Efficient Integration of Structure Indexes of XML

Taro L. Saito Shinichi Morishita

University of Tokyo, Japan,
{leo, moris}@cb.k.u-tokyo.ac.jp

Abstract. Several indexing methods have been proposed to encode tree structures and path structures of XML, which are generally called *structure indexes*. To efficiently evaluate XML queries, it is indispensable to integrate tree structure and path structure indexes as a multidimensional index. Previous work of XML indexing have often developed specialized data structures tailored to some query patterns to handle this multidimensionality, however, their availability to the other types of queries has been obscure. Our method is based on the multidimensional index implemented on top of the B+-tree, and also it is general and applicable to the various choice of XML labeling methods. Our extensive experimental results confirm great advantages of our method.

1 Introduction

XML databases require the capability to retrieve nodes by using a variety of structural properties of XML, which are basically derived from *tree structures* of XML such as document order of nodes, subtree, sibling, ancestor, descendant nodes, etc. The other properties are *path structures* of XML, that consist of sequences of tag and attribute names, e.g //news/Japan. These various aspects of XML make its query processing difficult, and index structures for this purpose, which are generally called the *structure indexes* [1], have attracted research attention. Most of the proposed structure indexes aim to efficiently process XPath [2] queries, which is the *de facto* standard for navigating XML. XPath contains a mixture of tree and path traversal with several axis steps, e.g. the child-axis (/), the descendant-axis (//), ancestor-axis, sibling-axis, etc.

Since 1996, as XML gradually has established its position as a data representation format, tremendous number of structure indexes have been proposed, which are optimized for specific query patterns, including structural joins [3, 4], twig queries [5], suffix paths [6], ancestor queries [7], etc. They are proved to be fast for their targeted queries, however, most of them introduce special purpose data structures implemented on disks, and ends up losing *flexibility* of choices of node labels. For example, XR-tree [7], which is optimized for retrieving ancestor nodes that have specific tag names, cannot incorporate other efficient path labels such as *p-labels* [6], which is the fastest for suffix-path queries. That means XR-tree achieves fast ancestor query performance in exchange for the performance of suffix path queries.

Care should be taken to devise a specialized data structure on a disk, since an industrial strength DBMS has to support transaction management, but its implementation cannot be dependent from several essential components of the DBMS; page buffer, lock

manager, database logging for recovery, and also access methods, such as B+-trees or R-trees [8]. These modules seem to be able to implement independently, however, all of them have a lot of interdependencies. Index structures of DBMS usually include intricate protocols for latching, locking, and logging. The B+-trees in serious DBMSs are riddled with calls to the concurrency and recovery code, and this logic is not generic to all access methods [9]. That is a reason why the transaction management of R-tree, which is famous as a multidimensional index structure, is not seriously supported in most of the DBMS products, including both of commercial and open-source programs.

A natural question that follows is whether we can utilize a B+-tree, which is a well-established disk-based data structure, to achieve good performance for various types of XML queries. Our answer to this question is affirmative. In this paper, we show XPath queries can be performed with combinations of only two types of indexes; tree-structure and path-strucure indexes. A challenging problem is that these scans must be performed in a combined way, for example, we have to query ancestor nodes that belong to some suffix path.

Our approach to this problem is to integrate tree-structure and path-structure indexes into a multidimensional index implemented on a B+-tree. It accelerates query processing for complex combinations of structural properties. And also, it is possible to incorporate various types of labeling methods. As an integration approach, constructing multiple secondary B+-tree indexes does not help multidimensional query processing, since they work for only a single dimension, not the combinations of multiple dimensions. Moreover, the existence of multiple secondary indexes not only enlarges the database size, but also deteriorates the update performance. We overcome these obstacles by using space-filling curve technique [10–12] to align XML nodes in a multidimensional space into one-dimensional space so that these nodes can be stored in a single B+-tree. We show this approach is beneficial in both of the query performance and database size.

There are hundreds of combinations of labeling strategies for XML and some of them demand special purpose data structures implemented on disks. What we would like to reveal in this paper is how the *integration* of tree and path structure indexes works for various types of queries consisting of combinations of structural properties.

