Phantom Protection Method for Multi-dimensional Index Structures

Seok Jae Lee
School of Electrical&Computer Engineering
Chungbuk National Univ., Cheongju, Korea

Seok Il Song*
Dept. of Computer Engineering
Chungju National Univ., Chungju, Korea

Jae Soo Yoo
School of Electrical&Computer Engineering
Chungbuk National Univ., Cheongju, Korea

ABSTRACT

Emerging modern database applications require multi-dimensional index structures to provide high performance for data retrieval. In order for a multi-dimensional index structure to be integrated into a commercial database system, efficient techniques that provide transactional access to data through this index structure are necessary. The techniques must support all degrees of isolation offered by the database system. Especially degree 3 isolation, called "no phantom read," protects search ranges from concurrent insertions and the rollbacks of deletions. In this paper, we propose a new phantom protection method for multi-dimensional index structures that uses a multi-level grid technique. The proposed mechanism is independent of the type of the multi-dimensional index structure, i.e., it can be applied to all types of index structures such as tree-based, file-based, and hash-based index structures. In addition, it has a low development cost and achieves high concurrency with a low lock overhead. It is shown through various experiments that the proposed method outperforms existing phantom protection methods for multi-dimensional index structures.

***Keywords**: Concurrency Control, Phantom Protection, Multidimensional Index.*

1. INTRODUCTION

In the past couple of decades, applications that use modern databases, such as geographic information systems (GIS), mobile location services (MLS), computer-aided design (CAD), medical image repositories and multimedia applications, have been developed. The applications commonly are required to manipulate multi-dimensional data. For example, GISs store and retrieve two-dimensional geographic data about various types of objects such as a building, a river, a city, and so on. MLS systems provide clients with the current locations of moving objects such as mobile phones. The locations of moving objects are represented as points in a two-dimensional space.

To help satisfy the requirements of these database applications, various multi-dimensional index structures have been proposed. There are space-partitioning methods like Grid-file [1], K-D-B-tree [2], and quad-tree [3] that divide the data space along predefined or predetermined lines regardless of the data distributions. On the other hand, methods such as R-tree [1], R+-tree [4], R*-tree [5], X-tree [6], SR-tree [7], M-tree [8], TV-tree [9] and CIR-tree [10] are data partitioning index structures that divide the data space according to the distribution of data objects inserted or loaded into the tree. Hybrid-tree [11] is a hybrid approach of data partitioning and space partitioning methods, The VA-file [12] uses a flat file structure and [13] uses hashing techniques.

In order for multi-dimensional index structures to support modern database applications, they need to be integrated into existing database systems. Even though integration is an important and practical issue, not much previous research exists regarding this issue. To integrate an access method into a DBMS, we must consider problems such as concurrency control and recovery. The concurrency control mechanism has two independent challenges. First, techniques must be developed to ensure the consistency of the data structure in presence of concurrent insertions, deletions, and updates. Second, phantom protection methods that protect searchers' predicates from subsequent insertions, and the rollbacks of deletions before the searchers commit must be developed [14], [15]. Database systems offer transactional isolation besides

* *Corresponding author. E-mail : sisong@chungju.ac.kr*
Manuscript received May 14, 2007 ; accepted June 11, 2007

the maintenance of the physical consistency of index structures. The ISO and ANSI SQL standards mandate true isolation as the default. However, most DBMSs support four degrees of isolation such as dirty read, committed read, repeatable read, and no phantom read. They allow SQL users and application programmers to choose a proper isolation degree for their application domain for performance reasons. On the first issue, which is the maintenance of physical consistency of index structures, several methods that use lock coupling techniques and link techniques have been proposed for multi-dimensional index structures [16-20]. On the other hand, on the second issue, which is the need for a phantom protection technique, only a few methods have been proposed [14-15], [18].

Several matured phantom protection methods for B+-Tree exist, e.g., key range locking [21] and next key locking [22]. They rely on the presence of a total order over the underlying data based on their key values. However, in multi-dimensional index structures no such an ordering between keys exists so the existing phantom protection methods for B+-Tree are not applicable. Therefore, the first developed phantom protection method for multi-dimensional index structures uses a modified predicate locking mechanism [18] instead of the techniques described in references [21] and [22]. Predicate locking offers potentially higher concurrency. However, the lock overhead of a predicate locking approach is much higher than that of a granular locking approach. For that reason, a granular locking method is preferred [14]. Consequently, Chakrabarti and Mehrotra proposed new approaches that use a granular locking mechanism in multi-dimensional index structures as described in [14] and [15]. The granular locking approaches are more efficient than the predicate locking approach in reference [18]. However, these approaches still have some problems. First, the lockable granules are the nodes of index trees. Therefore, it is difficult to integrate them with existing concurrency control algorithms that use locks on the nodes of index trees. Second, they work efficiently in space partitioning methods that do not allow overlaps between lockable granules. However, in data partitioning methods with more general index structures, they are less efficient because of the overlaps between the index nodes. Thus, the insert algorithms of the index structures must be modified. Finally, they are only applicable to tree-based index structures. Even though most of the existing multi-dimensional index structures are tree-based, there are several nontree-based index structures such as the VA-file [12] and the hashing technique [13]. They do not need complex concurrency control algorithms to maintain the consistency of index structures unlike the tree-based index structures. However, proper phantom protection methods should be provided.

In this paper, we propose a new phantom protection method that uses a hybrid approach consisting of predicate locking and granular locking. Actually, our target application domain contains Geographical Information Systems (GIS), multilevel secure data bases (MLS), spatio-temporal systems and others where the number of dimensions is 2 or 3. The basic idea of the proposed method is to partition the multi-dimensional data space into a fixed number of cells and to assign a unique number to each cell. Then, we use the cells as lockable units. A searcher's predicate is mapped to a set of a number of cells by selecting cells that are overlapped with the searcher's predicate. The searcher acquires locks on the cells to protect phantoms. An inserter (deleter) maps an object to be inserted (deleted) to a number of cells and acquires locks on the cells. The proposed phantom protection method does not require any modification to the original algorithm of the index structures. Also, it does not obtain any locks on the nodes of index structures. Therefore, it is easy to integrate the proposed method with existing concurrency control algorithms. Finally, it supports all kinds of index structures regardless of whether these structures are tree-based, file-based, or hash-based index structures. We perform various experiments to verify the superiority of the proposed phantom protection method. The experimental results show that our method outperforms the existing methods in terms of response times and lock conflict ratios of various operations.

