Active Adjustment: An Effective Method for Keeping the TPR*-tree Compact*

SANG-WOOK KIM, MIN-HEE JANG AND SUNGCHAE LIM+
Department of Information and Communications
Hanyang University
Seoul, 133-791 Korea
+*Department of Computer Science*
Dongduk Women's University
Seoul, 136-714 Korea

Recently, with the advent of diverse applications based on locations of moving objects, it has become crucial to develop efficient index schemes for spatio-temporal databases. The TPR*-tree is popularly accepted as an index structure for processing future-time queries on such spatio-temporal databases. In the TPR*-tree, the future locations of moving objects are predicted based on the Conservative Bounding Rectangle (CBR). Since the areas predicted from CBRs tend to grow rapidly over time, CBRs thus enlarged lead to serious performance degradation in query processing. To solve the problem, we propose a novel method to adjust CBRs to be tight, thereby preventing the performance degradation of query processing. Our method examines whether the adjustment of a CBR is necessary when accessing a leaf node for processing a user query. Thus, it does not incur extra disk I/Os in this examination. Also, in order to make a correct decision, we devise a cost model that considers the I/O overhead for the CBR adjustment and the performance gain in the future-time owing to the CBR adjustment. With the cost model, we can prevent unusual expansions of BRs even when updates on nodes are infrequent and also avoid unnecessary execution of the CBR adjustment. For performance evaluation, we conducted a variety of experiments. The results show that our method improves the performance of the original TPR*-tree significantly.

Keywords: moving objects, spatio-temporal databases, spatio-temporal indexing, future-time queries, TPR-tree, TPR*-tree

1. INTRODUCTION

The recent advances of technologies in mobile communications and global positioning systems have increased people's attentions to an effective use of location information on the objects moving in 2-dimensional space. Moving objects usually send their current positions to a central server in a periodic fashion, and their positional data has a spatio-temporal feature [1]. Here, the spatio-temporal feature means that the spatial locations of objects continuously change over time [1]. The spatio-temporal data of moving objects needs to be stored in a database for its efficient use, and such a database is commonly termed as the moving object database [2].

Received October 20, 2008; revised March 9, 2009; accepted June 4, 2009.
Communicated by Tei-Wei Kuo.
* This work was supported by the Mid-Career Researcher Program through the NRF (National Research Foundation) grant funded by the MEST (Ministry of Education, Science, and Technology) (Grant No. 2008-0061006) and by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-(C1090-1011-0009)). This work was also supported by the Brain Korea 21 Project in 2010.

Users' spatio-temporal queries issued against the moving object databases can be roughly categorized into two types: past-time queries [3, 4] and future-time queries [5]. The past-time query is to retrieve or track the history of object's movements [3, 4], while the future-time query is to predict object's positions in a particular future-time interval of interest [5]. As a typical example of the future-time query, we can consider a question that "Inform me about all the vehicles that will pass over the Golden Gate Bridge at 1 pm". The systems answering the future-time queries are required for diverse applications such as GIS, location-based services, traffic information services, and air traffic control systems [1]. Due to their usefulness in diverse application fields, we focus our attention on the efficient method for processing the future-time queries.

Many efforts have been done to devise efficient index structures suitable for future-time queries. For example, the index schemes of the VCR-tree [6], the TPR-tree [7], the TPBR-tree [8], and the TPR*-tree [9] are proposed based on the classical index tree called the R-tree. Recently, researches on enhancing the update performance of the previous methods are presented in STRIPES [10], the B^{dual}-tree [11]. In particular, the B^{dual}-tree performs well in the overall performance aspects, such as disk utilization, update cost, and query processing time [11]. However, the TPR*-tree is reported to provide a best performance in terms of the query processing times [11]. Since our research aimed at enhancing the query processing time in the moving object databases, we will compare our method with the TPR*-tree in section 4.

The TPR*-tree basically adopts the data structure of the R*-tree [13] to capture moving objects in a node. For less usage of disk space, the TPR*-tree does not record all the velocities of individual moving objects. Instead, a group of moving objects is captured by a conservative bounding rectangle (CBR), and the CBR records only the representative velocities of the same group of objects. More specifically, the CBR saves the maximum speeds of moving objects indexed in it with respect to 2-dimensional axes. The future-time positions are predicted by expanding the minimum bounding rectangle (MBR) of a CBR [13] at its representative maximum speeds. The MBR of the CBR is a rectangle that has enclosed all the object positions at the CBR's update or creating time. Since the predicted rectangular regions from the CBR data cannot compactly capture the time-varying real positions of moving objects, dead spaces in the TPR*-tree are unavoidable [13]. Since the dead spaces are indexing areas in which no object appears in reality, dead spaces deteriorate the TPR*-tree performance because of many of useless node accesses [9].

Against such a dead space problem, the TPR*-tree takes an approach to updating the CBR data for its compaction, if possible. Such operation for CBR compaction is termed as the CBR adjustment in this paper. With the appropriate CBR adjustments, the TPR*-tree can partially solve the dead space problem [9]. However, since the CBR adjustment of the TPR*-tree is possible only at a node being updated, the nodes with infrequent updates tend to have a huge size of dead spaces [14]. For this reason, the TPR*-tree is apt to experience performance degradations.

In this paper, we propose a novel CBR adjustment method for the TPR*-tree. The basic idea of our method is that if the process for query processing can do CBR adjustments actively, the TPR*-tree could be compacted in a more efficient manner. Using our method, the TPR*-tree can reduce dead spaces even in a situation when the node updates rarely arise. However, if we want to allow the query process to do the CBR adjustment, then there is a very critical point to be taken into account. In contrast to the CBR adjust-

ment by the node update process, the CBR adjustment by the query process leads to addition I/O overhead. Since the CBR adjustment by the update process is a mandatory operation for retaining the consistency of the TPR*-tree, its CBR adjustment operation cannot be regarded as an addition overhead. However, the CBR adjustment by any query process is not. Therefore, we need to evaluate the performance gains from the CBR adjustment by the query process, in order to decide on the need of the CBR adjustment by that process.

