GLIP: A Concurrency Control Protocol for Clipping Indexing

Chang-Tien Lu, *Member*, *IEEE*, Jing Dai, *Student Member*, *IEEE*, Ying Jin, and Janak Mathuria

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 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.

Index Terms—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 will be discussed in Section 2.1.

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

- *C.-T. Lu and J. Dai are with the Department of Computer Science, Virginia Tech., 7054 Haycock Road, Falls Church, VA 22043. E-mail: {ctlu, daij}@vt.edu.*
- *Y. Jin is with the Department of Computer Science, Virginia Tech., 2160 Torgersen Hall, Blacksburg, VA 24060. E-mail: jiny@vt.edu.*
- *J. Mathuria can be reached at 4117 Marble Lane Fairfax, VA 22033. E-mail: janak_mathuria@yahoo.com.*

Manuscript received 22 Feb. 2008; revised 27 July 2008; accepted 31 July 2008; published online 2 Sept. 2008.
Recommended for acceptance by A. Tomasic.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-02-0107.
Digital Object Identifier no. 10.1109/TKDE.2008.183.

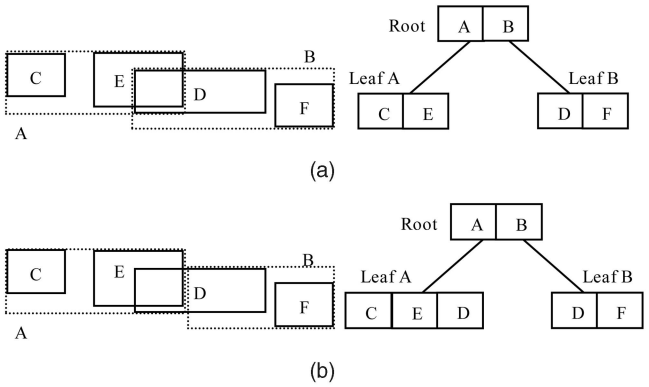


Fig. 1. Examples of R-tree and R+-tree. (a) An R-tree example. (b) An R+-tree example.

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

This paper is organized as follows: Section 2 reviews concurrency control methods and indexing structures in multidimensional databases. Section 3 introduces the structure and characteristics of the proposed ZR+-tree. The details of the concurrency control approaches are discussed in Section 4. Experimental results for both real and synthetic data are analyzed in Section 5. Final conclusions are drawn and future directions are suggested in Section 6.

2 Related Research and Motivation

In this section, we review the structure of the R-tree family, discuss some limitations that affect R+-trees, survey major concurrency control algorithms based on B-trees and R-trees, and summarize the challenges inherent in applying concurrency control to R+-trees.

2.1 The R-Tree Family

The R-tree, an extension of the B-tree, is a hierarchical, height-balanced multidimensional indexing structure that guarantees its space utilization is above a certain threshold. In the R-tree, the root node has between 1 and M entries. Every other node, either leaf or internal node, has between m and M entries $(1 < m \leqq M)$. The leaf node holds references to the actual data and the Minimum Bounding Rectangle (MBR), which covers all the objects stored in that node. The internal node holds references that point to its children (leaf nodes or the next level of internal nodes), the MBRs corresponding to its children, and its own MBR. Unlike B-trees, the keys in R-trees are multidimensional objects that are difficult to define in a linear order. The R-tree is one of the most popular multidimensional index structures as it provides a robust tradeoff between performance and implementation complexity [8]. Many variants based on the R-tree have been proposed to construct optimized indices [28] or to manage spatiotemporal or high-dimensional data [1], [24]. However, as the R-tree allows overlap in the same level nodes, in some cases, the R-tree will have nodes with excessive space overlap and "dead space." This significantly degrades its search performance, because for one search region, there might be several MBRs in each level that need to be visited. To optimize data retrieval performance, several variants have been proposed. For example, the R*-tree [2] tries to minimize overlap for internal nodes and minimize the covered area for leaf nodes via forced reinsert.

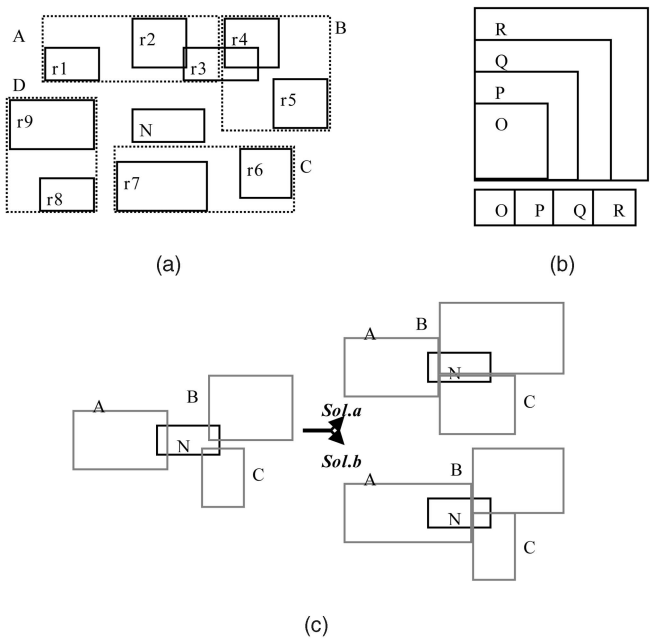


Fig. 2. Limitations in R+-trees. (a) Unable to insert. (b) Unable to split. (c) Different solutions to expand.

The R+-tree was first proposed in [23]. The R+-tree uses a clipping approach to avoid overlap between regions at the same level [7]. As a result of this policy, a point query in the R+-tree corresponds to a single path tree traversal from the root to a single leaf. For search windows that are completely covered by the MBR of a leaf node, the R+-tree guarantees that only a single search path will be traversed. Examples of the R-tree and R+-tree are given in Fig. 1, where A and B are leaf nodes, and C, D, E, and F are MBRs of objects. Because objects D and E overlap with each other in the data space, leaf nodes A and B have to overlap in the R-tree in order to contain the objects. In contrast, in the R+-tree, leaf nodes do not have to cover an entire object, so object D can be included in both leaf nodes A and B. As a result, the R+-tree clearly has a better search performance compared to the R-tree. Experimental analyses of the relative performances of R-trees and R+-trees indicate that R+-trees generally perform better for search operations [8], [12], although this benefit comes at the cost of higher complexity for insertions and deletions. The performance gain for search operations makes the R+-tree ideally suited for large spatial databases where search is the predominant operation.

However, it is important to note the following limitations of the original definition and the operations of the R+-tree. First, some objects may not be inserted into an existing tree, because of an extension conflict between several nodes on the same level. Fig. 2a illustrates a 2D example of this problem. In this case, when an object with MBR N is inserted

into the tree, any one of nodes A, B, C, or D could be chosen to extend to encompass the new object. The region N thus causes a deadlock in this scenario, because whichever node is selected to include N will then overlap with another node. For instance, A will be intersected if B is extended, and B will be overlapped if C is enlarged. According to the definition and the insertion algorithm for the R+-tree, none of these nodes is allowed to cover N while overlapping with other nodes. Therefore, the new object cannot be inserted into the R+-tree. This issue was raised in [7], but no modified algorithm has been presented to resolve it. Second, in some cases, it may not always be possible to split a node in a manner that satisfies all the properties of the R+-tree. In an obvious case, a split is not possible when $M + 1$ MBRs in a node with a capacity of M have the property such that the lower left corners (or upper right corners) of all the MBRs are the same. Fig. 2b shows an example of this problem. Third, the original R+-tree algorithm does not discuss how to clip an inserted object that overlaps with multiple untouched nodes. In the case of an insertion, the nodes that overlap with the object should be enlarged to cover the whole space of the object. As shown in Fig. 2c, there could be multiple ways to perform the node expansion, each leading to a different tree structure. Of the two solutions shown in the figure, solution b will generate a better indexing tree because nodes A, B, and C cover less dead space than in solution a. The proposed ZR+-tree is designed to resolve all these issues.

