MLIP: A Concurrent Approach for Clipping Indexing

Majoju Ravinder
M.Tech (CSE) Student, Dept of CSE
S. R. Engineering College
Warangal, India
ravinder1204@gmail.com

R.Vijay Prakash, Assoc. Prof.
Dept. of Computer Science Engineering
S. R. Engineering College
Warangal, India
vijprak@hotmail.com

***Abstract*— Multidimensional databases are beginning to be used in a wide range of applications. To meet this fast-growing demand, the R-tree family is being applied to support fast access to multidimensional data, for which the R+-tree exhibits outstanding search performance. In order to support efficient concurrent access in multi-user environments, concurrency control mechanisms for multidimensional indexing have been proposed. However, these mechanisms cannot be directly applied to the R+-tree because an object in the R+-tree may be indexed in multiple leaves. This paper proposes a concurrency control protocol for R-tree variants with object clipping, namely, Micro level Locking for clIPping indexing (MLIP). MLIP is the first concurrency control approach specifically designed for the R+-tree and its variants, and it supports efficient concurrent operations with serializable isolation, consistency, and deadlock-free. Experimental tests on both real and synthetic data sets validated the effectiveness and efficiency of the proposed concurrent access framework.**

Keywords-Micro level locking; serializable; concurrent operations; clipping Indexing

I. INTRODUCTION

As the fast-growing demand for multi-dimensional databases applications, the development of efficient access methods for multidimensional data has become a crucial aspect of database research. Many indexing structures (e.g., the R-tree family, Generalized Search Trees (GiSTs), grid files, and z-ordering) have been proposed to support fast access to multidimensional data in relational databases. An important issue related to indexing, concurrency control methods that support concurrent access in traditional databases are no longer adequate for today's multidimensional indexing structures due to the lack of a total order among key values. In order to support concurrency control in R-tree structures, several approaches have been proposed, such as Partial Locking Coupling (PLC), and granular locking approaches for R-trees and GiSTs. In multidimensional indexing trees, the overlapping of nodes will tend to degrade query performance, as one single point query may need to traverse multiple branches of the tree if the query point is in an overlapped area.

This paper proposes a concurrency control protocol for R-trees with object clipping, Micro level Locking for clipping indexing (MLIP), to provide phantom update protection for the R+-tree and its variants. We also introduce the ZR+-trees, which resolve the limitations of the original R+-tree by eliminating the overlaps of leaf nodes. MLIP, together with the ZR+-tree, constitutes an efficient and sound concurrent access model for multidimensional databases.

The major contributions are as follows:

- The concurrency control protocol, MLIP, provides serializable isolation, consistency, and deadlock-free operations for indexing trees with object clipping.
- The proposed multidimensional access method, ZR+-tree, utilizes object clipping, optimized insertion, and reinsert approaches to refine the indexing structure and remove limitations in constructing and updating R+-trees.
- MLIP and the ZR+-tree enable an efficient and sound concurrent framework to be constructed for multidimensional databases.

- A set of extensive experiments on both real and synthetic data sets validated the efficiency and effectiveness of the proposed concurrent access framework.

II. Applying Concurrency Control on R+-Trees

Several efficient key value locking protocols to provide phantom update protection in B-trees have been proposed [3], [17], [18]. However, they cannot be directly applied to multidimensional index structures such as R-trees, because for multidimensional data, a total order of the key values on which these protocols are based is undefined. Granular locking protocols such as GL/R-tree [4], [5] for multidimensional indices have been proposed, but none can be directly applied to the R+-tree.

An example will show why the original GL/R-tree is not sufficient to provide phantom update protection for the R+-tree. The GL/R-tree defines two types of lockable granules: leaf granules that correspond to the MBR for each leaf node and external granules that are defined as ext (internal node) = (MBR for the internal node) — (MBRs for each of its children). In Fig. 1, assuming *A* and *B* are leaf nodes, the search window *WS* requires shared locks to be placed on the lockable granules *A*, whereas the update window *WU* requires exclusive locks to be placed on *B*. However, as in an R+-tree, the object *D* is shared by both leaf nodes and both locks only affect their own granules. In this case, the GL/R-tree protocol does not provide sufficient phantom update protection for the object *D*. One possible solution to this problem would be to lock objects rather than leaf granules. In this way, the objects' MBRs can be viewed as leaf granules, and the external granules would be defined similarly for leaf nodes.

