The Journal of Systems and Software 83 (2010) 1500–1511

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Historical index structure for reducing insertion and search cost in LBS

Young Jin Jung [a,*], Keun Ho Ryu [b,**], Moon Sun Shin [c], Silvia Nittel [d]

[a] Korea Institute of Science and Technology Information, South Korea
[b] Chungbuk National University, South Korea
[c] Anyang University, South Korea
[d] University of Maine, USA

ARTICLE INFO

Article history:
Received 13 March 2009
Received in revised form 15 March 2010
Accepted 16 March 2010
Available online 21 March 2010

Keywords:
Moving object index
Separated buffer node
Projection storage

ABSTRACT

A major issue in LBS (Location Based Service) is the handling of numerous historical moving object data, affecting query performance and service quality in application systems. In order to store and search lots of data rapidly, an effective index structure is required for improving not only the insertion method, but also the search performance.

In order to improve the performance of both applications, we propose the GIP$^+$ (Group Insertion tree with Projection Plus) for historical data management such as the trajectory of a vehicle. This index structure, based on the GIP, employs the separated buffer node method for reducing overlaps. The GIP$^+$ also uses projection storage for improving search performance by grouping the intersected child node in a node. Additionally, the link between the buffer nodes is designed to directly connect to the next buffer node. To effectively combine these methods and improve the performance, different node levels in the GIP$^+$ are also arranged for applying the separated buffer node, the projection storage, and the link. The designed historical index structure is useful for inserting and searching data which is arranged on a time axis.

1. Introduction

Since the beginning of history, maps have been utilized to represent the positions of spatial or moving objects and subsequently, allowed considerable social progress through travel, exploration, and transportation. With the advancement of technology, there has been a growing need for maps to locate information. It is a challenge to create effective maps for these purposes, especially since the information itself, data, frequently moved over time. These moving objects have the property of continuously changing their positions and/or shape over time (Forlizzi et al., 2000; Wolfson et al., 1998). Managing the numerous historical moving object data is very important in various historical data management applications of LBS (Location Based Service), because the data is endlessly accumulated over time. Therefore, the systems require an effective index technique to search multiple spatial and non-spatial elements. Research on such indices for moving objects has been published (Mokbel et al., 2003) which obtain useful ideas from the studies for spatial data such as the R-tree (Guttman, 1984) and B-tree. Several index techniques (Pelanis et al., 2006) are proposed for dealing with the historical, current and future positions of moving objects. Still, there is a need for better methods and because inserting data occurs more frequently than searching data, it behooves researchers to find ways to reduce insertion costs as well as search costs (Mokbel et al., 2003).

Challenges of the historical data management includes the bulk loading, the uncertainty management, the topological analysis, the update strategy, the packing, the clustering, the movement pattern analysis, and the effective data search for a time slice query, a spatiotemporal range query, and a trajectory query. For example, these techniques are widely used for LBS applications such as the analysis of the historical route of the bank burglar, the cargo transportation optimization, and the vehicle movement pattern analysis in a city. In this paper, we focus on the bulk loading, the data search for a time slice query and a range query. Traditionally, the R-tree inserts data by OBO (one by one) technique which loads data into an index in a sequential way (An et al., 2003). However, OBO needs more cost for insertion. Especially, the OBO method is not effective in a moving object management application, because a huge volume of moving object data is continuously created over time. To improve the loading efficiency, various bulk loading strategies have been proposed (Chen et al., 2002) such as packing (Kamel and Faloutsos, 1993), STLT (Chen et al., 2002). However, the bulk

* Correspondence to: Y.J. Jung, Department of Cyber Environment Development, Supercomputing Center, Korea Institute of Science and Technology Information, 335 Gwahangno, Yuseong, Daejeon 305 806, South Korea. Tel.: +1 207 581 6104; fax: +1 207 581 2206.
** Correspondence to: K.H. Ryu, School of Electrical & Computer Engineering, Chungbuk National University, Cheongju, Chungbuk 361 763, South Korea.
E-mail addresses: yjjung@kisti.re.kr (Y.J. Jung), khryu@dblab.chungbuk.ac.kr (K.H. Ryu).

0164-1212/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.03.041

loading method sometimes makes some problems to reduce the search performance such as overlaps. Reducing overlaps is required for rapidly accessing a large volume of data. In particular, in the R-tree based indices, it is crucial to minimize dead spaces and overlaps during data insertion, because the reduced overlaps are useful for improving their search performance (Lee et al., 2003). So, an effective index structure is needed to improve not only the loading efficiency but also the search performance.

We designed the GIP$^+$ (Group Insertion tree with Projection Plus), which is based on the GIP (Jung and Ryu, 2006a) as a one of approaches which effectively combine useful insertion and search methods. The GIP$^+$ employs the separated buffer node method for minimizing overlaps and the projection storage for quickly processing a time slice query. The GIP$^+$ also uses the link between the buffer nodes to improve search performance for a spatial range query, which searches all trajectories included in the rectangle of the range. The proposed index structure is used for handling historical data accumulated over time.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 describes the structure of the GIP$^+$, the separated buffer node approach, and the projection storage. Section 4 presents the algorithms in the GIP$^+$, and Section 5 covers the analyzed insertion cost of the GIP$^+$. Finally, Section 6 shows the experimental results about the insertion and the search performance, and Section 7 concludes.

2. Related work

An LBS system monitors the locations and states of moving objects. The system sends the information with a map to the client system depending on user's queries (Hu et al., 2005; Jung and Ryu, 2006b). It is important to be equipped with a rapid search function since the positions of the moving objects are frequently changed over the flow of time. Accordingly, index techniques (Chen et al., 2008) to deal with a great amount of information have been actively studied (Mokbel et al., 2003). Most moving object indices are based on the R-tree or the B$^+$-tree. With MBR (Minimum Bounding Rectangle), the R-tree based indices handle the line segment splits, which represent the trajectory of moving objects, as illustrated in Fig. 1. These MBRs are included in a node, which is a basic data unit in a tree. For example, the small MBRs, which have line segments, are contained in the leaf nodes. The large MBR, which encloses the small MBRs, is also included in the non-leaf node. Existing indices are roughly divided into two types according to the field of applications (Pfoser et al., 2000). One is the index for the history and trajectory. The other is the index which deals with current and future positions such as the QU-trade (Tzoumas et al., 2009) and the movement prediction (Zhang et al., 2009).

The insertion approaches of the existing indices are roughly classified into two approaches: top-down and bottom-up. The top-down approach inserts and searches data by finding a fit child

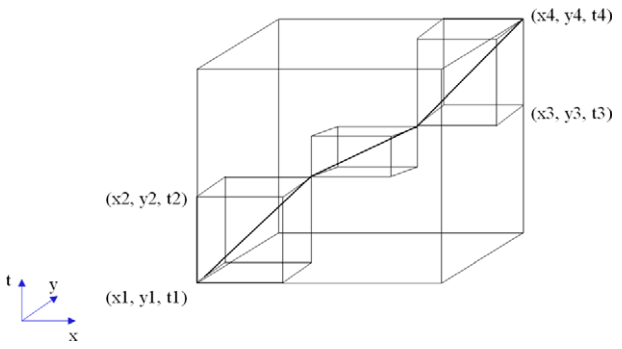


Fig. 1. Moving object trajectory representation with the R-tree.

node from the root to the leaf node. The TB-tree (Pfoser et al., 2000) finds the trajectory of a moving object rapidly by connecting the leaf nodes included in the same trajectory. The STR-tree (Pfoser, 2002) considers the spatial attributes and the trajectory protection of moving objects, but its performance is not as good as that of the R-tree. The TPR-tree (Saltenis et al., 2000) deals with the current and near future positions. The TPR reduces the frequency of updates by employing a time parameterized bounding with the function of time. The MP-tree (Lee et al., 2004) focuses on improving search performance in time slice query and temporal range query. The MP-tree utilizes projection storage and avoids unnecessary access by checking the boundaries of the projection storage. Its search performance is better than that of the STR-tree (Pfoser, 2002) and the R-tree in time slice and temporal range query. However, projection storage requires more memory space and its insertion performance is not as good as others.

The bottom-up approach quickly inserts and updates the data by using a hash table to insert the data directly into the leaf node such as the LUR-tree (Kwon et al., 2002), the bottom-up approach (Lee et al., 2003), and the TB*-tree (Lee et al., 2004). This method reduces the insertion cost of the top-down approach, because the bottom-up approach does not find the appropriate child node from the root. The LUR-tree uses the strategy to extend an MBR to reduce frequent updates, but its search performance is sometimes not as good as that of the R-tree. Also, The LUR-tree needs pointers to update the boundaries of parent nodes and the memory space for the hash table. The LGU (Lazy Group Update) algorithm (Lin et al., 2005) based on the TPR-tree has also been presented to reduce the update cost of current moving object data. A trajectory splitting model (Rasetic et al., 2005) has also been introduced for efficient spatiotemporal indexing through utilizing a cost model with a linear heuristic algorithm that minimizes the number of disk accesses. The single B$^+$-tree (Jensen et al., 2004) is utilized for partitioning moving object data according to their timestamp by dealing with moving object locations as vectors. The B^x tree structure based on the B$^+$-tree is grafted into the existing database system in a cost effective way. In insertion of the B^x tree, as the first range expires, a new range is appended.

