SIGMOD 99

Efficient Concurrency Control in Multidimensional Access Methods *

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

Sharad Mehrotra
Department of Information and Computer Science
University of California at Irvine
sharad@ics.uci.edu

Abstract

The importance of multidimensional index structures to numerous emerging database applications is well established. However, before these index structures can be supported as access methods (AMs) in a "commercial-strength" database management system (DBMS), efficient techniques to provide transactional access to data via the index structure must be developed. Concurrent accesses to data via index structures introduce the problem of protecting ranges specified in the retrieval from phantom insertions and deletions (the *phantom problem*). This paper presents a dynamic granular locking approach to phantom protection in Generalized Search Trees (GiSTs), an index structure supporting an extensible set of queries and data types. The granular locking technique offers a high degree of concurrency and has a low lock overhead. Our experiments show that the granular locking technique (1) scales well under various system loads and (2) similar to the B-tree case, provides a significantly more efficient implementation compared to predicate locking for multidimensional AMs as well. Since a wide variety of multidimensional index structures can be implemented using GiST, the developed algorithms provide a general solution to concurrency control in multidimensional AMs. To the best of our knowledge, this paper provides the first such solution based on granular locking.

1 Introduction

Database systems are being increasingly deployed to support emerging applications such as computer-aided design (CAD), geographical information systems (GIS), multimedia content-based retrieval systems, time-series databases, medical/health care applications, spatio-temporal databases etc. To support these applications efficiently on top of a DBMS, database systems must allow application developers to (1) define their own data types and operations on those data types, and (2) define their own indexing mechanisms on the stored data which the database query optimizer can exploit to access the data efficiently. The Object Relational DBMS (ORDBMS)/ Universal Server (US) technology addresses the first problem effectively [21]. But the ability to allow application developers to easily define their own access methods (AMs) still remains an elusive goal.

The *Generalized Search Tree* (GiST) [9] addresses the above problem. GiST is an index structure that is extensible "both" in the data types it can index and in the queries it can support. It is like a "template" – the application developer can implement her own AM using GiST by simply registering a few extension methods with the DBMS. GiST solves two problems:

- Over the last few years, several multidimensional data structures have been developed for specific application domains. Implementing these data structures from scratch every time requires a significant coding effort. GiST can be adapted to work like these data structures, a much easier task than implementing the tree package from scratch.
- Since GiST is extensible, if it is supported in a DBMS, the DBMS can allow application developers to define their own AM, a task that was not possible before.

Although GiST considerably reduces the effort of integrating new AMs in DBMSs, before it can be supported in a "commercial strength" DBMS, efficient techniques to support concurrent access to data via the GiST must be developed. Developing concurrency control (CC) techniques for GiST have several important benefits. (1) Since a wide variety of index structures can be implemented using GiST, developing CC techniques in the context of GiST would solve the CC problem for multidimensional index structures in general. (2) Experience with B-trees has shown that the implementation of CC protocols requires writing complex code and accounts for a major fraction of the effort for the AM implementation [8]. Developing the protocols for GiST is particularly beneficial since it would need writing the code *only once* and would allow concurrent access to the database via *any* index structure implemented in the DBMS using GiST, thus avoiding the need to write the code for each index structure separately.

Concurrent access to data via a general index structure introduces two independent concurrency control problems:

- *Preserving consistency of the data structure* in presence of concurrent insertions, deletions and updates.
- *Protecting search regions from phantoms*

This paper addresses the problem of phantom protection in GiSTs. In our previous research, we had studied a granular locking (GL) solution for phantom protection in R-trees [4]. We refer to it as the *GL/R-tree* protocol. Due to fundamental differences between R-tree and GiST in the notion of a search key, the approach developed for R-trees is not a feasible solution for GiST. Specifically, the GL/R-tree protocol needs several modifications for making it applicable to GiSTs and

* This work was supported in part by the National Science Foundation under Grant No. IIS-9734300, in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003 and in part by NASA under Grant No. B9U415912.

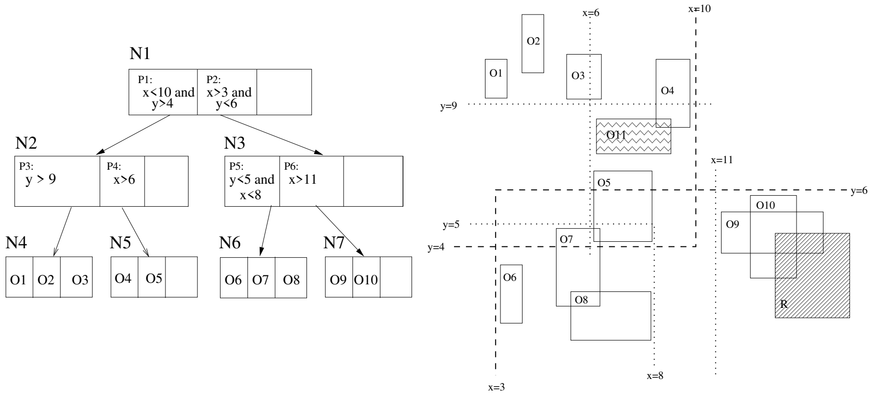


Figure 1: A GiST for a key set comprising of rectangles in 2 dimensional space. $O11$ is a new object being inserted in node $N5$. R is a search region. Predicates $P1$ through $P6$ are the BPs of the nodes $N2$ through $N7$ respectively.

the modified algorithms, when applied to GiSTs, impose a significant overhead, both in terms of disk I/O as well as computational cost, on the tree operations. To overcome this problem, we develop a new granular locking approach for phantom protection in GiSTs in this paper. We refer to it as the *GL/GiST* protocol. The GL/GiST protocol differs from the GL/R-tree protocol in its strategy to partition the predicate space and hence defines a new set of lockable resource granules. Based on the set of granules defined, lock protocols are developed for the various operations on GiSTs. For an R-tree implemented using GiST, GL/GiST protocol provides similar performance as the GL/R-tree protocol. On the other hand, for index structures where the search keys do not satisfy the *"containment hierarchy"* constraint, the GL/GiST protocol performs significantly better than the GL/R-tree protocol. Examples of such index structures include distance-based (centroid-radius based) index structures (e.g., M-tree, SS-tree). In summary, GL/GiST provides a *general solution* to concurrency control in multidimensional AMs rather than a specific solution for a particular index structure (e.g., GL/R-tree), without any compromise in performance.

The problem of phantom protection in GiSTs has previously been addressed in [10] where the authors develop a solution based on predicate locking (PL). As discussed in [8], although predicate locking offers potentially higher concurrency, typically granular locking is *preferred* since the lock overhead of predicate locking is much higher compared to that of granular locking. The reason is while granular locks can be set and cleared as efficiently as object locks ($\sim$ 200 RISC instructions), setting of a predicate lock requires checking for predicate satisfiability against the predicates of all concurrently executing transactions. For this reason, all existing commercial DBMSs implement granular locking in preference to the predicate based approach. Our experiments on various "real" multidimensional data sets show that (1) GL/GiST scales well under various system loads and (2) Similar to the B-tree case, GL provides a significantly more efficient implementation compared to PL for multidimensional AMs as well.

The rest of the paper is developed as follows. Section 2 reviews the preliminaries. Section 3 describes the space partitioning strategy for GiSTs and discusses the difficulty in applying the R-tree approach to GiSTs. Section 4 presents the dynamic granular locking approach to phantom protection in GiSTs. The experimental results are presented in Section 5. Finally, Section 6 offers the conclusions and future work.

2 Preliminaries

In this section, we first review the basic GiST structure. Next we describe the phantom problem, its solutions for B-trees and why they cannot be applied to multidimensional data structures. Finally, we state the desiderata of a granular locking solution to the phantom problem in multidimensional index structures followed by the terminology used in presenting the algorithms.

Generalized Search Trees GiST is a height balanced multi-way tree. Each tree node contains a number of node entries, $E = \langle p, ptr \rangle$, where $E.p$ is a predicate that describes the subtree pointed by $E.ptr$. If N is the node pointed by $E.ptr$, $E.p$ is defined to be the bounding predicate (BP) of N, denoted by $BP(N)$. The BP of the root node is the entire key space S. Figure 1 shows a GiST for a key space comprising of 2-d rectangles.