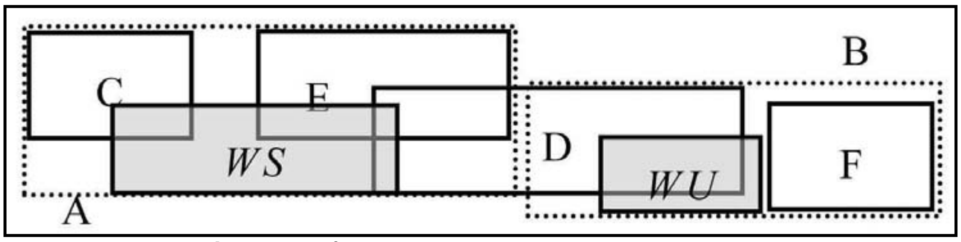


Fig. 1. Example operations for GL/R-tree on an R+-tree.

Although this solution solves the above problem for deletions (and updates), the object-level locking substantially increases the number of locks. For example, if a search window were to return 10,000 objects, this would require 10,000 object-level locks to be placed for the duration of the search and then released at the time of commitment. Using coarse leaf granules, as proposed in the GL/R-tree, and assuming 100 maximum entries per node and an average fill factor of 0.5, only 200 such locks would need to be requested. Therefore, for applications where selection is the predominant operation, locking at the object level may not be a desirable solution, and a new locking protocol is therefore required to provide phantom update protection efficiently for indexing trees with object clipping.

III. Definition of MLIP and ZR+-Tree

Before proceeding to the details of the proposed concurrent access framework, we first define the notations that will be used throughout this paper.

Terms and Notations

The presence of a standard lock manager [15] is presumed to support conditional and unconditional lock requests, as well as instant, manual, and commit lock durations in MLIP. A conditional lock request means that the requester will not wait if the lock cannot be granted immediately; an unconditional lock request means that the requester is willing to wait until the lock becomes grantable. Instant duration locks merely test whether a lock is grantable, and no lock is actually placed. Manual duration locks can be explicitly released before the transaction is completed. If they are not released explicitly, they are automatically released at the time of commit or rollback. Commit duration locks are automatically released when the transaction ends. Conventionally, five types of locks, namely, S (shared locks), X (exclusive locks), IX (Intention to set X locks), IS (Intention to set S locks), and SIX (Union of S and IX locks) [6] are used. In the proposed protocol, only S and X locks are used to support concurrent operations with relatively simple maintenance processes.

The lock manager in MLIP is presumed to support the acquisition of multiple locks as an atomic operation. If this is not the case, such a procedure can be conveniently implemented by acquiring the first lock in a list unconditionally and all subsequent locks conditionally, with the procedure releasing all the acquired locks and restarting if any of the conditional locks cannot be acquired. Furthermore, a transaction can place any number of locks on the same granule as long as they are compatible. The lock manager will place separate locks for each granule, and each lock will be distinct even if the lock modes are the same. When releasing manual duration locks, both the lock granule and lock mode must be specified.

TABLE 1 ZR+-Tree Node Attributes

Term	Description
capacity	maximum number of entries in the node
entries	number of entries in the node
mbr	minimum bounding rectangle of the node
level	level of the node in the tree
child$_i$	i$_{th}$ child of the node
rect$_i$	MBR of the i$_{th}$ child of the node
isLeaf	true for a leaf node

The terms used to describe the ZR+-tree structure are listed in Table 1.Suppose T denotes a ZR+-tree, then T:root refers to the root node of this tree. For each node P in T, P:isLeaf indicates whether the node P is a leaf node or not, P:level gives the level of P in T,P: entries denotes the current number of entries in the node ,and P: capacity is the maximum number of entries the node P can hold. P:mbr gives the MBR for the node P and is defined as an empty rectangle when P is NIL. For internal nodes, P:child$_i$ is an entry pointing to a node, which is P's ith child, and P:rect$_i$ gives the MBR of the ith entry. For leaf nodes, P:child$_i$ gives the object pointed to by the ith entry, and P:rect$_i$ refers to the MBR of this entry. For each rectangle R, R:l denotes the lower left corner and R:h denotes the upper right corner.

