Managing Frequent Updates in R-Trees for Update-Intensive Applications

著者	Song MoonBae, Kitagawa Hiroyuki
journal or publication title	IEEE transactions on knowledge and data engineering
volume	21
number	11
page range	1573-1589
year	2009-11
権利	(C) 2009 IEEE
URL	http://hdl.handle.net/2241/103773

doi: 10.1109/TKDE.2008.225

Managing Frequent Updates in R-Trees for Update-Intensive Applications

MoonBae Song and Hiroyuki Kitagawa, *Member*, *IEEE Computer Society*

Abstract—Managing frequent updates is greatly important in many update-intensive applications, such as location-aware services, sensor networks, and stream databases. In this paper, we present an R-tree-based index structure (called R^{sb}-tree, *R-tree with semibulk loading*) for efficiently managing frequent updates from massive moving objects. The concept of semibulk loading is exploiting a small in-memory buffer to defer, buffer, and group the incoming updates and bulk-insert these updates simultaneously. With a reasonable memory overhead (typically only 1 percent of the whole data set), the proposed approach far outperforms the previous works in terms of update and query performance as well in a realistic environment. In order to further increase buffer hit ratio for the proposed approach, a new page-replacement policy that exploits the level of buffered node is proposed. Furthermore, we introduce the concept of deferring threshold ratio (dtr) that simply enables deferring CPU- and I/O-intensive operations such as node splits and removals. Extensive experimental evaluation reveals that the proposed approach is far more efficient than previous approaches for managing frequent updates under various settings.

Index Terms—Indexing moving objects, R-trees, location-aware services, update-intensive applications, frequent updates.

1 Introduction

With the rapid advances in positioning systems—such as Global Positioning System (GPS) and Radio-Frequency Identification (RFID)—and mobile computing technologies, managing up-to-date information about the locations of massive moving objects has become a critical area of research [1]. Typical examples of these moving objects are delivery trucks, vehicles/pedestrians in a metropolitan area, goods in stock, things that are tagged with electronic devices like RFIDs, etc. More theoretically, this research area can be extended into a broad class of problems where a complex data object can be represented as a point in multidimensional (*feature*) space by utilizing a dimensionality reduction technique such as GEMINI [2]. These data include 1) measurement data sensed from tiny sensors distributed around a city, 2) complex time-series data in high-dimensional space, and 3) scientific data streams from a satellite and telescopic observations. In a broad sense, we call these problems as "update-intensive applications."

As an illustrative example, suppose that we are monitoring the locations of 1 million cell phone users in a city. Each user updates their location every 10 seconds, and a single *location server* keeps track of them. The location server continuously receives the location update stream as a sequence of location update records in a form of $\langle oid, p(x, y)\rangle$, where oid is an object identifier and p is its location which consists of x and y-coordinates. The location server needed to properly handle 100,000 updates per second. For each update, a spatial index maintained by the location server is needed to be updated for answering various spatial and location-dependent queries. Maximizing the update performance (update throughput) by minimizing the above-mentioned update cost for each update is, therefore, a critical issue for various update-intensive applications.

As a de facto standard index structure, R-tree [3] is originally designed for query-intensive setting, i.e., one of primary concerns in R-tree is to minimize the search cost of spatial queries. In R-tree, the update operation is very costly because it will be divided into two separated operations such as DELETE and INSERT sequentially. We believe certainly that the original R-tree is inappropriate for update-intensive setting in which frequent updates are continuously generated.

In order to tackle this problem, several techniques for managing frequent updates have been proposed [4], [5], [6]. These techniques are mainly motivated by the fact that, owing to a multipath search, DELETE is much more costly than INSERT. In order to avoid the multipath search for locating the old entry during DELETE operation, Kwon et al. [6] developed lazy updates performed in a bottom-up manner by adopting a secondary index on the R-tree. Lee et al. [5] improved this idea by developing an in-memory summary structure to help both updates and queries. More recently, Xiong and Aref proposed a memo-based update in which deletions are deferred and performed in a batch manner [4]. For this purpose, the authors proposed a memory data structure called Update Memo that stores recent updates in R-tree. In these techniques, at least two disk I/Os are needed to perform immediately for updating R-trees accordingly against each incoming update.

- *M. Song is with the Intelligent HCI Convergence Research Center, School of Information and Communication Engineering, Sungkyunkwan University, Jangan-Gu, Cheoncheon-Dong 300, Suwon-Si, 440-746 South Korea. E-mail: mbsong@gmail.com.*
- *H. Kitagawa is with the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tennohdai, Tsukuba, Ibaraki 305-8573, Japan. E-mail: kitagawa@cs.tsukuba.ac.jp.*

Manuscript received 25 Sept. 2007; revised 22 Apr. 2008; accepted 23 Oct. 2008; published online 4 Nov. 2008.
Recommended for acceptance by V. Ganti.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2007-09-0474.
Digital Object Identifier 10.1109/TKDE.2008.225.

In this paper, we propose a new update strategy that is trying to defer and group the insertions as well as the deletions. The main idea is to adopt a small in-memory buffer to buffer, defer, and group the incoming updates, then inserts them into disk at once (called *semibulk loading*). In order to maximize the update performance, it is essential to 1) defer the disk I/Os for insertions by exploiting in-memory buffer, 2) group-insert the buffered updates (if needed) and reduce the number of disk I/Os as much as possible, and 3) filter out the repeated accesses from the requested I/Os by making the best use of a page buffer. Our major contributions can be summarized as follows:

- We propose an R-tree variant index (called $\mathrm{R^{sb}}$-tree) that utilizes an in-memory buffer structure of a reasonable size in order to minimize the update cost and query cost as well.
- We introduce the concept of deferring threshold ratio (dtr) that simply enables deferring CPU- and I/O-intensive operations such as node splits and removals by exploiting the in-memory buffer and inserting meaningless entries into the underflowed leaf node, respectively.
- In order to improve buffer hit ratio, a simple variation of *least recently used* (LRU) replacement policy (called Level-Aware LRU) that is aware of level of nodes stored is proposed.
- Analytical study for the update and storage costs and an extensive set of experiments are conducted. The results show that the proposed approach can greatly reduce the overall update cost by about three to five times compared with the previous works with a reasonable memory overhead (1 percent of the whole data set).

The rest of the paper is organized as follows: Section 2 discusses the related work on supporting frequent updates for R-trees. In Section 3, we present our approach and its detailed algorithms. In Section 4, we discuss advanced techniques for further improving the proposed approach. Section 5 reports the analytical results. Section 6 presents the results of a performance evaluation of the proposed $\mathrm{R^{sb}}$-tree and previous studies. Finally, we conclude in Section 7 with directions for future work.

2 Related Work and Preliminaries

2.1 Indexing Moving Objects with R-Trees

As one of the most promising indexes for searching spatial data, R-tree was proposed by Guttman [3]. It is a d-dimensional extension of $\mathrm{B^{+}}$-tree for multidimensional objects. Any geometric object is represented by its minimum bounding rectangle (MBR). An MBR is minimal approximation of a geometric object and a d-dimensional (hyper-)rectangle R in the data space. Every node has at least m and at most M entries ($m \leq M/2$) unless it is the root node. Leaf nodes in R-tree contain leaf entries of the form (oid, mbr), where oid is a unique object identifier in the database and mbr is the MBR of the spatially indexed object. Nonleaf nodes (also called index nodes; we use the terms interchangeably) contain index entries of the form (ptr, mbr), where ptr is a pointer to a child node and mbr is the MBR that spatially covers all MBRs in its child nodes. Fig. 1 shows an R-tree for point set P in which points that are spatially close in space are clustered in the same leaf node. Nodes are then recursively grouped together with the same principle until the top level, which consists of a single node (called $Root$).

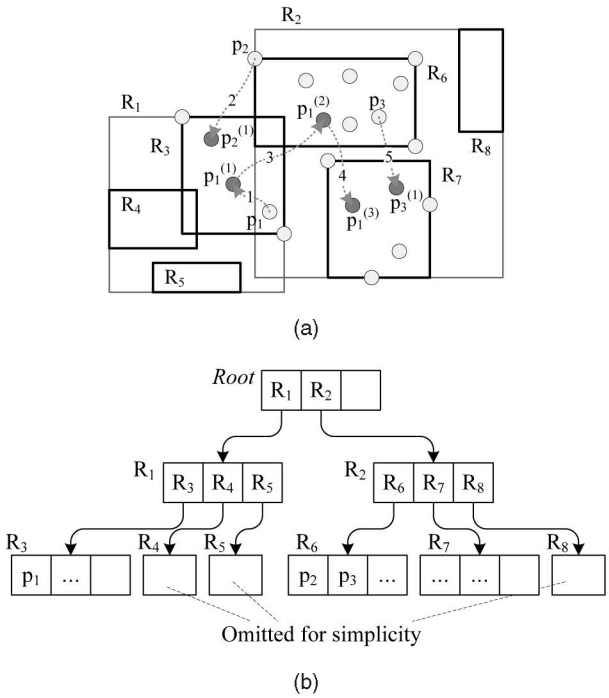

Fig. 1. An example of R-tree and five updates in which $M = 8, m = 3$. (a) The planar representation for an R-tree. (b) Directory structure of R-tree.

In the last few years, many index structures have been proposed in the area of moving object indexing. These data structures are classified into two categories based on the type of data to be stored. One category is for the trajectories of the moving objects [7], and the other category is for the current and anticipated location of moving objects [4], [5], [6], [8], [9], [10], [11]. The latter can also be classified into two subcategories such as *indexing current location* [4], [5], [6], and *indexing based on movement vector* [8], [9], [10], [11]. A good survey for the issue of moving objects indexing is [12].

Thanks to its simplicity, the former will be more generally applicable. Furthermore, due to its close relationship with our proposal, we will discuss these techniques in detail later in Section 2.3.

The underlying principle of the latter is the linear movement of moving objects; i.e., the object's location at any future time t can be obtained as $p_{ref} + \bar{v} \cdot (t - t_{ref})$ where p_{ref} is a reference point at some reference time t_{ref} ($t > t_{ref}$) and $\bar{v}$ is a movement vector. Based on this concept, TPR-tree indexes time-parameterized MBRs by using the insertion/deletion algorithms of the $\mathrm{R^{*}}$-tree [13]. Tao et al. [10] improved the idea of TPR-tree by employing a different set of insertion and deletion algorithms in order to minimize the query cost. Similar to these techniques, Prabhakar et al. [9] proposed so-called velocity-constrained indexing (VCI) for efficient processing of continuous range queries. In VCI, each node has an additional field V_{max} to describe maximum speed among all objects in the subtree.

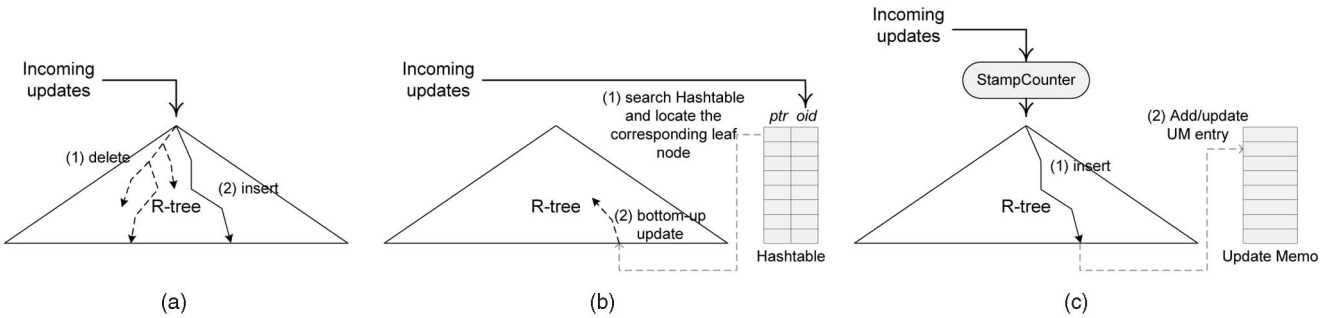

Fig. 2. Three possible ways of updating R-trees: (a) traditional top-down update [3], [13], (b) bottom-up update [5], [6], and (c) memo-based update [4].

2.2 Reducing I/O Cost with Buffer: Update Buffer versus Page Buffer

Another important aspect in designing an index structure is employing a buffer to increase overall I/O performance. Generally, there are two kinds of buffer such as update buffer and page buffer. In this section, we discuss the several aspects of these two schemes.

2.2.1 Page Buffering

Page buffering which is a standard feature of all DBMSs is the most common way to reduce the number of I/Os. The most popular algorithm for buffer replacement is LRU algorithm that replaces the page that was least recently accessed or used. There are several improvements over the LRU technique [14], [15], [16].