The traditional OBO method of the R-tree based indices needs more insertion cost, when they inserts a large volume of data such as historical moving object locations. The effective bulk loading approaches, which improve the loading efficiency, have been proposed such as packing (Kamel and Faloutsos, 1993), STLT (Chen et al., 2002). An insertion process in the STLT (Small Tree and Large Tree) (Chen et al., 1998; Choubey et al., 1999; Chen et al., 2002) is described by two steps: clustering spatial data (making a small-tree) and inserting a small-tree into the large tree. The STLT has proven beneficial in cutting insertion index costs by inserting clustered data instead of a singular data. However, the cost of clustering data is high. In addition, this insertion method causes numerous overlaps in the large tree, because the small-tree already occupies a large area of clustering data. As a result, its search performance is sometimes weaker than that of the R-tree. This approach is also utilized to other indices for supporting the bulk loading such as GU-tree (Jung and Ryu, 2005), the bulk loading of the PMR quad tree (Hjaltason and Samet, 2002), BIOR (An et al., 2003). The GU-tree (Jung and Ryu, 2005) uses the buffer node approach to reduce the insertion cost, but it still has the overlaps by the buffer node. BIOR (Bulk Insertion in Oracle R-trees) (Hjaltason and Samet, 2002) is designed for minimizing overlaps with the restricted small-tree size. The quad tree buffer with the analyzed I/O and CPU cost BIOR (An et al., 2003) is used for rapidly construct PMR quad tree which handle the arbitrary spatial objects. To improve the update efficiency of TPR-tree with non-uniform datasets, a histogram-based bottom-up algorithm (HBU) (Lin and Su, 2004) is designed. The G-tree (Wang et al., 2005) shows good update performance by using the med-

ian-down approach to reduce the number of disk access. These index techniques are effective in processing spatiotemporal queries (Jensen et al., 2004; Patel et al., 2004; Yufei and Papadias, 2001). However, it is difficult to improve both insertion and search performance in most of the indices.

3. Group Insertion tree with Projection Plus

In this paper, we focus on the design of the historical index structure for combining the effective insertion and search methods. This index is used for handling the historical moving objects' locations that is the trajectory of a vehicle such as the STR-tree in dealing with temporal dimension in the category of spatiotemporal indexing methods (Mokbel et al., 2003). We did not consider the multi-version indices for the discrete changing of data such as the HR-tree, the MV3R-tree (Yufei and Papadias, 2001), and PIST (Botea et al., 2008).

3.1. The structure of GIP+

The proposed GIP+ is the index for handling past locations of moving points. Fig. 2 shows a set of nodes and how they are indexed by the GIP+ with a fanout of 3. The GIP+ consists of the non-leaf node with projection storage, the buffer node (non-leaf node), and a leaf node as shown in Table 1. In this paper, the buffer node like a small-tree of the STLT is a non-leaf node which is a parent node of a leaf node. The depth of the buffer node is limited to 1 for reducing the overlaps in the GIP+, because a small-tree with long depth causes numerous overlaps in the large tree with the large area of the small-tree.

The GIP+ employs the non-leaf node with the projection storage, which is designed to lower the count for checking the node's boundary during the data search (Lee et al., 2004). The storage is used for improving search performance for a time slice query even though making the storage requires extra insertion cost (Jung and Ryu, 2006a). However, when an index using the storage processes a range query, the index needs more node access: as many as the number of the projection storages in the range. To avoid having to provide this additional access, the GIP+ makes a link between the buffer nodes, which directly connects to the next buffer node. If all the nodes in the GIP+ have projection storages, it requires more space and greater insertion cost. The cost changes depending on the node level for utilizing the storage. Deciding the level for applying the storage is also important for the trade-off between the insertion and the search performance. If the buffer node uses the storage, the insertion cost steeply increases with additional memory space for the storage, because data is frequently inserted in the buffer node and the storage. The suitable level for using the projection storage in the GIP+ is the upper level of the buffer node in the test for finding the most effective level for the projection storage. Besides, the buffer node, the non-leaf node, is also used for improving the insertion performance by making a group of the inserted data and putting the group to an index (Jung and Ryu, 2006a).

Table 1 shows the form of a non-leaf node, a buffer node, and a leaf node. In a node, *rect* is an MBR and *ptr* is a pointer to indicate a child node, because the GIP+ is based on the R-tree. The non-leaf node with the projection storage uses *pid* for pointing to a projection storage, which has *seg* for indicating the overlapped boundary and *ptr* for storing the child nodes included in the line segment. The buffer node which is a parent node of the leaf nodes has *link* for pointing the next buffer node for rapidly searching data records. To avoid the overlaps, the GIP+ uses the separated buffer node which is modified from the buffer node of the GIP for minimizing the temporal boundary [t1, t2] of the buffer node.

3.2. The separated buffer node insertion approach

The top-down approaches require a data insertion cost which increases in proportion to the data frequency and the height of a tree. To reduce this insertion cost, the buffer node is processed according to the data group insertion steps: (1) inserting data to the buffer node and (2) inserting the buffer node to the GIP+ as shown in Fig. 3.

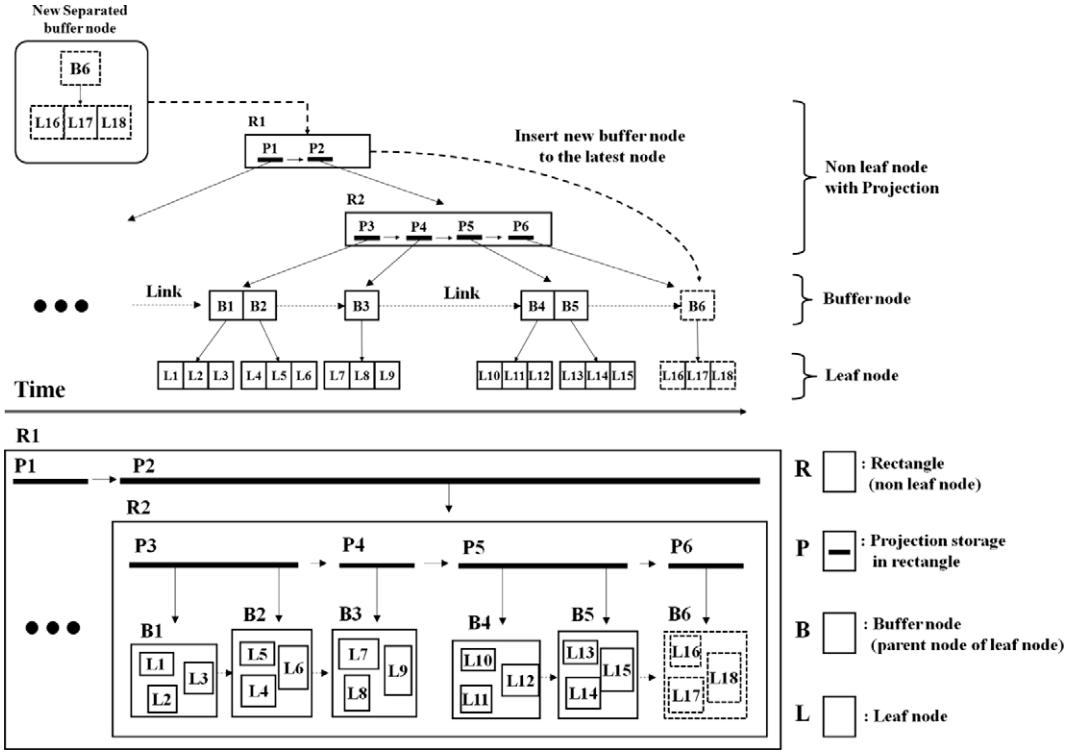


Fig. 2. Structure of the group insertion tree with projection.

Table 1
The node types of the GIP+.

Node type (form)		Entries
Non-leaf node with projection storage	Non-leaf node (*pid*, *rect*) Projection storage (*seg*, *ptr*)	*pid* is to indicate the projection storages *rect* is the MBR that encloses all the MBRs of child nodes *seg* is the line segment that encloses the overlapped boundaries of the child nodes *ptr* is the pointer to the child nodes which included in a line segment boundary
Buffer node (non-leaf node) (*ptr*, *rec*, *link*)		*ptr* is the pointer to the leaf nodes *rect* is the MBR that encloses all the MBRs of the child nodes *link* is to indicate next inserted buffer node
Leaf node (*oid*, *rect*)		*oid* is the identification of the object *rect* is the MBR of the object

Table 2
The separated buffer node.

