LUGrid: Update-tolerant Grid-based Indexing for Moving Objects *

Xiaopeng Xiong[1] Mohamed F. Mokbel[2] Walid G. Aref[1]

[1]Department of Computer Science, Purdue University, West Lafayette, IN

[2]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN

Abstract

Indexing moving objects is a fundamental issue in spatio-temporal databases. In this paper, we propose an adaptive Lazy-Update Grid-based index *(LUGrid, for short) that minimizes the cost of object updates. LUGrid is designed with two important features, namely,* lazy insertion *and* lazy deletion. Lazy insertion *reduces the update I/Os by adding an additional memory-resident layer over the disk index.* Lazy deletion *reduces update cost by avoiding deleting single obsolete entry immediately. Instead, the obsolete entries are removed later by specially designed mechanisms. LUGrid adapts to object distributions through cell splitting and merging. Theoretical analysis and experimental results indicate that LUGrid outperforms former work by up to eight times when processing intensive updates, while yielding similar search performance.*

1 Introduction

The integration of mobile devices and positioning technologies enables new environments where locations of moving objects can be tracked continuously. In such environments, objects send their current locations to a server either periodically or based on their moving distance. The server collects the location information and processes interested queries. A wide range of applications rely on the maintenance of current locations of moving objects. Examples of these applications include traffic monitoring, nearby information accessing and enhanced 911 service, etc.

However, existing indexes for moving objects (as discussed in Section 2) suffer from intensive updates. The reasons are observed from the following three aspects. First, most of the indexing approaches process one single update at a time, which hinders largely the update ability of the index. Second, most of existing approaches locate and remove the old object entry upon an update. This process incurs large disk overhead. Third, to quickly search the old object entry to be removed, many index structures maintain a secondary index on object IDs (e.g., [8, 9, 10, 24]). For each update, the secondary index is searched to locate the old object entry. Further, the secondary index has to be updated every time an object changes its locality of disk page.

In this paper, we propose $LUGrid$, an adaptive *Lazy-Update Grid-based index* for indexing current locations of moving objects. LUGrid aims to avoid all the above mentioned drawbacks of existing indexing techniques. LUGrid is designed with two important features: (1) *Lazy-insertion*. In LUGrid, object updates going to the same disk page are buffered together before flushed to disk in one run. Lazy-insertion avoids excessive I/O costs caused by multiple independent updates so that the amortized I/O cost for one update is low; (2) *Lazy-deletion*. In contrast to other indexing approaches, LUGrid does not require deleting old entries before inserting new entries. Instead, LUGrid delays the deletion process until the disk pages where the old entries reside are retrieved into memory. This is achieved by a structure called "miss-deletion memo" (MDM). LUGrid guarantees that the size of MDM is upper-bounded to a small size so that it can be easily accommodated in main memory. Besides, LUGrid adapts to object distribution by inheriting the structure of *Grid file*.

The contributions of this paper are summarized as follows:

1. We propose LUGrid; an adaptive update-tolerant indexing structure for indexing current locations of moving objects. LUGrid is designed to minimize the cost of processing object updates.
2. We present the structure of LUGrid and algorithms for update and query processing along with an analysis of the update cost in LUGrid.
3. We provide a comprehensive set of experiments demonstrating that LUGrid outperforms significantly former work in update processing while maintaining similar query performance.

The rest of this paper is organized as follows. Section 2

*This work was supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120.

highlights related work. LUGrid is discussed in Section 3. Section 4 gives an analysis of the update cost of the proposed *update* scheme. Experiments evaluating the performance of LUGrid are presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Traditional spatial access methods (e.g., the Grid file [11] and R-tree [5]) are designed mainly for static data. Updating traditional structures is cumbersome where the update is treated as a delete followed by an insert. The claim is that updates are not frequent in traditional applications. However, in spatio-temporal databases, objects continuously send new-location updates to the index as they move.