The contributions of our paper can be summarized as follows. First, the proposed phantom protection method is easy to implement. Also, the integration of the proposed method into existing DBMSs is straightforward. Second, it can be used for all kinds of index structures regardless of their basic data structures, e.g., tree-based, file-based, or hash-based. Finally, various experiments have shown that the performance of the proposed method outperforms existing methods.

This paper is organized as follows. Section II provides a detailed description of existing phantom protection methods and presents the motivating factors that helped lead to our proposed algorithm. In Section III, we describe the proposed phantom protection method. In Section IV, we show performance results and finally, Section V concludes this paper.

2. RELATED WORK

2.1 Existing Phantom Protection Methods

Several matured phantom protection methods for the B+-Tree index have been proposed such as key range locking [21] and next key locking [22]. They rely on the presence of a total order over the underlying data based on their key values. However, in multi-dimensional index structures, no such an ordering between keys exists so the existing phantom protection methods for the B+-Tree index are not applicable to them.

In predicate locking mechanisms, searchers register their search predicates in a tree-global table, so that inserters and deleters can check conflicting concurrent searchers' predicates. Similarly, inserters and deleters register their keys as predicates in the tree-global table, which is checked by searchers for conflicts with their predicates. The single predicate lock is sufficient for a search operation to protect the entire search range, so no locks have to be placed on data records.

However, predicate locks are less efficient to set and check locks. Every check must go through the entire tree-global list of existing predicates which can be very time consuming. Also, searchers must set their predicate locks before the index is accessed and any data records are retrieved. Unlike [21] and [22], the locked key range is not expanded gradually. This can reduce the concurrency if search operations are performed in an interactive fashion.

To our knowledge, Kornacker, Mohan, and Hellerstein proposed the initial phantom protection method for multi-dimensional index structures [18]. It addressed the above problems of the predicate locking mechanism. They proposed hybrid approaches that synthesize two-phase locking of a data record with predicate locking. In the hybrid mechanism, data records that are scanned, inserted, or deleted are protected by the two-phase locking protocol. In addition, searchers set predicate locks to prevent phantoms. Furthermore, the predicate locks are not registered in a tree-global list before the searcher starts traversing the tree. Instead, the locks are directly attached to nodes.

Predicate attachments are performed so that the following invariant is true at all times. If a searcher's predicate is overlapped with a node's minimum bounding rectangle (MBR), the predicate must be attached to the node. An inserter checks only the predicates attached to its target leaf. A deleter performs a logical delete, i.e., a leaf entry is not physically deleted but is only marked as deleted. Searchers attach their predicates to the nodes that they visit. The predicates of the nodes are only removed when the owner transactions commit. Since the tree structure changes dynamically as nodes split and MBRs are expanded during key insertions, the attached predicates have to adapt to the structural changes. In order to handle this problem, the method proposed in reference [18] replicates existing predicates to newly overlapped nodes by structural changes. Possible structural changes mentioned in [18] are node splits and MBR updates.

The first case is a node split which creates a new node whose MBR might be consistent with some of the predicates attached to the original node. The invariant is maintained by attaching those predicates to the new node. The second case involves the expansion of a node's MBR causing it to become consistent with additional search predicates. The additional search predicates at other nodes must be attached to the node. The updater that expanded the MBR must traverse the tree to find predicates.

The hybrid mechanism of [18] has some drawbacks. First, each node of the index trees has an additional space for a predicate table consisting of predicates of searchers, inserters, and deleters. The size of the table is variable. The contents of the table must be changed whenever the MBR updates or node splits are performed. These properties make the maintenance of predicate tables expensive. Second, the lock range is not expanded gradually. The reason is that predicates have to be attached to the visited nodes top-down, starting with the root. This can block an insertion into the search range, even if the leaf where the insertion takes place has not been visited by the search operation.

To overcome the shortcomings of the hybrid mechanism of [18], Chakrabarti and Mehrotra have proposed a granular locking approach in references [14] and [15]. While predicate locking offers potentially higher concurrency, typically granular locking is preferred since the lock overhead of a predicate locking approach is much higher than that of a granular locking approach. References [14] and [15] define the lowest level MBRs as the lockable granules. Each lowest level MBR corresponds to a leaf node of the R-tree. The granules dynamically grow and shrink with insertions and deletions of entries to adapt the data space to the distribution of the objects. The lowest level MBRs alone may not fully cover the embedded space, i.e., the set of granules may not be able to properly protect search predicates resulting in phantoms. Accordingly, they define additional granules called external granules for each non-leaf node in the tree, such that the lowest level MBRs together with the external granules fully cover the embedded space.

Updaters (inserters and deleters) acquire ix-locks on a minimal set of granules sufficient to fully cover the object followed by an x-lock on the object itself. Searchers 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. Chakrabarti and Mehrotra refer to this strategy as the cover-for-insert and overlap-for-search policy. The reverse policy is the overlap-for-insert and cover-for-search. In this policy, ix-locks are acquired on all overlapping granules for inserters and deleters and s-locks are acquired on the minimal set of granules that cover the scan predicate for search that could also be followed.

However, the above two locking policies are not sufficient to prevent phantoms from arising when the granules are dynamically changing because of insertions and deletions. Therefore, some additional locking strategies are proposed. The ultimate lock protocols are summarized as follows. First, inserters acquire ix-locks on all granules that contain the newly inserted object. If the MBR of a node is changed by a new entry, they obtain short duration ix-locks on all overlapping nodes. If overflow occurs, they acquire a six-lock on the overflowed node before a split, and acquire ix-locks on the original node and newly created node after the split and then a s-lock on its parent node's external granule. Second, searchers obtain s-locks on all overlapping granules with the search predicate. Finally, deleters acquire ix-locks on all granules that contain the object to be deleted when logically deleting it and physically obtain short duration ix-locks on the granule that contains the object when deleting the entry.

The granular locking mechanism is much more efficient than the predicate locking mechanism. The lockable granules are nodes of index trees so it uses the existing object locking mechanism of the database systems. Also, unlike the predicate locking mechanism, this mechanism does not need to maintain additional information at each node for storing predicates. However, when the granules are changed or overflow occurs, this mechanism must acquire ix-locks on all nodes overlapped with the object. This requires the inserters to traverse the index tree from the root to find overlapping nodes. Since this mechanism acquires locks on index nodes, it is difficult to integrate it with existing concurrency control algorithms because of conflicts of purpose of the locks.

