An Enhanced Concurrency Control Scheme for Multi-Dimensional Index Structures

Seok Il Song*, Young Ho Kim**, Jae Soo Yoo*

* Department of Computer and Communication Engineering, Chungbuk National University
prince@netdb.chungbuk.ac.kr, yjs@cbucc.chungbuk.ac.kr
** Electronics and Telecommunications Research Institute. Computer and Software Technology Labs. Internet Service Department.
kyh05@etri.re.kr

Abstract

In this paper, we propose an enhanced concurrency control algorithm that minimizes the query delay efficiently. The factors that delay search operations and deteriorate the concurrency of index structures are node splits and MBR updates in multi dimensional index structures. In our algorithm, to reduce the query delay by split operations, we optimize exclusive latching time on a split node. It holds exclusive latches not during whole split time but only during physical node split time that occupies small part of whole split time. Also to avoid the query delay by MBR updates we introduce partial lock coupling(PLC) technique. The PLC technique increases concurrency by using lock coupling only in case of MBR shrinking operations that are less frequent than MBR expansion operations. For performance evaluation, we implement the proposed algorithm and one of the existing link technique-based algorithms on MIDAS-III that is a storage system of a BADA-III DBMS. We show through various experiments that our proposed algorithm outperforms the existing algorithm in terms of throughput and response time.

1. Introduction

Multi-dimensional index structures are the hearts of similarity search systems based on multidimensional feature vectors search such as GIS, content-based image retrieval systems, multimedia database system and so on. For the couple of past decades, various multidimensional index structures have been proposed, for example R-Tree[1], R*-Tree[12], TV-Tree[8], X-Tree[16], SS-Tree[4], SR-Tree[13], CIR-Tree[6] and Hybrid-Tree[7]. They need concurrency control and recovery methods to be used for real life applications in multi-user environments. Consequently, several concurrency control and recovery methods have been proposed [5, 10, 11, 15, 17].

In multidimensional index structures, split operations need rather longer time than other unidimensional index structures such as B-Tree and B+-Tree. Generally the entries of internal nodes are not ordered so calculating split dimensions and split positions require expensive costs. Most of existing concurrency control algorithms for multi-dimensional index structures hold exclusive locks(x-lock) or latches(x-latch) on the nodes where split operations are being performed and block search operations. Split operations ascend index tree to propagate splits to ancestor nodes and may cause other splits of ancestor nodes. These split operations are one of the primary factors that deteriorate the concurrency of multi-dimensional index structures. Also, minimum bounding region(MBR) update operations block search operations. The MBR update of a node is less expensive than a split operation. However, MBR updates are much more frequent than split operations so they significantly deteriorate the concurrency of index structures.

Even though several concurrency control algorithms were proposed for multi-dimensional index structures, none of them could completely avoid the query delay. Actually, it is impossible to demolish the above query delay completely but we can minimize the query delay. In this paper, we propose an enhanced concurrency control algorithm that minimizes the query delay. We introduce a partial lock coupling(PLC) technique to decrease the query delay by MBR updates. Also, to alleviate blocking factors by split operations, we propose a split method that optimizes x-latch time during node splits.

The remainder of this paper is organized as follows. In section 2, we describe and analyze existing concurrency control algorithms for multidimensional index structures. We then describe the proposed algorithm in detail in section 3. In section 4, we perform experiments to show the superiority of our concurrency control algorithm and discuss the results of the performance evaluation. Finally section 5 concludes this paper.

0-7695-0996-7/01 $10.00 © 2001 IEEE

2. Related works and motivations

Several concurrency control algorithms for multi-dimensional index structures were proposed. They can be classified simply into Link-based and Lock Coupling-based algorithms. The lock-coupling based algorithms[5,17] release the lock on the current node when the lock on the next node to traverse is granted while processing search operations. While processing node splits and MBR(Minimum Bounding Region) changes, the scheme holds multiple locks simultaneously that significantly degrade concurrency.