2.2 Concurrency Controls

Several concurrency control algorithms have been proposed to support concurrent operations on multidimensional index structures, and they can be categorized into lock-coupling-based and link-based algorithms. The lock-coupling-based algorithms [6], [19] release the lock on the current node only when the next node to be visited has been locked while processing search operations. During node splitting and MBR updating, these approaches must hold multiple locks on several nodes simultaneously, which may deteriorate the system throughput.

The link-based algorithms [13], [14], [15], [16], [25] were proposed to reduce the number of locks required by lock-coupling-based algorithms. These methods lock one node most of the time during search operations, only employing lock coupling when splitting a node or propagating MBR changes. The link-based approach requires all nodes at the same level be linked together with right or bidirectional links. This method reaches high concurrency by using only one lock simultaneously for most operations on the B-tree.

The link-based approach cannot be used directly in multidimensional data access methods as there is no linear ordering for multidimensional objects. To overcome this problem, a right-link style algorithm (R-link tree) [14] has been proposed to provide high concurrency control by assigning logical sequence numbers (LSNs) on R-trees. However, when a node splitting propagates and its MBR updates, this algorithm still applies lock coupling. Also, in this algorithm, additional storage is required to retain extra information for the LSNs of associated child nodes. To solve this extra storage problem, Concurrency on Generalized Search Tree (CGiST) [15] applies a global sequence number, the Node Sequence Number (NSN). The counter for NSN is incremented for each node split, with the original node receiving the new value and the new sibling node inheriting the previous NSN and its right-link pointer. In order for the algorithm to work correctly, multiple locks on two or more levels must be held by a single insert operation, which increases the blocking time for search operations.

Several mechanisms, such as top-down index region modification (TDIM), copy-based concurrent update (CCU), CCU with nonblocking queries (CCUNQ) [13], and partial lock coupling (PLC) [25], have been proposed to improve the concurrency based on the above linking techniques. However, the link-based approach with these improvements is still not sufficient to provide phantom update protection.

Phantom updating refers to updates that occur before the commitment, in the range of a search (or a following update), and are not reflected in the results of that search (or the following update). Concurrent data access through multidimensional indexes introduces the problem of protecting a query range from phantom updates. The dynamic granular locking approach (DGL) has been proposed to provide phantom update protection in the R-tree [4] and GiST [5]. The DGL method dynamically partitions an embedded space into lockable granules that adapt to the distribution of objects. The leaf nodes and external granules of internal nodes are defined as lockable granules. External granules are additional structures that partition the noncovered space in each internal node to provide protection. According to the principles of granular locking, each operation requests locks on sufficient granules such that any two conflicting operations will request conflicting locks on at least one common granule. Although the DGL approach provides phantom update protection for multidimensional access methods and granular locks can be efficiently implemented, the complexity of DGL may impact the degree of concurrency.

2.3 Challenges of 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 $\mathrm{ext}(\text{internal node}) = (\text{MBR for the internal node}) - (\text{MBRs for each of its children})$. In Fig. 3, 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. Although this solution solves the above problem for deletions (and updates), the object-level

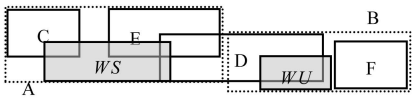


Fig. 3. Example operations for GL/R-tree on an R+-tree.

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.

3 Definition of GLIP 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.

3.1 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 GLIP. 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 GLIP 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.

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.

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

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.

3.2 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]:

1. 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.
2. 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 $RECT$.
3. 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.
4. The root has at least two children unless it is a leaf.
5. All leaves are at the same level.

Some modifications can be made to the original R+-tree to make it suitable for the situations mentioned in Section 2.1.

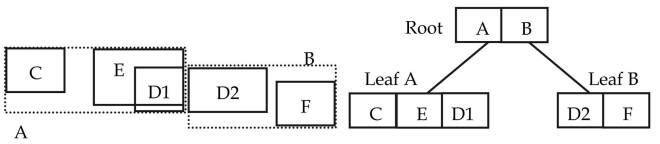


Fig. 4. An example of ZR+-tree for the data in Fig. 1.

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: 1) 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. 4), 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 Fig. 1b). 2) 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. For example, in Fig. 5b, the first entry in the leaf node A contains segmented objects O, P_1, Q_1, and R_1, with the same MBR, and the second entry in the leaf node A contains segmented objects P_2, Q_2, and R_2, with the same MBR. These segmented objects with the same MBR are combined into a single entry. These two features in the ZR+-tree can help to resolve the unable-to-split problem illustrated in Fig. 2b, as well as to reduce the number of leaf nodes after clipping objects. As the proposed object clipping ensures zero overlap in the entire search tree, the structure and the operations become more orthogonal. Furthermore, this zero-overlap design avoids the limitations associated with duplicating the links between objects as discussed in Section 2.1. An example of the ZR+-tree that can be compared to the R-tree and the R+-tree in Fig. 1 is given in Fig. 4, where the object D is clipped into D_1 and D_2 to achieve zero overlap and avoid the construction limitations of the R+-tree.

The definition of the ZR+-tree is given in the form of a revised version of the earlier definition of the R+-tree by modifying property 1 and 2 as follows:

1. A leaf node has one or more entries of the form (*objectlist*, $RECT$) where *objectlist* gives the identifiers for each object that completely encloses or covers $RECT$. Note that a single bounding rectangle with multiple object *ids* is still counted as a single entry, even though it requires extra space in the node. An alternative is to use a pointer as *objectlist* to the entry in a table that stores the corresponding object *ids*.
2. An internal node has one or more entries of the form $(p, RECT)$ where p points to a ZR+-tree leaf or internal node R such that $RECT$ is the MBR of all $(p_i, RECT_i)$ in R. Thus, the definition of the ZR+-tree is more orthogonal as a result of eliminating the difference in rules for the MBRs of leaf nodes and internal nodes. However, the MBR of an object may be fragmented, such that the union of all the fragments equals the MBR of the object, and each of the fragments may be inserted into the same or different leaf nodes.

