2005

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

Xiaopeng Xiong

Mohamed F. Mokbel

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Report Number:
05-022

LUGRID: UPDATE-TOLERANT GRID-BASED INDEXING FOR MOVING OBJECTS

Xiaopeng Xiong
Mohamed F. Mokbel
Walid G. Aref

CSD TR #05-022
October 2005

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. Therefore, a batch of updates can be flushed to disk at one time, and consequently the cost of multiple updates is amortized.* Lazy deletion *reduces update cost by avoiding deleting single obsolete entry out of the index immediately. Instead, the obsolete entries are removed later by specially designed mechanisms. LUGrid adapts to object distributions through cell splitting and merging. In the paper, we extensively discuss the structure of LUGrid and the algorithms for update and query processing. Moreover, we provide theoretical analysis for estimating the update cost of LUGrid. Comprehensive experimental results indicate that LUGrid outperforms former work 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.

Usually, the number of moving objects tends to be huge, and the server needs to build indexes on the current locations of objects to accelerate the processing of standing queries. However, most of existing spatial indexes (e.g., R-tree-like indexes) are designed for static data and exhibit poor performance under frequent updates. Recently, some new techniques (e.g., see Section 2) have been proposed to alleviate the situation. However, as we demonstrate in Section 6, these techniques are still not efficient enough for quickly handling large amounts of updates in a short period of time. The problem of indexing continuously moving objects is far from being resolved.

Existing approaches on indexing moving objects suffer from large volumes of updates. The reasons are observed from the following three aspects. First, most of the indexing approaches process single updates independently. Since every call for index updating is expensive, processing single update one at a time hinders largely the scalability of the index. Second, when an update arrives, existing approaches try to remove the old entry for the object. If the old entry resides on a disk page different from the page where the new entry is to be stored, additional disk I/Os are required to purge the old entry from the index. In many cases, eliminating old entries is more costly than inserting new entries. Third, to quickly locate the old entry for an object, many index structures maintain a secondary index on object IDs (e.g., [8, 9, 11, 24]). Maintaining a secondary index is expensive in itself. The secondary index has to be updated every time an object changes its locality of disk page. Further, for each update, at least one page of the secondary index is searched in order to locate the old entry of the updating object, which adds more burden to the updating process.

In this paper, we propose *LUGrid*, an adaptive *Lazy-Update Grid-based index* for indexing current locations of moving objects. LUGrid aims to avoid the above mentioned drawbacks of existing indexing techniques. LUGrid minimizes the I/O costs for updates by adopting the concept of *lazy-update*. LUGrid is designed with two important features: (1) *Lazy-insertion*. In LUGrid, object updates going to a same disk page are grouped together and are 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 updating is kept very low. (2) *Lazy-*

deletion. In contrast to other indexing approaches, LUGrid does not require deleting old entries before inserting updated entries. Instead, LUGrid delays the deletion process until the disk pages where the old entries reside are retrieved into memory. Therefore, I/O cost to search and delete old entries from disk is saved. This is achieved by a memory-resident data structure, namely, the "miss-deletion memo" (MDM). MDM is a hash-based data structure that maintains ONLY those objects that "miss" at least one deletion. LUGrid guarantees that the size of MDM is upper-bounded to a small size so that it can be easily accommodated in main-memory.

LUGrid adapts to arbitrary object distributions through its adaptive grid structure that is borrowed from the *Grid file* [12]. Queries on LUGrid are answered by accessing on-disk entries as well as in-memory buffered object updates. Query answers are ensured that no obsolete entries are included and that no current entries are overlooked. We demonstrate that, under various object distributions, the updating performance of LUGrid is 2 to 8 times superior to former indexing approaches. Meanwhile, LUGrid maintains efficient querying performance when compared to former approaches.

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 extensively discuss the structure of LUGrid and algorithms for update and query processing. We analyze the update cost of LUGrid theoretically.

3. We provide a comprehensive set of experiments demonstrating that LUGrid outperforms largely former work in update processing while maintaining similar querying performance.

The rest of this paper is organized as follows. Section 2 highlights the related work in the literature. The proposed LUGrid is discussed in Section 3. Query processing in LUGrid is addressed in Section 4. Section 5 analyzes the update cost of the proposed *update* scheme. An extensive set of experiments that evaluates the performance of LUGrid is given in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

Traditional spatial access methods (e.g., the Grid file [12] and R-tree [5]) are designed mainly to support query processing. Updating traditional structures is cumbersome where it is considered as a delete operation followed by an insert operation. The claim is that updates are not frequent in traditional applications. However, in spatio-temporal databases, objects continuously send location updates to the index structure as they move. For the past decade, several research efforts focus on developing variations of traditional access methods to support continuously moving objects (e.g., see [10] for a survey).

In an attempt to reduce the frequency of updates to index structures, a prediction scheme helps predict the updates for a certain period of time in the future. Predicted future updates are presented as trajectories. Thus, indexing continuous moving objects is reduced to indexing future trajectories. Four approaches have been investigated for indexing future trajectories: (1) Duality transformation (e.g., see [1, 3, 7, 13]). The main idea is to map the future trajectory to a single point in another domain, then a duality transformation is used to transfer all queries to the new space, (2) Quad-tree-based methods (e.g., see [21]), (3) R-tree-based index structures (e.g., see [14, 15, 16, 17, 20]), and (4) B-tree-based structures [6]. However, indexing future trajectories solves only part of the updating 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) It is implicitly assumed that the updates of future trajectories are much less than the updates to the object location. However, this is not always true where in many cases the prediction scheme fails (e.g., moving freely in a downtown area or pedestrian movement). Frequent updates to the prediction scheme would suffer from the same drawbacks of frequent updates in traditional data structures.