On the other hand, the link-technique based algorithms [10, 11, 15] were presented to solve the problems of lock-coupling based concurrency control algorithms. The schemes need not perform lock-coupling during search operations. That is, at most one lock is held for each search operation. However, while backing up trees for node splits and MBR updates, the scheme employs lock-coupling, i.e., it keeps the child node write-locked until a write-lock on the parent is obtained.

The link-technique, proposed by Leman and Yao in [14], was originally for B-Tree. The tree structure is modified so that all nodes at the same level are chained all together through a right-link on each node, which is a pointer to its right sibling node. When a node is split into two nodes, appropriate right links are assigned to them. All nodes in a right link chain at the same level are ordered by their highest keys. When a search process visits a node that was split and not yet propagated to the parent node, it detects that the highest key on that node is lower than the key it is looking for and correctly concludes that a split must have taken place. This guarantees that at most one lock is needed at any case, so insert operations can be performed without blocking search processes.

Unfortunately, in multi-dimensional index structures is no such an ordering between nodes at the same level. In that reason, the algorithm proposed in [11] assigns logical sequence number(LSN) at each node besides right links and an entry associated with a node has the LSN of the node. The ordering of LSNs is used to compensate missed split. However, while ascending the trees to perform node splits and MBR updates, this algorithm employs lock coupling, i.e., it keeps child node write locked until a write lock on the parent is obtained. The lock on the child node may be kept during I/O time in certain case [11]. It degrades the concurrency of the index trees. Also in this algorithm, each entry of internal nodes has extra information to keep the LSNs of associated child nodes. This extra information reduces storage utilization.

Another link-based concurrency control algorithm[10] for multi-dimensional index structures, called CGIST was proposed for reducing the extra information problem. It eliminates the needs to keep extra information by using global sequence number. In that research, global counter is introduced as the method to eliminate the extra information from the internal node entry. However, it is accompanied by some side effects. The node sequence number(NSN) which is the LSN of [11] is taken from a tree-global, monotonically increasing counter variable. During a node split, this counter is incremented and its new value assigned to the original node and the new sibling node receives the original node's prior NSN and right-link.

In order for the algorithm to work correctly, when splitting a node, we must attain the lock on its parent node first, split the node and assign the NSN, and increment global counter. Because of that reason, while processing node splits it keeps multiple locks on two or more levels. This affects search operations and explicitly increases the wait time of search operations.

The concurrency control algorithms briefly explained above get multiple locks or latches exclusively on index nodes from multiple levels participating in node splits and MBR updates. The exclusive locks or latches block concurrent search operations and as the results, overall search performance is degenerated extremely.

[15] is the most recent link-technique based concurrency control algorithm for multi-dimensional index structures trying to solve the problems mentioned in the previous paragraph. It introduces top-down index region modification(TDIM) technique. That is, when an insert operation traverses an index tree to find the most suitable node for a new entry, MBR updates are performed. In addition to TDIM, the locks that are obtained on nodes during MBR updates are compatible with queries. It is achieved by the modification of MBR in a piecemeal fashion. In [15], also, optimized split algorithms are proposed such as copy based concurrent update(CCU) and copy based concurrent update with nonblocking queries(CCU_NQ).

The basic idea of the CCU is to perform a split on a local copy of a node rather than the shared copy on a buffer pool. Queries are free to access the shared copy of the node while the split is in progress. Once the split completes, the changes are copied back to the shared data structure using exclusive locks. By adopting this approach, queries may now be blocked only for the duration of the copy back rather than the entire split process.

The CCU_NQ makes queries completely wait free, not blocking even for the "copy back" interval as in the CCU scheme. This wait freedom for queries is achieved by atomically switching the existing and updated copies of a node during a node split. To atomically switch between different versions of a node, it has to ensure that no operations are left using the older versions. To ensure this, it introduces a logical address(LA) for each node.

We skip the detailed descriptions on CCU_NQ to save space.