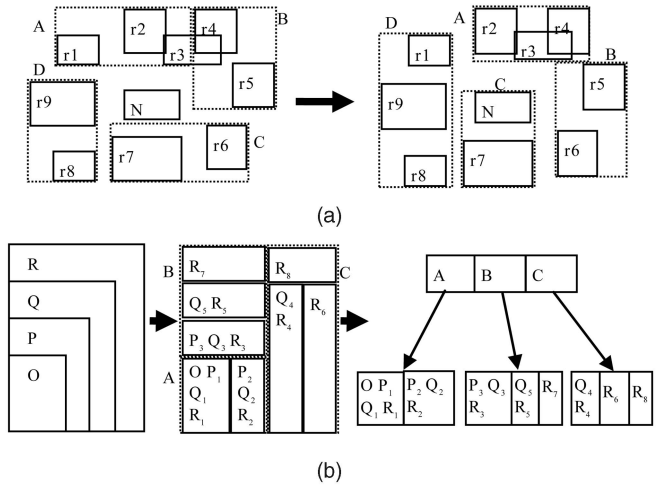


Fig. 5. ZR+-tree solution to the problems in Figs. 2a and 2b. (a) Clustering-based reinsert in ZR+-tree. (b) Object clipping in ZR+-tree.

In addition to the structure evolution, two operation strategies are proposed to improve insertions on the ZR+-tree and refine the indexing tree:

1. While performing an insert operation or a split operation, different plans are evaluated in terms of the the number of new object clippings and the overall coverage. For insert operations, each possible way to expand existing nodes to cover the new object is treated as a plan. Plans for splits are the possible hyperplanes that correspond to any dimension used to divide the node into two parts. The plan with the least number of object clippings, and then the smallest overall coverage, is selected to perform that operation.
2. Once a failure of insertion (as shown in Fig. 2a) or a split propagation caused by updating has occurred, a clustering-based reinsert operation will be performed to optimize the distribution of the nodes. The reinsert will group the entries that are spatially nearby and then construct new entries. The number of new entries will be the same as the number of old entries or the number of old entries plus one. If the reinsert operation fails to enable the insertion of the proper object, eventually, a compelled split, which requires object clipping, will be performed to accomplish the insert operation.

Figs. 5a and 5b show the ZR+-trees corresponding to the R+-trees in Figs. 2a and 2b, respectively, which result from the above modifications of properties. Note that in Fig. 5a, a reinsert has been performed in order to build new entries. These 10 objects are clustered into four groups based on their positions. This new clustering of the entries avoids the deadlock situation. Assured by the compelled split, the insertion deadlock can be resolved. In Fig. 5b, if P is inserted after O, P will need to be fragmented into three rectangles (P_1, P_2, P_3) before it can be inserted. If Q is then inserted after P, similarly, Q will be fragmented into five

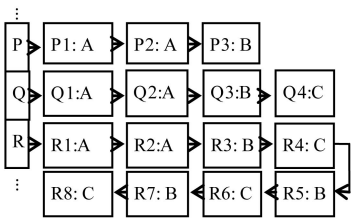


Fig. 6. A clip array for objects in Fig. 5b.

rectangles, in which Q_1, Q_2, and Q_3 are cut to correspond with P's existing rectangles, while Q_4 and Q_5 are fragmented due to the rectangle rules. Similarly, R will be fragmented into seven rectangles. In this way, the original entry of O is now holding the fragments of P, Q, and R, and the whole node can be easily split with these fragments.

In order to support the proposed index tree, additional metadata are required to store the information concerning object clipping. When updating a data object, the operations need to know how many pieces it has been clipped into and in which leaf nodes they are located, and then expand the operation to the remaining parts if necessary. An array of linked structures is designed to maintain the object information necessary to enable such operations. Each clipped object is added as an element of the array, and all the pieces of the object entries, represented by the pointers to the leaf nodes that contain these pieces, will be linked in this array element, as shown in Fig. 6.

As only one MBR and several *ids* for each clipped object are stored in this clip array, it is feasible to store the whole array in physical memory. Based on our experiments with real data, on the average, each object is clipped into less than 1.5 segments, so it is reasonable to assume that each clipped object can use two double integers to denote the MBR and 16 integers as eight links (two *ids* for each link). In this case, 100,000 objects occupy only 4 Mbytes, which is small compared to the memory size available in mainstream computers.

3.3 Lockable Granules

Each leaf node in the ZR+-tree is defined as a 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. 3, the leaf nodes A and B, $ext(A)$, $ext(B)$, and $ext(root)$ are defined as lockable granules. $ext(A)$ covers the region $A.mbr - (C.mbr \cup D_1.mbr \cup E.mbr)$, and $ext(root)$ covers the region $\mathrm{MBR}(A.mbr \cup B.mbr) - (A.mbr \cup B.mbr)$. 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 space. Another option is to define the $ext(root)$ itself to include $ext(T)$. When inserting objects into such space, either approach leads to the same level of concurrency, since any insertion outside the root's MBR leads to the growth of the MBR for the root node and thus conflicts with $ext(root)$. However, for select and delete operations, $ext(root)$ and $ext(T)$ do not necessarily conflict. For example, a delete operation that overlaps with the lock granules C, $ext(A)$, and $ext(root)$ can coexist with a select operation that overlaps with E and $ext(T)$. Thus, defining $ext(T)$ as a separate lockable granule leads to better concurrency. It also effectively handles situations where the tree is empty and the root is NIL. Summarizing the above analysis, the lockable granules in the ZR+-tree for GLIP are defined as all the leaf nodes, external of the nodes, and external of the tree.

4 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.
   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.

4.1 Select

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 1. Search Algorithm

```
Algorithm Select(W, T)
Input: search window W, ZR+-tree T
Output: set of objecteID 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(ext(T), S, Commit) / / Lock external of tree
Lock(ext(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 P.rect_i
    If P.rect_i ∩ R Then
      If P.isLeaf Then
        O := O ∪ P.child_i  / / Add the objects that are not in results
        Unlock(P, S)
      Else
        If P.child_i.isLeaf Then
          Lock(P.child_i, S, Commit)
        Else
          Lock(ext(P.child_i), S, Manual)
        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 = ∅)
    Unlock(ext(P),S) / / Release S Lock on ext(P) if not overlaps R
Return O
```

Selection starts by checking whether the search window overlaps with $ext(T)$. If so, a shared lock is placed on $ext(T)$, thus preventing a writer from inserting data into this space. A breadth-first traversal is then performed starting from the root node and traversing each node whose MBR overlaps with the search window. For each internal node that overlaps with W, an S lock is placed on its external area. This lock is released when all of its child nodes and its external granular have been inspected and locked if necessary. For each internal node, if the MBRs of its children do not fully cover the search window W, an S lock will be kept on the external granule for the node in order to prevent writers from modifying this region. This ensures consistency within the tree, as it prevents writers from modifying the internal node until all the child nodes have been properly inspected and protected. As discussed earlier, in order to reduce the number of locks that must be placed and released, we neither perform object-level locking, nor lock the corresponding objects in the clip array for the select operation. Instead, shared locks are placed on the leaf nodes that overlap with W. Since the same object *id* may recur in the same leaf node or across different leaf nodes, a set of object *ids* is maintained to avoid returning the same object *id* more than once. This is consistent with the expected result from a select statement. Finally, all the locks on the granules that overlap with W are released once the search is complete.

Fig. 7 illustrates the lock management for the window query in Fig. 3. For a search window WS that overlaps with C, E, and D, initially, an S lock will be placed on the root. An S lock is then placed on the leaf node A and the lock on the root is released. This prevents any other transactions from modifying the root (by placing an X lock on it) until all its children have been inspected. After the lock on the root has been released, the entry for node B in the root can be modified as long as the modification does not result in overlap with A. Thus, manual duration S locks are used to maintain consistency while at the same time maximizing the degree of concurrency.