Steps	Types Separated buffer node
Step 1	1. When an index receives new data, it inserts the data to the buffer node 2. If the buffer node is full or the temporal boundary of the received data record is beyond the defined temporal boundary of the buffer node, the GIP+ makes new buffer node for storing new data and go Step 2. (Ex) (Assumption) the buffer node has boundary [t1, t2] If an index receives a data record having boundary [t2, t3], it stops inserting data to the buffer node [t1, t2] and makes new buffer node [t2, t3] for storing new data [t2, t3]
Step 2	1. The separated buffer node [t1, t2] is inserted to the GIP+ 2. The buffer node is stored into the one of the newest nodes in the direction of the time axis in the large tree 3. GIP+ makes a link between the latest buffer node and new inserted buffer node
Effect	1. The separated buffer node like the small-tree of STLT reduces the insertion cost as much as the number of child nodes in the buffer node 2. The buffer node also improve the search performance for time point queries, but the insertion cost increases

M, the maximum number of entries in a node.

This buffer node method is based on the STLT (Chen et al., 1998; Choubey et al., 1999). The STLT makes some small-trees with data clustering and inserts the small-trees into a large tree. This STLT approach is useful for reducing the insertion cost by as much as the number of leaf nodes included in the small-tree, but the small-tree also makes some overlaps in the large tree, because the small-trees already have large MBRs. To have a small MBR, the size of a buffer node is defined as a non-leaf node which has a depth of one. To reduce the overlaps, the buffer node is also separated into small buffer nodes which have the same temporal period in the GIP+. The boundary of the buffer node is defined as small as possible. For example, the temporal boundary of the buffer node is defined as 10 min, because the simulated data is generated per every 10 min. These processing steps and the effects are described in Table 2.

Reducing an overlap is effective for improving the search performance, because the MBRs of the R-tree based indices are mostly allowed to overlap. Any range queries on the indices may result in multiple complete or partial paths being followed from the root to the leaf level. The more the overlap happens, the worse the branching behavior of a query becomes (Lee et al., 2003). Well organized nodes in an index are useful for improving space utilization and search performance.

To show the effect of the buffer node, Fig. 4 presents an example of the insertion cost of the separated buffer node under the assumption that $K = 5, 20$, $M = 20$ and the time period of the inputted leaf nodes are same. The buffer node approach inserts only one full buffer node to the large tree instead of M leaf nodes. The cost of the first step is at most M, because the buffer node can store as many as M leaf nodes. The cost of the second step is $1 \times$ (the height of the large tree $- 1$). The height of the tree is 1 less than the height of the top-down tree, because the buffer node – a non-leaf node which already has the leaf nodes – will be on the parent node level of a leaf node in the GIP+. The additional examples for evaluating the insertion cost of the separated buffer node are shown in Tables 4 and 5 in Section 5.

3.3. The projection storage

The storage is generally used for improving search performance of a time slice query, but the storage also requires extra insertion cost in order to make the projection storage (Lee et al., 2004). To reduce the insertion cost, the projection storage is used for combining only the temporal boundaries of the child nodes, because the storage demands a greater insertion cost for gathering the boundaries for all axes such as *x*, *y*, and time. In our experiment,

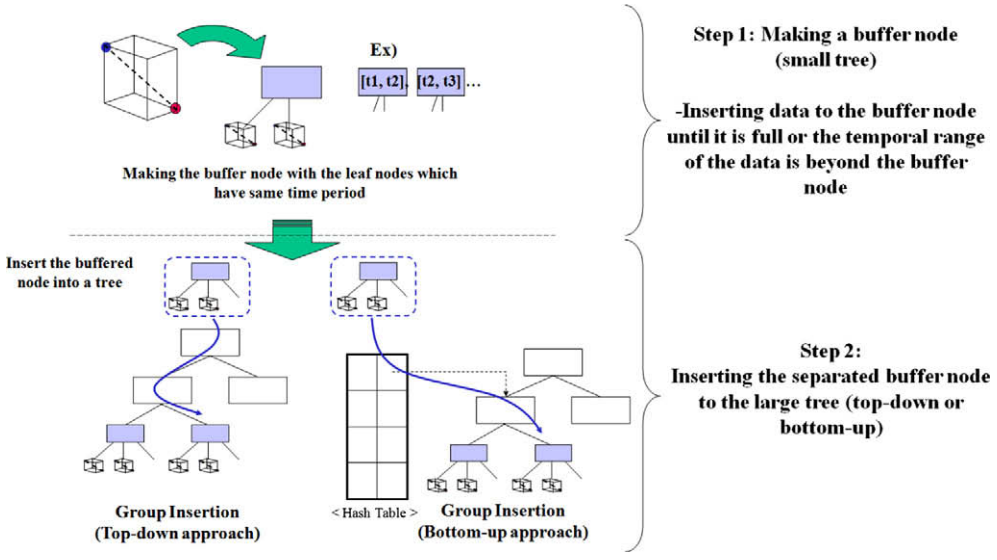


Fig. 3. The separated buffer node insertion approach.

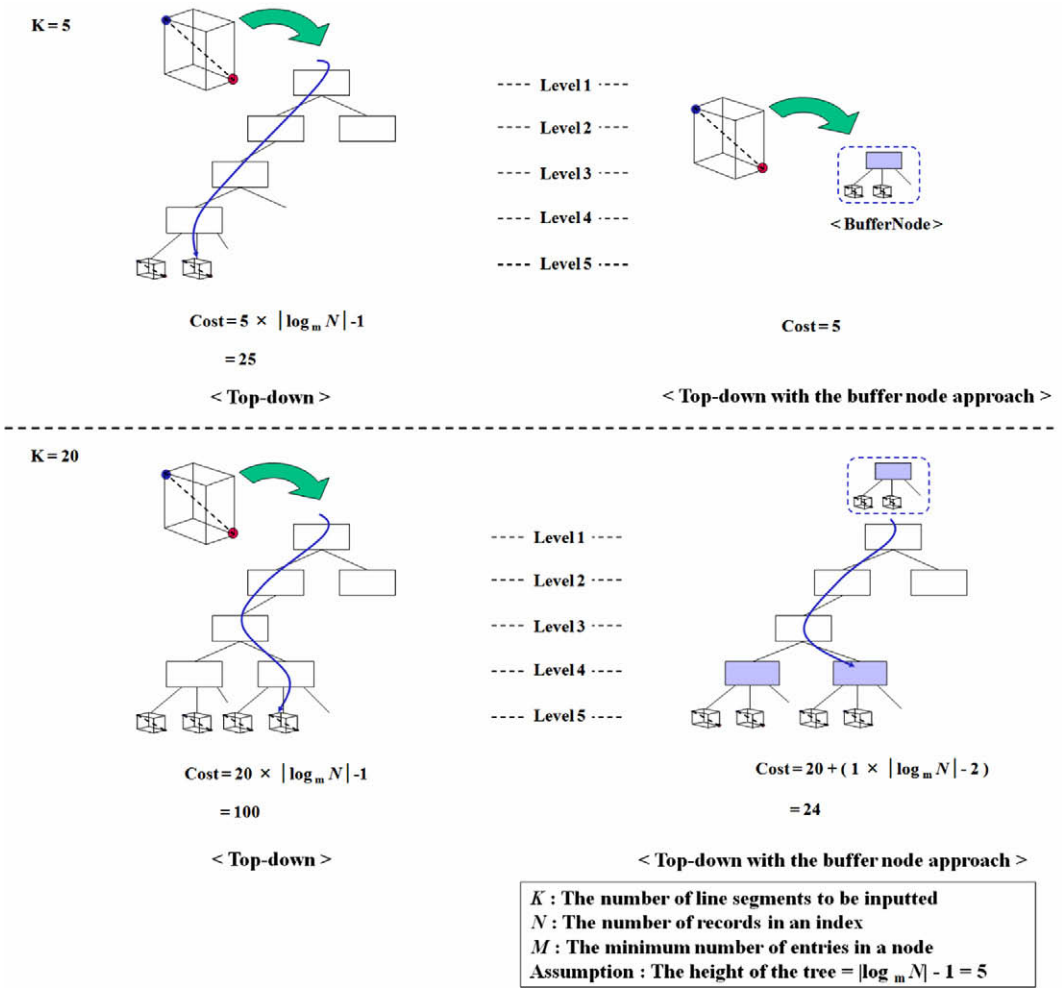


Fig. 4. The insertion cost of the buffer node approach.

Table 3
The boundaries of the nodes in the MBR.

Terms	Description
K	The number of line segments to be inputted
N	The number of records in an index
m	The minimum number of entries in a node
M	The maximum number of entries in a node
$\lvert\log_m N\rvert - 1$	The height of an index

Table 4
The insertion cost in the top-down approaches.

