ON CONCURRENCY AND UPDATABILITY OF XML DATABASES
XML データベースの並列性と更新性能についての研究

by

Taro L. Saito
斉藤 太郎

A Master Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Master of
Information Science and Technology
in Computer Science
February 4, 2004

Thesis Supervisor: Shinichi Morishita　森下 真一
Professor of Computer Science

Abstract

XML is a tree-structured data. Transactional databases for XML strongly demand the capability of efficient query processing because any update operation has to find the update target before any modification. One of the difficult path-expression queries of XML is to search for nodes by ancestor-descendant relationships. To handle this type of tree navigations, an index using interval containment has become popular today. However, node insersions exhaust the intervals too soon. Furthermore, we cannot overlook the cost of dynamic interval expansion and shrinkage since it may cause cascading updates and subsequently lose the concurrency. To solve these problems, we developed a novel multi-dimensional XML index on B+-tree using *extensible composite-numbers*. With our index, we can avoid the dynamic interval maintenance, and can quickly answer fundamental queries, such as not only ancestor-descendant queries, but also retrivals of subtrees, child and ancestor nodes. We also present a layered locking strategy for XML transaction. The first layer is a traditional page-level locking. The second one uses locks for hyper-rectangular regions and assumes intersections of rectangles as lock conflicts. In this paper, we prove the impact on transaction concurrency of our index and locking methods by several comaparative experiments.

論文要旨

XML は木構造を持ったデータである。トランザクション用の XML データベースでは、効率よく問い合わせ操作を実行できることが非常に重要である。なぜなら、あらゆる更新操作は、まず始めに、更新の対象となるノードを探さなくてはならないからである。経路表現の中でも難しい部類に入るのが、祖先-子孫関係による問い合わせである。近年では、このタイプの経路表現の探索に木構造を区間の包含関係を用いて表すのが一般的となってきた。しかし、区間はノードの挿入によってすぐに埋まってしまうだけでなく、動的に区間を拡張・縮小するようなコストは、連鎖的な更新を引き起こして並列性 (concurrency) を落とすので、過小評価はできない。このような問題を解決するため、我々は、拡張可能な合成数 (*extensible composite number*) を使った B+木上での XML の多次元索引を開発した。この索引を使うと、動的に区間を調節する必要がなくなり、祖先- 子孫関係だけでなく、部分木、子ノード、親ノードという基本的な問い合わせに対しても高速に答えられるようになる。また、本研究では XML トランザクションにおける階層化ロッキングの手法も提案する。第一層は従来のページレベルロックを使ったものであり、第二層は超長方形の領域のロックを用い、その交差関係をロックの衝突として扱うものである。この論文では、我々の索引とロック手法が、トランザクションの並列性向上に与える影響を種々の比較実験によって示している。

Acknowledgements

I have longed for completing this master thesis, the fruit of my continuous work since I was an undergraduate student. I would like to note here that I cannot complete this work without tremendous helps from my advisor, colleagues, and my family.

First of all, I want to say that this is my honor to have studied under Professor Shinichi Morishita, who is my adviser. Although he had been a very busy person, he spent a lot of hours and days for discussons with me. These oppotunities were very valuable to me to find hidden problems and to make clear my ideas. I also thank him for introducing me into such a wonderful world of academic research. He gave me a first motivation of this paper: dealing transacions in XML. At first, I could not find the importance of the problem. However, as I had continued to survey about XML databases, I found quite a little work keeping in mind *updates*; the main objective of this paper. This fact have been stimulating me greatly to study the theme. Also, it was the time that I knew the pleasure of tackling unsolved problems.

This work was accompanied with a lot of programming task in C++. While writing more than 50,000 lines of codes, advice from Masahiro Kasahara, who is one of my colleagus and also a genius hacker, helped me a lot to implement faster codes in a short time with less effort.

I am greatful to my colleages in Morishita Laboratory. They always keep in touch with bland-new programming technologies. The days I spend with these stimulative members have greately improved my programming skill. In particular, I thank them for letting me know the recent progress of XP (eXtreme Programming), which is a new style of programming and testing codes. Without the knowledge of XP, I would have faced so many harder and more intractable bugs since the codes of the XML database have become larger than I had expected. I am fortunate in that I could learn in advance the difficulties of maintaining such large codes and the strategies to cope with them.

Finally, I thank my wife, Naoko Bando for supporring me for everything and for sharing laborious daily housework. I also would like to note about my one-year old son, Yui. Your innocent smile always makes me happy and feel relax even when I am in the hardness of the research. I love you two.

Taro L. Saito,
January 5th, 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transactions in XML have to begin with path-expression queries to locate nodes to update. XPath [10], which is the *de facto* standard for navigating XML, contains tree traversal expressions such as child(/), descendant(//), ancestor, and sibling axis. To increase the transaction throughput, it is indispensable to design data structure that can efficiently process both of these tree-traversal queries and update operations, i.e. insertions, deletions and modifications of text contents.

Indexing XML document for database usage has been studied enthusiastically since around 1997. In recent years, the use of interval containment has become prominent for describing ancestor-descendant relationships [1, 25, 40]. **Figure 1.1** illustrates an example of the interval indexing. All interval nodes except the ref (*root* node), are labeled with subintervals of their parent intervals, and sibling subintervals are aligned to be disjoint. It is easy to see that the interval containment is equivalent to the ancestor-descendant relationship. The start numbers of the intervals maintain the node orders in the original XML document.

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 [22] does not solve this problem since they are also finite in computers. Thus, interval labels which remain *stable* after node insertions are required. In this paper, we provide a solution to the problem by using *extensible composite-number*, which is a variable length number and gives interval representation resistant to insertions. This idea is quite simple but, in the literature, such approach has not been examined seriously.

```
<ref>
  <book>
    <title> Data on the Web </title>
    <author> Serge Abiteboul </author>
  </book>
  <paper>
    <title>  XMill: an efficient compressor for XML data </title>
    <authors>
       <author> Hartmut Lifke </author>
       <author> Dan Suciu </author>
    </authors>
  </paper>
  ...
</ref>
```

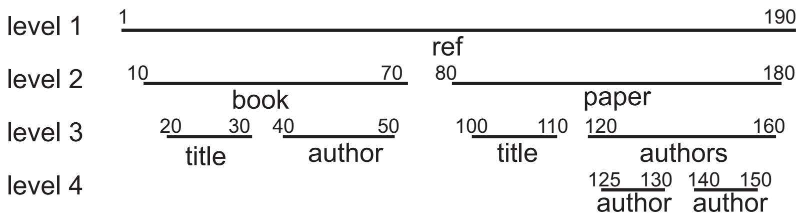
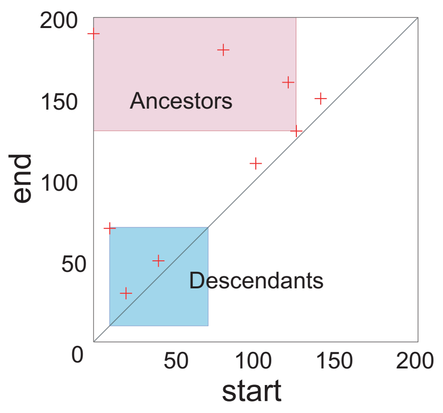


Figure 1.1: An XML document and its interval containment representation

Concurrency control is a system that manages simultaneous running transactions to guarantee the same effect as that of executing them independently in some serial order. Since XML has a tree structure, one might consider to use existing lock protocols for tree-structured data, such as the tree protocol [14] and the granular-locking protocol [18] which traverse trees in top-down manner. However, they are not applicable for ancestor traversals, and moreover the process of descendant queries is likely to scan a large part of the entire tree, as a consequence it ends up losing concurrency. Therefore, to identify target nodes of queries, another approach not using tree traversal is required.

Our solution to this issue is to pack target nodes of queries in hyper-rectangular regions, by describing every XML node as a point in the multidimensional space with four axes: start and end of an interval, level from the root node, and inverted path that denotes the sequence of tag names from a leaf node to the root. This multidimensional space effectively packs ancestor and descendant nodes in rectangular regions in the two-dimensional plain (**Figure 1.2**). In addition, the plain slice of a particular level value identifies sibling nodes instantly. The axis of inverted path is useful to select nodes having a common path suffix. Combination of these axes provides adaptive covers of target nodes of XPath queries, and narrows the search space of these queries.

We also use these hyper-rectangular regions as lock granules for XML. It demands a specialized lock manager that tests intersections of hyper-rectangles and also prevents



start	end	level	inverted path
1	190	1	ref
10	70	2	book.ref
20	30	3	title.book.ref
40	50	3	author.book.ref
80	180	2	paper.ref
100	110	3	title.paper.ref
120	160	3	authors.paper.ref
125	130	4	author.authors.paper.ref
140	150	4	author.authors.paper.ref

Figure 1.2: A projection of intervals in Figure 1.1 onto a 2D plain. The colored rectangles show ancestor and descendant query regions.

live-locks and dead-locks. To satisfy these requirements, we organized a lock manager based on a new lock-scheduling algorithm, called LAS (Lock Acquisition Scheduling)-protocol.

Another challenging task is to create an index of XML nodes which is efficient for concurrency control. Constructing multiple secondary indexes for all of the dimensions not only consumes a lot of disk space, but also imposes a heavy load on every update transaction to synchronize all of the indexes. Another approach is the use of R-tree [19], which groups together nodes that are in close spatial proximity, however, this criterion of clustering tends to pack nodes apparently not related. Considering page-level locks, it is problematic since even if there is a logically tight hyper-rectangular region of a query, it ends up blocking a lot of far-related nodes included in the accessed disk pages.

A more straightforward way of indexing is to make 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 that aligns XML nodes along the z-curve [30], 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. It remarkably saves the storage space in comparison with constructing multiple secondary indexes.