Similar to the R+-tree, the ZR+-tree is height balanced, so for each P in T, where P:isLeaf is true, P:level is the same. This also implies that if P is an internal node, then for all P:child$_i$, P:child$_i$:isLeaf is false, or for all P:child$_i$, P:child$_i$:isLeaf is true. As data objects in a ZR+-tree may be clipped, for leaf nodes, P:rect$_i$ may only indicate part of the MBR of a data object. Therefore, an object can be exclusively covered by multiple nodes. Furthermore, P:mbr must cover all the P:rect$_i$, regardless of whether P:child$_i$ is an internal node or not.

R+-Tree and ZR+-Tree

- R+-trees can be viewed as an extension of K-D-B-trees [22] to cover rectangles in addition to points. The original R+-tree has the following properties [23]:

- A leaf node has one or more entries of the form (oid; RECT), where oid is an object identifier, and RECT is the Minimum Bounding Rectangle (MBR) of a data object.
- An internal node has one or more entries of the form (p;RECT), where p points to an R+-tree leaf or internal node R, such that if R is an internal node, then RECT is the MBR of all the (p$_i$;RECT$_i$) in R. However, if R is a leaf node, for each (oid$_i$; RECT$_i$) in R, RECT$_i$ does not need to be completely enclosed by RECT; each RECT$_i$ simply needs to overlap with RECT.
- For any two entries (p$_1$; RECT$_1$) and (p$_2$; RECT$_2$) in an internal node R, the overlap between RECT$_1$ and RECT$_2$ is zero.
- The root has at least two children unless it is a leaf.
- All leaves are at the same level.

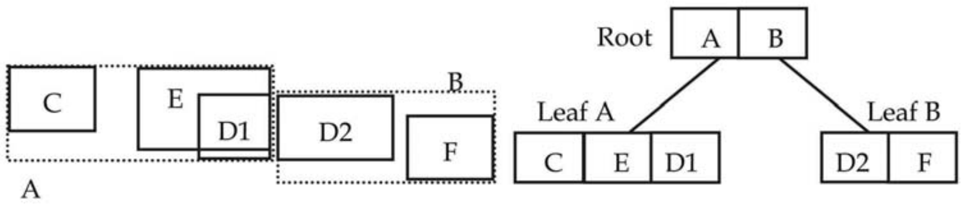


Fig. 2. An example of ZR+-tree for the data in Fig. 1

As the proposed tree structure eliminates overlaps even among entries in different leaf nodes, it is named the Zero-overlap R+-tree (ZR+-tree). The essential idea behind the ZR+-tree is to logically clip the data objects to fit them into the exclusive leaf nodes.

There are two fundamental differences between the clipping techniques applied in the ZR+-tree and the R+-tree:

- From the definition of the ZR+-tree, object clipping in the ZR+-tree must differentiate the MBRs of the segmented objects in leaf nodes (e.g., MBRs of D_1 and D_2 in Fig. 2), while the clipping in the R+-tree retains the original MBRs (e.g., MBRs of the two Ds in the leaf node A and leaf node B).

- In the ZR+-tree, each entry in a leaf node is a list of segmented objects that share the same MBR, while each leaf node entry in the R+-tree contains exactly one object.

Micro level Locks

Each leaf node in the ZR+-tree is defined as a micro level lockable granule. We also define an external lockable granule for each ZR+-tree node as the difference between the MBR of the node and the union of the MBRs of its children. In order to reduce the overhead associated with lock maintenance, objects are not individually lockable. The clip array introduced as an auxiliary structure to store the object clipping information does not need to be locked because the locking strategies on leaf nodes ensure the serializability of access for the same object, and updating one object will not affect the other objects.