2.2 Motivation

References [14] and [15] proposed an approach to phantom protection that uses the granular locking mechanism that is more efficient than the predicate locking approach. Even though this locking mechanism outperforms the predicate locking mechanism of [18], this mechanism has some problems.

First, the lockable granules are nodes of the index tree and a complex locking strategy is applied to them. Consequently, it is difficult to integrate the lockable granules into existing concurrency control algorithms that use locks on index nodes. Second, to apply them to R-trees that are one of the data partitioning index structures, the insert algorithm should be modified to support the phantom protection methods. The modified insert algorithms require additional node accesses. Finally, phantom protection methods are only applicable to tree-based multi-dimensional index structures. There are several nontree-based multi-dimensional indexes like structures. In this paper, we propose a new phantom protection method that is a hybrid approach that synthesizes predicate locking and granular locking mechanisms. The proposed mechanism is independent of the type of multi-dimensional index structures, i.e., it supports tree-based, file-based, and hash-based index structures. Also, the proposed new protection method does not need to obtain locks on data records since it uses predicate locking approaches. The proposed phantom protection method achieves a low development cost and high concurrency with a low lock overhead.

3. THE PROPOSED PHANTOM PROTECTION METHOD

3.1 The Basic Idea

The basic idea of the proposed phantom protection method is to partition the multi-dimensional data space into 2^b rectangular cells, where b denotes the user specified number of bits. Then, we allocate a unique bit-string of length b to each partitioned cell. Each unique bit string can be converted to a unique integer value. The integer value is used as a lock identifier of database systems. A searcher's predicate is mapped to a number of cells. The predicate uses the bit-string of each cell as a lock identifier. Then, the searcher acquires s-locks on all of the cells that correspond to the search predicates before starting search operations. On the other hand, inserters and deleters obtain x-locks on the cells corresponding to the objects to be deleted and inserted.

A small number of bits, b_i, are assigned for each dimension, i, Slices, 2^{b_i}, along the dimension, i, are determined in such a way that all slices are equal. Let b be the sum of all b_i, i.e., $b = \sum_{i=1}^{d} b_i$, where d is the number of the dimension. Then, the data space is divided into 2^b hyper-rectangular cells, each of which can be represented by a unique bit-string of length, b. Each cell covers the same size of region. The union of the cells covers the whole data space.

We use the partitioned cells as lockable units. A searcher's predicate can be converted to a number of cells. This is easily done by selecting cells that are overlapped with the predicate. Since we select all overlapped cells, the union of the selected cells covers the area of a search predicate. The searcher, then, acquires s-locks on all of the selected cells. Easy mapping of a given search predicate onto a set of lockable units is an important property for an efficient phantom protection method [9].

We should be able to easily map the cells to lock identifiers used by the standard lock managers of database systems to reduce the cost of lock management. Each cell has a unique bit-string of length, b. The bit-string can be mapped to a unique integer. This mapping will be achieved by casting the bit-string to an integer type. It means that each cell is represented with a unique integer that is generally used as a lock identifier for the standard lock managers. For example, in MIDAS [23], the record lock identifier of MIDAS consists of a page identifier (4 bytes), a slot number (2 bytes), and a reorganization slot counter (2 bytes) in a 32 bit machine. A bit-string can be mapped to a slot number of the record lock identifier. Since the slot number is 2 bytes, if we allocate one page virtually for the proposed phantom protection method, the maximum number of bits becomes 16, i.e., we can partition the data space to 2^{16} cells. If we allocate two pages, the maximum number of cells will be increased to $2 \cdot 2^{16}$.

With this mapping mechanism, the searcher can acquire s-locks on all of the selected cells by using the record locking mechanism of database systems. Similarly, an inserter (deleter) maps the MBR of an object to be inserted (deleted) to a number of cells, and acquires x-locks on the cells. This will protect against phantom problems. Figure 1 shows an example of the proposed algorithm. In this example, we assume that the data space is of 2 dimensions, the region is from (0, 0) to (15, 15), and b is 8. We divide the data space into 2^8 cells.

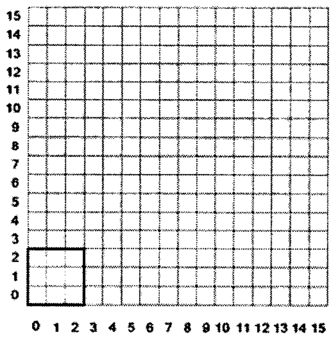


Fig. 1. Mapping of the search region

The shaded area represents a searcher's predicate which is from (0, 0) to (2, 2). The number of cells overlapped with the search predicate is 9. We map the cell that covers from (0, 0) to (1, 1) to 00000000(0). With the same method, the remained cells are mapped to 1, 2, 16, 17, 18, 32, 33 and 34. The searcher acquires s-locks on all of the cells by using a standard lock manager before starting its search operation. Subsequently, an inserter is trying to insert an object (0, 0) into the data space. Like the searcher, the inserter maps the object to a cell 0, and requests a commit duration x-lock on the cell before starting its insert operation. However, since the searcher already has an s-lock on the same cell, the inserter must wait until the searcher commits.

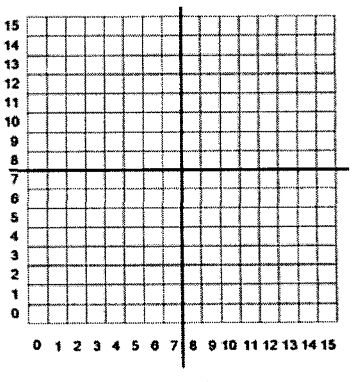
The number of locks of a searcher in the proposed method is totally dependent on the b and its query size, i.e., as the b and the query size increase, the number of locks increases. The number of locks of a searcher is calculated approximately by the following equation, $sn = \lceil 2^b \cdot querysize \rceil$, where *sn* is the number of cells of the searcher's predicate. The term, querysize is the query size of the searcher which is the ratio of the region size of the searcher's predicate to the total size of the data space.

When *b* is 10 and a querysize is 0.05, the *sn* is 51. If *b* is 16 with the same querysize, the *sn* is 3237. As will be shown in the performance evaluation presented in Section 4, the range 6 to10 is enough of a range for the value of *b*. Also, we can fix the b as a reasonable value. However, the *querysize* will be variable according to the users. Therefore, even though the *b* can be fixed to be 10, as the *querysize* is increased, the *sn* is linearly increased.