Assuming B-tree index, Sacco proposed two variations of LRU: ILRU (Inverse LRU) and OLRU (Optimal LRU) [15]. ILRU guarantees that a page will reside in the buffer until all of its children are removed from the buffer. Assigning different buffers to each level of the index tree, OLRU guarantees that top-level pages of a tree have higher priority compared to the rest. Leutenegger and Lopez [16] discuss the effect of pinning higher levels of R-tree index in the buffer. This pinning technique constantly keeps the higher pages in buffer, so that the buffer does not change. Due to this reason, lower page accesses are directly translated into disk I/Os. Brinkhoff proposed so-called *spatial page-replacement algorithms* optimized for spatial databases [14]. The replacement algorithm replaces a page based on spatial properties such as the area, the overlap and the margin of pages, and page entries. Moreover, the combination of spatial page-replacement strategies with LRU strategies was also proposed.

In this work, similar to OLRU, we propose Level-Aware LRU policy which guarantees that top-level pages have higher priority than the rest. Without pinning any nodes, it achieves the same effect as OLRU in a simpler way by merging all buffers into a single *level-aware* buffer.

2.2.2 Update Buffering

As discussed in [17], basically there are two ways for buffering updates; one is buffering updates within tree nodes [18], [19], and the other is buffering in separate structures [11], [20], [21].

Arge et al. [18] proposed the Buffer R-tree (BR-tree) for performing bulk updates and queries. In BR-tree, every node at $\lceil \log_f(\frac{R}{4M}) \rceil$th level of the tree has a buffer, where R is the number of rectangles that fit in the main memory. The attached buffer enables the operations such as insertion, deletion, and query to be performed in a lazy manner. In [19], Lin and Su proposed a lazy group update (LGU) algorithm that is extended from the BR-tree. LGU technique exploits disk-based buffers (I-Buffer) associated with each index node and a global memory-based buffer (D-Table) to perform group deletions in a bottom-up manner.

In [11], Cui et al. proposed a grid-based buffering scheme for TPR-tree, where active and inactive objects are separately managed by main-memory grid buffer and disk-based TPR-tree respectively. This will increase the query performance by reducing the overlap between MBRs in TPR-tree which is mainly caused by active moving objects. As an alternative to R-tree variants, Xiong et al. proposed a grid-file-based index called LUGrid which supports both lazy-insertion and lazy-deletion [20]. The former is supported by adopting memory grid (MG) for buffering unprocessed insertions, and the latter is supported by "miss-deletion memo" (MDM) for keeping track of deletions. Due to adopting grid-file-based structure, in contrast to our technique, LUGrid can only index point objects, which is a significant limitation for many practical applications. More recently, Biveinis et al. [21] proposed R^{R}-tree which exploits in-memory operation buffer in the form of another R-tree and supports bulk-insertion algorithms. Similar to other R-tree variants, unlike our proposal, this technique handles an update as a delete-insert pair; this means that buffer utilization will be decreased by half since the buffer is shared by both insert and delete entries.

For achieving similar goals, the proposed approach has a different buffer structure based on in-memory hashtable and histogram. Moreover, it provides an advanced mechanism for deferring CPU and I/O-intensive operations.

2.3 Different Update Strategies in R-trees and Our Motivation

In this section, we describe the three most promising strategies for updating R-trees in the category of "indexing current locations." In Fig. 1a, there are five updates needed to be handled serially. $p_i^{(j)}$ is the location of object o_i starting from p_i after j movements. During a given time interval, o_1 updates its location more frequently than others. Based on this example, we describe the detailed operations of updating R-trees (see Fig. 2).

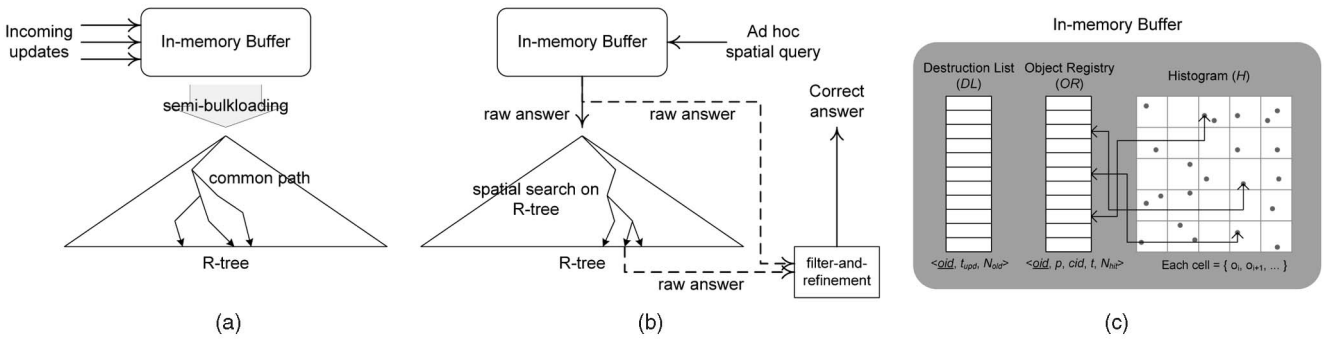

Fig. 3. The concept of semibulk loading. (a) The concept of semibulk loading. (b) Query processing on R^{sb}-tree. (c) The detailed view of in-memory buffer.

First, we introduce the top-down update strategy described in the original R-tree [3]. As shown in Fig. 2a, it performs in two steps: 1) delete an old entry by performing a multipath search and reorganize the tree along the search path, and 2) insert a new entry to the appropriate leaf node and then reorganize the tree along the insertion path again. Consider the first movement of o_1 from p_1 to $p_1^{(1)}$ depicted in Fig. 1a. The deletion of p_1 from R_3 incurs an underflow, and this also incurs consequential underflows at R_1 and $Root$ (so-called *cascading deletion*). As a result, R_2 becomes a new root node and the remaining entries will be reinserted in the tree. After the whole deletion process, the entry $p_1^{(1)}$ is inserted. If an overflow occurs, the nodes have to be splitted and therefore the tree has to be updated along the insertion path. Thus, we believe that the top-down update requires a very complicated process and it is not able to avoid a huge number of disk I/Os.

As a preliminary proposal for this problem, Kwon et al. proposed a bottom-up update strategy called Lazy update R-tree [6] that exploits a secondary index called *Hashtable* whose entry is a form of (oid, ptr), where ptr is a pointer to its leaf node. Reconsider the example in Fig. 1a, the movement from p_1 to $p_1^{(1)}$. First, via Hashtable, we can directly find the leaf node R_3 for p_1. After the update of $R_3.mbr$, the bottom-up update is terminated because $p_1^{(1)} \subseteq R_3.mbr$. In case of the third movement from $p_1^{(1)}$ to $p_1^{(2)}$, however, $p_1^{(2)}$ should be inserted by a top-down insertion because $p_1^{(2)} \not\subseteq R_3.mbr$. Compared to the top-down update, this 1) simplifies the deletion of an old entry that requires a multipath search and 2) provides the *localized* insertion of a new entry. Due to the frequent top-down insertions, this approach may suffer performance degradation when the average moving distance is increased. To overcome this problem, a generalized bottom-up update strategy was presented in [5]. The authors provide an in-memory summary structure to help both updates and queries.

In [4], Xiong and Aref proposed an R-tree variant called *R-tree with update memo* (for short RUM-tree) that provides a unique update operation called *memo-based update*. This approach utilizes a top-down insert as is for the insertion of a new entry, and the deletions of old entries are deferred and done in a batch manner. As shown in Fig. 2c, each newly incoming update is tgcqed by $StampCounter$, and then inserted into the tree. Then, the corresponding update memo (UM) entry is needed to be updated to indicate this change. The old entries get deleted later on by referring to UM. Recall the example in Fig. 1a. In the example, thus, only new updates will be inserted and the old entries will be removed afterwards and as a result, underflow at R_3 does not occur. However, the splits are more likely to occur frequently due to the increased number of entries. For instance, R_6 will be splitted because of the undeleted old entry p_2.

Motivational remarks. One of the most promising approaches that resolve I/O-intensive problems discussed in the above example is to adopt a special update buffer. Then, three movements of p_1 in Fig. 1a can be reduced into a single movement from p_1 to $p_1^{(3)}$ (third and fourth updates are performed in the memory buffer). We expect that the factor of this improvement will increase especially for fast moving objects. Moreover, inserting $p_1^{(3)}$ and $p_3^{(1)}$ to R_7 can be done simultaneously. We can also prevent the underflow at R_3 by deleting p_1 and inserting $p_2^{(1)}$ at a time. In this paper, we explore this idea of update buffering, and the details of our approach are given in the next section.

3 The R^{sb}-Tree Index

3.1 The Basic Concept and Structures

The basic idea underlying our approach is to put the spatially clustered updates into a group in which most part of their insertion paths must be shared. Thus, the total update cost will be drastically decreased. Conceptually speaking, our objective is to minimize the cost of overall update operations by increasing the number of updates processed by a single I/O operation. To this aim, we adopt a small in-memory buffer to defer/group the incoming updates as much as possible and to minimize the update cost by inserting them in a batch.

Fig. 3 shows the basic concept of the proposed structure. When the buffer is full, the buffered updates are inserted into the disk-resident R-tree simultaneously by exploiting the common path (see Fig. 3a). For query processing, combining two raw answers from the in-memory buffer and R-tree, and refining the result are needed (see Fig. 3b). As shown in Fig. 3c, R^{sb}-tree has the following components:

- *GlobalClock* is a monotonic increasing time stamp generator that increments whenever inserts/deletes on the in-memory buffer take place. This is the unique source of every entry's time stamp in R^{sb}-tree. Using this time stamp, a *fresh entry* (the most recent entry of an object) can easily be distinguished from all other old entries (called *obsolete entries*).
- **Object Registry (*OR*)** is a set of object-tuple $\langle oid, p, cid, tstamp, N_{hit}\rangle$[1] hashed based on object-id oid, where p is location of object oid, cid is an identifier of a histogram cell, $tstamp$ is a time stamp assigned by *GlobalClock*, and N_{hit} is the number of updates performed within OR (called *OR hit*). The capacity of OR ($ORSize$) is defined as a fraction of the total number of moving objects, N.
- **Histogram (*H*)** is a two-dimensional hash structure that disjointly divides the data space $DS = [0,1)^2$ into $g \times g$ grid cells, where g is called the *histogram granularity*. Each cell is denoted by $H[i,j], 0 \leq i,j \leq g-1$ and covers a region of space $[i\delta,(i+1)\delta) \times [j\delta,(j+1)\delta)$ generated by uniform partitioning, where δ is the length of cell size ($\frac{1}{g}$). Each cell contains the histogram value $hist$ and the object-id list (OL) of OR entries whose locations are completely enclosed by the cell ($|OL| = hist$). For any two-dimensional point $p = (p_1, p_2)$, the corresponding cell of point p can simply be computed as $H[\lfloor\frac{p_1}{\delta}\rfloor, \lfloor\frac{p_2}{\delta}\rfloor]$.
- **Destruction List (*DL*)** is a set of supplemental information in order to minimize the deletion cost of obsolete entries. The underlying principles are the same conceptually as the concept of update memo that we borrowed from [4]. Each entry in DL (denoted as $DL[oid]$) is a form of $\langle oid, t_{upd}, N_{old}\rangle$ hashed on object-id oid and denotes the number of obsolete entries which should be removed from the disk-resident R-tree. Time stamp t_{upd} is the time stamp of the fresh entry of oid, and N_{old} is the number of obsolete entries for the object oid in the disk-resident R-tree.

The disk-resident part of R^{sb}-tree is same as the traditional R-tree except that 1) all leaf entries are time stamped by *GlobalClock* and 2) all index entries are stored in their inserted order (see more details in Section 3.4). Thus, a leaf entry is the form of $\langle oid, mbr, t\rangle$, where oid and mbr are identical to those in the traditional R-tree, and t is a globally unique time stamp generated by *GlobalClock*. Without removing obsolete entries, we are able to distinguish the fresh entry from them by using the time stamp t. This will greatly reduce the deletion cost for old entries. Owing to the redundant obsolete entries, the total number of leaf entries stored in the tree will surely increase, and accordingly the search cost will increase. The introduction of time stamp will result in the reduced node capacity. In other words, the capacity of an index node (M_x) is larger than that of a leaf node (M_0). The R^{sb}-tree's height (h) is $h = 1 + \lceil \log_{f_x} \frac{N}{f_0} \rceil$, where N is the number of moving objects, and f_0 and f_x are the average fan outs of a leaf node and an index node, respectively (e.g., $f_0 = M_0 * 67\%$). Because the tree's height highly depends on f_x, the tree's height h is almost same as R*-tree. In this paper, the root is assumed to be at level $h-1$ and leaf nodes are assumed to be at level 0. For reference convenience, the notations used in the paper are listed in Table 1.

TABLE 1
Notations and Its Meaning

Notation	Meaning
d	Number of dimensions (dimensionality)
N	The number of moving objects in database
M_0, M_x	The capacity leaf and index node
m_0, m_x	The min. number of entries per leaf and index node
f_0, f_x	Average number of entries per leaf and index node
h	Height of the R-tree
n_i	The number of R-tree nodes at level $i \in [0, h-1]$
$OR[oid], DL[oid]$	The OR and DL entry of object oid

3.2 Insert, Update, and Delete on In-Memory Buffer

Algorithm 1 shows the INSERT, UPDATE, and DELETE algorithms of the R^{sb}-tree which are operated on the in-memory buffer completely. In the R^{sb}-tree, update is identical to an insert operation.[2] For each incoming update $\langle oid, p\rangle$, *GlobalClock* first increments and assigns a time stamp to the update (line 1). If the corresponding OR entry $OR[oid] \in OR$, it will be updated within OR, and its update count N_{hit} will be incremented (lines 3-4). If not, a new OR entry is inserted into OR (line 7). When there is no enough space in OR, we perform what we call *Flush* in order to make room for the newly incoming update (line 6). For the proper removal of obsolete entries, which are stored before, the corresponding DL entry $DL[oid]$ is registered or updated (lines 8-10).