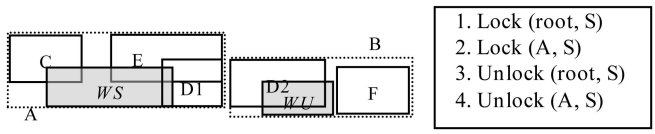


Fig. 7. Locking sequence for WS in Fig. 3.

4.2 Insert

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 (P_1, P_2, P_3) 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.

Algorithm 2. Insert Algorithm

```
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 := {}; S2 := {}
// Record required locks
If W ∩ P.mbr<>P.mbr  //root does not cover W
  M.enqueue({ext(T), X, Commit})
L.enqueue({P, W})
Loop until L is ∅
  (P, R) := L.dequeue
  If P.isLeaf
    S2 := S2 + {P, R}
    M.enqueue({P, X, Commit})
  Else
    If P.mbr covers R and !(∃ i, P.rect_i 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.child_i | P.child_i 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 every pair (P, R) in S2
  P.child(P.entries) := O
  P.rect(P.entries++) := R
  If R<>W  //The object is clipped
    StoreClipArray(O, 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
```

The insert operation without concurrency control protocol proceeds as follows: First, a breadth-first traversal is performed from the root node. When W is found to be covered by node N but not any single child of N, the child nodes of N are selected to extend if N is an internal node. If SC is the set of child nodes for N, SC is partitioned into two sets, S_1 and S_2, such that S_1 contains all the child nodes whose MBRs need not be changed, and S_2 is the set of nodes that must be changed in order to cover W. In order to select the appropriate set of nodes to extend the MBRs, a heuristic strategy is adopted to choose the fewest nodes involved and then the smallest coverage. This leads to a relatively small tree and a small coverage area, both of which contribute to a better search performance by fitting more index to memory and eliminating paths as early as possible.

No granules are locked during this traversal, although all the granules that overlap with W are recorded. After the traversal, X locks are placed on all of these lock granules in an atomic manner. If the locks are successfully acquired, the actual insertion can then be performed. Since the X locks are retained on all these granules until the transaction is complete, this guarantees that any other operations that need to traverse any part of the path impacted by the insertion must wait until the transaction is committed. At the same time, since any active selection will hold S locks on all the granules that it has covered and update operations always attempt to place S or X locks on the area they intersect, an insert operation will only be performed when no active insertion, deletion, or selection that overlaps with the insertion is present, thus ensuring serializability.

There is still a risk that two insertion transactions T_1 and T_2 could be performed at the same time, following intersecting paths and then waiting for X locks. If no selections are active and T_1 acquires the X locks first, it will perform the insertion and then commit. Now, T_2 can acquire the X locks, but the path it had previously traversed is *dirty*. In order to prevent T_2 from performing an insertion on this dirty data, a version number is maintained for each node. All requests for an X lock implicitly pass the current version of the node that the X lock is being requested for. When the X lock becomes grantable, the current version number is compared with the version number at the time of the request. If they do not match, the lock is released and a dirty signal is returned, causing the insert procedure to be restarted.

Conflicts between insertions that could cause deadlocks are avoided by simultaneously requesting all the X locks needed by an insertion. With the proposed protocol, as part of the insert operation, the insertion only holds X locks once and requires no lock before or afterwards. Thus, no deadlock can be induced using this protocol, since for any deadlock to occur, the protocol would need to request a conflicting lock while simultaneously holding other locks. If the X locks are not requested at the same time, and the insertion were to place X locks on each lockable granule it traverses, it is possible that an X lock has been propagated bottom-up by a node split in an insert operation, while at the same time, a select operation is attempting to acquire an S lock on the same node. This would cause a deadlock.

To conclude the insert algorithm shown in Algorithm 2, the actual insertion is performed as follows: Pending insertions into all the leaf nodes are performed first. At this point, nodes that overflow are not split but only marked for splitting. Using the minExtend function (shown in Algorithm 3), the nodes in S_2 are then expanded to include the new object W following an optimal plan with the fewest number of nodes and the smallest size of area involved. The node expansion is only logical, since they have not yet been locked. Should the expansion fail, a reinsert function (to be introduced in the next subsection) will be invoked to reconstruct S_2. After the expansion, the new object W is segmented into pieces that can be covered by the N nodes, where N is the size of S_2. This process repeats until all the segments of W have been inserted into leaf nodes. Even if the resulting leaf nodes overflow after inserting W, the overflowed nodes are not physically split at this point but only marked for splitting. Since $S_2.\mathrm{MBR}$ does not overlap with the MBR of any of its siblings, splitting S_2 into nodes will only produce nodes whose MBRs do not overlap with their siblings as long as the split does not extend the MBR of the splitting region. A split algorithm that follows the approach in [23] guarantees this. The node insertion is now completed, and all the nodes that are to be locked will be X locked. The protocol then splits each leaf node marked for splitting, and inserts the new leaf node in the lowest level internal nodes. If this insertion causes an overflow in the lowest level internal nodes, they are not split immediately but only marked for splitting. Once all the marked leaf nodes have been split, any lowest level internal nodes that had been marked for splitting are split. This splitting may propagate the tree as required. Since not all the internal nodes are locked, this may cause the split to propagate to an internal node that has not been locked. In this case, this internal node is added to the list of nodes that require X locks, and the tree is restored to its original state before the insertion. The process is then repeated as it waits for locks on all the nodes.

Algorithm 3. minExtended Function

```
Function: minExtend(W, T)
//Choose a set of nodes to include object W with minimum cost
Input: key W, ZR+-tree T
Output: NodeList SC

N := {}; SS := {}
Loop until N = {T.child_i}
  P := the nearest child of T to W and not in N
  N := N+P
  For all combinations of nodes in N
    If the set of nodes SN can extend to cover W w/o overlapping
      SS := SS+SN
  If SS <> ∅
    SC := solution in SS with least nodes and then least coverage
    Break
If SS = ∅
  Lock(ext(T.root), X, Manual) //Lock this branch for re-insert
  Re-insert(W, T)
  Unlock(ext(T.root), X)
  SC := minExtend(W, T)
Return SC
```

Assuming that the update window WU in Fig. 7 indicates an object G to be inserted, this algorithm can be processed as follows. In the step of recording required locks, the leaf node B is selected to contain the object G and recorded for X lock requests. After an X lock is placed on the unchanged leaf node B, the algorithm modifies B by adding the information for the object G. Finally, this X lock is released before commission.

Clearly, if the select requests from other transactions continue to arrive while the insertion is waiting for the X locks to be granted, it is possible that the transaction that is waiting for insertion never acquires its lock, resulting in a starvation. To prevent this, a scheduling mechanism is used to ensure S locks are granted on the resources that other transactions are waiting for an X lock on, if and only if the transaction that requests the S lock arrives before the X lock request. The details of this policy are not discussed in this paper, but interested readers may refer to [27].

4.3 Reinsert

In some cases, insertion may fail (as shown in Fig. 2a) because of complex spatial relationships among existing nodes. Moreover, propagated splits caused by updating are difficult to avoid in update operations. A reinsert function is therefore required to resolve any insertion deadlock and alleviate split propagation. The objective of a reinsert function is to decompose the existing nodes and form new nodes rationally based on their spatial locations. Compared with the existing reinforced insert operation in R*-tree [2], this reinsert function focuses on redistributing index entries of multiple sibling nodes rather than on optimizing the distribution of the children of only one node. Therefore, this reinsert operation can relieve the deadlock situation illustrated in Fig. 2a, which requires redistributing the objects with a common grandparent node that the existing reinforced insertion cannot properly handle. Ultimately, the reinsert method guarantees the success of the insert operation by a compelled split.

