Appeared in the Proceedings of International Conference on Data Engineering, February 23-27, 1998, Orlando

Dynamic Granular Locking Approach to Phantom Protection in R-trees *

Kaushik Chakrabarti

Department of Computer Science
University of Illinois at Urbana-Champaign
kaushikc@cs.uiuc.edu

Sharad Mehrotra

Department of Computer Science
University of Illinois at Urbana-Champaign
sharad@cs.uiuc.edu

Abstract

Over the last decade, the R-tree has emerged as one of the most robust multidimensional access methods. However, before the R-tree can be integrated as an access method to a commercial strength database management system, efficient techniques to provide transactional access to data via R-trees need to be developed. Concurrent access to data through a multidimensional data structure introduces the problem of protecting ranges specified in the retrieval from phantom insertions and deletions (the phantom problem). Existing approaches to phantom protection in B-trees (namely, key-range locking) cannot be applied to multidimensional data structures since they rely on a total order over the key space on which the B-tree is designed. This paper presents a dynamic granular locking approach to phantom protection in R-trees. To the best of our knowledge, this paper provides the first solution to the phantom problem in multidimensional access methods based on granular locking.

1 Introduction

Over the past few years, many multidimensional data structures (e.g., R-trees [7], grid files [18], hB-trees [3]) have been proposed to meet the requirements of emerging database applications like computer-aided design (CAD), geographical information systems (GIS), exploratory scientific applications, astronomical data archives, medical image repositories, and multimedia databases. Despite extensive research, most of the developed data structures have not been integrated into existing DBMSs as access methods[1]. Among the primary reasons is the lack of protocols to provide transactional access to data and to guarantee the consistency of the index structures in the presence of concurrent operations. This paper addresses the problems resulting from the concurrent access to data in the database via multidimensional access methods.

Concurrent access to data through a multidimensional index structure introduces two independent concurrency control problems. First, techniques must be developed to prevent concurrent insertions, deletions and updates from violating the consistency of the data structure. Usage of the standard *two-phase locking* protocol [4] for this purpose results in the data structure becoming a bottleneck and thus poor performance. Many approaches that exploit the structure and the semantics of the operations to provide high concurrency have been developed for B-trees [4]. These approaches need to be generalized to multidimensional data structures. Initial steps in this direction can be found in [11, 21] which discuss techniques to support concurrent operations on the R-tree and the grid file respectively.

Second, techniques must be developed to protect the ranges specified in the retrieval from subsequent insertions and deletions before the retrieval commits. Such insertions and deletions are referred to as the *phantoms*. Consider a *range query* over a B-tree based on the salary of an employee to retrieve all employees with salary between 10K and 20K. To answer such a query, the B-tree is traversed to retrieve the pointers to the data pages containing the records qualifying the predicate 10K $\leq$ salary $\leq$ 20K. Even if all objects currently in the database that satisfy the predicate are locked, the object-level locks will not prevent subsequent insertions into the search range. These insertions may be a result of insertion of new objects or rolling-back deletions made by other concurrent transactions. So if the searcher repeats its scan, it will find new objects in its range which seem to appear from nowhere (hence referred to as phantoms). The technique used to provide phantom protection in B-trees is *key range locking* (KRL)[15, 16]. KRL, however, relies on the presence of a total order over the underlying data based on their key value and hence is inapplicable to multidimensional index structures.

This paper concentrates on developing techniques to provide phantom protection to retrievals when the data is accessed via R-trees. The R-tree is one of the most popular multidimensional data structures as it provides the best tradeoff between performance and implementation complexity [5]. Several variants of R-trees (R+trees [23], R*-trees [1], Greene's R-tree [5]) have been proposed in the literature in order to optimize its performance. R-trees and its variants can be used for indexing both point and spatial data and is the only multidimensional index structure known to have been incorporated as an access method into a commercial data management system [9]. It has also been implemented as a part of the Paradise parallel data management system [2]. Although we are addressing the problem in the context of R-trees, the approach developed in this paper can be applied to other tree-based multidimensional access methods as well.

Despite the importance of multidimensional access methods to emerging database applications, and the requirement to address the phantom problem in order to provide transactional access to data using these data structures, surprisingly little research exists on providing phantom protection to retrievals

*This work was supported in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003 and in part by NSF/DARPA/NASA Digital Library Initiative Program under Cooperative Agreement No. 94-11318.

[1]An exception to this is the Postgres System [6] developed at University of California at Berkeley,(and its commercial version the Informix Universal Server (IUS) [9]), which supports R-tree as an access method to optimize retrieval of multidimensional data. However, Postgres requires transactions to lock the entire R-tree thereby disallowing concurrent operations. The strategy used by IUS, if different from Postgres, is not public.

Lock Mode	IS	IX	S	SIX	X
IS	√	√	√	√	
IX	√	√			
S	√		√		
SIX	√				
X					

LOCK MODE	PURPOSE
S	Shared Access
X	Exclusive Access
IX	Intention to set shared or exclusive locks at finer granularity
IS	Intention to set shared locks at finer granularity
SIX	A course granularity shared lock with intention to set finer-granularity exclusive locks (union of S and IX)

Table 1: Lock mode compatibility matrix for granular Locks

over multidimensional data structures. The only other work we are aware of which is parallel to ours is the approach developed in [12] which uses a modified *predicate locking* mechanism to provide phantom protection over *Generalized Search Trees* (GiSTs) [8]. In contrast, the technique developed in this paper is based on *dynamic granular locking*. As discussed in [4], although the predicate locking offers potentially higher concurrency, typically the granular locking is *preferred* since the lock overhead of a predicate locking approach is much higher compared to that of a granular locking approach. We provide a detailed comparison of the granular locking approach developed in this paper to the predicate locking approach proposed in [12] in the section on related work.

The rest of the paper is developed as follows. Section 2 discusses the problem of phantoms in R-trees and the main challenges in developing a granular locking approach to solve it. In Section 3, we present the dynamic granular locking approach to prevent phantoms in R-trees. Section 4 provides a comparison to related work. Finally, section 5 offers the summary and the concluding remarks.

2 The Challenges

In this section, we discuss why the solutions to the phantom problem in B-trees (a one-dimensional access method) cannot be applied to R-trees. We then discuss the fundamental challenges in developing a granular locking approach to solve the the phantom problem in R-trees.

