Xerial: An Update Tolerant and High Concurrent XML Database

Taro L. Saito **Shinichi Morishita**
University of Tokyo
Office 244, Faculty of Science Bldg. 1(Old),
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
TEL/FAX: +81-3-5841-4693
{leo, moris}@cb.k.u-tokyo.ac.jp

Abstract

XML has a variety of properties such as tree structures, document order of nodes, a tag name accompanied each node, etc. These various aspects of XML make query processing difficult, and index structures for this purpose have attracted research attention. On the other hand, updates and concurrency control in XML have usually been left untouched. However, the design of the index structures and its concurrency control must not be discussed separately, since the index structures affect the granularity of locks. Naive indexing methods often cannot localize locks and lead to the loss of concurrency.

In this paper, we propose a dynamic index structure that is efficient in both of updates and query processing, and to minimize lock regions, we also present a new locking method, called adaptive granular locking, which utilizes hyper-rectangular locks in multidimensional space. In addition, organization of hyper-rectangular locks and page-level locks into proper layers achieves dramatic reduction of deadlock states in comparison with non-layered transaction model. ***Xerial****, an XML database system, was implemented on the basis of these techniques. Our extensive experimental results confirm its great advantages in transaction processing.*

To our knowledge, this is the first time that the issue of concurrency control in XML is seriously discussed with careful consideration of the design of the index structure.

1 Introduction

Updating XML is a challenging task. As for XML documents which are so small as they can be fully loaded into the main memory, updating them is not a difficult problem. However, for XML documents larger than the main-memory capacity, in order to locate XML nodes to update, some index structures on top of the secondary storage are needed. The difficulty of indexing XML lies in the facts that it must preserve not only tree-structures, but also document order of nodes, and it must process path-expression queries described by XPath [7], which is the *de facto* standard for navigating XML. XPath contains path traversal expressions with several axes, such as child(/), descendant(//), ancestor, sibling, etc. To handle these axes in XPath, the index structure must have capability to decide such structural relationship between given nodes instantly.

As an example to motivate transaction management in XML, we introduce a simplified version of an XML document, news.xml (**Figure 1**). The news.xml represents a web site broadcasting news to Japan and USA. The USA node has two states NY and CA, where their local news are provided. Each region has several kinds of news topics, e.g. headline, sports, traffic, editorial, etc. Item nodes represent news articles, and has some text contents (ommited in the figure). Consider the situation that reporters insert new items time-to-time, and many web-page readers access to arbitrary news contents. And also, old items will be removed periodically from the headlines and moved to archives.

To efficiently process queries of ancestor-descendant relationships, e.g. Japan//item, the use of interval encoding of tree structures (see also Figure 1) has become prominent in recent years [1, 19, 28]. With this labeling, the detection of ancestor-descendant relationship becomes equivalent to see the inclusion of intervals. In addition, this labeling method is favorable in that it can preserve document order of nodes, however, it is not tolerant of insertions of new nodes. Our solution to this problem is to label each interval with *extensible composite-numbers*, which are variable length numbers, and allows an arbitrary number of node insertions.

Another problem is insufficiency of the interval labeling; i.e. there is no information about depth of nodes and path structures consisting of tag names. The depth is essential to see the parent-child relationship. As for path structures, some approaches make groups of nodes separated by their tag names, and label nodes with intervals. The process of path expression queries, e.g. Japan//item, starts with retrievals of two node groups belonging to Japan and item from secondary storages, then computes their ancestor-descendant relationship. This algorithm is called *strucutal join* [1]. Its drawback, however, is that, even when some nodes have never been joined, it loads whole nodes in the groups.

Our motivation of this research comes from the expectations that by integrating both of tree and path structures into the database index, we can reduce unnecessary disk scans. In addition, it will yield the localization of lock regions, and as its outcome, better transaction performance. As an integration approach, however, constructing secondary indexes does not help such structural optimization since they work for only a single dimension, not the combinations of multiple dimensions. Moreover, the existence of multiple secondary indexes imposes heavy load on every update transaction to synchronize them.

Therefore, without using any secondary index, we designed an update tolerant multidimensional index with which we can pack target nodes of path expression queries into hyper-rectangular regions in the multidimensional space. This encapsulation of nodes is also useful to define appropriate lock granules for XML. To implement this hyper-rectangular locks, we devised a specialized lock manager that tests intersections of hyper-rectangles efficiently and also prevents live-locks and dead-locks.

Our major contributions in this paper are as follows:

- We provide a robust XML index for dynamic updates. It enables an arbitrary number of node insertions within any interval and avoids relocation of intervals.
- While some XML indexes support only descendant or tree-traversal queries, our index also facilitates ancestor, sibling and path-suffix queries. In spite of this multidimensionality, the index size remains compact.
- Queries in XML usually access a lot of nodes. On behalf of taking a fine-grained lock for each of them, we use locks for

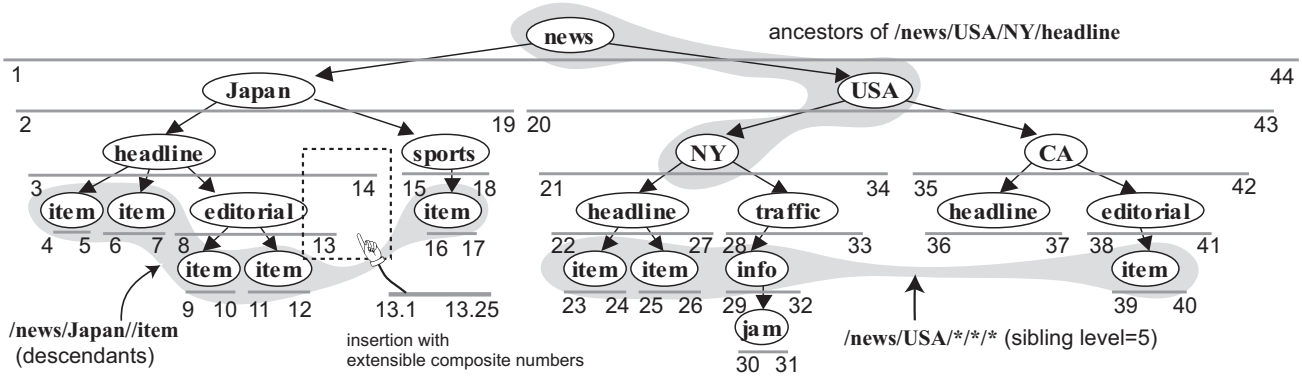

Figure 1: Tree structure of news.xml and its interval representaion.

