Purdue University

[Purdue e-Pubs](https://docs.lib.purdue.edu/)

[Department of Computer Science Technical](https://docs.lib.purdue.edu/cstech)

Department of Computer Science

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

Xiong, Xiaopeng; Mokbel, Mohamed F.; and Aref, Walid G., "LUGrid: Update-tolerant Grid-based Indexing for Moving Objects" (2005). Department of Computer Science Technical Reports. Paper 1636. https://docs.lib.purdue.edu/cstech/1636

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

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

 $\mathcal{L}^{\text{max}}_{\text{max}}$

Xiaopeng Xiong Mohamed F. Mokbel Walid G. Aref Walid G. Aref Xiaopeng Xiong Mohamed F. Mokbel

> **CSD TR #05-022** CSD TR #05-022 **October 2005** October 2005

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

 Xi openg Xiong^l Mohamed F. Mokbel² Walid G. Aref^l

¹Department of Computer Science, Purdue University, West Lafayette, IN 2 Department of Computer Science and Engineering, University of Minnesota. Minneapolis, MN

Abstract Abstract

I11deri11g ~vovirlg objects is afiozdanlental issue ilz slmtio-Indexing moving objects is *afundamental issue* in *spatioten1l~oral dntnbases. 111 this paper, we propose all ado]>- temporal databases. In this paper, we propose an adapfive* Lazy-Update Grid-based index *(LUGrid, for short) thnt tive* Lazy-Update Grid-based index *(LUGrid,for short) that mininlires r11e cost of object updates. LUGI-id is de~igned minimizes the cost of object updates. LUGrid* is *designed rvitll two inlportantfentures, nninely,* lazy insertion *and* lazy *with two importantfeatures, namely,* lazy insertion *and* lazy deletion. Lazy insertion *reduces the update 110s adding* deletion. Lazy insertion *reduces the update liDs by adding 017 addirional 17lenlop-reside171 layer over the disk i11de.x. an additional memory-resident layer over the disk index. Tl~ei-efow, abatch of updntes can be.pushed to disk (it one Therefore, a batch of updates can be flushed to disk at one time, and consequently the cost of niulriyle updntes is nnlor-time, and consequently the cost ofmultiple updates* is *amortized.* Lazy deletion *reduces update cost by avoiding deleting single obsolete entry out of the index immediately. Instend, the obsolete entries ore I-enloved Inter 0). specinll~ stead, the obsolete entries are removed later by specially designed nlecha~zisms. LUGrid adopts to object disfl-ibu-designed mechanisms. LUGrid adapts to object distributioils throligh cell splitting and ~nerging. III the papel; we tions through cell splitting and merging. In the papel; we e.xfe~isive!). disc~lss the strucrure of LUG]-id and the algo-extensively discuss the structure of LUGrid and the algoritl~ms for ~lpdate nnd quep processing. Moreovei; we pro- rithmsfor update and query processing. Moreovel; we pro-\vide theoretical annl.ysis for estimating /he updnte cost of vide theoretical analysis for estimating the update cost of LUGrid. Conlpl-ehensive e.xperinle~ztnl resr~lrs iizdicnte that LUGrid. Comprehensive experimental results indicate that LUGrid outpelfol-111s foj-iner work up to eiglir rinles when LUGrid outpelfonns fonner work up to eight times when processing illtensive updates, while yielding siinilnl- search processing intensive updates, while yielding similar search pe1foi-n1nnce. pelformance.*

1 Introduction 1 Introduction

The integration of mobile devices and positioning tech-The integration of mobile devices and positioning nologies enables new environments where locations of nologies enables new environments where locations of moving objects can be tracked continuously. In such en-moving objects can be tracked continuously. In such environments, objects send their current locations to a server vironments, objects send their current locations to a server either periodically or based on their moving distance. The either periodically or based on their moving distance. The server collects the location information and processes in-server collects the location information and processes terested queries. A wide range of applications rely on the terested queries. A wide range of applications rely on the maintenance of current locations of moving objects. Exam-maintenance of current locations of moving objects. Examples of these applications include traffic monitoring. nearby ples of these applications include traffic monitoring, nearby information accessing and enhanced 91 1 service. etc. information accessing and enhanced 911 service, etc.

Usually. the number of moving objects tends to be huge. UsuaJly, the number of moving objects tends to be huge, and the server needs to build indexes on the current lo-and the server needs to build indexes on the current locations of objects to accelerate the processing of standing cations of objects to accelerate the processing of standing

queries. However, most of existing spatial indexes (e.g., Rtree-like indexes) are desiped for static data and exhibit tree-like indexes) are designed for static data and exhibit poor performance under frequent updates. Recently, some poor performance under frequent updates. Recently, some new techniques (e.g.. see Section 2) have been proposed new techniques (e.g., see Section 2) have been proposed to alleviate the situation. However. as we demonstrate in to alleviate the situation. However. as we demonstrate in Section 6, these techniques are still not efficient enough for Section 6, these techniques are still not efficient enough for quickly handling large amounts of updates in a short period quickly handling large amounts of updates in a short period of time. The problem of indexing continuously moving ob-of time. The problem of indexing continuously moving objects is far from being resolved. jects is far from being resolved.

Existing approaches on indexing moving objects suffer Existing approaches on indexing moving objects suffer from large volumes of updates. The reasons are observed from large volumes of updates. The reasons are observed from the following three aspects. First. most of the index-from the following three aspects. First, most of the indexing approaches process single updates independently. Since ing approaches process single updates independently. Since every call for index updating is expensive, processing sin-every call for index updating is expensive, processing single update one at a time hinders largely the scalability of the gle update one at a time hinders largely the scalability of the index. Second, when an update arrives: existing approaches 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 resides on a disk page different from the page where the new entry is to be stored, additional disk 110s are required new entry is to be stored, additional disk I/Os are required to purge the old entry from the index. In many cases, elimi-to purge the old entry from the index. In many cases, eliminating old entries is more costly than inserting new entries. nating old entries is more costly than inserting new entries. Third, to quickly locate the old entry for an object, many Third, to quickly locate the old entry for an object, many index structures maintain a secondary index on object IDS index structures maintain a secondary index on object IDs (e.g.: [8, 9, 11, 241). Maintaining a secondary index is ex-(e.g., [8,9, **11,24]).** Maintaining a secondary index is pensive in itself. The secondary index has to be updated ev-pensive in itself. The secondary index has to be updated every time an object changes its locality of disk page. Further. ery time an object changes its locality of disk page. Further. for each update, at least one page of the secondary index for each update, at least one page of the secondary index is searched in order to locate the old entry of the updating is searched in order to locate the old entry of the updating object, which adds more burden to the updating process. object, which adds more burden to the updating process.

In this paper, we propose *LUGrid,* an adaptive *Lazy-*In this paper, we propose *LUGrld,* an adaptive *Lazy-Update Grid-Dosed inde-x* for indexing current locations of *Update Grid-based index* for indexing current locations of moving objects. LUGrid aims to avoid the above mentioned moving objects. LUGrid aims to avoid the above mentioned drawbacks of existing indexing techniques. LUGrid mini-drawbacks of existing indexing techniques. LUGrid mizes the 110 costs for updates by adopting the concept of mizes 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 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 amor-caused by multiple independent updates so that the amortized 110 cost for one updating is kept very low. (2) *Lazy-*tized I/O cost for one updating is kept very low. (2) *Lazy-* *deletion.* In contrast to other indexing approaches, LUGrid *deletion.* In contrast to other indexing approaches, LUGrid does not require deleting old entries before inserting up-does not require deleting old entries before inserting updated entries. Instead, LUGrid delays the deletion process dated entries. Instead, LUGrid delays the deletion process until the disk pages where the old entries reside are retrieved until the disk pages where the old entries reside are retrieved into memory. Therefore, I10 cost to search and delete old into memory. Therefore, I/O cost to search and delete old entries from disk is saved. This is achieved by a memory-entries from disk is saved. This is achieved by a memoryresident data structure, namely, the "miss-deletion memo" resident data structure, namely, the "'miss-deletion memo'· (MDM). MDM is a hash-based data structure that maintains (MDM). MDM is a hash-based data structure that maintains ONLY those objects that "miss" at least one deletion. LU-ONLY those objects that "miss" at least one deletion. LU-Grid guarantees that the size of MDM is upper-bounded to Grid guarantees that the size of MDM is upper-bounded to a small size so that it can be easily accommodated in main-a small size so that it can be easily accommodated in mainmemory. memory.