Our major contributions in this paper are as follows:

- We introduce an efficient multidimensional index structure, which is a combination of existing node labeling strategies in literature. While some XML indexes facilitate a few set of query patterns, our index is adaptive for various types of queries.
- We show an implementation of the proposed multidimensional index on top of the B+-tree, utilizing the space-filling curve technique.
- We show the multidimensional range query algorithm that can be performed without changing the B+-tree implementation.

Based on the above techniques, we have implemented an XML database system called **Xerial** (pronounced as [eksiriəl]) [1]. Our experiments in Section 4 demonstrate Xerial's all-around performance for various types of queries. In spite of this faculty, its index size remains compact.

[1] Our system will be available at http://www.xerial.org/

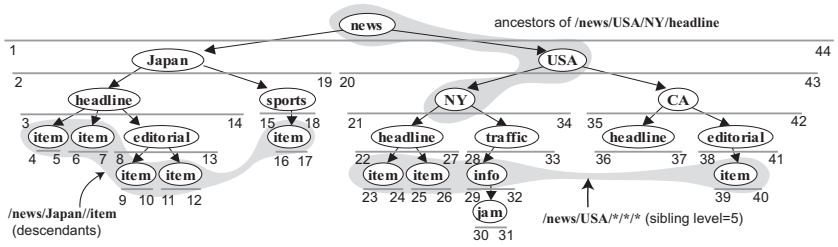


Fig. 1. Interval labels of XML

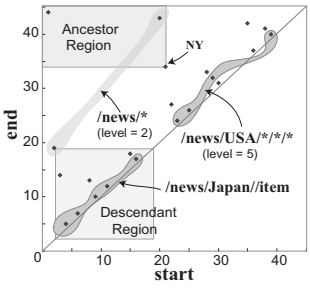
Organization of the rest of the paper is as follows: in Section 2, we explain tree-structure and path-structure indexes of XML, and show examples that motivate the need of multidimensional index for XML In Section 3, we introduces its design and implementation. In Section 4, we provide results of experimental evaluation. Finally, we report related work in Section 5 and conclude in Section 6.

2 Backgrounds

Tree-Structure Indexes. XML has a tree structure, however, ancestor and descendant axes cannot be efficiently evaluated with standard tree navigations such as depth-first or breadth-first traversals. To make faster the process of ancestor-descendant queries, two types of node labeling methods have been developed; the *interval label* [3] and the *prefix label* [13]. The interval label (see also **Fig. 1**) utilizes containment of intervals to represent ancestor-descendant relationships of XML nodes. For example, if an interval label of a node p contains another interval of a node q, p is an ancestor of q. The prefix label assigns node id paths from the root node to each node so that if the label of a node p is a prefix of the label of a node q, p is an ancestor of q. Both of the node labeling strategies are fundamentally same in that they are designed to instantly detect ancestor-descendant relationships of two XML nodes. This operation is called *structural joins* [14]. **Fig. 1** shows an example of the interval label. The interval assigned to each node subsumes intervals of its child and descendant nodes. These node labels are favorable in that they can be aligned in the document order of nodes by seeing start values of these intervals. Therefore, these labels can also be used to traverse the tree structures of XML in both of the depth-first and breadth-first manner. We call this type of indexes for tree navigation, *tree-structure* indexes.

Path-Structure Indexes. To efficiently evaluate path expression queries, in addition to the tree-structure indexes, we also need the *path-structure* indexes, which reduce the overhead of tree navigation by clustering nodes that belong to the same tag or attribute name paths. There are many proposals how to encode path structures of XML, varying from simply assigning an integer id to each independent path in the XML document [1] to creating a summary graph of path structures [15, 16].

When a query contains the descendant-axis (//), we can localize the search spaces for the descendant nodes. For example, an XPath query //Japan//item can be decom-



path ID	inverted path
1	news\
2	Japan\news\
3	headline\Japan\news\
4	headline\NY\USA\news\
5	headline\CA\USA\news\
6	item\headline\Japan\news\
7	item\headline\NY\USA\news\
8	item\editorial\Japan\news\
9	item\editorial\CA\USA\news\
10	item\sports\Japan\news\

path ID	inverted path
11	editorial\Japan\news\
12	editorial\CA\USA\news\
13	sports\Japan\news\
14	USA\news\
15	NY\USA\news\
16	traffic\NY\USA\news\
17	info\traffic\NY\USA\news\
18	jam\info\traffic\NY\USA\news\
19	CA\USA\news\

Fig. 2. Projection of the interval labels on a 2D-plain (left). Inverted path labels (right).

posed into two paths; //Japan and //item, and the search space for //item will be localized according to the results of //Japan (**Fig. 1** and **Fig. 2**). Therefore, the tree-structure and path-structure indexes should be integrated to evaluate these types of queries.