To reduce the frequency of updates, a prediction scheme helps predict the updates for a certain period of time. Predicted updates are presented as trajectories. Four approaches have been investigated for indexing such trajectories: (1) The duality transformation (e.g., see [1, 3, 7, 12]), (2) Quad-tree-based methods (e.g., see [20]), (3) R-tree-based index structures (e.g., see [13, 14, 15, 16, 19]), and (4) B-tree-based structures [6]. However, indexing future trajectories solves only part of the update problem. Two main drawbacks still remain: (1) The ability of prediction is controlled by the prior knowledge and/or assumptions of the object velocity, which is not always available. (2) In many cases when the prediction scheme fails (e.g., moving freely in a downtown area or pedestrian movement), frequent updates would suffer from the same drawbacks as in traditional data structures.

The insufficiency of indexing moving objects by their future trajectories motivates the need for special data structures. The Lazy-update R-tree (LUR-tree) [8] modifies the original R-tree structure to support frequent updates. The main idea is that if an update to a certain object p would result in a deletion followed by an insertion in a new R-tree node, it would be better to increase slightly the size of the minimum boundary rectangle of the R-tree node which p lies in to accommodate its new location. The Frequently Updated R-tree (FUR-tree) [9] extends the LUR-tree by performing a bottom-up approach in which a certain moving object can move to one of its siblings. Both the LUR-tree and the FUR-tree rely on an auxiliary index to locate the old locations of moving objects. One of the key features of our proposed LUGrid is that we eliminate the use of such auxiliary disk indexes.

The difficulties in dealing with tree-based structures motivate the use of simpler data structures (e.g., hash-based and grid-based data structures) that are updated easily. A hash-based structure is used in [17, 18] where the space is partitioned into a set of overlapped zones. An update is processed only if an object moves out of its zone. SETI [2] is a logical index structure that divides the space into non-overlapped zones. Both SETI and hash-based structures ignore deleting the old location of a moving object. Thus, an update is reduced to only an insertion where past trajectories are maintained. Grid-based structures have been used to maintain only the current locations of moving objects (e.g., see [4, 10, 24]). However, two drawbacks can be distinguished: (1) The used grid is fixed, so it is not suitable in the case of a non-uniform distribution of data. One property of the proposed LUGrid is adapting to data distribution through its underlying grid file basis. (2) In many cases, the old location can be in a grid cell that is different from the one containing the new location. In this case, an extra search and extra I/Os are needed to clean up the old entry. Our proposed LUGrid efficiently resolves the issue of deletion, where a delete is performed *lazily*. Thus, no overhead I/O is incurred due to deletion.

In our recent work [22], we use an *Update Memo* to reduce the update cost. The main idea is to avoid immediate deletion of obsolete entries by maintaining a memo structure in main memory. [22] only focuses on R-tree-based indexes. Motivated by [22], in this paper, we explore similar ideas in the context of adaptive grid-based indexes to achieve *lazy deletion*. Furthermore, by utilizing *lazy insertion* along with *lazy deletion*, the update performance is significantly enhanced.

3 LUGrid: Lazy Update Grid-based Index

LUGrid adopts a grid structure that is similar to the *Grid file* [11]. In LUGrid, however, the directory of grid cells is maintained in memory instead of being stored on disk. Furthermore, the grid directory is extended to buffer object updates. We refer to the extended in-memory directory as the *Memory Grid*, and refer to the set of in-disk bucket pages as the *Disk Grid*. Additionally, a hashing-based structure termed the *Miss-Deletion Memo* is maintained to identify obsolete entries.

3.1 LUGrid Indexing Structure

Disk Grid (DG)

The *Disk Grid* (DG, for short) consists of a set of non-overlapped disk-based grid cells. A DG cell has the format $(N_E, E_1, \cdots, E_n)$ $(n > 0)$, where N_E is the number of objects stored in the DG cell. E_1 to E_n are object entries, each of them stores an object identifier along with its corresponding location.