A key in GiST can be any arbitrary predicate. The application developer can implement her own AM by specifying the key structure via a key class. The design of the key class involves providing a set of six extension methods which are used to implement the standard *search*, *insert* and *delete* operations over the AM. A more detailed description can be found in [9].

Serializability Concepts and the Phantom Problem Transactions, locking and serializability concepts are well documented in the literature [17, 18, 8]. The phantom problem is defined as follows (from the ANSI/ISO SQL-92 specifications [12, 2]): Transaction T1 reads a set of data items satisfying some `<search condition>`. Transaction T2 then creates data items that satisfy T1's `<search condition>` and commits. If T1 then repeats its scan with the same `<search condition>`, it gets a set of data items (known as "phantoms") different from the first read. Phantoms must be prevented to guarantee serializable execution. Object level locking *does not* prevent phantoms since even if all objects currently in the database that satisfy the search predicate are locked, concurrent insertions into the search range cannot be prevented. These insertions may be a result of insertion of new objects, updates to

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. The purpose of the various lock modes are shown alongside.

existing objects or rolling-back deletions made by other concurrent transactions.

Approaches to Phantom Protection There are two general strategies to solve the phantom problem, namely *predicate locking* and its engineering approximation, *granular locking*. In predicate locking, transactions acquire locks on predicates rather than individual objects. Although predicate locking is a complete solution to the phantom problem, the cost of setting and clearing predicate locks can be high since (1) the predicates can be complex and hence checking for predicate satisfiability can be costly and (2) even if predicate satisfiability can be checked in constant time, the complexity of acquiring a predicate lock is proportional in the number of concurrent transactions which is an order of magnitude costlier compared to acquiring object locks that can be set and released in constant time [8]. In contrast, in granular locking, the predicate space is divided into a set of lockable resource granules. Transactions acquire locks on granules instead of on predicates. The locking protocol guarantees 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. Granular locks can be set and released as efficiently as object locks. For this reasons, all existing commercial DBMSs use granular locking in preference to predicate locking. A more detailed comparison between the two approaches can be found in [8].

An example of the granular locking approach is the *multi-granularity locking protocol* (MGL) [11]. MGL exploits additional lock modes called *intention* mode locks which represent the intention to set locks at finer granularity (see Table 1). Application of MGL to the key space associated with a B-tree is referred to as *key range locking*(KRL) [11, 13]. KRL cannot be applied for phantom protection in multidimensional data structures since it relies on the total order over the underlying objects based on their key values which does not exist for multidimensional data. Imposing an artificial total order (say a Z-order [16]) over multidimensional data to adapt KRL would result in a scheme with low concurrency and high lock overhead since protecting a multidimensional region query from phantom insertions and deletions will require accessing and locking objects which may not be in the region specified by the query (since 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). It would severely limit the usefulness of the multidimensional AM, essentially reducing it to a 1-d AM with the dimension being the total order.

Desiderata of the Solution Since KRL cannot be used in multidimensional index structures, new techniques need to be devised to prevent phantoms in such data structures. The principal challenges in developing a solution based on granular locking are:

- *Defining a set of lockable resource granules* [1] over the multidimensional key space such that they (1) dynamically adapt to key distribution (2) fully cover the entire embedded space and (3) are fine enough to afford high concurrency. The importance of these factors in the choice of granules has been discussed in [8]. The lock granules (i.e. key ranges) in KRL satisfy these 3 criteria.
- *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 (LM).
- *Ensuring low lock overhead* for each operation.
- *Handling overlap among granules* effectively. This problem does not arise in KRL since the key ranges are always mutually disjoint. In multidimensional key space partitioning, the set of granules defined may be, in GiST terminology, "mutually consistent". For example, there may be spatial overlap among R-tree granules. This complicates the locking protocol since a lock on a granule may not provide an "exclusive coverage" on the entire space covered by the granule. For correctness, the granular locking protocols must guarantee that any two conflicting operations will request conflicting locks on at least one granule in common. This implies that at least one of the conflicting operations must acquire locks on all granules that *overlap* with its predicate while the other must acquire conflicting locks on enough granules to fully *cover* its predicate [4]. This leads to two alternative strategies:
  - *Overlap-for-Search and Cover-for-Insert Strategy (OSCI)* in which the searchers acquire shared mode locks on all granules consistent with its search predicate whereas the inserters, deleters and updators acquire IX locks on a minimal set of granules sufficient to fully cover the object being inserted, deleted or updated.
  - *Cover-for-Search and Overlap-for-Insert Strategy (CSOI)* in which the searchers acquire shared mode locks on a minimal set of granules sufficient to fully cover its search predicate whereas the inserters, deleters and updators acquire IX locks on all granules consistent with the object being inserted, deleted or updated.

  While the former strategy favors the insert and delete operations by requiring them to do minimal tree traversal and disfavors the search operation by requiring them to traverse all consistent paths, the latter strategy does exactly the reverse. Intermediate strategies are also possible. For GL/GiST, we choose the OSCI strategy in preference to the rest. The OSCI strategy effectively does not impose *any* additional overhead on any operation as far as tree traversal is concerned since searchers in GiST anyway follow all consistent paths. The CSOI strategy may be better for index structures where inserters follow all overlapping paths and searchers follow only enough paths to cover its predicate. The R+-tree is an example of such an index structure [19]. We assume that the OSCI strategy is followed for all

[1] In this paper, we use the term "granules" to mean lock units – resources that are locked to insure isolation and not in the sense of granules in "granule graph" of MGL [8]. This is discussed in further detail in Section 4.1.

discussions in the rest of the paper.

Terminology In developing the algorithms, we assume, as in [11], that a transaction may request the following types of operations on GiST: Search, Insert, Delete, ReadSingle, UpdateSingle and UpdateScan. In presenting the solution to the phantom problem, we describe the lock requirements of each of these and present the algorithms used to acquire the necessary locks. The lock protocols assumes the presence of a standard LM which supports all the MGL locks modes (as shown in Table 1) as well as conditional and unconditional lock options [14]. Furthermore, locks can be held for different durations, namely, instant, short and commit durations [14]. While describing the lock requirements of various operations for phantom protection, we assume the presence of some protocol for preserving the physical consistency of the tree structure in presence of concurrent operations. The lock protocol presented in this paper guarantees phantom protection independent of the specific algorithm used to preserve tree consistency. In our implementation, we have combined the GL/GiST protocol with the latching protocol proposed in [10]. We do not describe the combined algorithms in this paper due to space limitations but can be found in the longer version of this paper [5].

3 Why the R-tree protocol cannot be applied to GiSTs?

The most obvious solution to the phantom problem in GiSTs is to treat GiSTs as extensible R-trees and apply the GL/R-tree protocol we developed in [4] to GiSTs. In this section, we argue that GL/R-tree protocol is not a feasible solution for GiSTs. We first briefly review the approach developed for phantom protection in R-trees [4]. We do this for two main reasons: (1) it builds the context for the solution developed for GiSTs and (2) it enables us to illustrate why GL/R-tree cannot be applied to GiSTs. Subsequently, we define the resource granules in GiST. We conclude the section by discussing why GL/R-tree is inapplicable to GiSTs.

3.1 The R-tree granular locking protocol

In GL/R-tree, we define the following two types of lockable granules:

(1) A *leaf granule* associated with each leaf level index node L of the R-tree. We denote it by $TG(L)$ i.e. the tree granule associated with the leaf node L. The *bounding rectangle (BR)* associated with L defines the lock coverage of $TG(L)$.
(2) An *external granule* associated with each non-leaf node N of the R-tree. We denote it by $ext(N)$ i.e. the external granule associated with the non-leaf node N. The lock coverage of $ext(N)$ is defined to be the space covered by the BR of N which is not covered by the BRs of any of its children.

The search operation acquires locks on all leaf granules and external granules overlapping with the search predicate (referred to as SP/R-tree).