Unlike the traditional R-tree, only an object-id oid is needed for DELETE procedure. The reason behind this is that we can find the obsolete entries not by their spatial locations, but instead by their oids and time stamps. This characteristic will make an application simpler by removing maintenance cost for the location information required to delete the obsolete entries. As in INSERT, *GlobalClock* increments first (line 1). If $OR[oid]$ exists, it is possible to remove the obsolete entry within OR without any disk accesses (lines 2-3). If not (this means the obsolete entry is already stored into disk), then the corresponding DL entry $DL[oid]$ must be registered or updated (lines 4-6).[3]

Algorithm 1. INSERT, UPDATE, and DELETE algorithms
INSERT (oid, p) // UPDATE(oid, p) is exactly the same.
Input: oid is object-id and p is location (p_1, p_2).
1: $t \leftarrow$ *GlobalClock*; Increment *GlobalClock*;
2: **if** $(oid \in OR)$ **then** // Called *OR hit*
3: Update $OR[oid]$ with $\langle p, H[\lfloor\frac{p_1}{\delta}\rfloor, \lfloor\frac{p_2}{\delta}\rfloor], t\rangle$;

1. For the purpose of crash recovery, every change in OR is logged into a stable storage. In our future work, we will investigate the (log-based) recovery issue in detail.

2. Due to this reason, meaningless DL entries (called *phantom entry*) can be created. We will discuss a solution for this issue in Section 3.4.

3. If the object oid have not been inserted before, this also generate an extra DL entry, and this can also be treated as a phantom entry discussed in INSERT.

4: Increment $OR[oid].N_{hit}$;
5: **else**
6: **if** (OR is full) Invoke FLUSH();
7: Insert OR entry $\langle oid, p, H[\lfloor \frac{p_1}{\delta} \rfloor, \lfloor \frac{p_2}{\delta} \rfloor], t \rangle$ into OR;
8: **if** ($oid \in DL$)
9: **then** Increment $DL[oid].N_{old}$ and $DL[oid].t_{upd} \leftarrow t$;
10: **else** Insert new DL entry $\langle oid, t, 1 \rangle$;

DELETE(oid)
Input: oid is object-id.
1: $t \leftarrow GlobalClock$; Increment $GlobalClock$;
2: **if** ($oid \in OR$) **then**
3: Remove the OR entry $OR[oid]$;
4: **else if** ($oid \in DL$)
5: **then** Increment $DL[oid].N_{old}$ and $DL[oid].t_{upd} \leftarrow t$;
6: **else** Insert new DL entry $\langle oid, t, 1 \rangle$;

3.3 Semibulk Loading

Semibulk loading is the process of storing the location information on the in-memory buffer to the disk-resident R-tree. It is far different from the conventional bulk loading which is performed in an offline manner because the ultimate goal is not to maximize the quality of the R-tree, but to optimize the update and search costs together. In order to improve the I/O efficiency of semibulk loading, we choose a proper subset of OR, which have a strong probability to be inserted to the same leaf node. Based on the way of choosing the subset (denoted as OL_{flush}), we propose four possible policies as follows:

- FlushAll: The entire OR will be chosen as OL_{flush}. This incurs a low I/O efficiency, because the average number of OR entries to be inserted to a leaf node is decreased to $\frac{ORSize \cdot f_0}{N}$. Moreover, this will decrease OR hit ratio also by emptying the entire OR at once.
- FlushCell: In order to maximize I/O efficiency, we choose a cell that has the maximum histogram among $g \times g$ cells. The performance benefit increases as the skewness of incoming updates increases. Also, this does not decrease OR hit ratio rapidly because the entries still exist in OR even after the flush. The size of the chosen OL_{flush} is far larger than the average number of OR entries in a Histogram cell ($|OL_{flush}| \geq \frac{ORSize}{g^2}$), and the performance benefit of this policy is affected by the Histogram granularity (g).
- FlushLRUCell: This policy chooses the cell that has the maximum histogram except for the top Δ percent entries whose time stamps ($tstamp$) are bigger than others. In short, this approach filters out the recently updated entries from OL_{flush} to be inserted to disk.
- FlushLFUCell: Similar to FlushLRUCell policy, this policy chooses the cell with the maximum histogram except for the top Δ percent entries whose update counts N_{hit} are bigger than others. In short, this approach filters out the most frequently updated entries from OL_{flush} to be inserted to disk.

The FlushAll policy can be effective in the initial state where the data does not exist (or only the $Root$ node exists). In this case, it is better for I/O efficiency to insert the entire OR by a conventional bulk-loading algorithm such as Sort-Tile-Recursive technique [22]. We call this phase *BootRoot process* where the whole framework of index structure is generated. This will not only improve the speed of insertion but also maximize the index quality. At this phase, the objective is to maximize the storage utilization and to produce as many leaf nodes as possible. As a result, $\lceil \frac{ORSize}{C_{Boot}} \rceil$ leaf nodes with $C_{Boot} = \min(\max(\frac{ORSize}{M_x}, m_0), M_0)$ entries will be generated. If the generated leaf nodes cannot be accommodated by a single index node called root (i.e., $\lceil \frac{ORSize}{C_{Boot}} \rceil > M_x$), we recursively apply STR for generating the resulting R-tree with height $h = 1 + \lceil \log_{f_x} \lceil \frac{ORSize}{C_{Boot}} \rceil \rceil > 2$.

FlushLFUCell and FlushLRUCell policies will be efficient especially when the update probability of each moving object is very different. The most typical case is the speed difference (recall the example in Fig. 1a). Consider a fast moving object that updates its location very frequently. If we flush the object continuously, this object will register its new update into OR again and the rest of flushed entries will become obsolete entries quickly. This will increase the number of obsolete entries in the tree. Obviously, this type of frequently updated entries have to be filtered out from the chosen OL_{flush}. To this end, FlushLFUCell and FlushLRUCell utilize the number of OR hits N_{hit} which is explicitly accumulated and the recent update time $tstamp$ of each OR entry, respectively.

Algorithm 2. FLUSH algorithm
FLUSH(OL_{flush})
Input: OL_{flush} is an object-id list chosen from OR.
1: Initialize OL based on OL_{flush}.
2: A set of path stacks $S \leftarrow$ CHOOSEOPTIMALPATHS (OL);
3: **foreach** path stack $s \in S$ **do**
4: Leaf node $L \leftarrow s.pop()$;
5: CLEAN(L); // clean-upon-touch process
6: Insert $\forall$ entry $e \in OL$ such that $e.stack = s$ into L;
7: **if** (overflow occurs) Invoke OVERFLOWTREATMENT(L);
8: **if** (underflow occurs) Invoke UNDERFLOWTREATMENT(L);
9: **repeat**
10: Remove all duplicate stacks in S;
11: **foreach** path stack $s \in S$ **do**
12: Index node $N \leftarrow s.pop()$; Update N;
13: **until** (S is not empty)

CHOOSEOPTIMALPATHS(OL)
Input: OL is a set of entries $\langle oid, sid, optCh \rangle$ where oid is object-id, sid is a stack index for stack pool S, and $optCh$ is the node id of next optimal child.
Output: Stack pool S is a set of stacks which present optimal paths to the optimal leaf nodes.
1: Initialize stack s_0: s_0.push($Root$) and stack pool $S \leftarrow \{s_0\}$;
2: **foreach** entry $e \in OL$ **do**
3: $e.sid \leftarrow S$.indexOf(s_0); $e.optCh \leftarrow Root$;
4: **for** $i = 2$ **to** h **do**

5: **foreach** entry $e \in OL$ **do**
6: $e.optCh \leftarrow$ FINDLEASTENLARGEMENT $(e, e.optCh)$;
7: **if** $(|S[e.sid]| < i)$ $S[e.sid]$. push$(e.optCh)$, **continue**;
8: **if** $(S[e.sid].top = e.optCh)$ **continue**;
9: Let s' be a stack $\in S$ s.t. $s'.top = e.optCh$;
10: **if** $(\nexists s')$ **then**
11: Allocate new stack s'; Copy the content of $S[e.sid]$ to s'.
12: $s'.top \leftarrow e.optCh; S \leftarrow S \cup \{s'\}$;
13: $e.sid \leftarrow S$.indexOf(s');

After selecting the subset OL_{flush}, inserting the OR entries in OL_{flush} is definitely needed through the exploitation of common insertion path as much as possible (see Algorithm 2). First of all, the minimum number of path stacks for the insertion procedure should be generated by invoking CHOOSEOPTIMALPATHS. We then remove the obsolete entries in the leaf node L at the top of each path stack and insert OR entries into L (line 3-6). If necessary, we call OVERFLOWTREATMENT(L) or UNDERFLOWTREATMENT(L) which are discussed later in Section 4.1 (Algorithm 4) (line 7-8). Finally, we reorganize the tree along the common path (line 9-13).

3.4 Freshening with Garbage Cleaner

The FLUSH operation discussed in the previous section is essentially bulk insertion, and eliminates the obsolete entries only in the leaf nodes where insertions are performed[4]; therefore, it will not be able to eliminate the obsolete entries which exist in unvisited leaf nodes. Consequently, a special process (called *Garbage Cleaner*; for short GC), which goes round the whole tree and eliminates obsolete entries is positively necessary. For the better quantitative discussion, we first define the concept of $freshness$ as a measurement of R^{sb}-tree's quality.

Definition 1 (Freshness). *Freshness $Fr(e)$ of a leaf entry e is defined as follows:*

$$Fr(e) = \begin{cases} 1, & \text{if } e.oid \notin DL \text{ or } DL[e.oid].t_{upd} \leq e.t, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Similarly, $freshness Fr(L)$ of a leaf node $L = \{e_1, e_2, \ldots, e_f\}$ is $Fr(L) = \frac{1}{f}\sum_{i=1}^{f} Fr(e_i)$. Freshness $Fr(T)$ of the disk-resident R-tree T is defined as the number of fresh entries stored in disk over the number of entire leaf entries, i.e.,

$$Fr(T) = \frac{1}{n_0}\sum_{L \in T} Fr(L) = \frac{N_{fresh}}{N_{fresh}+N_{obsolete}}, \quad (2)$$

where N_{fresh} and $N_{obsolete}$ are the total numbers of fresh and obsolete entries in the disk-resident part of R^{sb}*-tree, respectively.*

GC is a lazy group deletion operated once every K updates (K is called $GCInterval$) (see Algorithm 3). GC finds the next leaf node L (line 2) and removes the obsolete entries in L (line 3). In order to identify which leaf entry is an obsolete entry, we use the $freshness$ concept defined in Definition 1. In case of underflow, UNDERFLOWTREATMENT is invoked to deal with this situation (line 4). The total update cost will surely increase as K decreases; when K decreases, the search cost also decreases but the total update cost increases. Naturally, there is a trade-off between the update and search costs. These are not separated problems but closely related problems, since another search for finding the optimal location of a new entry and locating the corresponding leaf node of an old entry is needed to perform an update.

Algorithm 3. DFS Scanner for Garbage Cleaning

GARBAGE-CLEANER()
1: // Simple DFS traversal using $dfsStack$;
2: **foreach** leaf node L periodically **do** once every K updates
3: CLEAN(L);
4: **if** (underflow occurs) Invoke UNDERFLOWTREATMENT();

CLEAN(LeafNode L)
1: **foreach** leaf entry e in L **do**
2: **if** $(freshness\ Fr(e) < 1)$ **then**
3: Delete e from L and decrement $DL[e.oid].count$;
4: **if** $(DL[e.oid].count = 0)$ Remove $DL[e.oid]$ from DL;

Unlike RUM-tree, we simply make use of a stack-based DFS traversal. In RUM-tree, every node has a pointer to its parent node and all the leaf nodes are doubly linked in a cycle. Using these auxiliary pointers, the GC of RUM-tree visits leaf nodes in turn and adjusts the tree in a bottom-up manner. As stated in [5], having its parent address has a high maintenance cost. In this way, splitting an index node at level l incurs updating all parent-link fields of its child nodes at level $l-1$, i.e., on an average $(M+1)/2$ nodes are updated. When l is close to 0 (the leaf level), this will cause performance degradation due to frequent page buffer miss. Moreover, the links between leaf nodes also have to be updated during node splits and removals.