hyper-rectangular regions. It significantly reduces the number of locks acquired by transactions.

- By utilizing the layered locking [27], we implemented a lock manager which consists of page-level locks and hyper-rectangular locks. With this approach, we succeeded in reducing the occasion entering dead-lock states, thereby increasing the transaction throughput.

By using these techniques, we developed a transaction enabled XML database system called **Xerial** (pronounce as [eksíriəl]). The name of Xerial comes from XML and ensuring *serializable* execution of transactions. The index structure and lock management of Xerial allow us to efficiently process concurrent transactions.

It should be noted that quite a few researches have studied on concurrency control for XML, since it has been usual that the settings of XML databases are static, or XML documents are embedded into relational databases and the concurrency control is left to them. As a consequence, the needs for concurrency control specific to XML have been seldom mentioned.

To the best of our knowledge, S. Helmer et al. were the first that pointed out the inefficiency of concurrency control in such embedded XML databases, and they implemented node and edge level locking for XML [15]. T. Grabs et al. proposed a lock manager on the DataGuide [11] in [12]. Extension of the granular-locking for XML is proposed in [17]. However, they all avoid processing of descendant queries. Moreover, the experimental results of the node and edge level locking in [15] indicate its negative aspect that suffers high percentage of deadlocks. Precision locking for XML [6] needs a lot of copies of XML documents and its efficiency is not proved yet. Therefore, we believe that this paper presents the first serious proposal that realizes a concurrency-control method for XML which can handle descendant queries and deadlocks.

Organization of the rest of the paper is as follows: in Section 2, we show examples that motivate the need for proper indexes and lock granules for XML. Section 3 introduces our novel multidimensional index called XerialMDI and its components. Then, we describe the concurrency-control and lock management of Xerial and some keys to exploit transaction performance in Section 4. In Section 5, we provide results of experimental evaluation. Finally, we report related work in Section 6 and conclude in Section 7.

2 Motivating Examples

2.1 Multidimensional Aspects of XML

The benefit of interval labeling becomes clear by seeing mapping of intervals, (start, end), into a two-dimensional plane. In **Figure 2**, ancestor and descendant nodes of some context nodes can be enclosed within upper left or lower right rectangular regions,

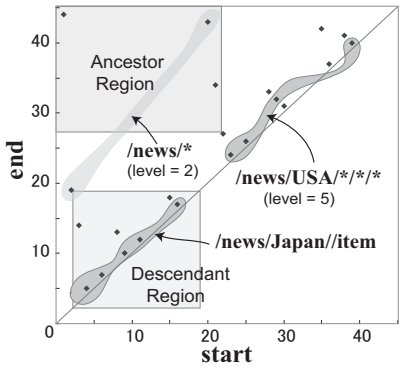

Figure 2: 2D representaion of news.xml

respectively. The process of query, say /news/Japan//item, has to accurately extract item nodes within the subtree rooted by Japan (a shaded region in the figures). Since item nodes exist out of this region (see **Figure 1**), index structure for XML demands the capability to capture nodes by the combination of start, end, and tag name values.

In addition, the use of wild-cards in path expressions, e.g. /news/*, requires level(depth) information of nodes in the tree. Actually, without any index on the level values, this query can be simply processed by starting with finding the Japan node with a depth-first traversal, then skipping the descendant nodes between (2, 19), and it reaches the USA node. However, consider a more complex but useful query, such as /news/USA/*/*/*, for retrieving news contents within arbitrary states and topics. Our experimental results confirm that the above-mentioned traversal approach does not work well, and Xerial which has multidimensional index consisting of start, end, level and tag name path shows an order of magnitude faster performance.

2.2 Granularity of Locks for XML

Traditional approaches to handle concurrency control in XML are, first, to translate XML data into relational database (RDB) tables [10], then leave the concurrency control to the existing techniques of RDB. Let us consider what kind of the lock granule is available in such databases. If XML nodes are distributed into distinct tables for each tag name, its available lock granule is node-level or tag-name level, while lock granules related to tree structure, namely, subtree-level locks are not available. On the other hand, if each row in the table contains some fragments of the XML document, i.e. subtrees, taking into account the fact that the state-of-the-art concurrency control in RDB is the row-level locking [27], it only exposures subtree-level concurrency. It is too coarse, since, for example, the least common ancestor of //item in **Figure 1** is

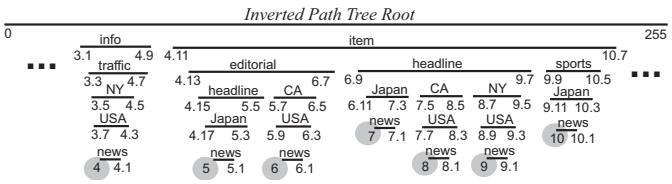

Figure 3: Inverted Path Tree

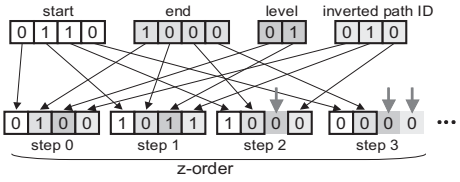

Figure 4: Behavior of the interleave function.

just the root node, thus the lock granule next larger than the node level becomes the entire tree.