The TDIM technique has some problems. It eliminates the necessity of lock-coupling in [11,10] during insert operations. However, it never considers delete operations. We need to perform exact match like tree traversal to find a target entry of a delete operation. Since multidimensional index structures has multiple paths from the root node to a target entry, we can not assure that the current node we visit is the correct ancestors of a target node that contains the target entry. Consequently, to modify MBR in top-down fashion, we must modify MBR after locating the target entry. The problem is not completely solved with these works.

When delete operations and insert operations are performed concurrently using TDIM to update MBRs, index trees may reach to inconsistent states since TDIM does not perform lock-coupling. We can image the following situation easily. An insert operation that tries to insert an entry(NE) visits a node(N) and chooses an entry(E) that contains a child node(CN) and its MBR. The insert operation concludes the MBR does not need to be modified and proceeds with its tree traversal. Subsequently, a delete operation visits N and modifies the MBR of CN in E. This MBR shrinking may exclude NE from the MBR of CN and the index tree becomes under the inconsistent state. Therefore, the TDIM can not be applied in real life applications without modifying MBR updates algorithm in some or most part because it can not handle delete operations that are necessary in real life applications.

Also, the CCU and CCU_NQ reduce the delay of queries extremely but they are not efficient. They need extra spaces to perform split operations and the CCU_NQ must perform garbage collection works periodically. These features make the implementation of the algorithm very difficult. The simplicity of an algorithm reduces the development costs.

In this paper we propose a novel MBR updates algorithm called partial lock coupling(PLC) that avoids lock-coupling in most cases and considers delete operations together. Also we introduce a simple split algorithm that minimizes the delay of queries without extra spaces and any additional works.

3. The proposed algorithm

In this section, we describe the proposed algorithm in detail. As mentioned in the previous section, the most crucial parts of our algorithm are MBR updates and node splits. Therefore, we explain these two algorithms first and describe overall algorithms after them.

3.1. MBR updates with partial lock coupling(PLC) technique

The existing concurrency control methods such as [10,11] employ latch(lock) coupling to maintain consistency of index tree while updated MBR is propagated or an overflow occurs. Even though the TDIM [15] was proposed to solve the problems, it is not a complete algorithms as we explain in section 2. In our concurrency control algorithm, we modify MBRs in bottom-up manner like [10,11]. However, unlike the existing concurrency control method, we avoid lock coupling partially by introducing PLC.

The PLC employs lock coupling only in case of the operations that cause MBR shrinking such as node splits and deletes. In case of the operations of MBR expansion, it modifies MBRs without using lock-coupling. Since usually MBR expansion cases are much more frequent than MBR shrinking cases in multidimensional index structures, the PLC guarantees high concurrency.

However, to use the PLC technique correctly, we must avoid the case that index trees are corrupted. Since we partially use lock-coupling, index trees may reach inconsistent states in some cases without proper compensation actions. We will describe this problem in the following paragraphs.

When an entry is inserted into a node, the MBR of the node is expanded to include the new entry and propagate the modified MBR to ancestors. Until the entry is deleted by another transaction or the transaction that inserts the entry is rollbacked, the expanded MBR is never shrunken. We use these properties to propagate updated MBR to ancestors. When an entry is placed on a leaf node and the MBR of the leaf node is changed, we will propagate the change to ancestors. While we propagate the changed MBR, the new entry just placed on the leaf node never can be deleted by other transactions since we hold x-lock on the leaf node. Therefore, we only need to check if the MBR of the node's parent contains the new entry. If the MBR associated with the leaf node in the parent node contains the new entry, we do not need to change MBR any more. Otherwise, we change the MBR in the parent to include the new entry and ascend the tree to propagate the change in the same manner.

During delete operation, we have to employ lock coupling. Checking if the MBR contains the new entry is not enough when the delete operation and insert operation are concurrently performed. The shrunken MBR due to the delete of an entry can be expanded by other insert operations before the transaction that deletes the entry is committed. Therefore we cannot decide whether the MBR should be modified in case of delete operations. In this case, lock-coupling can resolve the problem.