To avoid this high maintenance cost, GC in this study is based on DFS traversal that utilizes a special stack (called $dfsStack$). The $dfsStack$ stores all node-ids (at most h) along the path that leads from *Root* to the leaf where GC is performed. The node id of ith level is stored in $dfsStack[h-i]$ and the id of leaf node L where GC is being performed exists on the top of $dfsStack$ ($dfsStack[h]$). If we want to visit the next leaf node, pop out Ls id from $dfsStack$ and find the next leaf node by using the remaining $dfsStack$'s $h-1$ nonleaf nodes. If leaf node L is the last entry of level 2 nonleaf node (with probability of $\frac{1}{f_x}$), it must visit the upper level. Consequently, GC overhead for visiting the next leaf node will be $\sum_{i=0}^{h-2} f_x^{-i}$. However, this cost is sufficiently low as shown below.

3.4.1 Additional Cost of $dfsStack$-Based GC

Given an R-tree of height h, the total number of nodes in the tree is as follows:

$$\sum_{i=0}^{h-1} n_i \approx \sum_{i=0}^{h-1} \frac{N}{f_x^i \cdot f_0} < \sum_{i=0}^{\infty} \frac{N}{f_x^i \cdot f_0} = \frac{N \cdot f_x}{(f_x - 1) f_0}, \quad (3)$$

4. This is called *clean-upon-touch* process, and it does not incur any additional disk I/Os.

where n_i is the number of node at level i. Hence, the total number of nodes increases $\frac{f_x}{f_0}$ times rather than the case of $f_x = f_0$ (the original R-tree). For instance, when we use 4 kB page ($M_x = 204$, $M_0 = 145$), the growth of tree is about $\frac{204}{145} \approx 1.41$. The ratio of the number of nonleaf nodes to that of leaf nodes is

$$\frac{\#\text{ of nonleaves}}{\#\text{ of leaves}} < \frac{\frac{N \cdot f_x}{(f_x-1)f_0} - \frac{N}{f_0}}{\frac{N}{f_0}} = \frac{1}{f_x - 1}. \quad (4)$$

The most important point of (4) is that it becomes a very small value when f_x increases. For 4 kB page, it is smaller than 0.5 percent. Another point is that most of nonleaf nodes will be in the page buffer. The result of our initial experiment shows that the average buffer hit ratio for visiting additional nonleaf nodes is over 99.5 percent. Accordingly, it is clear that our $dfsStack$-based GC has much more smaller maintenance cost than that of RUM-tree.

3.4.2 Phantom Entry Problem in DL

Since there is no differentiation between insert and update, the insert operation will generate a new DL entry $\langle oid, t, 1\rangle$, even though there is no such obsolete entry. Such a DL entry (called *phantom entry*) will never get removed from DL because its N_{old} will never be zero [4]. For this problem, similar to RUM-tree, we employ a *phantom inspection* process that periodically detects and eliminates phantom entries. First, assign the time stamp of $GlobalClock$ to τ. After this, GC visits all the leaf nodes and garbage-cleans those nodes. When it has visited all the nodes, at $\acute{\tau}$, DL entries whose t_{upd} are smaller than τ must be phantom entries. The inspection cycle of this scheme is surely $K \cdot n_0$ updates.

We can shorten the inspection cycle by generalizing the time stamp τ. Let τ_i be the time stamp of $GlobalClock$ when GC visits the ith leaf node, $i \in [0, n_0 - 1]$, where n_0 is the number of leaf nodes. We maintain an array of r time stamps $T, T[j] = \tau_{j\lfloor \frac{n_0}{r} \rfloor}, j \in [0, r-1]$. For example, $r = 3$ and $n_0 = 25$, then $T[0..2] = \{\tau_0, \tau_8, \tau_{16}\}$. The inspection will be performed in the following steps with r times shorter cycle: 1) whenever GC visits the $j\lfloor \frac{n_0}{r} \rfloor$th leaf node ($j \in [0, r-1]$), 2) remove all phantom entries in DL whose t_{upd} are smaller than τ_j, and 3) update τ_j by $GlobalClock$. Moreover, interfering with the decision of phantom entry, the insertion/deletion of leaf nodes can be handled properly by the following way. Increment every $\tau_i \in T$ by K if a leaf node is inserted. The deletion of a leaf node only incurs *false negative*; i.e., missing some of phantom entries is possible. They can be dealt with at the next cycle, so we simply ignore it. Whenever GC visits the 0th leaf node, every $\tau_i \in T$ is recomputed by using the updated n_0.

3.4.3 Managing $dfsStack$ and Relative Order of Index Entries

Unlike the B-tree family, the R-tree has no definition of relative order between index entries in a nonleaf node. By defining the relative order of index entries, we can certainly make GC to visit all the leaf nodes thoroughly. The relative order between index entries is simply defined as their inserted order; that is, if an index entry e_1 is inserted before an entry e_2, then e_1 precedes e_2 (formally, $e_1 \prec e_2$). For this purpose, the following conditions should be satisfied: 1) When a new entry is being inserted to an index node, it must be placed at the end of the node. 2) When an entry is being deleted, the relative order of remaining entries must be kept by shifting the succeeding entries consecutively. 3) For node splitting, we first assume that a nonleaf node $N \in dfsStack$ to be splitted into N_1 and N_2 (where $N_1 \prec N_2$). In this case, N_1 should contain the index entry that points to the child node along $dfsStack$. Otherwise, we make an exchange between N_1 and N_2.

Furthermore, as the R-tree dynamically changes, $dfsStack$ has to be updated accordingly. If a node $N \in dfsStack$ is deleted, every node along the path from N to leaf node L at the top of $dfsStack$ should be popped from $dfsStack$. Thus, it is possible that the node (called NN) residing on top of $dfsStack$ can be a nonleaf node. If so, we must fill $dfsStack$ with every node along the path from NN to the most preceding leaf node of the subtree rooted at node NN.

3.5 Query Processing on $\mathrm{R^{sb}}$-Tree

The general flow of query processing on $\mathrm{R^{sb}}$-tree is depicted in Fig. 3b. Now, we discuss range query processing on query window qw. First of all, the algorithm searches all OR entries that intersect with qw by using Histogram as a spatial index. Then, the ordinary search operation over the R-tree is conducted. However, the raw answer from the R-tree may contain obsolete entries. We can obtain the correct answer by combining these two raw answers and filtering the obsolete entries out by checking the freshness of their elements.

Searching k-nearest neighbor (kNN) is conducted in a somewhat different way. Let $kthdist$ and $kthdist_{OR}$ be the distances between the given query point q and its the kth nearest neighbor in the $\mathrm{R^{sb}}$-tree (R-tree and OR) and that only in OR, respectively. Intuitively, $kthdist_{OR}$ is an upperbound of $kthdist$ (i.e., $kthdist_{OR} \geq kthdist$). After obtaining the $kthdist_{OR}$ from OR, we perform a disk-based kNN search using $kthdist_{OR}$ as an initial value in a branch-and-bound manner [23]. During the search, it is needed to check the freshness of every possible answer. Finally, we can decide whether to take it as a correct answer or not.

4 Improving $\mathrm{R^{sb}}$-Tree with Advanced Techniques

4.1 Deferred Overflow/Underflow Treatment

As for the constraint of R-tree, the number of fan outs f of each node must satisfy the condition $f \in [m, M]$. Otherwise, the node must be splitted ($f > M$) or removed ($f < m$) to satisfy the condition. These operations are called overflow/underflow treatment procedures. Due to their high CPU and I/O cost, it is good to avoid these operations. With this in mind, we can simply use a smaller m (for example, 10 percent or 20 percent of M). However, this will change the entire tree structure and decrease the average number of fan outs and storage utilization of the tree. As a result, this will increase height of the tree, and overall update/search cost will be

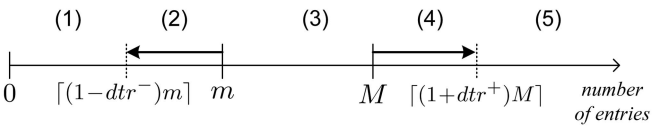

Fig. 4. Deferring overflow/underflow treatment with dtr coefficient.

increased. We suggest a new scheme that prevents underflow by inserting meaningless entries into the underflowed leaf node. Similarly, we can defer the insertion of new fresh entries in order to defer overflow treatment. Such a tolerance or flexibility is applied only for the leaf level, so that it does not change the whole structure of tree.

The meaningless entry is called *null entry*, a special class of obsolete entry, which has a form of $\langle$**null-id**, $null\_mbr, -\infty\rangle$, where **null-id** is a dedicated identifier for the null entry and $null\_mbr = [+\infty, -\infty]^d$. It does not belong to any specific object, while an obsolete entry belongs to a specific object and occupies a DL entry. Every time GC operates, obsolete entries of a leaf node are deleted and the null entry is filled if needed. The replacement of an obsolete entry with a null entry implies that it can be removed without any DL references. In order to limit the number of null entries in the R^{sb}-tree properly, we define the dtr coefficient as follows:

Definition 2 (Deferring Threshold Ratio (dtr)). *Given a constant $dtr \in [0, 1)$, called the* deferring threshold ratio, *overflow and underflow treatment for a leaf node L can be deferred. There are two kinds of dtr: dtr^- for m (underflow) and dtr^+ for M (overflow). By using these two dtr coefficients, underflow threshold m and overflow threshold M are modified to $\tilde{m} = \lceil(1 - dtr^-)m\rceil$ (called* deferred underflow threshold*) and $\tilde{M} = \lceil(1 + dtr^+)M\rceil$ (called* deferred overflow threshold*), respectively.*

We can consider a leaf node L with f leaf entries ($f \in [m, M], f = n_{fresh} + n_{obsolete}$, where $n_{obsolete}$ and n_{fresh} are the numbers of obsolete entries and fresh entries, respectively) and n_{OR} new entries to be inserted into L. (In case of operating GC, the procedure is identical except that overflow does not occur.) Let f' be the new number of leaf entries ($f' = n_{fresh} + n_{OR}$). There are five different cases for f':

$$\text{Five different cases for } f' = \begin{cases} \text{Case (1): } f' < \lceil(1 - dtr^-)m\rceil \\ \text{Case (2): } \lceil(1 - dtr^-)m\rceil \le f' < m \\ \text{Case (3): } m \le f' \le M \\ \text{Case (4): } M < f' \le \lceil(1 + dtr^+)M\rceil \\ \text{Case (5): } \lceil(1 + dtr^+)M\rceil < f'. \end{cases}$$

In summary, (3) is a normal or desired case, (2) and (4) are cases in which overflow/underflow is deferred by dtr, and (1) and (5) are cases in which split/deletion of node is actually performed. As seen in Fig. 4, the effect of the dtr coefficient is conceptually expanding the range $[m, M]$ where the split/deletion of node does not occur to the range $[\lceil(1 - dtr^-)m\rceil, \lceil(1 + dtr^+)M\rceil]$. We describe how to deal with each case below:

- Case (1): In the case of underflow, we first find the candidate set OL_{cand} from OR, such that $|OL_{cand}| \le M - f', OL_{cand} \cap OL_{flush} = \emptyset$, and $\forall o \in OL_{cand}, o \subset L.mbr$, and then make L to include OL_{cand} (lines 1-2 in Algorithm 4, UNDERFLOWTREATMENT). If underflow still occurs, the node must be deleted and the remaining entries are reinserted (line 5 in Algorithm 4, UNDERFLOWTREATMENT).[5] This case is depicted in Fig. 5a.
- Case (2): Like Case (1), find OL_{cand} and then put it into L. If the leaf node L meets the condition $f' \ge \lceil(1 - dtr^-)m\rceil$, fill it with $m - f'$ null entries in order to defer removing the leaf node (lines 3-4 in Algorithm 4, UNDERFLOWTREATMENT, see Fig. 5b).
- Case (4): Reload the remaining $f' - M$ (maximum $dtr^+ \cdot M$) entries into OR (lines 8-9 in Algorithm 4, OVERFLOWTREATMENT, see Fig. 5c). We adopt an R*-tree-like approach to choose M out of f' entries. For each axis, sort the entries by the lower then by the upper values of their rectangles and determine all possible $(f' - M + 1)$ distributions.[6] Choose a distribution with the *minimum-area* value. Therefore, the running time of the case (4) is $\mathcal{O}(d(f' \log f' + (f' - M + 1))) = \mathcal{O}(f' \log f')$, where $f' > M$ and $d = 2$.
- Case (5): This is the case that needs node split (see Fig. 5d). Create a new leaf node LL and write at most $2M$ entries to two leaf nodes L and LL. If there are remaining entries, load the entries into OR (lines 5-6 in Algorithm 4, OVERFLOWTREATMENT). In case of $f' > 2M$, choose two smallest MBRs of f' entries (lines 1-4 in Algorithm 4, OVERFLOWTREATMENT, see Fig. 5e). Applying the solution of "choose M out of f'" twice may cause the problem of local optima and a huge overlap between them. So, we devise a similar approach. For each axis, sort the entries by the lower then by the upper values of their rectangles and determine all possible $\sum_{i=1}^{f'-2M+1} i = \frac{(f'-2M+1)(f'-2M+2)}{2}$ distributions.[7] Then, we choose a distribution with the *minimum-area* value, and resolve ties by choosing the distribution with the *minimum-overlap* value. Therefore, the running time of the case (5) is $\mathcal{O}(d(M^2 + f'^2 - f'M)) = \mathcal{O}(f'^2)$, where $f' > 2M$ and $d = 2$.