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

The contributions of this paper are summarized as fol-The contributions of this paper are summarized as follows: lows:

- I. We propose LUGrid; an adaptive update-tolerant in-I. We propose LUGrid; an adaptive update-tolerant indexing structure for indexing current locations of mov-dexing structure for indexing current locations of moving objects. LUGrid is designed to minimize the cost ing objects. LUGrid is designed to minimize the cost of processing object updates. of processing object updates.
- 2. We extensively discuss the structure of LUGrid and al-2. We extensively discuss the structure of LUGrid and algorithms for update and query processing. We analyze gorithms for update and query processing. We analyze the update cost of LUGrid theoretically. the update cost of LUGrid theoretically.
- 3. We provide a comprehensive set of experiments 3. We provide a comprehensive set of experiments demonstrating that LUGrid outperforms largely former demonstrating that LUGrid outperforms largely former work in update processing while maintaining similar work in update processing while maintaining similar querying performance. querying performance.

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

2 Related Work 2 **Related Work**

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

In an attempt to reduce the frequency of updates to in-In an attempt to reduce the frequency of updates to index structures, a prediction scheme helps predict the up-dex structures, a prediction scheme helps predict the dates for a certain period of time in the future. Predicted dates for a certain period of time in the future. Predicted future updates are presented as trajectories. Thus, index-future updates are presented as trajectories. Thus, indexing continuous moving objects is reduced to indexing fu-ing continuous moving objects is reduced to indexing future trajectories. Four approaches have been investigated ture trajectories. Four approaches have been investigated for indexing future trajectories: (1) Duality transformation for indexing future trajectories: (I) Duality transformation (e.g., see [I. 3, 7. 131). The main idea is to map the future (e.g., see [I. 3, 7, 13]). The main idea is to map the future trajectory to a single point in another domain, then a dual-trajectory to a single point in another domain, then a duality transformation is used to transfer all queries to the new ity transformation is used to transfer all queries to the new space, (2) Quad-tree-based methods (e.g., see [21]), (3) R-space, (2) Quad-tree-based methods (e.g., see [21]), (3) Rtree-based index structures (e.g., see $[14, 15, 16, 17, 20]$). and (4) B-tree-based structures [6]. However, indexing fu-and (4) B-tree-based structures [6]. However, indexing future trajectories solves only part of the updating problem. ture trajectories solves only part of the updating problem. Two main drawbacks still remain: (1) The ability of pre-Two main drawbacks still remain: (I) The ability of prediction is controlled by the prior knowledge andlor assump-diction is controlled by the prior knowledge and/or assumptions of the object velocity, which is not always available. tions of the object velocity, which is not always available. (2) It is implicitly assumed that the updates of future trajec-(2) It is implicitly assumed that the updates offuture tories are much less than the updates to the object location. tories are much less than the updates to the object location. However, this is not always true where in many cases the However, this is not always true where in many cases the prediction scheme fails (e.g., moving freely in a downtown prediction scheme fails (e.g., moving freely in a downtown area or pedestrian movement). Frequent updates to the pre-area or pedestrian movement). Frequent updates to the prediction scheme would suffer from the same drawbacks of diction scheme would suffer from the same drawbacks of frequent updates in traditional data structures. frequent updates in traditional data structures.

The inefficiency of indexing moving objects by their fu-The inefficiency of indexing moving objects by their future trajectories motivates the need for special data struc-ture trajectories motivates the need for special data structures that are suitable for frequent updates. The Lazy-tures that are suitable for frequent updates. The update R-tree (LUR-tree) [8] modifies the original R-tree update R-tree (LUR-tree) [8] modifies the original R-tree structure to support frequent updates. The main idea is that structure to support frequent updates. The main idea is that if an update to a certain object p would result in a deletion 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 followed by an insertion in a new R-tree node, it would be better if we can increase slightly the size of the minimum better if we can increase slightly the size of the minimum boundary rectangle of the R-tree node in which p lies in to boundary rectangle of the R-tree node in which *p* lies in to accommodate its new location. The Frequently Updated R-accommodate its new location. The Frequently Updated Rtree (FUR-tree) [9] extends the LUR-tree by performing a tree (FUR-tree) [9] extends the LUR-tree by performing a bottom-up approach in which a certain moving object can bottom-up approach in which a certain moving object can move to one of its siblings instead of having deletion fol-move to one of its siblings instead of having deletion followed by an insertion. Both the LUR-tree and the FUR-tree lowed by an insertion. Both the LUR-tree and the FUR-tree use an auxiliary structure to index objects based on their use an auxiliary structure to index objects based on their identifiers. These auxiliary indices locate the old locations identifiers. These auxiliary indices locate the old locations of moving objects. One of the key features of our proposed of moving objects. One of the key features of our proposed data structure LUGrid is that we eliminate the use of such data structure LUGrid is that we eliminate the use of such auxiliary disk indexes since in our proposed scheme the old auxiliary disk indexes since in our proposed scheme the old location is *lacily* visited and deleted. location is *laz.ily* visited and deleted.

The difficulties in dealing with tree-based structures and The difficulties in dealing with tree-based structures and the complexity of dual transformations motivate the use of the complexity of dual transformations motivate the use of simpler data structures (e.g., hash-based and grid-based data simpler data structures (e.g., hash-based and grid-based data

I

structures) that are updated easily. A hash-based structure structures) that are updated easily. A hash-based structure is used in 118. 191 where the space is partitioned into a set is used in [18. 19] where the space is partitioned into a set of overlapped zones. An update is processed only if an of overlapped zones. An update is processed only if an object moves out of its zone. SETI [2] is a logical index object moves out of its zone. SETI [2] is a logical index structure that divides the space into non-overlapped zones. structure that divides the space into non-overlapped zones. Both SETI and hash-based structures ignore deleting the old Both SETI and hash-based structures ignore deleting the old location of a moving object. Thus. an update is reduced location of a moving object. Thus, an update is reduced to only an insertion where past trajectories are maintained. to only an insertion where past trajectories are maintained. Grid-based structures have been used to maintain only the Grid-based structures have been used to maintain only the current locations of moving objects (e.g., see [4, 11, 241). current locations of moving objects (e.g., see [4, 11,24]). However, two drawbacks can be distinguished: (1) The used However, two drawbacks can be distinguished: (I) The used grid is fixed where it is just a regular partitioning of space grid is fixed where it is just a regular partitioning of space into equal sized non-overlapped zones. This approach is not into equal sized non-overlapped zones. This approach is not suitable in the case of a non-uniform distribution of data, suitable in the case of a non-uniform distribution of data, (2) Deleting an old location of a certain object is still cum-(2) Deleting an old location of a certain object is still cumbersome, where in many cases, the old location can be in a bersome, where in many cases, the old location can be in a grid cell that is different from the one containing the new grid cell that is different from the one containing the new location. In this case, an extra search and extra 110s are location. In this case, an extra search and extra I/Os are needed to clean LIP the old entry. needed to clean up the old entry.

In our recent work (231, we initially proposed making In our recent work [23], we initially proposed making use of an *Upclote Me1710* to reduce the update cost. The use of an *Update Memo* to reduce the update cost. The main idea is to avoid immediate deletion of obsolete entries main idea is to avoid immediate deletion of obsolete entries by maintaining a memo structure in main memory. [23] 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 plore similar idea in adaptive Grid-based indexes to achieve *lazy deletion.* Furthermore, by utilizing *lazy insei-tiorz* along *lazy deletion.* Furthermore, by utilizing *lazy insertion* along with *lazy deletior?,* the update performance is significantly with *lazy deletion,* the update performance is significantly enhanced. enhanced.