3.2. Node splits with minimal query delay

In multi dimensional index structures, node splits are usually very expensive operations because the entries of internal nodes are not ordered. Most of existing concurrency control algorithms for multi-dimensional index structures hold exclusive locks or latches on the nodes where split operations are being performed and block search operations. Split operations ascend index tree to propagate splits to ancestor nodes and may cause other splits of ancestor nodes.

Generally, the node splits are performed in the two steps. The first step is to compute split dimensions and split positions. This step takes rather long time since the entries of index nodes are not ordered. The second step is to divide the overflowed node into two nodes physically with the split dimensions and split positions. As described earlier, the first step takes up most time of split operations. In figure 1, we compared the time of the first step and the second step. As we can see in the figure, the first step occupies about 93% of the total split time. In the proposed algorithm, we hold x-latch on the overflowed node only during the second step and during the first step, s-latch are held on the overflowed node. It enables queries to access overflowed nodes during the first step that occupies most of the total split time.

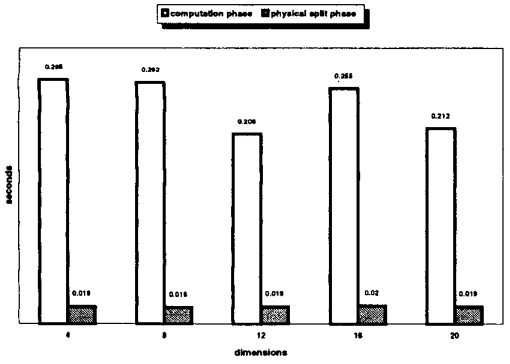

Figure 1 Computation time vs. physical split time

3.3. Insert operations

An insert operation is carried out in two stages. In the first stage, we traverse the tree from root node to find the leaf node to insert the new entry. In this stage, we store the nodes on the path we take while descending the tree to a stack, called the path stack. In the second stage, the new entry is inserted to the leaf node. After adding the new entry to the node, if the leaf's MBR has changed, we propagate the updated MBR to its ancestor nodes until we reach a node that does not need to be changed any more. On the other hand, a split operation proceeds in the leaf node if it does not have enough room to accommodate the new entry. We serialize split operations with tree lock. The tree-lock is used for a recovery scheme that is not mentioned in this paper. Before performing splits we must obtain tree lock first. Subsequently, we must insert a new internal entry that results from the split operation to the parent node and modify the MBR for the split node. If overflow occurs in the parent node recursively, we split the parent node like the leaf split. Above steps are repeated until we reach at a node with enough room to accommodate a new entry or split the root.

Figures 2, 3, 4 and 5 show the pseudo code of the insert operation of the proposed concurrency control algorithm in this paper. The function FindNode in the figure 3 finds a leaf node for a new entry. The individual procedures do the following. FindNode descends to the leaf node to insert a new entry, obtaining s-latch on internal nodes and recording the path along the way, and finally gets s-latch and x-lock on the leaf. We obtain x-lock on the leaf node for two reasons. The first reason is that as we described in section 3.1, the x-lock is used to guarantee the consistency of index trees while performing MBR updates with PLC technique. The second reason is for recovery issues that are not mentioned in this paper.

The function SplitNode in the figure 5 is called when the leaf node does not have enough room to accommodate the new entry. It splits the leaf node to cope with the overflow as described above recursively. When ascending index trees to propagate node splits, we get x-lock on the parent node first. We obtain the x-lock on the parent node because of the following two reasons. The PLC requests s-latches on nodes while processing MBR updates and the SplitNode holds s-latches on nodes that split operations are being performed during computation phase. This situation may drive index trees to inconsistent states. To resolve this situation we use x-lock. The MBR updates operation, also, holds x-locks on nodes during MBR updates. The second reason is to perform PLC. The function FixMBR in the figure 4 is called to propagate the changed MBR when the leaf's MBR is changed. The FixMBR performs lock-coupling during MBR shrinking. That is, it keeps x-lock on the current node until it obtains x-lock on the parent of the current node in case of MBR shrinking.