5. In case (1), we can consider an alternative approach that reloads the remaining entries into OR (only for the leaf level). In our preliminary experiment, as stated in [3], [13], we observed that the reinsertion is superior than "reloading into OR" in the overall I/O improvement by restructuring the whole tree structure gradually.

6. In case (4), the sorted array of f' entries can be divided into three subarrays with i, M, and j entries, respectively, where $i + j = f' - M$ and $0 \le i, j \le f' - M$. The number of possible distributions is, therefore, $(f' - M + 1)$; i.e., $(0, M, f' - M), (1, M, f' - M - 1), \ldots, (f' - M, M, 0)$.

7. Similar to case (5), the sorted array of f' entries can be divided into five subarrays with i, M, j, M, and k entries respectively, where $i + j + k = f' - 2M, 0 \le i \le f' - 2M$, and $0 \le j \le f' - 2M - i$. The number of possible distributions is

$$\sum_{n=1}^{f'-2M+1} n = \frac{(f' - 2M + 1)(f' - 2M + 2)}{2}.$$

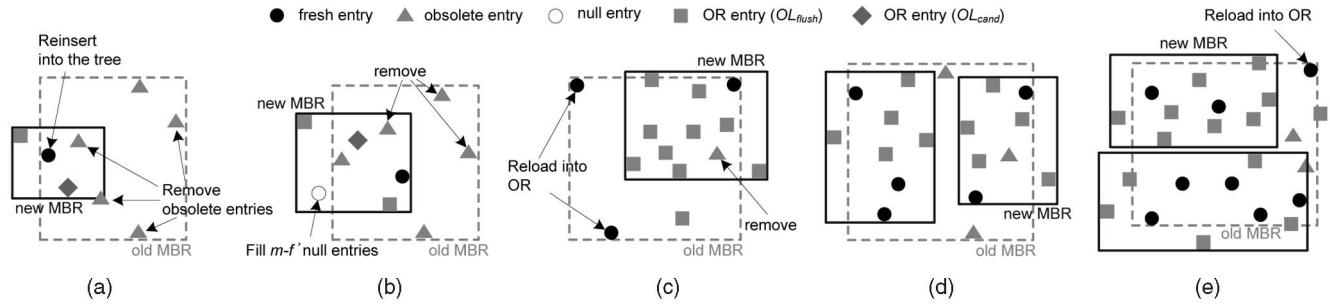

Fig. 5. Five different cases of underflow/overflow. In this example, $M = 10, m = 5, dtr^- = 0.3, dtr^+ = 0.3, \lceil (1 - dtr^-)m \rceil = 4, \lceil (1 + dtr^+)M \rceil = 13$. (a) Case (1). (b) Case (2). (c) Case (4). (d) Case (5): split. (e) Case (5): choose $2M$ entries.

Discussion. Intuitively speaking, for small dtr, the tree keeps its optimal status by performing frequent reorganizations such as node splits and reinsertions. For large dtr, however, the tree has a tendency to maintain its current status as much as possible. More specifically, a large dtr^- depreciates the tree's freshness by generating a great number of null entries, and this fact causes a bad influence on both update and search. Therefore, it is important to find the optimal dtr^- that decreases the update cost by delaying node deletion without any loss of the freshness. A large dtr^+ depreciates update performance by reducing the number of entries which are inserted from OR to the tree. However, this brings an affirmative effect of increasing the storage utilization due to delaying node splits and reducing the search cost eventually.

Algorithm 4. Overflow and Underflow Treatment algorithms

UNDERFLOWTREATMENT(Leafnode L)
1: Find OL_{cand} a set of (at most $M - f'$) OR entries that are completely enclosed by $L.mbr$.
2: Write OR entries in OL_{cand} into L;
3: **if** $(f' \geq \lceil (1 - dtr^-)m \rceil)$ **then**
4: Fill $m - f'$ null entries and write L into disk;
5: **else** Remove node L and reinserts the remaining f' entries;

OVERFLOWTREATMENT(Leafnode L)
1: **if** $(f' > 2M)$ **then**
2: Choose $2M$ out of f';
3: Create leaf node LL; Distribute $2M$ entries into L and LL;
4: Load the remaining $f' - 2M$ entries into OR;
5: **else if** $(f' > \lceil (1 + dtr^+)M \rceil)$ **then**
6: Create leaf node LL and split f' entries into L and LL;
7: **else**
8: Choose M out of f' entries (composing smallest MBR) and store those entries into L;
9: Load the remaining $f' - M$ entries into OR;

4.2 Improving LRU Page Buffer Hit Ratio

Assuming both the update and page buffer, the most efficient strategy for minimizing the update cost is 1) to defer and group generated updates as much as possible, and 2) to utilize the page buffer for performing these updates. Node accesses in R-trees have a property of hierarchical repeated accesses in nature. Consequently, the access probability of higher nodes is very high, but that of lower nodes is relatively low. From this observation we can introduce the following lemma.

Lemma 1 (Node Reuse Probability). *Under the assumption that every fan out in a node has the same access probability,* Node Reuse Probability *of the ith node $N_{l,i}$ at level l stored in a page buffer is defined as follows:*

$$P_{reuse}(N_{l,i}) = \frac{1}{f_{index}^{h-l-1}}, \tag{5}$$

where h is the height of the tree, and f_{index} is the average fan out of index node in the tree.

According to Lemma 1, the majority of the LRU page buffer at specific time will be filled with nonleaf nodes. However, every visited leaf node is also maintained within the buffer based on LRU page-replacement policy. Those leaf nodes (or the nodes which are close to the leaf-level) are rarely reused according to the lemma presented above. Therefore, by excluding these low reuse probability nodes from the buffer, we can increase possibility that highly reusable nodes are maintained.

Observation 1. Buffering leaf nodes with low reuse probability have a harmful influence on buffer hit ratio. It is sufficient to buffer only the nonleaf nodes for the buffer performance. More specifically, the cost-effective threshold of the buffer size will be decided by the number of nonleaf nodes $\sum_{i=1}^{h-1} n_i$ which is $\frac{1}{f_x}$ of the total number of nodes, because

$$\sum_{i=1}^{h-1} n_i \approx \frac{1}{f_x} \sum_{i=0}^{h-1} n_i < \frac{N}{(f_x - 1)f_0}. \tag{6}$$

More buffer memory allocation will not much help to improve performance (buffer hit ratio).

We can devise a more efficient buffer replacement policy by exploiting the level of nodes stored in the buffer.

Heuristic 1 (Level-Aware LRU replacement policy). *The Level-Aware LRU (LA-LRU for short) replacement policy is aware of the levels of nodes stored in the buffer space, and it chooses the least recently used entry among the lowest level entries in the buffer as a victim.*

Although the proposed LA-LRU policy has workload-dependent performance, it is remarkably efficient for the

repeated top-down update/search on the R-tree. Especially, this can also maximize the buffer hit ratio for GC whenever it needs to access nonleaf nodes. By using the proposed LA-LRU policy, we achieve more than 99.9 percent of the buffer hit ratio for accessing nonleaf nodes. In fact, Observation 1 can also be a good rationale for the concept of update buffer (OR). Namely, for a given fixed buffer space, we first allocate enough space to the LRU page buffer to keep nonleaf nodes that have higher reuse probabilities. In order to maximize the buffer efficiency, the rest of the buffer space will be allocated to OR.

5 Analytical Study

In this section, we now analyze update and storage costs of the three update strategies such as top-down update, memo-based update, and semibulk-loading-based update. We only discuss the case of data space $DS = [0,1)^d$, where dimensionality $d = 2$. And our analysis is based on [24], [25].

Lemma 2. *Let r_1 and r_2 be d-dimensional rectangles randomly distributed in the data space $DS = [0,1)^d$. Then, the probability $P_{Intr}(r_1, r_2)$ that they intersect with each other is $P_{Intr}(r_1, r_2) = \prod_{i=1}^{d}(|r_1|_i + |r_2|_i)$, where $|r|_i$ is the MBR length of rectangle r along ith dimension. Given a set of n_i rectangles with average size s and a query window q, the average number of intersected rectangles $intsect(n_i, s, q)$ is*

$$intsect(n_i, s, q) = n_i \cdot P_{Intr}(s, q) = n_i \prod_{j=1}^{d}(|s|_j + |q|_j). \quad (7)$$

Lemma 3. *The expected number of node accesses for answering a range query q is*

$$NA(q) = \sum_{i=0}^{h-1} intsect(n_i, s_i, q) = \sum_{i=0}^{h-1}\left[n_i \prod_{j=1}^{d}(|s_i|_j + |q|_j)\right], \quad (8)$$

where s_i is the average size of n_i rectangles at ith level. If we assume uniformly distributed data set and a well-structured R-tree with square-like MBRs, then the average extent of an ith level node, s_i is calculated by $\forall_{1 \le j \le d}|s_i|_j = (\frac{1}{n_i})^{1/d}$.

Hence, the expected number of node accesses for answering a point query q_0 s.t. $\forall |q_0|_i = 0$ is $NA(q_0) = \sum_{i=0}^{h-1}[n_i \prod_{j=1}^{d}|s_i|_j] = \sum_{i=0}^{h-1} n_i \|s_i\|$, where $\|s_i\|$ is the area of rectangle s_i; that is, $\|s_i\| = \prod_{j=1}^{d}|s_i|_j$.

5.1 Update Cost Analysis

In this section, we first analyze the update cost (UC) without considering the existence of a page buffer. Then, we revise this cost by considering a smart page buffer that filters out the accesses of nonleaf nodes. In the latter case, we only consider I/O cost for accessing leaf nodes (denoted as UC^{buffer}).

5.1.1 Top-Down Update Cost

As we already discussed in Section 2, the top-down update strategy performs an update as a delete-insert pair. Thus, we need to perform multipath search and one-path update for deletion, and one-path search and one-path update for insertion, respectively. By summing up these costs, we have

$$UC_{TD} = \overbrace{2h}^{\text{insertion cost}} + \overbrace{h + \sum_{i=0}^{h-1} n_i \|s_i\|}^{\text{deletion cost}}, \quad (9)$$

where tree height $h = 1 + \lceil \log_f(N/f) \rceil$ and f is the average *fan out*. And the update cost assuming a smart buffer can easily be computed as $UC_{TD}^{buffer} = 3 + n_0\|s_0\|$, where n_0 is the total number of leaf nodes, and s_0 is their average extent.

5.1.2 Memo-Based Update Cost

The memo-based update utilizes the traditional top-down insert and deferred deletion of obsolete entries. GC performs every K updates. Thus, its update cost is

$$UC_{Memo} = \overbrace{2h}^{\text{insertion cost}} + \overbrace{\frac{h+1}{K}}^{\text{GC cost}}. \quad (10)$$

And, the update cost assuming a smart buffer is

$$UC_{Memo}^{buffer} = 2 + \frac{2}{K}. \quad (11)$$

5.1.3 Semibulk-Loading-Based Update Cost

Assume that we have totally U updates to be handled ($U \gg K$), and γ is the OR hit ratio during the U updates ($\gamma > 0$). Thus, we have $U(1-\gamma)$ OR entries to be flushed into disk. Using FlushAll policy, we need to perform $\lfloor \frac{U(1-\gamma)}{ORSize} \rfloor$ flushes and the average update cost of FlushAll policy is

$$UC_{\text{FlushAll}} = \frac{\overbrace{\frac{2N \cdot f_x}{(f_x - 1)f_0}(1-\gamma)}^{\text{flush cost}}}{ORSize} + \frac{\overbrace{h + 1 + \sum_{i=0}^{h-2} f_x^{-i}}^{\text{GC cost}}}{K}. \quad (12)$$

This is upperbound because we assume that there are enough number of OR entries, so that every node in the tree will surely be visited (cf. (3) in Section 3.4). And the update cost assuming a smart buffer can easily be computed as

$$UC_{\text{FlushAll}}^{buffer} = \frac{2N/f_0(1-\gamma)}{ORSize} + \frac{2}{K}. \quad (13)$$

From now on, we will discuss the update cost of the FlushCell policy which chooses only the maximum histogram cell to be flushed. And we assume that the chosen cell has $\frac{\rho(g) \cdot ORSize}{g^2}$ entries, while the whole cells have $\frac{ORSize}{g^2}$ entries on the average. We call $\rho(g)$ the *skewness function*, and definitely $\rho(g) > 1$. Without knowing the details of data distribution, it is not easy to estimate $\rho(g)$ in a given data set.

$$UC_{\text{FlushCell}} = \frac{\overbrace{2 \cdot NA\left(\frac{1}{g} \times \frac{1}{g}\right)(1-\gamma)}^{\text{flush cost}}}{\rho(g) \cdot ORSize/g^2} + \frac{\overbrace{h + 1 + \sum_{i=0}^{h-2} f_x^{-i}}^{\text{GC cost}}}{K}. \quad (14)$$

And the update cost assuming a smart buffer is

$$\begin{aligned} UC_{\text{FlushCell}}^{buffer} &= \frac{2 \cdot intsect\left(\frac{N}{f_0}, s_0, \frac{1}{g} \times \frac{1}{g}\right)(1-\gamma)}{\rho(g) \cdot ORSize/g^2} + \frac{2}{K} \\ &= \frac{2\left(g + \sqrt{\frac{N}{f_0}}\right)^2(1-\gamma)}{\rho(g) \cdot ORSize} + \frac{2}{K}, \end{aligned} \quad (15)$$

where $s_0 = \frac{1}{n_0}$ because $|s_0|_1 = |s_0|_2 = \sqrt{1/n_0}$.