Our proposed LUGrid structure distinguishes itself from Our proposed LUGrid structure distinguishes itself from all other approaches where it has all the following proper-all other approaches where it has all the following properties: (1) LUGrid indexes the current positions of moving ties: (l) LUGrid indexes the current positions of moving objects, no predication scheme is used, (2) LUGrid is based 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 cells can adapt to data distribution through cell splitting and merging. (3) LUGrid efficiently resolves the issue of dele-merging. (3) LUGrid efficiently resolves the issue of deletion, where a delete is performed *lazily.* Thus, no overhead tion, where a delete is performed *lazily.* Thus, no overhead or UO is incurred due to deletion. or I/O is incurred due to deletion.

3 LUGrid: Lazy Update Grid-based Index 3 LUGrid: Lazy Update Grid-based Index

In this section, we propose *LUGrid,* an adaptive grid-In this section, we propose *LUGrid,* an adaptive gridbased index structure that efficiently handIes the continuous based index structure that efficiently handles the continuous updates of objects' locations. LUGrid exploits two tech-updates of objects' locations. LUGrid exploits two techniques. namely, *lazy-insertion* and *lozjl-deletiorz.* In *lazy-*niques, namely, *lazy-insertion* and *lazy-deletion.* In *lazyinsertion*, incoming updates are grouped together based on the updated disk-page and are *lazily* flushed into disk once. the updated disk-page and are *lazily* flushed into disk once. Thus, multiple updates are reduced to only a single disk Thus, multiple updates are reduced to only a single disk update. In *lazy-deletiorz.* obsolete entries (i.e., entries that update. In *lazy-deletion,* obsolete entries (i.e., entries that receive an update) remain in disk rather than being immedi-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 can *lazily* remove the obsolete entries only when their disk pages are accessed, e.g., via an insertion. Thus. a *delete* op-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 *lazy-deletion* can be used either independently or together to boost the performance of frequent updates in traditional to boost the performance of frequent updates in traditional index structures. index structures.

3.1 LUGrid Indexing Structure 3.1 LUGrid Indexing Structure

LUGrid adopts a grid structure that is similar to the *Grid* LUGrid adopts a grid structure that is similar to the *Grid jle* [I 21. In LUGrid, however, the directory of grid cells *file* [12]. In LUGrid, however, the directory of grid cells is maintained in memory instead of being stored on disk. is maintained in memory instead of being stored on disk. Also. we extend the grid directory to buffer object updates. Also, we extend the grid directory to buffer object updates. We refer to the extended in-memory directory as the *Mem-*We refer to the extended in-memory directory as the *o~ Grid,* and refer to the set of in-disk bucket pages as the *ory Grid,* and refer to the set of in-disk bucket pages as the *Disk Grid.* Additionally, a hashing-based structure termed *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 con-entries. These three structures act together to maintain continuous object updates in LUGrid. tinuous object updates in LUGrid.

Disk Grid (DG) Disk Grid (DG)

The *Disk Grid* (DG, for short) consists of a set of non-The *Disk Grid* (DG, for short) consists of a set of nonoverlapped disk-based grid cells. Each grid cell is stored overlapped disk-based grid cells. Each grid cell is stored in one disk page. A DG cell stores information of objects in one disk page. A DG cell stores information of objects that lie within the cell boundaries. Each DG cell covers that lie within the cell boundaries. Each DG cell covers an exclusive portion of the data space that is determined 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, \dots, 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 of *(OID,OLoc),* where *OlD* 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 rest of the paper, we use the terms *disk page* and DG *cell* as synonyms. synonyms.

Memory Grid (MG) Memory Grid (MG)

The *Memor~ Grid* (MG, for short) is an in-memory two-The *Memory Grid* (MG, for short) is an in-memory twodimensional array, where each element of the array is a dimensional array, where each element of the array is a *Memory Grid cell.* Each MG cell points to a DG cell where *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 *repositov cell* of sponding DG cell *d,* we refer to *d* as the *repository cell* of *m,* and refer to m as the *bzlffer cell* of *d.* To avoid under-m, and refer to m as the *buffer cell* of *d.* To avoid underutilizing disk pages. several neighbored MG cells may have utilizing disk pages, several neighbored MG cells may have the same repository cell given that the united space region the same repository cell given that the united space region of these MG cells forms a rectangle. However, in any case, 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 coverage of a DG cell is the united space region of all its buffer cells. buffer cells.

Each MG cell has a limited amount of memory that Each MG cell has a limited amount of memory that can buffer object updates temporarily. Object updates are can buffer object updates temporarily. Object updates are *double-hashed* in MG. First, one update is inserted as a *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. entry is linked in a hash link based on the object identifier.

Figure 1. Example: Buffering and Flushing Figure 1. Example: BUffering and Flushing

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

An MG cell has the form of $(N_u, M_{Region}, D_{id}, N_E)$. $D_{Region}, E_1, \dots, E_m)$ *(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 object updates buffered in this MG cell. An object update entry has the form of (OID. OLoc), where OID is the ob-entry has the form of *(OJD, OLoe),* where *OJD* is the object identifier, and *OLoc* is the latest *received* location for the object. the object.

Miss-Deletion Memo (MDM) Miss-Deletion Memo (MDM)

In LUGrid, old object entries may co-exist with current en-In LUGrid, old object entries may co-exist with current entries since the deletion of old entries is delayed. The *Miss-*tries since the deletion of old entries is delayed. The *Miss-Deleti017 Me1710* (MDM, for short) is employed to distin-*Deletion Memo* (MDM, for short) is employed to guish obsolete entries from current entries. MDM is an in-guish obsolete entries from current entries. MDM is an memory hash-based table that keeps track of those objects memory hash-based table that keeps track of those objects that miss at least one deletion. In addition, it keeps a counter that miss at least one deletion. In addition, it keeps a counter with the number of deletions that each object missed. An 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 ob-*OJD* is the object identifier, *OLoe* 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 $(O_{12}, (34, 64), I)$ 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 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 NlDM to reduce the entry can be safely removed from the MDM to reduce the memory usage. memory usage.

A running example. We use the example given in Fig-A running example. We use the example given in Figure 1 to illustrate our ideas. Figure I. 1 (a) gives a DG struc-ure I 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 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 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. 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** 3.2 Processing Updates

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

- **Stage I: Buffering updates.** Initially, continuously received updates are buffered in MG. received updates are buffered in MG. continuously
- **Stage II: Flushing updates into disk.** Flushing inmemory data into disk pages is triggered by any of memory data into disk pages is triggered by any of the following two events: (I) An in-memory grid the following two events: (I) An in-memory grid cell C_M is full. In this case, C_M is flushed into its corresponding repository disk-based grid cell, its corresponding repository disk-based grid cell, (2) The overall memory becomes full. In this case, a (2) The overall memory becomes full. In this case, a certain cell is chosen as a victim and is flushed into certain cell is chosen as a victim and is flushed into its corresponding DG repository cell. Notice that it its corresponding DG repository cell. Notice that it may be the case that the whole memory is full while may be the case that the whole memory is full while none of the in-memory cells are full. This is due to the none of the in-memory cells are full. This is due to the fact that we use two different thresholds, one for the fact that we use two different thresholds, one for the maximum number of updates that can be buffered in maximum number of updates that can be buffered in each cell, and the second is for the maximum number each cell, and the second is for the maximum number of updates that can be buffered in the whole memory. of updates that can be buffered in the whole memory. The reason behind this is to allow for more efficient The reason behind this is to allow for more efficient buffering capabilities. The process of flushing an buffering capabilities. The process of flushing an in-memory cell to disk needs a special coordination in-memory cell to disk needs a special coordination among the three used data structures, DG, MG, and among the three used data structures, DG, MG, and MDM. MDM.
- Stage III: Splitting/Merging cells. Finally, if a DG cell is over-full or is under-utilized, cell splitting or merg-over-full or is under-utilized, cell splitting or ing takes place in both in-memory and disk-based grid ing takes place in both in-memory and disk-based grid structures. structures.