The shortcoming of the proposed algorithm is that the required number of locks for a searcher's predicate can be too large according to its query size. This lock overhead may degrade overall performance. To overcome this problem, we hierarchically organize the partitioned cells like a Multi-Level Grid-file. On each level, we group cells to $2 \cdot l \cdot d$ clusters of cells, where l is the level. For example, as shown in Figure 2, on level 1, 4 clusters exist and each cluster contains 64 cells; and on level 2, there are 16 clusters, and each cluster contains 16 cells. On the highest level, level 0, only one cluster that covers all of the cells exists. The number of levels is determined by the equation, $\left\lceil \frac{b}{d} \right\rceil + 1$.

After clustering the cells on each level, we also assign a unique bit-string of length (*lb*+*b*) to each cluster, where lb denotes the number of bits for representing the level. A bit string for a cluster is composed of a bit string for a level of length, *l*, *b* and the bit string of the lower left cell in the cluster. In this hierarchical approach, lock identifiers are determined as follows. After obtaining overlapped cells with a searcher's predicate, for each level, clusters are selected from the selected cells. This is done in ascending order of level.

For example, in Figure 3, we map a searcher's predicate which is from (0, 0) to (2, 2) to lockable units. Overlapped cells with the search predicate are 0, 1, 2, 16, 17, 18, 32, 33 and 34. Then, we find clusters from the selected cells. There is one cluster on level 3, and 5 clusters on level 4. The cluster on level 3 is mapped to 011(level) + 00000000 (bit-string for the lower left cell in the cluster), which is 768 as a decimal integer. The remaining clusters can be also mapped by the same method to integers. The searcher acquires s-locks on all of the mapped clusters. The number of locks required for the searcher is reduced to 6 compared to that of Figure 1. With this strategy, we can reduce the number of locks.



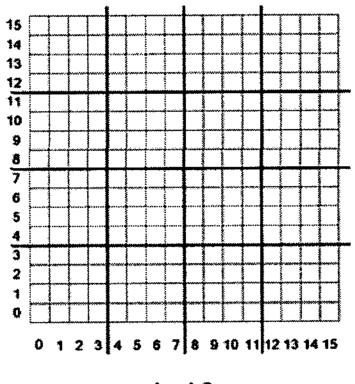


Fig. 2. Hierarchical organization

In this hierarchical approach, for an inserter (deleter), x-locks on the cells that are overlapped with the MBR of an entry to be inserted (deleted) are not sufficient. For example, in Figure 3, the searcher keeps an s-lock on a cluster on level 3 and s-locks on 5 clusters on level 4. Then, an inserter is trying to insert an object (0, 0). The inserter maps the entry to a cell 0, requests an x-lock on the cell, and acquires an x-lock on it since the searcher keeps s-locks not on the cell 0 but the cluster 768. The entry (0, 0) will be a phantom for the searcher. To avoid this situation, the inserter must acquire ix-locks on all clusters that are overlapped with the MBR of the entry except the lowest level cluster. The x-lock must be acquired on the lowest level cluster. Again, in Figure 3, the inserter must acquire ix-locks on cluster 000+00000000, 001+00000000, 010+00000000 and 011+00000000, besides an x-lock on 100+00000000. In this case, the inserter cannot acquire the ix-lock on 011+00000000 until the searcher commits.

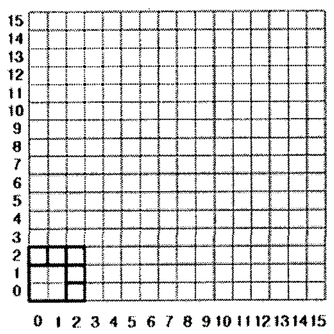


Fig. 3. Mapping of the hierarchical approach

3.2 Dynamic Phantom Protection Method

The proposed phantom protection method described in the previous subsection assumes that the multi-dimensional data space is static. If the data space of an application is static and we preliminary know the data space area, the proposed algorithm works well. However, when the entire data space is dynamically changed, we should estimate the maximum data space area to apply our proposed method. However, this will increase the dead space and the overall concurrency may be downgraded.

Consequently, we propose a dynamic phantom protection method. First, we assume reading and writing of words are atomic as in most of the modern computer architectures [24]. In order to efficiently protect phantoms in the dynamically changing data space, 2b rectangular cells must be adapted as the data space grows or shrinks. Figure 4(a) shows the original data space. Figure 4(b) shows the expanded data space by a number of insertions so the cells are resized to be adapted to the changed data space. Once the data space is changed and cells are resized, inserters, deleters, and searchers must acquire locks on the resized cells.

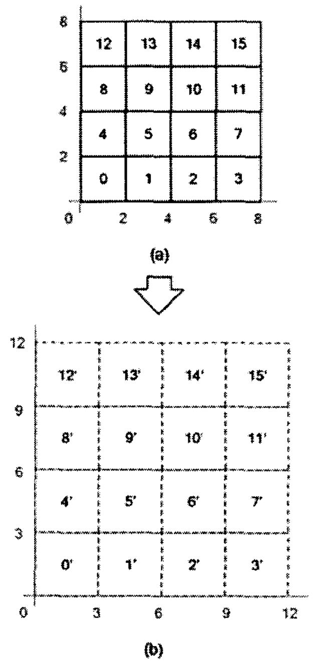


Fig. 4. The expansion of data space - 1

For example, in Figure 4, a searcher with a predicate, (6, 6) to (8, 8), which is initiated before the data space is changed, has acquired an s-lock on cell 15 which is overlapped with the search predicate. Then, the data space grows and the cells are resized to be adapted to the changed data space as in Figure 5. An inserter requests an x-lock on the cell 10' to insert a new point entry (7, 7). It can acquire the x-lock on the cell since none keeps locks on the 10'. Subsequently, the searcher lost the s-lock on the area (6, 6) to (8, 8). The new entry (7, 7) can be a phantom for the searcher if the searcher scans the area repeatedly.