Specifically, as shown in Algorithm 4, the reinsert function works as follows: Given a set of entries from K nodes (including the object to insert if the aim is to resolve an insertion deadlock), a clustering algorithm is used to generate the center entries of K clusters of the entries. These K center entries are used to form K nodes as the first level of a subtree. Consequently, the remaining entries are inserted into this subtree based on their minimal distances from any center entry. After inserting all the entries, these nodes will replace the original nodes in the ZR+-tree. If after this stage, the insertion still fails, a compelled split (Algorithm 5) is performed to split one of the K nodes, so that a subset of the K nodes can be extended to cover the new object. During the processing of the reinsert, an X lock will be requested on the parent of the K nodes by its invoker to protect this subtree from concurrent update operations.

Algorithm 4. Reinsert Function

```
Function: Re-insert(W, T)
// Re-arrange the entries in T's children according to their distribution
Input: key W, ZR+-tree T
Output: NIL

SC := {child nodes of T.root}; SCC := {children of SC and W}; C := Centroid(T.root.rmb); SP:= ∅
P := the farthest mbr to C in SCC // Based on centroid-to-centroid distance
SP := SP+P
For N:=2 to SC.size
  P:= the farthest mbr to SP in SCC
  SP := SP+P
SP := cluster(SCC, SP) // Find SC.size items from SCC as central objects for each cluster using SP as feeds
T := construct(SP) // Construct SC.size children for T.root using each node in SP as the only entry in each child
For every member n in SCC but not in SP, sorted by their minimal distance to any mbr in SP
  Insert(n.mbr, n, T)
Compelled-split (W, T) // Split a child of T, to enable the insert of W
Return
```

Algorithm 5. Compelled-split Function

```
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
```

The classical k-means algorithm is used for this clustering task because the number of clusters is fixed. Other clustering algorithms that can return a fixed number of clusters may also be applied. One essential step to reduce the complexity of the clustering is to choose appropriate K seeds to initiate the clustering. An optimal strategy is to select K seeds that are as far away from each other as possible. A similar idea has been applied in the CURE clustering algorithm [9].

This clustering-based reinsert function can group the entries according to their distributions. With the compelled split, this function prevents insertion failures and also alleviates the need for excessive propagated splits. Furthermore, the tree structure will be refined after applying the reinsert function, because the affected objects are more likely to be grouped into their natural spatial clusters, regardless of the order of insertions.

4.4 Delete

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. As addressed earlier in Section 3, 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 6. Delete Algorithm

```
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
```

Since the delete operation requests X locks on the leaf nodes that contain segments of the objects to be deleted, this will conflict with the S locks placed by the select operation. In cases where this delete operation does not cause nodes to merge, all the lockable granules that intersect with the deletion window are exclusively locked before the actual deletion is performed. Once underflow occurs, X locks will be placed not only on the underflow node but also on its parent, as long as its MBR needs to be shrunk or removed because of the underflow. Thus, any search that commits after the deletion is complete will not retrieve the objects affected by this deletion. The delete operation also requests S locks on $ext(P)$, where P is an internal node, and $ext(P)$ overlaps with the deletion window while not being exclusively locked. Therefore, no new objects that intersect with $ext(P)$ can be inserted before the commitment of this deletion. While accessing the clip array to find fragments of the selected objects, X locks will also be requested on the leaf nodes that cover these fragments, thus providing phantom access protection.

The example in Fig. 7 can be used to illustrate the delete algorithm by considering WU as a deletion window such that all the objects that intersect with WU must be deleted. In the step of recording required locks, the leaf node B is recorded first since it covers WU. Next, the leaf node A is recorded because it contains the object segment D_1, whose original object has another segment D_2 that intersects with WU. After investigating the intersected objects and required locks, X locks are placed on nodes A and B at the same time. Both leaf nodes are then modified by removing D_1 and D_2 accordingly. Meanwhile, the entry for D is deleted from the clip array. At the time of commission, these X locks will be released.

4.5 Analysis

Based on the proposed GLIP protocol, ZR+-tree operations meet the requirements of serializable isolation, consistency, and no additional deadlocks. Specifically, serializable isolation is guaranteed by the strategy of requesting S locks on reading and X locks at the same time on updating. These locks are granted on the affected granules before the actual actions and provide protection until the process is complete. Therefore, the intermediate status of one operation cannot be exposed to any other operations. The consistency requirement is ensured by implementing version checking and restarting the insertion or deletion when the version does not match. This version checking prevents the update operations from modifying a version of the ZR+-tree that differs from the one investigated. Finally, the deadlock-free in GLIP can be validated as Proof 1, based on the conclusion that common resources are not accessed in opposing orders, which can be proved by contradiction. A major benefit of the proposed design is that phantom update protection is assured by the ability to lock on different granules.

Proof 1: Deadlock-fee in GLIP.

Proof:

∵ The select algorithm requests S locks following the tree traversal from root to leaf and then from left to right on the same level.

∵ The strategy of lock-at-one-time for insertion and deletion makes sure that all the affected granules are exclusively protected in an atomic manner. In other words, the X locks in one operation are either placed on all the affected granules or pending until all these granules become available for X locks.

∵ There is no deadlock if the shared resources are not accessed in the opposite orders.

∴ There will never be two operations in GLIP that exclusively hold a potion of the resources required by each other.

Proved.

As the proposed GLIP protocol takes into account object clipping, it can be extensively applied in the R+-tree and its assorted variants. If it is applied in the R+-tree, the necessary modification will be to simplify the clip array until it contains only references to leaf nodes that cover the same object. Because the R+-tree uses the reference to a complete object as each entry in a leaf node, with this change, GLIP provides phantom update protection in the R+-tree.

The ZR+-tree guarantees that if a query window is entirely contained in the MBR of a leaf node, only a single search path is followed. It also ensures that only one search path will be followed for point queries. Neither of these is true in R-trees. Therefore, given an R-tree and a ZR+-tree with the same height, the ZR+-tree is likely to provide better search performance, similar to that of the R+-tree. Not only is following multiple paths inefficient, but a search in an R-tree would also result in a point query locking multiple leaf granules, thus reducing concurrency. Compared to the R+-tree, the ZR+-tree refines the node extension function in insertion, applies the reinsertion approach, and adopts the orthogonal object clipping technique. In this way, the ZR+-tree optimizes tree construction and removes insertion and splitting limitations.

According to the definition, the number of entries in ZR+-trees may be larger than the number of actual objects due to fragmentation. These extra entries lead to additional space requirements for the ZR+-tree and might also increase the height of the ZR+-tree, which would possibly degrade the efficiency of the search operation. In the worst case, if the total number of leaf nodes in the ZR+-tree that can be extended to cover part of the inserted object W without overlapping with other nodes is N, neglecting potential splits, W will need to be fragmented into at most N fragments. Note that this worst case is applicable only when no fragments in W that are covered by extending a leaf node in N can be covered by extending another leaf node in N. When fewer leaf nodes are covered by the inserted window, the number of fragmentations due to the insertion decreases. Furthermore, if the corresponding segments from different objects have exactly the same MBR, they are treated as a single entry in the leaf node. This approach keeps the number of entries in the ZR+-tree similar to or smaller than for the R+-tree, which has been validated by our experiments. As a result, a ZR+-tree where the size of the data set varies exponentially could be expected to increase the height linearly, given a suitable fan out.