To prevent insertion of objects into search ranges of uncommitted searchers, we follow the OSCI policy. Although the plain OSCI policy guarantees phantom protection when the operations do not change the granules, phantoms may arise when the granule boundaries dynamically change due to insertions and deletions. To prevent phantoms, inserters in GL/R-tree follows the following protocol (referred to as IP/R-tree):

Let g be the granule corresponding to the leaf node in which the insertion takes place (referred to as the *target* granule) and O be the object being inserted. IP/R-tree handles the following 2 cases separately:

- *Case 1 - Insertion does not cause g to grow*: In this case, the inserter acquires (1) a commit duration IX lock on g and (2) a commit duration X lock on O.
- *Case 2 - Insertion causes g to grow (to say, g')*: In this case, it acquires (1) a commit duration IX lock on g (2) a commit duration X lock on O and (3) short duration IX locks on *all* granules into which it grew i.e. all granules overlapping with $(g' - g)$. (3) ensures that there exists no old searchers which could lose their lock coverage due to the growth of g. Note that acquiring the extra locks of (3) may cause the inserter to perform additional disk accesses.

A detailed discussion of the lock requirements for other tree operations and the protocols followed to acquire the locks can be found in [4].

3.2 Space partitioning strategy for GiSTs

The first task in developing a granular locking solution to the phantom problem is to develop a strategy to partition the key space. Note that the BPs in GiST, unlike the BRs in R-tree, *cannot* be used to define the granules since the BPs, unlike the BRs, are *not* arranged in a *"containment hierarchy"* i.e. given a node T, for any node N *under* (i.e. reachable from) T, $BP(N) \rightarrow BP(T)$ is *not* necessarily true. So, for a search with predicate P, there might exist a leaf (or external) granule that is consistent with the search predicate P *under* a non-leaf node N whose BP is not consistent with P. For example, in Figure 1, the search predicate R is not consistent with $BP(N2)$ (i.e. $P1$) but is consistent with $TG(N5)$ (i.e. $P4$) where $N5$ lies *under* $N2$ in the tree. This means that to follow the OSCI policy (i.e. get locks on all consistent granules), the searcher cannot "prune" its search below $N2$ as it would normally do. This is impractical since the searcher would have to access extra nodes (and possibly extra disk accesses) for the purpose of getting locks.

It is clear from the above discussion that we must define granules such that their lock coverage satisfy the "containment hierarchy" constraint even if the BPs do not. For that purpose, we define a *granule predicate* associated with every index node of a GiST.

Definition 1(Granule Predicate): Let N be an index node and P be the parent of N. The *granule predicate* of N, denoted by $GP(N)$, is defined as:

$$GP(N) = BP(N) \text{ if } N \text{ is the root} \quad (1)$$
$$= BP(N) \wedge GP(P) \text{ otherwise} \quad (2)$$

Note that GPs, unlike BPs, are guaranteed to satisfy the "containment hierarchy" property.

Using GPs, we define the following two types of granules:

(1) A *leaf granule* $TG(L)$ associated with each leaf node L whose coverage is defined by GP(L). For example, in Figure 1, there are 4 leaf granules: TG(N4), TG(N5), TG(N6) and TG(N7) with lock coverage s lock coverage s $P1 \wedge P3$, $P1 \wedge P4$, $P2 \wedge P5$ and $P2 \wedge P6$ respectively
(2) An *external granule* $ext(N)$ associated with each non-leaf node N whose coverage defined as

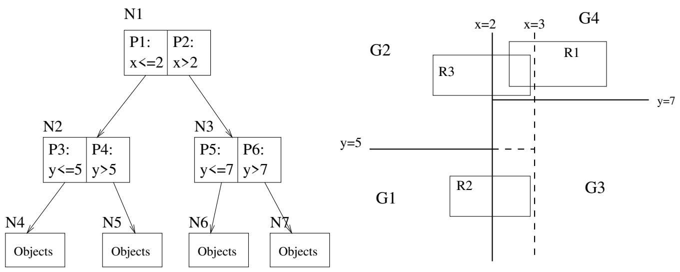


Figure 2: Insertion causes growth of tree granules that are outside the insertion path.

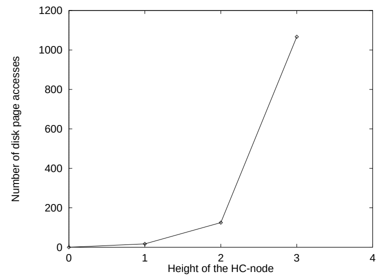


Figure 3: Increase of I/O overhead with the height of the HC-node

$(GP(N) \wedge \neg(\bigvee_{i=1}^{n} GP(Q_i)))$. where $Q_1, Q_2, ...Q_n$ are the children of N. For example, in Figure 1, there are 3 external granules: ext(N1), ext(N2) and ext(N3) will have lock coverages $S \wedge \neg(P1 \vee P2)$, $P1 \wedge \neg((P1 \wedge P3) \vee (P1 \wedge P4))$ and $P2 \wedge \neg((P2 \wedge P5) \vee (P2 \wedge P6))$ respectively.

Apart from the fact that the granules obey "containment hierarchy", the above definition has another motivation. In GiST, for any index node N, $BP(N)$ holds for each object in the subtree rooted at N. For example, in Figure 1, $P1$ holds for objects $O1, O2, O3, O4$ and $O5$ while both $P1$ and $P3$ holds for objects $O1$, $O2$ and $O3$. This implies that if an insertion does not change the BP of any node, it is guaranteed to be covered by the BP of each node in the path from the root to the leaf in which the object is being inserted. For example, in Figure 1, the object $O11$ (being inserted in node $N5$) is covered by both $P1$ and $P4$. So the leaf granule $TG(N5)$ should have lock coverage of $P1 \wedge P4$ since that is what the inserter needs for covering the object. This is exactly the definition of GP.

Having defined the new set of granules, we next try to apply GL/R-tree on GiST.

3.3 Problems in Applying GL/R-tree to GiSTs

Let us consider the GiST shown in Figure 2. There are 4 leaf granules $G1$, $G2$, $G3$ and $G4$ corresponding to nodes $N4, N5, N6$ and $N7$ with GPs $P1 \wedge P3$, $P1 \wedge P4$, $P2 \wedge P5$ and $P2 \wedge P6$ respectively. For simplicity, the partitioning of the space has been so chosen that all the external granules are empty.

Let t_s be a transaction searching region $R1$. Let t_{ins} be a new transaction that arrives to insert $R2$ into N4. After the insertion, t_{ins} updates $P1$ from $x \leq 2$ to $x \leq 3$. This causes t_s to lose it lock coverage. GL/R-tree prevents this by requiring t_{ins} to acquire locks on all granules which the target granule $G1$ has grown into. This is not sufficient for GiSTs since, unlike in R-trees, the target granule is *not* the only granule that can grow due to an insertion. For example, in Figure 2, both $G1$ and $G2$ grow due to the insertion. Assuming that only the target granule can grow can lead to phantoms. Under that assumption, t_{ins} would request a short duration IX lock on only $G3$ since that is the only granule into which $G1$ has grown, get the lock and commit. Now if t_{ins}^{new} arrives to insert $R3$ into $N2$, it would get the IX lock on $G2$ and proceed with insertion. Now if t_s repeats its scan, it would find $R3$ has arrived from nowhere. Growing of multiple leaf granules can happen in GiSTs because the lock coverage of the leaf granules, due to the definition of GP, depend of the BPs of the parents. So if an inserter modifies a node, the lock coverage of any granule under that node can possibly change. This is not possible in GL/R-tree since the lock coverage of a granule is independent of the BRs of its parent nodes.

To prevent phantoms, if the insertion changes any granule, it must acquire the following locks:
Let HC-node (Highest Changed Node) denote the the highest node in the insertion path path from root to leaf in which insertion takes place) whose BP (hence GP) changes due to the insertion. In Figure 2, $N2$ is the HC-node for the insertion of $R2$. Let G' be the new GP of HC-node after the insertion (e.g., $x \leq 3$ is the new GP of $N2$). Since *any* granule that grows due to the insertion is fully covered by G', short duration IX locks on *all* granules consistent with G' would ensure that no searcher loses its lock. In Figure 2, since all the 4 leaf granules are consistent with the predicate $x \leq 3$, t_{ins} would need to acquire short duration IX locks on $G2, G3$ and $G4$ in addition to the commit duration lock on $G1$ and X lock on $R2$. This would prevent t_{ins}(by the conflicting lock on $G4$) till t_s commits, thus preventing the phantom.