The phantom problem is solved in existing commercial data management systems by using *granular locking*, which is an engineering approach towards implementing predicate locks. The key idea is to divide the predicate space into a set of resource granules that may include or overlap with other resource granules. Transactions acquire locks on granules instead of on predicates. Transactions either acquire locks in *Shared* mode (S lock) or *eXclusive* mode (X lock). The locking protocol must guarantee that if two transactions request conflicting mode locks on predicates p and p' such that $p \wedge p'$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common.

An example of a granular locking approach is the *multi-granularity locking protocol* (MGL) [15]. MGL exploits additional lock modes called *intention* mode locks which represent the intention to set locks at finer granularity[15]. An intention mode lock on a node prevents other transactions from setting coarse granularity (i.e S or X) locks on that node. (see the lock compatibility matrix shown in Table 1). MGL follows the DAG locking protocol as discussed in [4].

Application of MGL to the key space associated with a B-tree is referred to as *key range locking*(KRL). In KRL, the semi-open ranges $(k_i, k_{i+1}]$, defined by the ordered list $(k_1, k_2, ..., k_n)$ of attribute values present in the B-tree such that $k_i < k_{i+1}$, serve as the lockable granules. A scan acquires locks to completely cover its query range. Similarly, a transaction that wishes to insert, delete, or update an object that lies in a given range, acquires an IX lock on the range which denotes its intention to change an object in that range. Dynamic key range schemes are more adaptive to the changes in the key space over time and provides a higher degree of concurrency. However, since the granules may dynamically change, the locking protocols are significantly more complex. Further details about granular locking and KRL can be found in [4].

KRL cannot be applied for phantom protection in R-trees since it relies on the total order over the underlying objects based on their key values. While such a total order exists for a single dimensional data, no such natural order exists over multidimensional data [22]. Imposing an artificial total order (say a Z-order [19]) over multidimensional data to adapt the key range idea for phantom protection is unnatural and will result in a scheme with a high lock overhead and a low degree of concurrency. The reason is that the protection of a multidimensional region query from phantom insertions and deletions will require accessing additional disk pages and locking objects which may not be in the region specified by the query (an object will be accessed as long as it is within the upper and the lower bounds in the region according to the superimposed total order) [14]. This severely limits the usefulness of the multidimensional access method, essentially reducing it to a single dimensional access method with the dimension being the total order. Hence new techniques need to be developed for preventing phantoms in R-trees. This paper presents a solution based on dynamic granular locking. The main challenges are:

1. Defining a set of lockable granules over the multidimensional key space such that they
- dynamically adapt to key distribution
- fully cover the entire embedded space
- are as fine as possible so as to maximize concurrency without causing high lock overhead

In KRL, the key ranges are used as lockable granules. The key ranges satisfy all the three criteria above. On the other hand, in an R-tree, the objects themselves do not fully cover the embedded space. Moreover, object ranges cannot be defined as the embedded space is multidimensional.

2. Easy mapping of a given predicate onto a set of granules that needs to be locked to scan the predicate. Subsequently, the granular locks can be set or cleared as efficiently as object locks using a standard lock manager.

3. Overlap of granules. This problem does not arise in KRL since the key ranges are always mutually disjoint. On the other hand, due to the spatial data overlap, in an R-tree (and all its variants R*-tree, R+-tree, Greene's R-tree etc., as well as other index structures like TV-trees and P-trees which partition the space based on least bounding polygons), the granules may overlap with each other. The overlapping between the granules complicates the locking protocol since a lock on a granule may not provide exclusive coverage on the entire space covered by the granule. [2]

In our approach, the lowest level bounding rectangles (BRs) are defined as the lockable granules. The granules dynamically grow and shrink with insertions into and deletions from the R-tree, thus adapting to the distribution of the objects. Since the lowest level BRs alone may not fully cover the embedded

[2]Some multidimensional index structures like K-D-B-trees and hB-trees always partition the space into spatially disjoint subspaces. As will be discussed later, the locking protocol is simpler for these index structures.

space, we define additional granules called *external* granules, one for each non-leaf node in the tree, such that the lowest level BRs (referred to as *tree* granules) together with the external granules fully cover the embedded space. The granules defined above not only cover the entire space but also adapt to the key distribution and as argued in Section 3.1, are optimized for high concurrency.

Based on the above granules, locking protocols for various operations have been developed. The developed protocols guarantee that the overlap between the granules does not cause phantoms. Finally, the number of locks acquired per operation is low - searchers need to acquire commit duration shared locks on all overlapping granules to protect the search range whereas the inserters and deleters need to acquire just one commit duration lock. The page ids of the leaf nodes (nodes representing the lowest level BRs) are the resource ids used to lock the leaf granules. For external granules, the page ids of the non-leaf nodes are the resource ids. Thus, a logical range can be easily transferred into a sequence of purely physical locks which can be set and checked very efficiently.

3 The Granular Locking Approach to Phantom Protection in R-trees

In this section, we present the dynamic granular locking approach to phantom protection in R-trees. In developing the algorithms, we assume that transactions may request the following operations over the R-tree:

- **Insert** that inserts an object into the R-tree,
- **Delete** that deletes an object from the R-tree,
- **ReadSingle** that retrieves a single specific object based on the indexed attributes,
- **ReadScan** that retrieve all objects overlapping with a given region (that is, region search),
- **UpdateSingle** that is similar to ReadSingle except that it may modify the qualified object,
- **UpdateScan** that is similar to ReadScan except that it may modify some or all of the qualifying objects.

The update operations do not modify the indexing attributes of the object since that may cause relocation of the object in the R-tree. Such an operation is modeled by the deletion of the original object followed by the insertion of the modified object into the R-tree. In presenting the solution to the phantom problem, we describe the locks that need to be acquired by transactions for each of the above operations. The exact algorithms used to acquire the necessary locks are presented elsewhere. The locking protocols assumes the presence of a the standard lock manager (LM) and that the LM supports the following lock options [17]:

- the *conditional* lock request which means that the requester is not willing to wait if the lock is not grantable immediately
- the *unconditional* lock request which means that the requester is willing to wait until the lock becomes grantable.

Furthermore, locks can be held for different durations [17]:

- *short duration* locks which are released immediately after the operation is over, typically long before the transaction termination.
- *commit duration* locks which are released only when the transaction terminates i.e. after commit or rollback is completed.