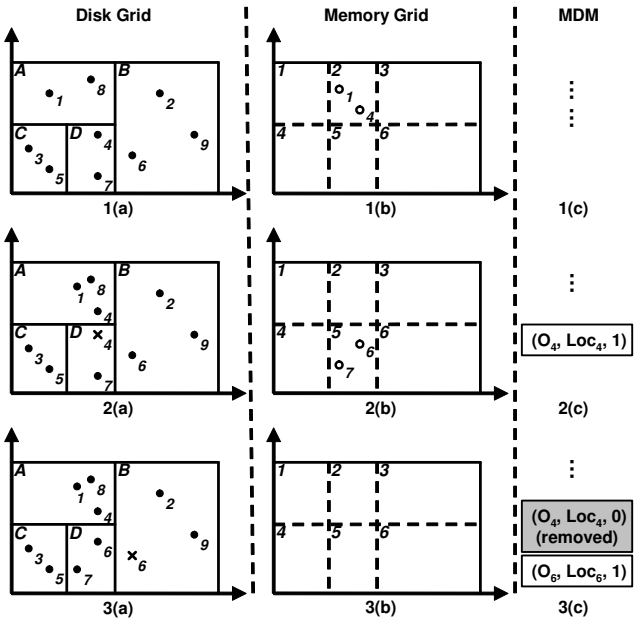


Figure 1. Example: Buffering and Flushing

Memory Grid (MG)

The *Memory Grid* (MG, for short) consists of a set of non-overlapped memory-based grid cells. Each MG cell has a limited amount of memory to buffer object updates. Each MG cell points to a DG cell where its flushed data is stored. For an MG cell m and its corresponding DG cell d, we refer to d as the *repository cell* of m, and refer to m as the *buffer cell* of d. To avoid under-utilized disk pages, several neighbored MG cells may point to a common repository cell. However, one MG cell always has exactly one repository cell.

An MG cell has the form of (N_u, M_{Region}, D_{id}, N_E, D_{Region}, E_1, $\cdots$, E_m) ($m > 0$), where N_u is the number of updates buffered in this MG cell, M_{Region} is the space region covered by this MG cell, D_{id} is the disk page identifier of the repository cell, N_E is the total number of object entries stored in the repository cell, D_{Region} is the space region covered by the repository cell, and E_1 to E_m are object entries storing an object identifier along with its new location.

Miss-Deletion Memo (MDM)

In LUGrid, old object entries may co-exist with current entries since the deletion of old entries is delayed. The *Miss-Deletion Memo* (MDM, for short) is employed to distinguish obsolete entries from current entries. MDM is an in-memory hash-based table, it uses a counter to track the number of deletions that each object has missed. An MDM entry has the form $(OID, OLoc, MDnum)$, where OID is the object identifier, $OLoc$ is the most recent object location that has been flushed to DG, and $MDnum$ is the number of missed deletions for the object OID. As an example, an MDM entry (O_{12}, (34, 64), 1) is interpreted as that the object with identifier O_{12} has missed the deletion of old entry for 1 time (i.e., there is 1 entry of O_{12} in DG that is obsolete but that has not been deleted yet), and the newest location of O_{12} is (34, 64). For one MDM entry, if $MDnum$ changes to 0, which means all obsolete entries for the object OID have been deleted, the MDM entry can be safely removed from the MDM to reduce the memory usage.

Example 1. We use the example given in Figure 1 to illustrate our ideas. Figure 1.1(a) gives a DG structure with the four DG cells A, B, C and D. Nine objects o_1 to o_9 are stored in DG. Figure 1.1(b) gives the MG structure that is partitioned into six cells, 1 to 6. MG cells 1 and 2 have the same repository cell (DG cell A), while MG cells 3 and 6 have the same repository cell (DG cell B). Assume at this moment, there is no obsolete entry that exists on disk. Thus MDM, given in Figure 1.1(c), is empty.

3.2 Processing Updates

Update processing in LUGrid is performed in three stages.

Stage I: Buffering updates. Initially, continuously received updates are buffered in MG.

Stage II: Flushing updates into disk. Flushing buffered updates into disk cells is triggered when an in-memory grid cell C_M is full. In this case, C_M is flushed into its corresponding repository disk grid cell.

Stage III: Splitting/Merging cells. If a DG cell is overfull or under-utilized, cell splitting/merging takes place in both memory grid and disk grid.

Buffering updates. Figure 2 gives the pseudo code for buffering incoming updates in MG. For a certain MG update entry u, we denote the MG cell containing u as $MGC(u)$. Further, we say that u *is consumed* if u is flushed to disk.

Since it may happen that one update arrives to the server while the previous update for the same object has not been consumed, the buffering algorithm starts by searching MG for the entry with the same object identifier (OID). The search is performed through an OID hash link that links all updates in MG based on their OIDs. If an entry with the same OID is found, the found entry is deleted from MG.