In order to avoid this situation, the inserter must acquire x-locks on cells 15', which is 10 in the original data space, as well as 10'. However, if transactions that are initiated before the data space is changed are not remained, the inserter does not need to acquire the x-lock on 15. From now on, a transaction that is initiated before the data space is changed and a transaction that is initiated after the data space is changed are called an old transaction and a new transaction, respectively. In Figure 5, we can easily know why cell 15 should be locked. The shaded cell, (6, 6) to (9, 9), is 10'. The cell 10' is overlapped with a cell 15, (6, 6) to (8, 8), on the old data space. From this fact, we can determine that the cell 15 of the original data space must be locked. Therefore, final cells to be locked are 10' and 15' on the changed data space. For another example, if a searcher tries to obtain an s-lock on the cell 5' in Figure 5, the searcher must also acquire locks on 5, 6, 9 and 10 of the original data space that are overlapped with cell 5' of the changed data space.

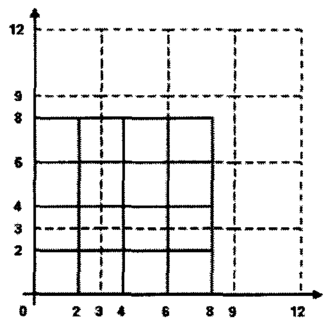


Fig. 5. The expansion of data space -2

Rule 1. If the data space has been changed and old transactions are still being performed, new transactions must acquire locks on the cells of the changed data space; and, moreover, the cells of the original space that are overlapped with the locked cells of the changed data space.

With Rule 1, we can protect phantoms from arising even when the data space is dynamically changed. However, we must solve the following two problems to apply this rule. First, who does determine the time to change the MBR of the data space and how does he or she inform other transactions of the changed MBR? Second, how can new transactions know that old transactions remain?

In tree-based index structures, the MBR of the root node represents the MBR of the data space. When the index structure is nontree-based, we can easily maintain the MBR of the data space by updating the MBR on every insertion. An inserter that updates the MBR of the root node can determine the time to change the MBR of the data space. If the MBR of the data space is changed whenever the MBR of the root node is updated, the overhead to handle the change may reduce the concurrency. Therefore, we fix the amount of the change to decide the time to change the MBR of the data space as $\left\lceil \frac{sidelength_i}{2^{b_i}} \right\rceil$, where *sidelength$_i$* denotes the i^{th} dimension's length of the data space and bi denotes the number of bits for the i^{th} dimension. That is, we change the MBR of the data space and inform others of the change when one of the *sidelength$_i$* is increased or decreased by the side length of a cell along the dimension, i.

In order to inform others about the change, we use data structures shown in Figure 6. In Figure 6, the array dataspace_mbr[] is used to store old and new MBRs of the data space. The current_mbr is a flag that has 0 or 1 as values. This flag is used as the index number of array dataspace_mbr[]. The current_mbr indicates which one is the new MBR between the dataspace_mbr[current_mbr] and dataspace_mbr[1-current_mbr]. Finally, the array cnt[] represents the number of old and new transactions that currently are being performed. cnt[current_mbr] means the number of transactions that are referring to the new MBR, dataspace_mbr[current_mbr].

```
struct Header
{
      current_mbr;          // 0 or 1
      cnt[2];
      dataspace_mbr[2];
}
```

Fig. 6. Data structure for the proposed algorithm

Whenever an inserter changes the MBR of the root node, the inserter checks the amount of the change by comparing the current MBR of the root or the whole data space with *dataspace_mbr[current_mbr]*. If one of the $sidelength_i$ is changed more than $\left\lceil \frac{sidelength_i}{2^{b_i}} \right\rceil$, the inserter updates the *current_mbr* as *1-current_mbr* so that *current_mbr* indicates the other *dataspace_mbr[]* which was the old MBR. Then, it updates the *dataspace_mbr [current_mbr]*.

Inserters, deleters, and searchers increase *cnt[current_mbr]* by 1 before starting their operations and decrease it by 1 when the initiating transactions commit or rollback. Also they generate locks according to Rule 1. At this time, they can decide whether old transactions still exist by reading the *cnt[current_mbr]*, i.e., new MBR and *cnt[1-current_mbr]*, i.e., old MBR. Inserters, deleters, and searchers acquire s-latches on the Header data structure to maintain consistency. They increase or decrease *cnt[]* without acquiring x-latches since we assume reading and writing of words are atomic. Only inserters and deleters acquire x-latches on the Header when updating *dataspace_mbr[]*. The data structure can be added to an index table that is created when an index is opened. We can ignore the overhead to maintain the Header data structure because the data structure is small enough and most operations acquire s-latches on the Header for instant duration. X-latches are acquired only when the *dataspace_mbr[]* is updated. Usually, the update of *dataspace_mbr[]* occurs very infrequently. Also, the data structure does not need to be stored permanently.

Finally, we must consider a situation as indicated in Figure 7. An inserter can insert an object (the point in Figure 7) to the outside of the data space. Also, a searcher's predicate (the circle in Figure 7(a)) can be placed on the outside of the data space in Figure 7(a). However, the inserter and the searcher only acquire locks on the cells that are overlapped with the search predicate or the MBR of an entry to be inserted. Therefore, the search predicate is not fully protected from arising phantoms. In this example, the inserter does not acquire any lock. The searcher acquires s-locks on cells 14 and 15. Consequently, the inserter can insert the entry to the search predicate. It will be a phantom for the searcher.

To solve this problem, we define additional lock units. As shown in Figure 7(b), the shaded cells are the additional lock units that unlimitedly cover the outside of the current data space. The actual data space is covered by cells 0 to 15. To cover the outside of the data space, we partition the outside of the data space into $2^{b_i} \cdot 2d + 2^d$ cells. For example, additional 2^{b_i} cells are along the upper and lower sides of each dimension i, and 2^d cells are on corners of the data space. Then, we apply our mapping method; that is, we select all cells that are overlapped with the MBR of an entry to be inserted or a searcher's predicate.

The overall algorithms of the proposed method are presented in Figures 8 and 9. Searchers release a s-latch on the Header after getting *cell_ids[]*, that is, an array of cell IDs to be locked. The locks are released when the initiating transaction commits, i.e., commit duration locks. If *cnt[1-current_mbr]* is not 0, there are old transactions that are referring to *dataspace_mbr[1-current_mbr]*. In this case, we acquire locks on old and new data spaces.