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

Figure 2. Overview of LUGrid Figure 2. Overview of LUGrid

Buffering updates. Figure 3 gives the pseudo code of **Buffering** updates. Figure 3 gives the pseudo code of processing incoming updates in LUGrid. Once an object 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 consunied* if u is flushed to disk. Since it may happen is *consumed* if ¹¹ is flushed to disk. Since it may happen that one update arrives to the server while the previous up-that one update arrives to the server while the previous update for the same object has not been consumed, the buffer-date for the same object has not been consumed, the buffering algorithm starts by searching MG for the entry with the ing 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 is found, the found entry is deleted from MG. The reason is that the existing in-memory entry becomes obsolete, thus, 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 a single update, at most one entry with the same OlD may exist in MG, because earlier entries with the same OID are either consumed to disk or are deleted from memory due to either consumed to disk or are deleted from memory due to a newer update. a newer update.

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

Flushing updates. Figure 4 gives the pseudo code for **Flushing** updates. Figure 4 gives the pseudo code for flushing updates into DG cells. The flushing algorithm *con-*flushing updates into DG cells. The flushing algorithm *consun7es* the buffered updates in an in-memory grid cell (MG) *sumes* the buffered updates in an in-memory grid cell (MG) by flushing them to the corresponding repository cell. First, by flushing them to the corresponding repository cell. First, the repository cell is read into memory (Step 1 in Figure 4). the repository cell is read into memory (Step I in Figure 4). For entries in the repository cell, it is possible that some en-For entries in the repository cell, it is possible that some tries have become obsolete due to newer updates in other tries have become obsolete due to newer updates in other disk cells. To identify such objects, for each entry in the 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 same identifier is found, and the location stored in MDM

Procedure onReceivingUpdate(UpdateTup1e *u(oid. loc))* Procedure onReceivingUpdate(UpdateTuple *u(oid. loc»*

- **1.** *Search u.oid in MG by exploring the OID has11 link ill MG.* I. *Search 11.oid in MG by exploring the DID hash link in MG. If an MG entry* m *where* $m. OID$ *equals to u oid is found*
	- (a) *Deleie rn,f,-o~n MG;* (a) *Dele/e mfrom MG;*
	- (b) $MGC(m)$. N_u --; *usedSlots--*;
- *2. Inserr* u *into the MG cell whose* **Ad~~~i~~** *covers 21.10~; 2. Insert u into the MG cell whose* Ai*Region covers 11.loc;*
- *3. MGC(1r). N,* ++; *usedSlois++; 3. MGC(u).Nu ++; usedSlo/s++;*
- *4. Link u in MG's OID hash link based otz u.oid; 4. Link u in MG's DID hash link based on 11.oid;*
- 5. *If* $(MGC(u).N_u \geq MaxUpdateMGCell)$

(a) *Call FlushingUpdates(MGC(~~));* (a) *Call FlushingUpdmes(MGC(u));*

- $6.$ *If (usedSlots >= MaxSlots)*
	- (a) mc_{max} = *the MG cell that buffers the largest number of updates; of updates;*
	- (b) *Call Fll~shii7gUpdates(1n c,,,,,);;* (b) *Call FlushingUpdmes(mc*max);;

Figure 3. On Receiving Object Update Figure 3. On Receiving Object Update

does not correspond to that of the disk entry, the disk entry does not correspond to that of the disk entry, the disk entry is considered obsolete. In this case, the obsolete disk entry is considered obsolete. In this case, the obsolete disk entry is removed from the repository cell (Step 2 in Figure 4), and 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 by one. In the case that the MDM entry indicates all old entries on disk have been deleted for this object (i.e., the entries on disk have been deleted for this object (i.e., the *rliiss deletioiz number-* equals zero), the MDM entry itself is *miss deletion number* equals zero), the MDM entry itself is removed from MDM. removed from MDM.

After deleting obsolete entries, each update in the MG After deleting obsolete entries, each update in the MG cell searches its original entry in the repository cell. The cell searches its original entry in the repository cell. The original entry may or may not exist. If the original entry original entry mayor may not exist. If the original entry is found. the entry is updated with new location informa-is found, the entry is updated with new location tion. In this case, if an MDM entry exists for the object, tion. In this case, if an MDM entry exists for the object, the location field of the MDM entry needs to be updated. the location field of the MDM entry needs to be updated. At the end: the update is deleted from the MG cell (Step 3a At the end, the update is deleted from the MG cell (Step 3a in Figure 4). Otherwise, if no original entry for the up-in Figure 4). Otherwise, if no original entry for the dating object is found, then the original entry must reside dating 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 (Step 3b in Figure 4). In this case, if one MDM entry exists for the object, the MDM entry is updated with new location 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 ex-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 ists, 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). set to one (Step 3b(ii) in Figure 4).

Following the above processing, the MG cell contains 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 with-all such updates can be added to the repository cell without causing overflowing, they are inserted into the repos-out causing overflowing, they are inserted into the repository cell and are removed from the MG cell, and related itory cell and are removed from the MG cell, and related counters are changed accordingly (Step 4a in Figure 4). All counters are changed accordingly (Step 4a in Figure 4). All buffer cells that point to this repository cell need update buffer cells that point to this repository cell need update

Procedure FlushingUpdates(MGCell *me)* Procedure FlushingUpdates(MGCeJl *me)*

- *n*, *dc* = *the repository cell of mc; Read dc into memory;*
- *2. For eoch entr? d iri dc, if on MDM eiirr?. e wher-e c..OID 2. For each el1lrv d* in *de. if an MDM emr)" e where e.OID equols to d.OJ D is,forrnd equals to d.OI D isfound*
	- (a) $If (d.OLoc \neq e.Oloc)$
		- $\ddot{\textbf{i}}$. *Delete d from dc; dc.N_E--; e.MDnum--; A. lf(e.MDn~rrri* == *0) delete e,fiorii MDM;* A. *!j'(e.MDnum* == 0) *delete efrol/1 MDM;*
- **3.** *For eoch eiitv In bi ,rnc 3. For each entry* m in me
	- (a) *If a DG entr? dold iri dc whel-e dOcd.OID equols ro* (a) *if a DG entry do/d* in *de where do/d.OI D equals* ¹⁰ *m .OI D is,found m.OI D isfound*
		- *i. do,d.Oloc* ⁼*m ,010~;* i. *do/d.O/oe* = *m.O/oc;*
		- ii. *If an MDM entry e where e.OID equals to In .OID is,fourid m.OID is[ound*
			- *A. e.Oloc* = *in..Oloc;* A. *e.O/oe* = *m.O/oc;*
		- *iii. Delete m from mc; mc.N_u --; usedSlots--;*
	- (b) *Else* //(f *such d,,rd does riot exist* (b) *Else* / */i[such dold does not exist*
		- *i. If an MDM entry e where e.OID equals to 171 .OID is,forrnd m.OI D isfound*
			- *A.* $e.Oloc = m.Oloc$; $e.MDnum++$;
		- *ii. Else llifs~rck edoes not exist* ii. *Else* / / if*such e does not exist*
			- *A. Creote e os o new MDM enrr?.: e.OID* = A. *Create e as a new MDM emr)"; e.OID* = *m.OID; e.OLoc* = *m.OLoc; e.AlDn7rni m.OI*D; *e.oLoc* = *m.OLoe; e.MDnvm* = *I; 117ser-r e blro MDM;* = *i; insert e into MDM;*
- 4. *If* $(mc.N_u + dc.N_E \leq MaxEntPerDCCell)$
	- (a) *Move oll remoinirlg MG enri-ies iri rnc to dc; dc.N~* (a) *Move all remaining MG emries* in *mc to dc; dc.NE* $= dc.N_E + mc.N_u$; $usedSlots = usedSlots - mc.N_u$; $mc_{\gamma}N_{u} = 0;$ $mc_{\gamma}N_{E} = dc_{\gamma}N_{E};$
	- (b) For all buffer cells of dc, set their values of N_E to $dc.N_E;$
	- (c) *CoII Mer-gingCell(rnc, dc);* (c) *Call MergingCell(mc, dc);*
- *5. Else call SplittingCell(nic, dc); 5. Else call SplittingCell(mc, dc);*

Figure 4. Flushing Buffered Updates Figure 4. Flushing Buffered Updates

their counters for the number of disk entries (Step 4b in their counters for the number of disk entries (Step 4b in Figure 4). Then, a merging function is called to seek the Figure 4). Then, a merging function is called to seek the opportunity of merging this DG cell with neighbored cells opportunity of merging this DG cell with neighbored cells (Step 4c in Figure 4). Otherwise, if putting all remaining (Step 4c in Figure 4). Otherwise, if putting all remaining updates into the repository cell causes overflowing of the updates into the repository cell causes overflowing of the repository cell, the repository cell is split to two disk cells repository cell, the repository cell is split to two disk cells (Step 5 in Figure 4). (Step *5* in Figure 4).