With regard to lock management, it is usual that a transaction in XML requests a lot of page-level read locks to locate node positions to update, then it upgrades some of these locks to writable modes. This pattern of lock conversion: *large-read and partial-write* is likely to cause deadlocks. In fact, we experienced frequent deadlocks in the implementation using flat-model transactions. We observe that to organize page-level locks and hyper-rectangular locks into layers is a key factor to make XML transactions cope with deadlocks.

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 hyper-rectangular regions. It significantly reduces the number of locks acquired by transactions.
- By utilizing the layered locking [38], 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, and increase the transaction concurrency.

By using these techniques, we developed a transaction enabled XML database system called **Xerial**. 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 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 the embedded XML databases [20], and they implemented node and edge level locking for XML [21]. T. Grabs et al. proposed a lock manager on the DataGuide [16] in [17]. Extension of the granular-locking for XML is proposed in [23]. However, they all avoid processing of descendant queries. Moreover, the experimental results of the node and edge level locking in [21] indicates its negative aspect that suffers high percentage of deadlocks. Precision locking for XML [9] needs a lot of duplication 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.

Research scope In this paper, we do not use any *a priori* knowledge about the structure constraints of XML, such as DTD. Thus, we do not mention index compression and query optimization using such information. In addition, for simplicity, we will deal with transactions on a single XML document, i.e. a single rooted tree. Extension to multi-document databases is trivial since the interval index can also be used to represent a forest of multiple XML documents.

Organization of the rest of the paper is as follows: in Chapter 2, we show an example motivates the need for the proper indexing of XML. Chapter 3 introduces our novel multidimensional index called XerialMDI and its components. Then, we describe concurrency-control and lock management of Xerial and some keys to exploit transaction performance in Chapter 4. Chapter 5 describes algorithms that handle several types of queries including structural joins. Chapter 6 shows implementation of Xerial. In Chapter 7, we provide results of experimental evaluation. Finally, we report related work in Chapter 8 and conclude in Chapter 9.

Chapter 2

Motivating Example

First, we show that sibling retrieval is not efficiently processed in the index just sorted by start order of intervals (we call it start index). For indexes using interval labeling, capability to collect nodes in the same level is quite important to find blank spaces to insert new intervals, and also it is essential to compute parent-child join since interval containment cannot detect this relationship without using level values. **Figure 2.1** is an illustration of the process of sibling retrieval. While the start index can efficiently skip descendant nodes, it does not reduce the number of access to the disk pages. This is because the start index is likely to distribute sibling nodes into separate disk pages. This inefficiency is solved by using a linked-list structure of sibling nodes, however, to maintain consistency of the list structure, we must introduce locks for sibling edges [21]. Using fine grained locks can be a bottle neck of the transaction throughput. Although Xerial does not uses the linked-list structure, our experimental results show that Xerial is considerably faster than start index for sibling retrievals.

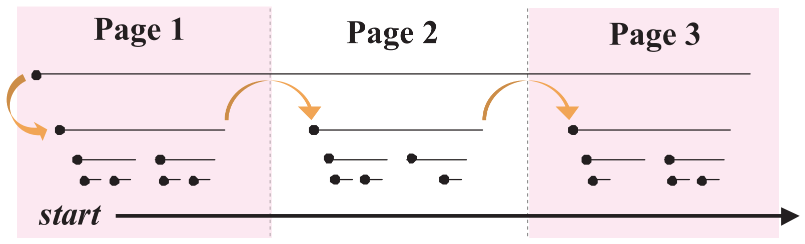


Figure 2.1: Sibling query processing in start index reads a lot of disk pages.

Chapter 3

Multidimensional XML Index

Although traditional use of intervals for labeling XML nodes was static and was not ready for updates [40], the dynamic setting that reserves space of intervals for future insertions has recently attracted attention [8, 22, 25], however, this strategy cannot endure node insertions at a close site. To overcome this vulnerability, we use the *extensible composite number* in the following section for labeling intervals.

3.1 Extensible Composite Number

An *extensible composite-number* $C = c_1.c_2.\ldots.c_n$ $(n > 0, c_i \in D)$ is a dot-separated list of non-negative integers that are taken from some domain D of non-negative integers, for instance, the set of short unsigned integers $\{0, 1, ..., 255\}$. The order of two extensible composite-numbers is same as lexicographical order when assuming each c_i as an alphabet. For convenience of the bit-interleaving , all composite-number are virtually padded by an infinite sequence of 0s. For example, when $C_1 = 18.5$, $C_2 = 18.5.10$, $C_3 = 1.5$, 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.

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, DataGuide [16] uses a path tree which aggregates common paths in the XML document (**Figure 1.1**). However, it lacks the capability to handle descendant-axis(//) queries. This class of queries includes path-suffix queries such as //A, //A/B, etc.

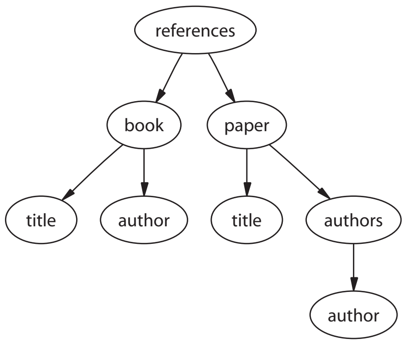


Figure 3.1: A path tree constructed from the XML in Figure 1.1

To improve accessibility to path suffixes, we devised a new data structure, the *inverted path tree*. Similar to the interval representation of XML, the inverted path tree is also an interval tree using extensible composite numbers (**Figure 3.2**). In the inverted path tree, every sequence of tag names from some interval to the root represents a path suffix in the XML document. For example, the interval of author $(1.1, 4.5)$ indicates the range of //author, and book $(3.5, 4.3)$ is the range of //book/author. We treat a start order of a leaf interval as an ID of the corresponding inverted path denoted as $IP_{start}(\textit{inverted path})$, e.g. IP_{start}(author.book.ref.)=4. We use these IDs to label XML nodes.

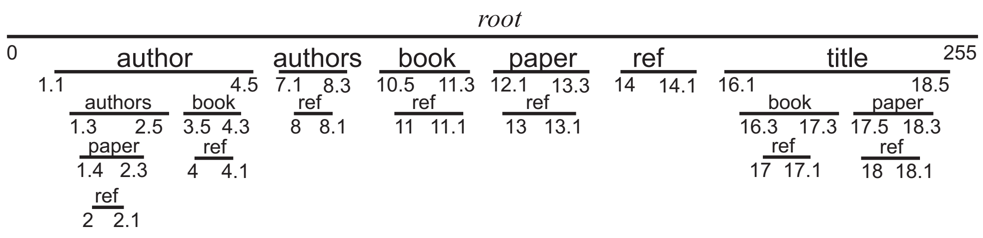


Figure 3.2: An inverted path tree of Figure 1.1

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 $(1.3.1, 1.3.5)$ for //article/authors/author is inserted, the query range of the //authors/author $(1.3, 2.5)$ 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). All of these 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 [11], and hence it is usually sufficient to use the 8-bits integer as the domain. Nevertheless, it is possible to make it extensible for values more than 255 by encoding them with some function which maps values in a semi-infinite range into ones within a limited range, e.g. $\tan^{-1}(x)$. For brevity, we omit the details.

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 `<book id="1">` is (1.1, 1.3, 1, IP_{start}(book)), its attribute element is denoted as (1.1, 1.3, 1, IP_{start}(@id.book)).

3.4 Embedding Multidimensional Data to One Dimensional Space

XerialMDI utilizes one-dimensional data structure, B+-tree, to represent the multidimensional data by filling the z-curve (**Figure 3.3**) into the multidimensional space. This idea is also utilized in the zkd-BTree [30] and the UB-tree [3]. Their approach is to split each dimension of the multidimensional space, and create finite number of disjoint subcubes or clusters of them that are contiguous along with the z-curve. Each point in the multidimensional space belongs to one of the subspaces.

It should be noted that the zkd-BTree and the UB-tree 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. When the number of points included in the minimal cluster exceeds a pre-defined threshold, they must be moved to overflowed pages. These overflowed pages are usually placed out of the index structure, and it significantly reduces the search performance. This problem is very serious, since insertions into a particular interval are frequent in XML. Therefore, instead of splitting the dimensions, we take another approach that assumes the z-curve fills the space densely like a fractal, and uses it just to assign orders to the points in the multidimensional universe.

To align points along with the z-curve, we can use the z-orders generated by the

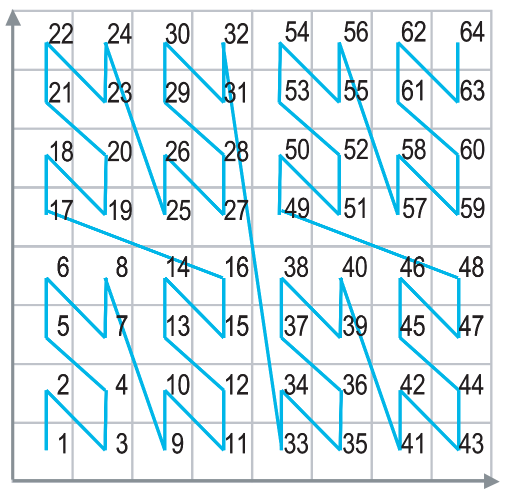


Figure 3.3: Z-curve and z-order

interleave function (**Figure 3.4**). It receives coordinate values of a point as input. From their bit-string representations, by retrieving single bits from heads of coordinate values in a round-robin manner, it computes the z-order, which is a position on the z-curve. A step in the z-order consists of all bits from the input bit-strings, which have the same bit ordinal, i.e. step 0 contains the most significant bits from the all coordinate values. As we noted before, an extensible composite number is virtually padded with an infinite sequence of 0s. This is because to make the interleave function serve stable z-orders even when the lengths of the extensible composite numbers vary. In steps 2 and 3 of **Figure 3.4**, three 0s are inserted which do not actually exist in the coordinates of level and inverted path ID.