The inefficiency of indexing moving objects by their future trajectories motivates the need for special data structures that are suitable for frequent updates. 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 if we can increase slightly the size of the minimum boundary rectangle of the R-tree node in 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 instead of having deletion followed by an insertion. Both the LUR-tree and the FUR-tree use an auxiliary structure to index objects based on their identifiers. These auxiliary indices locate the old locations of moving objects. One of the key features of our proposed data structure LUGrid is that we eliminate the use of such auxiliary disk indexes since in our proposed scheme the old location is *lazily* visited and deleted.

The difficulties in dealing with tree-based structures and the complexity of dual transformations 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 [18, 19] 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, 11, 24]). However, two drawbacks can be distinguished: (1) The used grid is fixed where it is just a regular partitioning of space into equal sized non-overlapped zones. This approach is not suitable in the case of a non-uniform distribution of data, (2) Deleting an old location of a certain object is still cumbersome, where 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.

In our recent work [23], we initially proposed making use of 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. [23] only works for R-tree-based indexes. In this paper, we explore similar idea in adaptive Grid-based indexes to achieve *lazy deletion*. Furthermore, by utilizing *lazy insertion* along with *lazy deletion*, the update performance is significantly enhanced.

Our proposed LUGrid structure distinguishes itself from all other approaches where it has all the following properties: (1) LUGrid indexes the current positions of moving objects, no predication scheme is used, (2) LUGrid is based on the *Grid file* [12] where grid cells are not equal sized, cells can adapt to data distribution through cell splitting and merging. (3) LUGrid efficiently resolves the issue of deletion, where a delete is performed *lazily*. Thus, no overhead or I/O is incurred due to deletion.

3 LUGrid: Lazy Update Grid-based Index

In this section, we propose *LUGrid*, an adaptive grid-based index structure that efficiently handles the continuous updates of objects' locations. LUGrid exploits two techniques, namely, *lazy-insertion* and *lazy-deletion*. In *lazy-insertion*, incoming updates are grouped together based on the updated disk-page and are *lazily* flushed into disk once. Thus, multiple updates are reduced to only a single disk update. In *lazy-deletion*, obsolete entries (i.e., entries that receive an update) remain in disk rather than being immediately deleted. By keeping necessary *memo* information, we can *lazily* remove the obsolete entries only when their disk pages are accessed, e.g., via an insertion. Thus, a *delete* operation does not incur any I/O overhead. *lazy-insertion* and *lazy-deletion* can be used either independently or together to boost the performance of frequent updates in traditional index structures.

3.1 LUGrid Indexing Structure

LUGrid adopts a grid structure that is similar to the *Grid file* [12]. In LUGrid, however, the directory of grid cells is maintained in memory instead of being stored on disk. Also, we extend the grid directory 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. These three structures act together to maintain continuous object updates in LUGrid.

Disk Grid (DG)

The *Disk Grid* (DG, for short) consists of a set of non-overlapped disk-based grid cells. Each grid cell is stored in one disk page. A DG cell stores information of objects that lie within the cell boundaries. Each DG cell covers an exclusive portion of the data space that is determined by its corresponding *Memory Grid cell(s)*. A DG cell C_D has the format $(N_E, E_1, \cdots, E_n)$ $(n > 0)$, where N_E is the number of object entries stored in C_D. E_1 to E_n are the stored objects in C_D. An object entry E_i has the form of $(OID, OLoc)$, where OID is the object identifier, and $OLoc$ is the latest object location that has been *flushed* to C_D. Since a DG cell corresponds to a disk page, for the rest of the paper, we use the terms *disk page* and *DG cell* as synonyms.

Memory Grid (MG)

The *Memory Grid* (MG, for short) is an in-memory two-dimensional array, where each element of the array is a *Memory Grid cell*. 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 underutilizing disk pages, several neighbored MG cells may have the same repository cell given that the united space region of these MG cells forms a rectangle. However, in any case, one MG cell can have exactly one repository cell. The space coverage of a DG cell is the united space region of all its buffer cells.

Each MG cell has a limited amount of memory that can buffer object updates temporarily. Object updates are *double-hashed* in MG. First, one update is inserted as a new update entry to an MG cell whose space region covers the new object location. Meanwhile, the same update entry is linked in a hash link based on the object identifier.

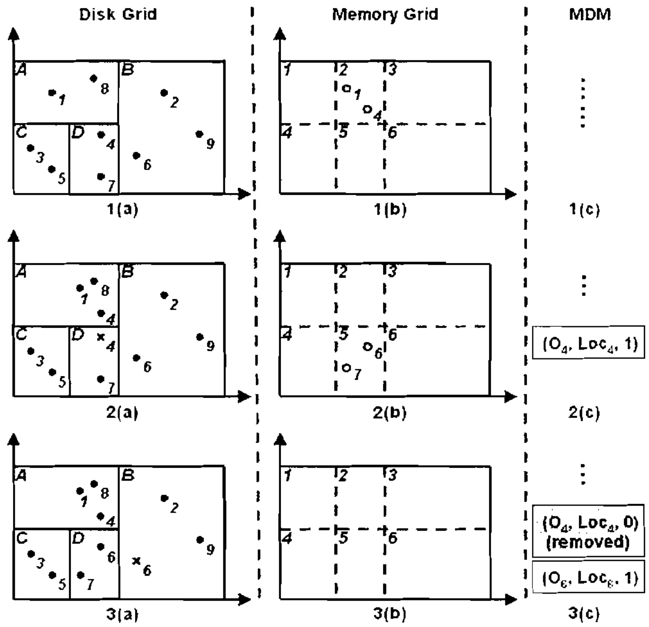

Figure 1. Example: Buffering and Flushing

By double-hashing, an object update in MG can be quickly reached either by its new location or by its identifier.