In addition, we usually have to query not only with tag names of XML nodes but also with *suffix paths*. For example, an XPath //Japan//headline/item contains a suffix path //headline/item. Rather than querying headline and item nodes individually, it is far more efficient to scan nodes that have suffix paths //headline/item directly, since there are many item nodes whose parents are not the headline nodes. To improve accessibility to suffix paths, the p-label has been proposed [6]. The essence of its technique is to invert the sequences of paths occurring in the XML document, which we call *inverted paths*, and align these inverted paths in the lexicographical order considering each tag or attribute name in the paths as a comparison unit. **Fig. 2** shows an example of inverted paths, where each inverted path is labeled with an integer id. To evaluate an XPath query //item, we have to collect nodes whose inverted path ids are contained in the range $[6, 11)$. When a more detailed path is specified, for example, //headline/item, the query range narrows to $[6, 8)$. With the inverted path ids, we can perform a suffix path query with a range search.

Multidimensional Aspects of XML. Here, we show why the integration of tree-structure and path-structure indexes is so important. **Fig. 2** shows the mapping of the intervals (start, end) in **Fig. 1** into a two-dimensional plane. A benefit of the interval labeling is that we can enclose all ancestor (descendant) nodes of some nodes within its upper left (lower right) rectangular region. For example, all ancestor nodes of the NY node $(21, 34)$ is enclosed in its upper left rectangle. The process of a query, say /news/Japan//item, has to accurately extract item nodes within the subtree rooted by Japan (a shaded region in the figures). Since some item nodes exist out of this region, the index structure for XML demands the capability to capture nodes by the combination of start, end, and a path.

Although the 2D plane is useful to track ancestor and descendant nodes, we also need the information of the node depth (level) to process wild-cards in path expressions. For example, XPath expressions /news/* and /news/US/*/*/*, which are useful to investigate the structure of XML database, require the level information to efficiently collect the answer nodes, since without indexes for the level values, its process has to traverse the tree-structure in depth-first or breadth-first manner; it is inefficient when the

depth is deep. Our experimental results confirm the inefficiency of these tree traversal methods for wild-card queries, i.e. the sibling-axis steps.

XPath [2] has 11 types of axis steps for tree navigations, that are *ancestor*, *descendant*, *parent*, *child*, *attribute*, *preceding-sibling*, *following-sibling*, *ancestor-or-self*, *descendant-or-self*, *preceding*, and *following*[2]. Among them, the six types of axis steps, ancestor(-or-self), descendant(-or-self), preceding and following, can be processed with the two-dimensional indexes for the interval labels (start, end). The parent, child, preceding-sibling and following-sibling axis-steps require all of start, end, level values, since start and end values are not sufficient to detect parent-child relationships of nodes. If attribute nodes of XML are modeled as child nodes of tags, the attribute-axis can be seen the same with the child-axis. Therefore, all of 11 axis-steps can be processed with the combination of (start, end, level) indexes, i.e. tree-structure indexes.

In addition to the tree-structure indexes, if we have the path-structure indexes, we can efficiently answer XPath queries, even if these answers are contained in meandering regions as illustrated in the query region for /news/Japan//item in **Fig. 2**. Therefore, multidimensionally indexing tree-structures and path-structures of XML is a key to accelerate XML query processing.

3 Multidimensional XML Index

In order to construct XML databases, we utilize a combination of tree-structure indexes, and path-structure indexes. Although, there are many proposals for labeling each type of index, we adopted labels that can be easily represented with integers.

We encode every XML node with a label:

(start, end, level, path, text),

where start and end represent interval labels of XML, and level is the node depth. The path is the inverted path id described in Section 2. The text is a text content enclosed in the tag or attribute. Every attribute element in XML is assigned the same interval and level value with its belonging tag, so as to learn the subtree range of the tag from the index of the attribute node.