Insertion method	Insertion cost	
Top- down	Total cost	= the number of insertion records × the height of the tree $= K \times (\lvert\log_m N\rvert - 1)$
Buffer node with top-down	Total cost	= the cost in the buffer node + the cost in the large tree $= K + \{(K/M\} \times (\lvert\log_m N\rvert - 2)\}$
	∵ the insertion cost in the large tree	= the number of insertion buffer nodes × the height of the large tree $= (K/M\} \times \{(\lvert\log_m N\rvert - 1) - 1\}$
Insertion cost difference		= Top-down cost − buffer node cost $= \{K \times (\lvert\log_m N\rvert - 1)\} - \{K + K/M \times (\lvert\log_m N\rvert - 2)\}$ $= (K - K/M) \times (\lvert\log_m N\rvert - 2)$

Table 5
Example of the insertion cost.

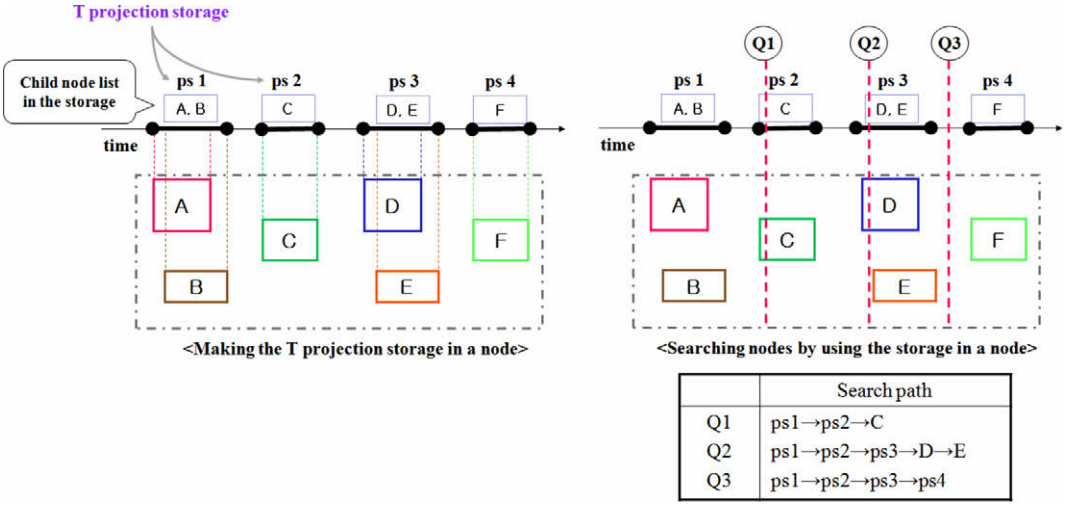
Insertion manner	Insertion cost
Simulated values	K = 100,000, m = 10, M = 20, N = 1000,000 Height of tree = $\lvert\log_m N\rvert - 1 = \lvert\log_{10} 10^6\rvert - 1 = \lvert 6\rvert - 1 = 5$
Top-down	$= K \times (\lvert\log_m N\rvert - 1) = 100{,}000 \times 5 = 500{,}000$
Buffer node with top-down	$= K \times \{(K/M\} \times (\lvert\log_m N\rvert - 2)\} = 120{,}000$
Cost difference	$= (K - K/M) \times (\lvert\log_m N\rvert - 2) = 380{,}000$

it turns out that using the projection storages of all axes requires almost three times as much memory space as utilizing only the storage for time axis.

Fig. 5 shows the boundaries of child nodes and projection storages. The storage is a repository for the ranges and the pointers to indicate the child nodes contained in the range. These ranges are useful for avoiding needless node access which does not satisfy the queries. The storage is also effective for dead space, because the dead space is not included in the range of the storage. Especially, in the search path of Q3, an index checks only projection storages, not the boundaries of the child nodes. Minimization of dead spaces directly contributes to improving the ability to find appropriate data during the data search process (Lee et al., 2003). The GIP$^+$ reduces the search cost by checking the range of storage, not all boundaries of nodes. Good search performance has been demonstrated in the experiment by reducing the number of node accesses.

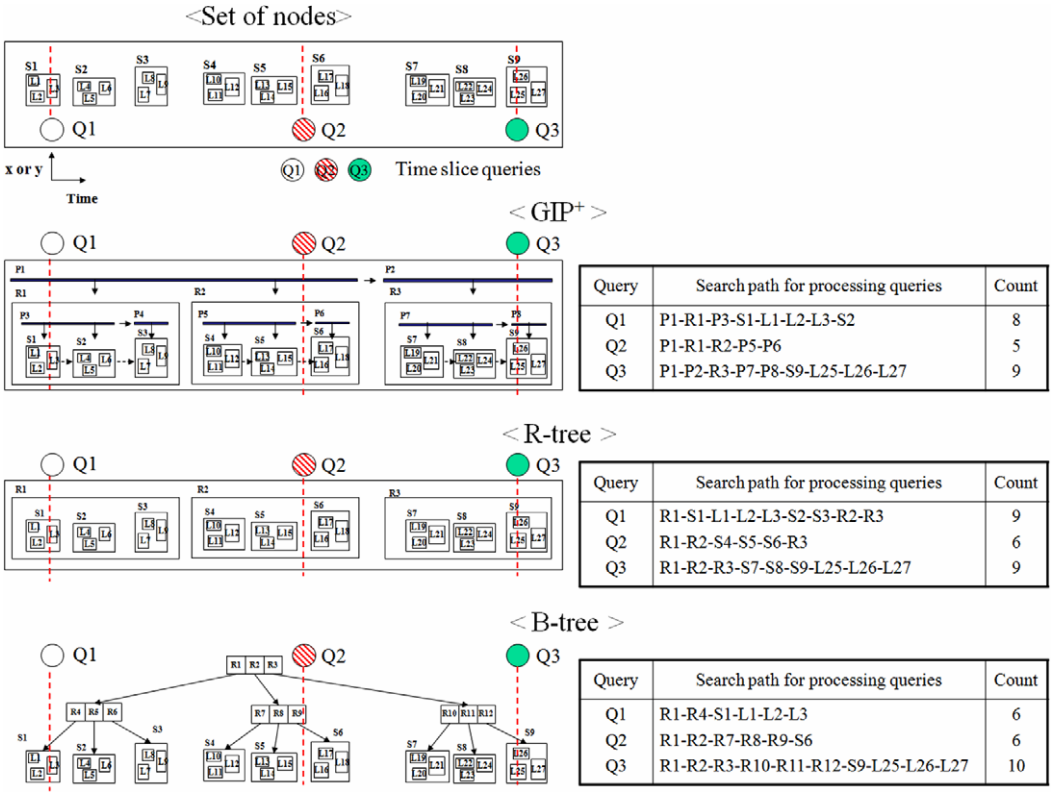
To show the usage of projection storage, we describe the examples for searching data in the GIP+, the R-tree, and B-tree as shown in Fig. 6. The example shows the search path and the count for checking nodes boundaries with the different structures. In the GIP+, there are two steps for checking the boundary in a node. First, the GIP+ finds the temporal boundaries of the storages which intersect the query range. Next, the spatiotemporal boundary of the node is checked in the storage. The GIP+ does not check the all child nodes in a node against most of the R-tree based indices by using the projection storage. If we consider only the time axis, the GIP+ is similar to the B-tree. The non-leaf node of the B-tree has the number of child nodes changed depending on the balance of the tree. However, the GIP+ has a flexible number of the storages and their ranges which are changed by the overlapped boundaries of the child nodes. This point, deciding on the performance of the indices, is very interesting: if the child nodes are divided into several groups such as the separated buffer node, the performance of time point queries in the GIP+ is better, as shown in the test. The effect of the storage is especially shown in Q2 (Query 2). The effect could be improved by a greater degree. The GIP+ reduces the search cost by checking the storage ranges. However if a node only has storage, the performance of the B-tree is better than the GIP+.

In wide range temporal queries, checking the storage requires a greater search cost, because two steps are required for each node. Originally, the search method using the storage checks all the boundaries of the nodes and the storage ranges which are included in the range of the query. However, the storage in the GIP+ requires a few counts for checking the boundaries in the test, because the storage is applied in only the upper nodes of the buffer node. Furthermore, the link between the buffer nodes is also used for reduc-



	Search path
Q1	ps1→ps2→C
Q2	ps1→ps2→ps3→D→E
Q3	ps1→ps2→ps3→ps4

Fig. 5. The projection storage for time axis in the MBR.



