Improved Concurrency Control Techniques for Multi-dimensional Index Structures[*]

K. V. Ravi Kanth, David Serena, Ambuj K. Singh
Department of Computer Science,
University of California, Santa Barbara, CA 93117.

Abstract

Multi-dimensional index structures such as R-trees enable fast searching in high-dimensional spaces. They differ from uni-dimensional structures in the following aspects: (1) index regions in the tree may be modified during ordinary insert and delete operations, and (2) node splits during inserts are quite expensive. Both these characteristics may lead to reduced concurrency of update and query operations. In this paper, we examine how to achieve high concurrency for multi-dimensional structures. First, we develop a new technique for efficiently handling index region modifications. Then, we extend it to reduce/eliminate query blocking overheads during node-splits. We examine two variants of this extended scheme – one that reduces the blocking overhead for queries, and another that completely eliminates it. Experiments on image data on a shared-memory multiprocessor show that these schemes achieve up to 2 times higher throughput than existing techniques, and scale well with the number of processors.

1 Introduction

Multi-dimensional indexing has applications in spatial, image, text and sequence databases. In spatial database systems such as the Alexandria [12], multi-dimensional index structures provide fast access to spatial/geographical data. In image databases such as QBIC [11], these index structures enable content-based retrieval of image data. Similarly, they enable content-based retrieval in text, audio and sequence databases.

Several multi-dimensional index structures such as R-trees [2, 3, 4] and SS-trees [15] have been developed in the database literature. These structures extend uni-dimensional disk-resident structures such as B-trees to multi-dimensional data. However, they differ from B-trees in the following two aspects. First, index regions in the tree may have to be updated with every *ordinary* insertion or deletion (i.e., an operation that does not cause a split or a merge). We refer to this as the *index region modification* problem in the rest of the paper. Second, node-splits are expensive due to a lack of a total ordering in the multi-dimensional space. In this paper, we propose new concurrency control techniques that specifically address these two factors and achieve high throughput.

First, we propose a new approach for performing *index region modifications* in a multi-dimensional index structure without simultaneously locking nodes from multiple levels. The scheme modifies index regions as it traverses the tree in a top-down manner to insert a data item; we refer to it as *top-down index region modification* (TDIM). This scheme reduces the locking overheads for concurrent insert operations. Experiments on real image data indicate that the TDIM approach achieves more than twice the throughput of existing schemes for insert-only workloads.

Next, we extend TDIM to reduce/eliminate the blocking overhead of query operations due to node-splits caused by concurrent updates. The new scheme first performs splits on local copies of nodes, and then, *copies back* the changes to the concurrent structure. We propose two variants of this scheme: one in which the changes are copied-back by obtaining an exclusive lock on the node for the required duration, and another in which the copy-back is performed atomically by switching between old and updated versions of nodes. The first variant is referred to as *Copy-based Concurrent Update* (CCU), and the second variant as *Copy-based Concurrent Update with Non-blocking Queries* (CCU_NQ). Note that the second variant achieves wait-freedom for queries. However, it requires garbage collection of out-

[*]Work supported by research grants NSF/ARPA/NASA IRI-9411330 and NSF Grant CDA96-01954.

dated versions of nodes. Delete operations are handled analogous to splits and are not treated separately in the paper. Experiments with real image data on the Origin2000, a shared-memory multi-processor machine, indicate that both variants yield 2 times higher throughput than existing schemes.

The rest of the paper is organized as follows. Related work is described in Section 2, TDIM in Section 3, CCU in Section 4, CCU_NQ in Section 5, and experimental comparisons of the proposed schemes in Section 6.