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 buffered updates 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 the object updates buffered in this MG cell. An object update entry has the form of $(OID, OLoc)$, where OID is the object identifier, and $OLoc$ is the latest *received* location for the object.

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 that keeps track of those objects that miss at least one deletion. In addition, it keeps a counter with the number of deletions that each object 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 *miss deletion number* 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} in DG is (34, 64). Note that 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.

A running example. 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

In this section, we discuss update processing in LUGrid. An update sent from a continuously moving object to the LUGrid contains the object identifier and the object's new location. Figure 2 depicts an overview of update processing in LUGrid that has the following three stages:

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

Stage II: Flushing updates into disk. Flushing in-memory data into disk pages is triggered by any of the following two events: (1) An in-memory grid cell C_M is full. In this case, C_M is flushed into its corresponding repository disk-based grid cell, (2) The overall memory becomes full. In this case, a certain cell is chosen as a victim and is flushed into its corresponding DG repository cell. Notice that it may be the case that the whole memory is full while none of the in-memory cells are full. This is due to the fact that we use two different thresholds, one for the maximum number of updates that can be buffered in each cell, and the second is for the maximum number of updates that can be buffered in the whole memory. The reason behind this is to allow for more efficient buffering capabilities. The process of flushing an in-memory cell to disk needs a special coordination among the three used data structures, DG, MG, and MDM.

Stage III: Splitting/Merging cells. Finally, if a DG cell is over-full or is under-utilized, cell splitting or merging takes place in both in-memory and disk-based grid structures.

In this section, we discuss the first two stages. The third stage is briefly discussed in Section 3.3.

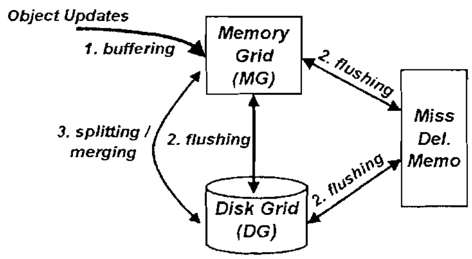

Figure 2. Overview of LUGrid

Buffering updates. Figure 3 gives the pseudo code of processing incoming updates in LUGrid. Once an object update is received, the update is buffered in MG immediately. For a certain MG update entry u, we denote the MG cell that contains 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). If an entry with the same OID is found, the found entry is deleted from MG. The reason is that the existing in-memory entry becomes obsolete, thus, is no longer needed (Step 1 in Figure 3). Notice that for a single update, at most one entry with the same OID may exist in MG, because earlier entries with the same OID are either consumed to disk or are deleted from memory due to a newer update.

After the deletion of one unconsumed update for the same object, the new update is inserted into the MG cell whose region covers the new location (Steps 2 and 3 in Figure 3). Recall that the updates in MG are organized in double-hashing fashion, so the update is also inserted in a hash link according to the object ID (Step 4 in Figure 3). If the MG cell where the object update is inserted becomes full after the insertion, LUGrid flushes all buffered updates in this MG cell to its repository cell (Step 5 in Figure 3). If the total number of buffered updates in all MG cells exceeds the maximum limit, LUGrid picks the MG cell that has the largest number of buffered updates and flushes the updates to its own repository cell (Step 6 in Figure 3). In both cases, the flushing function given in Figure 4 is called.

Flushing updates. Figure 4 gives the pseudo code for flushing updates into DG cells. The flushing algorithm *consumes* the buffered updates in an in-memory grid cell (MG) by flushing them to the corresponding repository cell. First, the repository cell is read into memory (Step 1 in Figure 4). 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 entry in the DG cell, the *miss-deletion memo* (MDM) is searched for the entry with the same OID. If an MDM entry with the same identifier is found, and the location stored in MDM

Procedure onReceivingUpdate(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));*
6. *If(usedSlots >= MaxSlots)*
   (a) *mc_{max} = the MG cell that buffers the largest number of updates;*
   (b) *Call FlushingUpdates(mc_{max});;*

Figure 3. On Receiving Object Update

does not correspond to that of the disk entry, the disk entry is considered obsolete. In this case, the obsolete disk entry is removed from the repository cell (Step 2 in Figure 4), and the *miss deletion number* of the MDM entry is decremented by one. In the case that the MDM entry indicates all old entries on disk have been deleted for this object (i.e., the *miss deletion number* equals 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. The original entry may or may not exist. 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 needs to be updated. At the end, the update is deleted from the MG cell (Step 3a in Figure 4). Otherwise, if no original entry for the updating object is found, then the original entry must reside on another DG cell and is obsolete due to the new update (Step 3b in Figure 4). In this case, if one MDM entry exists for the object, the MDM entry is updated with new location information, and the *miss deletion number* is incremented by one (Step 3b(i) in Figure 4). If no such MDM entry exists, a new MDM entry is created. The new entry is filled with the latest information and the *miss deletion number* is set to one (Step 3b(ii) in Figure 4).

Following 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, and related counters are changed accordingly (Step 4a in Figure 4). All buffer cells that point to this repository cell need update

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 $\neq$ 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 4. Flushing Buffered Updates

their counters for the number of disk entries (Step 4b in Figure 4). Then, a merging function is called to seek the opportunity of merging this DG cell with neighbored cells (Step 4c in Figure 4). 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 (Step 5 in Figure 4).

Example. In the example given in Figure 1, we assume that an MG cell needs flushing when the number of buffered updates in this MG cell reaches two. In Figure 1.1(b), MG cell 2 receives the location updates from objects 1 and 4. Since the buffer of cell 2 is full, cell 2 flushes the two updates to its repository cell which is DG cell A. Following the flushing algorithm, entries in DG cell A check with MDM to see whether there are obsolete entries. In our example, 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. Moreover, the update does not find an entry in MDM for o_4. So it creates an entry in MDM for o_4 to indicate that this update invalidates a former entry of o_4. Finally, this update is added into DG cell A. 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 becomes obsolete after the above processing, however, the obsolete entry (plotted with cross mark) still remains on disk.