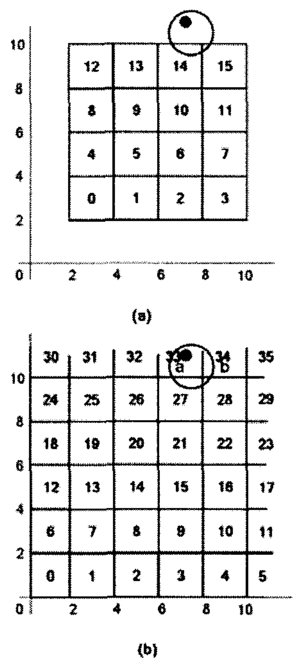


Fig. 7. Additional lock units

```
Search(query_range, etc)
{
    acquire s-latch on Header;
    cnt[current_mbr]++;
    read dataspace_mbr[current_mbr];
    cell_ids[] = get overlapped cells with query_range;
    if ( cnt[1-current_mbr] != NULL )
    {
        read dataspace_mbr[1-current_mbr];
        cell_ids[] = get cells to be locked;
    }
    release s-latch on Header;
    acquire commit duration s-locks on cell_ids[];
    perform operations;
    acquire s-latch on Header;
    cnt[current_mbr]--;
    release s-latch on Header;
}
```

Fig. 8. Search algorithm

```
Insert/Delete(entry, etc)
{
    acquire s-latch on Header;
    cnt[current_mbr]++;
    read dataspace_mbr[current_mbr];
    cell_ids[] = get overlapped cells with entry.mbr;
    if ( cnt[1-current_mbr] != NULL )
    {
        read dataspace_mbr[1-current_mbr];
        cell_ids[] = get cells to be locked;
    }
    release s-latch on Header;
    acquire locks on cell_ids[];
    perform insert/delete operation;
    acquire s-latch on Header;
    if (root_mbr is changed more than threshold value)
    {
        if(cnt[1-current_mbr] == 0)
        {
            upgrade s-latch on Header to x-latch;
            current_mbr = 1-current_mbr;
            update dataspace_mbr[current_mbr];
            cnt[1-current_mbr]--;
            release x-latch;
        }
        release s-latch;
    }
    else
    {
        cnt[current_mbr]--;
        release s-latch;
    }
}
```

Fig. 9. Insert/Delete algorithm

Inserters and deleters perform more complex actions than searchers to protect phantoms. They need to check if the *dataspace_mbr[current_mbr]* needs to be updated whenever completing their operations. When visiting the root to propagate updated MBRs or splits, they check that the *root_mbr* is changed more than the threshold value as described above. If the amount of change is greater than the threshold value, they check if old transactions that are referring to *dataspace_mbr[1-current_mbr]* are remaining. If there are old transactions, they are finished without updating *dataspace_mbr[1-current_mbr]* since old transactions are referring to *dataspace_mbr[1-current_mbr]*. The update will be done later by other transactions that update the root MBR. Otherwise, they upgrade s-latch to x-latch, update *current_mbr* to *1-current_mbr*, and *dataspace_mbr [current_mbr]* to the *root_mbr*. The x-latches can be overhead. However, the updates of *dataspace_mbr[]* occur infrequently.

4. PREFORMANCE EVALUATION

4.1 Environments

To evaluate the performance of our phantom protection method, we compared its performance to the granular locking (GL) approach proposed in [15]. We integrated the GL approach and our proposed phantom protection method into the concurrency control algorithm of [20], which is called the RPLC. The RPLC is mainly focused on the physical consistency of an index structure.

We implemented both phantom protection methods based on the man-machine integration design analysis system (MIDAS) that is a multi-user storage system for a base of aircraft data (BADA) DBMS [23]. Both of them are implemented with locks, latches, and the logging APIs of MIDAS. In order to implement the GL, we modified the insertion algorithm of the RPLC so as to perform correctly its granular locking algorithm. The modified insertion algorithm performed additional tree traversing whenever MBRs were changed by insertions or deletions of entries. In addition, we modified the locking strategy of the RPLC to adapt to the granular locking method.

Also, our phantom protection method was implemented on the RPLC. The implementation was simple and easy. We did not need to modify the original insert, search, and delete algorithms of the RPLC. We just added functions to generate locks for search predicates and objects to be inserted or deleted. Then, the additional data structure was added into the index table of MIDAS. We allocated the root node for lock identifiers of partitioned cells since the root node of the RPLC was not changed unless the index structure was destroyed. The first and second records of the root node were reserved as a tree lock and a node lock so the available lock identifiers for phantom protection are 2 ~ 2b+2.

We used a uniformly distributed 200000, 2~3 dimensional synthetic point data set. One of the important parameters of an index tree is a node size. According to the node size, the performance of the index trees is varied. We performed experiments with varying the node size ranging from 4 Kbytes to 16 Kbytes. In all experiments, our phantom protection method outperformed GL. Overall performance was improved as the node size became bigger. Also, the performance gap of both algorithms increased about 10~15 % when the node size was 16 Kbytes. The increase of the performance gap may be from the growth of contention. We will discuss the results of experiments when the node size is 16Kbytes and the number of dimension is 2 for brevity.

Initially, a CIR-tree [10] was constructed by bulk loading techniques. Subsequently, feature vectors were inserted concurrently by multiple processes under a specific workload. Table 1 shows the workload parameters. According to the input parameters, the workload generators decided the number of search and insert processes, the number of concurrent processes, the initial number of feature vectors to construct index trees, and the selectivity of range queries.

Subsequently, the workload generators passed the decided values to a driver program that was written with C and MIDAS APIs. The driver executed search and inserted processes. It randomly selected feature vectors from an already inserted data set for queries and from a data set to be inserted for insertions. Each process executed multiple transactions. We fixed the number of buffer pools as 100 when initiating MIDAS. The platform used in our experiments was a dual Ultra Sparc processor, Solaris 2.7 with 128 Mbytes of main memory.

Table 1. Paramenters and values

Parameters	Values
Number of feature vectors	200000
Insert probability	0% ~ 100 %
Range of queries	0.2 ~ 0.8 %
Number of concurrent Processes (MPL)	10 ~ 50
Number of bits	6, 8, 10 (64, 256, 1024 cells)

4.2 Experimental Results

Fig. 10 and Fig. 11 compare the performance of the proposed phantom protection method and granular locking (GL) method in terms of response time when the numbers of insert and search processes are varied. Also, we varied the number of bits from 6 to 10. The 64, 256 and 1024 in Figures 10 and 11 denote the response times of the proposed algorithm when the number of bits is 6, 8 and 10 respectively.