2 Related Work

Extensive research has been done on concurrency control techniques for in 1-dimensional index structures such as B-trees [5, 13]. In B-trees, index regions are not modified except in the case of splits (and merges). Therefore, most techniques try to minimize the locking overheads due to a node-split (or a node-merge) using the link technique of Lehman and Yao [9]. Whenever a node is split into two, a link is maintained between the two resulting nodes. Queries that reach the split node before the split propagates to the parent follow the link if necessary. This ensures that inserts can be performed without excessive locking and without affecting query behavior. The total ordering of data items is implicitly used for ensuring such correct query behavior. Unfortunately, such an ordering does not exist in multiple dimensions. Therefore, for multi-dimensional structures such as R-trees [4], Kornacker and Banks [7] associate sequence numbers with child entries in an index node and use the total ordering on these numbers to ensure correct query behavior. The extra space for these sequence numbers reduces the capacity of tree nodes, which may lead to a degradation in query performance. Consequently, Kornacker et al. [8] propose a new technique that reduces this extra information from one for every child entry in a node to one for the entire node. This extension for concurrency control is proposed in the context of GiST structures. We refer to it as the Concurrent GiST (CGiST) technique. This scheme, unlike the B-link trees, holds multiple locks both during index region modification and during node splits. Holding on to locks on multiple nodes may restrict the concurrency. This is aggravated by the high costs for node splits. In contrast, we propose a new concurrency control technique called CCU that (1) locks at most one node at a time in most cases, and (2) reduces the blocking overhead for a query from the entire duration of a node-split to just the copy-back time. When combined with predicate locking mechanism of [8], CCU ensures serializability of concurrent transactions. In this paper, we only focus on index performance without going into the serializability issues. CCU_NQ, a variant of CCU, completely eliminates the blocking overhead for a query due to other concurrent updates. In contrast to existing constructions for wait-free objects [14], we choose to make only the queries wait-free. Otherwise, the cost of achieving wait-freedom for all operations can become quite high due to excessive copying and wasted parallelism [1]. Our technique for ensuring wait-free queries in multi-dimensional structures is similar to the scheme proposed for binary search trees by Valois [14].

3 Top-Down Index Region Modification (TDIM)

In this section, we describe a new approach for index region modification, called *top-down index region modification* (TDIM). This scheme performs index region modifications by operating on at most one node along the insertion path for most insertions. Queries are never blocked due to a concurrent insert except when the insert causes a node-split.

Before proceeding further, we assume the following model of a multi-dimensional index. The index is a balanced tree structure. Each node of the tree has between m and M child entries, where $m < M$. The value of M is determined by the disk page size. The number of child entries that are currently not empty is indicated by a *count* associated with the node. Each child entry is of the form $[cp, reg]$, where cp points to a child node and reg represents the index region covered by the subtree rooted at cp. The index regions in a node are used to determine if a subtree is relevant for a query (i.e., if it intersects the query region). The index regions are also used to determine the best subtree to insert a new data point. As new insertions occur in a subtree cp, the associated index region reg has to be updated to reflect the area covered by the subtree (index region modification). In addition to this basic structure, we also maintain a node sequence number (*nsn*) with each node and a global sequence number (*global_nsn*) with the entire structure. Together, these sequence numbers can be used to detect node splits due to concurrent insertions as in [8].