Assume that after some time, the MG cell 5 receives two updates from object 6 and 7 and starts flushing to DG cell D (Figure 1.2(b)). By comparing entries in DG cell D with MDM, 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 should be removed out of MDM because all obsolete entries of o_4 have been cleaned (the miss deletion number returns to zero). After that, 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, the update of o_6 creates an entry in MDM and adds o_6 to cell D. 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 Splitting and Merging Cells

To cope with overflowed and under-utilized grid cells, LUGrid utilizes the splitting and merging utilities that are inherited from the original Grid file [12]. In this section, we discuss briefly the splitting and merging scenarios in LUGrid.

Cell splitting. In LUGrid, splitting always happens when an MG cell M is flushed to its DG repository cell D. Let the set of object entries in M be S_M, and let the set of object entries in D be S_D. Further, let the union of S_M and S_D be S_{M+D}. Let the maximum number of entries that a DG cell can contain be N_{max}. When splitting happens, the number of entries in S_{M+D} is greater than N_{max}. Thus, D splits to two new DG cells and re-distributes the entries in S_{M+D} to the two new DG cells. We refer to the two new DG cells as D_1 and D_2, respectively.

There are two possibilities for splitting cell D: (1) D is split without affecting MG; (2) Both D and MG are split. First, the splitting process tries to split only D without affecting MG. In this phase, the splitting process collects the information of D's *buffer cells* from MG (In addition to M, there may be other MG cells that map to the same

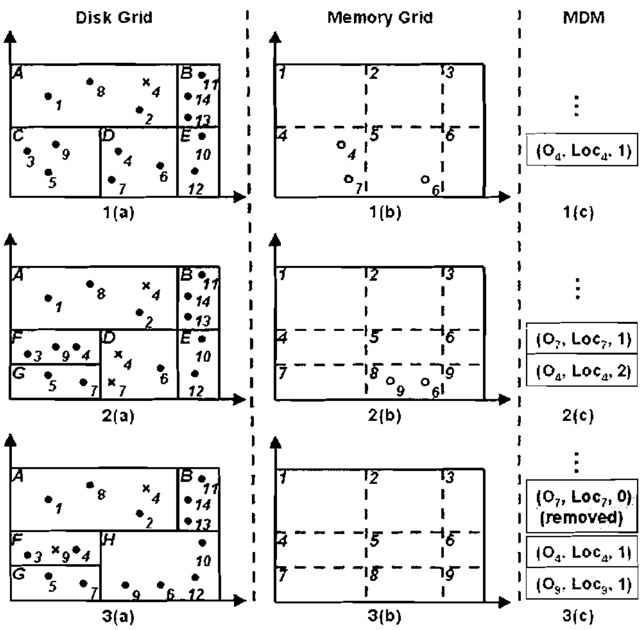

Figure 5. Example: Cell Merging and Splitting

DG cell D). The process tries to partition the buffer cells into two sets that satisfy the following two conditions: (A) The united space coverage for the cells in each set forms a rectangular box; (B) When we re-distribute the entries in S_{M+D} to these two sets based on their space coverage, the number of entries in each set is less than N_{max}. In the case that there are multiple ways to do the partitioning that satisfy the above conditions, the process chooses the partitions where the numbers of re-distributed entries have the least difference. Then, D_1 and D_2 are created. each of them serves as a repository cell for one set of the buffer cells that has been obtained from the last step. Entries in S_{M+D} are moved to D_1 and D_2 accordingly.

If the buffer cells of D cannot be partitioned into two sets based on the above conditions, D is split either horizontally or vertically. The split position lies at the median object along the split dimension. In this case, since one MG cell may have exactly one DG repository cell, MG needs splitting at the same split position. All MG cells that overlaps the splitting line are split. MG splitting will result in moving buffered updates from the original MG cells to the split MG cells, according to the space coverage of new MG cells.

Cell merging. Two neighbor DG cells may merge into one DG cell given the resulting DG cell does not overflow. LUGrid adopts the merging scheme of the *neighbor system* as used in the Grid file [12]. In the neighbor system scheme, each DG cell can merge with either of its two adjacent neighbors in each dimension given that the united space of the two DG cells forms a rectangular box.

To identify the opportunity of merging, only MG is referenced and no disk information is needed, since MG contains the necessary disk information for merging, e.g., the number of entries in the DG repository cell. A successful merging results in a new DG cell that covers the united space coverage of the two original DG cells. After the merging, all *buffer cells* of the two original cells need to point to the new DG cell as their new repository cell, and adjust their information fields accordingly.

Example. We present the example in Figure 5 to illustrate the process of cell splitting and merging in LUGrid. Figure 5.1(a) shows a DG with the five DG cells, A to E. Fourteen objects (o_1 - o_{14}) are stored in the DG, while an obsolete entry for o_4 exists in DG cell A. Figure 5.1(b) gives the corresponding MG that consists of six MG cells (cell 1 - cell 6). The MDM structure contains an entry for o_4 as shown in Figure 5.1(c). Assume that a DG cell can contain at most four entries, and an MG cell flushes updates once it receives two updates. Figure 5.1(b) shows three object updates that are buffered in MG. Since the number of updates in MG cell 4 reaches 2, MG cell 4 flushes updates to DG cell C. During the flushing process, DG cell C overflows as it now contains five entries (o_3, o_4, o_5, o_7, and o_9). Since only MG cell 4 is the buffer cell of C, it is not possible to split cell C without splitting MG. Then cell C splits to two new DG cells (cell F and cell G), and MG is split at the same splitting position. The states of DG, MG and MDM after the splitting are plotted in Figure 5.2(a), 2(b) and 2(c), respectively. During the splitting, the update for o_6 that was buffered in MG cell 5 needs to move to the new MG cell 8. Note that the *miss deletion number* of the MDM entry for o_4 becomes 2 due to the two obsolete entries for o_4 on disk (in DG cell A and D, respectively).