After the deletion of the unconsumed update for the same object, the new update is inserted into the MG cell whose region covers the new location. The update is also linked in the OID hash link to facilitate future search. If the MG cell where the object update is inserted becomes full after the insertion, the flushing stage is invoked to flush all buffered updates in this MG cell to its repository cell.

Procedure BufferingUpdate(UpdateTuple u(oid, loc))

1. *Search $u.oid$ in MG by exploring the OID hash link in MG. If an MG entry m where $m.OID$ equals to $u.oid$ is found*
   (a) *Delete m from MG;*
   (b) *MGC(m).N_u−−; usedSlots−−;*
2. *Insert u into the MG cell whose M_{Region} covers $u.loc$;*
3. *MGC(u).N_u++; usedSlots++;*
4. *Link u in MG's OID hash link based on $u.oid$;*
5. *If (MGC(u).N_u >= MaxUpdPerMGCell)*
   (a) *Call FlushingUpdates(MGC(u));*

Figure 2. Buffering Object Updates

Procedure FlushingUpdates(MGCell mc)

1. *dc = the repository cell of mc; Read dc into memory;*
2. *For each entry d in dc, if an MDM entry e where $e.OID$ equals to $d.OID$ is found*
   (a) *If (d.OLoc ≠ e.Oloc)*
      i. *Delete d from dc; $dc.N_E$−−; $e.MDnum$−−;*
         A. *If (e.MDnum == 0) delete e from MDM;*
3. *For each entry m in mc*
   (a) *If a DG entry d_{old} in dc where $d_{old}.OID$ equals to $m.OID$ is found*
      i. *$d_{old}.Oloc$ = $m.Oloc$;*
      ii. *If an MDM entry e where $e.OID$ equals to $m.OID$ is found*
         A. *$e.Oloc = m.Oloc$;*
      iii. *Delete m from mc; $mc.N_u$−−; usedSlots−−;*
   (b) *Else //if such d_{old} does not exist*
      i. *If an MDM entry e where $e.OID$ equals to $m.OID$ is found*
         A. *$e.Oloc = m.Oloc$; $e.MDnum$++;*
      ii. *Else //if such e does not exist*
         A. *Create e as a new MDM entry; $e.OID$ = $m.OID$; $e.OLoc$ = $m.OLoc$; $e.MDnum$ = 1; Insert e into MDM;*
4. *If ($mc.N_u$ + $dc.N_E$ <= MaxEntPerDGCell)*
   (a) *Move all remaining MG entries in mc to dc; $dc.N_E$ = $dc.N_E$ + $mc.N_u$; usedSlots = usedSlots - $mc.N_u$; $mc.N_u$ = 0; $mc.N_E$ = $dc.N_E$;*
   (b) *For all buffer cells of dc, set their values of N_E to $dc.N_E$;*
   (c) *Call MergingCell(mc, dc);*
5. *Else call SplittingCell(mc, dc);*

Figure 3. Flushing Buffered Updates

Flushing updates. Figure 3 gives the pseudo code for flushing updates into DG cells. First, the repository cell is read into memory. For entries in the repository cell, it is possible that some entries have become obsolete due to newer updates in other disk cells. To identify such objects, for each DG entry, the *miss-deletion memo* (MDM) is searched. If the MDM entry with the same OID exists and is associated with a location different to that of the DG entry, the DG entry is obsolete. Such obsolete disk entry is removed from the repository cell. Then, the *miss deletion number* of the MDM entry is decremented. In the case the *miss deletion number* returns to zero, the MDM entry itself is removed from MDM.

After deleting obsolete entries, each update in the MG cell searches its original entry in the repository cell. If the original entry is found, the entry is updated with new location information. In this case, if an MDM entry exists for the object, the location field of the MDM entry is updated. At the end, the update is deleted from the MG cell. Otherwise, if no original entry for the updating object is found, the original entry must reside in another DG cell and is obsolete due to the new update. In this case, if one MDM entry exists for the object, the MDM entry is updated with new location, and the *miss deletion number* is incremented by one. If no such MDM entry exists, a new MDM entry is created. The new entry is filled with the latest location and the *miss deletion number* is set to one.