It is significantly more complex to implement insert and delete operations for ZR+-trees, and these operations also consume extra CPU cycles and I/O operations. Thus, the insert and delete operations could be expected to be slower than their R-tree and R+-tree counterparts. However, the complexity of the algorithm implementation itself can be neglected for practical applications if the increase in performance for the select operation is significant, especially since the implementation is a one-time cost.

Summarizing the above analysis, implementing GLIP on the ZR+-tree can provide an efficient, stable, and extendable multidimensional access method that enables concurrent operations and is expected to outperform existing methods for searching-predominant applications.

5 Experiments

In order to evaluate the performance of the proposed concurrency control protocol, GLIP, two sets of experiments were conducted as illustrated in Fig. 8. The first set compared the construction and query performance of the ZR+-tree, the R+-tree, and the R-tree, while the other compared the throughput of GLIP on the ZR+-tree and Dynamic Granular Locking on the R-tree. The experimental design consists of four components: selecting/generating benchmark data sets, constructing multidimensional indices, executing query operations, and measuring respective performance. The experiments compared the ZR+-tree and various indexing trees using two benchmark data sets from the R-tree Portal [26], namely, major roads in Germany (28,014 rectangles), roads in Long Beach County, California (34,617 rectangles), and a uniformly distributed synthetic data set (50,000 rectangles). In the real data sets, rectangles were used to indicate segments of the roads. Relatively speaking, the data distribution of the roads in Long Beach County is skewed, while the roads in Germany are more globally uniformly distributed. The synthetic data set is uniformly distributed with a tunable density, which means that every point in the space is covered by a certain number of rectangles. As shown in Fig. 8, indexing trees were built for these data sets by varying size, controllable capacity, and fill factors. In the query operation stage, some data were randomly taken from each of the above data sets for insertion. The queries to be executed in both sets of experiments were generated by randomly choosing the query anchor from the data file and generating a bounding box by varying query window sizes. The numbers of disk accesses during execution were collected as the measure in the first set of experiments. In the second set of experiments, the write probability and concurrency level were changed to obtain the corresponding throughput.

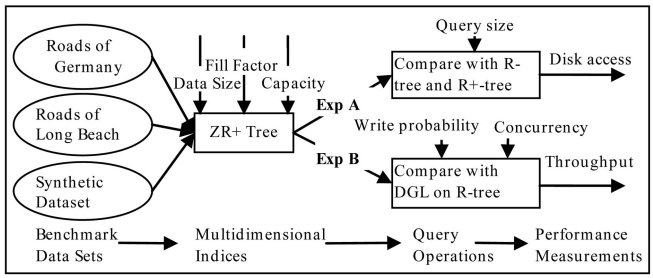


Fig. 8. Experimental design.

The experiments were conducted on a Pentium 4 desktop with 512 Mbytes memory, running a Java2 platform under windows XP. The implementations of the R-tree, the R+-tree, and the ZR+-tree were all based on the Java source package for R-tree obtained from the R-tree portal [26].

The first set of experiments evaluated the construction and query performance of the ZR+-tree. In these experiments, different data sizes were selected to construct the ZR+-trees, R-trees, and R+-trees. In evaluating the query performance, I/O cost is the determining factor, because the query process on the ZR+-tree does not introduce extra computation compared to the R+-tree. The disk accesses of the point queries were recorded by varying the number of rectangles. Additionally, the standard deviations of the number of disk accesses were calculated to compare the stability of the ZR+-tree and the R+-tree. Consequently, queries with different window sizes were executed on the constructed trees in order to record the execution cost. From the analysis of the algorithm given in the previous section, both the point query and window query performances of ZR+-trees are expected to be better than those of the R-trees. The number of disk accesses in this set of experiments was computed to be the average value for 1,000 random queries in order to reduce the impact of uneven data distribution.

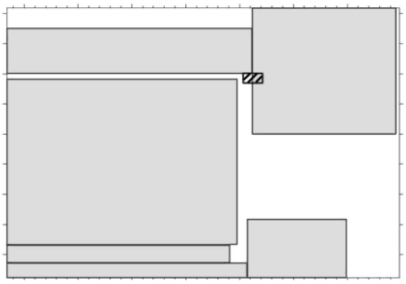


Fig. 9. Construction failure in R+-tree on Long Beach data.

The second set of experiments evaluated the throughput of GLIP on the ZR+-tree by comparing it with dynamic granular locking on the R-tree [4]. The throughputs for the two trees were evaluated under different write probabilities and concurrency levels.

5.1 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.

5.1.1 Point Queries

According to the design, the performance for point queries on the ZR+-tree should be better than that on the R-tree and comparable to that on the R+-tree. Fig. 10 compares the number of disk accesses of point queries for each of these three indexing trees, as well as the standard deviation of disk accesses for ZR+-trees and R+-trees. The left figures show the number of average disk accesses on the y-axis, and the size of data sets on the x-axis. The right figures plot the standard deviations on the y-axis and the size of data sets on the x-axis. While the disk accesses of the R-tree increases along with the size of the data set, the point query performance of the ZR+-tree and the R+-tree remains much lower than that of the R-tree as the number of objects increases. In both the roads of Long Beach County and the synthetic data sets, the number of disk accesses of the ZR+-tree remains almost constant, indicating that its performance is quite scalable. Interestingly, while constructing R+-trees, the program encountered a construction failure when the data size reached around 19,000 because of an insertion deadlock in the roads of Long Beach County data set. To make the comparison complete, this particular object was removed and repaired R+-trees were used (represented by a dashed curve in Fig. 10b). Fig. 9 shows the deadlock situation in detail, where the shaded rectangle indicates the object to be inserted, and the gray rectangles are the internal nodes in the R+-tree. In this situation, the nodes cannot be extended to cover the object without overlapping with each other. Although the I/O costs of the ZR+-tree and the R+-tree are similar, the ZR+-tree consistently achieved lower or equal (only twice) standard deviations in all three