In our paper, we call such CBR adjustment performed by the query process the *active CBR adjustment* (ACA). Since the ACA can be done by the query process, a lack of node updates does not severely affect the performance of our TPR*-tree. To decide on the need of an ACA execution, our query process predicts the performance benefit caused by its ACA execution. If the expected benefit is large enough to offset addition I/O overhead, then the query process will initiate its ACA action; otherwise, that process just does its query processing operations as in previous TPR*-tree.

This benefit prediction is based on a probabilistic cost model. The cost model takes into account two major cost factors, that is, the I/O cost for an ACA execution and the future-time's performance gain from that ACA. Since the ACA execution requires disk writes for updating the CBR data of the involved nodes, it has additional I/O overhead mentioned before. Meanwhile, CBR compaction reduces dead spaces, thereby potentially improving the overall query performance in the near future.

Since the benefit prediction can be performed by almost all query processes, its computational cost should not be costly. For its cheap computational cost, the query process in our method caches the CBR data during its downward search phase and uses a very efficient cost model for benefit prediction. Details about that will be described in section 3. From the proper evaluation based on the benefit prediction, our query process can do its ACA actions only at lucrative situations.

To reveal the performance advantages of our method, we conduct extensive experiments. From the experimental results, we observe that our method provides significant performance improvement in query processing time, compared to the original TPR*-tree.

This paper is organized as follows. As related work, section 2 briefly reviews the TPR-tree and the TPR*-tree, and introduces the concept of the CBR adjustment. Section 3 proposes our method in detail. Section 4 shows the results of performance evaluation for justifying the effectiveness of our approach. Finally, section 5 summarizes and concludes this paper.

2. RELATED WORK

Here, we present the basics of the TPR-tree and TPR*-tree, which are widely accepted as the indexing schemes for future-time queries, and then address their desirable features and shortcomings.

2.1 TPR-tree

The TPR-tree [7], which has been devised based on the R*-tree [13], predicts the future-time positions of moving objects by storing the current position and velocity of each object at a specific time point. To express the time-varying positions of moving objects,

the TPR-tree uses the notion of the conservative bounding rectangle (CBR), rather than the minimum bounding rectangle (MBR) used in the R*-tree.

A CBR is composed of two sorts of data: an MBR and a velocity vector. The MBR is a rectangle encompassing a group of moving objects indexed in a node, command the velocity vector represents the maximum and minimum speeds of the moving objects within the MBR along 2-dimensional axes. If it is necessary to predict the future-time position of a moving object, then the velocity vector is applied to the corresponding MBR in order to compute a rectangle covering the predictable positions of the queried object. Such a predicted rectangle in the index space is called the bounding rectangle (BR). The CBR of a leaf node is to express the BR for the moving objects in that node, and the CBR of a non-leaf node is used to represent the BR covering the BRs of its child nodes. Owing to the use of the velocity vector and MBR, the TPR-tree can index moving objects with a small size of disk space, rather than saving the individual positions of objects indexed by child nodes into their parent node.

Fig. 1 illustrates how to expand a CBR to obtain the corresponding BR for future-time prediction.

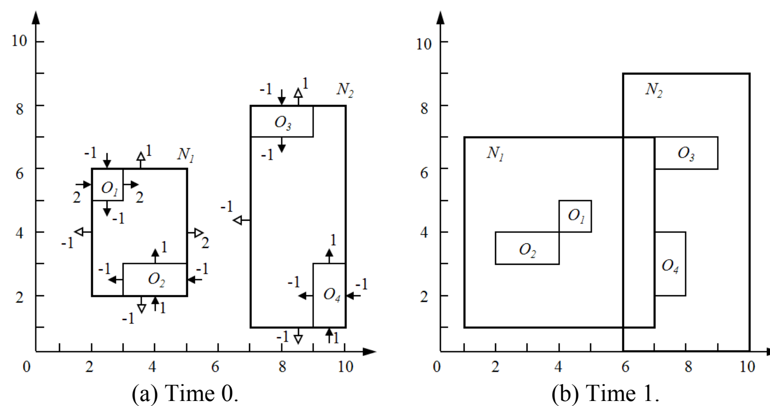


Fig. 1. CBR expansion for the future-time 1.

In Fig. 1 (a), O_1, O_2, O_3, and O_4 denote moving objects being indexed, and while N_1 and N_2 represent two MBRs for those four objects. At the initial time 0, we have two CBRs as in Fig. 1 (a). Here, a black-headed arrow with a numeral denotes both of the direction and the speed of a moving object, while a white-headed arrow represents the velocity vector, *i.e.*, the maximum or minimum speeds at which any object within the MBR is moving. At this initial time, the MBRs and BRs are equivalent to each other. From the CBRs of Fig. 1 (a), we can obtain BRs for time 1 as in Fig. 1 (b). Here, the rectangles N_1 and N_2 represent the BRs at time 1 and they are drawn by expanding N_1 and N_2 according to their respective velocity vectors.

To process a user query Q with a future prediction time t, the TPR-tree computes the BRs for time t by expanding the CBRs saved in the root node and recursively searches down the sub-trees whose BRs overlap with the target query region of Q [16, 12]. As shown in Fig. 1 (b), the BR predicted by a CBR continuously grows as time goes on. This

is because the BR boundaries are extended at the maximum and minimum speeds of the objects contained in the CBR. Such a fast BR growth normally leads to large dead space, and thus incurs more disk I/Os in query processing because of huge overlaps among BRs.

To prevent continuous growth of BRs, the TPR-tree updates the CBR data in such a way that it covers the positions of objects more tightly whenever any object pertaining to the CBR changes its velocity or location information [7]. By updating the CBR data, the TPR-tree can reduce the areas of the dead space. We call such a CBR update the CBR adjustment.

2.2 TPR*-tree

The data structure and the query processing algorithm of the TPR*-tree [9] are very similar to those of the TRP-tree. A difference between them is found in the algorithm for inserting a new object into the tree. That is, the TPR*-tree makes a modification to the insertion algorithm of the R*-tree in order to reflect the objects' mobility, while the TPR-tree employs the original insertion algorithm of the R*-tree without modification. Due to the enhanced insertion algorithm, the TPR*-tree is reported to provide a better performance in terms of insertion and query processing costs than the TPR-tree [9].