Query	Search path for processing queries	Count
Q1	P1-R1-P3-S1-L1-L2-L3-S2	8
Q2	P1-R1-R2-P5-P6	5
Q3	P1-P2-R3-P7-P8-S9-L25-L26-L27	9

Query	Search path for processing queries	Count
Q1	R1-S1-L1-L2-L3-S2-S3-R2-R3	9
Q2	R1-R2-S4-S5-S6-R3	6
Q3	R1-R2-R3-S7-S8-S9-L25-L26-L27	9

Query	Search path for processing queries	Count
Q1	R1-R4-S1-L1-L2-L3	6
Q2	R1-R2-R7-R8-R9-S6	6
Q3	R1-R2-R3-R10-R11-R12-S9-L25-L26-L27	10

Fig. 6. The search path and the count for time slice queries in the GIP+, the R-tree, and the B-tree.

ing the number of node accesses. When the GIP+ processes the range queries, first, it finds a buffer node for satisfying the low boundary of the range query. Next, it finds another buffer node included in the query by following the link until the boundary of the linked buffer node intersects the upper boundary of the range query. Thus, the GIP+ improves the cost of search performance for range queries in the GIP+.

4. The Algorithms in the GIP+

To explain the insertion and the search processing steps in the GIP+, Algorithms 1–3 show the separated buffer node insertion approach and the search with the projection.

Algorithm 1. The data insertion into the separated buffer node.

```
Algorithm Insert_data_to_separated_buffer_node(class node *root, class node *entry)
input: root // node of a tree
       entry // information of moving objects
method:
  // The buffer node is a non-leaf node which has 1 depth such as small-tree in STLT
  if buffer node is full or entry 's time is beyond the old_entry's time then
    insert_buffer_node_to_GIP_plus (root, the buffer node)
  // insert the buffer node into GIP+
    make new buffer node
  endif
  insert the entry into the buffer node
  old_entry ← entry // old_entry is global variable
end
```

Algorithm 2. The buffer node insertion into the GIP+.

```
Algorithm Insert_buffer_node_to_GIP_plus (class node *root, class buffernode *buffer_entry)
input: root // node of a tree
  buffer_entry // the buffer node, non-leaf node which will be inserted in GIP+ (the large tree)
method:
  if root is full then
    // Put the new buffer_entry into new node and keep the old one as it is
    make new root for inserting buffer_entry and adjust tree
  endif
  find the latest T projection in root
  if buffer_entry's boundary does not intersect the T projection's boundary then
    make new T projection storage
  endif
  insert buffer_entry into the T projection
  find the closest node in the T projection
  if buffer_entry's level is the child node level of the closest node then
    insert buffer_entry into the closest node
    make a link between latest buffer node and new inserted buffer_entry // for range queries
  else
    insert_node(the closest node, buffer_entry)
  endif
end
```

Algorithms 1 and 2 show the separated buffer node insertion method. When a non-leaf node in the GIP+ becomes full during the insertion process, the GIP+ simply creates a new node. The GIP+ does not split the node, and instead inserts data into the new one at non-leaf node level like TB-tree (Pfoser et al., 2000). Since a new node is created at the non-leaf node level, creating new node also serves to reduce the maintenance cost for the balance of the tree. Additionally, it is important to decide the proper size of the buffer node. The insertion cost increases in proportion to the size of a buffer node, but its search cost also increases as much as the size of a buffer node, because the buffer node creates overlaps in an index such as the large clustering small-trees in STLT (Choubey et al., 1999). The size of a buffer node is basically equal to that of the non-leaf node. If GIP+ uses a small buffer node such as the separated buffer node, its search performance increases with the projection storage, but its insertion performance decreases as much as the empty space of the buffer nodes.

Algorithm 3. The data search for a range query

```
Algorithm search_node(class node *root, class range *query)
input: root // non of tree
  query // query range
output: result set // set of leaf nodes
method:
  find the closest T projection and its intersected child node for satisfying the low boundary of the query
  if query' low boundary intersect the node's boundary in T projection then
    if the node is the buffer node then
      add the satisfied leaf node of the buffer node to the result set // result set
      follow the link to next buffer node and put its leaf nodes to the result set
      until the leaf node boundary intersects the upper boundary of the query
      return result set
    else
      search_node(the intersected node, query)
    endif
  endif
  return null
end
```

The GIP+ searches data for satisfying queries according to the search path with the projection storage as shown in Fig. 6. When a range query is processed, the GIP+ processes a time slice query for finding the earliest node of the query range. Then, the next data records is searched by following the link between the buffer nodes for avoiding the unnecessary node access which checks the ranges of other storages as shown in Fig. 10.

5. The insertion cost analysis in the GIP+

In this section, to concentrate on the insertion cost, we did not consider the *AdjustTree* algorithm and the cost depending on the change in depth of the trees. Table 3 lists the terms used to analyze the insertion cost: K means the number of leaf nodes, N records are already inserted in the indices.

To simply calculate the effect of the buffer node approach, we assume that every separated buffer node has same time period during data insertion. Table 4 shows the insertion cost of the top-down and the separated buffer node approaches. The buffer node method has the insertion costs in two aspects: the cost in

Table 6
Workload parameters.

Query	Parameter	Data range				
Data insertion	The number of records (the number of objects)	100,000 (100)	250,000 (100)	500,000 (100)	750,000 (100)	1000,000 (100)
	The number of records (the number of objects)	500,000 (50)	1000,000 (100)	1500,000 (150)	2000,000 (200)	2500,000 (250)
Time Slice query	Spatial range (%)	1, 3, 5, 7, 10, 25, 50, 75, 100				
	Time point (%)	10, 25, 50, 75, 90				
Spatiotemporal range query	Spatial range (%)	1, 3, 5, 7, 10, 25, 50, 75, 100				
	Temporal range (%)	1, 5, 10, 15, 20, 50, 100				

the buffer node and the cost in the large tree. In the GIP$^+$, the K/M buffer nodes are inserted into the large tree, because each buffer node has M leaf nodes.

As shown in the cost difference in Table 4, if $(K - K/M) > 0$ and $|\log_m N| - 2 > 0$, we prove the advantage of the proposed buffer node insertion method. First, $(K - K/M) > 0$ is satisfied, because K and M are greater than 0. Moreover, if $N \geqslant {}_m3$ ($\because |\log_m m^3| = 3$) is satisfied, $|\log_m N| > 2$ is also satisfied. Therefore, the buffer node method is effective when the height of a tree is greater than 2. An example with sample figures is shown in Table 5.

The effect of the buffer node is reassured by the difference between two indices with the simulated values in Table 5. The more K and M increase, the better the effect of the buffer node method gets. However, in the GIP$^+$, the buffer node is separated according to the time period of leaf nodes which is included in the buffer node. So, the insertion cost increases as much as the empty space of the buffer node, because the number of separated buffer nodes increases.

6. Performance evaluation

In the test, we determine the number of checking trials of the node's boundaries during the data insertion and the search in the indices such as the GIP$^+$, the GIP, the MP-tree, the GU-tree, the STR-tree and the R-tree with both top-down and bottom-up approaches. All indices treat time as an extra spatial dimension like Fig. 1. The number of node access is used for performance evaluation as shown in Fig. 6, because the time for processing queries can be changed depending on the hardware conditions such as CPU, RAM, and disk performance.

6.1. Experimental setup

All data records in these indices are stored in the main memory, not on a disk. A data generator is used to generate the simulated random positions of 100 objects every ten minutes like GSTD (Theodoridis and Nascimento, 2000). Each position is a two-dimensional data point within a square [in this case, 40,000 × 40,000 data points] that is movable within a radius of 30 points for each sampling interval. This moving object data that is the historical trajectory of a vehicle is different with the boundary of county such as TIGER (TIGER). The generated historical data, trajectory of a vehicle, is flexibly changed depending on the movement of a vehicle. The data generator is used for this experiment, because it is hard to get real data such as 1000,000 line segments of trajectories. The workload parameters used are summarized in Table 6. In this paper, we evaluated the cost of data insertion, a time slice query, and a range query.

In Table 6, the percent figures describe the ratio of the time point or the range size of the queries to the entire spatial or temporal range. For example, if a query has a 50% spatial range, the range is one half of the entire spatial range, for example 20,000 × 40,000. The insertion and the search performance are changed depending on the part of entire area even though queries have the same range (%), because the number of moving objects of the partial area is different. In our test, the average of the number of node accesses after searching all of the divided parts of the entire area with the defined range of a query is presented. These experimental parameters are derived from the test in (Pfoser et al., 2000) such as the temporal range (20%).