The above solution involves additional disk accesses to acquire those extra locks. In our experiments, we found that the number of disk accesses involved is significant and increases *exponentially* with the level of the HC-node. as shown in Figure 3. In general, the HC-node can be at any level of the GiST: all levels are equally likely. For the above experiment, performed on a 5-level GiST with fanout of about 100 and containing 400,000 2-d point objects, an insertion that causes a BP-change (about 6% of all insertions caused BP change) may need upto 1000 additional disk accesses to get all the locks (when the HC-node is at height 3 i.e. 3 levels above the leaf). This indicates that GL/R-tree can impose significant I/O cost for index structures where BPs do not obey "containment hierarchy" (e.g., distance-based index structures like M-tree).

Besides high cost, GL/R-tree has some other limitations for GiSTs: (1) It requires checking consistency with external granules during search, an extra task not performed by the regular GiST algorithm. This check can be computationally expensive in GiSTs. (2) It cannot allow an insertion or deletion to take place at an arbitrary level of the tree, a situation that can arise in GiSTs.

4 Phantom Protection in GiSTs

In this section, we present a dynamic granular locking approach to phantom protection in GiST. In the following subsections, we define the set of lockable resource granules for GiSTs and present lock protocols for various operations on GiSTs.

4.1 Resource granules in GiSTs

In GL/GiST, we define two types of granules:

(1) **Leaf granules:** This is the same as the previous GP-based definition of leaf granules. A leaf granule $TG(L)$ is associated with each leaf node L whose lock coverage is defined by GP(L).

(2) **Non-leaf granules:** This is a new set of granules. A *non-leaf granule* $TG(N)$ is associated with each non-leaf node N whose lock coverage, like leaf granules, is defined by $GP(N)$. In Figure 1, there are 3 non-leaf granules associated with the 3 non-leaf nodes $N1$, $N2$ and $N3$ with GPs S (entire key space), $P1$ and $P2$ respectively.

For both types of granules, the page ids of the index nodes are the resource ids used to lock the granules.

Thus, GL/GiST defines a different set of lock granules compared to those in the GL/R-tree protocol developed in [4]. External granules are no longer used as lockable granules. Non-leaf granules are used instead. There are several reasons for this choice: (1) it allows us to develop protocols that imposes absolutely no overhead (in terms of extra node accesses) on any tree operation (2) it causes almost no loss in concurrency since all commit duration locks held on non-leaf granules are *shared* mode locks (3) it has no extra computational cost since checking for consistency with non-leaf granules, unlike that with external granules, does not involve any extra checking other than what is performed anyway during the regular GiST search algorithm and (4) it allows the protocols to work even when insertions/deletions take place at arbitrary levels of the tree.

It is important to note that although non-leaf granules are introduced as lockable units, the GiST/GL protocol is completely different from and should not be confused with MGL. First, in MGL, the granules are hierarchically arranged to form a "granule graph" over which it follows the DAG protocol. In a granule graph, each node represents or "covers" a "logical" predicate. Since they are "logical", operations cannot dynamically change the predicate covered by any node in the graph. On the other hand, in GL/GiST, each node in a GiST represents a "physical" predicate: the GP of the node. Since GP is "physical" (i.e. defined based on the structure of the tree), operations (like insertions, deletions and updates) can dynamically change their lock coverages which complicates the protocol. Second, in MGL, a lock on a coarse (higher level) granule grants a certain lock coverage on the finer (lower level) granules under it. In GiST/GL, that is not the case: the higher level (non-leaf) granules are introduced in order to *cover* the entire embedded space and a lock on *does not* grant coverage on any granule under it. In summary, DAG locking and GL/GiST are *fundamentally different* protocols and serve different purposes. We believe that the idea of defining lock granules associated with non-leaf nodes is novel and, to the best of our knowledge, has been discussed before only in the context of bulk insertions in B-trees as an open problem in [8].

4.2 Search

In this section, we describe the lock protocol followed by the search operation in GiST. According to the OSCI policy, a searcher with search predicate Q acquires commit duration S mode locks on all granules consistent with Q. The concurrent search algorithm is described is Table 2.

We refer to the above lock protocol as SP/GiST (Search Protocol for GiST). SP/GiST is a straightforward protocol and does not require any modification to the basic tree-navigation algorithm of GiST. This gives rise to a possible discrepancy. Like the regular GiST search algorithm, SP/GiST uses the BPs

Algorithm Search(R, q, t)	
Input:	GiST rooted at R, predicate q, transaction t
Output:	All tuples that satisfy q
S1:	If R is root, request an S mode unconditional commit duration lock on R.
S2:	If R is non-leaf, check each entry E on R to determine whether Consistent(E,q). For each entry that is consistent, request an S mode unconditional commit duration lock on the node N referenced by $E.ptr$ and Search is invoked on the subtree rooted at N.
S3:	If R is a leaf, check each entry E on R to determine whether Consistent(E,q). If E is Consistent, it is a qualifying entry that can be returned to the calling process.

Table 2: Concurrent Search Algorithm

to do the "Consistency(E,q)" check during tree navigation. But the granules in GiST are defined in terms of the GPs. To show that SP/GiST is correct, we need to show that it guarantees that a searcher acquires locks on all the necessary granules i.e. for any index node T, *if* $GP(T) \wedge Q$ is satisfiable, *then* the searcher acquires an S lock on $TG(T)$.

To prove it, let us assume that $P_0, P_1, ..., P_m$ are the nodes in the path from the root to T where P_0 is the root and P_m is T. Since a searcher acquires a shared lock on $TG(T)$ iff it is consistent with with the BPs of all $P_i, i = [1, m]$, we need to prove that if $GP(T) \wedge Q$ is satisfiable, Q is consistent with the BP of $P_i, \forall i = [1, m]$. In other words, we need to prove that

$$GP(T) \wedge Q \text{ is satisfiable} \Rightarrow \bigwedge_{i=0}^{m} Consistent(BP(P_i), Q) \quad (3)$$

Using the definition of $GP(T)$,

$$GP(T) \wedge Q \text{ is satisfiable} \Leftrightarrow \left(\bigwedge_{i=1}^{m} BP(P_i)\right) \wedge Q\text{is satisfiable} \quad (4)$$

Since $\wedge$ is idempotent,

$$\left(\bigwedge_{i=1}^{m} BP(P_i)\right) \wedge Q \text{ is satisfiable} \Leftrightarrow \bigwedge_{i=1}^{m} (BP(P_i) \wedge Q)\text{is satisfiable} \quad (5)$$

Since $p \wedge q$ is satisfiable $\Rightarrow Consistent(p, q)$, so $\forall i, i = [1, m]$

$$(BP(P_i) \wedge Q) \text{ is satisfiable} \Rightarrow Consistent(BP(P_i), Q) \quad (6)$$

Since $(A \Rightarrow B \wedge C \Rightarrow D) \Rightarrow (A \wedge C \Rightarrow C \wedge D)$,

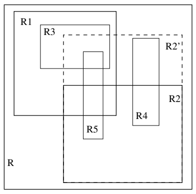
$$\bigwedge_{i=1}^{m} (BP(P_i) \wedge Q) \text{ is satisfiable} \Rightarrow \bigwedge_{i=0}^{m} Consistent(BP(P_i), Q) \quad (7)$$

Equations (4) and (7) together implies (3). ■

4.3 Insertion

The locking protocol for an insert operation must guarantee:

- *Full Coverage of the object being inserted till the time of transaction commit/rollback*: We say an object O being



1. t1 arrives to scan R3; acquires S lock on R1.
2. t2 arrives to insert R4; acquires IX locks on R2 and ext(R) and X lock on R4
3. R2 grows to R2'
4. t2 commits; releases all locks
5. t3 arrives to insert R5; acquires IX lock on R2' and X lock on R5
6. t1 repeats its scan; R4 has appeared from nowhere

Figure 4: Loss of lock coverage can cause phantoms.

inserted (deleted) is fully covered by a set of granules $\mathcal{G}$ *iff* $O \Rightarrow \bigcup_{g \in \mathcal{G}} g$. An insertion (as well as a deletion or an update) operation must acquire commit duration IX locks on $\mathcal{G}$ such that $\mathcal{G}$ *fully covers* O. Full coverage guarantees that an insertion is permitted *only if* O does not conflict with the predicate of any uncommitted searcher *assuming* that each searcher hold commit duration locks on all consistent granules.