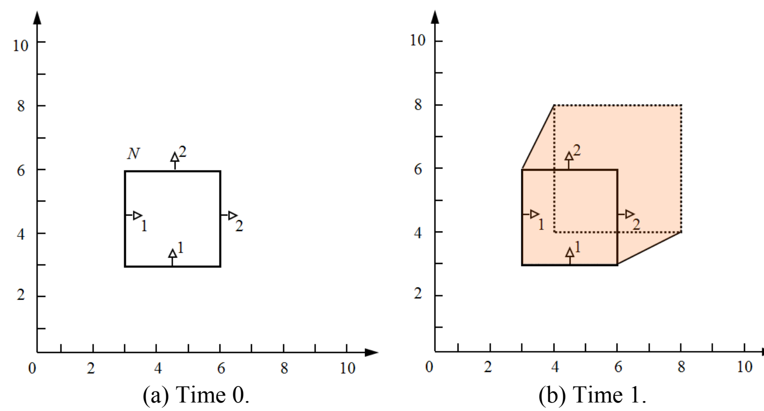
While inserting a new object into the tree, the TPR-tree assesses the overall changes of the sizes of MBR areas, the lengths of MBR boundaries, and the overlapping regions among MBRs that are caused by this object insertion. By selecting the tree-path where such changes remain smallest, the TPR-tree yields the least space possible.

Although the approach can be very efficient for indexing stationary objects as in the R*-tree, it cannot be a good solution in the case of moving objects. Note that the TPR-tree does not consider the parameter of time, since it estimates the MBR changes found only at the insertion time without considering the time-varying sizes of the BRs. To solve the shortcoming, the TPR*-tree uses another insertion algorithm to reflect the characteristic of time-varying BRs, which result from the objects' mobility.

For this, the TPR*-tree employs an insertion algorithm that considers how much the BR will sweep the index space from the insertion time to a specific future-time. For instance, consider two different time points of t_1 and t_2 ($t_1 < t_2$). The sweeping region of a BR from t_1 to t_2 is defined to be an index space area that is swept by the BR expanding during the time interval ($t_2 - t_1$). Fig. 2 depicts an example of a sweeping region in the TPR*-tree. At the initial time 0, we have the BR *N* of Fig. 2 (a). Until time 1, the BR sweeps the index space as in Fig. 2 (b), where the sweeping region is denoted by the gray-filled polygon. In this example, the sweeping region is 23 in size.

The insertion algorithm searches down the TPR*-tree for a leaf node by recursively choosing the child pointers to the sub-trees where its insertion will occur. During the downward search, the algorithm chooses its insertion paths so that the overall sweeping region remains smallest. Therefore, the insertion process has to compute the change in sweeping regions due to its object insertion. Such computation may consume more CPU time than that of the TPR-tree. However, the TPR*-tree can provide a better response time for the future-time queries because its compactness of CBRs.

In addition, owing to the enhanced deletion algorithm, the TPR*-tree performs better for update operations of deletions as well as insertions. According to the experimental results in [9], the TPR*-tree shows up to five times better performance than the TPR-tree.



(a) Time 0. (b) Time 1.

Fig. 2. Sweeping region from time 0 to time 1.

3. THE PROPOSED METHOD

In this section, we propose a new method for improving the query performance in the TPR*-tree. At first, our research motivation is presented in section 3.1, and the basic strategy of our method is then described in section 3.2. Section 3.3 presents the detailed algorithm.

3.1 Motivation

Because the CBR of the TPR*-tree stores only the maximum and minimum speeds of moving objects in the MBR, the BR predicted from the CBR enlarges at a fast and continuous rate. Such rapid growth of the BR leads to a huge dead space and thus causes large overlaps among nodes' BRs as time goes on. That may significantly impair the performance of query processing because an increasing number of node accesses is required for query processing. Against this problem, as mentioned before, the TPR*-tree executes CBR adjustment on a node whenever an update operation is needed to reflect the change of object's velocity or its current position on that node.

To learn about the benefit of the CBR adjustment, we use Fig. 3. In Fig. 3 (a), a CBR is created or updated to capture the objects of O_1 and O_2 at time 0 and its MBR is denoted by rectangle *R*. Note that the BR from the CBR is the same as the rectangle *R*, since BR expansion does not begin at this initial time. Fig. 3 (b) depicts the positions of those objects after the time interval of 1. As known from the figure, objects O_1 and O_2 moved closely to each other. If a CBR adjustment arises at time 1 due to a node update on that, we can have a smaller MBR like *New* in Fig. 3 (b). On the other hand, the predicted BR from the initial CBR of Fig. 3 (a) comes to be a larger rectangle of R' of Fig. 3 (b). With a CBR compaction at time 1, we can eliminate the dead space of $R' - New$, thereby eliminating some node accesses misled to the area of $R' - New$ in the near future.

Although the CBR adjustment may be a solution for better performance, it has a problem in that its execution relies on update operations. That is, the TPR*-tree cannot adjust the CBR data of a node *N*, if no object indexed in *N* changes its position or velocity. In other words, the size of dead space in node *N* enlarges continuously for a very long

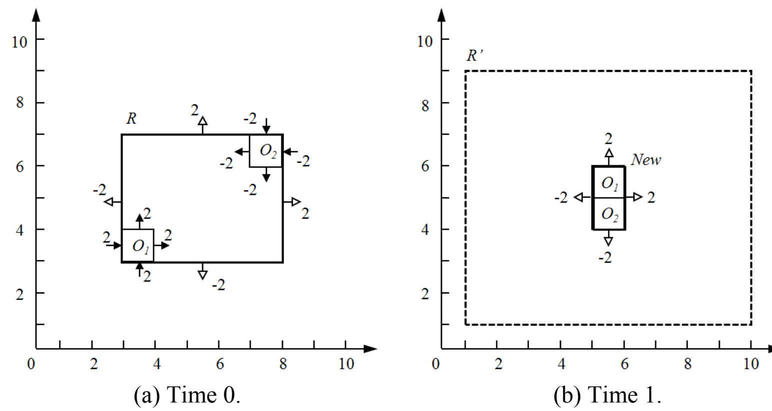


Fig. 3. MBR R at the initial time 0 and its expansion R' at time 1.

time, if N is a node without updates. Therefore, if query processes frequently searches against a sub-tree with rare updates, then their response times may get longer as time goes. Since the hot regions for user queries in the tree can be different from them for node updates, such an undesirable situation is feasible.

