Concurrency Control Protocol for Multidimensional databases using Clipping & Indexing

Radhika Kyadagiri  
Associate Professor, Dept of CSE  
JPNCE – Mahabubnagar, A.P.

Prof. D. Jamuna  
Professor & HOD, Dept of CSE  
JPNCE – Mahabubnagar, A.P.

Sharath Chandra L  
M Tech. Research Scholar  
JPNCE – Mahabubnagar, A.P.

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 multiuser 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, Granular Locking for clipping indexing (GLIP). GLIP is the first concurrency control approach specifically designed for the R+-tree and its variants, and it supports efficient concurrent operations with serializability, 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
Concurrency, Indexing methods, Spatial databases.

1. INTRODUCTION
In recent years, multidimensional databases have begun to be used for a wide range of applications, including geographical information systems (GIS), computer-aided design (CAD), and computer-aided medical diagnosis applications. As a result of this fast-growing demand for these emerging 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 [10] family, Generalized Search Trees (GiSTs) [11], grid files [20], and z-ordering [21]) have been proposed to support fast access to multidimensional data in relational databases. Among these indexing structures, the R-tree family has attracted significant attention as the tree structure is regarded as one of the most prominent indexing structures for relational databases. On the other hand, as 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) [25], and granular locking approaches for R-trees and GiSTs [4], [5].

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. The R+-tree [23] has been proposed based on modifications of the R-tree to avoid overlaps between regions at the same level, using object clipping to ensure that point queries can follow only one single search path. The R+-tree exhibits better search performance, making it suitable for applications where search is the predominant operation. For these applications, even a marginal improvement in search operations can result in significant benefits. Thus, the increased cost of updates is an acceptable price to pay.

However, the R+-tree is not suitable for use with current concurrency control methods because a single object in the R+-tree may be indexed in different leaf nodes. Special considerations are needed to support concurrent queries on R+-trees, while as far as we know, there is no concurrency control approach that specifically supports R+-trees. Furthermore, there are some limitations in the design of the R+-tree, such as its failure to insert and split nodes in some complex overlap or intersection cases [7].

This paper proposes a concurrency control protocol for R+-trees with object clipping, Granular Locking for clipping indexing (GLIP), to provide phantom update protection for the R+-tree and its variants. We also introduce the Zero overlap R+-tree (ZR+-tree), which resolves the limitations of the original R+-tree by eliminating the overlaps of leaf nodes. GLIP, 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, GLIP, 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.
- GLIP 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.

2.1 EXISTING SYSTEM
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. The R+-tree has been proposed based on modifications of the R-tree to avoid overlaps between regions at the same...
level, using object clipping to ensure that point queries can follow only one single search path. The R+-tree exhibits better search performance, making it suitable for applications where search is the predominant operation. For these applications, even a marginal improvement in search operations can result in significant benefits. Thus, the increased cost of updates is an acceptable price to pay.

2.2 PROPOSED SYSTEM

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

3.1 OPERATIONS WITH GLIP ON ZR+-TREE

To support concurrent spatial operations on the R+-tree and its variants, a granular locking-based concurrency control approach, GLIP 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.

   a. 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. Select 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.

   b. 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.

3.2.1 Algorithm for Select operation

The select operation, shown in Algorithm 1, returns all object ids given a search window W. It is necessary to place locks on all granules that overlap with the search window in order to prevent writers from inserting into or deleting from these granules until the transaction is completed.

```
Algorithm Select(W, T)
Input: search window W, R+-tree T
Output: set of object ID O
O := []; P := T.root
If(P is NIL) or (not(P.mbr ∩ W))
  return O
If W ∩ P.mbr := P.mbr / / Root does not cover W
  Lock(extend(T), S, Commit) / / Lock external of tree
Lock(extend(P, S, Manual)); / / Lock the root
Stack L := [P.mbr ∩ W, P]]
// Traverse the indexing tree and lock/unlock the visited nodes
Loop until L is ∅
  (R, P) := L.pop
  For each i in Precs,
    If Precs ∩ R Then
      If P.isLeaf Then
        O := O ∪ P.child // Add the objects that are not in results
        Unlock(P, S)
      Else
        If P.child.isLeaf Then
          Lock(P.child, S, Commit)
        Else
          Lock(extend(P.child), S, Manual)
          L.push((R.prece, ∩ R, P.child)); / / Put the child of P in stack
          R := R - Precs.
        If (not P.inLeaf) and (R = ∅)
          Unlock(extend(P, S)); / / Release S Lock on extend(P) if not overlaps R
        Return O
```

3.2.2 Algorithm for Insert operation

Compared with R+-trees, the insert operation for ZR+-trees (Algorithm 2) takes into account additional considerations. To illustrate the insert operation, we name the MBR of the object to be inserted as W. First, consider all the fragments of W that do not overlap with any other objects’ MBRs. These fragments must be inserted into the leaf nodes of the tree. However, the fragments that intersect with existing objects’ MBRs may result in clipping these MBRs if they are not equal. Considering the objects in Fig. 5b, if P is inserted after O, P will need to be fragmented into three rectangles ∅P1; P2; P3 before it can be inserted. Similarly, if Q were to be inserted after P, the same clipping would also be required. The number of fragments that an insertion will create is a function of the gaps in the objects.
3.2.3 Algorithm for Delete Operation

The delete operation, as shown in Algorithm 6, works in a similar way to the insert operation. For a delete operation, since the same object may be fragmented and stored in multiple leaf nodes, it is necessary to assure that all the fragments of an object are deleted. A deletion window W may not select all the object fragments; deleting only the fragments that intersect with the deletion window can thus leave residual fragments. A clip array is maintained to store object id and pointers to the leaf nodes that store the fragments of the object. First, all ids of the objects that intersect with the deletion window are selected. The corresponding elements in the clip array are then read to locate all the fragments in other leaf nodes, after which the object deletion is performed. However, it is inefficient to read the clip array for each selected object, because in many cases, the object MBR may not be fragmented in the tree at all. An optimized strategy is to store a bit to indicate whether the MBR in the leaf node is the complete object MBR. The algorithm thus needs to read the clip array only when the search window selects a fragmented MBR.