Our approach that utilizes hyper-rectangular regions in the multidimensionally indexed space as lock granules gives highly adaptive locks for XML, since even if target regions of queries are meandering as the shaded regions in **Figure 2**, there exist hyper-rectangular regions that can tightly capture them.

3 Indexing XML

3.1 Extensible Composite Number

The interval labeling has a defect that node insertions easily exhaust intervals. In such cases, intervals must be expanded to make room for insertions. However, such updates usually cause cascading relocation of intervals and lead to poor transaction performance. Furthermore, to our knowledge, the problem of static estimation of such costs remains open. A proposal using floating-point numbers for intervals [16] does not solve this problem since they are also finite in computers.Thus, interval labels which remain *stable* after node insertions are required.

To give such stable intervals, we introduce *extensible composite-number*. It is denoted by $C = c_1.c_2.\ldots.c_n$ $(n > 0, c_i \in D)$, a dot-separated list of non-negative integers that are taken from some domain D, for instance, the set of short unsigned integers $\{0, 1, ..., 255\}$. The order of two extensible composite-numbers is given by comparing each composite from head to tail. For convenience of the bit-interleaving that will be described later, all composite-numbers are virtually padded by an infinite sequence of 0s.

For example, when $C_1 = 2.3.5$, $C_2 = 2.4.1$, $C_3 = 1.5$, and $C_4 = 1.5.0$, then $C_1 < C_2$, and $C_3 = C_4$. Note that C_3 and C_4 are different in their lengths, but both mean the same number. The box enclosed in dotted lines in **Figure 1** shows an example of inserting a new interval, $[13.1, 13.25]$. It seems to fill the space between 13 and 13.1; however, the extension of these numbers, i.e. 13.0.1 and 13.1.255, provides additional capacity.

3.2 Inverted Path Tree

XML itself is a verbose language since it usually contains a lot of same tags and paths, thus it is inefficient to store every path to the database as it is. To compress the path information, DataGuides [11] uses a path tree which aggregates common paths in the XML document. However, it lacks the capability to handle descendant-axis(//) queries. This class of queries includes path-suffix queries such as //A, //A/B, etc.

To improve accessibility to path suffixes, we devised a new data structure, the *inverted path tree*. An inverted path is a sequence of tag names from a leaf node to the root. Similar to the interval representation of XML, the inverted path tree is also an interval tree using extensible composite numbers. **Figure 3** is a part of the inverted path tree of Figure 1. In this structure, every sequence of tag names from some interval to the root represents a path suffix in the XML document. For example, the interval of item $(4.11, 10.7)$ indicates the range of //item, and editorial $(4.13, 6.7)$ is the range of //editorial/item. We treat a start order of a leaf interval as an ID of the corresponding inverted path denoted as IP_{start}(*inverted path*), e.g. IP_{start}(item.headline.Japan.news)=7. We use these IDs to label XML nodes.

The inverted path tree is useful to retrieve nodes by tag names or path suffixes, and also durable for dynamic updates. For example, even when a new interval $(10.3.1, 10.3.5)$ for //NY/sports/item is inserted, the query range of the //sports/item $(9.9, 10.5)$ is not affected, i.e. as long as the modification to the inverted path tree is atomic, concurrent transactions do not face inconsistency while they are looking into the inverted path tree.

3.3 XerialMDI

XerialMDI (Xerial Multi-Dimensional Index) is a representation of XML nodes with the schema, (start, end, level, inverted path ID) where (start, end) denotes an interval of an XML node, and inverted path denotes a sequence of tag names from a leaf node to the root. All of these four attribute values are extensible composite-numbers. An interval of XerialMDI, (start, end) is ready to accept an arbitrary number of subintervals within it. Extension of the level value rarely occurs since the average depth of an XML document is low [8], and hence it is usually sufficient to use the 8-bits integer as the domain.

Every attribute element in XML is assigned the same interval and level value with the tag that it belongs to, so as to learn subtree range of the tag from the index of the attribute. For example, if the index of `<item id="1">` is (1.1, 1.3, 1, IP_{start}(item)), its attribute element is denoted as (1.1, 1.3, 1, IP_{start}(@id.item)).

3.4 Indexing Multidimensional Data

To index multidimensional data, it is general to use R-tree [14], which groups together nodes that are in close spatial proximity, however, this criterion of clustering tends to pack nodes apparently not related. This problem sometimes referred as the *curse of dimensionality*. Instead of this, we took more straightforward approach, making clusters of nodes that have same attribute values as possible, for example, same level values and same path suffixes. This demand can be fulfilled by bit-interleaving of coordinate values. The interleave function (**Figure 4**) gives z-orders which are positions in the z-curve (**Figure 5**) [22], which is a space-filling curve embedded in a multidimensional space. This linear ordering of XML nodes enables us to implement the multidimensional index on top of the B+-tree. In order to make the interleave function serve stable z-orders even when the lengths of the extensible composite numbers vary, we virtually pad each bitstring with an infinite sequence of 0s. In steps 2 and 3 of **Figure 4**, three 0s are inserted which do not actually exist in the coordinates of level and inverted path ID.