Example. In the example given in Figure I, we assume Example. In the example given in Figure I, we assume that an MG cell needs flushing when the number of buffered that an MG cell needs flushing when the number of buffered updates in this MG cell reaches two. In Figure l .l(b), MG updates in this MG cell reaches two. In Figure 1.1 (b), MG cell 2 receives the location updates from objects 1 and 4. cell 2 receives the location updates from objects 1 and 4. Since the buffer of cell 2 is full, cell 2 flushes the two up-Since the buffer of cell 2 is full, cell 2 flushes the two dates to its repository cell which is DG cell A. Following the flushing algorithm, entries in DG cell *A* check with ing the flushing algorithm, entries in DG cell A check with

MDM to see whether there are obsolete entries. In our ex-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 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 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 however, the obsolete entry (plotted with cross mark) still remains on disk. remains on disk.

Assume that after some time. the MG cell 5 receives two Assume that after some time, the MG cell 5 receives two updates from object *G* and 7 and starts flushing to DG cell 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 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 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, final states ofDG, 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 3.3 Splitting and Merging Cells

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

Cell splitting. In LUGrid, splitting always happens 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 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 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 fecting MG. In this phase, the splitting process collects the information of D's *buffer cells* from MG (In addition to 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 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) into two sets that satisfy the following two conditions: (A) The united space coverage for the cells in each set forms The united space coverage for the cells in each set forms a rectangular box; (B) When we re-distribute the entries in 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 sat-that there are multiple ways to do the partitioning that satisfy the above conditions, the process chooses the partitions isfy the above conditions, the process chooses the partitions where the numbers of re-distributed entries have the least 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 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 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 zontally or vertically. The split position lies at the median object along the split dimension. In this case, since one MG object along the split dimension. In this case, since one MG cell may have exactly one DG repository cell, MG needs cell may have exactly one DG repository cell, MG needs splitting at the same split position. All MG cells that over-splitting at the same split position. All MG cells that overlaps the splitting line are split. MG splitting will result in laps the splitting line are split. MG splitting will result in moving buffered updates from the original MG cells to the moving buffered updates from the original MG cells to the split MG cells, according to the space coverage of new MG split MG cells, according to the space coverage of new MG cells. cells.

Cell merging. Two neighbor DG cells may merge into Cell merging. Two neighbor DG cells may merge into one DG cell given the resulting DG cell does not overflow. one DG cell given the resulting DG cell does not overflow. LUGrid adopts the merging scheme of the *11eigl7bor sys-*LUGrid adopts the merging scheme of the *neighbor sysre171* as used in the Grid file [12]. In the neighbor system *tem* as used in the Grid file [12]. In the neighbor system scheme, each DG cell can merge with either of its two ad-scheme, each DG cell can merge with either of its two adjacent neighbors in each dimension given that the united jacent neighbors in each dimension given that the united

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

Example. We present the example in Figure 5 to illus-Example. We present the example in Figure 5 to illustrate the process of cell splitting and merging in LUGrid. trate the process of cell splitting and merging in LUGrid. Figure 5.l(a) shows a DG with the five DG cells, A to E. Figure 5.I(a) shows a DG with the five DG cells, *A* to *E.* $\frac{1}{1}$ \cdots Fourteen objects (o_1 - o_{14}) are stored in the DG, while an obsolete entry for 04 exists in DG cell A. Figure 5. I (b) gives obsolete entry for 04 exists in DG cell *A.* Figure 5.1 (b) gives the corresponding MG that consists of six MG cells (cell 1 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 at most four entries. and an MG cell flushes updates once it receives two updates. Figure 5.l(b) shows three object up-receives two updates. Figure 5.1 (b) shows three object dates that are buffered in MG. Since the number of updates dates that are buffered in MG. Since the number of updates in MG cell 4 reaches 2: MG cell 4 flushes updates to DG 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 \overrightarrow{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), after the splitting are plotted in Figure 5.2(a), 2(b) and 2(c), respectively. During the splitting, the update for *06* that was respectively. During the splitting, the update for 06 that was buffered in MG cell 5 needs to move to the new MG cell 8. Note that the *i17iss deleti011 17ic117Der* of the MDM entry for 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). (in DG cell *A* and *D,* respectively).

If after the above processing, MG cell 8 receives another If after the above processing, MG cell 8 receives another update from *09* (in Figure 5.2(b)), MG cell 8 starts to flush update from 09 (in Figure 5.2(b)), MG cell 8 starts to flush updates to DG cell *D.* After the flushing, DG cell *D* seeks updates to DG cell D. After the flushing, DG cell D seeks the opportunity to merge with neighbor DG cells. DG cell 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 Fig-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 3.4 Obsolete Entry Cleaning

In this section. we discuss issues related to the number In this section, we discuss issues related to the number of obsolete disk entries (due to lazy deletion) and the size of of obsolete disk entries (due to lazy deletion) and the size of the MDM in LUGrid. Throughout this section, let *Nold* be the MDM in LUGrid. Throughout this section, let N *o1d* 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 *Nold* is always larger than or equal to to each other, and *Nold* is always larger than or equal to M_{ent} , since one MDM entry represents one or more missed deletions for a certain object. deletions for a certain object.

Recall that when flushing a memory cell, obsolete en-Recall that when flushing a memory cell, obsolete en-

÷,

Figure 6. Life Cycle Of Object Figure 6. Life Cycle Of Object

tries in the repository cell are first deleted. The removal 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 moval 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. of objects.

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

The cleaner adopts the similar technique as in [23]. The The cleaner adopts the similar technique as in [23]. The basic task for the cleaner is to pick a DG cell and clean all basic task for the cleaner is to pick a DG cell and clean all old entries whenever LUGrid accumulates a fixed number old entries whenever LUGrid accumulates a fixed number of updates. Such a fixed number is termed as *clean inter-*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 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. total number of DG cells.

To maximize the number of old entries deleted from a To maximize the number of old entries deleted from a DG cell, the cleaner always picks the DG cell that has ex-DG cell, the cleaner always picks the DG cell that has experienced the longest time since its latest flushing. In other perienced the longest time since its latest flushing. In other words: the *oldest* DG cell with respect to the latest flushing words, the *oldest* DG cell with respect to the latest flushing time is selected for cleaning. Such DG cell has the potential time is selected for cleaning. Such DG cell has the potential to contain more old entries than the other cells. To identify to contain more old entries than the other cells. To identify the oldest DG cell quickly, the cleaner maintains page iden-the oldest DG cell quickly, the cleaner maintains page identifiers of all DG cells in a *Least Recently Flushed* buffer. tifiers 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 op-to move to the end of the buffer. The split or merge operation results in adding or deleting an identifier from the eration results in adding or deleting an identifier from the buffer. When the cleaning process is invoked, the cleaner buffer. When the cleaning process is invoked, the cleaner picks the first page identifier in the buffer and cleans old picks the first page identifier in the buffer and cleans old entries in the corresponding page. entries in the corresponding page.

3.5 Registering and Dropping Objects 3.5 Registering and Dropping Objects

In practice, an object registers itself into the system, 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 re-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 an object. In the previous sections, we have addressed the problem of processing updating requests in LUGrid. In this problem of processing updating requests in LUGrid. In this section, we discuss how LUGrid gracefully processes regis-section, we discuss how LUGrid gracefully processes registering and dropping requests. tering and dropping requests.