Finally, in presenting the lock requirements of various operations for phantom protection, we assume the presence of some

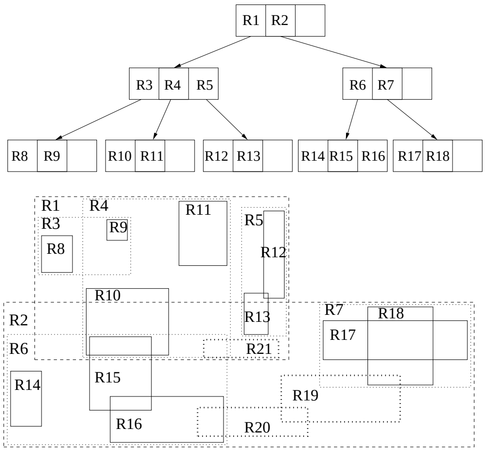

Figure 1: An R-tree along with the distribution of objects in 2 dimensional space. R3, R4, R5, R6 and R7 are the R-tree leaf granules. R19, R20 and R21 are the predicate rectangles.

protocol for ensuring the physical consistency of the tree structure in presence of concurrent operations. The approach developed in [12] can be used for this purpose. Obviously such a protocol needs to be combined with our approach to phantom protection for the complete solution of concurrency control in R-trees. Due to space limitations, we do not attempt such an integration in this paper and restrict ourselves to only addressing the problem of phantoms in R-trees.

3.1 Partitioning the Space into Granules

One of the major challenges in developing a granular locking solution to phantom problem in R-trees is to dynamically partition the embedded space into lockable granules that adapts to the distribution of objects. Moreover, to maximize concurrency, the granules should be made as fine as possible. In KRL, the semi-open key ranges are treated as lockable granules. However, key ranges cannot be defined in a multidimensional space due to the lack of ordering of the key domain. We define the *lowest level BRs* of the R-tree as the lockable granules. Each lowest level BR correspond to a leaf node of the R-tree (hence referred to as *leaf* granules) i.e. it is the smallest rectangle that covers all the objects stored in that particular leaf node. The page id of the leaf node can be used as the resource id to lock the corresponding granule. For example, in the R-tree shown in Figure 1 there are 5 such granules, one corresponding to each of the entries $R3, R4, R5, R6$, and $R7$.

If we are to use only the lowest level BRs as the lockable granules, since the R-tree partitions may not cover the entire embedded space, the set of lockable granules may not cover the embedded space. This presents a problem as the set of granules may not be able to properly protect search predicates resulting in phantoms. To see this, consider again the tree shown in Figure 1. Let transaction $t1$ request a scan on the rectangle $R19$ which only overlaps with granule $R7$. An S lock on $R7$ will not prevent the insertion of an object $R20$ by another transaction $t2$ since $R20$ does not overlap with $R7$. Insertion of $R20$, however, should not be permitted since it overlaps with the region being scanned by $t1$ (that is, $R19$). This problem would not arise if the set of granules fully covered the embedded space and $t1$ had acquired locks on enough granules to cover its search predicate.

A simple solution to cover the entire embedded space is to maintain a single extra lockable granule which covers the space that is not covered by the R-tree leaf granules. This solution is infeasible since the extra granule will become a hot spot leading to very low concurrency. Non conflicting predicates with conflicting lock modes that overlap with the space not covered by the leaf granules would not be able to execute concurrently as they would require conflicting locks on the extra granule. For example, in Figure 1, a read scan over rectangle $R19$ and an insertion of rectangle $R21$ would not be granted locks concurrently although the rectangles do not intersect.

We present a scheme that partitions the non-covered space into a set of granules referred to as *external* granules. An external granule is associated with each non-leaf node of the R-tree whose shape and size is determined by the R-tree partitionings. Let T be a non-leaf node of the R-tree and T_s be the space covered by T i.e. the space covered by the smallest rectangle that contains all the rectangles associated with its children (if T is the root node, T_s is the entire embedded space S). The external granule associated with T (denoted by $ext(T)$) is the space covered by T (that is, T_s) which is not covered by any of its children. More formally, let $E_1, E_2, ...E_n$ be the entries in T. Then, $ext(T)$ is the space $(T_s - \bigcup_{i=1}^{n} E_i.I)$. [3] The page id of the non-leaf node T itself can be is used as the resource id to lock external granule $ext(T)$. The mapping of granules to the resource ids is therefore very simple for both types of granules, which makes it easy to transfer a logical range to a sequence of purely physical locks which can be set and checked very efficiently by a standard lock manager.

In Figure 1, there are 3 external granules, one associated with each non-leaf node. The external granule associated with the root (i.e. $ext(root)$) node covers the space $S-(R1.I \cup R2.I)$ where S is the total embedded space. The external granules associated with the nodes bounded by $R1$ and $R2$ (i.e. $ext(R1)$ and $ext(R2)$) cover spaces $R1.I - (R3.I \cup R4.I \cup R5.I)$ and $R2.I - (R6.I \cup R7.I)$ respectively. Note that the external granules along with the lowest level BRs completely cover the embedded space. For example, for the R-tree in Figure 1, the union of $ext(root)$, $ext(R1)$, $ext(R2)$, $R3$, $R4$, $R5$, $R6$ and $R7$ is the entire embedded space S. [4]

Note that the partitioning strategy described above could alternatively be applied to the object level instead of the lowest level BRs. For example, for the R-tree in Figure 1, the extents of the objects $R8$ to $R18$ could be considered as the lockable granules. In that case, there would be an external granule associated with each of the index nodes $R1$ to $R7$. In this partitioning too, the set of granules cover the entire embedded space. Moreover, the granules are finer than the ones in which the lowest level BRs are used as the leaf granules and there are external granules associated with all index nodes, including leaf-level index nodes. Although the granules are finer, this scheme does not necessarily result in an improvement in concurrency. The reason is that a scan predicate typically spans more than one object extent and would thus always need to lock the external granule associated with the lowest level BR containing the object. Thus conflicting operations within the same lowest level BR would almost always conflict on the lock on the external granule associated with that BR. Thus, having finer granules would increase the lock overhead without providing much additional concurrency. Hence, we support the lowest level BRs as the leaf granules since it provides as high concurrency as supporting objects extents as granules but with considerably lower lock overhead.