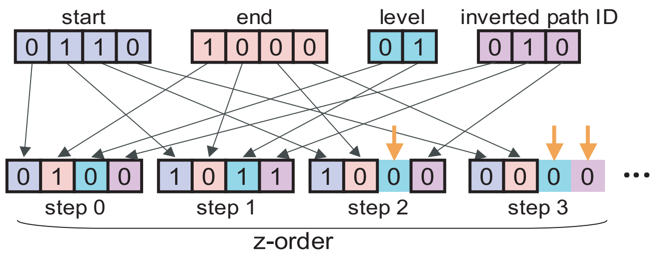


Figure 3.4: Behavior of the interleave function.

While the UB-tree generates integer values of the z-order and uses them as keys in a B+-tree, XerialMDI, however, does not instantiate these values. It is because the integer representation of z-orders wastes disk storages since the z-orders must have redundant 0s inserted for all of currently non-existing steps in order to make room for future extensions.

The data structure of XerialMDI is a sequence of all coordinate values accompanied with information of extension lengths of the coordinates. This information is replaced with shorter representation, *patten ID*, for storage efficiency (**Figure 3.5**). It gives a position where the bit-string of each coordinate begins. A B+-tree stores this structures as keys. To align these data in the B+-tree in the z-order, we replace the key comparison function of the B+-tree with the one that extracts the bits likewise in the order of the Figure 3.4 and compares them. **Figure 3.6** is the pseudo code of the comparison function of XerialMDI. The `zorderValue` function returns a bit value of a given bit position in the z-order value.

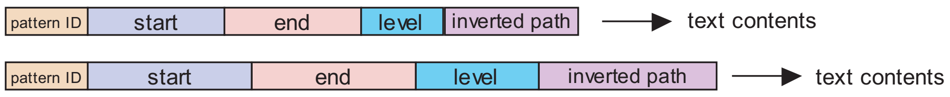


Figure 3.5: Data structure of XerialMDI. The second index is an example that extends the first one for each dimension.

Note that except the capability for future extensions, the performance difference due to the choice of transformation is subtle since the range query algorithm requires both type of conversion: from z-order into the coordinate representation and its vice versa.

3.5 Range Query

Queries in XerialMDI comprises 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 precedence on the z-order. Therefore, the process of hyper-rectangle range queries is to search leaf pages of the B+-tree linearly. **Figure 3.7** and **Figure 3.8** show the behavior of the range query. It 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 [31] efficiently

```
int XerialMDI::zorderValue(int zorderIndex)
{
  int dim = zorderIndex mod 4;
  int step = zorderIndex / 4;
  if(the step is larger than the length of
     the coordinate value of the dim)
      return 0;  // return virtually padded 0
  else
  {
     int base = (a bit position of the coordinate value in
                 dimension dim appears);
     return rawdata[base + step];  // return the corresponding bit
  }
}


// the compare function returns:
//   -1 : if a < b in the z-order,
//    0 : a = b,
//    1 : a > b.
int compare(XerialMDI a, XerialMDI b)
{
  int zorderLength = max(a.length(), b.length());
  for(int i=0; i<zorderLength; i++)
  {
     int diff = compare a.zorderValue(i) - b.zorderValue(i);
     if(diff != 0)
        return diff;
  }
  return 0;
}
```

Figure 3.6: A pseudo code of XerialMDI comparison function

computes the next z-order which returns into the query box again. It skips some nodes in the outside of the query box and saves disk I/O costs.

```
RangeQuery(QueryBox qb(XerialMDI lowerBound, XerialMDI uppderBound))
{
  z-order start = interleave(lowerBound);
  z-order end = interleave(uppderBound);
  z-order cursor = start;
  while()
  {
    if(cursor $>_z$ end) break; // out of the query box

    XerialMDI currentPoint = 4 dimensional coordinates the cursor points;
    if(the currentPoint is within the QueryBox qb)
    {
        output the currentPoint;
        move the cursor to the next leaf node of the B-tree.
    }
    else
        cursor = skipPoints(current, qb);
  }
}
```

Figure 3.7: Range Query Algorithm

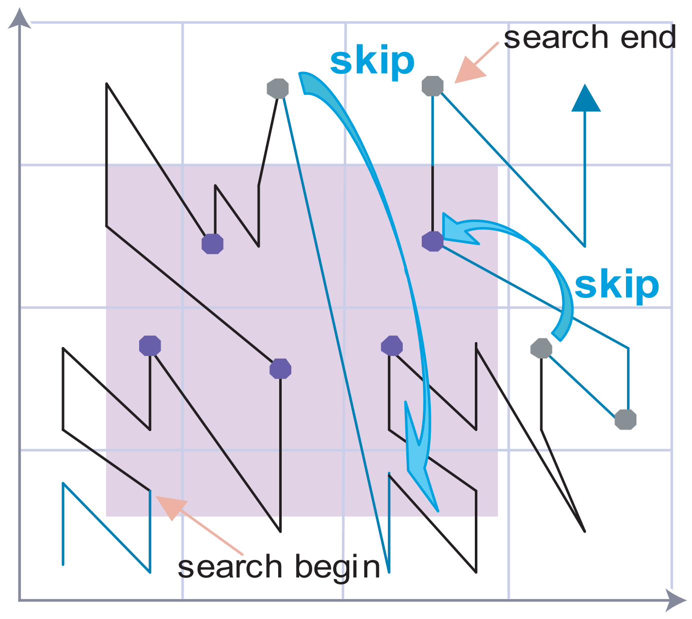


Figure 3.8: Illustration of behavior of the range query algorithm. The zigzag arrow represents the order of points in leaf pages of the B+-tree.

Chapter 4

Locks for XML

This section describes lock management for XML. Unlike the traditional locks for objects, Xerial uses locks for hyper-rectangle regions (HR locks).

4.1 HR Operations

First, we define fundamental operations for hyper-rectangular regions (HR operations). These operations are processed by the range query and page writes. In this paper, we assume that transactions may issue the following operations over a given hyper-rectangular region R:

- Scan retrieves nodes contained in R,
- Insert adds some nodes into R,
- Delete eliminates some nodes from R,
- Read Contents retrieves all nodes overlapping with R and it reads text contents associated with them,
- Modify Contents is similar to Read Contents except that it modifies text contents of some or all of found nodes.
- Intension to Update (IU) is similar to Read Contents except that it may perform subsequent Insert, Delete and Modify Contents operations in some portion of R.[1]

Table 4.1 shows the compatibility matrix of these operations. A sequence of corresponding operations in the table described as Yes is *commutable*, i.e. changing the

[1] To be precise, operations such as just increase or decrease text values of XML nodes may become necessary. However, to implement them is trivial, thus, in this paper, we omit these subtle operations for briefness.

		Lock requested					
		Scan	Insert	Delete	Read Contents	Modify Contents	Intention to Update (IU)
Lock held	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 4.1: Compatibility matrix of HR-operations

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 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 special case of using 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 large region unnecessarily blocks other read only transactions. In addition, consider the case that two transactions acquire Scan locks on the same region and both request to upgrade these locks to Modify Contents: it is a common pattern that causes a deadlock cycle. The solution to this problems is to have such transactions to acquire IU mode locks before any updates and then upgrade the portion of the lock to the writable mode, so that one of the write transactions has to wait until the other one releases the IU lock, but the other read-only transactions can proceed if there is no write-mode locks within the region locked with IU 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 in multidimensional space, 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.

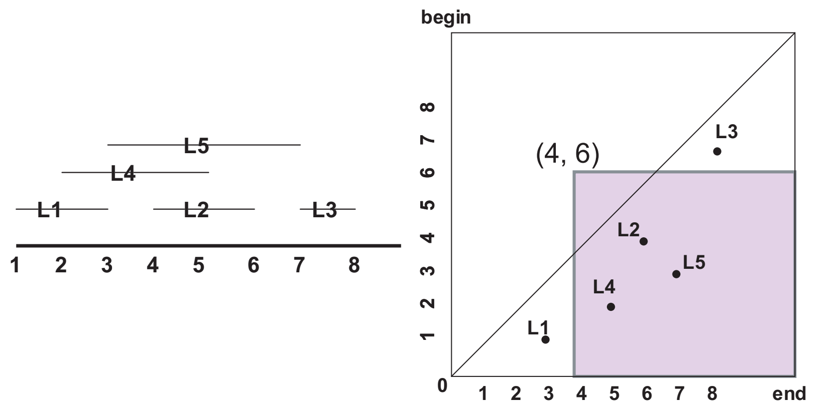


Figure 4.1: Enumerating intersecting intervals from a map on a two-dimensional plain

To handle this problem, we first explain the detection of one-dimensional interval intersections. If two intervals $I_1 = [a_1, a_2]$ and $I_2 = [b_1, b_2]$ intersect, one of the following inequalities is satisfied: $a_1 \leq b_1 \leq a_2 \quad or \quad b_1 \leq a_1 \leq b_2$. This relation becomes clear by seeing each interval as a point on the two-dimensional plain (**Figure 4.1**). For example, to enumerate all intervals intersect with $L2 = [4, 6]$, it is enough to pick up points contained in a rectangle region $[4, N] \times [-\infty, 6]$ where N denotes the maximum value of the x-coordinate. To answer this type of search quickly, a priority search tree [26] is useful (**Figure 4.2**). The priority search tree is a dynamic data structure that stores two dimensional points and facilitates semi-infinite range queries of the form $[x_1, x_2] \times [-\infty, y_1]$ in $O(\log n)$, where n is the number of the nodes.

To detect intersections of four dimensional hyper-rectangles, we construct multiple priority search trees for all of the dimensions, and attach lock IDs to the inserted intervals. Hyper-rectangular intersection is computed by querying intersecting lock IDs to every priority search tree, and then merge the results (**Figure 4.3**). The computational complexity of this method remains $O(\log n)$. A similar method is also described in [29].