Such a problem mainly comes from the passive characteristic of the CBR adjustment in the original TPR*-tree. To overcome such restrictive and passive characteristic of the TPR*tree, we propose a new method enabling the query process to do the CBR adjustment in an efficient manner.

3.2 Basic Strategy

In this section, we describe a basic strategy for the proposed method. For exposition, let us consider a situation where a query process P reaches a leaf node N of a TPR*-tree for its query processing. Let T_{u_j} and $T_{u_{j+1}}$ be the update times when the jth and the $(j + 1)$th CBR adjustments arise on N, respectively. Assume that process P happens to arrives at N during the interval of $[T_{u_j}, T_{u_{j+1}}]$, and process P is the ith query process accessing N in that interval. Then, let us denote the user query having released P by $Q_{j,i}$. That is, $Q_{j,i}$ is the ith user query accessing N after the jth update has occurred on N. If we let $T_{q_{j,i}}$ be the access time of $Q_{j,i}$ to N, and there are k queries accessing N in $[T_{u_j}, T_{u_{j+1}}]$, then we have an arrival sequence as below.

$$T_{u_j} < T_{q_{j,1}} < T_{q_{j,2}} < T_{q_{j,3}} < \ldots < T_{q_{j,k}} < T_{u_{j+1}}$$

Since the BR of N enlarges constantly during $[T_{u_j}, T_{u_{j+1}}]$, the user queries at $T_{q_{j,i}}$ $(1 \leq i \leq k)$ will view the growing BR of N and thus the possibility of their misleading to N increases during that interval. Note that if there is any query having an overlap between its target query region and the dead space of N, then the query will uselessly access N because of misleading.

With the observation above in mind, we return to the situation when the query process of $Q_{j,i}$ arrives at N. If this process is able to do its CBR adjustment on N at $T_{q_{j,i}}$, then we can prevent some of queries $Q_{j,x}$ $(i < x \leq k)$ from being misled to a dead space of N

during [$T_{q_{j,i}}$, $T_{u_{j+1}}$]. From that, we can reduce the same number of useless node access to N. In this paper, we call this sort of a CBR adjustment by the query process the *active CBR adjustment* (ACA). And we just call other CBR adjustments by node updates the CBR adjustment. By enabling such ACAs, our method can improve the query performance of the TPR*-tree.

Since the benefit from an ACA on N at $T_{q_{j,i}}$ can be assessed with the reduced number of queries misled to N after during [$T_{q_{j,i}}$, $T_{u_{j+1}}$], we need to compute such a reduction number for benefit prediction. As mentioned before, the benefit prediction should take into account the negative aspect of the ACA. Since the ACA is not a mandatory operation needed for the TPR*-tree consistency, in contrast to the CBR adjustment by the update process, we have to regard the I/O cost for node writes during the ACA as an additional overhead. When we term the performance benefit from an ACA as the CBR adjustment benefit (CAB), therefore, the amount of the CAB should be greater than the additional I/O cost of the ACA. Our benefit prediction considers such pros and cons of the ACA together.

For benefit prediction, we introduce a formula for computing the CAB in order to probabilistically count the reduction number of misled queries during [$T_{q_{j,i}}$, $T_{u_{j+1}}$]. Eq. (1) below is for computing the CAB being obtainable if the query process of $Q_{j,i}$ do its ACA actions at time $T_{q_{j,i}}$. In Eq. (1), the functions $SR(T_{q_{j,i}}, T_{u_{j+1}})$ and $SR'(T_{q_{j,i}}, T_{u_{j+1}})$ are for returning the sizes of the query spaces where two different BRs of N will sweep during the interval of [$T_{q_{j,i}}$, $T_{u_{j+1}}$]. Between them, the function $SR(T_{q_{j,i}}, T_{u_{j+1}})$ returns the size of the sweeping region that we will have if node N is not adjusted at $T_{q_{j,i}}$. The function $SR'(T_{q_{j,i}}, T_{u_{j+1}})$ returns the sweeping region size in the case that N is actively adjusted at $T_{q_{j,i}}$.

$$CAB(T_{q_{j,i}}, T_{u_{j+1}}) = \frac{SR(T_{q_{j,i}}, T_{u_{j+1}}) - SR'(T_{q_{j,i}}, T_{u_{j+1}})}{2} \times Q_{freq} \times (T_{q_{j,i}}, T_{u_{j+1}}) \tag{1}$$

In [9], it is proven that possibility of misleading to N during [$T_{q_{j,i}}$, $T_{u_{j+1}}$] is proportional to the size of $SR - SR'$. In addition, the average size of the reduced sweeping regions viewed by the queries $Q_{j,x}$ ($i < x \leq k$) is computed as its half as in Eq. (1). The notation Q_{freq} represents the query frequency per the unit of the query space. So, we can the CAB by integrating Q_{freq} and the time interval of ($T_{q_{j,i}}$, $T_{u_{j+1}}$) into the reduced dead space size. With Eq. (1), we can calculate the average number of reduced node access on N during [$T_{q_{j,i}}$, $T_{u_{j+1}}$], if an ACA arises on N at time $T_{q_{j,i}}$.

Now, we can compute the CAB of the ACA, if the next update time of $T_{u_{j+1}}$ is given. The time of $T_{u_{j+1}}$ of N can be predicted based on the statistical data of the average update period of a leaf node. This average update period is computed by dividing the average update period of a single moving object by the average object number per leaf node. Such statistical information can be gathered by analyzing the real behaviors of moving objects or is given according to a simulated environment.

In this paper, such update period is denoted by P_u, and thus $T_{u_{j+1}}$ can be replaced with $T_{u_j} + P_u$. Since we can get the value of the CAB of Eq. (1) by replacing $T_{u_{j+1}}$ with $T_{u_j} + P_u$, we only have to assess the worst-case cost for an ACA execution for our benefit prediction. We count this worst-case number as $H - 1$, where, H is the height of the used TPR*-tree. The worst-case number of node writes is for the case when propagations of CBR updates go up to the root node. Since the root node always resides in memory, we do not

count its update. As a result, we use the value of $H - 1$ as the worst-cast number of node writes for an ACA.

With the notations until now, we propose an inequity for the benefit prediction as below. If the query process of $Q_{j,i}$ finds that (*Condition i*) is satisfied at time $T_{q_{j,i}}$, the process decides on the ACA execution, since its action is always lucrative.