3.4. Search operations

Processing search operations in the proposed algorithm is the same as that of [10, 11]. Our search operations only use latches to read index nodes and does not perform latch-coupling. It means that search operations are blocked only by node splits, since we hold s-latches on index nodes while performing MBR updates. It will increase the concurrency of search operations extremely.

```
Function InsertEntry( Entry leafentry,
                      Node rootnode )

Function Start
leafnode := FindNode(leafentry, root, path);
If(overflow is occurred in leafnode due to leafentr)
   Obtain tree lock;
   Split(leafentry, leafnode, path);
   Release tree lock;
   Release all locks;
   Function End;
End If
Release s-latch on leafnode;
Obtain x-latch on leafnode;
Add leafentry to leafnode;
If ( the MBR of leafnode is changed )
   Release x-latch on leafnode.;
   FixMBR( leafnode, path );
   Release all locks;
   Function End
End If
Release x-latch on leafnode;
Release all locks;
Function End
```

Figure 2 Insert algorithm(1) - InsertEntry

```
Function FindNode( Entry leafentry, Node node,
                   PathStack path)

Function Start
Obtain s-latch on node;
currentlevel =node.level ;
Push [node, node.nsn] into path;
Loop
   childentry[Node node, MBR mbr] := Select the child
   entry from node;
   Subtract 1 from currentlevel;
   Release s-latch on node;
   node := childentry.node;
   s-latch on node;
   If ( currentlevel == leaf level )
      Obtain x-lock on node;
   End If
   If ( global_nsn > node.nsn )
      Select the most appropriate node from its sibling
      nodes;
   End If
   If ( currentlevel == leaflevel )
      Exist Loop
   End If
   Push [node, node.nsn] into path;
End Loop
Function End
```

Figure 3 Insert algorithm(2) - FindNode

```
Function FixMBR ( Node node, PathStack path )

Function Start
parentnode := POP( path ) //decide the parent of node;
If ( MBR Shrinking ) //delete operations
   Obtain x-lock and s-latch on parentnode;
   Release x-lock on node;
Else
   Obtain x-lock and s-latch on parentnode;
End If
Modify the mbr for node in parent parentnode;
Release s-latch on parentnode;
If ( the MBR of parentnode is changed )
   If ( MBR Expansion )
      Release x-lock on parentnode;
   End If
   FixMBR ( parentnode, path );
End If
Function End
```

Figure 4 Insert algorithm(3) - FixMBR

```
Function SplitNode(Entry leafentry, Node node,
                   PathStack path )

Function Start
Compute split dimension and split position of node;
newnode :=allocate a new node;
Obtain x-lock and x-latch on newnode;
entries := with split dimension and split position, select
entries from node to be moved to newnode;
Copy entries to newnode;
If ( node is not latched in exclusive mode )
   Release s-latch of node and obtain x-latch on node;
End If
Delete entries from node and reorganize node;
Copy sibling pointer of node to newnode and set sibling
pointer of node to newnode;
Copy node.nsn of node to newnode;
Increase global_nsn and install its value as the node.nsn;
Create an internal entry internalentry[newnode, mbr];
Release x-latch of node and newnode;
parentnode := POP( path ), i.e. decide the parent of node.;
Obtain x-latch and x-lock on parentnode, and modify the
mbr for node in parentnode;
If ( node is not leaf node )
   Release x-lock of node and newnode;
End If
If ( overflow occurred in parentnode due to internalentry )
   SplitNode( internalentry, parentnode, path );
End If
Add internalentry to parentnode.;
Release x-latch on parentnode;
If ( the MBR of parentnode is changed )
   Release x-latch on parentnode;
   FixMBR(parentnode, path );
End If
Function End
```

Figure 5 Insert algorithm(4) - SplitNode

3.5. Delete operations

The delete operations of our proposed algorithm proceeds in two phases like the insert operations. The first phase is to locate the target entry of a delete operation. Locating the target entry is performed in similar way to an exact match algorithm. After deleting the target entry, the second phase commences. If the MBR of the node that contained the target entry is changed, the delete operation calls the FixMBR. When calling the FixMBR, it does not release the x-lock on the node to perform lock-coupling.