The above analyzed update cost of the proposed approach highly depends on the average OR hit ratio (γ) that can certainly be derived from the update probability of each object. We first considered that every object has the same probability of being updated. The OR hit ratio ($\gamma_{uniform}$) under this assumption is

$$\gamma_{uniform} = \frac{1}{ORSize} \sum_{i=0}^{ORSize} \frac{i}{N} = \frac{ORSize+1}{2N}.$$

For more realistic scenarios, we consider a different situation where each object has different update probability such as Zipf distribution [26]. In Zipf distribution, probability density function (pdf) is defined as

$$f(k;\alpha,N) = \frac{1/k^{\alpha}}{\sum_{n=1}^{N} 1/n^{\alpha}} = \frac{1}{k^{\alpha} H_{N,\alpha}},$$

where N is the number of elements, k is their rank, α is the exponent characterizing the distribution, and $H_{N,\alpha}$ is the Nth generalized harmonic number $\sum_{i=1}^{N} \frac{1}{i^{\alpha}}$. And cumulative distribution function (cdf) of $f(\cdot)$ is $F(k;\alpha,N) = \frac{H_{k,\alpha}}{k^{\alpha} H_{N,\alpha}}$. Thus, γ_{Zipf} can easily be estimated as $\gamma_{Zipf} = F(ORSize;\alpha, N) = \frac{H_{ORSize,\alpha}}{ORSize^{\alpha} H_{N,\alpha}}$.

Let us find conditions for satisfying the property $UC_{Memo}^{buffer} > UC_{\text{FlushAll}}^{buffer} > UC_{\text{FlushCell}}^{buffer}$. First, in order to satisfy the subproperty $UC_{Memo}^{buffer} > UC_{\text{FlushAll}}^{buffer}$, the inequality $2 > \frac{2n_0(1-\gamma)}{ORSize}$ must be satisfied (cf. (11) and (13)). Considering the upperbound condition of $\gamma = 0$, the constraint $ORSize > n_0$ must be satisfied. Second, in order to satisfy the subproperty $UC_{\text{FlushAll}}^{buffer} > UC_{\text{FlushCell}}^{buffer}$, the inequality $n_0 > \frac{(g+\sqrt{n_0})^2}{\rho(g)}$ must be satisfied (cf. (13) and (15)). Thus, the constraint $\rho(g) > (\frac{g}{\sqrt{n_0}} + 1)^2$ must be satisfied, where $g > 1$. In summary, in order to satisfy the property $UC_{Memo}^{buffer} > UC_{\text{FlushAll}}^{buffer} > UC_{\text{FlushCell}}^{buffer}$, the following constraints must be satisfied: $ORSize > n_0$ and $\rho(g) > (\frac{g}{\sqrt{n_0}} + 1)^2$, where $g > 1$. Generally, these constraints hold.

5.2 Storage Cost Analysis

In this section, we analyze the extra storage costs of the memo-based update and the proposed approach over R-tree incurred by maintaining obsolete entries.

5.2.1 Memo-Based Storage Cost

GC visits the next leaf node every K updates and the traversal of the entire leaf nodes, therefore, needs $K \cdot n_0$ updates. This is the maximum number of obsolete entries in RUM-tree; i.e., $max(N_{obsolete}) = K \cdot n_0$. Statistically, about half of leaf nodes will be further cleaned by clean-upon-touch process [4]. Therefore, $avg(N_{obsolete}) = \frac{K \cdot n_0}{2}$.

5.2.2 Semibulk-Loading-Based Storage Cost

$avg(N_{obsolete})$ is quite similar to that of RUM-tree; however, it will be somewhat increased by $ORSize$ because of the reduced clean-upon-touch processes by buffered updates in OR; that is, $avg(N_{obsolete}) = \frac{(K \cdot n_0 + ORSize)}{2}$.

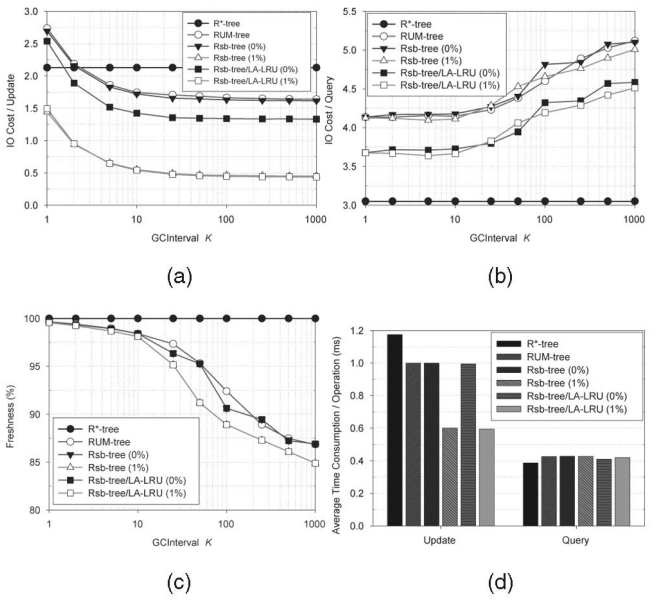

Fig. 6. Effect of $GCInterval\ K$. (a) Update cost. (b) Query cost. (c) Freshness. (d) Running time comparison (for chosen $K = 10$).

In case of $dtr^{-} > 0$, the deferred underflow threshold $\tilde{m} = \lceil (1 - dtr^{-})m \rceil$ (simply, $\tilde{m} = (1 - dtr^{-})m$). Therefore, the maximum number of null entries in a leaf node and in tree are $dtr^{-} \cdot m$ and $max(N_{null}) = n_0 \cdot dtr^{-} \cdot m$, respectively. Under the assumption that the number of non-null entries in a leaf node is uniformly distributed in $[\tilde{m}, M]$, the average number of null entries, $avg(N_{null}) = \sum_{\tilde{m} \le i \le m} \frac{n_0}{M - \tilde{m} + 1}(m - i) = \frac{n_0}{M - \tilde{m} + 1} * \frac{dtr^{-} \cdot m(1 + dtr^{-} \cdot m)}{2}$.

5.3 Optimizing $GCInterval\ K$

As analyzed in Section 5.1, the update cost of the proposed approach will simply be minimized with larger K. However, an overlarge K incurs too many obsolete entries in R^{sb}-tree so that the query cost will surely be increased. Furthermore, this also increases the cost for locating an optimal leaf node for inserting a new entry which is highly dependent on the tree's height h, then it will compensate the reduction of the update cost. Consequently, at some point of K, the update cost has no more performance gain (see Fig. 6a).

In this section, we intend to optimize K in order to minimize the update cost with considering the query cost simultaneously; that is, maximizing K (denoted as K^{*}) as long as it minimizes the update cost. In other words, our objective is to maximize $N_{obsolete}$ maintaining the height. In short, we have an optimization problem:

Update Cost Optimization Problem:

$$\text{Maximize } N_{obsolete} \text{ subject to } h(N) = h(N + N_{obsolete}),$$

where $h(n)$ returns the tree's height with n leaf entries; i.e., $h(n) = 1 + \lceil \log_{f_x} \frac{n}{f_0} \rceil$. Let $upperbound(N_{obsolete})$ be the maximized $N_{obsolete}$ that satisfies the above condition. Naturally, it would be

$$upperbound(N_{obsolete}) = f_0 \cdot f_x^{h(N)-1} - N, \qquad (16)$$

TABLE 2
Experimental Parameters and Their Values

Parameters	Value Used (Default)
N (The num. of moving objects)	1M $\sim$ 10M (**1M**)
U (The num. of updates)	3M $\sim$ 30M (**3M**)
$LRUBufferSize$	0% $\sim$ 10% (**1%**)
dtr^+, dtr^-	0 $\sim$ .5 (**0.0**)
$GCInterval\ K$	1 $\sim$ 1,000 (**10**)
$ORSize$ (The capacity of OR)	0% $\sim$ 10% of N (**1%**)
Movement Datasets	UniformNetwork, ZipfRandom

where $f_0 \cdot f_x^{h(N)-1}$ is the maximum number of leaf entries with the tree's height $h(N)$. Because $avg(N_{obsolete}) = \frac{(K \cdot n_0 + ORSize)}{2}$, similarly K^* would be

$$\begin{aligned} K^* &= \frac{2 \cdot upperbound(N_{obsolete}) - ORSize}{n_0} \\ &= \frac{2 \cdot upperbound(N_{obsolete}) - ORSize}{(N + upperbound(N_{obsolete}))/f_0} \\ &= \frac{2\big(f_0 \cdot f_x^{h(N)-1} - N\big) - ORSize}{f_x^{h(N)-1}}. \end{aligned}$$

Solving the above equation with the given variables in Section 6 such as $N = 10^6, ORSize = 1\ \% \cdot N, f_x = 136, f_0 = 97$ (67 percent storage utilization, 4 kB page) and $h(N) = 3$, we have $upperbound(N_{obsolete}) \approx 79.4\ \% \cdot N$ and $K^* \approx 85$. This result is in correspondence with the experiment results in Section 6 (see Fig. 6a).

6 Performance Evaluation

6.1 Experiment Setup

In this section, we evaluate the performances of R*-tree, RUM-tree, R^{sb}-tree with different $ORSize$ (0 percent and 1 percent of the database) and different buffer management policies such as LRU and LA-LRU with 1 percent of R-tree nodes. By default, we use LRU as replacement policy for all techniques. For fair comparison, the number of I/Os for updates and queries—as the primary performance metric—is carefully monitored in every experiment. In particular, our proposed approach has extra logging cost for OR under a 64 kB log buffer[8] and the FlushCell policy will be used by default. In order to verify the query processing power, we run 100,000 range queries whose side lengths are randomly chosen from $[0, 0.03]$. In all experiments, we set the page size as 4 kB, and histogram granularity (g) is fixed as 32. All experiments are conducted on Intel Pentium 4, 2 GHz and 2 GB RAM running on Linux system. Experimental parameters and their default values, given in bold, are summarized in Table 2.

For generating movement data set, we cannot generate a single perfect data set for simulating all kinds of characteristics of a real-world phenomena. We use two different data sets such as UniformNetwork and ZipfRandom in order to represent different characteristics such as data distribution, update frequency, and movement patterns. In UniformNetwork data set generated by the *Network-based Generator of Moving Objects* by Brinkhoff [27] with the road network of Oldenburge, there are 1 M moving objects and they move along the road network. In the data set, there is no specific update pattern, and every object has equally the same update probability. UniformNetwork is for simulating realistic movement patterns and data distribution, and its name stands for its *uniform* update pattern and *network*-based movement patterns. In ZipfRandom data set generated by our GSTD-like generator [28], there are 1 M moving objects and they random-walk in $[0, 1]^2$. In the data set, the update probability of each object follows Zipf-like distribution [26]; that is, there exist a few fast moving objects and many slow moving objects together. ZipfRandom is for simulating realistic update patterns, and its name stands for its *Zipf* update pattern and *random-walk* behavior in objects' movement. Both data sets are normalized into $[0, 1]^2$ data space, and have 3 M updates in total.

8. This is a typical size of log buffer in commercial DBMS. http://www.postgresql.org/docs/8.2/interactive/index.html.

6.2 Experiment with Network-Based Data Set (UniformNetwork)

6.2.1 Effect of Garbage Cleaner's Interval K

One of the most influential parameters for the proposed approach is $GCInterval\ K$ (K for short) that controls the freshness of the tree. Therefore, choosing an appropriate K is essential for performance optimization. The results varying K are given in Fig. 6.

Fig. 6a shows the update cost with different K. As expected, the results show that the update cost will be greatly decreased by increasing K, and it will remain the same in case of $K \geq 100$. This is because the GC cost will surely be decreased by K, but the increased flush cost by the reduced freshness (by increasing K) will cancel this performance gain. Except for $K = 1$, the proposed approach far outperforms the existing techniques, even for the case of $ORSize = 0\%$ where there is no additional memory requirement. As discussed in Section 3.4, this is mainly due to the fact that the GC of R^{sb}-tree exploits $dfsStack$ instead of complex pointers such parent-link and leaf-level links which have high maintenance cost. Moreover, applying LA-LRU further decreases the update cost without additional overhead especially for the case of $ORSize = 0\%$ where more leaf nodes are visited. One of the most interesting results is that the update costs of existing techniques are always higher than 1. The main reason for this behavior of existing techniques is that for performing an update, they need at least one leaf node access (two disk I/Os: read and write). Owing to the grouping behavior of incoming updates, however, the proposed approach can achieve excellent update performance; i.e., the I/O cost per update can be much lower than 1.0.