Although we utilized the interval labels for tree structures, other labeling schemes, such as prefix labels, can substitute them; the XML label will be (prefix-label, level, path, text). Each prefix label contains all prefix labels of its ancestor nodes, so there is no need to have end values for ancestor queries. The path labels also can be replaced simply with tag IDs or other labels.

The above labeling scheme is used to create multidimensional indexes. To index multidimensional data, it is general to use R-tree, which groups together nodes that are in close spatial proximity. However, implementations of the R-tree are not yet as matured as the B+-tree, which is broadly employed in the industrial strength DBMSs, while the R-tree is not. Although the B+-tree is a one-dimensional index structure, we

[2] The other two axis steps defined in XPath [2] are the *namespace* and *self* axis, which do not require any tree traversal.

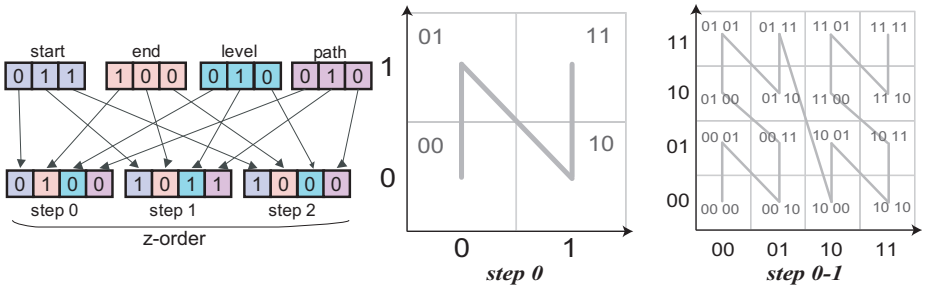


Fig. 3. Interleave function generates a z-order from an index (start, end, level, path), that specifies a position on the z-curve.

can store the multidimensional data into a B+-tree by using a space-filling curve [12], such as Hilbert curve, Peano curve etc. The space-filling curve traces the entire multidimensional space with a single stroke, and it can be used to align multidimensional points in one dimensional space.

However, what kind of space-filling curve is suited for XML indexing? To answer this question, let us confirm our objective to construct a multidimensional index; that is to make clusters of nodes that have same attribute values as possible, for example, same level values and same suffix paths, so that we can efficiently query nodes with combinations of these attribute values, i.e. start, end, level and path.

To meet this demand, we chose a straightforward approach; *bit-interleaving* of coordinate values. It gives a position on the *z-curve* [10, 12], which is also a space-filling curve. The interleave function illustrated in **Fig. 3** receives coordinate values of a point as input, and from their bit-string representations, it retrieves single bits from heads of coordinate values in a round-robin manner, then computes the *z-order*, which is an absolute position on the z-curve (**Fig. 3**). This linear ordering of XML nodes enables us to implement the multidimensional index on top of the B+-tree. In addition, each step in the z-order in **Fig. 3** has a role to split each dimension. The first step splits each dimension into two, and the second step split each slice into 2, resulting in $2^2 = 4$ slices, and so on. If two nodes are close in the multidimensional space, their z-orders also likely to be close in the some steps. It means these nodes will be probably placed in the same leaf page or its proximate pages in the B+-tree; this property is the nature of bit-interleaving.

Normalizing Index Resolution. The interleave function extracts bits beginning from the MSB (most significant bit). When value domains of the interleaved indexes are far different, for example, the domain of start values is $0 \leq start < 2^{10}$, and that of level values is $0 \leq level < 2^3$, the change of a value in a smaller domain has as equal significance to the z-order as that of the larger domains. In general, the depth of XML documents is not greater than 100, while the interval label for XML requires as large a value as the number of nodes, which can be more than 100,000. Thus, if we use the same bit-length number to represent each index value, the level values are less important in the z-order, and we fail to separate XML nodes level by level, deteriorating the sibling query performance.

To avoid this problem, we adjust the *resolution* of each index, which is the maximum bit length that is enough to represent all values in the index domain. We denote the resolution of an index as r. For example, when a domain of some index is a range $[0, v_{max})$, its resolution r is $\lceil \log_2 v_{max} \rceil$. The $normalize_m(v)$ function converts an integer value v, whose resolution is r, into an m-bit integer value. We define $normalize_m(v) = \lfloor v/2^{r-m} \rfloor$, ignoring the fraction. For example, when $m = 8$ and the resolution of each index of (start, end, level, path) is 10, 10, 3 and 4, respectively, an XML index (100, 105, 3, 2) is normalized to the 8-bit values (25, 26, 96, 32). By using normalized index values to compute z-orders, we can adjust the resolution so that level or path values, whose domains are usually small, affect much more to the z-order than start or end values. We simply denote this normalization process for some node p as $normalize(p)$.