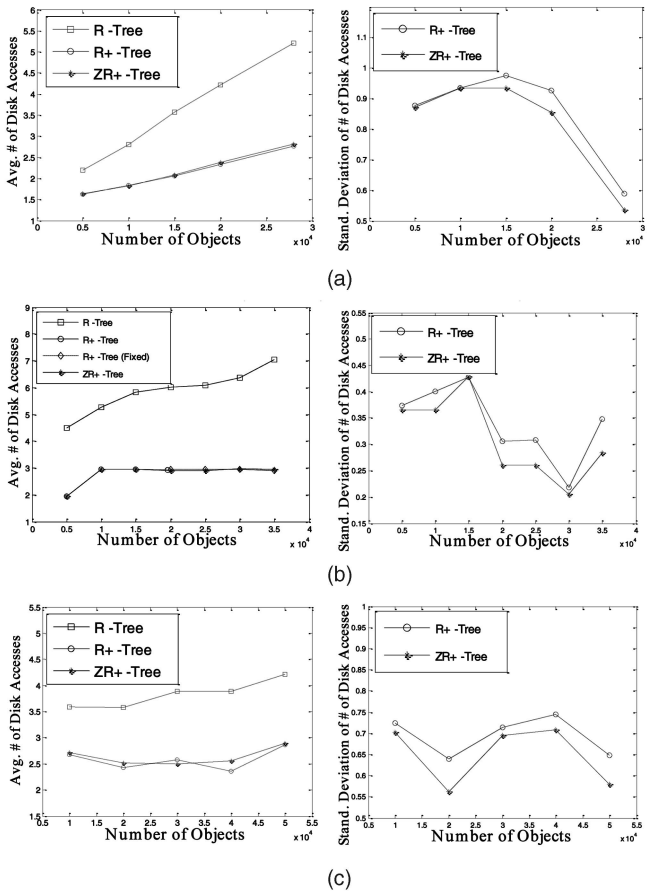


Fig. 10. Point query performance of R-tree, R+-tree, and ZR+-tree. (a) Point query on major roads of Germany. (b) Point query on roads of Long Beach County. (c) Point query on synthetic data.

data sets, which indicated that the ZR+-tree processed point queries more stably. An examination of these outputs showed that in most of the tested cases, the point query performance of the ZR+-tree was much better than the R-tree in terms of I/O cost and more stable than the R+-tree in terms of the standard deviation.

5.1.2 Window Queries

For window queries, the full data sets were used in the experiments (28,014 rectangles for Major German Roads, 34,617 rectangles for Long Beach County Roads, and 50,000 rectangles for the synthetic data). As Fig. 11a (left) reveals, the ZR+-tree has a similar curve of average disk accesses to that of the R+-tree, but the performance is consistently better. It also performs better than the R-tree when the query window size is set to be no larger than 1.2 percent of the data space. When the window size increases, because the size of the leaf nodes in the ZR+-tree and the R+-tree are usually smaller than those in the R-tree, which allows for overlap among the nodes, window queries in the ZR+-tree and the R+-tree will cover more leaf nodes than in the R-tree, thus increasing the number of disk accesses. For the same reason, the R+-tree performs worse than the R-tree in Fig. 11b for query windows larger than 0.2 percent in terms of disk accesses. In all three of the data sets, the performance of the ZR+-tree is generally better than the R-tree and the R+-tree, with the windows size varied

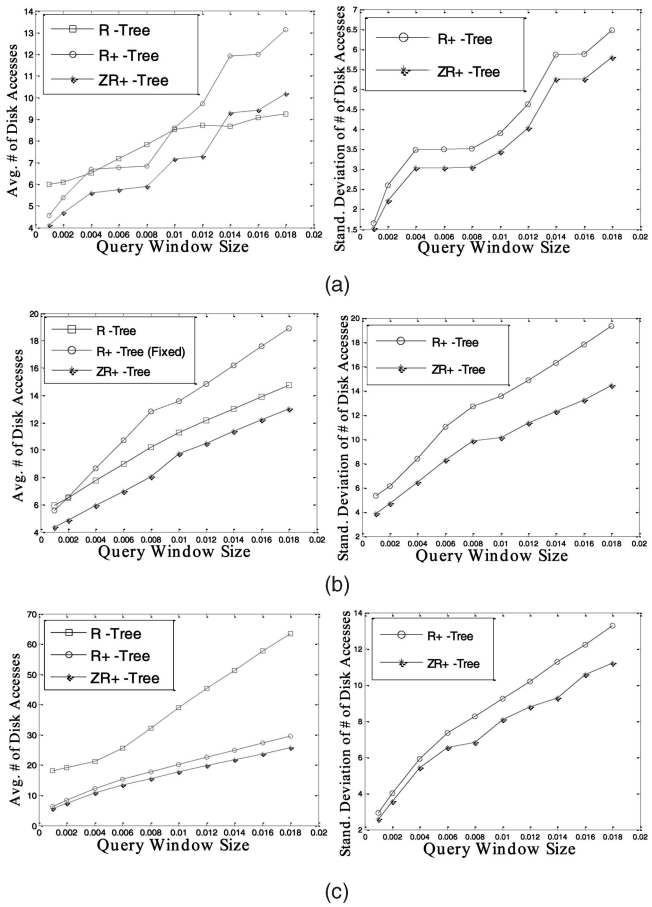


Fig. 11. Window query performance of R-tree, R+-tree, and ZR+-tree. (a) Window query on major roads of Germany. (b) Window query on roads of Long Beach County. (c) Window query on synthetic data.

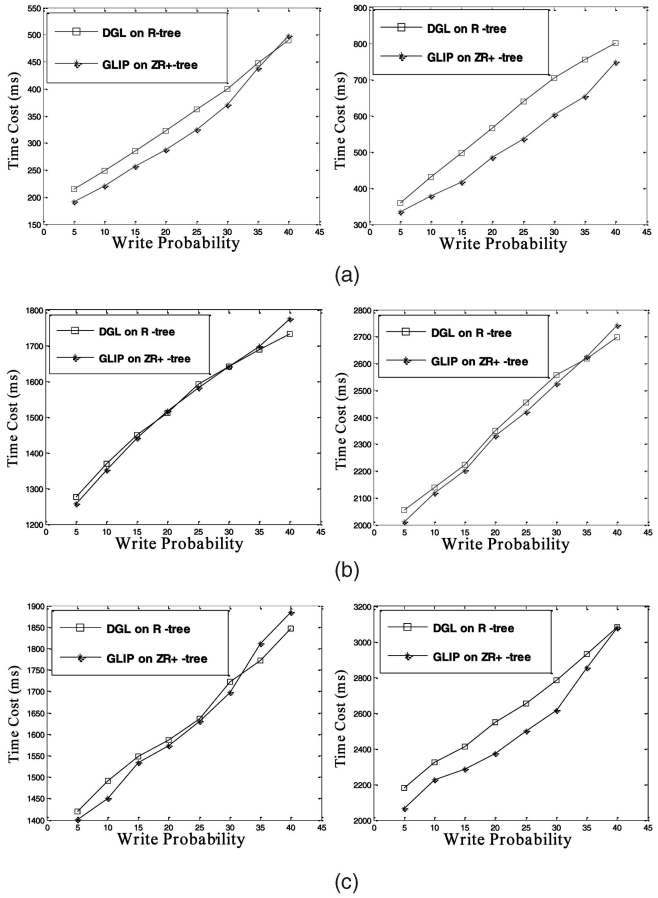


Fig. 12. Execution time for different concurrency levels. (a) Synthetic: Con. Level: 30. (b) Synthetic: Con. Level: 50. (c) Roads of Ger.: Con. Level: 30. (d) Roads of Ger.: Con. Level: 50. (e) Roads of LB: Con. Level: 30. (f) Roads of LB: Con. Level: 50.

from 0.1 percent to 2 percent of the data set. The only exception is when the window size is larger than 1.2 percent in the German Roads data set. Furthermore, the R+-tree has higher standard deviations than the ZR+-tree for the same query window sizes, shown in the right plots in Figs. 11a, 11b, and 11c. In most real applications, the size of the query window is much smaller than 1 percent of the whole data set, and these results showed that the ZR+-tree outperformed both the R+-tree and the R-tree for most window queries.