4.3 LAS Protocol

We developed lock management techniques of HR operations as LAS Protocol. LAS is an acronym of Lock-Acquisition Scheduling. It is based on the strong 2-phase locking (Strong2PL), 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. Note that it is sufficient to prevent *phantom problems* and ensure serializability of

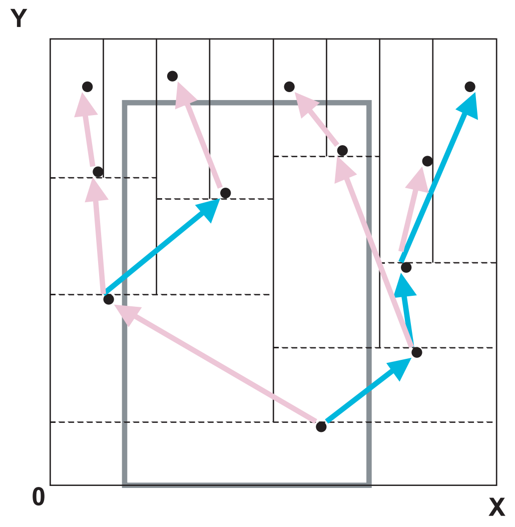


Figure 4.2: A priority search tree

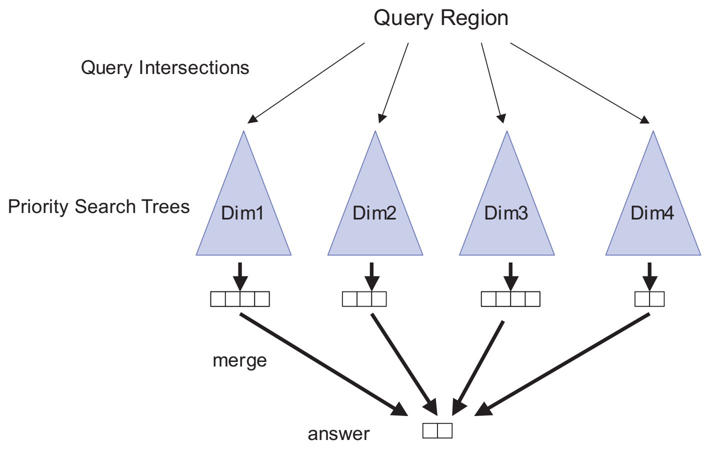


Figure 4.3: Detecting multidimensional hyper-rectangle interesections by using multiple priority search trees

transactions [38].

The waits-for graph [38] 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 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 request is 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 4.4**)
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.

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.

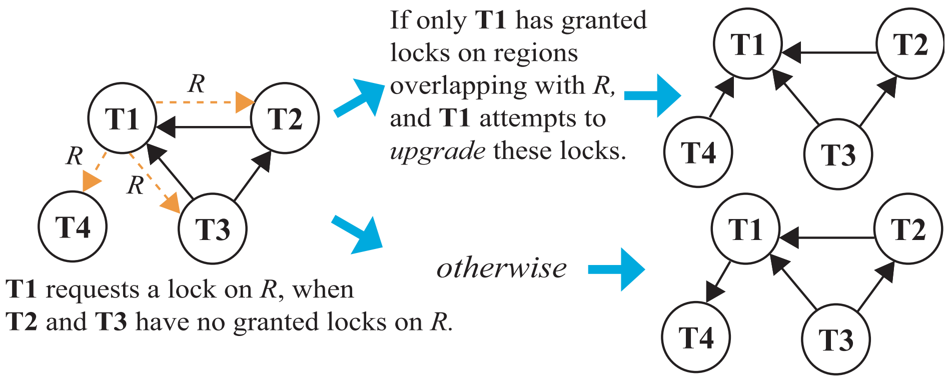

If only **T1** has granted locks on regions overlapping with *R*, and **T1** attempts to *upgrade* these locks.
otherwise
T1 requests a lock on *R*, when **T2** and **T3** have no granted locks on *R*.
T1
T2
T3
T4
R
R
R

Figure 4.4: Avoiding trivial deadlocks and handling lock upgrades. Transactions that $T4$ is awaiting are omitted for simplicity.

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 their awaiting locks.

This procedure must be also done atomically. Finally, we describe a procedure for rolling back transactions which run under Strong 2PL. Theoretical validity of such process is described in [38].

Rollback Procedure When Transaction Aborts

1. A transaction to be abort abandons current lock requests if they exists, then add *inverse operations* of already finished operations (*forward operations*) to the schedule of the transaction in order to roll back the effect of them. Note that *inverse operation* does not require any additional lock for the inverse operations since appropriate locks for them already have been taken.
2. After all of the *inverse operations* finish, call the Lock Release Procedure and commit the transaction, but report it as an abortion.

Let us reconsider about the step 2 of the Lock Acquisition Procedure. This step makes sure that no two transactions acquire incompatible locks on same regions R at the same time. To prove this, consider a time sequence of three transactions is $T_1 < T_2 < T_3$; namely, T_1 acquires a read lock on R, then T_2 and T_3 request write locks on R in this order. When T_1 commits, the situation that T_2 and T_3 acquire write locks on R simultaneously is depicted in the left graph of **Figure 4.5**. This graph, however, never occurs under LAS protocol since by the line 2 of the Lock Acquisition Procedure, T_2 must have made a lock request on R before T_3. Therefore, correct waits-for graph becomes as the right one in Figure 4.5. In this graph, when node T_1 is removed, T_2 acquires the lock on R. Since there is an edge from T_3 to T_2, T_3 has to wait until T_2 finishes. It is easy to see that we can generalize this observation for more than three transactions.

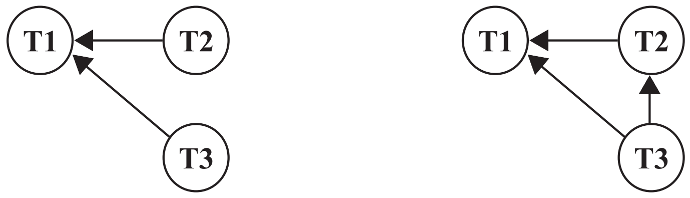


Figure 4.5: Waits-for graph of three transactions in the example. These edges represent the request to the region R. The left graph is invalid under LAS protocol. The right one exactly obeys LAS protocol.

4.4 Layered Locking

To prevent phantom problems in the B+-tree, page locks acquired by a transaction should be hold until it commits in the strong 2PL manner. Its exceptions are locks for inner pages of the B+-tree used for traversing the search structure, since the *lock-coupling* technique allows the transaction to release these locks before it commits [28, 38]. Nevertheless, 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 upgrade of these read locks to write ones usually causes deadlocks.

A solution to this dilemma is to arrange locks in layers; locks for B+-tree pages and locks for higher-level operations, such as the operations of Scan and Insert, etc. The motivation to use locks for higher-level operations is to exploit some semantic properties from sequences of page-level operations in order to improve transaction performance. The definition of lock compatibility matrix in **Table 4.1** gives such semantic information whether given two operations affect the result of each other. Thus, in addition to the page-level concurrency control, by integrating concurrency control of the HR-operation level, 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 increases transaction concurrency, and to avoid deadlocks between page-level locks. This integration of several kind of lock managers obeying 2PL is known as layered 2PL and to be *serializable* [38].

For a long time, the notion of layered locking have been used implicitly as record-level locking in commercial database systems, however, its theoretical principles have not been made explicit before the work of G. Weikum and C. Schek [38]. They organized these ideas and coined layered 2PL.

To illustrate the layered 2PL, consider an example from (**Figure 4.6**) with three layers consisting of page(L_0), HR-operation(L_1), and query(L_2) levels. The page-level and HR-operation level have their own lock manager, but the query level has no concurrency control. The key principle of layered 2PL is that locks at a certain level, say, L_i are on behalf of that of corresponding subtransactions in L_{i+1}, i.e. even with strong 2PL, a transaction can release page-level locks acquired in the duration of a subtransactions as their representative locks in the higher-level play the same role.

In **Figure 4.6**, subtransactions that define the scope of locks are highlighted by the colored triangles. The dashed lines indicate lock waits, and the arrows show duration of locks. **Figure 4.6** is an example that each of the transaction T_1 and T_2 are performing two independent queries, i.e. T_1 consists of two subtransaction in L_1 level, which are under control of LAS-protocol, and three L_0 subtransactions under lock-coupling with strong 2PL.

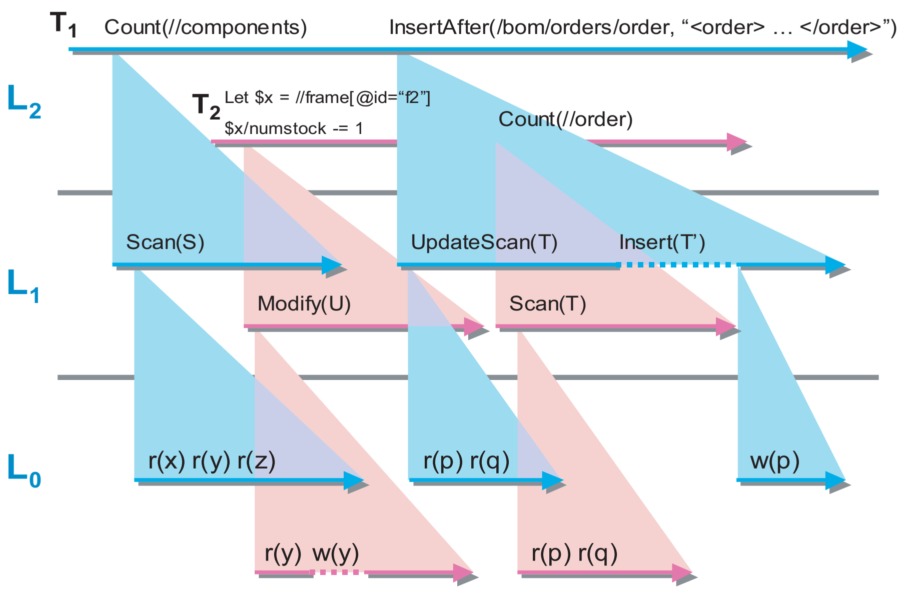


Figure 4.6: Illustration of the layered locking schedule. In the left side of the figure, there is a conflict in the page-level while its parent transactions are compatible in the HR-operation level. In the right side, there is an operation level conflict.