3.5 Range Query

Queries in XerialMDI comprise retrievals of nodes contained in hyper-rectangle regions in the multidimensional space. All points p in a given query box $Q(begin, end)$, where $begin$ and end are z-orders transformed from nodes of XerialMDI, satisfy the following property: $begin \leq_z p \leq_z end$ where $\leq_z$ represents the prece-

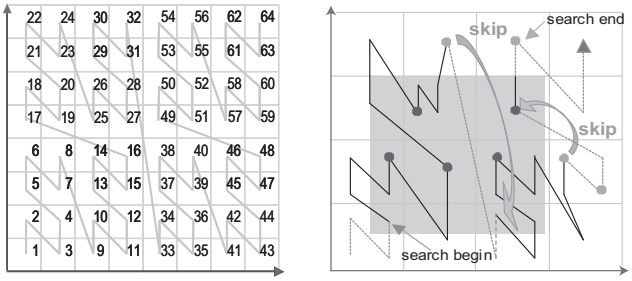

Figure 5: Left: z-curve and z-order. Right: range query algorithm

Lock held \ Lock requested	Scan	Insert	Delete	Read Contents	Modify Contents	Intention to Update (IU)
Scan	Yes	No	No	Yes	Yes	Yes
Insert	No	No	No	No	No	No
Delete	No	No	No	No	No	No
Read Contents	Yes	No	No	Yes	No	Yes
Modify Contents	Yes	No	No	No	No	No
Intention to Update (IU)	Yes	No	No	Yes	No	No

Table 1: Compatibility Matrix

dence on the z-order. Therefore, the process of hyper-rectangle range queries is to scan the linear ordering of leaf pages in the B+-tree. **Figure 5** shows the behavior of the range query. The zigzag arrow represents the order of points in leaf pages of the B+-tree. The query begins the search from the lower bound of the query box, then repeats the trace to a next leaf node of the B+-tree until the cursor exceeds the z-order of the upper bound. In the course of the search, it will find some points which are out of the query box. At this time, the *nextZValue* algorithm described in [23] efficiently computes the next z-order in the query box where the search restarts. It skips some nodes in the outside of the query box and saves disk I/O costs.

4 Locks for XML

4.1 HR Operations

In this paper, we assume that transactions may issue the operations for hyper-rectangular regions (HR operations) in **Table 1** over a given hyper-rectangular region R. These operations are processed by the range query and page writes. Table 1 also shows the compatibility between these operations. A sequence of corresponding operations in the table described as Yes is *commutable*, i.e. exchanging the execution order of the two operations does not affect the state of the databases. In Xerial, it is assumed locks for the regions designated by these HR operations are taken before executing corresponding range queries and updates.

The Modify Contents operations do not change the tree structure of XML but only the text contents, thus it is semantically compatible with Scan operations, which do not read any text content. On the other hand, Insert and Delete operations may modify tree structures by inserting or deleting nodes. Consequently, these operations are not compatible with any other operations. Insert or Delete operations are not used alone themselves, since they have to find their target intervals by scanning the tree structure. The common case of using Intention to Update (IU) operations is a query that reads a large region and may update a very small fraction of it. For such transactions, taking Insert or Modify Contents locks on such a large region unnecessarily blocks other read only transactions. Therefore, we made them acquire IU mode locks before any updates, then upgrade the portion of the lock to the writable mode.

4.2 Detection of HR lock Conflicts

Conflicts between distinct items, say A and B, are easy to detect. However, when the concept of the items is extended to hyper-rectangle regions, A and B are no longer distinct if they spatially intersect. The conflict detection of hyper-rectangle regions could be costly, and hence acceleration of this step is crucial. To handle this problem, we utilize a priority search tree [20] to detect interval intersections for each dimension. It can enumerate one-dimensional interval intersections in $O(\log n)$, where n is the number of the intervals. To extend this to hyper-rectangular intersections, we constructed multiple priority search trees for storing each edge (interval) of HR regions separately, and attached attached lock IDs to the inserted intervals. HR-lock intersection is computed by querying intersecting lock IDs to every priority search tree, and then merge the results. The computational complexity of this method remains $O(\log n)$.

4.3 Lock Acquisition Scheduling (LAS) Protocol

We developed lock management techniques of HR operations as LAS Protocol. It is based on the strong 2-phase locking, which demands that a transaction never releases its acquired locks until it commits, and when the transaction commits, it releases all acquired locks at once.

The waits-for graph [27] is a directed graph whose edges represent dependency of lock requests between transactions. We utilize the waits-for graph to detect deadlocks and also manage the order of lock acquisitions.

When a transaction tries to take a lock on some region, it must follow the procedure described below, and it must be done atomicly.

Lock Acquisition Procedure:

T_r : the lock-requesting transaction

1. Search the priority search trees (PSTs) for lock IDs that conflict with T_r in terms of the requesting lock region and lock mode. These lock IDs in conflict with T_r may contain both of currently awaiting and already granted locks. Then, insert the requested lock regions attached with a new lock ID into PSTs. In this phase, the lock is not granted yet.
2. If there are no conflicting lock IDs, grant the lock request to T_r immediately. Otherwise, in order to give the order of lock acquisition, to the waits-for graph, add out-edges from T_r to every transaction with conflict lock IDs, except there are already in-edges from these conflict transactions whose lock requests are not granted. In addition, if T_r is trying to upgrade its own granted locks overlapping with requested region, and if no other transaction has granted locks overlapping with it, instead of the out-edges, add in-edges from the incompatible transactions to T_r. (**Figure 6**)
3. By scanning the waits-for graph in a depth-first manner, detect all cycles caused by the lock request of T_r. In the course of search, calculate the cut set of each cycle, choose one transaction in the cut set at random (or by using some heuristics) as a victim, and mark it **"to be abort"**. Aborting the victim resolves the deadlock among transactions in the cycle. Furthermore, the victim may also appear in another cycle. Such a cycle is no longer effective and it is called *pseudo cycle*. These pseudo cycles are ignored during the search.
4. If the victim is the focusing transaction T_r, abort T_r. Otherwise, wake up the victim and send an abort order to it. When the victim is woken up, it must move to an abort phase.
5. Make T_r asleep until woken up by another transaction. After woken up, the requested lock of T_r is granted.