4. Performance evaluation

To evaluate the performance of the proposed algorithm relative to the concurrency control algorithm for GiST(CGIST), we applied the proposed algorithm(RPLC) to CIR-Tree and implemented it as an access method for MiDAS and compared it with the CGIST. We did not compare our algorithm with [15], since as we described the reason in section 2 we thought that it is not a complete concurrency control algorithm. The CGIST implementation was also applied to CIR-Tree and implemented on MiDAS-III. Actually, the CGIST consider the no phantom read consistency. However, the implemented CGIST in this paper just support the repeatable read consistency because currently only our proposed algorithm supports the repeatable read consistency.

Table 1 Performance parameters and values

Parameters	Values
DS(Database Size)	50000, 100000
NS(Node Size)	4K, 16K
NB(Number of Page Buffers)	40, 80, 120
ND(Number of Dimension)	10, 20
K(Number of the K of K-NN Queries)	1, 2, 4
NP(Number of processes)	50, 100

Our experiments are performed for various sizes of data set and various performance parameters such as node size, number of page buffers of MiDAS-III and so on. Table 1 shows the notations, the descriptions of the performance parameters and the values of each parameter are presented. To save space, we discuss the performance comparison only when randomly generated 5 n 0000, 20-dimensional feature vectors are used, the node size is 16Kbytes and the number of page buffers is 120. Since our experiments are not to evaluate performance of an index structure but to evaluate that of a concurrency control algorithm, we just use randomly generated data set. The platform was dual Ultra Sparc. processor, Solaris 2.5 with 128 Mbytes main memory. The maximum number of concurrent processes is 100. We experimented with different workloads of insert and query operations.

4.1. Results of various number of insert and search workloads

Figures 6, 7, 8 and 9 compare the performance of CGIST and RPLC in terms of response time and throughput when the number of insert and search processes is varied. Figure 6 shows the response time of search operations when the ratios of the number of search processes to the number of insert processes are varied from 90(search)-10(insert) to 10-90. The response time of RPLC is constant regardless of the change of the ratios while that of CGIST increases according to the increase of insert processes. Figure 7 shows the response time of insert operations under the same condition. Like the response time of search processes in figure 6, the performance of RPLC is much better than that of CGIST. Figure 8 and 9 show the throughput of search processes and insert processes respectively. The throughput of search processes of RPLC increases slightly according to the increase of ratios of insert processes to search processes while that of CGIST is reduced rapidly.