Range Query Algorithm. The idea that utilizing z-curve for multidimensional indexes is first mentioned in the zkd-BTree [10] and is improved in the UB-tree [17]. Although both of them extended the standard B+-tree structure to make it efficient for multidimensional queries, we introduce a multidimensional range query processing algorithm without modifying existing B+-tree structures. In our algorithm, we need only two standard functions for the B+-tree; *find* and *next*. The $find(k)$ receives an key value k and finds the smallest entry whose key value is greater than or equal to k. The $next(e)$ returns the next entry of an entry e in the B+-tree.

We denote the z-order of a node p = (start, end, level, path) as $zorder(p)$, and coordinates specified by the z-order z as $coord(z)$. Then, $coord(zorder(p)) = p$. Each entry in the B+-tree has the structure: $zorder(normalize(p)) \Rightarrow p$, where the left-hand side is a key to be used to sort XML nodes in the z-order. To perform a multidimensional query for a hyper-rectangle region $Q(p_s, p_e)$, where p_s and p_e are the multidimensional points specifying the beginning and end points of the query range, we can utilize a property of z-orders; all points p in the query range Q satisfies $zorder(p_s) \leq zorder(p) \leq zorder(p_e)$ [17].

Algorithm 1 shows the range query algorithm, and **Fig. 4** illustrates its behavior. Since all nodes are aligned in the z-order in the B+-tree, we have to scan the key range of z-order from $zorder(normalize(p_s))$ to $zorder(normalize_{ceil}(p_e))$, where $normalize_{ceil}$ is calculated from $\lceil v/2^{r-m} \rceil$ of each coordinate value v. That z-orders computed from normalized coordinate values may have round errors, so there is a case that $coord(normalize(p))$ is contained in the normalized query range $NQ(normalize(p_s), normalize_{ceil}(p_e))$, but p is not in Q, since if we de-normalize NQ, illustrated in **Fig. 4** as pseudo-query range, it is always equal to or larger than Q. Even though, the containment test for NQ (Step 10) is useful to detect whether the current z-order is completely out of range of Q. In this case, we can compute the *nextZorder* that re-enters into the query box NQ (Step 17). It skips some nodes in the outside of the query box and saves disk I/O costs. An efficient algorithm to compute next z-orders is described in [18]; this algorithm locates the most-significant bit-position, say j, in the z-order that can be safely set to one without jumping out of the query range, then adjusts other bit values which are lower than j so that the z-order becomes the smallest one contained in the query range but larger than the original z-order.

Algorithm 1 Range query algorithm

Input: $Q(p_s, p_e)$: query range
Output: A node set within the query range
1: $NQ = (normalize(p_s), normalize_{ceil}(p_e))$ // normalized query range of Q
2: $z_s = zorder(p_s)$, $z_e = zorder(p_e)$
3: $z = z_s$ // set the initial z-order to the beginning of the query range
4: // find an entry e in the B+-tree that has the smallest z-order larger than z.
5: $e = find(z)$
6: **while** e is not nil **do**
7: $z = e.z$ // $e.z$ is the z-order (key value) of the entry e
8: **if** $z > z_e$ **then**
9: return // end of the query
10: **if** $coord(z)$ is contained in NQ **then**
11: **while** e is not nil and $e.z == z$ **do**
12: // retrieve nodes whose z-order is z in the B+-tree
13: **if** the entry e is contained in Q **then**
14: output e
15: $e = next(e)$ // move to the next entry of e in the B+-tree
16: **else**
17: $nextZorder$ = the smallest z-order larger than z and contained in NQ.
18: $e = find(nextZorder)$

4 Experimental Evaluation

We evaluated the query performance of Xerial for several kinds of queries, e.g. ancestor, descendant, sibling, and path-suffix queries, which are the basic components to process more complicated queries such as structural joins, twig-queries, etc.