- *Prevent Phantoms due to Loss of Lock Coverage:* Since insertions (as well deletions and updates) can dynamically modify one or more granules which in turn can affect the lock coverage of transactions holding locks on other granules, full coverage is *not* sufficient to prevent phantoms. For example, the insertion of an object O into a leaf node L of a GiST may cause the granule $TG(L)$ to grow into the search range of an old uncommitted searcher, resulting in the searcher losing its lock. This loss of lock coverage may cause future insertions, in spite of satisfying the full coverage condition, giving rise to phantoms as illustrated in Figure 4. The insertion lock protocol must prevent such phantoms from arising.

To ensure full coverage and prevention of phantoms due to loss of lock coverage, the following protocol, referred to as IP/GiST (Insert Protocol for GiST), is used.

Let O be the object being inserted and g be the target granule. We consider the following two cases:

- *Case 1 - Insertion does not cause g to grow*: In this case, the inserter acquires (1) a commit duration IX lock on g and (2) a commit duration X lock on O.
- *Case 2 - Insertion causes g to grow*: Let LU-node (Lowest Unchanged Node) denote the lowest node in the insertion path whose GP does not change due to the insertion. For example, in Figure 2, $N1$ is the LU-node for the insertion operation of $R2$. The insertion acquires (1) a commit duration IX lock on g (2) a commit duration X lock on O and (3) a *short* duration IX lock on TG(LU-node). [2] For example, in Figure 2, t_{ins} would need to acquire a short duration IX lock on $TG(N1)$ in addition to the IX lock on $TG(N4)$ and X lock on $R2$.

The concurrent insert algorithm is described in Table 3.

IP/GiST is a simple and efficient protocol since it, unlike the IP/R-tree, does not impose *any* I/O or computational overhead on the insertion operation. As a result, IP/GiST is more efficient that IP/R-tree even on R-trees. Second, unlike IP/R-tree, IP/GiST works even if the target granule is a non-leaf granule i.e. when insertion takes place at a higher level of the tree.

Algorithm Insert(R, E, l, t)	
Input:	GiST rooted at R, entry E=(p, ptr) (where p is a predicate such that p holds for all tuples reachable from ptr), level l, transaction t.
Output:	New GiST resulting from insert of E at level l
Variables:	root is global variable (const) pointing to the root node of the GiST. L is a lock initialized to NULL.
I1:	If R is not at level l, check all entries $E_i = (p_i, ptr_i)$ in R and evaluate Penalty(E_i,E) for each i. Let m be $argmin_i\ (Penalty(E_i, E))$. If ((L == NULL) $\wedge$ (Union(E.p, $E_m.p_m$) $\neq E_m.p_m$)), request a unconditional IX mode lock L on R (for short duration). Insert is invoked on the subtree rooted at the node referenced by $E_m.ptr_m$.
I2:	Otherwise (level of insertion reached), request a commit duration unconditional IX lock on R and a commit duration unconditional X lock on $E.ptr$. If there is room for E on R, install E on R. Otherwise invoke Split(root, R, E, t).
I3:	AdjustKeys(root, R, t).
I4:	If $L \neq NULL$, release L.

Table 3: Concurrent Insert Algorithm

Now we show that IP/GiST satisfy the above requirements of correctness. First, we prove full coverage. In Case 1, g fully covers O, so commit duration IX lock on g ensures full coverage. In Case 2, at the start of the operation, g does not fully cover O but TG(LU-node) does. So full coverage is provided by the sequence of 2 locks: (1) the short duration IX lock on TG(LU-node) from the beginning of the operation till the end of the operation [3] (2) the commit duration IX lock on g from the end of the operation till the end of the transaction (since g has already grown to accommodate O).

Next we show prevention of phantoms due to loss of lock coverage. In Case 1, there can be no loss of lock coverage of any searcher. In Case 2, the short duration IX lock on TG(LU-node) guarantees that no searcher can lose it lock coverage. Let us first consider a searcher t_s already executing when the inserter t_{ins} arrives to insert O. Let Q be the search predicate of t_s. Let h be a granule that grows to h' due to the insertion of O. t_s can lose its lock *iff* $h \wedge Q$ is not satisfiable but $h' \wedge Q$ is satisfiable. From the definition of LU-node, $h' \Rightarrow$ TG(LU-node). $(h' \wedge Q)$ is satisfiable *and* ($h' \Rightarrow$ TG(LU-node)) imply (TG(LU-node) $\wedge Q$) is satisfiable which in turn implies *Consistent(TG(LU-node), Q)*. This means that t_s can lose it's lock coverage *iff* it has an S lock on TG(LU-node) (since searcher acquires S locks on all consistent granules). Thus, the IX lock requirement on TG(LU-node) prevents any searcher from losing its lock coverage. The IX lock on TG(LU-node), being a short duration lock, would prevent any loss of lock by even those searchers that arrive during the operation. Any searcher that arrives after the completion of the insertion operation cannot lose its lock coverage due to the insertion.

[2] The short duration IX lock can be released immediately if the AdjustKeys operation is performed right away i.e. in a top-down fashion rather than bottom-up as is done in GiSTs. This would avoid holding the lock across I/O operations.

[3] Note that this the best we can do since, at this point of time, TG(LU-node) is the *smallest* granule in the insertion path that *fully covers* O.

Operation	Lock Requirements	Other Actions
Insertion(no granule change /no node split)	Commit dur. IX on g; Commit dur. X on O	None
Insertion (granule change)	Short dur. IX on TG(LU-node); IX on g; X on O	None
Insert (node split)	If T is leaf : Instant dur. SIX on $TG(T)$ before split; IX on either $TG(T)$ or $TG(TT)$, whichever contains O after split If T is non-leaf : Instant dur. SIX on $TG(T)$;	Inherit S locks to $TG(TT)$ if itself holding S lock on $TG(T)$
Search	S on all consistent leaf and non-leaf granules	None
Delete (Logical)	IX on g; X on O	Mark O deleted; Remove O from page
Delete (Deferred)	If node is not empty: Short dur. IX on TG(HC-node); IX on g; X on O. If becomes empty: If T is leaf, Short dur. SIX on TG(T); If T is non-leaf , Short dur. IX on TG(T)	Eliminate node if empty
ReadSingle	S on O	None
UpdateSingle	If no indexed attribute changed: IX on g; X on O Otherwise: Delete O; Insert modified O	None
UpdateScan	S on all consistent granules; For every individual object updated, same requirement as UpdateSingle	None

Table 4: Lock requirements for various operations in the dynamic granular locking approach. g is the target granule for insertion/deletion, O is the object being inserted/deleted/updated.

4.4 Node Split

We now consider the special case where the insertion by a transaction t into an already full node causes the target granule g to split into granules g_1 and g_2. Insertions causing node splits follow the IP/GiST except that it needs to acquire some additional locks when it causes the splits.

If the insertion by t causes g to split, since the IX lock held by t on g is lost after the split, 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 holds an S lock on g, it needs to inherit its S lock on g to g_1 and g_2. This is because g_1 and g_2 are the only additional granules that may become consistent with the search predicate of t due to the split.