3.2 Problems due to Growing Granules

Once the embedded space has been partitioned into a dynamic set of covering granules, we now discuss how the granular locking protocol can be applied to the set of granules. Following the principles of granular locking, each operation requests locks on enough granules to guarantee that any two conflicting operations request conflicting locks on at least one granule in common. One strategy is to require an insert (delete) operation to acquire IX locks on a minimal set of granules sufficient to fully cover the object (followed by an X lock on the object itself) and a scan operation to acquire S locks on all granules that overlap with the predicate being scanned. In this strategy, the insertion of an object that overlaps with the search region of a query is not permitted to execute concurrently, thereby preventing phantoms from arising. For example in Figure 1, a transaction wishing to insert rectangle $R21$ acquires IX locks on granules $ext(R1)$ and $R4$ [5]. A searcher wishing to scan predicate $R19$ acquires S locks on $ext(R2)$ and $R7$. We refer to this policy as the *cover-for-insert* and *overlap-for-search* policy. The reverse policy, namely *overlap-for-insert* and *cover-for-search*, in which IX locks are acquired on all overlapping granules for insert and delete operations and S locks on the minimal set of granules that cover the scan predicate for search could also be followed.

The lock requirements of an UpdateScan operation also depends on the policy that is followed. Since an UpdateScan requires an S as well as IX mode locks on the scanned area [15], for the first policy, it needs to acquire SIX locks on the minimal set of granules sufficient to fully cover the predicate and S locks on all the remaining granules that overlap with the predicate. In the reverse policy, an UpdateScan would acquire SIX locks on the minimal set of granules sufficient to fully cover the predicate and IX locks on all the remaining granules that overlap with the predicate. The lock requirements of the ReadSingle and UpdateSingle operations do not depend on the policy chosen. We next present a couple of examples to show that when the granules are dynamically changing due to insertions and deletions, neither of the above two policies are sufficient to prevent phantoms from arising.

First let us consider the cover-for-insert and overlap-for-search policy. In Figure 2(a), $R1$ and $R2$ are two leaf granules within the same BR R. Assume a transaction $t1$ arrives to scan the rectangle $R3$. It acquires an S lock on $R1$ and proceeds with the scan. Then another transaction $t2$ arrives to insert rectangle $R4$, acquires IX locks on $R2$ and $ext(R)$, an X lock on $R4$ and inserts $R4$. As a result, $R2$ grows to $R2'$. Then $t2$ commits and releases all its locks. Now a third transaction $t3$ arrives to insert rectangle $R5$. $R5$ is completely covered by $R2'$ and hence if the above protocol is followed, $t3$ just needs to acquire an IX lock on $R2'$. $t3$ gets the lock and proceeds with insertion which should have been prevented because if $t1$ now repeats its scan, it would find $R5$ has appeared from nowhere.

The problem in the above example arises since growth of the BR $R2$ to $R2'$ causes $t1$ to lose its S lock on part of its scan predicate $R3$ (specifically, on the region $R3 \cap R2'$). Since

[3] A non-leaf node T in an R-tree contains entries of the form *(I, child-pointer)* where *child-pointer* is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries.

[4] Note that for those index structures where it is always possible to split a node into disjoint subspaces (referred to as *space partitioning data structures*) like K-D-B-trees [20], hB-trees [3]) etc., the set of leaf granules alone cover the entire embedded space. Therefore the external granules are not required. Moreover, the granules never overlap with each other. This makes the granular locking approach much simpler to apply to space partitioning data structures.

[5] Since both $R1$ and $R2$ fully covers $R21$, either path can be selected. If $R2$ was selected, then the transaction would acquire IX locks on $ext(R2)$ and $R6$

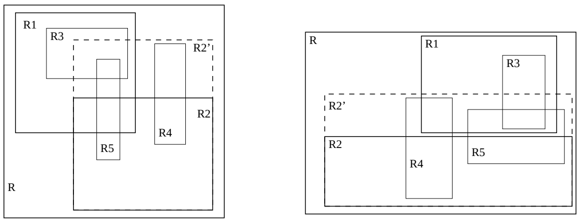

Figure 2: Insertions cause leaf granules to grow into other tree granules. (a) shows the problem with the cover-for-insert and overlap-for-search policy (b) shows the problem with overlap-for-insert and cover-for-search policy.

the locking protocol developed in the previous section follows the cover-for-insert and overlap-for-search policy, for $t1$ to have an S lock on $R3$ *after* the rectangle $R2$ had grown to $R2'$, it requires S locks on both granules $R1$ and $R2'$. An S lock on $R1$ alone does not provide the necessary protection against phantoms. Since a search operation does not cause granules to either grow or shrink, at first glance it may appear that the reverse policy in which a search acquires locks on a minimal set of granules to cover the search predicate and a insert (delete) acquires locks on all granules overlapping with the object being inserted (deleted) may resolve the problem. Indeed, if the reverse policy is used, the above discussed situation will not arise. However, even the reverse policy does not prevent phantoms from arising as shown in the next example.

In Figure 2(b), $R1$ and $R2$ are two leaf granules within the same BR R. Assume a transaction $t1$ arrives to insert rectangle $R3$. It acquires an IX lock on $R1$ as $R3$ overlaps only with $R1$, an X lock on $R3$ and proceeds with insertion. Then another transaction $t2$ arrives to insert rectangle $R4$. $R4$ overlaps with $R1$, $R2$ and $ext(R)$. So $t2$ acquires IX locks on $R1$, $R2$ and $ext(R)$, an X lock on $R4$ and inserts $R4$. As a result, $R2$ grows to $R2'$. Then $t2$ commits and releases all its locks. Now a third transaction $t3$ arrives to scan rectangle $R5$. $R5$ is completely covered by $R2'$ and hence $t3$ just needs to acquire an S lock on $R2'$. $t3$ gets the lock and proceeds with the scan which again should have been prevented because if $t1$ aborts and $t3$ repeats its scan it will find the object $R3$ has disappeared.

To identify the lock requirements of the operations when the granules can dynamically change, we need to carefully analyze how operations modify granules and how the modifications affect the lock coverage of transactions that had previously acquired locks. We conduct such an analysis for the case in which the cover-for-insert and overlap-for-search policy is used. This policy is preferred to the reverse policy (namely, cover-for-search and overlap-for-insert) for R-trees since searchers in R-trees anyway follow all overlapping paths while inserters follow single paths. A similar analysis can also be conducted and solutions be developed if the reverse policy were to be followed. In conducting the analysis, we differentiate between the leaf granules and the external granules.

The operations that may modify the current set of granules are the insert and the delete operations. In the simple case, an insertion may cause the BR of the leaf node into which the object is inserted to grow, thereby causing the corresponding leaf granule to grow. When the BRs are adjusted bottom-up, the BRs associated with the ancestors of the node into which the insertion takes place may also grow affecting the sizes of the external granules associated with those nodes. An insertion may also cause node splits which may propagate upwards causing the higher level nodes to split which may in turn affect the external granules associated with them.