$CAB(T_{q_{j,i}}, T_{u_j} + P_u) > H - 1$ (*Condition i*)

Lastly, we discuss the hidden CPU cost of our benefit prediction for checking (*Condition i*). Although the CPU cost for evaluating (*Condition i*) is tiny, it is better to reduce useless evaluations of that, if possible. Especially, since that the condition checking may occur at every query processing time, we need to have any appropriate mechanism to reduce its evaluation times. To this end, we use (*Condition ii*) below, which is much cheaper than (*Condition i*) in CPU usage.

$T_{q_{j,i}} + epsilon < T_{u_j} + P_u$ (*Condition ii*)

If (*Condition ii*) does not hold at time $T_{q_{j,i}}$, our query process gives up its ACA on N. Therefore, (*Condition i*) is not checked as well. If we can properly choose the value of *epsilon* of (*Condition ii*), then we can avoid unnecessary checking of (*Condition i*). Here, the *epsilon* acts as a minimum interval time within which an ACA is possible in reality. Since the shorter time until $T_{u_j} + P_u$ leads to less possibility of a lucrative ACA, there is no need to evaluate (*Condition i*) if the remaining time to the next node update is too short. In our research, we set the value of *epsilon* to the time taken to read H nodes. This is because (*Condition ii*) cannot be satisfied, if the remaining time until $T_{u_j} + P_u$ is less than such a time. By checking (*Condition ii*) before a benefit prediction using (*Condition i*), our method can save CPU time.

The advantages of the proposed ACA method are threefold. First, since the query rate is higher than the node update rate in general, we can have a better chance to perform the CBR adjustments, if needed, than in the previous TPR*-tree. Second, even though a certain sub-tree S has no node update on it, our method is able to execute ACAs for S while user queries are being. If the ACAs do not arise due to infrequent user queries against S, the dead space in S will not impair the query performance because there is no need of node reads for query processing on S. Third, our method does not need any additional node reads for benefit prediction. While checking (*Condition i*) of benefit prediction, the query process needs the involved CBR data for computing the size of reduced dead space, *i.e.*, $SR - SR'$. If these data require any extra I/Os, then performance gains are rarely expected in our method. However, since we use the query process as the ACA executors, the process can use the CBR data read during its downward search phase. For this, we force every query process to cache all the CBR data accessed during its search phase. Details about this CBR caching and the algorithm for the query process are presented in the next section.

3.3 Algorithm

A future-time query from a user is expressed in the form of (Q_r, Q_t). Here, Q_r denotes the query target region in the 2-dimensional index (or query) space, and Q_t is the target

Algorithm Search (Q_r, Q_t, *rootNode*)
Input: (Q_r, Q_t) = future-time query answered.
rootNode = address to the root of a sub-tree being searched.
Return: S_r = objects' *id*s satisfying (Q_r, Q_t).
1.IF *rootNode* is a non-leaf node THEN
2.FOR EACH index entry *e* in *rootNode*
3. IF there is an overlap between the BR of *e* end Q_r within Q_t THEN
4. Cache the content of *e* into the memory buffer area.
5. $childS_r \leftarrow$ Search(Q_r, Q_t, *childNode*).
6. /* *childNode*: the child node pointed to by *e* */
7. IF an ACA execution is needed on *childNode* THEN
8. Adjust the CBR data of *e*.
9. END IF
10. $S_r \leftarrow S_r \cup childS_r$.
11. END IF
12.END FOR
13.RETURN S_r.
14.ELSE /* leaf node */
15.$S_r \leftarrow$ **I**. /* initialization */
16.FOR EACH moving object *O* in *rootNode*
17. IF *O* is expected computed to be positioned *d* in Q_r within interval Q_t THEN
18. $S_r = S_r \cup$ *O's id*.
19. END IF
20.END FOR
21. $T_{u_j} \leftarrow$ *rootNode*.TS.
22.IF $T_{q_{j,i}}$ + *epsilon* < $T_{u_j} + P_u$ THEN /* (*Condition ii*) */
23. IF $CAB(T_{q_{j,i}}, T_{u_j} + P_u) > H - 1$ THEN /* (*Condition i*) */
24. Notify the need of an ACA on this leaf node, *i.e.*, *rootNode*.
25. END IF
26.END IF
27.RETURN S_r.
28.END IF
Algorithm End.

Fig. 4. Proposed algorithm *Search*() for the query process.

query time interval of that query. The future-time query is answered by predicting the moving objects that will pass Q_r during the interval of Q_t. The proposed search algorithm is given in Fig. 4, where it returns a set of object *id*s satisfying a query of (Q_r, Q_t).

The algorithm searches down for the leaf level from the root node. During the downward search, the search process finds its search paths by computing the BRs from each entry *e* in the internal nodes. In the meantime, the process will cache the encountered CBR data as in line 4. When arriving at a leaf node in line 14, this algorithm selects a group of object *id*'s satisfying the given query condition, and return them.

In lines 22 and 23, the conditions of (*Condition ii*) and (*Condition i*) are checked for the ACA execution. The value of T_{u_j} is obtained from the **TS** field in the leaf node in line 21, where **TS** stands for a time stamp. This field is added to in every leaf node of our TPR*-tree. With this field, the leaf node records its last update time. This field is set by the update process at the node update time.

Besides this algorithm for the query process, some algorithms for insertion and deletion are required. However, we do not present such updater's algorithms in this paper. Because those algorithms are identical with the previous ones in [9] except that they update the TS field of the leaf node, we do not give them for space limitation.

4. PERFORMANCE EVALUATION

In this section, we compare the query processing performance between our proposed method and the original TPR*-tree. From the experiment comparisons, we show the performance advantages of the proposed method over the TPR*-tree.

4.1 Experimental Environment

We generated datasets for our experiments using the GSTD [15], a well-known data generator used in many previous researches on performance evaluations of the index schemes for moving objects [16, 17]. With the GSTD, we generated 100,000 moving objects have a random speed each in the range of [0, 70]. Those objects are simulated to move around within the normalized 2-dimensional query space of the size 10,000 by 10,000. In that query space, an object is represented as a point and its initial position is chosen according to three sorts of distribution, *i.e.*, uniform, skewed, and Gaussian distribution.