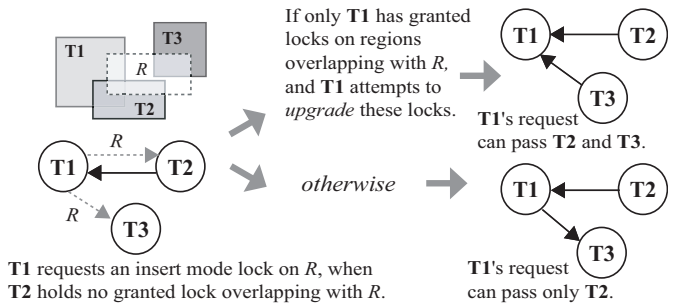

Figure 6: Avoiding trivial deadlocks

To put the above procedure briefly, when we want to lock a region, first, we have to search PSTs for conflicting lock requests and granted locks, then update the waits-for graph. If we find a deadlock, we resolve it by aborting some transaction as a victim. If the lock request is not granted immediately, the transaction has to wait until the lock becomes available.

The waits-for graph also maintains the information of responsibility that which transaction has to give its own lock region to another transaction requesting it. The strategy of updating the waits-for graph in Step 2 prevents deadlocks easily solvable, and gives priority to transactions requesting lock upgrades.

Next depicts the actual process of delegating locks:

Lock Release Procedure:

1. For every in-edge of the finished transaction, T_f, such that the source transaction has no other out-edges in the waits-for graph, push the transaction ID of the source transaction to list L.
2. Remove node T_f and all in/out-edges of T_f from the waits-for graph, and all acquired lock regions by T_f from the PSTs.
3. Wake up every transaction in the list L, then grant its awaiting locks.

This procedure must be also done atomically.

In LAS-protocol, there is also a rollback procedure. Likewise the traditional rollback methods as in [27], when some transaction is to be abort, it abandons current lock requests if they exists, and the scheduler inserts inverse operations which undo the non-committed operations. Note that these inverse operations do not require any additional lock since appropriate locks for them already have been taken.

4.4 Layered Locking

To prevent the phantom problem [27] in the B+-tree, page-level locks acquired by a transaction should be held until it commits in the strong 2PL manner. Thus, locks on the leaf pages must be held until the transaction commits. A transaction in XML, however, usually requires locks on large regions of the index structure. As a consequence, strong 2PL on page-level ends up blocking other transactions for a long time, and what is worse is even if we intend to lock separate HR regions, false conflicts between their corresponding page-level locks occur frequently. Our experiment in Section 5 shows that it causes a significant number of deadlocks.

A solution to these problems is to arrange locks in layers; locks for B+-tree pages and locks for HR regions. This method is called *layered locking*. The motivation to use layered locking is to exploit some semantic properties from sequences of page-level operations. The definition of lock compatibility matrix in **Table 1** gives such semantic information whether the given two operations affect the result of each other. Thus, in addition to the page-level

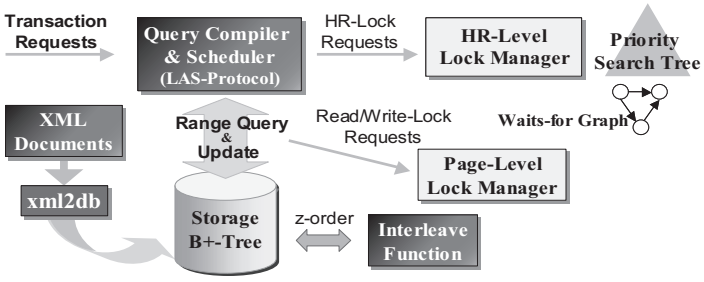

Figure 7: Xerial system overview

concurrency control (i.e. *lock-coupling* [27]), by integrating concurrency control of the HR locks, i.e. LAS-protocol scheduler, the transaction can release acquired page-level locks before it commits as a whole, as long as it holds the corresponding HR-level locks, i.e. HR-level locks substitute for page-level locks. Such early release of page-level locks is promising to increase transaction concurrency, and to avoid deadlocks between page-level locks. This integration of several kind of lock managers obeying 2PL in each level is known as *layered 2PL* and to be *serializable* [27]. Thus, as LAS protocol is designed based on 2PL, transactions in Xerial are assured to be serializable.

5 Experimental Evaluation

We evaluated the query performance of Xerial for several kinds of queries, e.g. ancestor, descendant, sibling, and path-suffix queries. We also tested the effect of the layered locking on transaction throughput and deadlock avoidance.

Implementation Xerial is implemented in C++. It consists of several components, e.g. a database generator (xml2db), query algorithms, LAS-protocol scheduler, HR-level lock manager etc. The overview of Xerial is illustrated in **Figure 7**. To construct B+-trees, we used the BerkeleyDB library [25], which is an open source database library and it supports page-level transaction management using the *lock-coupling* technique [21]. On top of this, we implemented our own hyper-rectangular(HR) lock manager of LAS-protocol which consists of priority search trees and a waits-for graph.

Machine Environment As a test vehicle, we used an Windows XP, Pentium III 1GHz (Dual Processor) machine with 2GB main memory and two 10,000 rpm SCSI HDD (32GB) which are used separately for databases and logs.

5.1 Query Performance

To see the query performance of Xerial, we prepared two competitors, start index and tag-start index. The start index sorts XML nodes in the order of start. It has the data structure (start $\Rightarrow$ end, level, tag name, *text content*) in B+-tree. The tag-start index, ((tag name, start) $\Rightarrow$ end, level, *text content*), sorts nodes first by tag names, then by start orders, which is devised in [5] to accelerate structural join queries.