Similar to an insertion, a deletion may also cause the granule associated with the leaf node from which the object is deleted as well as the external granules associated with the ancestors of the node to change. Occasionally, a deletion may cause node elimination and thus causing the granule associated with the node to disappear. As a result, the external granules associated with the ancestors of the eliminated node may also change. Furthermore, the entries of the eliminated node may get reinserted into some other part of the tree which may cause the granules associated with the nodes into which reinsertion takes place and the external granules associated with their ancestors to change.

In the next 5 subsections, we describe how each of these operations affects the lock coverage of other transactions and develop solutions to prevent phantoms that may arise due to the changes in the lock coverage.

3.3 Insertion

We first consider the case when the insertion does not cause any node split. Insertion of an object into a granule g may cause the following two things to happen:

- **Growth into Leaf Granules:** The leaf granule g grows into some other leaf granule(s) causing the overlap between the two granules to increase. For example, in Figure 2(a), the growth of $R2$ to $R2'$ increases the overlap between $R1$ and $R2$.
- **Growth into External Granules:** The granule g grows into the external granule associated with the parent of g causing the external granule to shrink in size. As the BRs are adjusted from the leaf to the root, the external granules associated with the non-immediate ancestors of g may change as well. For example, in Figure 2(a), the growth of $R2$ to $R2'$ shrinks the external granule $ext(R)$ from $R - (R1 \cup R2)$ to $R - (R1 \cup R2')$. Notice that only the external granules associated with the ancestors of the granule into which the insertion took place can change.

Both the above situations could result in phantoms. The two examples discussed earlier illustrate the problem arising due to the growth of a granule g_2 into a leaf granule g_1. Such a growth may result in a transaction holding an S lock on g_1 prior to growth of g_2 to lose its lock coverage on the portion of g_1 into which g_2 grew. The reason is that to acquire an S lock on a region, a transaction t needs to S lock all granules that overlap with the region. Since after the growth, g_2 also may overlap with the search region of t, a lock on g_1 alone may not provide the required protection from future insert or delete operations in the same region resulting in phantoms.

One approach to preventing phantoms is for all the locks held by transactions on granule g_1 to be inherited by the granule g_2 that is growing, where g_1 is a leaf granule into which g_2 grows. Even if such an approach can be made to work, it will be expensive since it requires traversing down the tree to identify the leaf granules that g_2 grew into and inheriting all the locks.

One feasible approach is to require all future inserters wishing to insert an object O that overlaps with the region of g_1 into which g_2 (e.g., $R1 \cap (R2' - R2)$ in Figure 2(a)) grew to ensure that there exists no old searchers that had acquired an S lock on g_1 prior to the growth of g_2 and which lost the coverage of their S lock due to growth of g_2. A simple way to achieve this is to require the new inserters to acquire IX locks (since they must conflict with S locks held by searchers) on all the

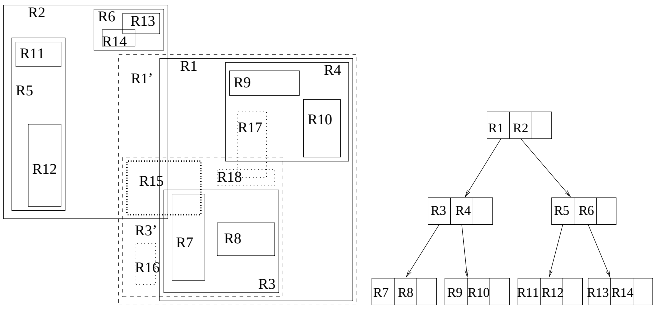

Figure 3: Insertions cause external granules to shrink. $R15$ is the object being inserted. R16, R17 and R18 are other predicate rectangles. The R-tree corresponding to the initial spatial distribution of the objects is shown alongside.

overlapping granules instead of just the minimally covering set of granules. Since the purpose of the extra locks (i.e. locks on the granules that overlap with the predicate but are not needed to minimally cover it) is to detect presence of old searchers and after the operation $g2$ alone covers O, these locks are acquired only for a *short duration* i.e. they are released at the end of the operation.

For example, in Figure 2(a), the transaction $t3$ wishing to insert $R5$ would need to acquire a short duration IX lock on $R1$ in addition to the commit duration IX lock on $R2'$ thus ensuring that the insertion takes place only after transaction $t1$ scanning $R3$ has released its S lock on $R1$, thereby preventing phantoms.

While the above discussion describes techniques to overcome problems that arise due to the growth of a granule into other leaf granules, phantoms may also arise due to growth of a granule into an external granule causing the latter to shrink. Such a situation is illustrated next.

In Figure 3, let us consider a transaction $t1$ wishing to insert rectangle $R15$ into the granule $R3$. $t1$ acquires an IX lock on $R3$ and short duration IX locks on $ext(R1)$ and $ext(R2)$ and proceeds with the insertion. The insertion causes $R3$ to grow to $R3'$ which in turn causes $R1$ to grow to $R1'$. Let us consider an old active transaction $t2$ wishing to insert rectangle $R17$ into granule $R4$. It had acquired an IX lock on $R4$ and a short duration IX lock on $ext(R1)$. Due to growth of $R3$ to $R3'$, $ext(R1)$ shrinks due to which $t2$ loses its short duration lock on $ext(R1)$. After $t1$ commits, if a searcher arrives to scan region $R18$, it will be incorrectly permitted to do so because it only needs an S lock on $R3'$. Similar to inserters, searchers also may lose their lock due to shrinkage of external granules. For example, in Figure 3, a searcher scanning region $R16$ loses its lock on $ext(root)$ when $R1$ grows to $R1'$.

To prevent loss of locks held by other transactions, an inserter must ensure that no other transaction is holding any lock on the external granule before growing into it. This can be achieved by acquiring short duration SIX locks (since they conflict with all lock modes except the IS mode which is never used by the protocol) on the external granules that change due to the insertion of the object. [6] In the above example, $t1$ should acquire a short duration SIX lock on $ext(R1)$ before adjusting $R3$ to $R3'$ and a short duration SIX lock on $ext(root)$ before adjusting $R1$ to $R1'$.

While acquiring SIX locks on the external granules prevents other transactions from losing their locks on the external granules, if the inserting transaction t itself was holding an S lock on the external granule that shrinks due to the insertion, it may lose its S lock coverage after the growth. To prevent this, t needs to acquire an S lock on the growing granule g. Acquiring an S lock on g provides t with the same lock coverage it had prior to the growth of g since the amount of coverage lost due to shrinkage of the external granule is fully covered by the growth of g.