Dropping objects. Dropping an object out of LUGrid is Dropping objects. Dropping an object out of LUGrid is equivalent to marking the current entry of the object as ob-equivalent to marking the current entry of the object as obsolete. LUGrid achieves this through the same lazy-deletion solete. LUGrid achieves this through the same lazy-deletion technique as for object updates. A dropping command is 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 location to a special "non-existing" location. To maintain the consistency of the system, the whole dropping process the consistency of the system, the whole dropping process consists of the following two steps: (I) Delete any former consists of the following two steps: (I) Delete any former update for the object out of MG if the update has not been 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 flushed to disk yet. (2) If an MDM entry for the object exists, change the location field of the MDM entry to "non-exists, change the location field of the MDM entry to "nonexisting", and increment the *liziss deletion llurnbet-* by one. existing", and increment the *miss deletion number* by one. Otherwise, a new MDM entry is created. The new MDM Otherwise, a new MDM entry is created. The new MDM entry fills the location filed as "non-existing" and sets the entry fills the location filed as "non-existing" and sets the *miss deletion rlunlber* to one. Following the above two *miss deletion number* to one. Following the above two steps, LUGrid identifies all object entries for the dropping steps, LUGrid identifies all object entries for the dropping object as *obsolete,* thus lazily delete them from disk. 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 into the system when the object is activated for the first time, or when it reconnects after a previous dropping ac-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 does not contain any current entry for o. Note that LUGrid may contain obsolete entries for *o,* and consequently con-may contain obsolete entries for 0, and consequently contains a corresponding MDM entry. This happens if o has issued a dropping command previously, but not all obsolete 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 registers. LUGrid treats a registering object as an ordinary update tuple. However, the registering object neither cre-update tuple. However, the registering object neither ates a new MDM entry (if no MDM entry for the object is ates 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). ing MDM entry (if an MDM entry for the object is found). If one MDM entry for the object exists, only the location If one MDM entry for the object exists, only the location information of the MDM entry is changed to the registering information of the MDM entry is changed to the registering location of the object. For the registering command, a tricky location of the object. For the registering command, a tricky situation happens when LUGrid receives an update from an situation happens when LUGrid receives an update from an object while the object's previous registration information object while the object's previous registration information has not been consumed. To maintain the consistency of the has not been consumed. To maintain the consistency of the system, LUGrid replaces the registering object with a new system, LUGrid replaces the registering object with a new update in memory, and marks the update as a registering update in memory, and marks the update as a registering object. object.

4 Query Processing in LUGrid 4 Query Processing **in** LUGrid

In this section, we discuss query processing in LUGrid. In this section, we discuss query processing in LUGrid. In Section 4. I, we provide a process for distinguishing ob-In Section 4.1, we provide a process for distinguishing obsolete and current entries in LUGrid. In Section 4.2, we solete and current entries in LUGrid. In Section 4.2, we discuss the processing steps for standing queries, and pro-discuss the processing steps for standing queries, and vide the algorithm for processing range queries. vide the algorithm for processing range queries.

Function IdentifyingEnlry(Entry e) Function IdentifyingEntry(Enrry e)

- 1. *I/(e is in DG)* 1. *If(eisinDG)*
	- (a) *I/ (tker-e exisrs ari MG entry 111 where rii.01D eqzmls* (a) *If (there exists all MG entl)' m where m. OlD equals e.OID), return OBSOLETE; e.OJD), return OBSOLETE;*
	- *(b) If ([here e.ris/s 017 MDM enlr~ 1i1d where 1nd.OID* (b) *If (there exists an MDM entry md where md.OID* e *quals to* e .*OID and* md . $OLoc \neq e$. $OLoc$), *return OBSOLETE; OBSOLETE;*
- *2. For oily ofher- cnses, ret~ini CURRENT; 2. For any other cases, return CURRENT;*

Figure 7. Identifying CurrentIObsolete En-Figure 7. Identifying Current/Obsolete Entries tries

4.1 Identifying Obsolete Entries 4.1 Identifying Obsolete Entries

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

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

4.2 Answering Queries 4.2 Answering Queries

Query processing in LUGrid exploits both memory grid Query processing in LUGrid exploits both memory grid (MG) and disk grid (DG). This is because the latest object (MG) and disk grid (DG). This is because the latest object locations are either temporarily buffered in MG or persis-locations are either temporarily buffered in MG or persistently stored on DG. The steps of answering a query are tently stored on DG. The steps of answering a query are generalized as follows: (1) Identify a set of MG and DG generalized as follows: (l) Identify a set of MG and DG cells that cover all objects needed in answering the query; cells that cover all objects needed in answering the query; Procedure RangeQueryProcessing(QueryArea rgn)

- 1. *QueryAnswer* = \emptyset ;
- *2. Senrch,/or MG cells whose spnce covernge over-lop ~:i/Ii r-gri.* 2. *Search for MG cells whose space coverage ol'erlap with rgn. Put pointel-s ro such MG cells into n set Set,,; Put pointers to such MG cells into a set Set ^m ;*
- *3. For each MG cell 171 referred iii Set,,, 3. For each MG cellm referred in Set",*

(a) *For ench updnre rr in in* (a) *For each update* II *in* m

i. *If (u. OLoc is inside rgrl), Qz~epAiiswer* + *u;* i. *If(u.OLoc is inside rgl1), QueryAnswer* <-- *u;*

- 4. $Set_{pid} = \emptyset$;
- *5. For ench MG cell iii referred in Set, 5. For each MG cellm referred in Set",*

(a) $Set_{pid} = Set_{pid} \cup m.D_{id}$;

- *6. For ench diskpage d referred iri Set,,d 6. For each disk page d referred in Setpid*
	- (a) *Rend d into merilory;* (a) *Read d into memory;*
	- (b) *For- each entry e iri d* (b) *For each entl)' e in d*
		- i. If *((ldentfiingEiitry(d)* == *CURRENT) nrld* i. *If «(1dentifyingEnrl)'(d)* == *CURRENT) alld* $(d. OLoc is inside rgn)$, $QuenyAnswer \leftarrow d;$
- *7. Returri QuetyAriswer; 7. Return QueryAnswer;*

Figure 8. Processing Range Queries Figure 8. Processing Range Queries

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

The above steps are processed in LUGrid elegantly and The above steps are processed in LUGrid elegantly and efficiently. Figure 8 gives the pseudo code for processing efficiently. Figure 8 gives the pseudo code for processing a *range quejy* Q using LUGrid. Initially, the Q's answer is a *range quely Q* using LUGrid. Initially, the *Q's* answer is set to empty. The algorithm starts by searching for MG cells set to empty. The algorithm starts by searching for MG cells whose space coverage overlap with the query area (Step 2 whose space coverage overlap with the query area (Step 2 in Figure 8). For updates buffered in these MG cells, the in Figure 8). For updates buffered in these MG cells, the algorithm checks whether they are within the query area. algorithm checks whether they are within the query area. Entries within the query area are added to the query an-Entries within the query area are added to the query swer (Step 3 in Figure 8). Then, for MG cells that overlap swer (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). tifiers of their repository cells (Step 4 and *5* in Figure 8). In the case that multiple MG cells share one DG cell, the In the case that multiple MG cells share one DG cell, the page identifier of the DG cell is stored only once to avoid 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* cuted using the algorithm given in Figure 7. For the *current* entry, if its location is inside the query area, it is included in entry, if its location is inside the query area, it is included in the query answer (Step 6 in Figure 8). Finally. the answer the query answer (Step 6 in Figure 8). Finally, the answer set is returned. Section 6 shows that the query processing in LUGrid is 110 efficient. LUGrid is I/O efficient.