6.2. The insertion cost analysis

The insertion cost grows depending on the number of inserted data as shown in Fig. 7. The MP-tree (top-down) shows the worst insertion performance, because its insertion process needs more node access for making the projection storages for all axes. The advantage of the buffer node approaches is proved in the GU-T, the GU-B, and the GIP$^+$ performance. The GU-T is judged to be a

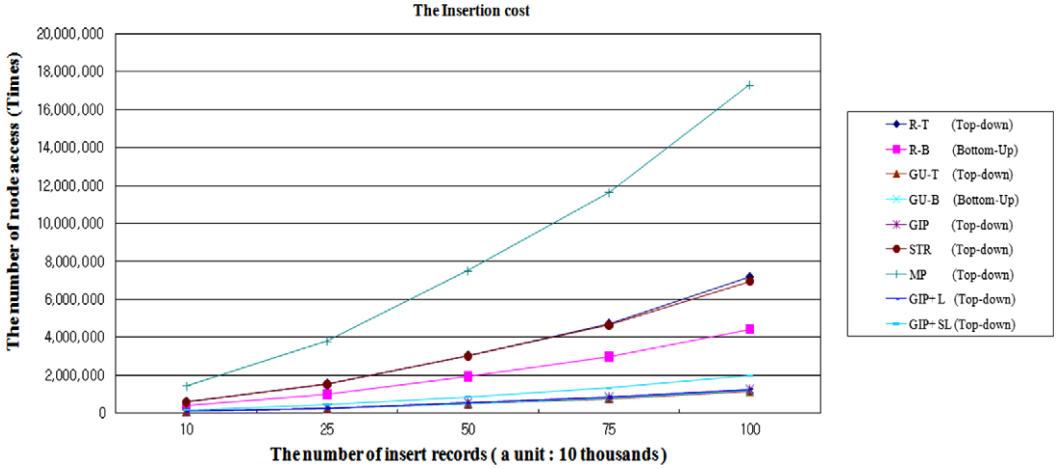


Fig. 7. The insertion cost analysis of the indices.

little better than the GU-B, because the GU-B has to check the hash table for the data insertion. This is the difference between the GU-T and the GU-B.

The GIP and the GIP$^+$ are less effective than the GU-tree, because they require the additional insertion cost and space for the projection storage in the non-leaf node. To determine the effect of the separated buffer node, we devise two types of GIP$^+$; the GIP$^+$ L which uses the link between buffer nodes, and the GIP$^+$ SL which utilizes the separated buffer node and the link between buffer nodes. It was found that the insertion performance of the GIP$^+$ L is better than that of the GIP$^+$ SL because the separated buffer node did not use its full space and consequently, reduces its effect. However, the GIP$^+$ SL shows good search performance for time slice query in Figs. 8 and 10, because the separated buffer node reduces overlaps between nodes. As a result of the test, the insertion cost is GU-T ≒ GU-B <≒ GIP ≒ GIP$^+$ L < GIP$^+$ SL < R-B < STR <≒ R-T < MP.

6.3. The search cost analysis

The search performance of the GIP$^+$ is also analyzed according to the items in Table 6, because most indices have a trade-off between the insertion and the search performance. For example, the LUR-tree has good update performance, but sometimes its search performance is not as good as that of the R-tree.

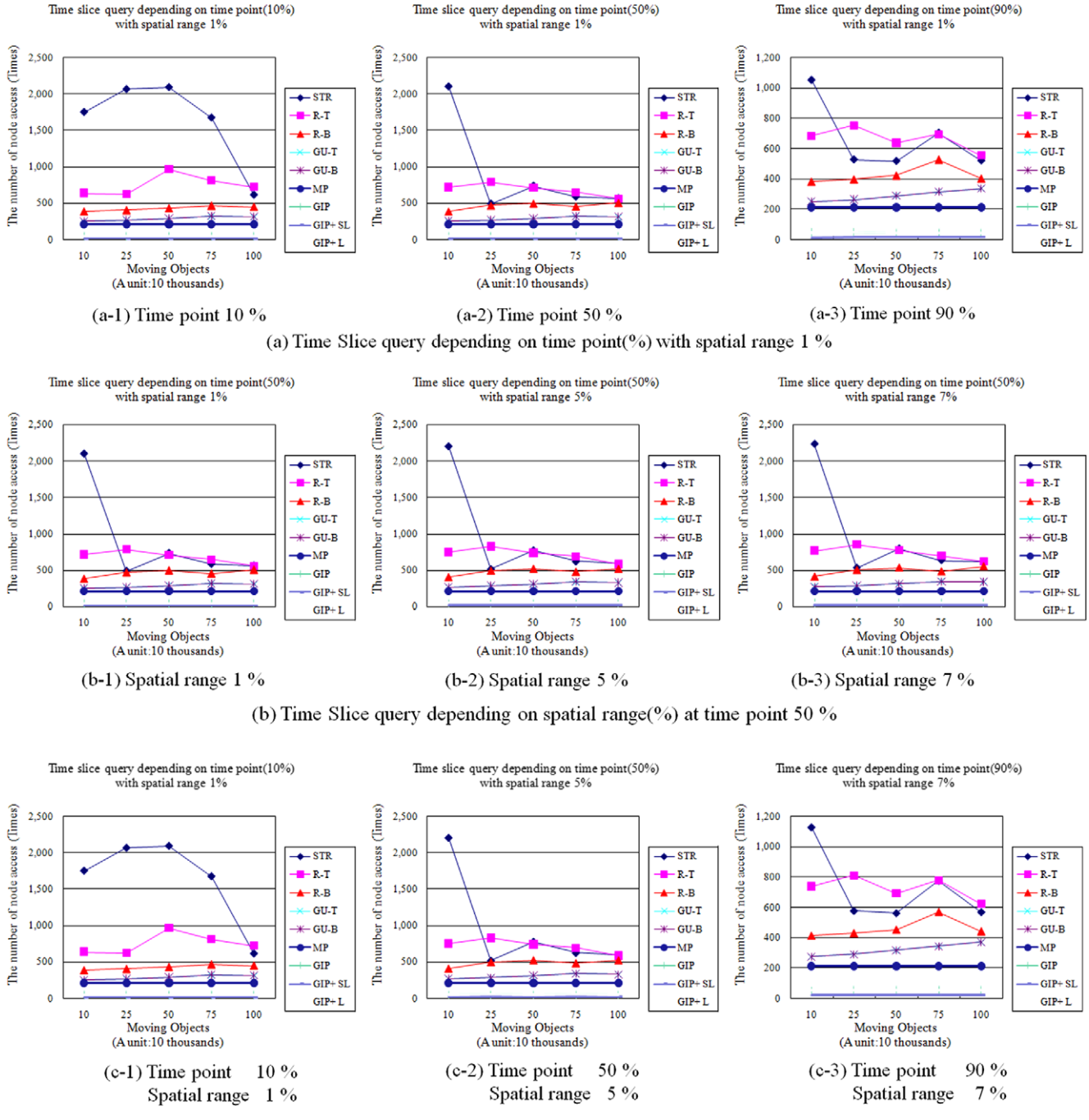


Fig. 8. The search cost for the time slice query depending on the spatial range (%) and the time point (%).

The performances of the STR, the R-T, and the R-B are frequently changed depending on the time point (%) and spatial range (%) in Fig. 8, because random spatial distribution could affect them. The performance of the GIP+, the GIP and the MP-tree is better than others by using the projection storage. The GIP+ SL is better than the GIP and GIP+ L, because the separated buffer node reduces the overlap between the buffer nodes in a tree. The spatial range (%) could weakly affect the performance for the time slice queries in the GIP+, but it strongly influences the performance in the spatiotemporal range queries. The search cost for the time slice query is GIP+ SL ≤≒ GIP+ L ≒ GIP ≤≒ MP < GU-T ≒ GU-B < R-T ≒ R-B ≤≒ STR.

The GIP and the GIP+ show good performance in the results for queries (a) in Fig. 9, because the projection storage is useful for searching data in time slice query even though temporal range is 100%. In particular, when the temporal range is 1%, the performance shows that the projection storage is effective. The projection storage in the GIP and MP-tree is not effective for the range queries with widen spatial range, because the storage has the only summarized temporal range of a child nodes for quickly processing a time slice query. However, in the result of Fig. 9, the storage does not require not much additional cost, because of the link between the buffer nodes. The GIP+ SL and the GIP+ L are better than the GIP, because they quickly search the next data by following the link between buffer nodes after the time slice query. The GIP needs to check the projection storages which are included in the query range, because the GIP has no link between the buffer nodes. In range queries (b) with changed spatial range (%), the results of