Note that we do not need to apply concurrency control to all of layers, but we can select some layers, as long as protocols in the selected layers satisfy order-preserving conflict serializability. On this condition, the collectness of generated schedules is also assured [38].

4.5 Intra-Transaction Parallelism

Another benefit of using layered locking in management of XML transactions, by grouping page-level operations into subtransactions in the operation level, the exisiting lock manger could handle parallel subtransactions that originate from the same transaction without any additional extension to it. This *intra-transation prallelism* is only possible when higher level operations are compatible, however, this is usual that multiple path queries, which are disjoint, are perfomed for the sake of structural join (**Figure 4.7**).

Although this parallelization does not reduce the number of disk read operations, however, the sorting operations of nodes as preparation for the structural join can be performed in parallel, thus the CPU can utilize the time of disk I/O waits for the sorting of another transaction.

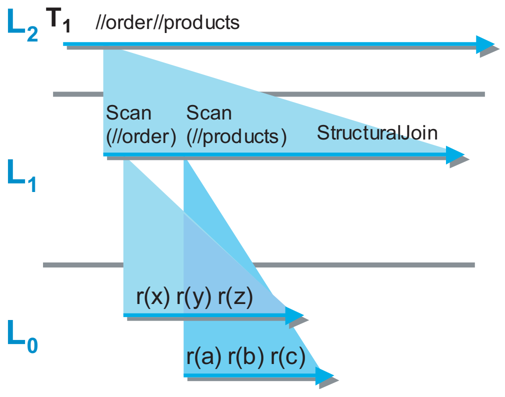


Figure 4.7: An illustration of intra-transaction parallelism. Two Scan operations are performed in parallel.

Chapter 5

Query Prosessing

By combining the fundamental operation describe in Chapter 4, Xerial processes XPath queries. In this section, we begins by introducing the process of the descendant-axis queries, which requires operations called structural joins.

5.1 Structural Join

Li et al.[25] propose the interval index for XML. This method traverses the XML tree in a depth-first and left-to-right manner, gives labels increasing numbers according to the order of visit, and finally associates with each node X, the range of the smallest and the largest numbers in the subtree rooted at X. A node in XML, say x, is represented by range $(start(x), end(x))$, where $start(x)$ and $end(x)$ are integers such that $start(x) \leq end(x)$. Node a is an ancestor of node d iff:

$$start(a) \leq start(d) \leq end(d) \leq end(a) \tag{5.1}$$

Inequality 5.1 is not sufficient to decide parent-child relation, thus to see that a is the parent of node d, the following equation is also used:

$$level(a) = level(b) - 1 \tag{5.2}$$

The *structural join* is an operation which finds node pairs having ancestor-descendant relasionship from given two sets of nodes.

Structural Join Algorithms

The process of a query $A//D$ begins by retrieving two node sets S_A and S_D which have either of tag names A or D from the database. The simplest structural-join algorithm to find all of nodes in ancestor-descendant relationship is to test for each element $a \in S_A$ and $d \in S_D$ whether a is an ancestor of d according to the Inequality 5.1. This method is used in [25], called EE-join, however, it does not utilize structures of the interval indexes but sequentially scans the input. Shu-Yao Chien, et al. improved this algorithm by skipping unnecessary node comparisons, and empirically proved its efficiency [8]. Similar algorithm is also described in [7].

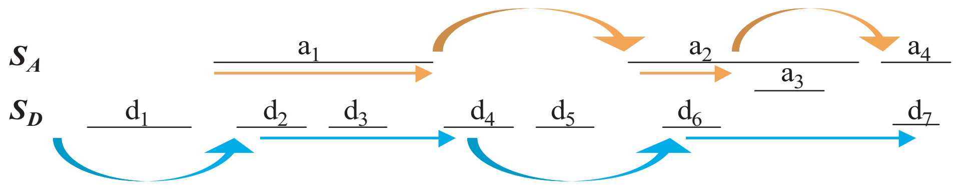


Figure 5.1: An structural join example when node skips occur

The details of the algorithm described in [8] are as follws (see also **Figure 5.1**): first, it sorts the two node sets S_A and S_D in the order of start. Then, it picks up the first element a_1 of S_A and moves the cursor on S_D to the leftmost descendant of a_1, d_2 in Figure 5.1. In this phase, a node d_1 which is not a descendant of a_1 is skipped. After finishing the search for descentands within the range of a_1, the cursor is moved to the next element a_2. It follows the move of the cursor on S_D to the leftmost descendant of a_3, which is d_6 This demonstrates another example of the descendant skip. When the cursor on S_D is moved to d_7, it is no longer nesessary to search the range of a_2 since the next element d_7 is completely after a_2. Therefore, the cursor skips a node a_3 and moves to a_4 (ancestor-skip).

In practive the size of structural join is likely to be smaller than the input size. Then, this algorithm becomes faster than the EE-join since the two steps of ancestor and descendant skips efficiently work to reduce the number of tests of ancestor-descendant relationship. In Xerial, to process ancestor-descendant queries, it utilizes the range query in Chapter 3 to retrieve two node sets S_A and S_D and the above algorithm to join them.

5.2 Subtree Query

To retrieve a subtree from the database, elements in a subtree exists within the range of the root node of the subtree, for given a root node (start, end, level, inverted path ID) of the subtree, Xerial searches the hyper-rectangular region:

$$(\text{start, start, level+1, } IP_{start}(root)) \text{ - } (\text{end, end, } \infty, IP_{end}(root)),$$

where $(IP_{start}(root), IP_{end}(root))$ denotes the root range of the inverted path tree.

5.3 Child Query

Similar to the subtree query, a child query retrieves nodes under some given point (start, end, level, inverted path ID),

$$(\text{start, start, level+1, } IP_{start}(root)) \text{ - } (\text{end, end, level+1, } IP_{end}(root)).$$

In the similar way, sibling queries also can be realized. The capability to handle these types of query is one of the notable characteristics of Xerial. It is because, to process these queries efficiently, existing indexes proposed in the literature such as [8] must maintain additional tree structure that memorize links between sibling nodes and edges from parents to their children. In the transaction environment, such tree structure must also be subject to the concurrency control. This fact dictates that lock management becomes more complex since the lock manager must take care of not only XML nodes but also their connected nodes and edges (for details of such lock management, see [20]). As XerialMDI has no tree edges and no need to keep consistency of such edges, we consider it an advantege that Xerial is easy to support complex concurrency control.

5.4 Path Query

A query expressed in XPath language has to be decomposed into several components so that the query can be processed in Xerial. For example, a path expression /references/book//author is decomposed into /references/book and //author; namely, the continuous path sequences separated by /, but not //. Then, Xerial issues range queries to the following regions, and merge them by a structural join:

$$(root_{start}, root_{start}, 2, IP_{start}(\text{book.references.})) \text{ to}$$
$$(root_{end}, root_{end}, 2, IP_{end}(\text{book.references.}))$$

and

$$(root_{start}, root_{start}, 3, IP_{start}(\text{author.})) \text{ to}$$
$$(root_{end}, root_{end}, \infty, IP_{end}(\text{author.})).$$

Investigation of teh result of the initial query often makes it possible to narrow the region of the subsequqent query. This optimization, a search space elimination, is also a fundamental part of ViST index structure [37]. It constructs an index which is a combination of path prefixes and *start* orders of intervals. It first searches by using path prefixes to retrieve book nodes which have a path prefix references, then from each subtree of the answers, find the target author nodes. As long as path query processing is concerned, their approach is efficient, however, ViST index structure does not facilitate other type of queries such as child and subtree queries.

Another optimization approach other than the above is the *structural-join order selection* [39]. When the number of queries is more than 2, the cost of the structural join varies due to selection of the join orders since, when the percentage of elements to be joined in the ancestor-list is high, the structural joins presents poor performance which does not differ much from EE-join. Therefore, selection of a join order that quickly reduces the number of the join answer is important. To compute this optimal join order, proper estimation of structural join size is crucial, but it seems to be impossible without some statistical information of XML docuemnts. It is because while the cost model in [39] is assumed to be proportional to the size of input, however, the computation time of the structural-join algorithm in the previous section is not necessarily proportional to the size of the input, but it highly depends on the percentage of descendants nodes to be joined [8]. In addition, the proposed estimation algorithms in [39] require that all of the costs to retrieve the target nodes of decomposed path expressions are computable in advance. On the other hand, optimization using the search space elimination changes such costs dynamically, as a consequence, the above two kind of optimization are not applicable.

In Xerial, search space elimination for path expression queries can be implemented more straightforwardly by using the inverted path tree. For example, by searching the range of author. (1.1, 4.5) in **Table 3.2**, we can find only a single range of inverted path which has book.references., i.e. $(4, 4.1)$. Therefore, the above query range is transformed into the following compact form:

$$(root_{start},\ root_{start},\ 3,\ IP_{start}(\text{author.book.references.}))\ \text{to}$$
$$(root_{end},\ root_{end},\ \infty,\ IP_{end}(\text{author.book.references.})).$$

This optimization, however, is applicable only when path structure of an XML document is simple, since the range author. (1.1, 4.5) contains many inverted paths, author.textbook, author.textbook.history, etc. In this case, the optimzed query region becomes a lot of segments of ranges, and to process each of them may not be efficient since it imposes much loads to the lock manager. For such cases, Xerial manages with two region locks and range queries, and one structural join.

5.5 Update Operations

To execute update operations, first of all, we must search positions of their update targets, since database users are not informed of any interval range of the database. It is because the intervals should be transparent to the users since it is not safe to allow them to use or modify intervals directly. Thus, update operations are always used together with some query operations in order to learn where they issue an Insert or Delete operation except Modify Contents operation, since it holds query and update operations concurrently. Once some position is specified, update operations: insertion, deletion, and modification of node values can be performed straightforwardly.