From Fig. 6b, we observe that the search costs of all techniques increase as K increases. Generally, the search cost of R^{sb}-tree ($ORSize = 0\%$) is almost similar to that of RUM-tree because of their similar behaviors. Due to the update buffering behavior of R^{sb}-tree ($ORSize = 1\%$), it performs fewer clean-upon-touch processes than RUM-tree. This effect incurs the decreased freshness, so that its search cost will be slightly increased. However, for a smaller K ($K \leq 10$), the query cost of R^{sb}-tree ($ORSize = 1\%$) is slightly smaller than that of RUM-tree and R^{sb}-tree ($ORSize = 0\%$).

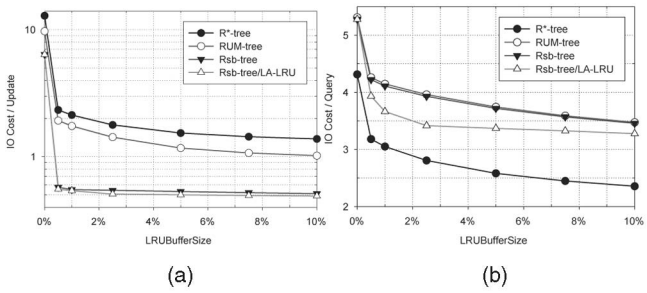

Fig. 7. Effect of page buffering. (a) Update cost. (b) Query cost.

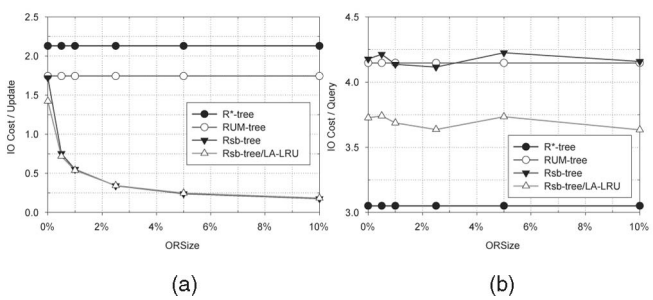

Fig. 8. Effect of the capacity of OR ($ORSize$). (a) Update cost. (b) Query cost.

This is mainly due to the fact that almost obsolete entries are eliminated from the tree by performing frequent GCs, so that the tree will be shrunk by $ORSize$.

Fig. 6c shows the results of freshness varying K. As expected, the freshness naturally decreases as K increases. Overall, RUM-tree has the highest freshness, followed by $\mathrm{R^{sb}}$-tree ($ORSize = 0\%$) and then $\mathrm{R^{sb}}$-tree ($ORSize = 1\%$). From the results of Figs. 6a, 6b, and 6c we can quantitatively decide an appropriate $K(= 10)$. In case of the chosen $K = 10$, the freshness of RUM-tree, $\mathrm{R^{sb}}$-tree ($ORSize = 0\%$), and $\mathrm{R^{sb}}$-tree ($ORSize = 1\%$) are 98.41 percent, 98.40 percent, and 98.11 percent, respectively.

Fig. 6d illustrates the average time consumption of each technique for performing an operation such as update and query. For average time consumption of update operation, the two proposed approaches are 40.0-49.2 percent faster than other techniques. For average time consumption of query operation, the proposed approaches offer 4.23 percent better performance than RUM-tree. However, they perform slightly worse (10.3 percent) than $\mathrm{R^*}$-tree.

6.2.2 Effect of Page Buffering

In this experiment, we investigate both the update and query cost varying the size of LRU page buffer ($LRUBufferSize$) from 0 percent to 10 percent. For all techniques, update cost (Fig. 7a) decreases with increased buffer size, as can be expected, and the proposed approach is significantly better than the rest. From the results, we can conclude that 1 percent LRU page buffer is sufficient to optimize the update cost, thanks to the skewed behavior of search and update in R-trees. Unlike the update cost, the query cost (Fig. 7b) of all techniques are continuously decreasing in proportion to $LRUBufferSize$. The reason is that in comparison to updates, queries retrieve a relatively large number of nodes especially leaf nodes. Due to the same reason, the performance improvement of LA-LRU for search cost is much greater than for update cost. In terms of both update and query cost, the proposed two approaches such as $\mathrm{R^{sb}}$-tree and $\mathrm{R^{sb}}$-tree/LA-LRU perform better than RUM-tree.

6.2.3 Effect of $ORSize$

We investigate the effect of the capacity of OR from 0 percent to 10 percent of N. The capacity of OR ($ORSize$) is mainly for improving the update performance, and it has three important aspects. First, the overall search cost will be somewhat increased as $ORSize$ increases. This is because $\mathrm{R^{sb}}$-tree performs fewer clean-upon-touch processes than RUM-tree, and it is owing to its update buffering behavior. A large $ORSize$ incurs the detrimental effect on the total number of obsolete entries and freshness of the tree consecutively. Second, as $ORSize$ increases, the OR hit ratio (γ) will be accordingly increased. As a result, removing the obsolete entries can be performed in memory instead of disk. And third, the disk-resident part of $\mathrm{R^{sb}}$-tree gets smaller with increasing $ORSize$. This interesting effect is owing to the fact that fresh entries migrate to the in-memory buffer (OR). The average shrinkage of R-tree index is $(1 - \frac{\rho(g)}{2g^2})ORSize$ for FlushCell policy.

As we can observe in Fig. 8, on the whole, the proposed approach outperforms the existing techniques in terms of update and query cost. Especially for the update cost of $ORSize \geq 5\%$, the proposed approach at least seven times outperforms existing approaches. However, over 5 percent of $ORSize$ has little effect on the update performance, since the FlushCell policy only chooses a single cell with the maximum histogram.

6.2.4 Effect of Different Speed Profiles

In this experiment, we vary the maximum distance between consecutive updates by choosing different speed profiles such as low-speed, mid-speed, and high-speed (defined in the Network-based generator [27]). As can be seen in Fig. 9, the update cost of $\mathrm{R^*}$-tree increases as movement speed increases. When objects move slowly, the update cost of $\mathrm{R^*}$-tree decreases because there is a tendency to perform an update at the same leaf node. However, the update performances of RUM-tree and $\mathrm{R^{sb}}$-tree nearly remain the same under different speed profiles because of their way of performing update: an update is decomposed into an insert and deferred group deletion. For query performance, all techniques gradually deteriorate as movement speed increases. This is due to the increased node extent and resultant increased node overlapping.

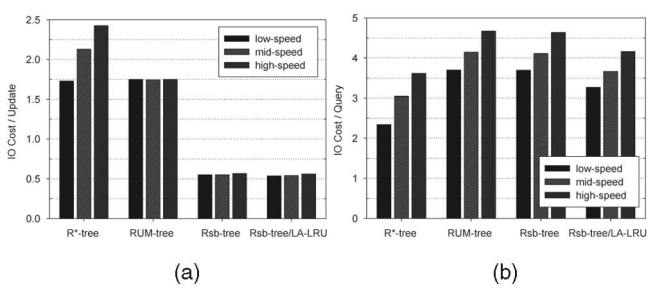

Fig. 9. Effect of different speed profiles. (a) Update cost. (b) Query cost.

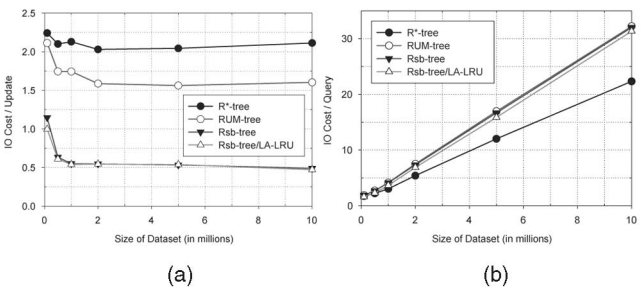

Fig. 10. Effect of the size of data set. (a) Update cost. (b) Query cost.

6.2.5 Effect of the Size of Data Set (Scalability)

In this experiment, we vary the size of data set (N) from 100,000 to 10 M in order to see how scalable the proposed approaches are compared to existing techniques. For update cost (Fig. 10a), the proposed approach outperforms all existing techniques under all conditions. Note that for a large N ($N \geq 2$ M), the update costs of all techniques remain nearly the same, since $LRUBufferSize$ also increases accordingly. Moreover, the update cost is closely related to the tree height which is a log-scale relationship with N. In spite of the increase of $LRUBufferSize$, the query cost monotonically increases in proportion to the size of data set (N), not to the tree height (see Fig. 10b). This is because the average extent of the query window is the same for all cases.

6.2.6 Effect of dtr Coefficient

We have discussed the performance benefit of the proposed approach, however this benefit can further be improved by fine-tuning the dtr coefficient. In this experiment, we vary two dtr coefficients (dtr^- and dtr^+) from 0.0 to 0.5 to investigate the effect of dtr in terms of I/O cost and CPU time. From the results of Figs. 11a and 11b, we can easily find the best pair ($dtr^- = 0.5, dtr^+ = 0.0$) for optimizing the update cost. This means that a practical solution for improving the update performance is to 1) defer underflow treatment (node removal and reinsertion of the remaining entries) as much as possible, and 2) not to defer overflow treatment. If overflow treatment is deferred, the number of OR entries actually flushed into disk will be decreased. With the optimal pair of $(0.5, 0.0)$, I/O cost and CPU time for performing an update are minimized by 4.5 percent and 18 percent, respectively.

From the results of Figs. 11c and 11d, we can observe the effect of dtr on search cost. The behavior of search cost is somewhat complicated. In general, a larger dtr^- that helps to minimize the update cost increases the query cost by producing a large number of underflowed nodes. On the other hand, a larger dtr^+ will help to minimize the search cost by maximizing storage utilization which is owing to the avoidance of frequent node splits. As we can see in Fig. 11c, the optimal pair for minimizing the query I/O cost is $(0.1, 0.5)$ and it reduces the cost by 5.3 percent. Moreover, by the pair $(0.5, 0.1)$, the query CPU time (Fig. 11d) can also be minimized by 14.5 percent.

6.2.7 Effect of Buffer Allocation

In this experiment, we investigate the update and query performance of all techniques under the same memory requirement. Given the total size of memory buffer (BS), buffer allocation is denoted as $BufferAlloc(BS, r)$, where r is the fractional size of BS for LRU page buffer ($r = 0$-100%). The rest of memory space ($BS \cdot (1 - r)$) is allocated for the proposed in-memory buffer (OR, H, and DL). DL is occupied by DL entries as much as obsolete entries in the tree at most, and the rest is available for OR and H. Naturally, the existing techniques can only have LRU page buffer. In this experiment, we adopt three representative BS of 512 kB, 1,024 kB, and 2,048 kB, and we assume that the byte sizes of an OR entry and a DL entry are 28 and 14 bytes, respectively.[9]

As we can observe in Fig. 12a, the proposed approach is superior to the others in terms of update cost, except two extreme cases ($r \leq 1\%$). We believe that one of the practically optimal threshold of r (denoted as r^*) for LRU page buffer is strongly related to the number of nonleaf nodes (6) and this must be closely related to the size of the hypothetical smart page buffer discussed in Section 5. The size of the page buffer which accommodates index nodes only is $BS \cdot r^* = PageSize \cdot \sum_{i=1}^{h-1} n_i$. Thus, we have

$$r^* = \frac{PageSize}{BS} \cdot \sum_{i=1}^{h-1} n_i \approx \frac{PageSize}{BS} \cdot \frac{N}{(f_x - 1) f_0}.$$

It should be satisfied that $r^* \in (0, 1)$ and $BS > PageSize \cdot \sum_{i=1}^{h-1} n_i$.

As we already discussed before, OR is mainly for improving update cost. Thus, the query cost (Fig. 12b) increases for a smaller r ($r \leq 10\%$), i.e., a larger OR. Generally, in case of $r \geq 25\%$, the query cost of the proposed approach is smaller than or equal to that of RUM-tree.

6.3 Experiment with Zipf Data Set (ZipfRandom)

In this section, we conduct experiments on ZipfRandom data set where users' update patterns follow a Zipf-like distribution. Assuming a Zipf-like distribution for users' update patterns, the update frequency for the ith hot object is proportional to $1/i^\alpha$, where α is called the *Zipf rank exponent*. By default, the exponent α is set to 1. The effect of the experimental parameters discussed so far on ZipfRandom data set is similar to that on UniformNetwork data set. In this section, we, therefore, conduct two newly designed experiments that is more suitable for ZipfRandom data set.

6.3.1 Effect of Zipf Rank Exponent α

In this experiment, we vary the Zipf rank exponent α from 0.25 to 2.0. As we can observe in Fig. 13a, the update cost of all techniques gradually decreases when α increases, and the proposed approaches are superior to the existing techniques. If the update pattern becomes skewed (i.e., a larger α), the majority of updates will be generated by a small number of objects. These updates can be performed by repeatedly visiting the minority of nodes, so that the page buffer hit ratio will be increased accordingly. For the

9. We assume that an OR entry is consisted of 4-byte object-id oid, two 4-byte floats for location p, two 1-byte for cid, 8-byte long for $tstamp$, 2-byte short integer for N_{hit}, and 4-byte integer for histogram entry for Histogram (H). We also assume that a DL entry consists of 4-byte object-id oid, 8-byte long for t_{upd}, 2-byte short integer for N_{old}.