After the above processing, the MG cell contains only object updates that are "new" to the repository cell. If all such updates can be added to the repository cell without causing overflowing, they are inserted into the repository cell and are removed from the MG cell. Related counters and pointers are adjusted accordingly. All buffer cells that point to this repository cell need update their counters for the number of disk entries. Then, a merging function is called to seek the opportunity of merging this DG cell with neighbored cells. Otherwise, if putting all remaining updates into the repository cell causes overflowing of the repository cell, the repository cell is split to two disk cells.

Splitting/merging cells. LUGrid inherits the splitting and merging mechanism of the grid file [11]. Special attentions are carried out to cope with our unique lazy-insertion and lazy-deletion techniques. Due to space limitation, we do not present the splitting/merging details in this paper. Please refer to our technical report [23] for detailed discussion.

Example 2. In the example given in Figure 1, we assume that an MG cell needs a flushing if two updates are buffered. In Figure 1.1(b), MG cell 2 receives updates from o_1 and o_4. Since cell 2 is full, it flushes the two updates to DG cell A. First, entries in DG cell A are checked with MDM. Since MDM is empty, both o_1 and o_8 are identified as current entries. Then, the update of o_1 finds the original

o_1 entry in DG cell A, and further updates the location of the entry. For the update of o_4, however, does not find the original entry in DG cell A. So it creates an entry in MDM to indicate that this update invalidates a former entry of o_4. The resulting DG and MDM are shown in Figure 1.2(a) and Figure 1.2(c), respectively. Note that the former entry of o_4 in DG cell D (plotted with cross mark) still remains on disk.

After some time, cell 5 receives two updates from o_6 and o_7 and starts flushing to DG cell D (Figure 1.2(b)). The entry for o_4 is identified as obsolete because the location of the o_4 entry does not equal to the location in MDM. Therefore, the obsolete entry is deleted out of cell D. Note that the MDM entry for o_4 is removed because the miss deletion number returns to zero. The update of o_7 replaces the original entry of o_7 with the new location. On the other hand, the update of o_6 does not find an original entry in cell D. Therefore, it creates an entry in MDM for o_6. The final states of DG, MG, and MDM are plotted, respectively, in Figure 1.3(a), Figure 1.3(b) and Figure 1.3(c).

3.3 Obsolete Entry Cleaning

In this section, we discuss issues related to the number of obsolete disk entries. Throughout this section, let N_{old} be the total number of old entries on disk, and let M_{ent} be the total number of MDM entries. Note that N_{old} is always larger than or equal to M_{ent}, since one MDM entry represents one or more missed deletions for a certain object.

Recall that when flushing a memory cell, obsolete entries in the repository cell are first deleted. The removal of obsolete entries reduces both N_{old} and M_{ent}, so that both N_{old} and M_{ent} are kept small. In our experiments (see Section 5), both numbers are less than 1% of the total number of objects.

To guarantee that MDM is of small size, LUGrid adopts a cleaning technique termed *cleaner* to bound the number of obsolete entries and the size of MDM. The basic task of the cleaner is to pick a DG cell and clean all old entries whenever LUGrid is updated a fixed number of times. Such a fixed number is termed the *clean interval*. The clean procedure follows the same step as we discussed in Section 3.2 (see Step 2 in Figure 3). With the cleaner, the maximum value of N_{old} and M_{ent} is limited by $(i * P)$, where i is *clean interval*, and P is the total number of DG cells.

To maximize the number of old entries deleted from a DG cell, the cleaner always picks the DG cell that has experienced the longest time since its latest flushing. Such DG cell has the potential to contain more old entries than the other cells. To identify the oldest DG cell quickly, LUGrid maintains page identifiers of all DG cells in a *Least Recently Flushed* list.

3.4 Query Processing in LUGrid

In this section, we discuss query processing in LUGrid. Query processing in LUGrid exploits both memory and disk grids. The steps for answering a query are generalized as follows: (1) Identify a set of MG and DG cells that cover all objects needed in answering the query; (2) Obtain an initial answer set by processing only the entries in the MG and DG cells from the last step; (3) Obtain the final answer set by purging obsolete entries from the initial answer set.