Thus, in the case of the indexing tree in Fig. 1, the leaf nodes *A* and *B, ext(A), ext(B),* and *extiroot)* are defined as lockable granules. *ext(A)* covers the region *A.mbr — (C.mbrU Di.mbrU E.mbr),* and *extiroot)* covers the region *MBTL(A.mbr* U *B.mbr) — (A.mbr* U *B.mbr)).* The above lockable granules cover the entire MBR of the tree root. However, all of these lockable granules do not fully cover any search windows that are partially or fully located outside the MBR of the root. One option is to define *ext(T)* as a lockable granule that covers all such INDEXING space.

IV. Operations with MLIP on ZR+-Tree

To support concurrent spatial operations on the R+-tree and its variants, a granular locking-based concurrency control approach, MLIP, that considers the handling of clipped rectangles is proposed. The approach is designed to meet the following requirements:
1. The following concurrent operations should be supported.
Select for a given search window. This is presumed to be the most frequent operation. This operation could result in the selection of a large number of objects, though this may be only a fraction of the total number of objects. Hence, it is desirable to have as few locks as possible that must be requested and released for this operation.
Insert a given object. Having redefined the properties of the R+-tree with clipped objects, a new algorithm must be provided for insertion in the ZR+-tree.
Delete objects intersected with a search window. Since an object in the ZR+-tree may be clipped and the search window might not select all the fragments of a given object, the algorithm is required to delete all fragments of the selected objects in order to maintain consistency.
2. The locking protocol should ensure serializable isolation for transactions, thus allowing any combination of the above operations performed.
3. The locking protocol should ensure consistency of the ZR+-tree under structure modifications. When ZR+-tree nodes are merged or split in cases of underflow or overflow, the occasionally inconsistent state should not lead to invalid results.
4. The proposed locking protocol should not lead to additional deadlocks.
Details of the algorithms are provided in the following sections with formal algorithm descriptions.

1)Select:

Algorithm Select(W, T)
Input: search window *W*, ZR+-tree *T* Output: set of objectelD *O*
O := {}; P := T.root

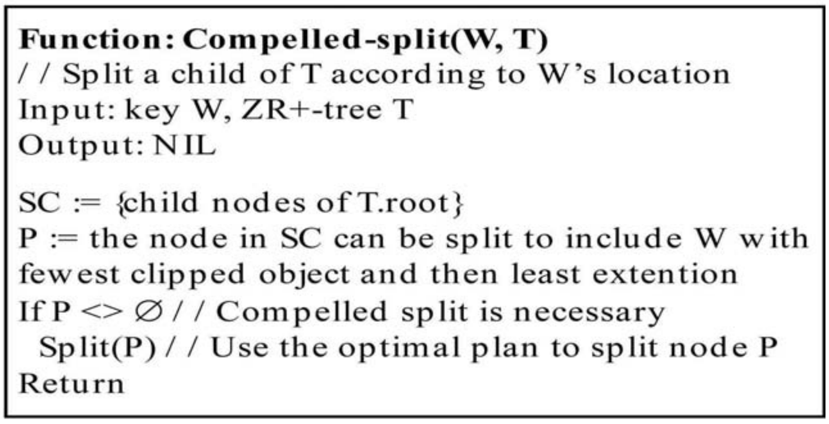
```
If (P is NIL) or (not(P.mbr n W))
return O If W n P.mbroP.mbr / / Root does not cover W
Lock(ext(T), S, Commit) / / Lock external of tree
Lock(ext(P), S, Manual) / / Lock the root Stack L := {( P.mbr n W, P)}
Loop until L is 0 (R, P) := L.pop For each i in P.rect IfP.rectnRThen If P.isLeaf Then
0:=Ou P.chikL / / Add the objects that are not in results Unlock(P, S) Else
If P.chikLisLeafThen
LockCP.child,, S, Commit) Else
LockCextCP.child;), S, Manual) L.push( {(P.rect n R, P.chikt)}) / / Put the child of P in stack
 R := R - P.rect If (not P.isLeaf) and (R = 0) Unlock(ext(P),S) / / Release S Lock on ext(P) if not overlaps R Return
```

2)Insert:

Algorithm Insert(W, O, T)

```
Input: key W, object O, ZR+-tree T, queue of X locks to request M Output: NIL
L := {}; P := T.root; M := {}; $S_2$ := {}
If W H P.mbroP.mbr //root does not cover W
M.enqueue({ext(T), X, Commit}) L.enqueue({P, W}) Loop until L is 0 (P, R) := L.dequeue If P.isLeaf $S_2$ := $S_2$ + {P, R} M.enqueue({P, X, Commit}) Else If P.mbr covers R and !(3 i, P.rect covers R) M.enqueue({ext(P), X, Commit})
SC := minExtend(W, P) //Choose list SC in P to extend to include W with minimum cost and update MBRs (Algorithm 3) L.enqueue({each node in S and its extended MBR}) break Else
n := P.childi | P.childi covers R L.enqueue(n, R)
If LockAll(M) //Request all the X locks and check version For every pair (P, R) in $S_2$ P.child(P.entries) := O P.rect(P.entries++) := R If R<>W //The object is clipped
StoreClipArray(0, R, P) //Store object in clip array
If P.entries > P.capacity //Overflow Split(P) //If splits propagate to a node not in M then add the node to M and restart from LockAll Else
Insert(W, O, T) //Restart insert operation Return
```

3)Window Split:



Function: Compelled-split(W, T)

```
/ / Split a child of T according to W's location
Input: key W, ZR+-tree T
Output: NIL

SC := {child nodes of T.root}
P := the node in SC can be split to include W with fewest clipped object and then least extention
If P <> ∅ / / Compelled split is necessary
  Split(P) / / Use the optimal plan to split node P
Return
```

4)Delete:

Algorithm Delete(W, T)

```
Input: deletion window W, ZR+-tree T
Output: NIL

O := {}; O1 := {}; P := T.root; V := {}; M := Clip Array; Stack L := {( P.mbr ∩
W, P)}
If (P is NIL) or (not(P.mbr ∩ W))
  Return
// Record required locks
If W ∩ P.mbr<>P.mbr  //Root does not cover W
  V.enqueue({ext(T), S, Commit}) //Lock external of tree
Loop until L is ∅  //Traverse the indexing tree
  (R, P) := L.pop
  For each i in P.rect_i
    If P.rect_i ∩ R Then
      If P.isLeaf Then
        O := O ∪ P.child_i  //Add the objects that are not yet in results
        O1 := O1∪ {leaf nodes in M that covers P.child_i} //Add leaf
                                     nodes from the object link in clip array
      Else
        If P.child_i.isLeaf Then
          V.enqueue({P.child_i, X, Commit})
        L.push({(P.rect_i ∩ R, P.child_i)})  //Put the child of P in stack
      R := R - P.rect_i
  If (not P.isLeaf) and (R ≠ ∅)
    V.enqueue({ext(P), S, Commit}) //S Lock on ext(P) if it overlaps R
For every node n in O1 //Lock all the leaf nodes that cover the objects to
                         be deleted
  V.enqueue({n, X, Commit})
For every internal node n in T whose MBR will shrink or be removed
after deleting set O
  V.enqueue({ext(n), X, Commit})
// Request locks and delete object, or re-do if conflict occurs
If LockAll(V) //Request all the locks and check version
  For each object n in O
    Delete n in the leaf nodes in O1; Delete n in M
  For each underflow leaf node n in O1
    Merge(n) //Propagate if necessary
Else
  Delete(W, T) //Restart the delete operation
Return
```

V. QUERY PERFORMANCE

The performance for concurrent query execution was evaluated both for the R-tree with granular locking and the ZR+-tree with the proposed MLIP protocol. In order to compare these two multidimensional access frameworks, two parameters, namely, concurrency level and write probability, were applied to simulate different application
environments on the three data sets. Here, concurrency level is defined as the number of queries to be executed simultaneously, and write probability describes how many queries in the whole simultaneous query set are update
queries. The execution time measured in milliseconds was used to represent the throughput of each of the approaches.