If after the above processing, MG cell 8 receives another update from o_9 (in Figure 5.2(b)), MG cell 8 starts to flush updates to DG cell D. After the flushing, DG cell D seeks the opportunity to merge with neighbor DG cells. DG cell D merges with DG cell E and produces a new DG cell H. The final states of DG, MG and MDM are given in Figure 5.3(a), 3(b) and 3(c), respectively.

3.4 Obsolete Entry Cleaning

In this section, we discuss issues related to the number of obsolete disk entries (due to lazy deletion) and the size of the MDM in LUGrid. 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. N_{old} and M_{ent} are related to each other, and 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 en-

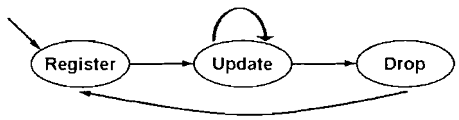

Figure 6. Life Cycle Of Object

tries in the repository cell are first deleted. The removal of obsolete entries reduces both N_{old} and M_{ent}. Such removal process is executed for each flushing cell, so that both N_{old} and M_{ent} are kept small. In our experiments (see Section 6), both numbers are less than 1% of the total number of objects.

Under some unusual situations, however, N_{old} and M_{ent} may grow large. Imagine a scenario where most objects do not update their locations except for a set of objects that traverse the space simultaneously and follow the same route. In such scenario, N_{old} is continuously growing and M_{ent} never decreases. As an enhancement for robustness, LUGrid adopts a cleaning technique termed *cleaner* to bound the number of obsolete entries and the size of MDM.

The cleaner adopts the similar technique as in [23]. The basic task for the cleaner is to pick a DG cell and clean all old entries whenever LUGrid accumulates a fixed number of updates. Such a fixed number is termed as *clean interval*. The clean procedure follows the similar steps as we discussed in Section 3.2 (see Step 2 in Figure 4). With the cleaner, it is easy to prove that the maximum value of N_{old} is given 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. In other words, the *oldest* DG cell with respect to the latest flushing time is selected for cleaning. Such DG cell has the potential to contain more old entries than the other cells. To identify the oldest DG cell quickly, the cleaner maintains page identifiers of all DG cells in a *Least Recently Flushed* buffer. The flush operation causes the identifier of the flushed cell to move to the end of the buffer. The split or merge operation results in adding or deleting an identifier from the buffer. When the cleaning process is invoked, the cleaner picks the first page identifier in the buffer and cleans old entries in the corresponding page.

3.5 Registering and Dropping Objects

In practice, an object registers itself into the system, sends a series of location updates, and then drops out of the system. The above process repeats when the object reregisters into the system. Figure 6 depicts the life cycle of an object. In the previous sections, we have addressed the problem of processing updating requests in LUGrid. In this section, we discuss how LUGrid gracefully processes registering and dropping requests.

Dropping objects. Dropping an object out of LUGrid is equivalent to marking the current entry of the object as obsolete. LUGrid achieves this through the same lazy-deletion technique as for object updates. A dropping command is interpreted in LUGrid as if the dropping object updates the location to a special "non-existing" location. To maintain the consistency of the system, the whole dropping process consists of the following two steps: (1) Delete any former update for the object out of MG if the update has not been flushed to disk yet. (2) If an MDM entry for the object exists, change the location field of the MDM entry to "non-existing", and increment the *miss deletion number* by one. Otherwise, a new MDM entry is created. The new MDM entry fills the location filed as "non-existing" and sets the *miss deletion number* to one. Following the above two steps, LUGrid identifies all object entries for the dropping object as *obsolete*, thus lazily delete them from disk.

Registering objects. An object needs to registers itself into the system when the object is activated for the first time, or when it reconnects after a previous dropping action. Upon the arrival of a registering object, say *o*, LUGrid does not contain any current entry for *o*. Note that LUGrid may contain obsolete entries for *o*, and consequently contains a corresponding MDM entry. This happens if *o* has issued a dropping command previously, but not all obsolete entries have been cleared out of LUGrid by the time *o* reregisters. LUGrid treats a registering object as an ordinary update tuple. However, the registering object neither creates a new MDM entry (if no MDM entry for the object is found) nor increments the *miss deletion number* of an existing MDM entry (if an MDM entry for the object is found). If one MDM entry for the object exists, only the location information of the MDM entry is changed to the registering location of the object. For the registering command, a tricky situation happens when LUGrid receives an update from an object while the object's previous registration information has not been consumed. To maintain the consistency of the system, LUGrid replaces the registering object with a new update in memory, and marks the update as a registering object.

4 Query Processing in LUGrid

In this section, we discuss query processing in LUGrid. In Section 4.1, we provide a process for distinguishing obsolete and current entries in LUGrid. In Section 4.2, we discuss the processing steps for standing queries, and provide the algorithm for processing range queries.

Function IdentifyingEntry(Entry e)

1. *If (e is in DG)*
   (a) *If (there exists an MG entry m where m.OID equals e.OID), return OBSOLETE;*
   (b) *If (there exists an MDM entry md where md.OID equals to e.OID and md.OLoc $\neq$ e.OLoc), return OBSOLETE;*
2. *For any other cases, return CURRENT;*

Figure 7. Identifying Current/Obsolete Entries

4.1 Identifying Obsolete Entries

In LUGrid, current entries are mixed with obsolete entries. A challenge lies on identifying obsolete entries from the current entries. The function given in Figure 7 provides the pseudo code for this identification process.