To clarify the benefit of our method, we prepared two competitors for Xerial; start index and path-start index. The start index simply sorts XML nodes in the order of start values. It has the data structure (start ⇒ end, level, path, text) in B+-tree. The path-start index, ((path, start) ⇒ end, level, text), sorts nodes first by path, then by start orders. These structure can localize search space of path queries within some subtree range, and similar structure is utilized in [4]. However, the following experiments reveal that such simple integration of indexes has several weak points.

Implementation. All of the indexes are implemented in C++. Xerial's index structure is *z-order* ⇒ (start, end, level, path, text). Every z-order is represented with 64-bit integer, and it is a sort key in the B+-tree. And also,all indexes hold start, end, level and path values as 32-bit integers. To construct B+-trees, we used the BerkeleyDB library [19], and their page sizes are set to 1K.

Machine Environment. As a test vehicle, we used an Windows XP, Pentium M 2GHz notebook with 1GB main memory and 5,400 rpm HDD (100GB).

Database Size. We compared database sizes of start index and Xerial. **Fig. 4** shows their actual database sizes and construction times for various scaling factors (1 to 10) of the XMark's benchmark XML documents [20]. The secondary index in **Fig. 4** shows the database size if we constructs three B+-tree indexes for end, level, path values to complement the functionality of the start index. Even though Xerial has additional z-orders, its database size is almost the same with the start index, and also it is much more compact than creating multiple secondary indexes. It is mainly because the B+-tree of Xerial has many duplicate entries having same z-orders, and it makes lower the depth of the B+-tree.

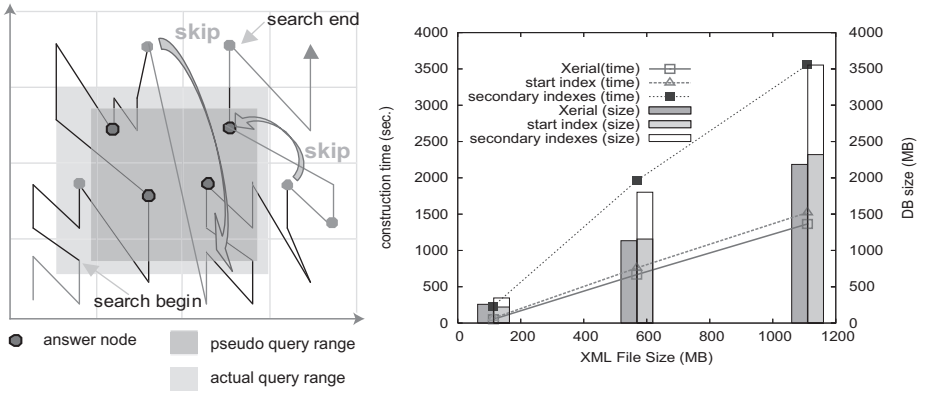


Fig. 4. Range query algorithm (left). DB construction time and DB size (right)

The following experiments are conducted on a XMark document (113MB, scaling factor = 1.0), and we measured the average times for individual query operations, ignoring the output costs of reporting the query results.

Suffix Path Query. First, we compared performance of suffix path queries. **Fig. 5** shows how fast each index can collect nodes that have the same path suffixes. The path-start index, which has clusters of suffix paths is the fastest, and Xerial performs as fast as the path-start index, because the interleave function of Xerial also plays a role to group together nodes which have the same suffix path. The start index is weak in processing this kind of query since it has to scan the whole index, since information of path is hidden in its data pages.

In order to show that the importance of having flexibility for the choice of node labels, we also compared the performance of suffix path queries when inverted path cannot be used. The tag-start index uses tag IDs instead of inverted path IDs, so it must perform several nested structural joins [14] to achieve the answer, and shows poor performance other than the $Q3$, that is the tag-only query.

Subtree Retrieval. The start index is the most suitable data structure for subtree retrievals because nodes in a subtree are sequentially ordered. It shows the fastest result (**Fig. 5**). Nevertheless, both of Xerial and path-start index show almost identical performance to the start index.