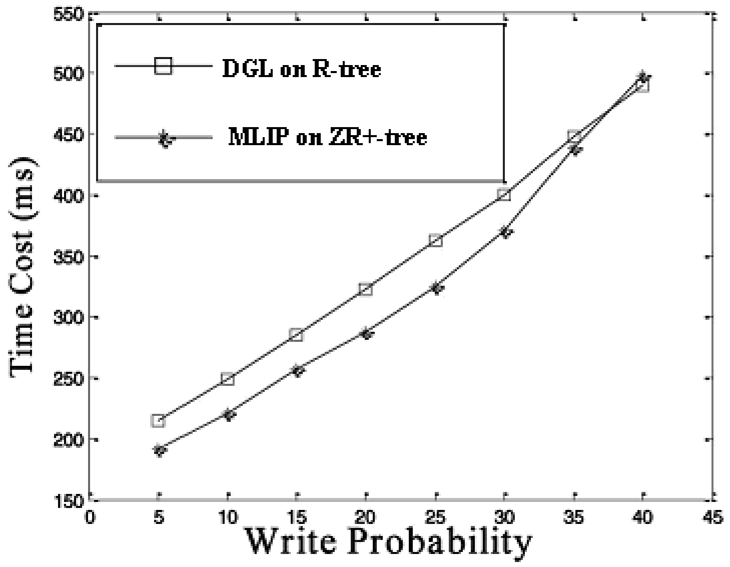


Fig. 3. Execution time for different concurrency levels.

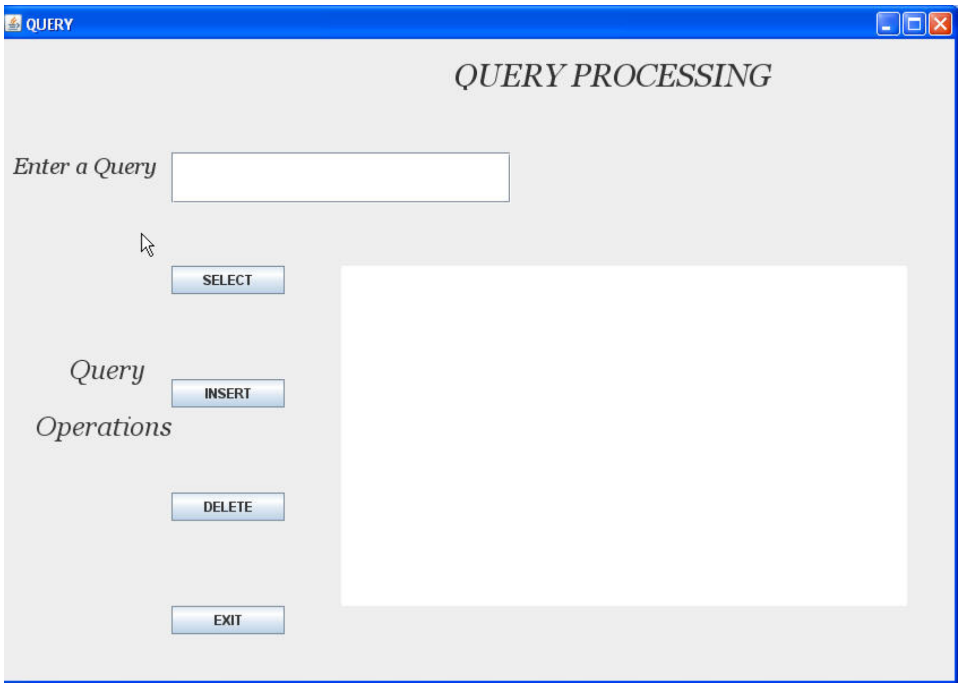


Fig. 4. Execution of MLIP.

The execution time costs for the three data sets with a fixed concurrency level and changing write probabilities when the query range is 1 percent of the data space. The concurrency level was fixed at two levels 30 and 50 as representative levels, while the write probability varied from 5 percent to 40 percent. The y-axis in these figures shows the time taken to finish these concurrent operations, and the x-axis indicates the portions of update operations in all the concurrent operations in terms of percentages. Both approaches degrade the throughput when the write probability increases. Comparing the performance from the different write probabilities, MLIP on the ZR+-tree performs better than granular locking on the R-tree when the write probability is small. When the write probability increases, the throughput of the concurrency control on the R-tree comes close to and exceeds that of the ZR+-tree. Specifically, when the concurrency level is 30, the throughput of the ZR+- tree is better with a write probability lower than 30 percent in real data sets. When the concurrency level is raised to 50, the particularly significant for evenly distributed data sets compared to DGL on the R-tree.