As the first two steps are straightforward and depend on the query type, here we focus on the last step, namely, to identify obsolete entries in the initial query answer. First, any MG entry is a current entry because, as discussed in Section 3, the newer update replaces the older one in MG. For a DG entry, if it satisfies any of the following two conditions, the entry is obsolete. (1) An MG entry for the same object is found in MG; (2) An MDM entry for the same object is found, and the location of the MDM entry does not equal to the location of the DG entry. The existence of such MDM entry indicates that a newer update has been flushed to some other DG cell. If a DG entry has not been identified as obsolete, the entry is a current entry and is put in the final answer set.

4 Cost Analysis

In this section, we analyze the update cost for the proposed techniques. Our analysis studies three cases: applying *lazy-insertion only*, applying *lazy-deletion only*, and applying *lazy-insertion plus lazy-deletion*. The analysis is based on a uniform distribution of moving objects in the two-dimensional space. As a dominating metric, the number of I/O operations is investigated for the updating cost.

Let U represent the maximum number of updates that can be buffered in MG, and let N_m represent the total number of MG cells. We define δ as the percentage of updates that a certain object moves from a certain DG cell to another one.

Lazy-insertion only. Applying only *lazy-insertion* means that incoming updates are buffered and grouped in memory cells before they are flushed to disk in batches. At every time of flushing, old disk entries of the flushed updates must be cleaned. This requires an auxiliary index on object IDs for quickly locating old object entries.

Assume that n_u is the number of buffered updates in a flushing MG cell. For *lazy-insertion only*, the total I/O cost for flushing an MG cell is given by Equation 1.

$$IO_{LI} = 2 + 4(n_u * \delta) \quad (1)$$

In Equation 1, 2 is introduced by reading and writing the repository cell only once for all n_u updates. $(n_u * \delta)$ represents the expected number of updates that have old entries

in other DG cells. For each such update, four additional I/O operations are required to delete the old entry. The operations include searching and updating

both the auxiliary index and the old DG cell. The expected number of updates in an MG cell is U/N_m. So, according to Equation 1, the *average* updating cost for one object is:

$$IO_{LI\_avg} = \frac{2N_m}{U} + 4\delta \quad (2)$$

Lazy-deletion only. Applying only *lazy-deletion* means that an object update goes to disk immediately whenever it arrives. If the old entry for the object resides in a different disk page, the old entry stays on disk until it is cleaned later. MDM hash table is used to identify old entries of objects. No secondary index is required for locating objects.

For the case of *lazy-deletion only*, the overall update I/O cost consists of the following: (1) Reading the repository DG cell to memory, (2) Writing the DG cell back to disk, and (3) If the cleaner is invoked, reading and writing the cleaned DG cell. Hence, if let c be the clean interval, the expected I/O cost per update is given by:

$$IO_{LD\_avg} = 2 + \frac{2}{c} \quad (3)$$

Lazy-insertion plus lazy-deletion. Combining *lazy-insertion* with *lazy-deletion* minimizes the updating cost. For a set of n_u updates in an MG cell, only two I/O operations and cost of periodical cleaning are required. Therefore, the average updating cost is simply given by:

$$IO_{LI\&LD\_avg} = \frac{2}{n_u} + \frac{2}{c} = \frac{2N_m}{U} + \frac{2}{c} \quad (4)$$

5 Performance Evaluation

PARAMETERS	VALUES USED
Object distribution	**Uniform**, Normal distribution
Object velocity	10, 50, **100**, 500 miles/hour
Update ratio of objects	0%, 30%, **5%/cycle**
MG buffer size	0%, **1%**, 2% of object number

Table 1. Experiment Parameters and Values

In this section, we evaluate the performance of LUGrid with various settings. LUGrid is compared with the Frequently Updated R-tree (FUR-tree, for short) [9] in both update and query processing. To make our comparison fair, we make the following two changes. (1) The first two levels of the FUR-tree are assumed to reside in memory; (2) Whenever LUGrid consumes some amount of memory, we give FUR-tree a same size buffer maintained in a *Least Recently Used* (LRU) manner. FUR-tree makes use of the buffer when accessing the R-tree nodes and the auxiliary index.