As datasets, we used Shakespeare's plays in XML format, shakespeare.xml [9], and XML documents provided by XMark benchmark project, standard.xml [24], since they are frequently used as a dataset in the literature. **Table 3** shows their database sizes and construction times using Xerial and these indexes. Since Xerial needs the bit-interleaving to sort keys in B+-tree, it takes longer time to construct the databases.

//ACT//SPEECH				
	ACT (185)	SPEECH (31028)	Join (30951)	Total (sec.)
Xerial	0.015	0.593	0.203	**0.811**
Tag-Start Index	0.015	0.547	0.219	**0.781**
Start index	1.718		0.220	1.938

//SPEECH/SPEAKER			
SPEECH (31028)	SPEAKER (31081)	Join (31018)	Total (sec.)
0.578		-	**0.578**
0.437	0.5	0.25	1.281
1.688		0.25	1.938

Subtree (5031)	Sibling level = 4 (1605)	Sibling level = 5 (36552)	Ancestor level <= 7 (7)
0.07	**0.031**	**0.672**	**0.062**
0.07	4.531	4.875	0.719
0.06	0.78	1.406	**0.016**

(Numbers in parentheses represent the numbers of answer nodes of the queries.)

Table 2: Query performance in shakespeare.xml

	shakespeare.xml (7.5 MB: 179,690 nodes)		XMark standard.xml (111 MB: 2,048,193 nodes)	
	Time (sec.)	Size	Time (sec.)	Size
Xerial	22.3	17 MB	340.8	225 MB
Tag-Start	14.1	20 MB	210.5	270 MB
Start	11.3	12 MB	187.4	213 MB

Table 3: Database size and construction time

In the following experiments, we measured the average times of five hot-runs for individual operations, and ignored the output costs of reporting the query results. All of the indexes are implemented with B+-tree, and their page sizes are set to 1K.

Ancestor-Descendant Query First, we simply compared performance of structural-join [1], which prepares two node sets corresponding to ancestors and descendants by scanning the databases, then merge them and pick up node pairs having the ancestor-descendant relationship. The left table in **Table 2** shows the performance of structural join query. Since these three indexes use the same structural-join algorithm described in [5], the performance difference depends on how fast they can collect nodes that have the same tag name. Therefore, the tag-start index, which has clusters of tag names is the fastest. Nevertheless, Xerial performs as fast as the tag-start index, because the interleave function of Xerial also plays a role to group together nodes which have the same tag name. The start index is weak in processing this kind of query since it has to scan the whole index as information of tag names is hidden in its data pages.

Path Suffix Query The middle table in **Table 2** shows the performance of the path suffix query. Since Xerial has the index for path-suffix, it can save the join costs and improves the performance in comparison with the other indexes.

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 (**Table 2** and **Figure 8**). This is because these indexes except Xerial have difficulty to find nodes in the target level.

Sibling retrieval algorithm used in this experiments of the start index repeats searching the tree for a node in the target level with a depth-first traversal and skips of its descendants. The tag-start index performs this process in every cluster of tags. This descendant skip phase 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. This inefficiency of the start index becomes prominent in the level more than 4 as illustrated in the **Figure 8**. The tag-start index has a difficulty to find siblings in lower levels. This is because, the lower the target level, the more frequent the node skip is performed. Most of the nodes in a cluster of some tag name usually consist of sibling nodes in the same level, thus the node skip is likely to be ineffective, and its overuse brings about a bottleneck of the performance. To see this inefficiency, we also provided the result using sequential scan of the tag-start index, and it is faster than the start index and tag-start index for deep levels.

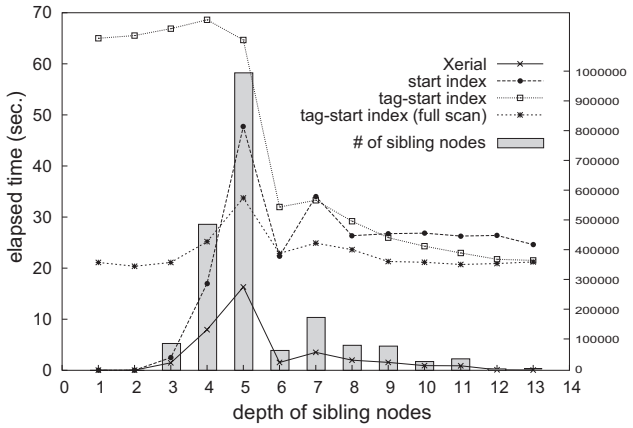

Figure 8: Sibling retrieval in standard.xml

Subtree Retrieval In shakespeare.xml, we could not see the significant performance difference of subtree retrieval (the rightmost table in **Table 2**). Therefore, we made the experiment by using the larger document, i.e. standard.xml (**Figure 9**). The start index is the most suitable data structure for subtree retrievals as nodes in a subtree are sequentially ordered, and shows the fastest result, although Xerial is fairly comparable 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. **Figure 9** shows the performance of the ancestor queries for various positions of base nodes of the query. The start index processes this query from the root node, and it can efficiently skip subtrees which are not the ancestor of the base node. Xerial is also efficient as it can eliminate the search spaces by using the end axis. However, the tag-start index breaks down the start order into multiple clusters grouped by tag names, in consequence, 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.

5.2 Update Performance

As for deletion of nodes and modification of text contents, they make no difference to the cost of page write in B+-tree, therefore, we only report the result for insertions for simplicity.