5.6 Query Processing Algorithm for Other Indexes

In this section, we describe query-processing algorithms for other type of XML indexes. Most of XML databases create index structures against one or more of four attributes; start, end, level, and tag-name. Indexes in [40, 25, 22] sort XML nodes in the order of start, and Chien et al. use a combination of (tag-name, start) as a key [8]. We call them *start index* and *tag-start index*, respectively.

In the following sections, we show the process of ancestor, subtree, sibling, and structural join queries for each type of the indexes.

5.6.1 Start Index

The *start index* is sorted in the order of start of intervals; the order is equivalent to that given by a depth-first search of XML nodes. It is easy to retreive ancestor nodes of a given node by searching intervals containing it backward from its *start*. In the similar way, subtrees can be retreived by searching the range of a given nodes.

However, there is a difficulty in *start index* to process a structural join, since it has no search structure for tag names, as a consequence, it has to scan the whole index. In regards to sibling query, first, it tries to find a node in the target level by scanning from the beginning of the index, then it continuously finds next sibling nodes by searching from the position which has the smallest *start* value larger than the end value of the last found node. This can efficiently avoid unnecessary scan of descendant nodes. In the experiment, however, we will show this sibling queries is not as fast as expected since, apparently, *start index* does not localize the nodes in the same level.

5.6.2 Tag-Start Index

The *tag-start index* groups together nodes which has same tag names and sorts them in the order of *start*. This property greatly speeds up the phase of node retrieval and sorting in the structural join algorithm.

It may not be efficient, however, to process subtree and sibling queries. This is because the *tag-start index* decomposes the node sequence of an original XML document into blocks consisting of nodes which have same tag-names. Without some knowledge of all kinds of tag-names exists in the subtree it has to scan, for each tag-tagname, blocks containing it.

In addition, the worse thing is, in the process of sibling queries, it must repeat the same poor algorithm as that of the *start index* for each block of tag names. A solution of this problem is to make additional link structure, however, as we noted before, it is not desireble in transaction environtment.

Chapter 6

Implementation

Xerial is implemented in C++. The program has about 50,000 lines of source codes. It consists of several components, e.g. an database generator, query algorithms, LAS-protocol scheduler, HR-level lock manager etc. The overview of Xerial is illustrated in **Figure 6.1**. To create XML databases, we implemented a program named xml2db. First, it parses original XML documents and creates SAX events, then from these events, xml2db generates a set of XerialMDI nodes by using extensible composite-numbers and stores them to the database. The sort order of these nodes are defined by z-orders provided by the interleave function. We used the BerkeleyDB library [33] to construct B+-trees. The BerkeleyDB is an open source database library and it supports page-level transaction management using the *lock-coupling* technique [28]. 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. Routines of range query and update operations based on extensible composite numbers are also implemented.

6.1 XPath Decomposition

A query expressed in XPath language has to be decomposed into several components so that the query can be processed in Xerial. For example, a path expression /ref/book//author consists of /ref/book and //author; namely, the continuous path sequences separated by /, but not //. Then, Xerial issues range queries to the corresponding regions, and merge the result sets of the intervals by structural join operation [1]. In Xerial, we use a faster version of the structural join algorithm, described in [8].

To create lock schedules of Xerial, we implemented a prototype of XQuery compiler. XQuery [4] is a query language for XML developed in the World Wide Web Consortium

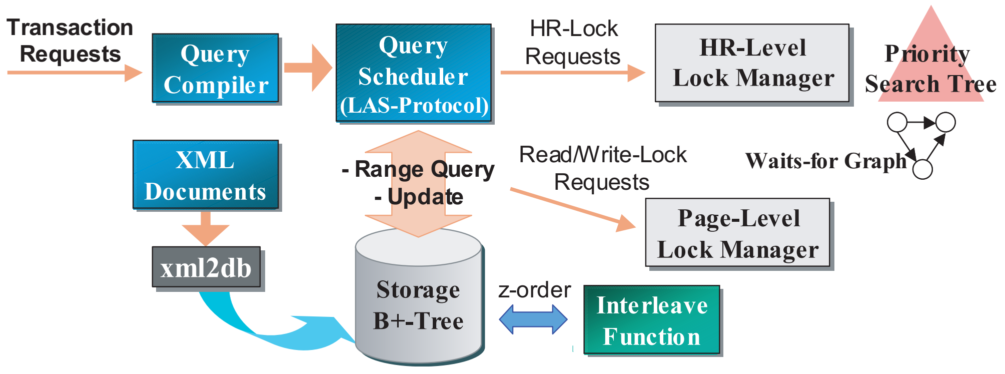


Figure 6.1: Xerial system overview

(W3C). XQuery itself has no update description, but Igor Tatarinov et al. extended them with update clause which can contain some update operations, such as insert, delete, replace, etc. [35] The following syntax illustrates the update operation which searches a region specified by the XPath expression, and binds found nodes to the `$var`, then inserts the XML content under the `$var`:

for $var in *XPath-expr* update $var { insert *xml-content* }

From this syntax, the query scheduler of Xerial generates the layered locking schedules. While evaluating the above XPath expression, the transaction requests IU locks to the corresponding hyper-rectangular regions. Once these locks are granted under the LAS-protocol, it proceeds to access the B+-tree pages with the lock-coupling concurrency control. After finishing them, it releases only the page-level locks, and acquires Insert-mode locks, then executes insertions of the XML contents with write-locks for the disk pages. When there is no update clauses in query statements, the transaction simply acquires some kinds of read-only locks, such as Scan or Read Contents locks.

Chapter 7

Experimental Evaluation

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

7.1 Experimental Settings

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.

Dataset Unfortunately, there seems to be no suitable benchmark to test update performance of XML databases. However, we decided to use Shakespeare's plays in XML format, shakespeare.xml [13], and XML documents provided by XMark benchmark project, standard.xml [32], since they are frequently used as a dataset to test query performances in the literature. Following experiments are conducted with these two XML documents, shakespeare.xml and standard.xml.

7.2 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 [8] to accelerate structural join queries. **Table 7.1** shows the database sizes and construction times of shakespeare.xml and standard.xml 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.

	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 7.1: Database size and construction time

In the following experiments, we measured the average times of five hot-runs for each 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 The process of the ancestor-descendant query, called structural-join [1], first prepares two node sets corresponding to ancestors and descendants by scanning the databases, then merge them and pick up node pairs having ancestor-descendant relationships. The left table in **Table 7.2** shows the performance of structural join query. Since these three indexes use the same structural-join algorithm described in [8], the performance difference depends on how fast they can collect nodes that have 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 save tag name. The start index is weak for 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 7.2** 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.

Subtree Retrieval In shakespeare.xml, we could not see the significant performance difference of subtree retrieval (rightmost table in **Table 7.2**). Therefore, we made the experiment by using the larger document, i.e. standard.xml (**Figure 7.1**). The start index is the most suitable data structure for subtree retrievals as nodes in a subtree is sequentially ordered, and shows the fastest result, though Xerial is fairly comparable to the start index.

Sibling Retrieval Notable usage of sibling node retrievals is to find blank spaces for node insertions, or to compute parent-child joins. Xerial remarkably outperforms the other indexes (**Table 7.2** and **Figure 7.2**), This is simply because, as in the example in Section 2, these indexes except Xerial do not group together sibling nodes.

	//ACT//SPEECH				//SPEECH/SPEAKER				**Subtree** (5031)	**Sibling** level = 4 (1605)	**Sibling** level = 5 (36552)	**Ancestor** level <= 7 (7)
	ACT (185)	**SPEECH** (31028)	**Join** (30951)	**Total** (sec.)	**SPEECH** (31028)	**SPEAKER** (31081)	**Join** (31018)	**Total** (sec.)				
Xerial	0.015	0.593	0.203	**0.811**	0.578		-	**0.578**	0.07	**0.031**	**0.672**	**0.062**
Tag-Start Index	0.015	0.547	0.219	**0.781**	0.437	0.5	0.25	1.281	0.07	4.531	4.875	0.719
Start index	1.718		0.220	1.938	1.688		0.25	1.938	0.06	0.78	1.406	**0.016**

Table 7.2: Query performance in shakespeare.xml. Numbers in the parentheses represent the numbers of answer nodes of the queries.

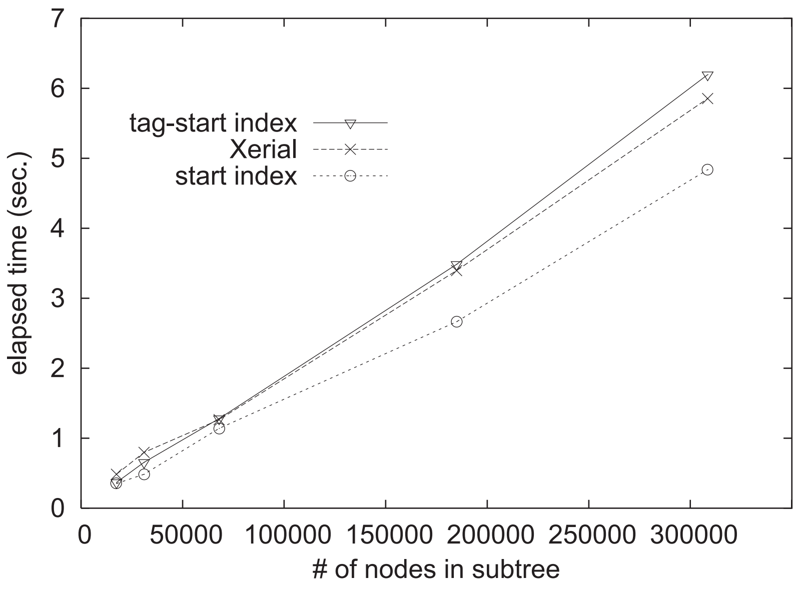


Figure 7.1: Subtree retrieval

Sibling retrieval algorithm of the start index used in the experiment repeats searching the tree for an interval in the target level with depth-first traversal, and skipping 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 tree traversal increases. This inefficiency becomes prominent in the level more than 4 as illustrated in the **Figure 7.2**. 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 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.