VI. CONCLUSION

This paper proposes a new concurrency control protocol, MLIP, with an improved spatial indexing approach, the ZR+-tree. MLIP is the first concurrency control mechanism designed specifically for the R+-tree and its variants. It assures serializable isolation, consistency, and deadlock free for indexing trees with object clipping. The ZR+-tree segments the objects to ensure every fragment is fully covered by a leaf node. This clipping-object design provides a better indexing structure. Furthermore, several structural limitations of the R+-tree are overcome in the ZR+-tree by the use of a nonoverlap clipping and a clustering-based reinsert procedure. Experiments on tree construction, query, and concurrent execution were conducted on both real and synthetic data sets, and the results validated the soundness and comprehensive nature of the new design. In particular, the MLIP and the ZR+-tree excel at range queries in search-dominant applications.

VII. REFERENCES

[1] M. Abdelguerfi, J. Givaudan, K. Shaw, and R. Ladner, "The 2-3TR-Tree, a Trajectory-Oriented Index Structure for Fully Evolving Valid-Time Spatio-Temporal Datasets," *Proc. 10th ACM Int'l Symp. Advances in Geographic Information System (ACMGIS '02),* pp. 29-34, 2002.

[2] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD '90,* pp. 322-331, 1990.

[3] Biliris, "Operation Specific Locking in B-trees," Proc. Sixth Int'l Conf. Principles of Database Systems (PODS '87), pp. 159-169, 1987.

[4] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-Trees," *Proc. 14th IEEE Int'l Conf. Data Eng. (ICDE '98),* pp. 446-454, 1998.

[5] K. Chakrabarti and S. Mehrotra, "Efficient Concurrency Control in Multi-Dimensional Access Methods," *Proc. ACM SIGMOD '99,* pp. 25-36, 1999.

[6] J.K. Chen, Y.F. Huang, and Y.H. Chin, "A Study of Concurrent Operations on R-Trees," *Information Sciences,* vol. 98, nos. 1-4, pp. 263-300, May 1997.

[7] V. Gaede and O. Gunther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, June 1998.

[8] D. Greene, "An Implementation and Performance Analysis of Spatial Data Access Methods," *Proc. Fifth IEEE Int'l Conf. Data Eng. (ICDE '89),* pp. 606-615, 1989.

[9] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," *Proc. ACM SIGMOD '98,* pp. 73-84 1998.

[10] Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD '84,* pp. 47-57, 1984.

[11] J. Hellerstein, J. Naughton, and A. Pfeffer, "Generalized Search Trees in Database Systems," *Proc. 21st Int'l Conf. Very Large Data Bases (VLDB '95),* pp. 562-673, 1995.

[12] E.G. Hoel and H. Samet, "A Qualitative Comparison Study of Data Structures for Large Line Segment Databases," *Proc. ACM SIGMOD '92,* pp. 205-214, 1992.

[13] K.V.R. Kanth, D. Serena, and A.K. Singh, "Improved Concurrency Control Techniques for Multi-Dimensional Index Structures," *Proc. Ninth Symp. Parallel and Distributed Processing (SPDP '98),* pp. 580-586, 1998.

[14] M. Kornacker and D. Banks, "High-Concurrency Locking in R-Trees," *Proc. 21st Int'l Conf. Very Large Data Bases (VLDB '95),* pp. 134-145, 1995.

[15] M. Kornacker, C. Mohan, and J. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," *Proc. ACM SIGMOD '97,* pp. 62-72, 1997.

[16] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM Trans. Database Systems,* vol. 6, no. 4, pp. 650-670, Dec. 1981.

[17] D. Lomet, "Key Range Locking Strategies for Improved Concurrency," *Proc. 19th Int'l Conf. Very Large Data Bases (VLDB '93),* pp. 655-664, 1993.

[18] Mohan and F. Levin, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," Proc. ACM SIGMOD '92, pp. 371-380, 1992.