5.2 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. 12 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

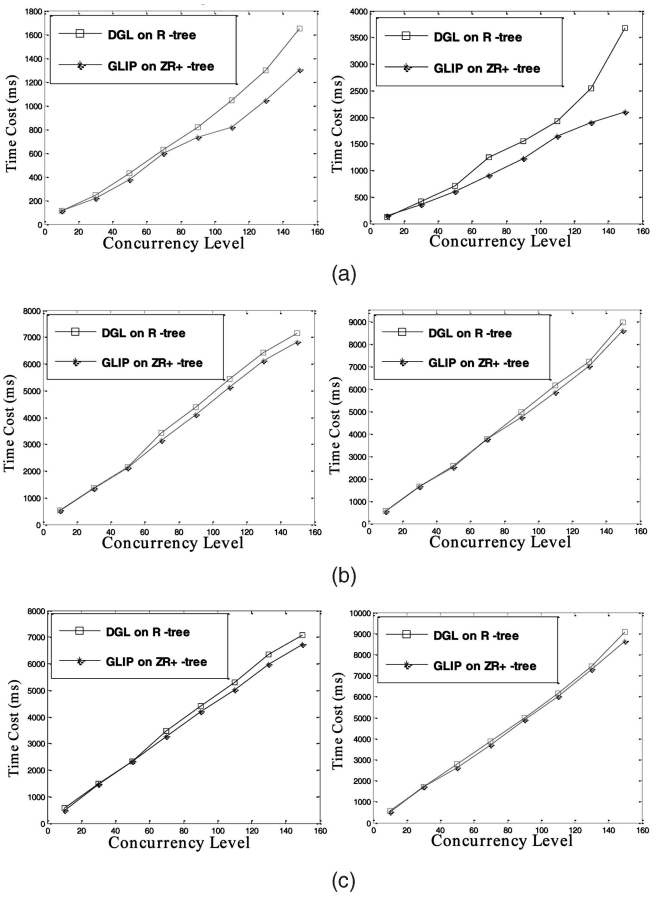


Fig. 13. Execution time for different write probabilities. (a) Synthetic: Write Prob.: 10 percent. (b) Synthetic: Write Prob.: 30 percent (c) Roads of Ger.: Write Prob.: 10 percent. (d) Roads of Ger.: Write Prob.: 30 percent (e) Roads of LB: Write Prob.: 10 percent. (f) Roads of LB: Write Prob.: 30 percent.

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.

Fig. 13 illustrates how the concurrency control protocols perform with fixed write probabilities under different concurrency levels. The y-axis shows the time costs to finish the concurrent operations in milliseconds, and the x-axis represents the number of concurrent operations. The write probabilities were fixed as 10 percent and 30 percent as representative values to reveal trends, while the concurrency level varied from 10 to 150. In these experiments, GLIP on the ZR+-tree consistently performed better than or similar to the DGL on the R-tree. When the concurrency level increases, the advantage of GLIP on the ZR+-tree becomes more and more significant compared to DGL on the R-tree. As these figures show, the advantage in the execution time of GLIP on the ZR+-tree is significant when the concurrency level is more than 50 in the two real data sets and more than 10 in the synthetic data set, with a write probability of 10 percent. All the figures in Fig. 13 show a similar trend, namely, that the advantage of GLIP on the ZR+-tree increases as the number of concurrent operations increases and is particularly significant for evenly distributed data sets compared to DGL on the R-tree.

To summarize our experimental results on both query performance and concurrency control throughput, the ZR+-tree outperformed the R-tree in terms of both point query and window query costs and outperformed the R+-tree in terms of both I/O cost and the stability of both point queries and window queries. Comparing the concurrency control protocols, GLIP on the ZR+-tree performed better than dynamic granular locking on the R-tree, especially with high concurrency and low write probability. It is therefore particularly suited to applications that access multidimensional data with high concurrency and low write probability.

6 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 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 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.

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] A. 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.

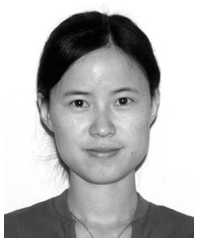
[9] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," *Proc. ACM SIGMOD '98,* pp. 73-84, 1998.

[10] A. 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] C. 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.
[19] V. Ng and T. Kamada, "Concurrent Accesses to R-Trees," *Proc. Third Symp. Advances in Spatial Databases (SSD '93),* pp. 142-161, 1993.
[20] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Systems,* vol. 9, no. 1, pp. 38-71, Mar. 1984.
[21] J.A. Orenstein and T.H. Merrett, "A Class of Data Structures for Associative Searching," *Proc. Third Symp. Principles of Database Systems (PODS '84),* pp. 181-190, 1984.
[22] J.T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD '81,* pp. 10-18, 1981.
[23] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th Int'l Conf. Very Large Data Bases (VLDB '87),* pp. 507-518, 1987.
[24] L. Shou, Z. Huang, and K.-L. Tan, "The Hierarchical Degree-of-Visibility Tree," *IEEE Trans. Knowledge Data Eng.,* vol. 16, no. 11, pp. 1357-1369, Nov. 2004.
[25] S.I. Song, Y.H. Kim, and J.S. Yoo, "An Enhanced Concurrency Control Scheme for Multidimensional Index Structure," *IEEE Trans. Knowledge Data Eng.,* vol. 16, no. 1, pp. 97-111, Jan. 2004.
[26] Y. Theodoridis, "The R-Tree Portal," http://www.rtreeportal.org, 2005.
[27] P.S. Yu, K.-L. Wu, K.-J. Lin, and S.H. Son, "On Real-Time Databases: Concurrency Control and Scheduling," *Proc. IEEE,* vol. 82, no. 1, pp. 140-157, Jan. 1994.
[28] D. Zhang and T. Xia, "A Novel Improvement to the R*-Tree Spatial Index Using Gain/Loss Metrics," *Proc. 12th ACM Int'l Symp. Advances in Geographic Information Systems (ACMGIS '04),* pp. 204-213, 2004.

Chang-Tien Lu received the MS degree in computer science from the Georgia Institute of Technology, Atlanta, in 1996 and the PhD degree in computer science from the University of Minnesota, Twin Cities, in 2001. He is an associate professor in the Department of Computer Science, Virginia Polytechnic Institute and State University and is the founding director of the Spatial Lab. He served as a program cochair of the 18th IEEE International Conference Tools with Artificial Intelligence in 2006 and the 2007 IEEE International Workshop on Spatial and Spatial-Temporal Data Mining. His research interests include spatial databases, data mining, data warehousing, geographic information systems, and intelligent transportation systems. He is a member of the IEEE.

Jing Dai received the BS degree in computer science from Fudan University, China, in 2001 and the MS degree in computer science from the National University of Singapore in 2003. He is currently a PhD student in the Department of Computer Science, Virginia Tech. His research interests include spatial databases, data mining, concurrency control, and intelligent transportation systems. He is a student member of the IEEE.



Ying Jin received the BS degree in computer science from Fudan University, China, in 2001 and the master's degree in computer science from Shanghai Jiaotong University, China, in 2004. She is currently a PhD student in computer science at the Department of Computer Science, Virginia Tech. Her research interests include data mining and bioinformatics.

Janak Mathuria received the MS degree in computer science from Virginia Tech in 2004. He has extensive industry experience in very large databases, automated program analysis, and reengineering tools and logic programming. 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.