Since before the split the inserter acquires an IX lock on g, other inserters and deleters may also be holding IX locks on g. When g splits, all transactions holding IX locks on g must acquire IX locks on g_1 and g_2 after the split. This is sufficient as all the insert and/or delete ranges (logical deletion) is guaranteed to be protected by the IX locks on g_1 and g_2 since all objects in g will be either in g_1 or g_2. It may not possible for t to change lock requests of other transactions using a standard lock manager. The problem can be avoided if the inserter acquires a instant duration SIX lock on g in case it causes g to split. After the split, the inserter acquires a commit duration IX lock on either g_1 or g_2, whichever contains O.

The splitting of the granule may propagate upwards causing the non-leaf nodes to split. As in the case of leaf node split, the transaction causing a non-leaf node N to split acquires a instant duration SIX lock on $TG(N)$ to prevent any other transaction losing its lock. If t itself was holding an S lock on $TG(N)$, it needs to inherit its S lock on the two granules formed after split.

The node split operation can be allowed to be carried out "asynchronously". This requires maintaining the information of an "outstanding split" in the node - the transaction can subsequently commit while a separate transaction executes the split operation later by checking the "outstanding split" flags. The lock requirements remain the same as in the "synchronous" case.

4.5 Deletion

Similar to insertion, to delete an object O, the deleter requires an IX lock on the region that covers O. However, unlike insertion, (in which the granule where the object is inserted grows and covers the inserted object), the granule g from which O is deleted may shrink due to the deletion and may not cover O. To protect the delete region, the deleter would need a *commit duration* IX lock on TG(LU-node) (here it is the LU-node of the deletion of operation) since TG(LU-node) is the smallest granule to fully cover O at the completion of the deletion operation. This would result in low concurrency since a large number of searchers may be unnecessarily prevented till the deleter commits. For this reason, we do not consider this approach any further. Instead, deletes are performed logically. We present the lock needs of the logical and physical deletions in the following subsections.

4.5.1 Logical Deletion

The logical deleter needs to acquire a commit duration IX lock on only the leaf granule g that contains the object and an X lock on O itself. The IX lock on g is sufficient to cover O since even if the GP of g changes due to other insertions and deletions (physical) since g would still cover O. Subsequently, it removes the object from the page and marks it as deleted. If the transaction aborts, the changes are undone, the delete mark is removed and the locks are released. On the other hand, if it commits, the physical deletion of O from the GiST is executed as a separate operation.

If the transaction requests deletion of an object O that does not exist, other transactions 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 consistent granules just like a search operation with O as the search predicate.

4.5.2 Deferred (Physical) Deletion

The deferred delete operation removes the logically deleted object from the GiST and adjusts the BPs of the ancestors. To physically delete an object from a granule g, a short duration IX lock on g is acquired to prevent other searchers having S locks on g from losing their lock coverage. The IX lock is sufficient as inserters and other deleters holding locks on g would not lose the necessary lock coverage even after g shrinks due to the physical deletion. Deletion of an entry from the node may also result in the node becoming empty in which case it is eliminated from the GiST. Since a node is eliminated only when it becomes empty, no transaction can lose its IX lock due to elimination of g as g does not cover any object. So the IX lock on g is sufficient even if the deletion causes the elimination of the node.

In either case, since the change of g may propagate upwards causing BPs of the ancestor nodes to change, the non-leaf granules associated with the ancestors may shrink. Since only searchers hold locks on non-leaf granules (inserters request only instant-duration locks), only searchers can lose their lock coverage due to this shrinkage. Note that only the searchers whose predicates are consistent with the HC-node (i.e. the highest index node in the deletion path whose BP changes due to the deletion) can lose lock coverage, possibly giving rise to phantoms. The loss of lock coverage of the searchers can be prevented by acquiring a short duration IX lock on TG(HC-node). Note that for insertion, it was the TG(LU-node) on which the short duration IX lock had to be acquired. The difference comes from the fact that insertion causes granules to grow while deletion causes them to shrink.

4.6 Other Operations

The locks needs for the other operations are:

- The *ReadSingle* operation just acquires an S lock on the object.
- The *UpdateSingle* operation, if none of the attributes indexed by GiST are changed, just needs an IX lock on the granule containing the object and an X lock on the object. Otherwise, it first executes a deletion operation of the object to be updated followed by the insertion of the updated object obeying the respective lock protocols.
- The *UpdateScan* operation acquires S locks on all consistent granules just like a Search operation. For every individual object O updated, it requires the same locks as an UpdateSingle operation on O.

The lock requirements for the various operations is shown in the Table 4.

5 Experimental Evaluation

We performed several experiments to (1) evaluate the performance of the GL/GiST protocol under various degrees of system loads and (2) compare it with other protocols in terms of concurrency and lock overhead. In this section, we discuss our implementation of the protocols followed by the performance results.

5.1 Implementation

Implementation of the Protocols We implemented the complete GL/GiST protocol as described in this paper. To evaluate the performance of the GL/GiST protocol, we also implemented the pure predicate locking (referred to as the *PurePL* protocol) to serve as the baseline case. In PurePL, each search operation checks its predicate against the objects of the insert/delete/update operations of all currently executing transactions. If there is any conflict, it blocks on that transaction by requesting an S lock on that transaction ID, assuming that every transaction acquires an X lock on its own ID when it starts up. Otherwise it proceeds with the search. Similarly, each insert/delete/update operation checks its object against the predicates of the search operations of all currently executing transactions and in case of a conflict, blocks on the conflicting transaction.

Parameters	Meaning
MPL	multiprogramming level
Transaction Size	the number of operations per transaction
Write Probability	the fraction of operations in a transaction that are writes (i.e. inserts)
Query Size	the average selectivity of a search operation
External Think Time	mean time between transactions
Restart Delay	mean time after which an aborted transaction is restarted

Table 5: Workload Parameters

Construction of GiST We conducted our experiments on two different GiSTs constructed over the following two datasets:

- The **2-d dataset:** is the 2-d point data set of the Sequoia 2000 benchmark [20]. It contains locations(easting and northing values) of 62,556 California places extracted from the US Geological Survey's Geographic Names Information System (GNIS)). The points are geographically distributed over a 1046km by 1317km area.
- The **3-d dataset:** is derived from the FOURIER dataset [6]. The FOURIER dataset data set comprises of 1.2 million vectors of fourier coefficients produced by fourier transformation of polygons. We constructed the 3-d dataset by taking the first 3 fourier coefficients of each vector.

We set aside some points (by random choice) from the above data files for insertion into the GiST during the run of transactions. The searches to be executed during the run are generated by randomly choosing the query anchor from the data file and generating a bounding box by choosing a proper side length needed to obtain desired search selectivity. The set-aside points and the queries are stored in two separate files which are used by the workload generator.

We created the GiSTs by bulkloading the remaining points. The two GiSTs are described below:

- *2-d GiST:* constructed on 56,655 2-d points with 2K page size (fanout 102, 821 nodes). Since the size of the data set is small, we use a comparatively small page size to make the GiST of significant size.
- *3-d GiST:* constructed on 480,471 3-d points with 8K page size (fanout 292, 2360 nodes)

In both cases, we configured the GiST to behave as an R-tree by specifying the extension methods appropriately.