5 Cost Analysis 5 Cost Analysis

In this section. we theoretically analyze the costs of up-In this section, we theoretically analyze the costs of dating operations for the proposed techniques. Two unique dating operations for the proposed techniques. Two unique techniques are utilized in LUGrid, namely, *lazy-insertion* and *log.-deletion.* Since both techniques can be applied and *lazy-deletion.* Since both techniques can be applied independently or together. our analysis studies four cases: independently or together, our analysis studies four cases: *~iai~~e approach, loz~l-insertion oizly, la:?-deleti011 only,* and *naive approach, lazy-insertion only, lazy-deletion only,* and *lazy-i~~sertio~zplus lazy-deletion. lazy-insertion plus lazy-deletion.*

The analysis is based on a uniform distribution of mov-The analysis is based on a uniform distribution of ing objects in the two-dimensional space. However, since ing objects in the two-dimensional space. However, since LUGrid adapts to various data distributions, other forms of LUGrid adapts to various data distributions, other forms of object distributions have similar effects as uniform distri-object distributions have similar effects as uniform distribution. In the analysis, we focus on the steady-state where bution. In the analysis, we focus on the steady-state where cell splitting and merging rarely take place. Thus, the cost cell splitting and merging rarely take place. Thus, the cost of splitting and merging is ignored in our cost analysis. As of splitting and merging is ignored in our cost analysis. As a dominating metric, the number of 110 operations is inves-a dominating metric, the number ofl/O operations is investigated for the updating cost. tigated 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 ber of MG cells. In a uniform distribution, MG cells have approximately equal sizes. We define 6 as the percentage of approximately equal sizes. We define *0* as the percentage of updates that a certain object, say *P:* moves from a certain updates that a certain object, say *P,* moves from a certain disk-based cell to another one. Since no 110 cost is involved disk-based cell to another one. Since no I/O cost is involved when buffer updates in memory. we only consider the flush-when buffer updates in memory, we only consider the flushing phase in our analysis. ing phase in our analysis.

Naive approach. The naive approach utilizes none of Naive approach. The naive approach utilizes none of the proposed techniques in update processing. An update the proposed techniques in update processing. An update goes to disk as soon as it arrives to the system, meanwhile 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 110 cost for processing these updates is updates, the total I/O cost for processing these updates is given by Equation I. given by Equation I.

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

In Equation **1,** 2n, is induced by reading and writing 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 pected number of updates that have old entries in other disk cells. For each such update, the auxiliary index structure cells. For each such update, the auxiliary index structure is searched once to get the page p of the old entry. Then, 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, *p* is read and written once to delete the old entry. Finally, the auxiliary index is changed and flushed back to disk to 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 operations are required for each single update that results in a cell change. According to Equation 1, the *average* cost for a cell change. According to Equation I, the *average* cost for one update under the naive approach is given by Equation 2: one update under the naive approach is given by Equation 2:

$$
IO_{naive_\alpha v} = 2 + 4\delta \tag{2}
$$

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

object IDS for quickly locating old object entries. Notice object IDs for quickly locating old object entries. Notice that the MDM hash table in LUGrid is not applicable in 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 UO cost flushing MG cell. For *lazy-insertion only,* the total I/O cost for flushing an MG cell is given by Equation 3. for flushing an MG cell is given by Equation 3.

$$
IO_{L1} = 2 + 4(n_u * \delta)
$$
 (3)

In Equation 3, 2 is introduced by reading and writing the 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 same as that of Equation l. The expected number of updates in an MG cell is U/N_m . So, according to Equation 3, the *aver-age* updating cost for one object is: *average* updating cost for one object is:

$$
IO_{LI_\text{avg}} = \frac{2N_m}{U} + 4\delta \tag{4}
$$

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

$$
IO_{L1_diff} = 2N_m(\frac{1}{U_1} - \frac{1}{U_2})
$$
 (5)

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

For the case of *lazy-deletion only,* the overall update 110 For the case of *lazy-deletion only,* the overall update I/O cost consists of the following: (1) Reading the repository cost consists of the following: (1) Reading the repository DG cell to memory, (2) Writing the DG cell back to disk, DG cell to memory, (2) Writing the DG cell back to disk, and (3) If the cleaner is invoked, reading and writing the and (3) If the cleaner is invoked, reading and writing the cleaned DG cell. Hence, if let c be the clean interval, the expected 110 cost per update is given by: 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-*Lazy-insertion plus lazy-deletion. Combining *lazyinsertion* 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. There-ations and cost of periodical cleaning are required. fore, the average updating cost is simply given by: fore, the average updating cost is simply given by:

$$
IO_{L1\&L1\...ovg} = \frac{2}{n_u} + \frac{2}{c} = \frac{2N_m}{U} + \frac{2}{c}
$$
 (7)

6 Performance Evaluation 6 Performance Evaluation

In this section, we evaluate the performance of LUGrid In this section, we evaluate the performance of LUGrid with various settings. LUGrid is compared with the Fre-with various settings. LUGrid is compared with the Frequently Updated R-tree (FUR-tree, for short) [9] in both quently 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 Table 1. Experiment Parameters and Values

update processing and query processing. FUR-tree mod-update processing and query processing. FUR-tree ifies the original R-tree by processing object updates in a ifies the original R-tree by processing object updates in a bottom-up fashion. For FUR-tree. we implement the *global* bottom-up fashion. For FUR-tree, we implement the *global bottonr-up* approach as proposed in [9] and tune its parame-*bottom-up* approach as proposed in [9J and tune its parameters to achieve best performance. To make our comparison ters to achieve best performance. To make our comparison fair, we make the following two changes. (1) The first two fair, we make the following two changes. (I) The first two levels of the FUR-tree are assumed to reside in memory and 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 the first two R-tree levels is about 8 pages, approximately equals to the size of MG without the buffer for object up-equals to the size of MG without the buffer for object updates. (2) Whenever LUGrid consumes some amount of dates. (2) Whenever LUGrid consumes some amount of memory, we give FUR-tree a same size buffer that is main-memory, we give FUR-tree a same size buffer that is maintained in a *Least Rece17tly Used* (LRU) manner. FUR-tree tained in a *Least Recently Used* (LRU) manner. FUR-tree makes use of the buffer when accessing leaf pages of the makes use of the buffer when accessing leaf pages of the R-tree and the pages of the auxiliary index on object identi-R-tree and the pages of the auxiliary index on object identifiers. fiers.

In all experiments. we collect the results of LUGrid when In all experiments, we collect the results ofLUGrid when the system becomes stable. In this case, cell splitting or the system becomes stable. In this case, cell splitting or merging rarely happens. This does not affect the quality merging rarely happens. This does not affect the quality of the results because we are interested in the updating and of the results because we are interested in the updating and querying performance of LUGrid. as well as resource con-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 maximum number of updates that one MG cell can buffer is set to the number of entries that one disk page can con-is set to the number of entries that one disk page can contain. In all experiments, the clean interval is set to 50. tain. In all experiments, the clean interval is set to 50.

All the experiments are conducted on an Intel Pentium All the experiments are conducted on an Intel Pentium IV machines with CPU 3.2GHz and 512MB RAM. In all IV machines with CPU 3.2GHz and 512MB RAM. In all experiments, 1,000,000 objects are moving inside a space 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 num-tinuously moving with given velocities. We count the ber of object updates in *cycles,* each cycle takes I0 seconds. ber of object updates in *cycles,* each cycle takes IO seconds. In each cycle, a certain ratio of objects report their new lo-In each cycle, a certain ratio of objects report their new cations by issuing updating requests. We conduct experi-cations by issuing updating requests. We conduct experiments using various disk page sizes and various MG buffer ments using various disk page sizes and various MG buffer sizes. To test the adaptation to object distributions, exper-sizes. To test the adaptation to object distributions, iments are carried out under both uniform distribution and iments are carried out under both uniform distribution and normal distributions. We use $Normal(\mu, \sigma)$ to denote a normal distribution with mean μ (miles) and variance σ (miles). Three types of distributions are used in our experiments, Three types of distributions are used in our experiments, namely, *U1~ifon71, Nor177a1(500, 200)* and *Nor-tna1(500, 100).* namely, *Uniform, Normal(500, 200)* and *Nonnal(500, JOO).* We generate the original objects and consequent updates in We generate the original objects and consequent updates in a way similar to GSTD [22]. Various parameters for our a way similar to GSTD [22]. Various parameters for our