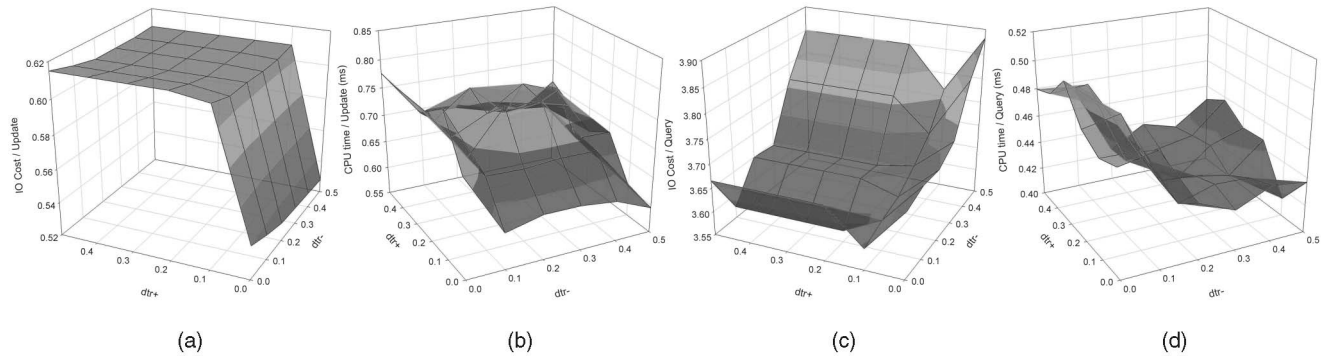

Fig. 11. Effect of dtr coefficient. (a) Update I/O cost. (b) Update CPU time. (c) Query I/O cost. (d) Query CPU time.

same reason, moreover, R*-tree outperforms RUM-tree for highly skewed data sets ($\alpha > 1.0$); generally, R*-tree has fewer nodes than RUM-tree and $\mathrm{R^{sb}}$-tree because of their space overhead (the time stamp tagged by every leaf entry), so that given a fixed page buffer, the buffer hit ratio of R*-tree is surely bigger than that of RUM-tree and $\mathrm{R^{sb}}$-tree. The superiority of the proposed approach over the existing techniques can be explained by the concept of update buffer (OR). By exploiting OR, a large number of updates can be filtered (updated within OR) from the disk-based R-tree index. This will make a huge performance benefit.

In terms of query cost (Fig. 13b), R*-tree always outperforms the rest, and the proposed approaches perform better than RUM-tree. Overall, the query costs of all techniques remain still without a direct influence of α, since this parameter has an important influence mainly on the update cost, not the query cost.

6.3.2 Performance Comparison of Different Flush Policies Varying FlushDeltaFactor Δ

Fig. 14 shows the results of different flush policies varying Δ (called *FlushDeltaFactor*) discussed in Section 3.3. Overall, as Δ increases, the update costs of the FlushLRUCell and FlushLFUCell increase (see Fig. 14a). In case of a smaller Δ ($\Delta < 0.5$), however, FlushLFUCell policy outperforms FlushCell policy which is used as default. Naturally, when Δ increases, OR hit ratios of FlushLRUCell policy and FlushLFUCell policy increase, since more hot objects will be kept in memory (see Fig. 14b). However, the badly chosen hot objects can incur the deterioration of the update performance, even though OR hit ratio is increased. As we can observe in Fig. 14, this undesirable phenomenon can occur in FlushLRUCell policy; this means that identifying the hot objects on the basis of N_{hit} in FlushLFUCell is better than FlushLRUCell based on $tstamp$. Similar to the previous experiment, the parameter Δ has no direct influence on the query costs of all flush polices.

7 Conclusions and Future Work

In this paper, we have investigated the problem of indexing moving objects and managing their frequent updates in update-intensive environments. We proposed an R-tree-based solution, called $\mathrm{R^{sb}}$-tree, that exploits a small in-memory buffer to buffer, defer, and group incoming updates. With reasonable memory overhead of 1 percent of database size (or even with the same memory requirement), our approach is at least two times faster and incurs I/O cost much lower by a factor of three to five than the existing techniques. With the concept of dtr, we can further improve the CPU and I/O cost of the proposed approach. We believe that the proposed approach is orthogonal and it can be applied to other index structures, e.g., B-trees, Quadtrees, and k-d-B-trees.

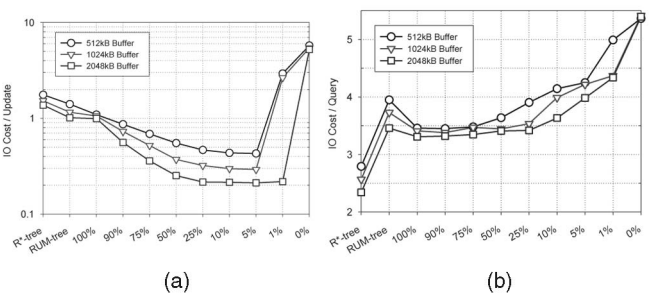

Fig. 12. Effect of Buffer Allocation $BufferAlloc(BS, r)$. (a) Update cost. (b) Query cost.

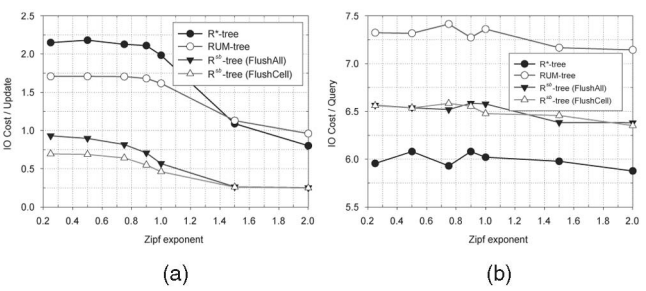

Fig. 13. Effect of Zipf rank exponent α. (a) Update cost. (b) Query cost.

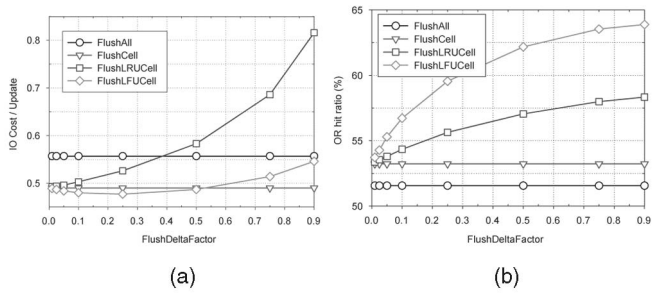

Fig. 14. Performance comparison of different flush policies varying Δ.

Our future work may include

- Extensions of the proposed flush algorithm in order to improve update and query performance together.
- Efficient data declustering technique for supporting decentralized environments such as peer-to-peer and cluster computing environments.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions that improved the quality of this paper. They appreciate the support by JSPS Postdoctoral Fellowships for Foreign Researchers. This work has been supported in part by the Grant-in-Aid for Scientific Research from JSPS (#18-06368, #18200005, and #21240005), MEXT (#19024006), and JST CREST. MoonBae Song's work was also supported in part by the Korea Research Foundation Grant funded by the Korean Government (KRF-2006-352-D00156).

REFERENCES

[1] O. Wolfson, "Moving Objects Information Management: The Database Challenge," *Proc. Workshop Next Generation Information Technologies and Systems (NGITS),* 2002.
[2] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 419-429, 1994.
[3] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 47-57, 1984.
[4] X. Xiong and W.G. Aref, "R-Trees with Update Memos," *Proc. Int'l Conf. Data Eng. (ICDE),* 2006.
[5] M.-L. Lee, W. Hsu, C.S. Jensen, B. Cui, and K.L. Teo, "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach," *Proc. Int'l Conf. Very Large Data Bases (VLDB '03),* pp. 608-619, 2003.
[6] D. Kwon, S. Lee, and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree," *Proc. Int'l Conf. Mobile Data Management (MDM '02),* pp. 113-120, 2002.
[7] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," *Proc. Int'l Conf. Very Large Data Bases (VLDB '00),* pp. 395-406, 2000.
[8] S. Saltenis, C.S. Jensen, S. Leutenegger, and M. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 331-342, 2000.
[9] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch, "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects," *IEEE Trans. Computers,* vol. 51, no. 10, pp. 1124-1140, Oct. 2002.
[10] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 790-801, 2003.
[11] B. Cui, D. Lin, and K.-L. Tan, "IMPACT: A Twin Index Framework for Efficient Moving Object Query Processing," *Data and Knowledge Eng.,* vol. 59, no. 1, pp. 63-85, 2006.
[12] B.C. Ooi, K.-L. Tan, and C. Yu, "Frequent Update and Efficient Retrieval: An Oxymoron on Moving Object Indexes?" *Proc. Int'l Conf. Web Information Systems Eng. Workshop,* pp. 3-12, 2002.
[13] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-331, 1990.
[14] T. Brinkhoff, "A Robust and Self-Tuning Page-Replacement Strategy for Spatial Database Systems," *Proc. Int'l Conf. Extending Database Technology (EDBT '02),* pp. 533-552, 2002.
[15] G.M. Sacco, "Index Access with a Finite Buffer," *Proc. Int'l Conf. Very Large Data Bases (VLDB '87),* pp. 301-309, 1987.
[16] S. Leutenegger and M. Lopez, "The Effect of Buffering on the Performance of R-Trees," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 1, pp. 33-44, Jan./Feb. 2000.
[17] G. Graefe, "B-Tree Indexes for High Update Rates," *ACM SIGMOD Record,* vol. 35, no. 1, pp. 39-44, 2006.
[18] L. Arge, K. Hinrichs, J. Vahrenhold, and J.S. Vitter, "Efficient Bulk Operations on Dynamic R-Trees," *Algorithmica,* vol. 33, no. 1, pp. 104-128, 2002.
[19] B. Lin and J. Su, "Handling Frequent Updates of Moving Objects," *Proc. Int'l Conf. Information and Knowledge Management (CIKM '05),* pp. 493-500, 2005.
[20] X. Xiong, M. Mokbel, and W. Aref, "LUGrid: Update-tolerant Grid-based Indexing for Moving Objects," *Proc. Int'l Conf. Mobile Data Management (MDM '06),* pp. 13-13, May 2006.
[21] L. Biveinis, S. Saltenis, and C.S. Jensen, "Main-Memory Operation Buffering for Efficient R-Tree Update," *Proc. Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 591-602, 2007.
[22] S. Leutenegger, M. Lopez, and J. Edgington, "STR: A Simple and Efficient Algorithm for R-Tree Packing," *Proc. Int'l Conf. Data Eng. (ICDE '97),* pp. 497-506, 1997.
[23] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.
[24] Y. Theodoridis and T. Sellis, "A Model for the Prediction of R-Tree Performance," *Proc. ACM Int'l Symp. Principles of Database Systems (PODS '96),* pp. 161-171, 1996.
[25] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis, "An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 10, pp. 1169-1184, 2004.
[26] G. Zipf, *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.
[27] T. Brinkhoff, "Network-Based Generator of Moving Objects," http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/, 2009.
[28] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento, "On the Generation of Spatiotemporal Data Sets," *Proc. Symp. Large Spatial Databases (SSD '99),* pp. 147-164, 1999.

MoonBae Song received the PhD degree in computer science from Korea University in 2005. From 2005 to 2006, he worked as a research professor at Korea University, Seoul. He was a visiting researcher at the University of Tsukuba invited and supported by the Japan Society of Promotion of Science (JSPS), from 2006 to 2008. Currently, he is a research professor at Sungkyunkwan University, Suwon, and has been since December 2008. His research interests include mobile and ubiquitous computing, system support for location/context-awareness, query processing and optimization, and data management techniques in large-scale distributed systems. He served as a program cochair for the First International Workshop on Data and Semantic Engineering for Emerging Technology and Services (DASE '08), and as a program committee member for the 20th International Conference on Database and Expert Systems Applications (DEXA '09). He is a member of KISS and IEICE.

Hiroyuki Kitagawa received the BSc degree in physics and the MSc and DrSc degrees in computer science, all from the University of Tokyo, in 1978, 1980, and 1987, respectively. He is currently a full professor at Graduate School of Systems and Information Engineering and at Center for Computational Sciences, University of Tsukuba. His research interests include integration of information sources, data mining, stream-based ubiquitous data management, web data management, XML, and scientific databases. He has published more than 150 papers in refereed journals and international conference proceedings. He is a fellow of the IPSJ (Information Processing Society of Japan) and the IEICE (The Institute of Electronics, Information and Communication Engineers), a Trustee of the DBSJ (The Database Society of Japan), and a member of the ACM, the IEEE Computer Society, and the JSSST. He served as Chairperson of the IEICE Special Interest Group on Data Engineering from 1999 to 2001, Chairperson of ACM SIGMOD Japan Chapter from 2003 to 2007, and Vice Chairperson of DBSJ from 2005 to 2007.