To the best of our knowledge, there has been proposed no algorithm which efficiently reallocates intervals labeled not using extensible composite-numbers, thus it is difficult to see the difference between Xerial and other indexes such as the tag-start and start index. However, consider the worst case scenario of interval maintenance, i.e. plenty of nodes are inserted into the front part

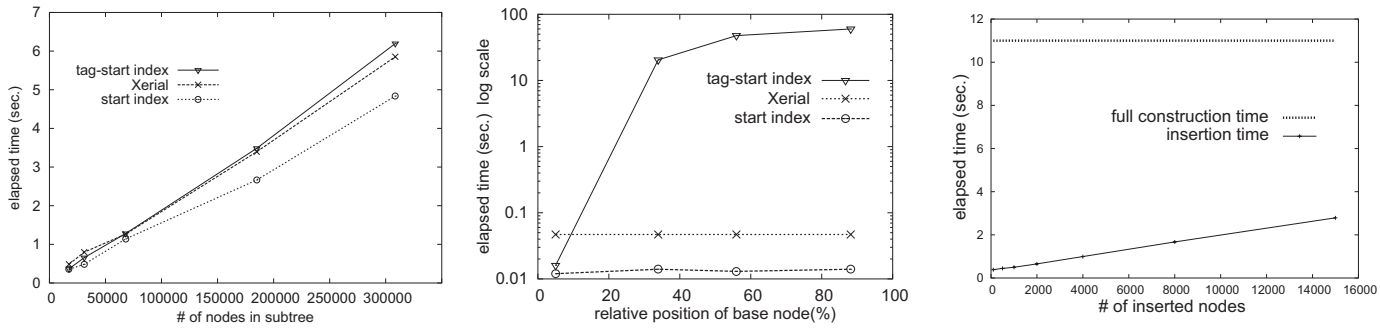

Figure 9: Performance of subtree retrieval (left), ancestor retrieval for level ≤ 12 (center), and insertions (right) in standard.xml.

Operation	Path Expression	Node	HR Lock	Exec. Time (sec.)	S1	S2
Insertion	/news/.../headline	58.8	3	0.122	10 %	20%
Deletion	/news/...//item[n]	1031.5	3	0.035	5 %	10%
Modify Content	/news/.../item[n]	330.3	1	0.094	5 %	10%
Subtree	/news/.../item[n]//*	331.8	2	0.076	30 %	30%
Sibling (shallow)	/news/*/*	18	2	0.004	20 %	10%
Sibling(deep)	/news/.../*	6.8	2	0.008	25 %	15%
Structural Join	/news/...//weather/item	476.8	2	0.215	5 %	5%

for 10,000 Transactions	S1		S2	
	# of Aborted Transactions	Time (sec.)	# of Aborted Transactions	Time (sec.)
Flat Model	1874	540.2	2667	750.9
Layered Model	**51**	**409.3**	**93**	**424.8**

Node: average # of nodes retrieved. **HR Lock**: # of HR locks acquired.
n : a random number used to select nth element.

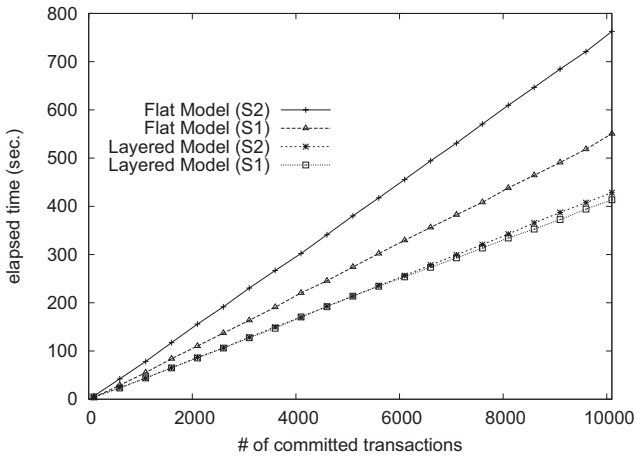

Figure 10: Mixture percentage of transactions (left) and performance of Xerial (right)

of intervals, because it results in reallocation of all subsequent intervals. Approximate estimation of the cost of such maintenance is given by the time constructing the index from scratch (**Table 3**). **Figure 9** shows insertion performance of Xerial when several nodes are inserted to the front part of the root interval. This result indicates that the performance of Xerial is just proportional to the number of inserted nodes and it never results in the worst case since there is no need to move intervals of Xerial. If a non-extensible interval does not have enough capacity, it may show a sudden increase of update time as much or more of the full construction time.

5.3 Transaction Throughput

To evaluate the efficiency of the LAS-protocol and its deadlock handling, we compared throughputs of two transaction models, a flat-transaction model and layered-transaction model using the LAS-protocol. The flat model uses only page-level lock management of strong 2PL with lock coupling.

As a dataset, we used news.xml. Its database size is 16MB, and it contains about 100,000 nodes. A simplified version of this XML is illustrated in **Figure 1**. It consists of the structure of news, continents (Asia, Europe, etc.), countries (USA tag contains about 50 nested states), news categories (headline, sports, weather, etc.), news contents (items), etc.

The upper left table in **Figure 10** shows the work sets of the transactions used in the experiments using news.xml. A '...' mark in each path expression denotes an existing path randomly chosen from news.xml. Theses queries are designed to be isolated in subtree level or shallow/deep level within the tree in order to increase their *logical* concurrency as possible. The set **S1** puts weight on the read-only transactions, and the set **S2** on the write transactions. To measure transaction cost, for each transaction, we computed the average execution time when performed alone. The maximum number of threads that concurrently accessed the database was set to 20. If some threads have finished their task, a new transaction request is passed to it immediately. When some transaction is aborted because of a deadlock, its transaction request is pushed to the end of the work queue, and resumed after a while.