Next, we describe how to combine index region modification with a top-down traversal of the tree during an insert operation. An insert operation starts at the root of the tree and identifies a child subtree S in which the new data item fits best. It then updates the corresponding index region for S in the root and then proceeds with the insertion of the data item in the subtree rooted at the child S. Note that the child S may split (the split being detected by comparing the $global\_nsn$ observed in the root and the nsn of S) before the insert operation reaches it. In that case, the insert operation is restarted at the root. This procedure is repeated until the insert reaches a leaf node wherein the corresponding data item is inserted. Unlike the technique of [8], our strategy combines index region modification with tree traversal and avoids locking of nodes from multiple levels of the tree at the same time. Since the modification of index regions is done in a piecemeal fashion without excluding query access, queries are not blocked except during node-splits. Next, we describe the algorithm in detail. Assume that we are inserting a data item with key k and an identifier id in a subtree rooted at n using an observed value, *og_nsn*, of the global sequence number.

```
procedure Insert([KeyType $k$, IdType $id$], NodeType $n$, Int $og\_nsn$)
 begin
   1. If $(og\_nsn < n.nsn)$, i.e., the node $n$ has split after the insert
      operation left the parent of node $n$, then abandon the insertion
      in $n$ and restart it at its parent.
   2. Otherwise,
      (a) If $n$ is a leaf node, then
          i. Obtain a lock that excludes all other updates.
          ii. If $(r.count = M)$, i.e., the node has no room to
              accommodate the new data item, then
              A. Escalate the lock on the node so that even
                 queries are excluded.
              B. Split(n), i.e., split the node $n$ in a bottom-up
                 manner up the tree.
              C. Release all locks acquired during the split
                 except the one on $n$.
              D. De-escalate the lock on $n$ so as to allow
                 queries.
          iii. Add $[k, id]$ to the node $n$ (include it as the first
               unfilled entry).
          iv. Increment $n.count$.
          v. Release the lock on the node $n$.
      (b) Otherwise,
          i. Obtain a shared lock.
          ii. Identify the child entry $[cp, reg]$ that is most ap-
              propriate for including key $k$.
          iii. If $reg$ has to be modified, then release and re-
               obtain the lock so that other updaters are ex-
               cluded. Update the index region $reg$ in a piece-
               meal fashion, i.e., modify the interval of $reg$ in
               each dimension to include key $k$.
          iv. Read global_nsn. Let the value be $og\_nsn_2$.
          v. Insert([k,id], cp, $og\_nsn_2$);
          vi. Release the lock on the node $n$.
 end
```

Node splits propagate in a bottom-up fashion using exclusive locks. The split algorithm is nearly identical to that of CGiST. However, our algorithm avoids two minor errors in the split algorithm of CGiST. These errors are described in [6]. We also note that the performance of CGiST can be enhanced by a simple *split-optimization*: the splitting of a node into two is done before locking its parent as opposed to after locking the parent in CGiST. We use this optimization for CGIST when we compare it with our algorithms. The query algorithm in TDIM is same as that of [8] and is omitted for brevity.

```
procedure Split(NodeType $n$)
 begin
   1. Determine the parent $p$ of $n$.
   2. Create an index entry $[n', N']$ into $n'$ where $N'$ is the index
      region for node $n'$.
   3. Split node $n$ to $n$ and $n'$.
   4. Assign sibling pointer value of n to $n'$. Set sibling pointer of
      $n$ to $n'$.
   5. Lock the parent $p$ in exclusive mode.
   6. Assign nsn of node $n$ to node $n'$.
   7. Increment $global\_nsn$ and install its value as the new $nsn$ for
      node $n$.
   8. If $(p.count = M)$, i.e., the parent does not have room to
      accommodate the split, then Split(p) . Since either one of the
      nodes $p$ and $p'$ (created during split of $p$) is equally likely to
      contain the node $n$, let $p$ be the node that contains a pointer to
      node $n$. The lock on the other node $p'$ is released.
   9. Modify the index region for node $n$ in parent $p$.
   10. Insert $[n', N']$ in the first unfilled entry in parent $p$.
   11. Increment $p.count$.
   12. Release lock on parent $p$.
 end
```

As we see later in experiments, the above approach for index region modification reduces locking overheads for concurrent inserts and improves concurrency for inserts-only workloads. Queries, however, continue to operate as in CGiST and, therefore, may be blocked for the entire duration of concurrent node-splits (which operate using exclusive node locks). In the next section, we describe a comprehensive scheme that extends TDIM so that the blocking overhead for a query due to a concurrent node-split is either completely eliminated or reduced considerably.

4 Copy-based Concurrent Update (CCU)

In this scheme, we reduce the blocking overhead for a query operation. A query may be blocked due to either an index region modification or a node-split. Blocking overhead due to index region modification is eliminated using TDIM. We now address how the blocking overhead due to a node split is reduced. The general idea is to first perform a split on a local copy of a node rather than the shared copy. 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. This process is depicted in Figure 1. Each action of the split process is numbered in order of its execution. By adopting this approach, queries may now be blocked only for the duration of the *copy-back* rather than the entire split process.