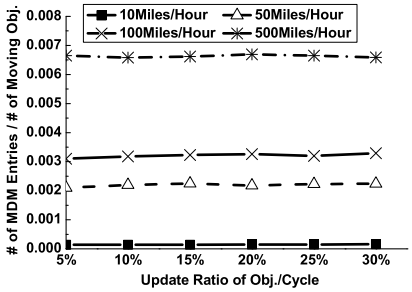


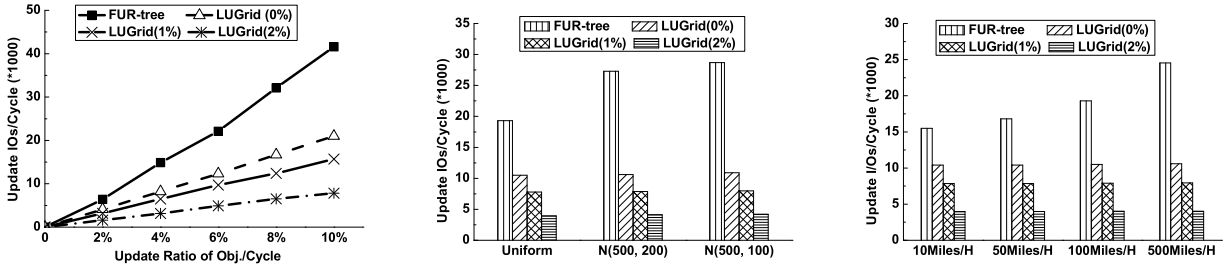
Figure 4. Size of the Miss-Deletion Memo

In all experiments, we collect the results of LUGrid when the system becomes stable. In this case, cell splitting or merging rarely happens. The number of updates that MG can buffer is an experiment parameter. In all experiments, the clean interval of the cleaner is set to 50.

All the experiments are conducted on an Intel Pentium IV machines with CPU 3.2GHz and 512MB RAM. In all experiments, 100,000 objects are moving inside a space that represents 1000 * 1000 square miles. We generate the original objects and consequent updates in a way similar to GSTD [21]. Objects continuously move with given velocities. We count the number of object updates in *cycles*. In each cycle, a certain ratio of objects report their new locations by issuing updating requests. Experiments are carried out under both uniform distribution and normal distributions. We use *Normal(μ, σ)* to denote a normal distribution with mean μ (miles) and variance σ (miles). Three types of distributions are used in our experiments, namely, *Uniform*, *Normal(500, 200)* and *Normal(500, 100)*. In our experiments, the page size is fixed as 4096 bytes. Various parameters for our experiments are outlined in table 1, the default values are given in **bold**.

5.1 Size of MDM

We first study the size of the Miss-Deletion Memo. Figure 4 gives the size of the MDM with various object velocities (i.e., 10, 50, 100, and 500 miles/hour). For each studied velocity, we increase the ratio of objects that report updates per cycle from 5% to 30%. The size of MDM is expressed as the ratio of the number of MDM entries over the total number of objects. When the object velocity increases, the size of MDM becomes larger. The main reason is that when objects move with a higher velocity, more objects will move out of their original cells. Consequently, more MDM entries are needed to track information of these cell-changing objects. However in all cases, the number of MDM entries



(a) Upd. cost vs. # of upd.

(b) Upd. cost vs. obj. dist.

(c) Upd. cost vs. obj. vel.

Figure 5. Update Performance

is less than 0.7% of the total number of objects. On the other hand, the update ratio of objects does not affect the size of MDM. The reason is that the size of MDM is determined only by the number of cell-changing objects in one MG flushing. The number is not affected by the update ratio. The experiment demonstrates that the size of MDM is small in practice and hence can fit in main memory.

5.2 Update Performance

In this section, we study the update performance of LUGrid and FUR-tree. For LUGrid, we use different sizes of MG buffers. Specifically, the MG buffer is set as 0%, 1% and 2% of the indexed objects. A 0% size buffer represents a *lazy-deletion only* scenario, as discussed in Section 4. Figure 5(a) plots the number of I/Os when the ratio of objects that report updates increases from 0 to 10% per cycle. For different update ratios, LUGrid outperforms the FUR-tree consistently. The update costs for LUGrid range from 20% to 50% of that for the FUR-tree. The efficiency in updates in LUGrid with 0% size buffer comes solely from the lazy-deletion technique. When the MG buffer becomes larger, the update cost becomes lower because more updates are flushed to disk at one time by lazy-insertion.