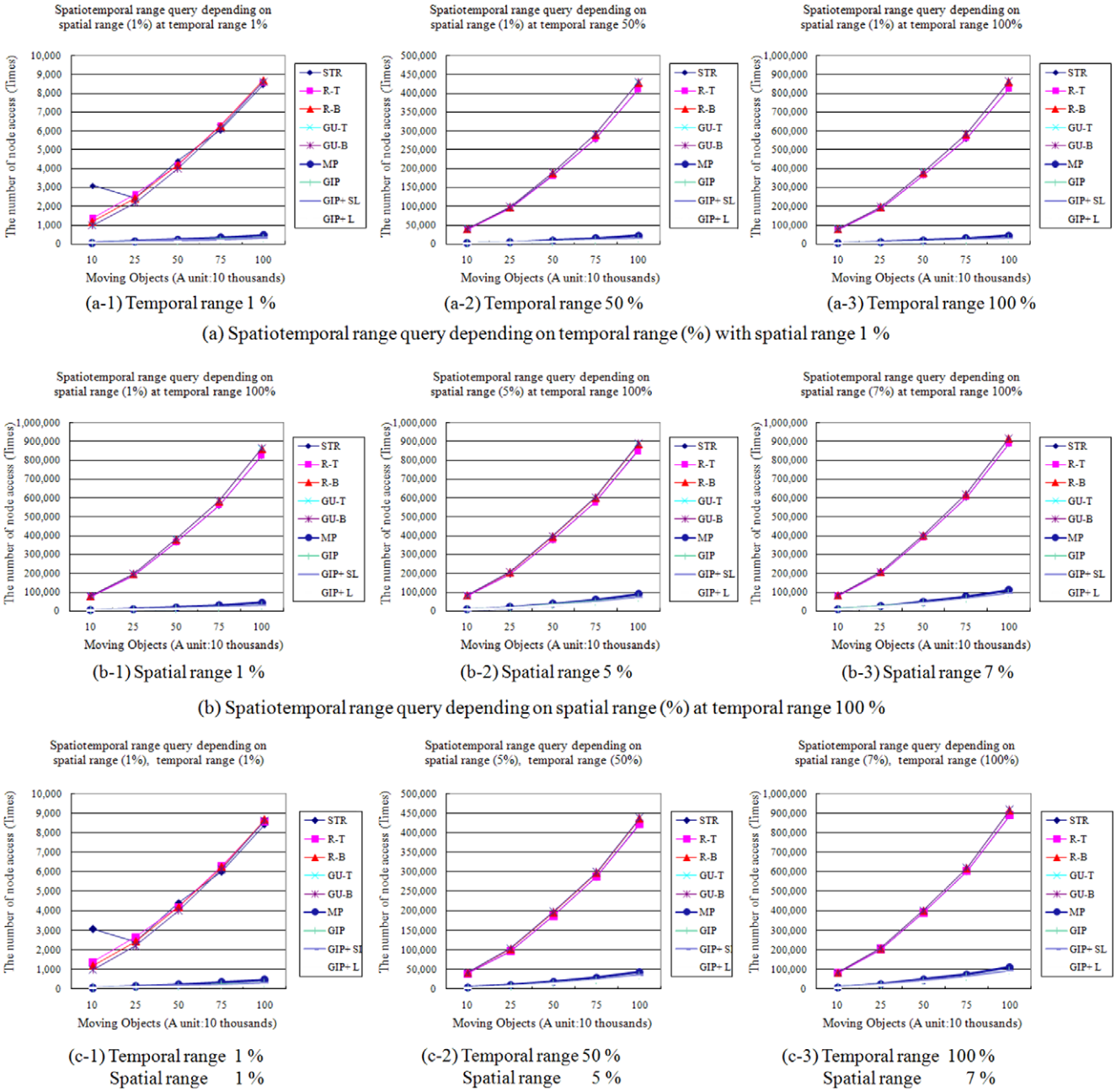


Fig. 9. The search cost for the range query depending on the spatiotemporal range (%).

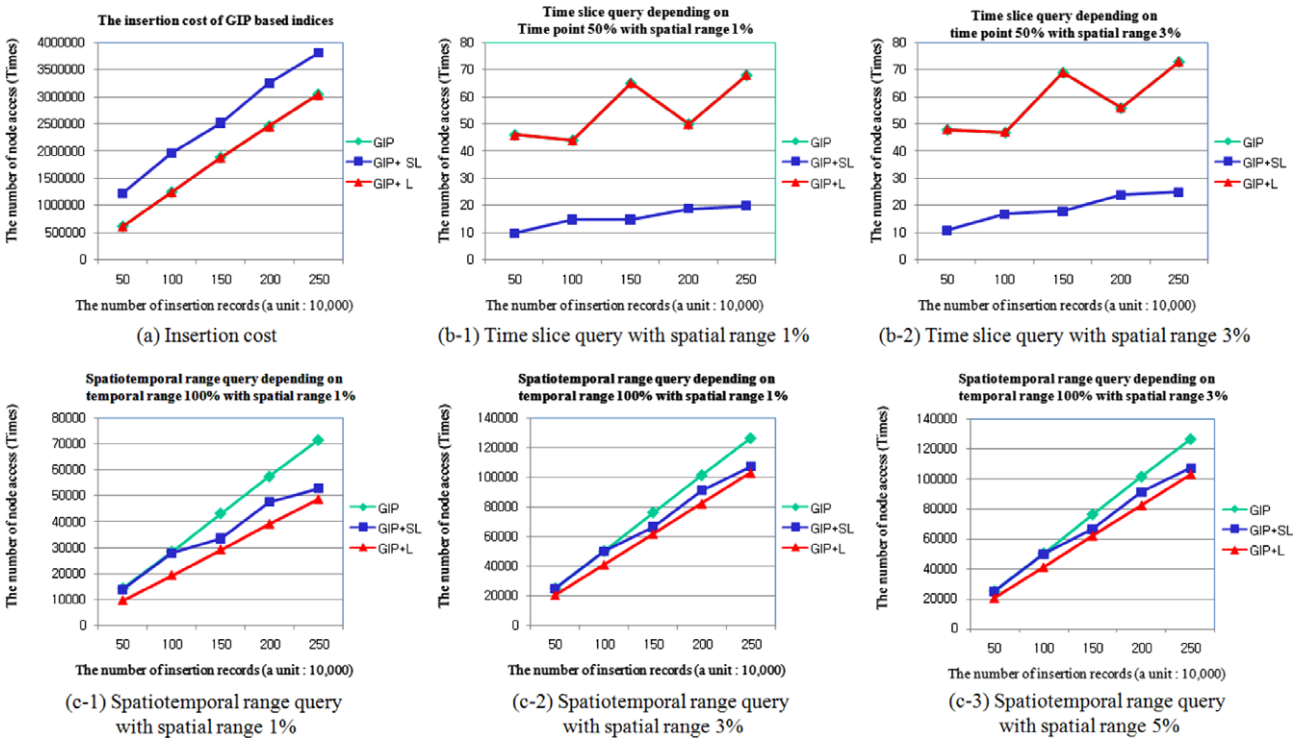


Fig. 10. The insertion and the search cost of the GIP and the GIP$^+$.

the GIP, the GIP$^+$ SL, and the GIP$^+$ L are rapidly changed. The result means that the spatial ranges mostly influence the performance of the GIP and the GIP$^+$. However, the performance of GIP, the GIP$^+$ SL, and the GIP$^+$ L is not worse than the one of MP-tree and GU-T even though 100% spatial range. The search cost for the spatiotemporal range query is GIP$^+$ L $\leqq$ GIP$^+$ SL $\leqq$ GIP $\leqq$ MP < GU-T $\fallingdotseq$ GU-B $\leqq$ R-T $\leqq$ R-B $\leqq$ STR.

To evaluate the differences among the GIP indices such as the GIP, the GIP$^+$ L, and the GIP$^+$ SL, Fig. 10 shows the results for insertion and search costs with other simulated data records (500,000–2500,000) which depend on the changed number of objects such as 50, 100, 150, 200, and 250. In the insertion cost (a), the GIP$^+$ L and the GIP present the insertion cost lower than the GIP$^+$ SL. The separated buffer node of the GIP$^+$ SL needs additional insertion cost for storing the increased buffer nodes because its buffer nodes are separated. The order of insertion cost is GIP$^+$ L $\fallingdotseq$ GIP $\leqq$ GIP$^+$ SL. In a time slice queries (b) at time point 50% with spatial range 1% and 3%, the GIP$^+$ SL shows less cost than others, because the less overlaps reduce the node accesses. The order of the search cost of this slice query is GIP$^+$ SL < GIP$^+$ L $\fallingdotseq$ GIP. In the range queries (c) with temporal range 100% and spatial range 1%, 3%, and 5%, the order of the search cost of the range query is GIP$^+$ L $\leqq$ GIP$^+$ SL < GIP. The GIP$^+$ L and the GIP$^+$ SL have less cost than the GIP, because following the link reduces the number of node access. The GIP$^+$ L shows good performance, because the GIP$^+$ L has nodes less than the GIP$^+$ SL. The performance of the GIP$^+$ SL is closed to one of the GIP$^+$ L, because the GIP$^+$ SL has good performance for the time slice query of the search algorithm. If the query keeps a narrow spatial range, the GIP$^+$ SL also shows good performance in a range query.