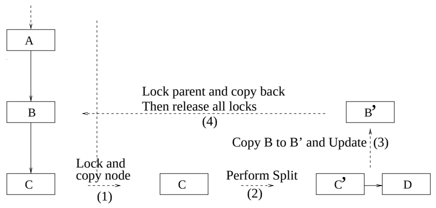

Figure 1. Splitting of a node C in CCU: Numbers in parentheses indicate order of execution

5 Copy-based Concurrent Update with Non-Blocking Queries (CCU_NQ)

In this section, we describe how to make queries completely wait-free, not blocking even for the "copy-back" interval as in a 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, we have to ensure that no operations are left using the older versions. To ensure this, we introduce a logical address (*LA*) for each node. At the time of creation of a node, its logical address is set to the node's physical address. Lock synchronization among different operations is performed on the logical address of the node, which never changes while corresponding node is updated. Whenever a modified copy of a node is ready for incorporation in the shared structure, the logical address of the node is updated to the modified node. The sibling pointer of a node now points to the logical address of the corresponding sibling node rather than the sibling node itself.

Figure 2 illustrates the split of a node C. The steps of this split process are numbered in the order of their execution. Node C is first copied and the split is performed, resulting in nodes C' and D. A new *LA* pointing to D is created. Node B is then copied to B'. We assume for simplicity that B has room to accommodate the split. A new entry is created in B' with the index region of D and a pointer to LA_D. The index region for C' is also updated in B'. The logical addresses LA_B and LA_C are then updated to the nodes B' and C' respectively.

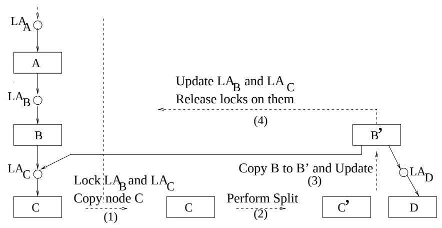

Figure 2. Splitting of a node C in CCU_NQ: Numbers in parentheses indicate order of execution

In order to achieve lock-freedom, queries operate without obtaining locks. As a result, some interleavings of queries with splits caused by insert operations lead to queries returning incomplete results. For example, consider the execution sequence below.

1. The first three operations of a node split of Figure 2 are completed.
2. A query Q accesses node B. It finds that index region of C intersects with its query window. Therefore, Q decides to access the corresponding child pointed to by LA_C.
3. LA_C is switched from C to C'.
4. Query Q follows LA_C to C'.

Query Q only accesses C' and returns incomplete results.

The above anomaly occurs because the query operation has no information of the on-going split of node C and vice-versa. To ensure correct query behavior, the query has to be somehow informed of the split. In our solution, we maintain a split_seq_number (*ssn*) in node B to inform queries of an ongoing split of a child. This number is initially set to ∞. When a child C of node B splits into C' and D, the *ssn* of node B is set to the value of the *nsn* in C'. (Note that the nsn of C' is obtained by incrementing the *global_nsn* as described in procedure *Split*). The logical address LA_C is then switched from C to C' followed by an update of the logical address LA_B. Whenever query Q accesses node C', it first obtains the *ssn* from node B, and then traverses the sibling pointers of C' until it encounters a node whose *nsn* is less than the observed *ssn*. This ensures that a query operation observes all splits of B's children that occurred after the query accessed B. Note that this strategy involves a parent access for every child node access. Since the parent nodes are usually cached, this strategy ensures correct query behavior without compromising on performance.

The above scheme achieves wait-freedom for queries by eliminating blocking of queries due to node-splits. However, a node split that propagates up to k levels in the tree may create $k+1$ garbage nodes. Garbage collection can be performed using the drain technique of [10] and is not addressed in this paper. In the next section, we present some performance results.