Above, we have described how phantoms that may arise due to dynamic changes to the granules resulting from an insertion can be prevented. To insert an object, since cover-for-insert and overlap-for-search policy is used, a single commit duration lock on the granule into which the object is inserted is sufficient. After the insertion, the granule grows to cover the inserted object. All the other locks are acquired for a short duration to prevent the potential loss of locks by other operations due to the dynamic changes to the granules resulting from the insertion.

[6] The maximum number of external granules that can change is equal to the height of the R-tree minus one. The number of external granules that actually change may be less because the change may not propagate all the way up to the root.

Nature of data	Fanout	Height	Avg. Number of Disk Accesses Per Insertion (ADA)		
			Level 2	Level 3	Level 4
Point	50	3	1.567	-	-
Spatial	50	3	1.871	-	-
Point	24	4	1.154	1.938	-
Spatial	24	4	1.197	2.702	-
Point	12	5	1.012	2.144	2.640
Spatial	12	5	1.033	2.043	3.137

Table 2: Experimental results. The levels of the tree are numbered from top to bottom, the root is level 1 and the lowest index level is level h (the tree height). The ADA for levels between 2 and $h-1$ are listed (the value for the root level is 1 and there is no additional I/O at level h). The average I/O overhead per inserter for any level is the ADA for that level minus one as the inserter anyway accesses one page per level.

3.4 Overhead Optimization

In the above protocol, since inserters need to traverse all paths that overlap with the object O to be inserted to acquire short duration IX locks on all overlapping granules and O may overlap with BRs other than those in the insertion path (path from the root to the leaf into which O will get inserted), the inserter may need to pay the overhead in terms of disk I/O to access the additional disk pages. To estimate the average amount of overhead, we performed experiments on both point and spatial datasets. The point dataset consists of 32,000 uniformly distributed randomly generated points. The spatial dataset consists of 32,000 uniformly distributed randomly generated two-dimensional rectangles, the extents of the rectangles being, on average, 5% of the extent of the total region over which the rectangles are distributed along the same dimension. We calculated the average number of disk pages accessed at each level of the R-tree if the inserter is forced to follow all overlapping paths. Note that an inserter never needs to access the lowest level index nodes for acquiring the short duration locks. The results of the experiments are summarized in Table 2.

The experimental results show that for a 5-level tree, the average I/O overhead paid by an inserter by following all overlapping paths is approximately 3 disk accesses (1 additional disk access for level 3 and 2 additional disk accesses for level 4). For a 4-level tree, the overhead is approximately 1 disk I/O.

The overhead is expected to be lower with a reasonably large buffer and a frequently used R-tree since the pages corresponding to the three highest levels of the R-tree will always be kept in memory thus requiring no I/O to access them. [7] If the three highest levels are always is main memory, the inserter incurs *no* I/O overhead even for a 4-level R-tree (which can store 6.25 million objects with a fanout of 50 and 100 million with a fanout of 100). In a 5-level tree, the I/O overhead is only due to page accesses at level 4, that is, 2 extra disk accesses. Since a 5-level tree with a reasonably high fanout can index a large number of data items (312.5 million with a fanout of 50 and 10 billion with a fanout of 100), the height of an R-tree is usually never greater than 5. The inserters also need to pay some computational overhead at each node accessed to determine the paths overlapping with O.

The above discussion indicates that the granular locking approach imposes some overhead on inserters for reasonably deep trees. One approach to overcoming this overhead is to follow a modified insertion policy. In Figure 2(a), the growth of the granule $R2$ to $R2'$ caused the second inserter $t3$ to conflict with the old searcher $t1$. If an inserter, that causes a granule g to grow, itself acquires short duration IX locks on *all* granules into which it grew, thereby ensuring that there exists no old searchers which could lose their lock coverage due to the growth of g, then no future inserters, which do not cause any granule growth, need to follow all overlapping paths. Thus, *only* those inserters that change the granule boundaries need to incur the overhead of additional disk accesses and acquire the extra locks. Furthermore, even if an inserter I causes the granule g to grow, it needs to follow only those paths containing granules that overlap with the region into which g grew and have at least one active searcher. Thus, if one or more of the overlapping paths do not contain any active searchers, the inserter does not need to traverse those paths.

To verify the impact of the modified insertion policy on reducing the overhead, we implemented the policy without the optimization based on checking the presence of active searchers. [8] We performed experiments to estimate the number of inserters that change the granule boundary and hence need to pay the extra I/O overhead. The experiments demonstrate that the number of inserters that change the granule boundary depends on the fanout of the R-tree. The larger the fanout, the larger the average number of objects in a granule, the larger the average granule size, the lower the probability that an insertion changes the granule boundary and hence the lower the fraction of inserters that change the granule boundary. For both point and spatial data, about 35-38 % inserters change the granule boundary for a fanout of 12, about 15-19 % for a fanout of 24, about 6-8 % for a fanout of 50 and about 3-4 % for a fanout of 100. This implies for a tree with a reasonably large fanout, only 3-4 % of the inserters have to incur the additional overhead of traversing overlapping paths (which for a 5-level R-tree, as discussed earlier, is approximately 2 extra I/Os). Thus, the amortized overhead over all inserters is quite small. This overhead can be further reduced by using the optimization based on checking for the presence of active searchers before traversing overlapping paths.

[7] Assuming 60 transactions per second operating on the R-tree and an R-tree fanout as high as 100, the average frequency of access of the pages in the top three levels are once every 0.01667, 1.667 and 166.7 seconds respectively. According to the five-minute rule [4] of the buffer manager, such pages would be kept in memory.

[8] The implementation of this optimization is quite involved as it requires a testbed load of transactions accessing the R-tree.

3.5 Node Split

We now consider the case in which the insertion by a transaction t into an already full node causes the corresponding granule g to split into granules g_1 and g_2. Since after the split, the IX lock held by t on g is lost, t needs to acquire IX locks on g_1 and g_2 to protect the inserted object. Since t acquires an IX lock on g before the insertion, no other transaction (besides t itself) can be holding an S lock on g. If t itself was holding an S lock on g, just inheriting the S lock of t on g to g_1 and g_2 will not prevent phantoms since after the split $g_1 \cup g_2$ may not fully cover g. To cover the region originally covered by g, t will need to acquire SIX locks on g_1 and g_2 after the split as well as S locks on all additional granules that overlap with both with the predicate $g' = g - (g_1 \cup g_2)$ and the search region.The only additional lock it needs to satisfy the above requirement in an S lock on $ext(P)$ where P is the parent of g.