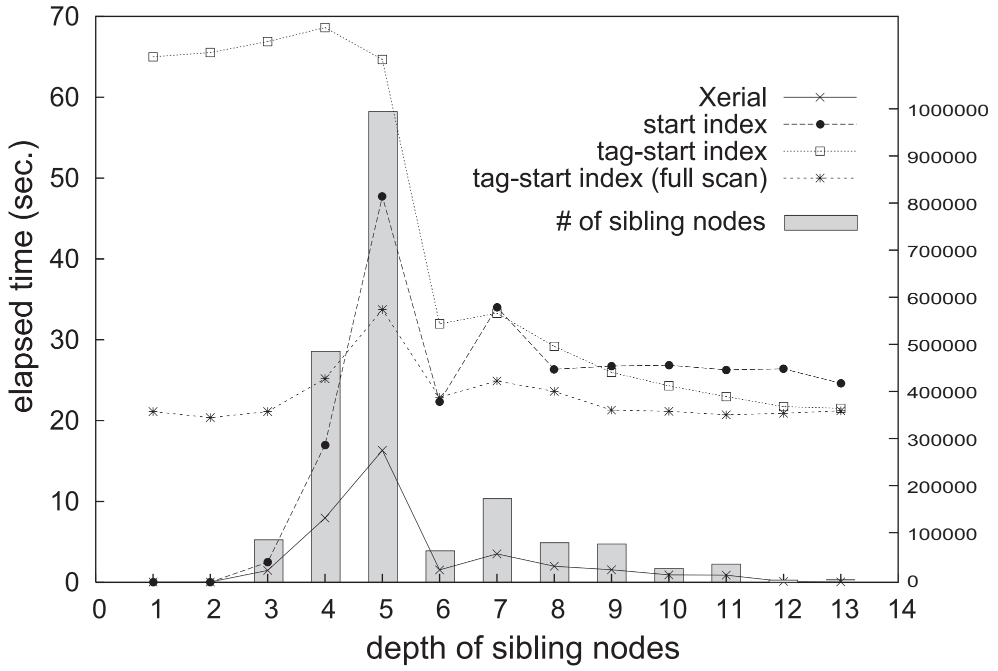


Figure 7.2: Sibling retrieval

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 $start < s \land e < end$, where (s, e) are start and end position of the base node of the query.

Figure 7.3 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 (**Figure 7.4**). Xerial is also effective 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 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.

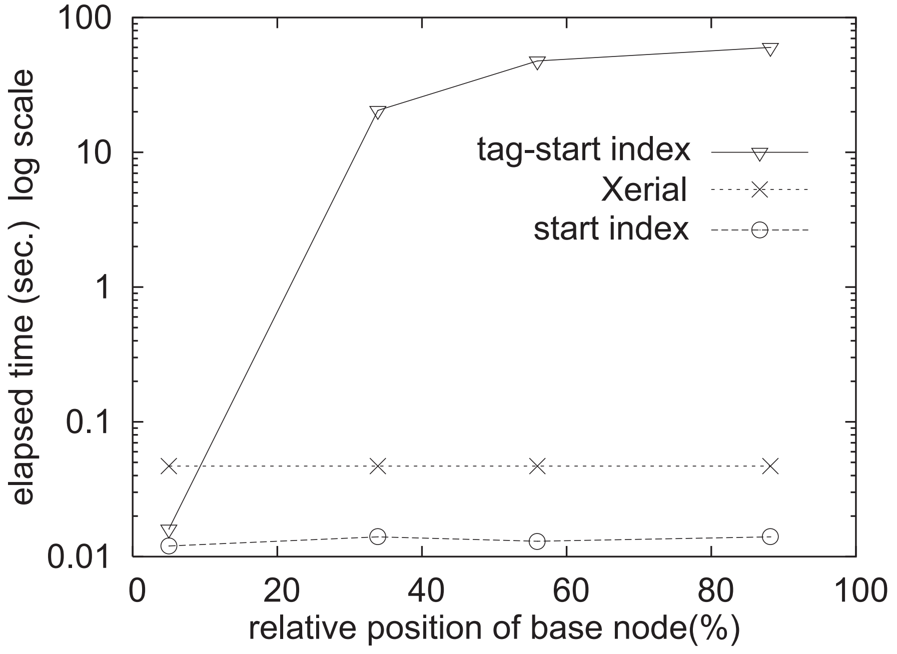


Figure 7.3: Performance of ancestor retrieval for nodes in level 12 in standard.xml

While there is no big difference in terms of database size, our indexing method of XML proved to be efficient for all of the queries described above. Furthermore, we consider the fact that Xerial does not use any secondary index is beneficial to update operations.

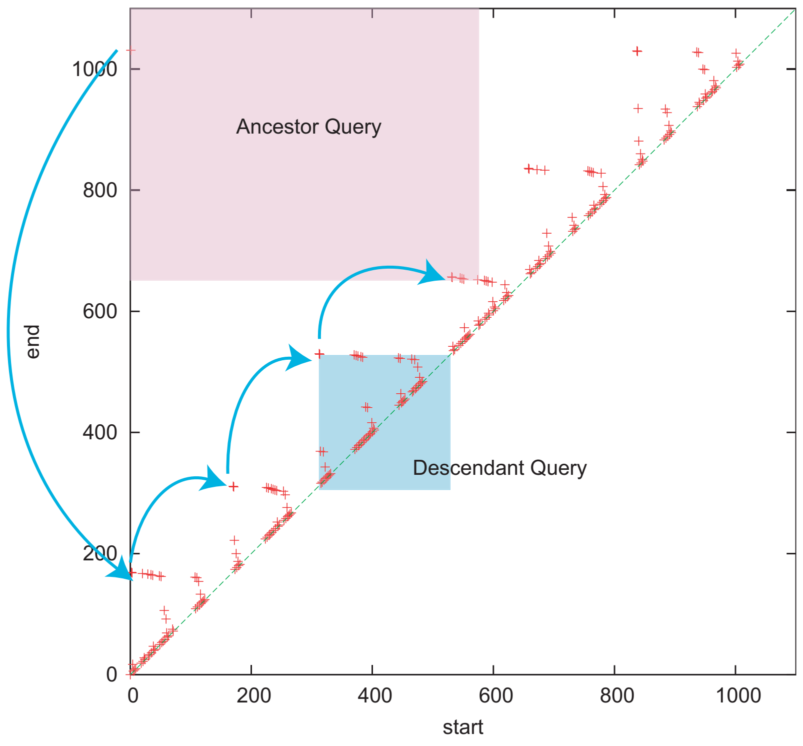


Figure 7.4: Search steps of an ancestor query on the start index

7.3 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 of indexes without 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 of intervals and 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 7.1**). **Figure 7.5** 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 sudden increase of update times as much of the full construction time.

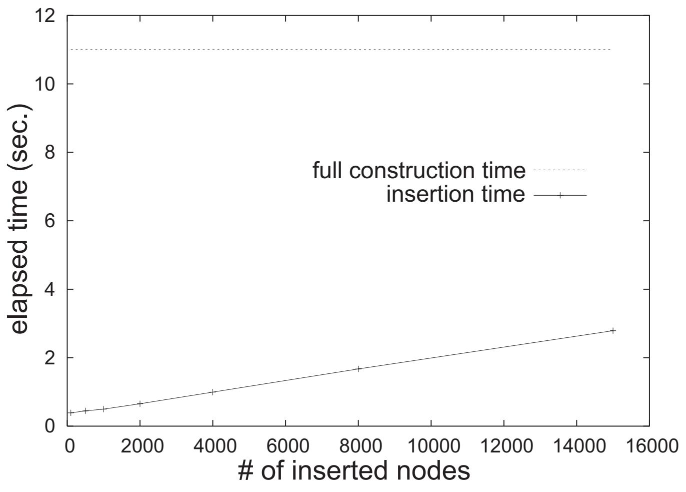


Figure 7.5: Performance of insertions into the front part of shakespeare.xml

7.4 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.

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. Following experiments were conducted with this locking strategy.

The upper left table in **Figure 7.6** shows the work sets of the transactions used in the experiments using shakespeare.xml. The set **S1** puts weight on the read-only transactions, and the set **S2** on the write transactions. As measures of transaction cost, for each transaction, we computed the average execution time when performed alone and the number of nodes retrieved from the database. The maximum number of threads that concurrently accessed the database was set to 50. If some threads had finished its 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 restarted after a while.

Figure 7.7 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 enter such deadlock state.

	Path Expression	# of Retrieved Nodes	Execution Time (sec.)	S1	S2
Path suffix	`//PGROUP/PERSONA`	1057	0.016	20 %	10%
Structural Join	`//PLAY//TITLE`	1068	0.031	20 %	10%
Subtree	`//ACT[n]//SPEECH[m]//*`	223 - 688	0.015	20 %	10%
Subtree	`//PERSONAE[n]//*`	53 - 91	0.016	10 %	10%
Insert (a LINE)	`//ACT[n]//SPEECH[m]`	224 - 688	0.156	10 %	20%
Delete (a LINE)	`//ACT[n]//LINE[m]`	195 – 1142	0.032	10 %	20%
Replace (a LINE)	`//ACT[n]//LINE[m]`	195 – 1142	0.024	10 %	20%

	S1		S2	
	# of Aborted Transactions	Time (sec.)	# of Aborted Transactions	Time (sec.)
Flat Model	54469	774.0	89530	1317.41
Layered Model	**0**	**338.2**	**0**	**485.0**

Figure 7.6: Mixture percentage of transactions. n and m in the path expressions are random numbers used to select a nth or mth element.