6 Experiments

In this section, we compare the performance of our proposed concurrency control techniques with existing ones such as CGiST [8]. We first describe the experimental setup.

6.1 Setup

We experimented with a database of 26K 48-dimensional texture feature vectors obtained from the Corel Image Collection. These feature vectors are indexed using an R*-tree, which was implemented in C under IRIX 6.2. The maximum capacity of the leaf nodes of the tree is 41 and that of the non-leaf nodes is 21. These values correspond to a page size of 8K. The experiments ran on Origin2000 – a shared memory multi-processor machine.

The Origin2000 has 32 200MHz R10000 CPUs with 2GB of main memory and 35Gb of disk space. The shared-memory abstraction uses *directory-based cache coherence* protocol and achieves a read latency of 7ns. Concurrent operations on the index are performed using multiple threads of control. A specified workload of query and insert operations is first generated. The different threads of control, then picked one operation at a time and applied it on the concurrent data structure in the shared memory. For simplicity, we bound every thread of control to a separate processor on the multiprocessor machine. On the Origin2000 machine, this is achieved using the *sysmp* library function provided by IRIX 6.2. Scheduling of multiple threads on the same processor is orthogonal to our techniques and is not addressed here. Lock synchronization between different threads is achieved using a compiler-inlined version of *Compare&Swap*[1].

We experimented with different workloads of insert and query operations. Each workload had 10000 operations. For each operation, we randomly generated a point in the 48-dimensional space. If the operation is an insert, then this point serves as the key for the insert. Otherwise, the point generated serves as the center of a range query. A square query box of 1% of the total domain area is fitted around this center.

For each of the generated workloads, we varied the *disk-access ratio* of the index — the fraction of node accesses that read from or write to the disk. The logical addresses in CCU_NQ are kept in memory due to their low storage requirements. Note that these logical addresses always keep track of the physical addresses of the nodes – be it on the disk, or in memory. As in [13], we mainly worked with two principal configurations for an index: an in-memory index configuration (disk-access ratio=0) and a partial-disk-resident configuration with disk access ratio set to 25%. In what follows, we evaluate the performance of the proposed concurrency control techniques and compare them with CGiST [8] (incorporating the split-optimization described in Section 3). We first describe the experimental results for an inserts-only workload. For this workload, the three schemes that are proposed in this paper — TDIM, CCU and CCU_NQ — are identical. We then present the results for more realistic database workloads consisting of a small fraction of insert operations.

6.2 Results: Inserts-only workload

Figure 3 compares the performance of TDIM with CGiST for an inserts-only workload and an in-memory index configuration. The number of processors (multiprogramming level) is plotted along the x-axis and the throughput (in txn/s) is plotted along the y-axis. We note that TDIM scales better with the number of processors than CGiST does. This is due to less locking overheads in TDIM in comparison to the CGiST; TDIM avoids locking of multiple nodes on an insertion path during index region modification. This results in higher concurrency and better scalability as depicted in the figure.

Next, we compare the performance of TDIM and CGiST when the disk-access ratio is varied. Figure 4 plots the corresponding throughput results for 32 processors. We notice that TDIM yields 2 times higher throughput than CGiST even when the index is on disk,

6.3 Results: Insert-Query workloads

We next consider realistic database workloads which contain a high percentage of queries and a small percentage of updates. For these workloads, we expect that CCU and CCU_NQ — the extensions of TDIM that

[1] On the MIPS II architectures, *load-linked* and *store-conditional* are the underlying primitive operations for *Compare&Swap*.