Figure 5(b) compares the update costs under various object distributions. LUGrid exhibits almost stable update performance independent of object distributions. This is because the update cost of LUGrid is determined primarily by the flushing frequency. Object distribution does not dramatically affect the flushing frequency. However, the FUR-tree incurs larger update cost when object distribution is skewed. The main reason is that when more objects are clustered together, the R-tree contains more nodes with small *Minimal Bounding Rectangles* (MBRs). Therefore, an object is more likely to move out of its MBR quickly and invalidates the bottom-up updating technique.

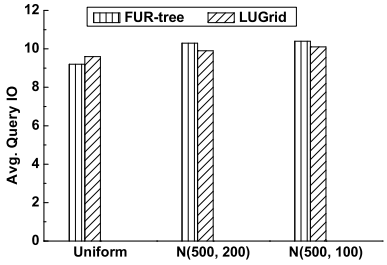


Figure 6. Query cost vs. obj. distribution

Figure 5(c) demonstrates the effect of object velocity, where objects are moving with various velocities (10, 50, 100 and 500 miles/hour). As shown in Figure 5(c), when object velocity increases, the FUR-tree incurs a growing I/O overhead due to updates. This is because with a larger velocity, an object moves out of the MBR of its original node more frequently, and voids the endeavor of the bottom-up update. In contrast, for LUGrid, the I/O from updates is not affected by object velocities. This is because LUGrid does not delete old entries when updating, so objects moving out of their original cells do not affect the performance.

5.3 Query Performance

In this section, we study the query performance of LUGrid. We focus on the processing of range queries as it is one of the most important types of spatial queries. In our experiments, queries are specified as squares and are uniformly distributed in space. Figure 6 compares the querying costs with respect to object distributions. In this experiment,

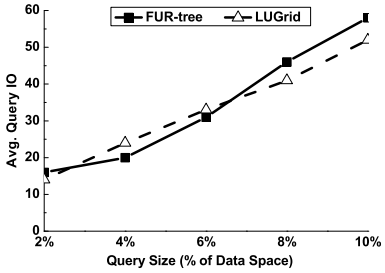


Figure 7. Query cost vs. query size

each query covers 1% of the whole space. The experiment shows that under all object distributions, LUGrid is similar to FUR-tree. Both FUR-tree and LUGrid are slightly affected by object distribution.

Figure 7 gives the effect when different sizes of queries are issued. We increase the query size from 2% to 10% in terms of the percentage of the whole space. Both FUR-tree and LUGrid have almost linear increase on querying costs. Again, the performance of LUGrid is similar to the performance of FUR-tree in all cases. These experiments show that LUGrid achieves similar range search performance as the FUR-tree.

6 Conclusion

In this paper, we proposed LUGrid; an adaptive *Lazy-Update Grid-based* indexing structure. LUGrid efficiently handles object updates by its unique *lazy-update* features. Lazy-deletion converts the update cost from traditional "insertion cost plus deletion cost" to "insertion cost only". The lazy-deletion functionality is provided by maintaining a *memo* structure to identify obsolete entries. Lazy-insertion groups updates and flushes multiple updates at one time, thus amortize the cost for single update. We believe that the proposed lazy-update schemes can be applied to other index families.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *PODS*, pages 175–186, May 2000.

[2] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, 2003.

[3] H. D. Chon, D. Agrawal, and A. E. Abbadi. Storage and Retrieval of Moving Objects. In *Mobile Data Management*, pages 173–184, Jan. 2001.

[4] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.

[5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[6] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.

[7] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *PODS*, 1999.

[8] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.

[9] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[10] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.

[11] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1), 1984.

[12] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD*, pages 637–646, 2004.

[13] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *SSTD*, pages 59–78, Redondo Beach, CA, July 2001.

[14] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[15] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[17] Z. Song and N. Roussopoulos. Hashing Moving Objects. In *Mobile Data Management*, 2001.

[18] Z. Song and N. Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management, MDM*, pages 340–344, Jan. 2003.

[19] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.

[20] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3), 1998.

[21] Y. Theodoridis, J. R. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *SSD*, 1999.

[22] X. Xiong and W. G. Aref. R-trees with Updated Memos. In *ICDE*, 2006.

[23] X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: Update-tolerant Grid-based Indexing for Moving Objects. *Purdue University Department of Computer Sciences Technical Report, No. CSD TR 05-022*, 2005.

[24] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, 2005.