Ilyas, I. F., & Aboulnaga, A. (2006). FIX: A Feature-based Indexing Technique for XML Documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (pp. 259-271). Seoul, Korea: ACM.