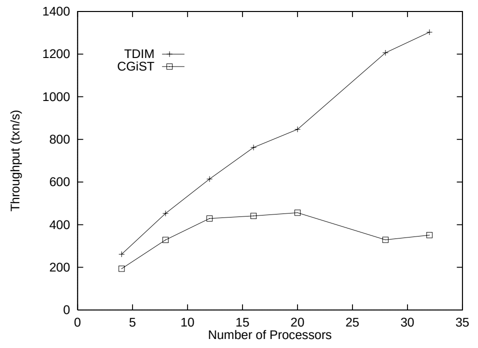

Figure 3. Scalability of TDIM and CGiST with number of processors: Inserts-only workload, in-memory index

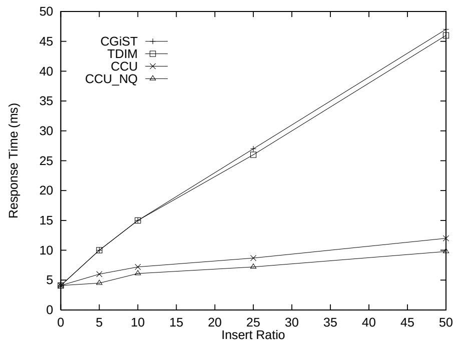

Figure 5. Query Response Times for varying insert-ratios: 32 processors, in-memory index

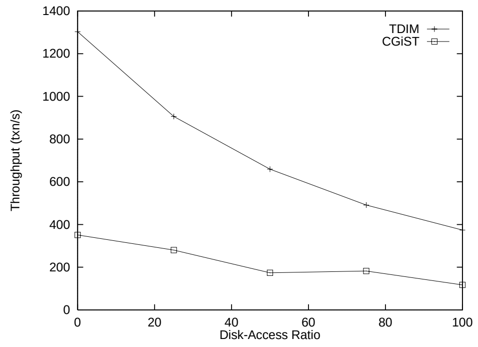

Figure 4. Comparison of TDIM and CGiST when disk-access ratio is varied: 32 processors, inserts-only workload

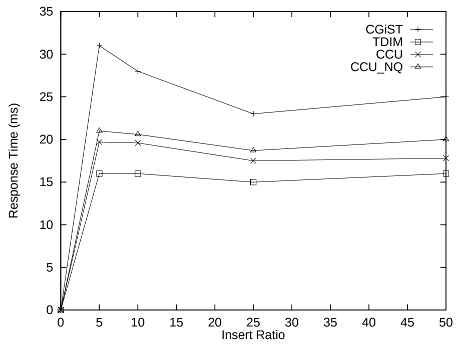

Figure 6. Insert Response Times for varying insert-ratios: 32 processors, in-memory index

are designed to reduce/eliminate blocking overheads for queries — would achieve high throughput. Next, we present results validating this hypothesis.

Figures 5 and 6 plot the query and insert response times of the four concurrency control schemes for different workloads. The percentage of inserts (insert-ratio) in a workload is varied from 0 to 100. Figure 5 plots the query response times and Figure 5 the insert response times. We notice that the best query response times are achieved by copy-based schemes (CCU and CCU_NQ) and the best insert response times by TDIM. TDIM does not minimize query overheads and therefore has nearly same query times as CGiST. These results show that the copy-based schemes combine fast query responses with good insert times.

The overall throughput of these schemes is plotted in Figure 7. We observe that the copy-based schemes achieve the best throughput for all insertion ratios. Since they are extensions of TDIM, their performance is identical to TDIM for an inserts-only workload.

Next, we compare the scalability of the copy-based schemes and CGiST with the number of processors. We vary the number of processors for different workloads (5%, and 25% inserts) and observe the overall throughput of the system. Figure 8 plots these results for a 5%, and 25% (insert-ratio) workloads when the index is in memory. We observe that the difference between copy-based schemes and CGiST increases as we increase the insertion ratio. The copy-based techniques, which are extensions of TDIM, reduce locking overheads for queries as well as updates. As the percentage of inserts is increased, the number of updates that conflict with other queries and updates also increase. This contention increases with an increase in the number of processors. Consequently, the CGiST technique performs poorly both as we increase the percentage of inserts as well as the number of processors. In contrast, the copy-based schemes achieve nearly linear speedups as we increase the number of processors. Of the two copy-based variants, the non-blocking version achieves the maximum throughput because it achieves wait-freedom for queries. Similar results are also obtained for scalability comparisons when the index structures are partially resident on disk. These results are described in detail in [6].