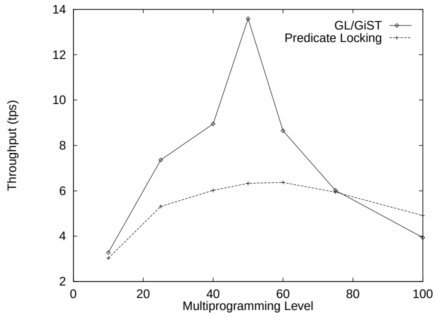


Figure 5: Throughput at various MPLs for 2-d data (write probability=0.2, transaction size=10, query selectivity=0.1%)

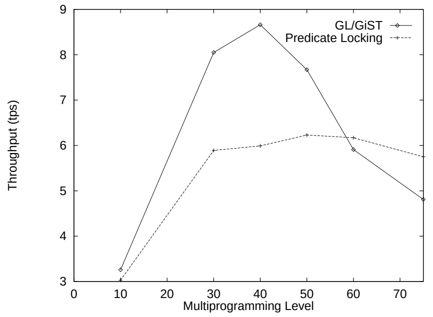


Figure 6: Throughput at various MPLs for 3-d data (write probability=0.2, transaction size=10, query selectivity=0.05%)

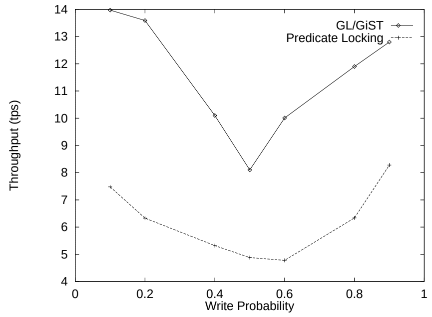


Figure 7: Throughput at various mixes of reads and writes (MPL=50, transaction size=10, query selectivity=0.1%)

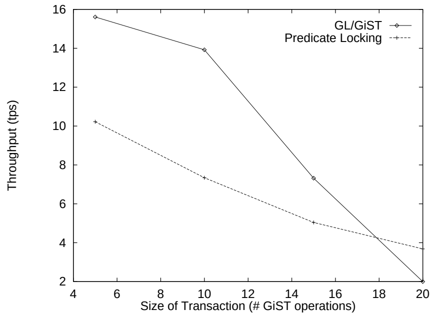


Figure 8: Throughput at various transaction sizes (MPL=50, write probability=0.1, query selectivity=0.1%)

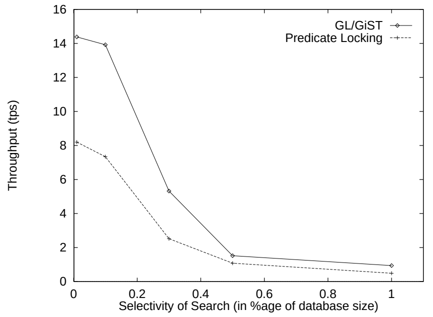


Figure 9: Throughput at various query sizes (MPL=50, transaction size=10, write probability=0.1)

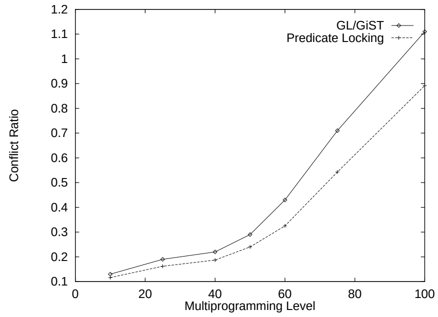


Figure 10: Conflict Ratio (transaction size=10, write probability=0.2, query selectivity=0.1%)

Workload Generator and the Lock Manager The workload generator (WG) generates a workload based on the input parameters shown in Table 5. The WG assigns some search operations (from the bounding box query file) and some insertion operations (from the set-aside point file) to each transaction. Each transaction executes as a separate thread. We use the Pthread library (Solaris 2.6 implementation) for creating and managing the threads [15]. One thread only executes one transaction: it is created at the beginning of the transaction and is terminated when the latter commits. The WG maintains the MPL at the specified value by using an array of flags (MPL number of them): when a thread finishes, it sets a flag. The main WG thread constantly polls on this array and when it detects the setting of a flag, it starts a new thread and assigns the next transaction to it. The thread waits for some time (external think time) and starts executing the transaction: it executes one operation after another on the GiST following the lock protocols. If any lock request returns an error (due to a deadlock or a timeout), the transaction aborts. If it aborts, it is re-executed within the same thread after a certain restart delay (each transaction remembers its constituent operations till it commits for possible re-execution). Our implementation of the WG consists of 3 main C++ classes (TransactionManager, Transaction and Operation). The TransactionManager class also maintains the global statistics of the run (e.g., throughput, conflict-ratio, number of locks acquired, number of aborts etc.) which are used to measure the performance of the various protocols. Although the other 4 simulation parameters are varied, we fix the external think time to 3 seconds and the restart delay to 3 seconds for all our experiments. Also, for the two GiSTs, the buffer sizes are set such that about 75% of the pages fit in memory.

For the lock manager (LM) implementation, we reused most of the LM code of MiniRel system obtained from the University of Maryland. The LM code closely follows the description in [8].

All experiments were performed on a Sun Ultra Enterprise 3000 Server running Solaris 2.6 with two 167MHz CPU, 512MB of physical memory and several GB of secondary storage.

5.2 Experimental Results

Evaluation of the GL/GiST protocol We conducted experiments to evaluate the performance of the GL/GiST protocol under various system loads. Performance is measured using throughput i.e. the ratio of the total number of transactions that completed during the period when the transactions ran at full MPL (ignoring the starting phase and the dying phase when the MPLs are lower) to the total duration of the full-MPL phase [1]. Figures 5 shows the throughput of GL/GiST and PurePL protocols at various MPLs for the 2d dataset. Initially, the throughput increases with the MPL as the system resources were underutilized at low MPLs. For GL/GiST, the throughput reaches a peak (~ 14 tps) at an MPL of 50 while for PurePL, the peak (~ 6 tps) is reached at an MPL of 60. Beyond that point, the throughput starts decreasing as the system starts thrashing. Figures 6 shows the performance of the two protocols for the 3d dataset. Like the 2-d dataset, the GL/GiST achieves significantly higher throughput compared to PurePL.

We also varied the system load by tweaking the other parameters like write probability, transaction size and size of search [1].

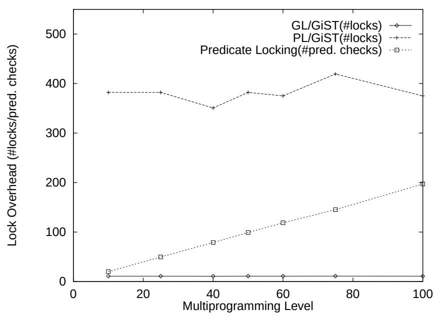


Figure 11: Lock Overhead of Search Operation (transaction size=10, write probability=0.2, query selectivity=0.1%)

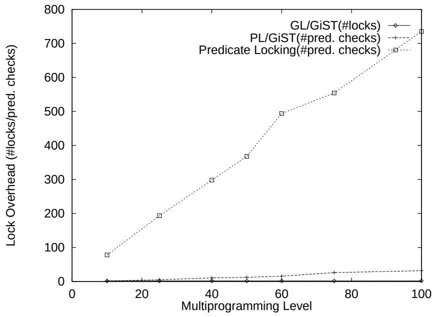


Figure 12: Lock Overhead of Insert Operation (transaction size=10, write probability=0.2, query selectivity=0.1%)

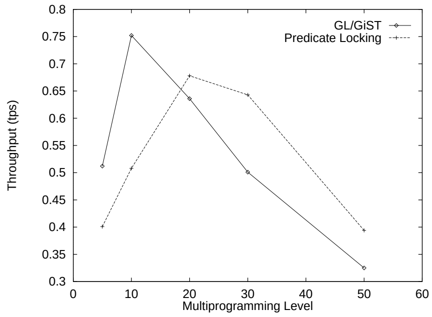


Figure 13: Throughput at various MPLs for 5-d data (write probability=0.1, transaction size=10, query selectivity=0.1%)

These experiments were conducted on the 2-d dataset. Figure 7 shows the performance of the two protocols under various mixes of read(search) and write(insert) operations. GL/GiST significantly outperforms PurePL under all workloads. Figure 8 shows the throughputs at various transaction sizes. Again, GL/GiST mostly outperforms PurePL. At an MPL of 50, for transactions with 20 or more operations, since a large portion of the GiST is locked by some transaction or the other, GL/GiST starts thrashing due to high lock contention leading to decrease in throughput. Figure 9 shows the performance for various query sizes. Once again, GL/GiST performs better than PL for all workloads.