First, any update entry buffered in MG cells is current. Assuming it is not, a newer update for the same object must exist in LUGrid. As discussed in Section 3, the arrival of the newer update would have replaced the older one in MG, which is a contradiction. (see Step 1 in Figure 3). The contradiction justifies that any existing entries in MG are current. For a disk entry, however, it takes two steps to identify a current entry. First, if an update entry for the same object is found in MG, then the disk entry is an obsolete one. The entry in MG must arrive after the one on disk, hence it causes the disk entry to be obsolete. Second, if an MDM entry for the updated object is found, and the location information of the MDM entry does not equal to the location information of the disk entry, then the disk entry is an obsolete one. The existence of such MDM entry indicates that a newer update has been flushed to some other disk cell. If after the above two steps, a disk entry has not been identified as obsolete, the entry is a current one. LUGrid is designed carefully so that the above identification process is performed efficiently. Recall that entries in MG are double-hashed on both object locations and object identifiers, and that MDM is hashed on object identifiers. Therefore, given a disk entry, we can directly reach the MG entry or the MDM entry for the same object, if one exists.

4.2 Answering Queries

Query processing in LUGrid exploits both memory grid (MG) and disk grid (DG). This is because the latest object locations are either temporarily buffered in MG or persistently stored on DG. The steps of 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;

Procedure RangeQueryProcessing(QueryArea rgn)

1. *QueryAnswer* = $\emptyset$;
2. *Search for MG cells whose space coverage overlap with rgn. Put pointers to such MG cells into a set* Set_m;
3. *For each MG cell m referred in* Set_m
   (a) *For each update u in m*
      i. *If (u.OLoc is inside rgn), QueryAnswer* $\leftarrow$ *u;*
4. $Set_{pid} = \emptyset$;
5. *For each MG cell m referred in* Set_m
   (a) $Set_{pid} = Set_{pid} \cup m.D_{id}$;
6. *For each disk page d referred in* Set_{pid}
   (a) *Read d into memory;*
   (b) *For each entry e in d*
      i. *If ((IdentifyingEntry(d) == CURRENT) and (d.OLoc is inside rgn)), QueryAnswer* $\leftarrow$ *d;*
7. *Return QueryAnswer;*

Figure 8. Processing Range Queries

(2) For each object entry in the selected cell set, identify the entry as current or obsolete. Obsolete entries are not considered further by the query. Current entries continue to go through the query operator and produce corresponding output.

The above steps are processed in LUGrid elegantly and efficiently. Figure 8 gives the pseudo code for processing a *range query Q* using LUGrid. Initially, the Q's answer is set to empty. The algorithm starts by searching for MG cells whose space coverage overlap with the query area (Step 2 in Figure 8). For updates buffered in these MG cells, the algorithm checks whether they are within the query area. Entries within the query area are added to the query answer (Step 3 in Figure 8). Then, for MG cells that overlap with the query area, Set_{pid} represents the set of page identifiers of their repository cells (Step 4 and 5 in Figure 8). In the case that multiple MG cells share one DG cell, the page identifier of the DG cell is stored only once to avoid redundant processing. Having obtained such a set of page identifiers (Set_{pid}), the algorithm reads every disk page in Set_{pid} into memory, and identifies every entry on the page to be *current* or *obsolete*. The identifying process is executed using the algorithm given in Figure 7. For the *current* entry, if its location is inside the query area, it is included in the query answer (Step 6 in Figure 8). Finally, the answer set is returned. Section 6 shows that the query processing in LUGrid is I/O efficient.

5 Cost Analysis

In this section, we theoretically analyze the costs of updating operations for the proposed techniques. Two unique techniques are utilized in LUGrid, namely, *lazy-insertion* and *lazy-deletion*. Since both techniques can be applied independently or together, our analysis studies four cases: *naive approach*, *lazy-insertion only*, *lazy-deletion only*, and *lazy-insertion plus lazy-deletion*.

The analysis is based on a uniform distribution of moving objects in the two-dimensional space. However, since LUGrid adapts to various data distributions, other forms of object distributions have similar effects as uniform distribution. In the analysis, we focus on the steady-state where cell splitting and merging rarely take place. Thus, the cost of splitting and merging is ignored in our cost analysis. 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. In a uniform distribution, MG cells have approximately equal sizes. We define δ as the percentage of updates that a certain object, say P, moves from a certain disk-based cell to another one. Since no I/O cost is involved when buffer updates in memory, we only consider the flushing phase in our analysis.

Naive approach. The naive approach utilizes none of the proposed techniques in update processing. An update goes to disk as soon as it arrives to the system, meanwhile its old entry is deleted mandatorily. Assume we have n_u updates, the total I/O cost for processing these updates is given by Equation 1.

$$IO_{naive} = 2n_u + 4(n_u * \delta) \tag{1}$$

In Equation 1, $2n_u$ is induced by reading and writing a disk cell for each update. $(n_u * \delta)$ represents the expected number of updates that have old entries in other disk cells. For each such update, the auxiliary index structure is searched once to get the page p of the old entry. Then, p is read and written once to delete the old entry. Finally, the auxiliary index is changed and flushed back to disk to reflect the new page of the object. Thus, four additional I/O operations are required for each single update that results in a cell change. According to Equation 1, the *average* cost for one update under the naive approach is given by Equation 2:

$$IO_{naive\_avg} = 2 + 4\delta \tag{2}$$

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. Notice that the MDM hash table in LUGrid is not applicable in case of *lazy insertion*.