Since before the split the inserter acquires an IX lock on g, other inserters and deleters may also be holding IX locks on g which will be lost when g disappears. To prevent this, all transactions holding IX locks on g must acquire IX locks on g_1 and g_2 after the split This is sufficient since the insert and/or delete ranges are protected just by acquiring locks on g_1 and g_2. This situation can be avoided if the inserter acquires a short duration SIX lock instead of an IX lock on g in case it causes g to split. After the split, the inserter just needs to acquire IX locks on g_1 and g_2.

The splitting of the granule may propagate upwards causing the higher level nodes to split. When a non-leaf node N splits, the external granule associated with N shrinks in size due to which a transaction holding a lock on the external granule may lose its lock. To prevent this, transactions need to acquire short duration SIX locks on the external granules associated with the parent nodes during split propagation. Since inserters anyway acquire short duration SIX locks on the external granules associated with the ancestors of the granule in which the insertion took place while propagating the changes in BRs (even if there is no node split), no additional locks are required on the external granules when the insertion causes the node to split. However, if the transaction itself was holding an S lock on the external granule $ext(N)$ that shrinks due to the split of N to N_1 and N_2, it may lose its S lock coverage after the split. So needs to acquire S locks on $ext(N_1)$, $ext(N_2)$ and $ext(P)$ where P is the parent of N to preserve the lock coverage.

3.6 Logical Deletion

Similar to insertion, to delete an object O an IX lock on the region that covers the deleted object is required. However, unlike insertion, (in which the granule where the object is inserted grows and covers the inserted object), the granule g from which the object is deleted may shrink (to g') and hence not cover O after adjustment.

To protect the delete region, the deleter needs to acquire commit duration IX locks on the minimal set of granules $\mathcal{C}$ sufficient to fully cover the deleted object. The set $\mathcal{C}$ includes g and the minimal set of additional granules whose union fully covers the predicate $O \cap (g - g')$ (as g only covers $O \cap g'$ after the operation). The predicate may or may not be a continuous region in space. Computing $\mathcal{C}$ requires a top-down tree-traversal. Further, multiple commit duration locks need to be acquired. For this reason, we do not consider this approach any further.

Instead, deletes are performed logically. The deleter needs to acquire only a commit duration IX lock on the granule that contains the object and an X lock on the object itself. If the transaction aborts, the R-tree remains unchanged. On

Operation	Overlapping granules ($\mathcal{E}$)	Minimal covering granules ($\mathcal{C}$)	Granule g containing object	Object (O)	Others ($\mathcal{X}$)
Insert (No split or granule change)	None	None	IX	X	None
Insert (Granule change)	Short duration IX	None	IX	X	Short duration SIX on $ext(P^*)$ †
Insert (Node split)	Short duration IX	None	*Before split*: Short duration SIX on g *After split*: IX on g_1 and g_2 if no S lock on g; SIX on g_1 and g_2 and S on $ext(P)$ where P is the parent of g if S lock on g	X	Short duration SIX on $ext(P^*)$ †
Delete (Logical)	None	None	IX	X	None
Delete (Deferred)	None	None	Short duration IX if g does not become underfull Short duration SIX if g becomes underfull	X	Short duration SIX on $ext(P^*)$ and locks for reinsertion of orphan entries
ReadSingle	None	None	None	S	None
ReadScan	S	None	None	None	None
UpdateSingle	None	None	IX	X	None
UpdateScan	S	SIX	None	None for read X for update	None

Table 3: Lock protocols and lock modes required for various operations in the dynamic granular locking approach. P^* denotes all the ancestors of g whose external granule changes due to the operation. Since $g \in \mathcal{C} \subseteq \mathcal{E}$, the locking of $\mathcal{E}$ implies locking of $\mathcal{C}$ and locking of $\mathcal{C}$ implies locking of g. † If the inserter itself was holding an S lock on $ext(P)$, the growing/splitting granule(s) must inherit the lock.

the other hand, if the deleter commits, the physical deletion of the object from the R-tree is executed as a separate operation.

If the transaction requests deletion of an object that does not exist, other transaction wishing to insert the same object should be prevented as long as the deleter is active. For this purpose, the deleter acquires S locks on all overlapping granules just like a ReadScan operation with the object as the scan predicate.

3.7 Deferred Deletion

The deferred delete operation removes the logically deleted object from the R-tree and adjusts the BRs of the ancestors. To physically delete an object from a granule g, an short duration IX lock on g is acquired to prevent other searchers who may have an S lock on g from losing their lock coverage. Deletion of an entry from the node may cause the node becoming underfull in which case the node is eliminated and its entries are later reinserted. In this case, even transactions holding IX locks on g may lose their lock coverage due to elimination of g. To prevent this, if the deletion causes the leaf node to become underfull, the deleter acquires a short duration SIX lock on g (instead of a short duration IX lock). In either case, since BR of the ancestor nodes of g (granule from which the object is physically removed) needs to be adjusted, the external granules associated with the ancestors may shrink. To prevent loss of lock coverage by other transactions due to this shrinkage, similar to insert, short duration SIX locks on external granules that shrink when the BRs are adjusted are acquired.

If deletion of the object had caused the node to become underfull and the underfull node eliminated, the remaining entries of the eliminated node are reinserted. The node elimination may propagate up causing elimination of higher level nodes. In such a case, the entries of the eliminated node must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree [7, 1].

When the entries are reinserted, the lock requirements depend on whether the entry E being reinserted is a data object (if the eliminated node was a leaf node) or an index node (if the eliminated node was a non-leaf node). In both cases, similar to an ordinary insert operation, an IX lock needs to be acquired on the granule in which the reinsertion takes place (along with the short duration IX locks on all overlapping granules and short duration SIX locks on the external granules associated with the parent nodes if the reinsertion causes the granule boundary to change). If the entry E is an index node, the external granule g associated with the node into which E is inserted may shrink causing a transaction holding a lock on g to lose its lock. To ensure that no transaction loses its lock coverage, a short duration SIX lock is required on g. On the other hand, if E is an object, it will be inserted into a leaf granule which can only grow due to the insertion. Since no transaction loses its lock in this case, the SIX lock is not required for object reinsertion.

3.8 Other Operations

The ReadSingle operation acquires an S lock on the object. The ReadScan operation acquires S lock on all granules overlapping with the scan region. The UpdateSingle operation acquires an IX lock on the granule containing the object and an X lock on the object. The UpdateScan operation acquires SIX locks on the minimal set of granules covering the predicate and S locks on the remaining granules overlapping with the predicate. The lock requirements for the various operations is summarized in the Table 3.