Figure 10 shows the response times of search operations when the insert process ratio is varied by 0 to 100 percent. The proposed algorithm outperforms the GL method regardless of the number of bits. However, when the insert process ratio is 0, all cases show almost the same results. Since their basic search algorithms except the locking strategy are the same, the results mean that the lock overheads of GL and the proposed method are similar.

As the insert process ratio increases, the performance gap between the GL method and the proposed method becomes more and more large. When the number of bits is 10, the performance gap is maximized. This increase means that the larger the number of cells, the higher the performance of the proposed algorithm. However, actually, the difference among graphs 64, 256, and 1024 is marginal. The reason is that the larger the number of cells, the more locks to be acquired. Consequently, the lock overhead is increased. A similar aspect is shown in Figure 11. However, the performance gap between GL and the proposed method is larger than that of Figure 10. The insert algorithm of GL is more complex than that of the proposed method. It must traverse the index tree to find nodes that are overlapped with the changed MBR by the insertion of a new entry.

Figures 12 and 13 show the response times of both methods when the selectivity is varying. As the selectivity increases, the overall response times of both methods increases. The proposed method outperforms GL in all cases. The reason is that as the area of a searcher's predicate increases, the lock contention increases. A drawback of the proposed method is that the number of locks increases as the size of the query increases. Consequently, this lock overhead may degrade the concurrency of the proposed method. However, as shown in Table 2, the average number of locks per a search is not so large. This lock overhead is compensated by high concurrency as we showed in the previous figures.

Table 2. Number of locks

Selectivity(%)	0.2	0.4	0.8
Number of locks	11.63	13.893	17.445

Fig. 14 and 15 show the scalability of the proposed method. The performance of the proposed method is superior to that of the GL method. As the number of concurrent processes increases, the performance gap between both methods becomes larger. This result means that our proposed method is scalable for the number of concurrent processes.

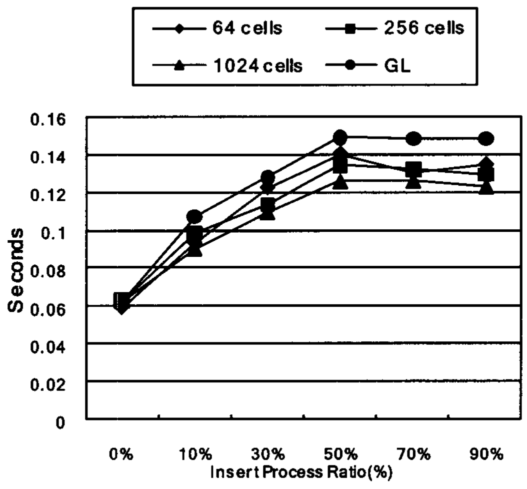


Fig. 10. Response time of search transactions (selectivity=0.2, database size = 200K, MPL = 50)

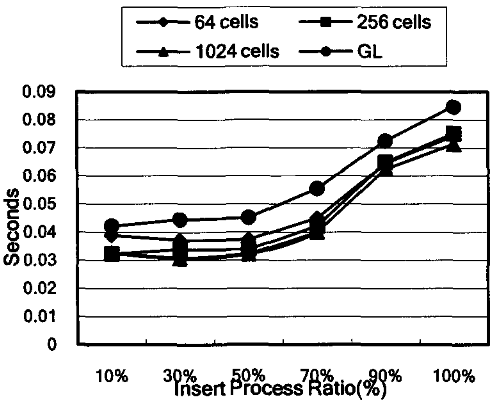


Fig. 11. Response time of insert transactions (selectivity=0.2%, database size = 200K, MPL = 40)

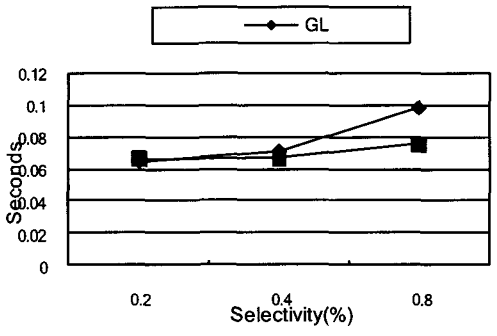


Fig. 12. Response time of a search transaction (MPL=40, insert probability=20%, database size=200K)

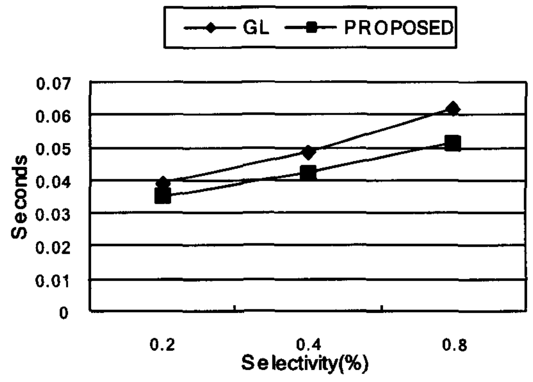


Fig. 13. Response time of an insert transaction (MPL=40, insert probability=20%, database size=200K)

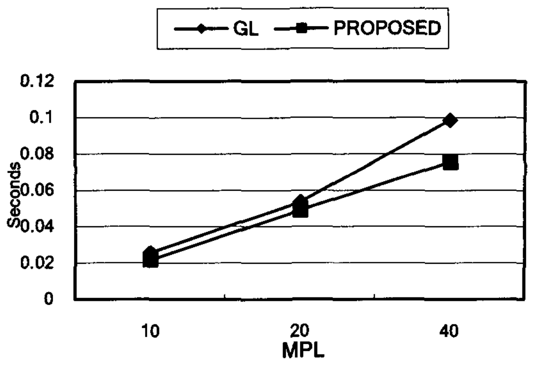


Fig. 14. Response time of a search transaction (selectivity=0.8%, insert probability=20%, database size= 200K)

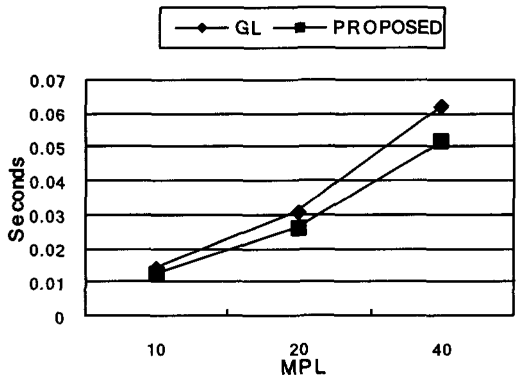


Fig. 15. Response time of an insert transaction (selectivity=0.8%, insert probability=20%, database size= 200K)

5. CONCLUSION

In this paper, we have proposed an efficient phantom protection method for multi-dimensional index structures. The proposed phantom protection method is a hybrid approach using predicate locking and granular locking. It does not require any modification of the original algorithm used by index structures and does not acquire any locks on the nodes of index structures. Therefore it is easy to integrate this proposed method with existing concurrency control algorithms. Also, it supports all kinds of index structures regardless of whether the structures are tree-based or nontree-based.