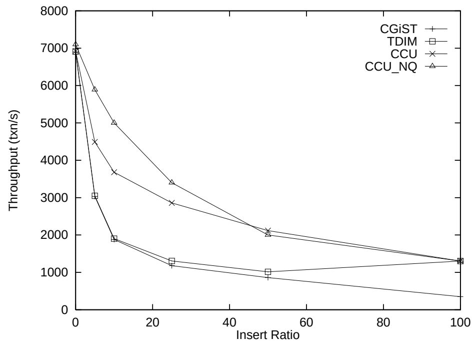

Figure 7. Throughput for varying insert-ratios: 32 processors, in-memory index

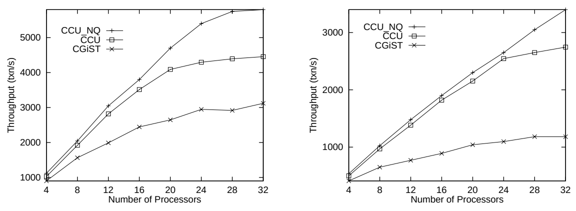

Figure 8. In-memory index throughput for varying number of processors: (a) 5%-insert and (b) 25%-insert workloads

In summary, the above results indicate that copy-based concurrency control schemes achieve up to 50% improvements in query and update response times over existing techniques for realistic database workloads (5% inserts). These schemes also scale well with the number of processors. The difference in performance of the two variants of the copy-based concurrent update scheme is small. Due to garbage collection issues and the cost of indirection [14], we recommend the use of the blocking variant of the copy-based concurrency control (CCU) for achieving high concurrency in multi-dimensional index structures.

7 Conclusions

In this paper, we examined concurrency control in multi-dimensional index structures. We proposed new techniques to reduce locking overheads of both query and update operations. In experiments on real image data using realistic database workloads, the proposed concurrency control techniques achieve up to 2 times higher throughput than existing ones.

References

[1] J. H. Anderson and M. Moir. Universal constructions for large objects. In *WDAG '95*, volume 972 of *LNCS*, pages 168–182, 1995.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, May 23-25 1990.

[3] S. Berchtold, D. A. Keim, and H. P. Kreigel. The X-tree: An index structure for high dimensional data. *Proceedings of the Int. Conf. on Very Large Data Bases*, 1996.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[5] T. Johnson and D. Shasha. The performance of concurrent B-tree algorithms. *Proc. ACM Symp. on Transactions of Database Systems*, 18(1), March 1993.

[6] K. V. Ravi Kanth, David F. Serena, and Ambuj K. Singh. Improved concurrency control techniques for multi-dimensional index structures. Technical report, Univ. of California at Santa Barbara, December 1997. http://www.cs.ucsb.edu/∼kravi/conc.ps.

[7] M. Kornacker and D. Banks. High concurrency locking for R-trees. *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 134–145, September 1996.

[8] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency control and recovery in GiST. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1997.

[9] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *Proc. ACM Symp. on Transactions of Database Systems*, 6(4):650–670, December 1981.

[10] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multi-action transactions operating on B-tree indexes. *Proceedings of the Int. Conf. on Very Large Data Bases*, August 1990.

[11] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker. The QBIC project: Querying images by content using color, texture and shape. In *Proc. of the SPIE Conf. 1908 on Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, Feb. 1993.

[12] T. R. Smith and J. Frew. Alexandria Digital Library. *Communications of the ACM*, 38(4):61–62, April 1995.

[13] V. Srinivasan and M. Carey. Performance of b-tree concurrency control algorithms. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 416–425, 1991.

[14] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[15] D. White and R. Jain. Similarity indexing with the SS-tree. *Proc. Int. Conf. on Data Engineering*, pages 516–523, 1996.