4 Comparison to Related Work

Table 4 shows the comparison of the granular locking approach developed in this paper with the predicate locking approach proposed in [12]. In the case of B-trees, granular locking is *always better* than predicate locking as it implements predicate locks much more efficiently without imposing any additional overhead to any of the operations. On the other hand, the granular locking approach in R-trees imposes a small I/O overhead on the insertion operation. We believe that this overhead, especially when the modified insertion policy is used, is minimal compared to the cost of maintaining predicates in the predicate-based approach. A more conclusive comparison between the performance of the two approaches is possible only through extensive experimentation under varying system loads. In this paper, we have concentrated on developing the granular locking approach for R-trees. A comparative analysis between the two approaches based on empirical studies will be reported elsewhere.

Criterion	Predicate Locking Approach (PL)	Granular Locking Approach (GL)
Lock Overhead	The *average number of locks required for search is higher* compared to GL as searchers need locks on all overlapping objects.	The *average number of locks required for search is lower* compared to PL as searchers need locks on all overlapping granules.
Concurrency	Provides the *maximum permissible concurrency*	Provides *high concurrency* by choosing finest possible granules
Scalability	*Not scalable to high system loads* since under a mixed load of read-only (searchers) and read-write (updators and inserters) transactions), the *lock overhead increases along with the system load.* The reason is that inserters (and updators) needs to check the object to be inserted (or updated) against the predicates of the concurrently executing transactions for potential conflict.	*Scalable to high system loads* since the *lock overhead is independent of the number of concurrent transactions.*
Complexity of search keys	*Overhead increases with the complexity of the key space* due to the increase in the cost of checking for predicate satisfiability for complex spaces (e.g., high dimensionality, arbitrary distance metric etc.). Prohibitive cost for many GiST applications.	*Complexity of the key space poses no extra overhead* to the tree operations i.e. the operations do not need to perform any extra predicate checking for the purpose of phantom protection.
Other Overhead	*Space overhead of storing the predicates*, namely, list of predicates per transaction, list of node attachments per transaction, list of predicates attached to each node etc. Also, *overhead of predicate management*, namely, predicate attaching/detaching by searchers to/from all overlapping nodes at all levels of the tree, predicate checking and replication if necessary from parent to the child nodes during BR adjustments, similar predicate checking and replication between sibling nodes during node split	*Disk I/O overhead on some inserters* i.e. those insertions that change the granule boundaries need to make additional disk accesses to acquire short duration locks for phantom protection. Experiments show that the amortized I/O overhead over all insertions is small. No I/O overhead is imposed for space partitioning index structures as there is no overlap among granules.

Table 4: Comparison of the predicate locking approach with the granular locking approach.

5 Conclusions

Multimedia systems and other applications dealing with non-traditional data types require the support of multidimensional index structures as access paths within the database. But most of the multidimensional data structures have not been integrated to any industrial strength system. One of the primary reasons is the lack of concurrency control protocols that guarantee consistency in the presence of concurrent operations. Permitting concurrent access to data via multidimensional index structures introduces two independent concurrency control problems. Firstly, techniques must be developed to ensure the consistency of the data structure in presence of concurrent insertions, deletions and updates. Secondly, mechanisms to protect search regions from phantom insertions and deletions must be developed. This paper addresses the problem of phantom protection in multidimensional access methods which [4] has identified to be as one of the challenges to transaction management for future database systems. This problem has also been studied in [12] which proposes a predicate locking mechanism to solve the phantom problem in GiSTs. In contrast, we present a granular locking approach to phantom protection in multidimensional access methods. The granular locking approach may offer slightly lower concurrency compared to predicate locking but it is preferred to the latter as granular locks can be implemented more efficiently compared to predicate locks. Although presented in the context of R-trees, the approach developed here can be applied to other tree-based multidimensional access methods as well (both object-partitioning methods like R*-trees [1], R+-trees [23], TV-trees [13], P-trees [10] etc. as well as space-partitioning methods like K-D-B-trees [20], hB-trees [3] etc.). To the best of our knowledge, this paper provides the first solution to the phantom problem in multidimensional data structures based on granular locking.

The work in this paper can be extended in several directions. The paper solves the phantom problem using only the MGL lock modes. Higher concurrency may be achieved by exploiting enhanced lock modes as is done in [15] for KRL. Another avenue of research is integrating the approach developed here with techniques to ensure tree consistency. Such an integrated approach will form the complete solution to the concurrency control problem in multidimensional access methods.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, May 1990.

[2] D. J. DeWitt., N. Kabra, J. Luo, J. M. Patel, and J. B. Yu. Client server paradise. In *Proceedings of VLDB*, September 1994.

[3] G. Evangelidis, D. Lomet, and B. Salzberg. The hb^π-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In *Proceedings of VLDB*, 1995.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[5] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of ICDE*, 1989.

[6] The POSTGRES group. Postgres reference manual. In *Technical Report, Electronic Research Laboratory, University of California, Berkeley*, April 1994.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD, pp. 47–57.*, 1984.

[8] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees in database systems. In *Proceeding of VLDB, pages 562-573*, September 1995.

[9] Informix Software Inc. Informix universal server reference manual. 1997.

[10] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of ACM SIGMOD, pages 332-342*, May 1990.

[11] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proceedings of Very Large Databases (VLDB), pages 134-145*, September 1995.

[12] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proceedings of SIGMOD*, June 1997.

[13] H. V. Lin, K. Jagadish and C. Faloutsos. The TV-tree - an index stucture for high dimensional data. In *VLDB Journal*, 1994.

[14] D. Lomet. A review of recent work on multi-attribute access methods. In *SIGMOD Record*, Sept. 1992.

[15] D. Lomet. Key range locking strategies for improved concurrency. In *VLDB Proceedings*, August 1993.

[16] C. Mohan. ARIES/KVL: A key value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *VLDB Proceedings*, August 1990.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, Vol. 17, No. 1:94–162, March 1992.

[18] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems(TODS)*, 1984.

[19] J. Orenstein and T. Merett. A class of data structures for associative searching. In *Proc. Third SIGACT News SIGMOD Symposium on the Principles of Database Systems, pages 181-190*, 1984.

[20] J. T. Robinson. The k-d-b-tree:A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD*,1981.

[21] B. Salzberg. Grid file concurrency. In *Information and Systems*, volume 11(3): 235–244, 1986.

[22] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc, 1990.

[23] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, 1987.