In the course of experiments, we found out that the frequent query pattern of scan of nodes using range queries followed by update operations, i.e. read-and-modify step easily causes multiple deadlock states between several transactions, and ends up not being able to proceed concurrent transaction processing. Our solution to this is to use write locks instead of read locks during the searches for update-target nodes for the flat model, and IU locks instead of Scan locks for the layered model, however, the page-level operations under the IU locks were performed with read locks, since with the layered model, a transaction can release such read locks before it commits, in other words, there is no need to have both of page-read and page-write locks at the same time, thus page-level deadlock is seldom occurred in the layered model. The following experiments were conducted with this locking strategy.

The graph in **Figure 10** shows the passage of time until 10,000 of transactions committed, excluding aborted transactions. While the flat model suffered from a lot of deadlocks and costs of subsequent abortions, the layered model using LAS-protocol did not encounter such a frequent deadlock state. This is because the layered model succeeded in exploiting logical level concurrency, while the flat model experienced a lot of page-level lock conflicts. The reason that there is only a little performance difference between **S1** and **S2** in the layered model is an effect of *disk thrashing*; even though the concurrency of **S1** is higher than **S2**, intensive disk accesses make its performance slower.

6 Related Work

There have been a few studies on concurrency control in XML. Studies in [15, 17] use tree models of XML, and they do not seem to be capable of processing descendant queries. The proposal of locks for DataGuide [12] has the same weakness. Choi et al. [6] devised precision locks for XPath, however, it provides no experimental results.

Several XML databases not using the interval encoding often label nodes with the contiguous orders [10] or Dewey [3] orders etc. However, node insertions or deletions need heavy maintenance of these numbers. Although I. Tatarinov et al. [26] provides some efficient renumbering strategies, it is not agreeable in the transaction environment since these extra updates can be a bottleneck of concurrency.

T. Grust et al. [13] proposed an efficient structural join algorithm, which saves unnecessary disk scans by utilizing tree structure. Similar optimization by creating XML specific index structure is researched by H. Jiang [18]. Neither of them, however, mentions the integration of tag names or path suffixes, which we have proved to be a key factor to improve ancestor or wild-card query performance.

To index multidimensional data, zkd-BTree [22] and UB-tree [2] also utilize z-curve. It should be noted, however, both of them use a fixed bit-length value for each dimension, and consequently they cannot split the clusters into smaller ones than the minimal unit: a *grid* of the dimensions. Since insertions into a particular interval are frequent in XML, their approach causes a significant number of page overflows, as the consequence, it deteriorates the search performance.

Edith Cohen et al. provides a dynamic labeling scheme for descendant processing, and the bounds of the length of such labels when there is additional information, called *clue*, which is an estimation of the subtree and sibling size to be inserted in future [8]. They claimed such clues could be estimated from DTD or statistics of XML documents. Although using such information about the structure of XML documents is beyond the scope of this paper, such dynamic labeling schemes are indispensable to reduce the size of Xerial databases.

To process XPath queries, we decomposed them into several components. More general descriptions of decomposition and query optimization techniques are addressed in [4, 19]. However, query optimization for Xerial is particularly focusing on utilizing the inverted path tree and eliminating the search space of range queries.

7 Conclusions & Future Work

To efficiently process concurrent query and update operations, we have developed a transaction-enabled database system for XML, called Xerial. It provides efficient processing of ancestor, descendant, sibling, and path-suffix queries, extensibility to future node insertions, and capability to avoid deadlocks. Our experimental results show advantages and disadvantages of query processing due to the indexing methods of the interval labeling of XML. Other queries not targeted in this paper are references by using IDREF edges or inverted indexes for the text contents. Even though it can leverage traditional database and IR technologies, it is worth investigating to incorporate such additional index structures into Xerial. For concurrency control, we devised the layered lock management for XML and deadlock handling mechanism. Our comparative experiment using the flat and the layered locking model suggests a strong need to manage deadlocks in XML transactions.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE*, 2002.

[2] R. Bayer and V. Markl. The UB-tree: Performance of multidimensional range queries. Technical report, 1998.

[3] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management - Native XML and XML Embeded Database Systems*. Addison Wesley Professional, 2003.

[4] Z. Chen, H. Jagadish, L. V. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *Proc. of VLDB*, 2003.

[5] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, 2002.

[6] E. H. Choi and T. Kanai. XPath-based concurrency control for XML data. In *Proc. of DEWS 2003*.

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999.

[8] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proc. of PODS*, pages 271–281, 2002.

[9] R. Cover. The XML cover pages. available from http://xml.coverpages.org/xml.html.

[10] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, 1999.

[11] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, 1997.

[12] T. Grabs, K. Bohm, and H.-J. Schek. XMLTM: Efficient transaction management for XML docuements. In *Proc. of CIKM*, 2002.

[13] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *Proc. of VLDB 2003*.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1984.

[15] S. Helmer, C.-C. Kanne, and G. Moerkotte. Lock-based protocols for cooperation on XML documents. Technical report, the University of Manheim, 2003.

[16] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native xml database, 2002.

[17] K.-F. Jea, S.-Y. Chen, and S.-H. Wang. Concurrency control in XML document databases: XPath locking protocol. In *Proc. of ICPADS 2002*. IEEE 2002.

[18] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proc. of ICDE 2003*.

[19] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of VLDB*, 2001.

[20] E. M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2), May 1985.

[21] Y. Mond and Y. Raz. Concurenty control in B+-Trees using preparatory oeprations. In *Proc. of VLDB*, 1985.

[22] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. of PODS*, 1984.

[23] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a database system kernel. In *Proc. of VLDB*, 2000.

[24] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. manolesch, and R. Busse. XMark: A benchmark for XML data management. In *Proc. of VLDB*, 2002.

[25] Sleepycat Software. BerkeleyDB.

[26] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, 2002.

[27] G. Weikum and G. Vossen. *Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[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 *Proc. of SIGMOD*, 2001.