---

Algorithm Insert(W, O, T)

Input: key W, object O, ZR+t-tree T, queue of X locks to request M
Output: NIL

L := (); P := T.root; M := (); S := ()
// Record required locks

if W \cap P.mbr = P.mbr // root does not cover W
  M.enqueue(ext(T), X, Commit())
  L.enqueue(P, W)
else
  if P is leaf
    S := S \cup (P, W)
    M.enqueue(P, X, Commit())
  else
    if P.mbr covers R and (R \cap P.rect) covers R
      M.enqueue(ext(P), X, Commit())
      SC := min Ext(W, P) // Choose list SC in P to extend to include W with minimum cost and update MBRs (Algorithm 5)
      L.enqueue(each node in SC and its extended MBR)
      break;
    else
      n := P.child. | P.child.covers R
      L.enqueue(n, R) // Traverse down

// Request locks and insert object, or re-do if conflict occurs

if LockAll(M) // Request all the X locks and check version
  for each pair (P, R) in S:
    p.child(P.entries) := O
    p.rect(P.entries) := R
  if R = W // The object is clipped
    if P.entries > P.capacity // Overflow
      Split(P) // If split propagates 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

---

Algorithm Delete(W, T)

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

O := (); O := T.root; V := (); M := Clip Array; Stack L := ([P.mbr \cap W, P])
if (P is NIL) or (not(P.mbr \cap W))
  return
// Record required locks

if W \cap 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
  (L, P) := L.pop
for each i in P.rect:
  if P.rect \cap R Then
    if P is leaf Then
      O := O \cup P.child. // Add the objects that are not yet in results
      O := O \cup leaf nodes in M that covers P.child. // Add leaf nodes from the object link in clip array
    Else
      if P.child.is.leaf Then
        V.enqueue(P.child, X, Commit())
        l.push([P.rect \cap R, P.child]) // Put the child of P in stack
        R := R - P.rect
      else
        V.enqueue(P, S, Commit()) // S Lock on ext(P) if it overlaps R
        for every node n in O // Lock all the leaf nodes that cover the objects to be deleted
          V.enqueue(n, X, Commit())
          For every internal node n in O 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(M) // Request all the locks and check version
          for each object n in O
            if n in the leaf nodes in O; Delete n in M
            for each underflow leaf node n in O
              Merge(n) // Propagate if necessary
        Else
          Delete(W, T) // Restart the delete operation
          return

---
4.1 Query Performance
Point query and window query operations were executed on the R-tree, the R+-tree, and the ZR+-tree in order to compare their query performance. In this set of experiments, the capacity of the index trees was set to 100; the fill factor was 70 percent, and the data size and query size varied. The density of the synthetic data was set to 4. Building the three types of indexing trees on two real data sets, the height of the trees was always three even for the R-tree, which had the least number of entries in leaf nodes.

4.2 Throughput of Concurrency Control
The performance for concurrent query execution was evaluated both for the R-tree with granular locking and the ZR+-tree with the proposed GLIP 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.

According to the algorithm analysis in the previous section, the ZR+-tree with concurrency control should perform better than the R-tree with granular locking when the write probability is low. This performance gain comes from not only the outstanding query performance of the ZR+-tree but also the finer granules of the leaf nodes in the ZR+-tree. The size of the queries executed was tunable in this set of experiments. The data sets used in these experiments were the same as those used in the query performance experiments, except that the size of the synthetic data set was reduced to 5,000 in order to assess the throughput in relatively small data sets compared to the real data sets. Fig. 1 shows 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, GLIP 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 concurrency control on the ZR+-tree outperforms the R-tree in cases where the write probability is less than 35 percent. From this set of figures, it can be concluded that in reading predominant environments, GLIP on the ZR+-tree provided better throughput than dynamic granular locking on the R-tree, although this advantage tended to decrease as the write probability increased.

5. CONCLUSION
This paper proposes a new concurrency control protocol, GLIP, with an improved spatial indexing approach, the ZR+-tree. GLIP is the first concurrency control mechanism designed specifically for the R4+-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 non-overlap 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 GLIP and the ZR+-tree excel at range queries in search-dominant applications. Extending GLIP and the ZR+-tree to perform spatial joins, KNN-queries, and range aggregation offer further attractive possibilities.

6. REFERENCES


2. Prof. D. Jamuna, Working as Professor & Head of CSE Dept. Jayaprakash Narayan College of Engineering, Mahabubnagar, M.Tech (SE) from School of Information Technology, JNTUH, Hyderabad. BE (CSE) from Vijayanagara Engineering College, Bellary. Experience 15 Years in Teaching Profession. Her areas of Interest are in Wireless Sensor Networks, Data Mining, Networking and guided M. Tech and B. Tech Students IEEE Projects. She is a Member of CSI.

3. Sharath Chandra Lagisetty, II Year - M. Tech (CSE) Research Scholar at CSE Dept. Jayaprakash Narayan College of Engineering, Mahabubnagar. 3 Years of experience in IBM India Software Labs, Bangalore. His research interests include purely specification based software development systems and non-normal form databases, particularly their application, concurrency control, and performance in high-volume transaction processing systems.