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 3.

$$IO_{LI} = 2 + 4(n_u * \delta) \tag{3}$$

In Equation 3, 2 is introduced by reading and writing the repository cell only once for all n_u updates. $4(n_u * \delta)$ is the same as that of Equation 1. The expected number of updates in an MG cell is U/N_m. So, according to Equation 3, the *average* updating cost for one object is:

$$IO_{LI\_avg} = \frac{2N_m}{U} + 4\delta \tag{4}$$

From Equation 4, we conclude that a larger memory pool for buffering updates results in a less average cost for object updates. Specifically, with fixing other parameters while increasing the memory buffer from U_1 to U_2, the average updating cost is reduced by:

$$IO_{LI\_diff} = 2N_m(\frac{1}{U_1} - \frac{1}{U_2}) \tag{5}$$

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. In *lazy-deletion only*, no secondary index is required for locating objects, and no memory buffer is needed to buffer updates.

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:

$$IO_{LD\_avg} = 2 + \frac{2}{c} \tag{6}$$

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:

$$IO_{LI\&LD\_avg} = \frac{2}{n_u} + \frac{2}{c} = \frac{2N_m}{U} + \frac{2}{c} \tag{7}$$

6 Performance Evaluation

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

PARAMETERS	VALUES USED
Object distribution	**Uniform**, Normal distribution
Object velocity	10, 50, **100**, 500 miles/hour
Update ratio of objects	0%, 30%, **5%/cycle**
Disk page size	2048, **4096**, 8192 bytes
MG buffer size	0%, **1%**, 2% of object number
MDM size	**No limitation**, 10k bytes

Table 1. Experiment Parameters and Values

update processing and query processing. FUR-tree modifies the original R-tree by processing object updates in a bottom-up fashion. For FUR-tree, we implement the *global bottom-up* approach as proposed in [9] and tune its parameters to achieve best performance. 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 and not counted for I/O costs. In our experiments, the size of the first two R-tree levels is about 8 pages, approximately equals to the size of MG without the buffer for object updates. (2) Whenever LUGrid consumes some amount of memory, we give FUR-tree a same size buffer that is maintained in a *Least Recently Used* (LRU) manner. FUR-tree makes use of the buffer when accessing leaf pages of the R-tree and the pages of the auxiliary index on object identifiers.

In all experiments, we collect the results of LUGrid when the system becomes stable. In this case, cell splitting or merging rarely happens. This does not affect the quality of the results because we are interested in the updating and querying performance of LUGrid, as well as resource consumption. To maximize the effect of lazy-insertion, the maximum number of updates that one MG cell can buffer is set to the number of entries that one disk page can contain. In all experiments, the clean interval 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, 1,000,000 objects are moving inside a space that represents 1000 * 1000 square miles. Objects are continuously moving with given velocities. We count the number of object updates in *cycles*, each cycle takes 10 seconds. In each cycle, a certain ratio of objects report their new locations by issuing updating requests. We conduct experiments using various disk page sizes and various MG buffer sizes. To test the adaptation to object distributions, 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)*. We generate the original objects and consequent updates in a way similar to GSTD [22]. Various parameters for our

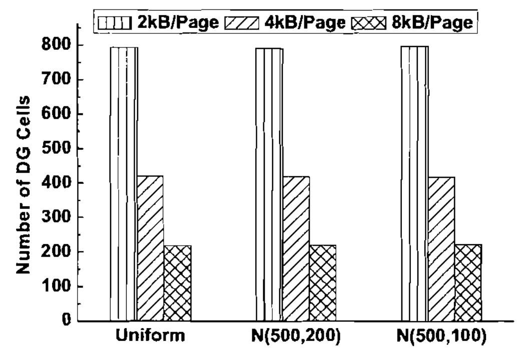

Figure 9. Number of DG cells

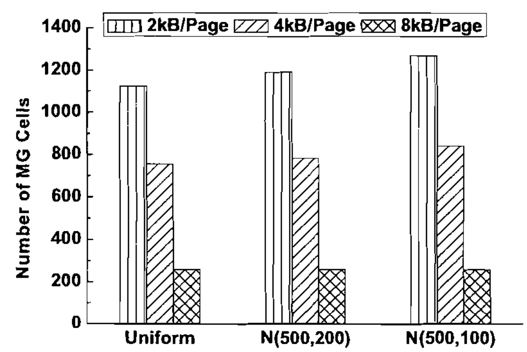

Figure 10. Number of MG cells

experiments are outlined in table 1, the default values are given in **bold**. Our experiments adopt the number of disk I/Os as the primary cost metric.

6.1 Resource Utilization

We analyze the disk and memory utilization of LUGrid under various page sizes (2048, 4096 and 8192 bytes/page) and various object distributions. Figures 9 and 10 give the number of disk cells and memory cells, respectively, used in LUGrid. In Figure 9, with different page sizes, the same numbers of disk pages are allocated for different distributions, which indicates that LUGrid is adaptive in terms of object distribution. The number of MG cells in Figure 10 slightly grows with the skewness in object distribution. The main reason is that MG is more apt to splitting when a disk cell splits if the object distribution is skewed. However, the maximum difference among the numbers of MG cells is less than 15% in our experiments, which shows that the number of MG cells is well balanced by data distributions.