Comparison to other techniques In this section, we compare GL/GiST protocol with the predicate locking protocol presented in [10]. We refer to the above protocol as the PL/GiST protocol. In PL/GiST, a searcher attaches its search predicate Q to all the index nodes whose BPs are consistent with Q. Subsequently, the searcher acquires S locks on all objects consistent with Q. An inserter checks the object to be inserted against all the predicates attached to the node in which the insertion takes place. If it conflicts with any of them, the inserter also attaches its predicate to the node (to prevent starvation) and waits for the conflicting transactions to commit. If the insertion causes a BP of a node N to grow, the predicate attachments of the parent of N is checked with new BP of N and are replicated at N if necessary. The process is carried out top-down over the entire path where node BP adjustments take place. Similar predicate checking and replication is done between sibling nodes during split propagation. The details of the protocol can be found in [10]. A complete performance study would require a full fledged implementation of the PL/GiST protocol (including implementation of the Predicate Manager, augment GiST with data structures to be able to attach/detach predicates to tree nodes etc.). Due to the complexity of the this task, we only compare the two protocols in terms of the degrees of concurrency offered and their lock overheads. Again PurePL is used to serve as the baseline case. All the experiments were conducted on the 2-d dataset.

Figure 10 compares the concurrency offered by the GL/GiST and the PL protocols. Concurrency is measured using conflict ratio i.e. the average number of times some transaction blocked on a lock request per committed transaction [1]. Lower the conflict ratio, higher the concurrency. Both PL/GiST and PurePL protocols offer the maximum permissible concurrency since transactions are blocked only when they truly conflict. On the other hand, GL/GiST offers lower concurrency due to "false conflicts" i.e. a situation where although the predicates do not conflict with each other, they end up requesting conflicting locks on the same granule (e.g., in R-trees, a search predicate and an object being inserted do not overlap with each other but they overlap with the BR of the same leaf node). More the number of false conflicts, higher the loss of concurrency. Figure 10 shows that false conflicts do not cause a significant loss of concurrency in GL/GiST compared to PL. This is an outcome of the "fineness" of the chosen granules.

Figure 11 and 12 shows the lock overheads imposed by the GL/GiST, PL/GiST and PurePL protocols for the search and insert operations respectively. The lock overhead is measured by the average number of locks acquired or the average number of predicate checks performed, as the case may be, measured on the same scale. Although the two costs (i.e. acquiring a lock and performing a predicate check) are within the same order of magnitude (between 50-200 RISC instructions) for 2d data, the costs would differ for higher dimensional data (predicate checking becomes costlier while the cost of acquiring a lock remains the same). While the lock overhead of predicate locking increases linearly with MPL, that of GL is independent of MPL. The figures show that for both search and insert operations, GL/GiST imposes considerably lower lock overhead compared to PL protocols.

To study the performance of GL at higher dimensionalities, we also conducted experiments on 5-d data. The 5-d dataset is derived from the FOURIER dataset and is constructed by taking the first 5 fourier coefficients of each vector. We built the GiST on 480,471 points of the 5-d dataset with 8K page size(fanout 136, 5186 nodes). The buffer size was set to about 10% of the size of the GiST. Figure 13 shows the performance the two approaches at various MPLs for 5-d data. Like 2-d and 3-d datasets, granular locking outperforms predicate locking for 5-d data as well.

In summary, there is a tradeoff between GL and PL – while GL enjoys lower lock overhead, it has lower concurrency compared to PL. Our experiments confirm that similar to granule based protocols for 1-d datasets, the GL protocol performs significantly better than PL for multidimensional datasets as well.

6 Conclusions and Future Work

Numerous emerging applications (e.g., GIS, multimedia, CAD) need support of multidimensional AMs in DBMSs. The

Generalized Search Tree (GiST) is an important step to meet that need. GiST, being an extensible index structure, when supported in a DBMS, will allow application developers to define their own AMs by supplying a set of extension methods. However, before GiSTs can be supported by any commercial strength DBMSs, efficient techniques to support concurrent access to data via the GiST must be developed. Concurrent access to data via a general index structure introduces two independent concurrency control problems. First, techniques must be developed to ensure the consistency of the data structure in presence of concurrent insertions, deletions and updates. Second, mechanisms to protect search regions from phantom insertions and deletions must be developed. Developing such mechanisms to guarantee transactional access to data via multidimensional data structures has been identified as one of the key challenges to transaction management in future database systems [8].

This paper presents a dynamic granular locking approach to phantom protection in GiSTs. The paper builds on our previous work on a dynamic granular locking strategy for R-trees [4]. Due to some fundamental differences between R-tree and GiST in the notion of a search key, the algorithms developed for R-trees do not provide a feasible solution for phantom protection in GiST. Motivated by the limitations of the previous approach in the context of GiSTs, we develop a new granular locking approach suited for concurrency control in GiSTs. The developed protocols provide a high degree of concurrency and have low lock overhead. Our experiments have shown that the granular locking technique (1) scales well under various system loads and (2) significantly outperforms predicate locking for low to medium dimensional datasets (2d, 3d and 5d). While most applications that involve dynamic datasets and require highly concurrent accesses to the data deal with low to medium dimensional spaces, [4] it is nevertheless interesting to explore approaches that provide good performance for high dimensional datasets as well. Although the granular locking proposed in this paper provides almost as high concurrency as the predicate locking approach for low to medium dimensionalities (see Figure 10), the loss of concurrency increases with the increase in dimensionality. The reason is that at high dimensionalities, the data space gets increasingly sparse (a phenomenon commonly known as the "dimensionality curse" [3]), resulting in coarser leaf granules which causes more "false conflicts" and hence a higher loss in concurrency. While at low to medium dimensionalities the efficiency of granular locking far outweighs the loss of concurrency resulting in better performance compared to predicate locking, it may not be the case at high dimensionalities. This is evidenced by the fact that for 5-d data, though granular locking still outperforms predicate locking, the performance gap between them is less compared to the 2-d and 3-d datasets. A simple approach to improve the concurrency offered by granular locking is to define finer granules. The benefit of such an approach is not clear since while the finer granules will improve concurrency, it will also increase the lock overhead of each operation. A hybrid strategy between the granular and predicate locking techniques may be a more suitable solution for high dimensional datasets. We intend to explore such a solution in the future.

[4] For example, GIS and CAD systems deals with spatial data which is either 2-d or 3-d. Spatio-temporal applications (e.g., management of moving objects) deals with 3-d or 4-d data. Multimedia retrieval systems like QBIC index images using 3-d feature vectors [7].

7 Acknowledgements

We wish to thank Mike Franklin for providing us with the MINIREL code. We would like to thank Stefan Berchtold for giving us the FOURIER dataset. We obtained the 2-d dataset from the Sequoia 2000 Project's FTP site (http://s2k-ftp.cs.berkeley.edu:8000/sequoia/benchmark/). Finally, we thank the reviewers for their comments.

References

[1] R. Agrawal, M. Carey, and M. Livny. Models for studying concurrency control performance: Alternatives and implications. In *SIGMOD*, May 1985.

[2] ANSI. Ansi x3.135-1992, american national standard for information systems - database language - sql. November, 1992.

[3] S. Berchtold, C. Bohm, D. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high dimensional data spaces. *PODS*, 1997.

[4] K. Chakrabarti and S. Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, February 1998.

[5] K. Chakrabarti and S. Mehrotra. Efficient concurrency control in multidimensional access methods. *Technical Report TR-MARS-97-12, Department of Computer Science, University of Illinois*, October 1998.

[6] K. Chakrabarti and S. Mehrotra. High dimensional feature indexing using hybrid trees. *Proc. of the 15th IEEE International Conference on Data Engineering (ICDE)*, March 1999.

[7] C. Faloutsos and et. al. Efficient and effective querying by image content. In *Journal of Intell. Inf. Systems*, July 1994.

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

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

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

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

[12] J. Melton and A. R. Simon. Understanding the new sql: A complete guide. *Morgan Kauffman*, 1993.

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

[14] 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.

[15] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996.

[16] 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.

[17] V. H. P. A. Bernstein and N. Goodman. Concurrency control and recovery in database systems. *Addison Wesley*, 1987.

[18] C. Papadimitriou. The theory of database concurrency control. *Computer Science Press*, 1986.

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

[20] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The sequoia 2000 storage benchmark. *Proc. of SIGMOD*, 1993.

[21] M. Stonebraker and D. Moore. Object-relational dbmss: The next great wave. *The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor*, 1996.