We have implemented the proposed method on MIDAS, a storage system of a BADA-3. We used the locking and logging application programming interfaces (APIs) that are provided by MIDAS. We, then, performed experiments under various conditions. The performance results showed that our proposed algorithm outperformed GL. Our method is scalable for the number of concurrent processes and the size of the query. The performance improvements are not so large, but the development cost of our method is much cheaper than that of the GL method.

In our further research, we will perform more extensive experiments. We described experiments with synthetic data in this paper. Even though the results were sufficient to show the superiority of our method, we need to perform experiments with a real data set for the completeness of the verification. Also, we need to show that our method works well in a high-dimensional data space.

REFERENCES

[1] J. Nievergelt, H. Hinterberger and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey Structure," **ACM Transactions on Database Systems**, Vol. 9, No. 1, 1984, pp. 38-71.

[2] Silberschatz and P. B. Galvin, "Operating System Concepts," **Addison-Wesley**, 1995.

[3] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," **Acta Informatica**, Vol. 4, 1974, pp. 1-9.

[4] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," **Proceedings of VLDB**, 1987, pp. 507-5.

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

[6] S. Berchtold, D. A. Keim and H. P. Kriegel, "The X-Tree: An Index Structure for High-dimensional Data," **Proceedings of VLDB**, 1996, pp. 28-39.

[7] N. Katayama and S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," **Proceedings of ACM SIGMOD**, 1997, pp. 369-380.

[8] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," **Proceedings of VLDB**, 1997, pp. 426-435.

[9] K. I. Lin, H. Jagadish and C. Faloutsos, "The TV-tree: An Index Structure for High Dimensional Data," **Journal of VLDB**, Vol. 3, No. 4, 1994, pp. 517-542.

[10] J. S. Yoo, M. G. Shin, S. H. Lee, K. S. Choi, K. H. Cho and D. Y. Hur, "An Efficient Index Structure for High Dimensional Image Data," **Proceedings of AMCP**, 1998, pp. 134-147.

[11] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High-dimensional Feature Spaces," **Proceedings of ICDE**, 1999, pp. 440-447.

[12] R. Weber, H. Schek and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," **Proceedings of VLDB**, 1998, pp 194-205.

[13] P. Indyk and R. Motwani "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," **Proceedings of STOC**, 1998, pp. 604-613.

[14] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-Trees," **Proceedings of ICDE**, 1998, pp. 446-454.

[15] K. Chakrabarti and S. Mehrotra, "Efficient Concurrency Control in Multidimensional Access Methods," **Proceedings of ACM SIGMOD**, 1999, pp. 25-36.

[16] J. K. Chen and Y. F. Huang, "A Study of Concurrent Operations on R-Trees," **Journal of Information Sciences**, Vol. 98, No. 1-4, 1997, pp 263-300.

[17] K. V. Ravi, Kanth, D. Serena and A. K. Singh, "Improved Concurrency Control Techniques for Multi-Dimensional Index Structures," **Proceedings of Symposium on Parallel and Distributed Processing**, 1998, pp. 580-586.

[18] M. Kornacker, C. Mohan and J. M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," **Proceedings of ACM SIGMOD**, 1997, pp. 62-72.

[19] M. Kornacker and D. Banks, "High-Concurrency Locking in R-Trees," **Proceedings of VLDB**, 1995, pp. 134-145.

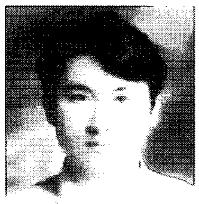
[20] S. I. Song, Y. H. Kim and J. S. Yoo, "An Enhanced Concurrency Control Algorithm for Multi-dimensional Index Structures," **IEEE Transactions on Knowledge and Data Engineering**, 2004, pp. 97-111.

[21] Mohan, "ARIES/KVL: A Key Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes," **Proceedings of VLDB**, 1990, PP. 392-405.

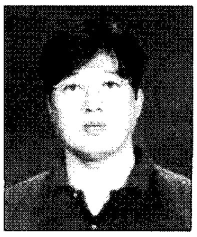
[22] Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," **Proceedings of ACM SIGMOD**, 1992, pp. 371-380.

[23] M. Chae, K. Hong, M. Lee, J. Kim, O. Joe, S. Jeon and Y. Kim, "Design of the Object Kernel of BADA-III: An Object-Oriented Database Management System for Multimedia Data Service," **Proceedings of Workshop on Network and System Management**, 1995.

[24] S. K. Cha, S. Hwang, K. Kim and K. Kwon, "Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems," **Proceedings of VLDB**, 2001, pp. 181-190.



Seok Jae Lee
He received the B.S., M.S. and Ph.D degrees in Computer and Communication Engineering in 2000, 2002 and 2006 from Chungbuk National University, Cheongju, South Korea. He is now a Post Doc. in Chungbuk National University. His main research interests are the database system, main memory storage system, cluster system and real-time distributed computing.



Seok Il Song
He received the B.S., M.S. and Ph.D degrees in Computer and Communication Engineering in 1998, 2000 and 2003 from Chungbuk National University, Cheongju, South Korea. He is now a professor in the department of Computer Engineering, Chungju National University, Chungju, South Korea. His main research interests are the database system, index structure, distributed computing and storage management system.

Jae Soo Yoo

He received the B.S. degree in Computer Engineering in 1989 from Chunbuk National University, Chunju, South Korea. And he received the M.S. and Ph.D. degrees in Computer Science in 1991 and 1995 from Korea Advanced Institute of Science and Technology, Taejeon, South Korea. He is now a professor in the department of Computer and Communication Engineering, Chungbuk National University, Cheongju, South Korea. His main research interests are the database system, multimedia database, distributed computing and storage management system.