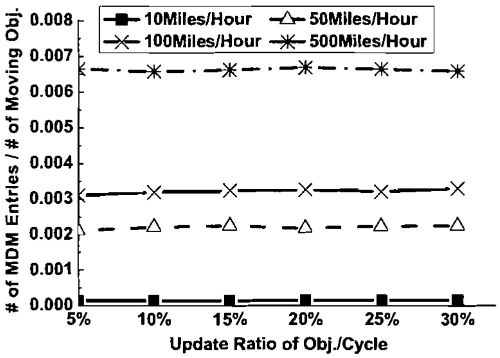

Figure 11. Size of the Miss-Deletion Memo

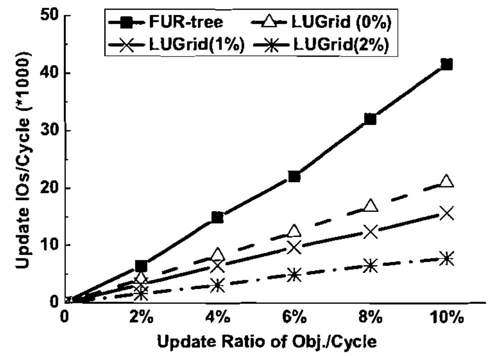

Figure 12. Update cost vs. No. of updates

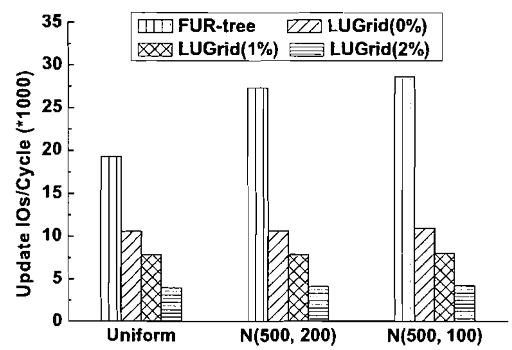

Figure 13. Update cost vs. obj. distribution

Figure 11 gives the size of the *Miss-Deletion Memo* 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 in one cycle from 5% to 30%. In Figure 11, the size of MDM is expressed as the ratio between the number of MDM entries to the total number of objects. In all cases, the number of MDM entries is less than 0.7% of the total number of objects. When the object velocity increases, the size of MDM gets 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 hold information of these cell-changing objects. However, the update ratio of objects does not affect the size of MDM. The reason is that the number of MDM entries is not determined by the total number of updates. Instead, the number of MDM entries is determined by the number of cell-changing objects in each flushing period. The old entries in a cell are deleted when the cell is flushed again. The deletion of old entries reduces the size of MDM. This experiment demonstrates that the size of Miss-Deletion Memo is extremely small and hence can easily fit in main memory.

6.2 Updating Performance

In this section, we study the update performance of LUGrid and compare it with FUR-tree. For LUGrid, we use different sizes of MG buffers. Specifically, the MG buffer is set as 0% (i.e., no buffer slots), 1% (i.e., 1000 slots) and 2% (i.e., 2000 slots) of the indexed objects. A 0% size buffer represents a *lazy-deletion only* scenario, as discussed in Section 5. Figure 12 plots the number of I/Os when the ratio of objects that report updates increases from 0 to 10% per cycle. As can be seen from the figure, 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 13 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, while the form of 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 14 demonstrates the effect of the object velocity, where objects are moving with various velocities (10, 50,

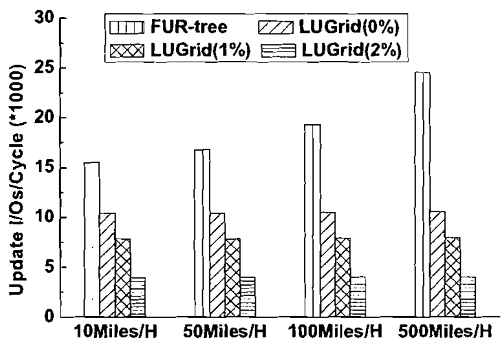

Figure 14. Updating cost vs. obj. velocity

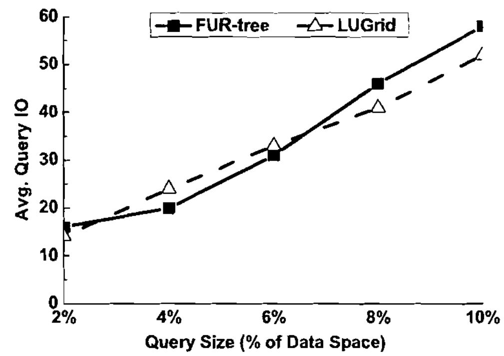

Figure 16. Query cost vs. query size

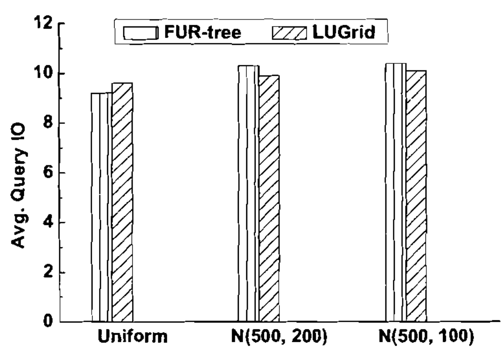

Figure 15. Query cost vs. obj. distribution

100 and 500 miles/hour). As shown in Figure 14, 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.

6.3 Querying Performance

In this section, we study the querying 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, range queries are specified as squares and uniformly distributed in space. Figure 15 compares the querying costs with respect to object distributions. In this experiment, each query covers 1% of the whole space. The experiment shows that under all object distributions, LUGrid is similar to FUR-tree in processing range queries. Both FUR-tree and LUGrid are slightly affected by object distribution. Figure 16 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. In all cases, the performance of LUGrid is similar to the performance of FUR-tree.

7 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, namely, *lazy-insertion* and *lazy-deletion*. 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. Further, lazy-insertion groups updates and flushes multiple updates at one time, so that the cost for single update is amortized. We believe that the proposed lazy-update techniques in LUGrid 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, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2), 2003.

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

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

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

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

[15] 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.

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

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

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

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

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

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

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

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

[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.