In the test about the size of the trees, the total size of non-leaf nodes is GU-T $\fallingdotseq$ GU-B $\leqq$ GIP $\leqq$ GIP$^+$ L $\leqq$ R-B < R-T < GIP$^+$ SL < MP < STR. The GIP$^+$ SL has more nodes which are made by the separated buffer node method. This separated buffer node makes the search cost high as much as the increased number of nodes. However, the GIP$^+$ SL is useful for processing the time point and the range queries with narrow spatial range such as 1%, 3%, and 5% even though its big size and its high insertion cost. When a temporal range query is processed, the GIP$^+$ L is useful for inserting and searching historical data which is continuously arranged on a time axis.

7. Conclusions

Numerous moving object data, the continuously created data with time flow, causes the high cost for the insertion and the search in moving object indices. In order to cost effectively reduce both, we propose the GIP$^+$, designed for effectively combining the advantages of the buffer node method and the projection storage. In the test, the GIP$^+$ L is effective for reducing the insertion and the search cost for processing a range query. The GIP$^+$ SL which uses the separated buffer node shows good performance in a time slice query and a range query with narrow spatial range.

In future studies, the GIP$^+$ structure will be applied and evaluated on various LBS applications, including the complement for processing spatial range queries and other queries such as aggregation queries. To improve the insertion and search performance, We will study the adjustable temporal boundary of the separated buffer node. In order to detect the environmental conditions for the LBS, the GIP$^+$ will be extended for the processing of great amounts of geosensor data.

Acknowledgements

This research was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD). No. KRF-2007-357-D00206, Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF No. 2010-0001732)

and (NRF No. 2009-0052728), and by the grant of the MEST (The Regional Core Research Program/Chungbuk BIT Research-Oriented University Consortium).

References

An, N., Kanth, R., Kothuri, V., Ravada, S., 2003. Improving performance with bulk-inserts in Oracle R-trees. In: 29th VLDB, pp. 948–951.
Botea, V., Mallett, D., Nascimento, M.A., Sander, J., 2008. PIST: an efficient and practical indexing technique for historical spatio-temporal point data. Geoinformatica 12 (2), 143–168.
Chen, L., Choubey, R., Rundensteiner, E.A., 1998. Bulk insertions into R-trees using the small-tree-large-tree approach. In: ACM GIS Workshop, pp. 161–162.
Chen, L., Choubey, R., Rundensteiner, E.A., 2002. Merging R-trees: efficient strategies for local bulk insertion. GeoInformatica 6 (1), 7–34.
Chen, S., Jensen, C.S., Lin., D., 2008. A benchmark for evaluating moving objects indexes. In: VLDB, pp. 1574–1585.
Choubey, R., Chen, L., Rundensteiner, E.A., 1999. GBI: a generalized R-tree bulk-insertion strategy. In: Symposium on Large Spatial Databases, pp. 91–108.
Forlizzi, L., Guting, R.H., Nardelli, E., Schneider, M., 2000. A data model and data structures for moving objects databases. In: ACM SIGMOD, pp. 319–330.
Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. In: ACM-SIGMOD, pp. 47–57.
Hjaltason, G.R., Samet, H., 2002. Speeding up construction of PMR quadtree-based spatial indexes. VLDB Journal 11 (2), 109–137.
Hu, H., Xu, J., Lee, D.L., 2005. A generic framework for monitoring continuous spatial queries over moving objects. In: ACM SIGMOD, pp. 479–490.
Jensen, C.S., Lin, D., Ooi, B.C., 2004. Query and update efficient B^+-tree based indexing of moving objects. In: VLDB, pp. 768–779.
Jung, Y.J., Ryu, K.H., 2005. A group based insert manner for storing enormous data rapidly in intelligent transportation system. In: ICIC, pp. 296–305.
Jung, Y.J., Ryu, K.H., 2006a. Group insert tree with projection for handling vehicle information effectively in LBS. In: XWICT, pp. 105–112.
Jung, Y.J., Ryu, K.H., 2006b. The vehicle tracking system for analyzing transportation vehicle information. In: APWeb Workshops (ICSE), pp. 1012–1020.
Kamel, I., Faloutsos, C., 1993. On packing R-trees. In: CIKM, pp. 490–499.
Kwon, D.S., Lee, S.J., Lee, S.H., 2002. Indexing the current positions of moving objects using the lazy update R-tree. In: MDM, pp. 113–120.
Lee, E.J., Ryu, K.H., Nam, K.W., 2004. Indexing for efficient managing current and past trajectory of moving object. In: Apweb, pp. 781–787.
Lee, E.J., Jung, Y.J., Ryu, K.H., 2004. A moving point indexing using projection operation for location based services. In: 9th DASFAA, pp. 775–786.
Lee, M.L., Hsu, W., Jensen, C.S., Cui, B., Teo, K.L., 2003. Supporting frequent updates in R-trees: a bottom-up approach. In: VLDB, pp. 608–619.
Lin, B., Su, J., 2004. On bulk loading TPR-tree. In: MDM 04, pp. 114–124.
Lin, B., Su, J., 2005. Handling frequent updates of moving objects. In: CIKM, pp. 493–500.
Mokbel, M.F., Ghanem, T.M., Aref, W.G., 2003. Spatio-temporal access methods. IEEE Data Engineering Bulletin 26 (2), 40–49.
Patel, J.M., Chen, Y., Chakka, V.P., 2004. STRIPES: an efficient index for predicted trajectories. In: ACM SIGMOD, pp. 637–646.
Pelanis, M., Saltenis, S., Jensen, C.S., 2006. Indexing the past, present and anticipated future positions of moving objects. ACM Transactions on Database Systems 31 (1), 255–298.
Pfoser, D., Jensen, C.S., Theodoridis, Y., 2000. Novel approaches in query processing for moving object trajectories. In: VLDB, pp. 395–406.
Pfoser, D., 2002. Indexing the trajectories of moving objects. Data Engineering Bulletin 25 (2), 4–10.
Rasetic, S., Sander, J., Elding, J., Nascimento, M.A., 2005. A trajectory splitting model for efficient spatio-temporal indexing. In; VLDB, pp. 934–945.
Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M., 2000. Indexing the positions of continuously moving objects. In: ACM-SIGMOD, pp. 331–342.
Theodoridis, Y., Nascimento, M.A., 2000. Generating spatiotemporal datasets. SIGMOD Record 29 (3), 39–43.
TIGER <http://www.census.gov/>.
Tzoumas, K., Yiu, M.L., Jensen, C.S., 2009. Workload-aware indexing of continuously moving objects. VLDB 2 (1–2), 1186–1197.
Wang, X., Zhang, Q., Sun, Weiwei, 2005. GTree: an efficient grid-based index for moving objects. Database Systems for Advanced Applications, 914–919.
Wolfson, O., Xu, B., Chamberlain, S., Jiang, L., 1998. Moving objects databases: issues and solutions. In: SSDBM, pp. 111–122 .
Yufei, T., Papadias, D., 2001. The MV3R-tree: a spatio-temporal access method for timestamp and interval queries. The VLDB Journal, 431–440.
Zhang, M., Chen, S., Jensen, C.S., Ooi, B.C., Zhang, Z., 2009. Effectively indexing uncertain moving objects for predictive queries. VLDB 2 (1–2), 1198–1209.

Youngjin Jung is a senior researcher at Supercomputing Center, Dept. of Cyber Environment Development, Korea Institute of Science and Technology Information since February 2010. He got his Ph.D. degree in computer science from the Chungbuk National University, South Korea, in 2007. He worked as a research scholar at the department of Spatial Information Science & Engineering, University of Maine from October 2007 to January 2010. His research interests include spatiotemporal database application, moving object index, sensor data processing, and environmental monitoring application.

Keun Ho Ryu received the Ph.D. degree from Yonsei University, Korea, in 1988. He is a professor at Chungbuk National University in Korea and a leader of database and bioinformatics laboratory. He served Korean Army as ROTC. He worked not only at University of Arizona as Post-doc and research scientist but also at Electronics and Telecommunications Research Institute, Korea. He has served on numerous program committees including a demonstration co-chair of the VLDB, a co-chair of the ADMA conference, the PC committee member of APWeb, the AINA, and so on. His research interests are included in temporal databases, spatiotemporal database, temporal GIS, ubiquitous computing and stream data processing, knowledgebase information retrieval, database security, data mining, and bioinformatics. He is a member of the IEEE as well as a member of the ACM since 1983.

Moon Sun Shin is an assistant professor at the Anyang University, South Korea. She got her Ph.D. degree in computer science from the Chungbuk National University, South Korea, in 2004. She worked as a lecturer professor at the department of computer science of Konkuk University from 2005 to 2008. Her research interests include spatiotemporal database, database security, data mining.

Dr. Silvia Nittel is an Associate Professor in the Department of Spatial Information Science and Engineering at the University of Maine. She received her Ph.D. in databases from the University of Zurich in 1994. Her current research focuses on database management techniques for sensor networks, especially sensor networks deployed in environmental and geographic applications. She is a co-founder the conference series on "Geosensor Networks".