Ancestor Retrieval. Ancestor query is useful to retrieve parent or ancestor information from some node directly accessed from additional secondary index structures such as the one for traversing IDREF edges, or inverted indexes for text contents. This query needs to find nodes which satisfy $start < s \wedge e < end$, where (s, e) are start and end position of the base node of the query. **Fig. 6** shows the performance of the ancestor queries for various positions of base nodes, whose level is 12. The start index processes this query from the root node, and it skips subtrees which are not the ancestor of the base node. The performance of Xerial is stable, because it can eliminate the search space by using a combination of start and end axes. On the other hand, the path-start index breaks the start order down into multiple clusters grouped by path IDs. Consequently,

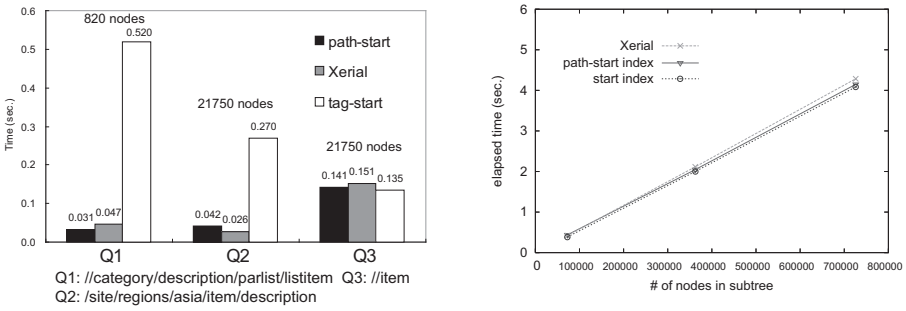


Fig. 5. Suffix-path (left) and subtree (right) query performance

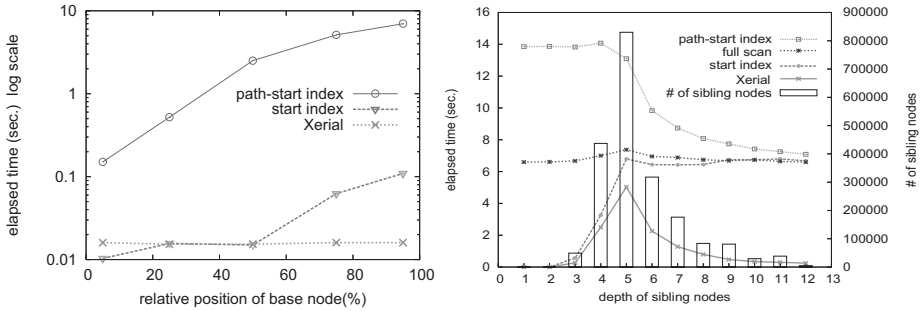


Fig. 6. Ancestor (left) and sibling (right) query performance

it cannot utilize the tree structure of XML. In addition, it cannot eliminate the search space by using the end values, therefore it is inefficient when the base node of the query has a lot of preceding nodes in the document order. The start index has the same deficit. This result indicates that the ancestor query performance of start and path-start indexes depends on the database size.

Sibling Retrieval. Notable usage of sibling node retrievals is to find blank spaces for node insertions, to compute parent-child joins and wild-card(*) queries. Xerial remarkably outperforms the other indexes (**Fig. 6**). This is because these indexes except Xerial have difficulty to find nodes in the target level. The start index must repeat searching the tree for a node in the target level with a depth-first traversal, while skipping unrelated descendant nodes occasionally. The path-start index performs this process in every cluster of paths. This descendant skip works well when the target depth of sibling is low; however, as the level becomes deeper, it cannot skip so many descendants and the cost of the B+-tree searches increases. To see this inefficiency, we also provided the result using sequential scan of the entire index, and it shows similar performance to the start index and path-start index for deep levels.

In summary, to efficiently process queries of suffix paths, siblings, subtrees and ancestors, the start-index and the path-start index require additional secondary indexes.

For example, start index should have indexes for level and path, and path-start index needs at least three indexes for end, level, and suffix path. Xerial has the ability to process all of these queries, and the fact it does not use any secondary index is beneficial to the database size and also to the costs of index maintenance due to updates.

5 Discussions & Related Works

Although the above experiments show advantages of our methods, we would like to mention some tips that finally lead us to this performance. At first, we used 32-bit integers to represent z-orders, but this implementations performs poorly for every types of queries in the experiments. This is because the 32-bit z-order splits each dimension only to 2^8 grids. It is too coarse and results in that too many nodes are assigned the same z-orders; there are many overflowed B+-tree pages and it slows down every search operations. On the other hand, the resolution becomes the finest when every point in the multidimensional space has a unique z-order. However, its bit length might be too long, and such key values will soon fill internal pages of the B+-tree, ending up lowering the B+-tree's branching factors. The optimal resolution is to make *each disk page* have a unique z-order. To achieve this, the UB-tree [17] has to extend the B+-tree implementation.