Table 1. Simulation parameters and their values.

Parameters	Parameter values
Update periods (per moving object)	5, 10, **50**, 100, 150
Query frequency (per unit time)	10, 20, **40**, 60, 80, 100
Average speed of moving objects	30, **50**, 70, 90
Size of target query regions	0.01%, **0.16%**, 0.64%, 2.56%
Future prediction time point	20, 40, **60**, 80, 100

To look at how well the proposed method works in diverse computing environments, we use five simulation parameters reflecting them. Table 1 gives such simulation parameters: the average update period of moving objects, the query rate, the sizes of target query regions, the average speed of moving objects, and the future prediction time points. In the table, the figures in the right-side column indicate such parameters' values to be picked in our experiments.

Since the different kinds of parameters affect the query performance in different ways, we executed five simulations for highlighting the influence of individual parameters listed in Table 1. For example, to see how the update period parameter affects the query performance, we change its values from 5 to 10, 50, 100, and 150 stepwise, while holding other parameters as fixed values, *i.e.*, as their *pivot* values. In Table 1, the pivot values of the simulation parameters are represented by boldface figures, each one in the parameter values' column in Table 1. In the case of varying update periods, the rest simulation parameters are fixed to 40, 50, 0.16%, and 60, respectively, from the higher rows

in Table 1. This is applied in the same way to experiment with other four simulation parameters.

As performance measures in our experiments, we use both of the average number of node accesses and the average response time for query processing. Since the response time highly depends on the number of node accesses required for query processing, its computation may be useless for performance comparison, if we use the number of node accesses. However, we compute the response time for the purpose of precise and fair performance comparisons. This is because our method could be in a trouble due to more CPU usage during the benefit prediction phase. If such CPU overhead is large, our method could have a poor performance in spite of less disk access. From performance analysis, however, we found that the overhead is less enough.

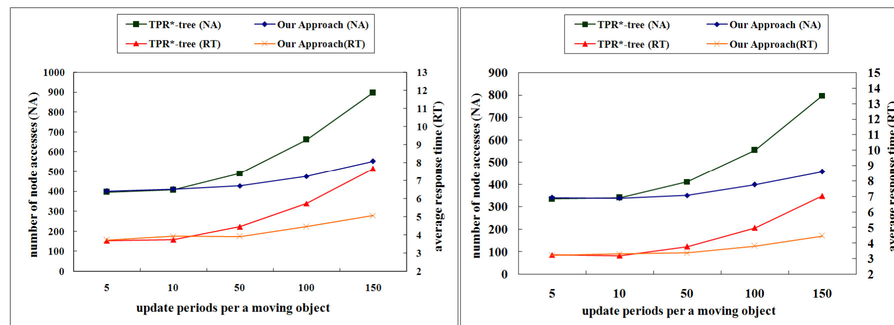
Our experiments were performed on a Windows 2000 server equipped with a Pentium 4 processor of 2.23 GHz and 512 MB of main memory. By comparing our method with the original TPR*-tree, we will analyze our good performance features.

4.2 Experiment Results and Analyses

In this section, we carry out various performance comparisons between our method and the original TPR*-tree. Due to space limitations, we omit the experiment results of the objects whose initial positions are chosen in Gaussian distribution, because the results are very similar to those of uniform and skewed distribution.

Firstly, we estimate the performance enhancement in our method according to the varying update rates. Its experimental results are depicted in Fig. 5, where the graphs show how the average node access number and the average response time change with the enlarged update periods, that is, with the decreasing update frequencies. In the figure, the average node access numbers are given on the left side of the *y*-axis, while the average response times are shown on the right side of that.

In Fig. 5, our method and the TPR*-tree both experience more node accesses and enlarged response times, as the update period gets longer. This phenomenon is natural, since more dead spaces are created from less CBR adjustment in nodes. However, even in the situation of infrequent node updates, our method has a better performance, compared to the TPR*-tree. In our method, the estimated two performance measures get larger very



(a) Uniform distribution. (b) Skewed distribution.

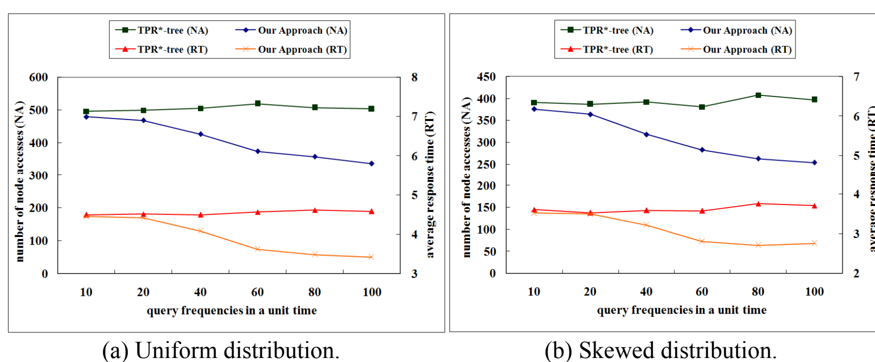
Fig. 5. Experiment results with respect to varying update periods.

slowly. From this, our method provides a better performance after the update period of 10. At the best case, our method can reduce 39% and 43% of nodes access, in the uniformed and skewed distributions, respectively. In addition, our method reduces up to 34% and 37% of the average response time, in the same distributions, respectively. The CPU time used for benefit prediction accounts for the less improvement in the average response time.

In Fig. 5, an interesting experiment result is found in the range of 5 to 10 of the update period, where node updates arise frequently. In this range, our method works nearly identically with the original TPR*-tree, because there is no chance to execute our ACAs. Thus, most of our query processes behaves the same as in the TPR*tree. In this range, if the computational cost for benefit prediction is high, then the performance of our method could be worse than the TPR*-tree. However, our method can avoid this problem, by checking (*Condition ii*) before benefit prediction of checking (*Condition i*). That is, we can skip the computation of benefit prediction in many times, since (*Condition ii*) is usually unsatisfied in the presence of high update rates. Of course, we can see a very tiny degradation in the average response time at the update period of 10 in Fig. 5 (a). However, we can say that this degradation is negligibly small and restrictive.