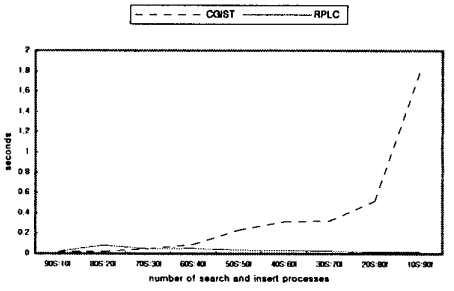

Figure 6 Response time of search transactions

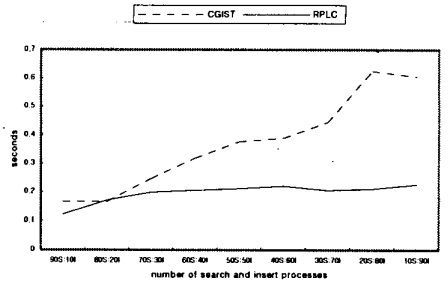

Figure 7 Response time of insert transactions

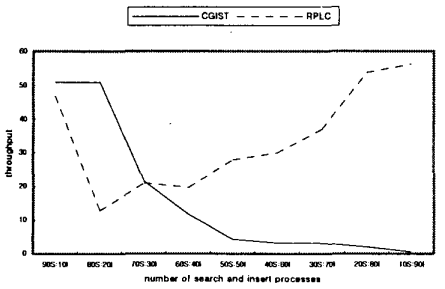

Figure 8 Throughput of search transactions

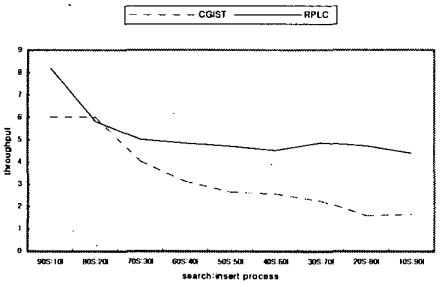

Figure 9 Throughput of insert transactions

4.2. Results of Fixed Search and Varied Insert Workloads

Figures 10, 11, 12 and 13 show the performance of CGIST and RPLC in terms of response time and throughput when the number of search processes is fixed as 40 and that of insert processes is varied from 10 to 60. Through these comparisons, we intended to show that search operations of RPLC are constant regardless of the number of insert processes. Figure 10 shows that the response time of search operations is almost constant even though the number of insert processes increase from 10 to 50. Also, the throughput of search processes is not affected by the number of insert processes as we can see in figure 12. The performance of insert processes is the same as that of search processes in figures 11 and 13.

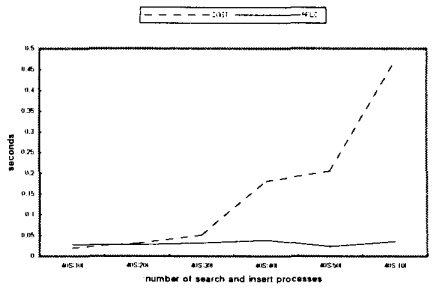

Figure 10 Response time of search transactions with fixed number of search transaction s

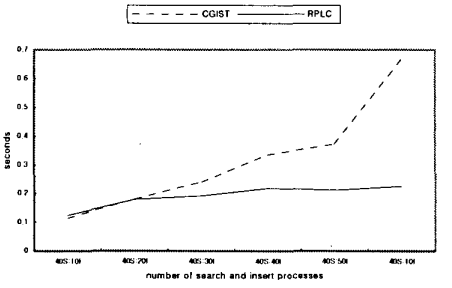

Figure 11 Response time of insert transactions with fixed number of search transactions

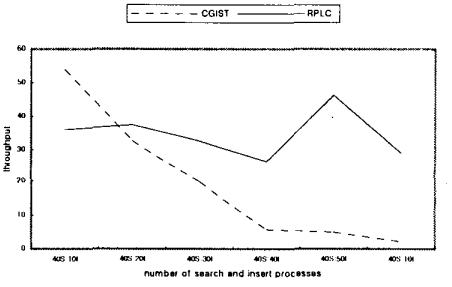

Figure 12 Throughput of search transactions with fixed number of search transactions

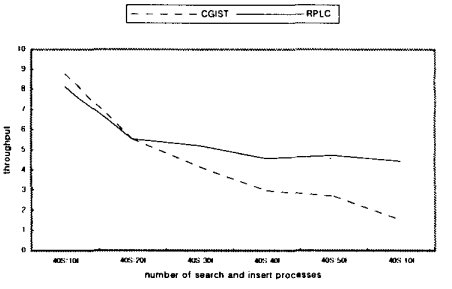

Figure 13 Throughput of insert transactions with fixed number of search transactions

4.3. Discussions

The results described in the section 4.2 indicate that RPLC achieves up to about 6 times in search response time and about 2 times in insert response time, search throughput and insert throughput over CGIST. The RPLC also scales very well with the number of processes and the ratios of insert processes to search processes. The reason why the RPLC outperforms the CGIST is very explicit. Search operations are blocked only by split operations during physical node split time in the RPLC. However, in the CGIST, they are blocked when MBR updates and node splits are performed. The MBR updates and node splits in the CGIST holds multiple x-latches on nodes from multiple levels. Also, the node splits in the CGIST hold x-latches on nodes where node splits are being performed during whole split time. These factors extremely deteriorate the search performance and insert performance. However, the RPLC repeats frequently latching and releasing during split operations to provide higher concurrency. It may deteriorate overall concurrency slightly.