The use of the UB-tree [17] to index XML documents is proposed in [21]. Its coordinates are combinations of text values, document IDs, and paths and their appearance orders generated from DTDs. Although this method cannot handle suffix path queries etc., the integration of text values is an interesting approach that we do not have mentioned in this paper. Note that, however, if we integrate text values to the index structure, every update to the text values invokes subsequent maintenance of the indexes. Kaushik et. al. proposed efficient algorithms to process queries containing predicates for text values [1]. Their approach assumes text value indexes are maintained separately from structure indexes, so it is more promising in that it can leverage traditional IR technologies to index text contents of XML.

We have not mentioned the updatability of the XML indexes. In fact, integer intervals are weak for updates, since blank space for future node insertions will be exhausted. There are some proposals to make these labels tolerant for node insertions, including ORDPATH [13] etc. As long as we can define the total order on node labels, it is possible to incorporate these labeling strategies to our method.

6 Conclusions & Future Work

In this paper, we proposed an efficient method to integrate tree-structure and path-structure indexes for XML. The proposed indexing method provides efficient processing of ancestor, descendant, sibling, suffix path queries etc. In addition, our index structure and multidimensional range query algorithm can be implemented on top of the standard B+-tree. Our experimental results show advantages and disadvantages of query processing due to the indexing methods. Other queries not targeted in this paper are references by using IDREF edges or inverted indexes for the text contents. It is worth investigating to incorporate such additional index structures into Xerial.

References

1. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: ICDE. (2004) 829
2. Clark, J., DeRose, S.: XML path language (XPath) version 1.0 (1999) available at http://www.w3.org/TR/xpath.
3. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: proc. of VLDB. (2001)
4. Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: proc. of VLDB. (2002)
5. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed xml documents. In: proc. of VLDB. (2003)
6. Yi Chen, S.B.D., Zheng, Y.: BLAS: An efficient XPath processing system. In: proc. of SIGMOD. (2004)
7. Jiang, H., Lu, H., Wang, W., , Ooi, B.C.: XR-Tree: Indexing xml data for efficient structural joins. In: proc. of ICDE. (2002)
8. Gray, J., Reuter, A.: Transaction Processing - Concepts and Techniques. Morgan Kaufmann (1993)
9. Hellerstein, J.M., Stonebraker, M.: Readings in Database Systems. Forth Edition. MIT Press (2005)
10. Orenstein, J.A., Merrett, T.H.: A class of data structures for associative searching. In: proc. of PODS. (1984)
11. Lawder, J.K., King, P.J.H.: Querying multi-dimensional data indexed using the Hilbert space-filling curve. SIGMOD Record **30**(1) (2001)
12. Sagan, H.: Space-Filling Curves. Springer-Verlag New York, Inc (1994)
13. O'Neil, P., O'Neil, E., pal, S., Cseri, I., Schaller, C.: Ordpaths: Insert-friendly xml node labels. In: proc. of SIGMOD. (2004)
14. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M.: Structural joins: A primitive for efficient XML query pattern matching. In: proc. of ICDE. (2002)
15. Goldman, R., Widom, J.: DataGuides: Enabling query formulation and optimization in semistructured databases. In: proc. of VLDB. (1997)
16. Milo, T., Suciu, D.: Index structures for path expressions. In: Database Theory - ICDT 99. Volume 1540 of Lecture Notes in Computer Science. (1999)
17. Bayer, R., Markl, V.: The UB-tree: Performance of multidimensional range queries. Technical report (1998)
18. Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., Bayer, R.: Integrating the UB-tree into a database system kernel. In: proc. of VLDB. (2000)
19. Sleepycat Software: (BerkeleyDB) available at http://www.sleepycat.com/.
20. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., manolesch, I., Busse, R.: XMark: A benchmark for XML data management. In: proc. of VLDB. (2002)
21. Bauer, M.G., Ramsak, F., Bayer, R.: Indexing XML as a multidimensional problem. Technical report (2002) TUM-I0203.