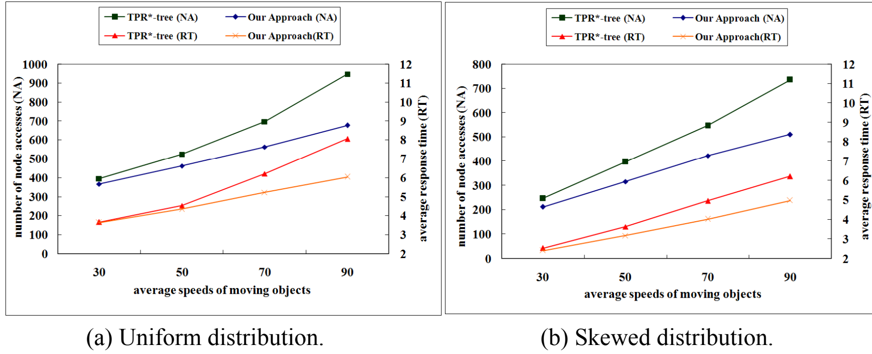
Besides the update period above, the query frequency also acts as a major parameter affecting the query performance. Fig. 6 shows how the query frequency parameter influences the query performance. During this simulation, the changes of parameter values rarely influence the behavior of the original TPR*-tree. Therefore, the performance graphs of the TPR*-tree nearly level off in the overall range of query rates. In contrast, our method reduces up to 34% and 37% of node access times in the distributed and skewed distributions, respectively. As known from the graphs of the average node access times, our method also does not provide performance enhancement after the query frequency point of 60. The phenomenon is straightforward. After any threshold of the query frequency, most of queries cannot do their ACA actions, since their ACAs are not lucrative. Note that such threshold rate relies on how much the TPR*-tree can be compacted. Our method also enhances the response times by at most 26% and 28% in the same distributions, respectively. These enhancement gaps between response times and node access times are explained as the time for benefit predictions.

From the experiments of Figs. 5 and 6, we have learned about how the update frequency and the query frequency affect the performance of our method.



(a) Uniform distribution. (b) Skewed distribution.

Fig. 6. Experiment results with respect to varying query frequencies.



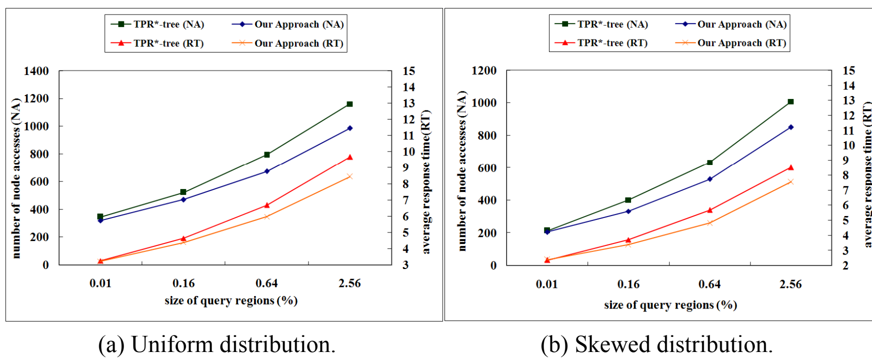
(a) Uniform distribution. (b) Skewed distribution.

Fig. 7. Experiment results with respect to varying average speeds of moving objects.

In turn, we will verify that our method retain good performance features in diverse simulated computing environments. To this end, we first experiment with the simulation parameter of the average speed of moving objects. Fig. 7 shows the experiment results, where the dataset with a lot of fast moving objects requires more disk accesses because of its large dead spaces. Recall that CBR data saves the maximum speeds of moving objects in it, and its expansion speed increases the average speed of objects being indexed.

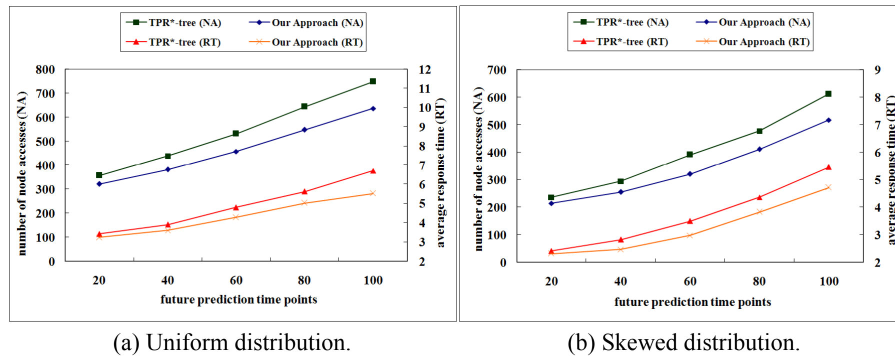
From Fig. 7, we can see that our method outperforms the TPR*-tree, in overall range of objects' speeds. In particular, our method works much better in a situation where the fast expansion of BRs is likely to severely impair the query performance. In Fig. 7, we can have the performance enhancement by up to 29% in uniform distribution and by up to 31% in skewed distribution. Correspondingly, the average response times are improved by at most 26% and 20% in the same distributions, respectively.

Fig. 8 shows how the simulation parameter of the query size affects the query performance of our method. Since this parameter is less dominant on the query performance than other three before, the performance gaps between our method and the TPR*-tree is not much. The figure reveals that the average number of node accesses is reduced by up to 17% in the both distributions. In the case of the average response time, our performance improves at most 14% in the uniform distribution and at most 13% in the skewed distribution. Since larger query regions incur more disk accesses basically, our method can provide some performance enhancement from more compacted nodes in the tree.



(a) Uniform distribution. (b) Skewed distribution.

Fig. 8. Experiment results with respect to varying sizes of the query regions.



(a) Uniform distribution. (b) Skewed distribution.

Fig. 9. Experiment results with respect to varying future prediction time points.

Lastly, in Fig. 9 we examine how the performance changes as the future prediction time point goes farther. From this figure, we can see that our method reduces the number of node accesses by up to 15% and 16% in uniform and skewed distributions, respectively. To the response time, the performance improves by up to 18% in uniform distribution and by up to 16% in skewed distribution. Note that this simulation parameter has a similar effect of the parameter of the query region size, since the queries of farther future times tend to have bigger query regions to be searched.