5. Conclusions

In this paper, we have proposed an enhanced concurrency control algorithm for multidimensional index structures. Our concurrency control algorithm alleviates the problems that delay search operations and deteriorate the overall concurrency of index structures. The proposed algorithm reduces the query delay by split operations by optimizing exclusive latching time on a split node. That is, only during the physical split phase we hold x-latches that exclude queries. Also, to avoid the query delay by MBR updates we introduce PLC technique. It uses lock coupling only in case of MBR shrinking operations that is less frequent than MBR expansion operations and augments the overall concurrency extremely. In experimental comparisons with CGIST, we have shown that our proposed algorithm outperforms it in terms of various factors. Our algorithm achieves up 4 times smaller response time and 2 times higher throughput over the CGIST. Currently our algorithm does not provide no phantom read consistency yet. We will consider this isolation level in the next research.

Acknowledgement
This work was supported by Korea Science and Engineering Foundation(KOSEF. 1999-1-303-007-3).

References

1. A. Guttman. "R-Trees: a dynamic index structure for spatial searching," Proceeding of International Conference ACM SIGMOD, 1984, pp. 47-57.
2. C. Mohan, D. Harderle, B. Lindsay, H. Pirahesh and P. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging," ACM TODS, 17 (1), March 1992, pp. 94-162.
3. C. Mohan and F. Levine. "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," Proceeding of International Conference ACM SIGMOD, June 1992, pp. 371-380.
4. D. A. White and R. Jain. "Similarity Indexing with the SS-tree, Proceeding of International Conference On Data Engineering," 1996, pp. 516-523.
5. J. K. Chen and Y.F. Huang. "A Study of Concurrent Operations on R-Trees," Information Sciences 98, 1997, pp 263-300.
6. Jae Soo Yoo, Myung Geun Shin, Seok Hee Lee, Kil Seong Choi, Ki Hyung Cho and Dae Young Hur. "An Efficient Index Structure for High Dimensional Image Data," Proceedings of AMCP'98, 1998, pp. 134-147.
7. K. Chakrabarti and S. Mehrotra. "The Hybrid Tree : an index structure for high-dimensional feature spaces," Proceeding of ICDE Conf., 1999, pp. 440-447.
8. K. I. Lin, H. Jagadish and C. Faloutsos. "The TV-tree : An Index Structure for High Dimensional Data," VLDB Journal, Vol 3, 1994, pp. 517-542.
9. M. Chae, K. Hong, M. Lee, J. Kim, O. Joe, S. Jeon and Y. Kim. "Design of the Object Kernel of BADA-III: An Object-Oriented Database Management System for Multimedia Data Service," The 1995 Workshop on Network and System Management, 1995.
10. M. Kornacker, C. Mohan and J. M. Hellerstein. "Concurrency and Recovery in Generalized Search Trees," Proceeding of International Conference ACM SIGMOD, May 1997, pp. 62-72.
11. M. Kornacker and D. Banks. "High-Concurrency Locking in R-Trees," Proceeding of International Conference VLDB, September 1995, pp. 134-145.
12. N. Beckmann, H.P. Kornacker, R. Schneider and B. Seeger. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," Proceeding of International Conference ACM SIGMOD., 1990, pp. 322-331.
13. N. Katayama and S. Satoh. "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," Proceeding of International Conference ACM SIGMOD, May 1997.
14. P. L. Lehmann and S.B. Yao. "Efficient Locking for Concurrent Operations on B-Trees," ACM TODS 6(4), December 1981, pp. 650-670.
15. K.V. Ravi. Kanth,.D. Serena; A. K. Singh, "Improved concurrency control techniques for multi-dimensional index structures," Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing (IPPS/SPDP), 1998., pp. 580 –586.
16. S. Berchtold, D. A. Keim, and H. P. Kriegel. "The X-Tree: an index structure for high-dimensional data," In Proc. of VLDB Conf., 1996, pp. 28-39.
17. V. Ng and T. Kamada. "Concurrent Accesses to R-Trees," Proceeding of Symposium on Large Spatial Databases, 1993, pp. 142-161.