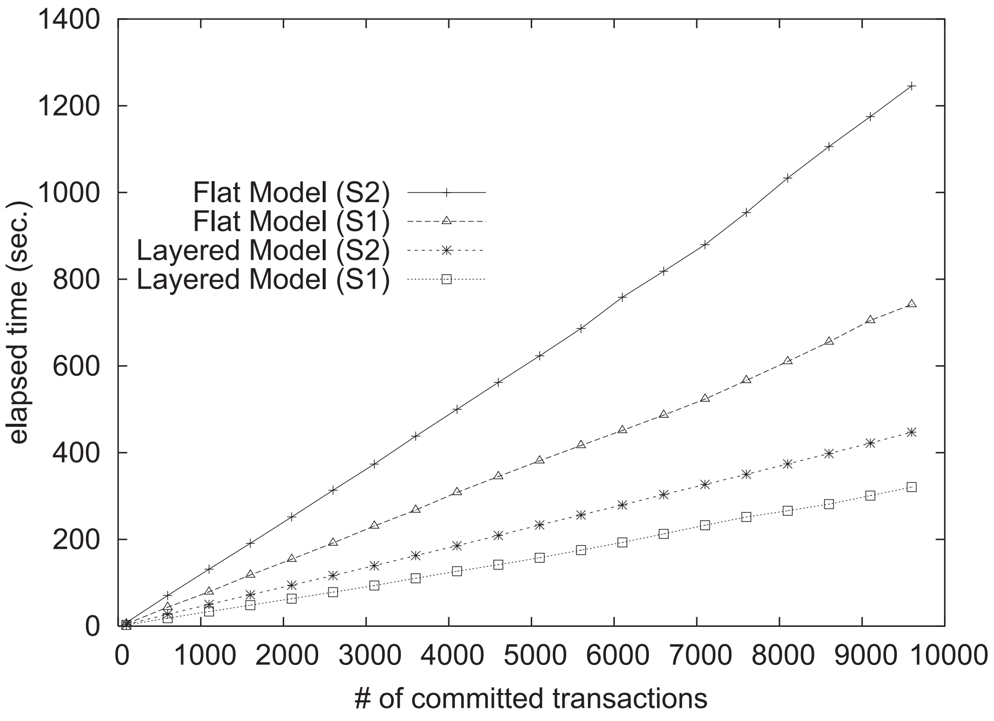


Figure 7.7: Transaction performance of Xerial

Chapter 8

Related Work

8.1 Concurrency Control

There have been a few studies on concurrency control in XML. Studies in [20, 21, 23] use tree models of XML, and they seem not to be capable of processing descendant queries. The proposal of locks for DataGuide [17] has the same weakness. Choi et al. [9] devised precision locks for XPath, however, it provides no experimental results.

As for R-tree, the method in [5] is the state of the art concurrency control. Since it is a page-level lock protocol, the cost of the lock management is lower than that of the layered locking. However, there is room for studying whether this protocol is actually suitable for XML transaction, whose queries usually span the wide range, and whether it can avoid deadlocks.

8.2 Commercial XML Databases

Commercial XML database systems can be classified into two groups, embedded databases on relational database systems and native XML database systems [6].

8.2.1 RDB Mapping Schemes (Embedding XML Databases)

Several mapping schemes to conventional RDB are well described in [15]. A unit of RDB is a table, thus it is not natural to store structured XML document into flat tables. However, there are a lot of situation that simple rules transforming XML structures into flat tables can be inferred, since an XML often has many repetitions of similar-structures. Thus, RDB mapping is useful when *a priori* knowledge of the structure of the XML document is available. However, to put it the other way around, the DTD or the XML Schema must be kept unchanged to avoid dynamic schema conversion whose cost is known to be extremely high. In practice, however, it is difficult to write such a stable and perfect schema that never changes. It is because there is a flexibility in the way of structuring XML document, i.e. it is usual that we later want to add, delete and change the structure of some components, as a consequence, we finally end up rewriting the schema. For the RDB mapping scheme, it is usually equivalent to reconstruction of the database.

In spite of such defects, the biggest reason that vendors are still developing RDB mapping applications is that it can use already available and well-tested reliable database systems to manage XML data. It greatly saves the cost of software-development in comparison with creating a database system from scratch.

8.2.2 Native XML Databases

Native XML databases (for example, BerkeleyDB XML[34], Niagara, eXist, and TIMBER[22] etc.) stores an XML document as it is, without any transformation. It resembles a file system in operating systems. In other words, it is a database of a text collection. Motivation to use native XML databases comes from managing a thousand of small XML documents. Since each document is very small, it is possible to load and parse it within a main memory. To support XPath query against multiple documents, most of native XML databases construct additional secondary indexes on the original XML documents.

The usage of native XML database is often limited to collect and manage small XML documents or to keep spacing or indentation of XML documents. In case that large XML documents are given as an input, native XML databases must break down XML documents into small fragments, i.e. subtrees. However, there are two problems of deciding an optimal or proper fragment of XML and managing parent nodes of these fragments.

Moreover, since a granule of native XML database is a subtree, from the viewpoint of concurrency control, it seems too large and improper for descendant queries which usually span several subtrees. In practice, however, taking into account that current state-of-the-art concurrency control in RDB is row-level locking[38], if a subtree of XML are as flat as a row in RDB, subtree-level locking may be sufficient for concurrency control. However, we should note that the range of locks easily broaden to the whole documents. Lock escalation is a technique which replace a hundreds of node-level locks with a single table-level lock, and it is often used in RDB to decrease the overhead of the lock manager. As we noted, in native XML database, a unit corresponding to a table in RDB is a parent node of small fragments of XML. However, a least common ancestor of broad queries (e.g. A//D) is tend to be a root of the document, thus, when we process such a wide-spanning query, we may have few choices: to lock the root node or several inner nodes to save the number of locks acquired at the sacrifice of concurrency, or to lock each fragment even if it imposes a heavy load on the lock manager.

8.2.3 Updates for XML databases

The work of Igor Tatarinov et al. [35] is the first that takes into account updates of embedded XML databases. However, they did not mention descendant queries and the performance of concurrent transactions.

Several XML databases not using interval containment often label nodes with the contiguous orders [15]. However, node insertions or deletions need heavy maintenance of these numbers. Although [36] provides some efficient renumbering strategies, it is not agreeable in the transaction environment since these extra updates can be a bottleneck of concurrency.

8.3 Other XML Indexes

The 1-index [27] and DataGuide [16] summarize structures of XML documents, by using *bisimilarity* relationship or its variant. These indexes are vulnerable tp heavy descendant queries when XML's structure is complicated. And also, they cannot take subtree-level locks since subtrees in these indexes are decomposed into several individual paths.

Updates for the 1-index is studied in [24]. It solves the problem of cascading updates of the 1-index caused when IDREF edges are added to it. However, processing of descendant queries still remains unsupported.

The Index Fabric [12] is similar to the DataGuide in that it indexes all paths starting from the root node. It supports *refined paths* which can contain descendant-axis and wild-cards(*), etc., and it also manages dynamic updates of the index. Queries not in the set of refined paths, however, require structural join operations which appear not to be supported in the Index Fabric.

The use of the UB-tree [3] to index XML documents is proposed in [2]. Its coordinates are combinations of text values, document IDs, and paths and their appearance orders generated from DTDs. Unlike Xerial, their approach assumes completely static settings.

8.4 Dynamic XML Labeling

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 estimation of the subtree and sibling size to be inserted in future [11]. They proved its bound with such *clues* to be $\Theta(\log n)$, where n is the size of the tree. They also shows the bound without clues is $\Theta(n)$. They said such clues can be estimated from DTD or statistics of XML documents. Although using such information about the structure of XML documents is beyond our scope in this research, such dynamic labeling schemes are indispensable to reduce the size of Xerial databases.

8.5 XPath Processing

To process XPath queries, we decomposed them into several components. More general descriptions of decomposition and query optimization techniques are addressed in [7, 25]. However, query optimization for Xerial may be the different one focused on utilizing the inverted path tree and eliminating the search space of range queries.

ViST [37] index structure utilizes path-prefix of XML and supports top-down descendant traversal. The benefit of this tow-down approach is that it can localize search regions of descendant tags within some subtrees. This combination of subtree regions and tag names is similar to the hyper-rectangular encapsulation of Xerial.

Chapter 9

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 shows 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. It is worth investigating to incorporate such additional index structures into Xerial to improve query performance and transaction throughput.

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 the 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] M. G. Bauer, F. Ramsak, and R. Bayer. Indexing XML as a multidimensional problem. Technical report, 2002. TUM-I0203.

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

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Floresch, J. Robie, and J. Simeon. XQuery 1.0: An xml query language - W3C working draft, November 2003. available at http://www.w3.org/TR/xquery.

[5] K. Chakrabarti and S. Mehrotra. Dynamic granular locking approach to phantom protection in R-Trees. In *Proc. of ICDE*, 1998.

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

[7] 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.

[8] 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.

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

[10] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999. available at http://www.w3.org/TR/xpath.

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

[12] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, 2001.

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

[14] A. Croker and D. Maier. A dynamic tree-locking protocol. In *Proc. of ICDE*, 1986.

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

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

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

[18] J. Gray and A. Reuter. *Transaction Processing - Concepts and Techniques.* Morgan Kaufmann, 1993.

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

[20] S. Helmer, C.-C. Kanne, and G. Moerkotte. Isolation in XML bases. Technical report, the University of Manheim, 2001.

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

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

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

[24] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *Proc. of VLDB*, 2002.

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

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

[27] T. Milo and D. Suciu. Index structures for path expressions. In *Database Theory - ICDT 99*, volume 1540 of *Lecture Notes in Computer Science*, 1999.

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

[29] M. D. P. A. Mukherjee. Experimental comparison of d-rectangle intersection algorithms applied to HLA data distribution. In *Fall Simulation Interoperability Workshop (SIW)*. IEEE, 1997.

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

[31] 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.

[32] 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.

[33] Sleepycat Software. BerkeleyDB. available at http://www.sleepycat.com/.

[34] Sleepycat Software. BerkeleyDB XML. available at http://www.sleepycat.com/.

[35] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. of SIGMOD*, 2001.

[36] 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.

[37] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *Proc. of SIGMOD*, 2003.

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

[39] J. P. Yuqing Wu and H. Jagadish. Structural join order selection for XML query optimization. In *Proc. of ICDE*, Mar. 2003.

[40] 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.