From the five experiment until now, we have verified that our method outperforms the original TPR*-tree in a variety of simulation environments. In particular, in the environments with much chance of ACA executions, our method works better. For instance, with less update rates or high query rates, our method can highly enhance query performance. By contrast, if there is no chance to do the ACA, then our method runs the same as the TPR*-tree. Consequently, compared to the TPR*-tree, the proposed method is more suitable to process future-time queries in all the situations.

5. CONCLUSIONS

In this paper, we proposed an efficient algorithm of the active CBR adjustment for the better query performance of the TPR*-tree. We also verified the performance advantage of the proposed algorithm through intensive experimental comparisons.

The TPR*-tree is an index structure devised to efficiently answer the future-time queries on moving objects. Although the TPR*-tree has a good characteristic of high storage utilization, it is apt to suffer from performance degradation because of quickly growing dead space and overlapping regions among bounding rectangles as time goes on. To partially solve the problem, the TPR*-tree adjusts the CBR data at the time a leaf node is updated to reflect the change of location information of any moving object. However, such a solution seems to have an obvious limitation in that the dead space of tree nodes enlarges continuously until those nodes are updated. Therefore, the infrequency of node updates can result in a severe degradation of query performance.

To solve this problem, we have proposed a new method capable of executing the CBR adjustment during query processing. In our method, the query processor can perform the CBR adjustment in an active manner. Since the query processor can cache the CBR data

from its search path on the buffer memory, the CBR adjustment could be very efficient, if it is well-devised. From that idea, we allow the query processor to perform the active CBR adjustment (ACA). For the ACA mechanism to be useful, we have also proposed a cost model. Based on the cost model, the proposed method can assess both of the performance benefit to be obtained during the future-time query processing and the I/O cost for an ACA execution. By checking difference between them, our method determines whether or not the ACA execution is profitable for the tree performance. To show the performance enhancement of the proposed method over the original TPR*-tree, we have conducted various performance experiments. From the simulation results, we can see that our method provides at most 40% of performance improvement, compared to the original TPR*-tree.

REFERENCES

1. D. L. Lee, J. Xu, B. Zheng, and W. C. Lee, "Data management in location-dependent information services," *IEEE Pervasive Computing*, Vol. 1, 2002, pp. 65-72.
2. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving objects databases: issues and solutions," *Scientific and Statistical Database Management*, 1998, pp. 111-122.
3. X. Xu, J. Han, and W. Lu, "RT-tree: An improved R-tree indexing structure for temporal spatial databases," *Spatial Data Handling*, 1990, pp. 1040-1049.
4. Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," *IEEE Multimedia Computing and Systems*, 1996, pp. 441-448.
5. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Modeling and querying moving objects," *IEEE Data Engineering*, 1997, pp. 422-432.
6. S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch, "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," *IEEE Transactions on Computers*, Vol. 51, 2002, pp. 1124-1140.
7. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," *ACM SIGMOD Record*, Vol. 29, 2000, pp. 331-342.
8. M. Pelanis, S. Saltenis, and C. S. Jensen, "Indexing the past, present, and anticipated future positions of moving objects," *ACM Transactions Database Systems*, 2006, pp. 255-298.
9. Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: An optimized spatio-temporal access method for predictive queries," in *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003, pp. 790-801.
10. J. Patel, Y. Chen, and V. Chakka, "STRIPES: An efficient index for predicted trajectories," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2004, pp. 637-646.
11. M. Yiu, Y. Tao, and N. Mamoulis, "The B^{dual}-tree: Indexing moving objects by space filling curves in the dual space," *The International Journal on Very Large Data Bases*, Vol. 17, 2008, pp. 379-400.
12. M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*,

Vol. 26, 2003, pp. 40-49.
13. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: An efficient and robust access method for points and rectangles,” *ACM SIGMOD Record*, Vol. 19, 1990, pp. 322-331.
14. M. H. Jang, S. W. Kim, and M. Shin, “Performance of TPR*-trees for predicting future positions of moving objects in U-cities,” in *Proceedings of the 1st KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, 2007, pp. 841-850.
15. Y. Theodoridis, R. Silva, and M. Nascimento, “On the generation of spatiotemporal datasets,” *Advances in Spatial Databases*, 1999, pp. 147-164.
16. D. Pfoser, C. S. Jensen, and Y. Theodoridis, “Novel approaches in query processing for moving objects,” in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000, pp. 395-406.
17. B. Lin and J. Su, “On bulk loading TPR-tree,” in *Proceedings of IEEE International Conference on Mobile Data Management*, 2004, pp. 395-406.

Sang-Wook Kim (金尚煜) received the B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea at 1989, and earned the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea at 1991 and 1994, respectively. From 1994 to 1995, he worked with the Information and Electronics Research Center in Korea, as a Senior Engineer. From 1995 to 2003, he served as an Associate Professor of the Division of Computer, Information, and Communications Engineering at Kangwon National University, Chunchoen, Kangwon, Korea. In 2003, he joined Hanyang University, Seoul, Korea, where he currently is a Professor at the School of Information and Communications. From 1999 to 2000, he worked with the IBM T. J. Watson Research Center, Yorktown Heights, New York, as a Post-Doc. He also visited the Computer Science Department of Stanford University as a Visiting Researcher in 1991. He is an author of over 80 papers in refereed international journals and conference proceedings. His research interests include storage systems, transaction management, main-memory DBMSs, embedded DBMSs, data mining, multimedia information retrieval, geographic information systems, and web data analysis. He is a member of the ACM and the IEEE.

Min-Hee Jang (張珉熙) received the B.S. degree from Hong-ik University, Seoul, Korea at 2003, and earned M.S. degrees in Information and Communications Engineering from Hanyang University, Seoul, Korea at 2006. In same year, he joined Hanyang University where he currently is on Ph.D. course in Information and Communications Engineering. His research interests include storage systems, data mining, geographic information systems, and multimedia information retrieval.

SungChae Lim (林成采) received the B.S. degree in Computer Engineering from Seoul National University at 1992, and achieved the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), at 1994 and 2004, respectively. He also worked for the Korea Wisenut Cooperation from 2000 to 2005, and he is currently an Assistant Professor in the Department of Computer Science at Dongduk Women's University. His research interest includes the high-performance indexing, mobile computing, and semantic Web.