Figure 9. Number of DG cells Figure 9. Number of DG cells

Figure 10. Number of MG cells Figure 10. Number of MG cells

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

6.1 Resource Utilization 6.1 Resource Utilization

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

Figure 11. Size of the Miss-Deletion Memo

Figure 1 I gives the size of the *Miss-Deletion Meino* Figure II gives the size of the *Miss-Deletion Memo* with various object velocities (i.e., 10, 50, 100, and 500 with various object velocities (i.e., 10, 50, 100, and 500 miles/hour). For each studied velocity, we increase the ra-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 ra-30%. In Figure II, the size of MDM is expressed as the ratio between the number of MDM entries to the total number tio between the number ofMDM entries to the total number of objects. In all cases: the number of MDM entries is less 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 velocity increases, the size of MDM gets larger. The main reason is that when objects move with a higher velocity, reason is that when objects move with a higher velocity, more objects will move out of their original cells. Conse-more objects will move out of their original cells. Consequently, more MDM entries are needed to hold information quently, more MDM entries are needed to hold information of these cell-changing objects. However, the update ratio of of these cell-changing objects. However, the update ratio of objects does not affect the size of MDM. The reason is that objects does not affect the size of MDM. The reason is that the number of MDM entries is not determined by the total the number of MDM entries is not determined by the total number of updates. Instead, the number of MDM entries is number of updates. Instead, the number of MDM entries is determined by the number of cell-changing objects in each determined by the number of cell-changing objects in each flushing period. The old entries in a cell are deleted when flushing period. The old entries in a cell are deleted when the cell is flushed again. The deletion of old entries reduces 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. can easily fit in main memory.

6.2 Updating Performance 6.2 Updating Performance

In this section, we study the update performance of LU-In this section, we study the update performance of LU-Grid and compare it with FUR-tree. For LUGrid, we use Grid and compare it with FUR-tree. For LUGrid, we use different sizes of MG buffers. Specifically, the MG buffer is 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 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 110s when the 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 up-per cycle. As can be seen from the figure, for different date ratios. LUGrid outperforms the FUR-tree consistently. date ratios, LUGrid outperforms the FUR-tree consistently. The update costs for LUGrid range from 20% to 50% of that

Figure 12. Update cost vs. No. of updates

Figure 13. Update cost vs. obj. distribution Figure 13. Update cost vs. obj. distribution

for the FUR-tree. The efficiency in updates in LUGrid with for the FUR-tree. The efficiency in updates in LUGrid with 070 size buffer comes solely from the lazy-deletion tech-0% size buffer comes solely from the lazy-deletion technique. When the MG buffer becomes larger, the update cost nique. When the MG buffer becomes larger, the update cost becomes lower because more updates are flushed to disk at becomes lower because more updates are flushed to disk at one time by lazy-insertion. one time by lazy-insertion.

Figure 13 compares the update costs under various ob-Figure 13 compares the update costs under various object distributions. LUGrid exhibits almost stable update ject distributions. LUGrid exhibits almost stable update performance independent of object distributions. This is performance independent of object distributions. This is because the update cost of LUGrid is determined primar-because the update cost of LUGrid is determined primarily by the flushing frequency, while the form of object ily by the flushing frequency, while the form of object distribution does not dramatically affect the flushing fre-distribution does not dramatically affect the flushing frequency. However, the FUR-tree incurs larger update cost quency. However, the FUR-tree incurs larger update cost when object distribution is skewed. The main reason is that when object distribution is skewed. The main reason is that when more objects are clustered together, the R-tree con-when more objects are clustered together, the R-tree contains more nodes with small *Miizimal Bouizdirzg Recrnngles* tains more nodes with small *Minimal Bounding Rectangles* (MBRs). Therefore, an object is more likely to move out (MBRs). Therefore, an object is more likely to move out of its MBR quickly and invalidates the bottom-up updating of its MBR quickly and invalidates the bottom-up updating technique. technique.

Figure 14 demonstrates the effect of the object velocity, Figure 14 demonstrates the effect of the object velocity, where objects are moving with various velocities (10, 50, where objects are moving with various velocities (10, 50,

Figure 14. Updating cost vs. obj. velocity Figure 16. Query cost vs. query size Figure 14. Updating cost vs. obj. velocity Figure 16. Query cost vs. query size

Figure 15. Query cost vs. obj. distribution Figure 15. Query cost VS. obj. distribution

I00 and 500 mileshour). As shown in Figure 14, when ob-100 and 500 miles/hour). As shown in Figure 14, when ject velocity increases, the FUR-tree incurs a growing UO ject velocity increases, the FUR-tree incurs ^a growing *va* overhead due to updates. This is because with a larger ve-overhead due to updates. This is because with a larger velocity, an object moves out of the MBR of its original node locity, an object moves out of the MBR of its original node more frequently, and voids the endeavor of the bottom-up more frequently, and voids the endeavor of the bottom-up update. In contrast, for LUGrid, the I/0 from updates is not update. In contrast, for LUGrid, the *va* from updates is not affected by object velocities. This is because LUGrid does affected by object velocities. This is because LUGrid does not delete old entries when updating, so objects moving out not delete old entries when updating, so objects moving out of their original cells do not affect the performance. of their original cells do not affect the performance.

6.3 Querying Performance 6.3 Querying Performance

In this section, we study the querying performance of In this section, we study the querying performance of LUGrid. We focus on the processing of range queries as it LUGrid. We focus on the processing of range queries as it is one of the most important types of spatial queries. In our is one of the most important types of spatial queries. In our experiments, range queries are specified as squares and uni-experiments, range queries are specified as squares and uniformly distributed in space. Figure 15 compares the query-formly distributed in space. Figure 15 compares the ing costs with respect to object distributions. In this exper-ing costs with respect to object distributions. In this experiment, each query covers I % of the whole space. The ex-iment, each query covers I% of the whole space. The experiment shows that under all object distributions, LUGrid periment shows that under all object distributions, LUGrid

is similar to FUR-tree in processing range queries. Both is similar to FUR-tree in processing range queries. Both FUR-tree and LUGrid are slightly affected by object dis-FUR-tree and LUGrid are slightly affected by object tribution. Figure 16 gives the effect when different sizes tribution. 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 query-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.

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

References References

- [I] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving [I] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *PODS,* pages 175-1 *86,* May 2000. Points. In *PODS,* pages 175-186, May 2000.
- (21 V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large [2] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In Proc. of the Conf. on *Itit~ovnri~le Dnto Systems Research, CIDR,* 2003. *Innovative Data Systems Research, CIDR,2003.*
- [3] H. D. Chon, D. Agrawal. and A. E. Abbadi. Storage and [3] H. D. Chon, D. Agrawal. and A. E. Abbadi. Siorage and Retrieval of Moving Objects. In *Mobile Dotn Morzngeinent,* Retrieval of Moving Objects. In *Mobile Data Management,* pages 173-1 *84,* Jan. 2001. pages 173-184, Jan. 2001.
- Continuously Moving Queries on Moving Objects in a Mo- Generation of Spatiotemporal Datasets. In *SSD,* 1999. Continuously Moving Queries on Moving Objects in a Mo-[4] B. Gedik and L. Liu. MobiEyes: Distributed Processing of bile System. In *EDBT, 2004.*
- [5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial *ICDE,* 2006. [5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD, 1984.*
- cient B+-Tree Based Indexing of Moving Objects. In *VLDB.* in Spatio-temporal Databases. In *ICDE,* 2005. cient B+-Tree Based Indexing of Moving Objects. In *VLDB.* 2004. 2004.
- [7] G. KolIios, D. Gunopulos, and V. J. Tsotras. On Indexing [7] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *PODS,* 1999. Mobile Objects. In *PODS. 1999.*
- [8] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions [8] D. Kwon, S. Lee. and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile* of Moving Objects Using the Lazy Update R-tree. In *Mobile Dnrn Mniingemenr, MDM,* 2002. *Data Management, MDM,2002.*
- [9] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting [9] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB,* 2003. *VLDB.2003.*
- [10] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatiotemporal Access Methods. *lEEE Dnrn Engirieering B~rlletiri,* temporal Access Methods. *IEEE Data Engineering Bulletin,* 26(2), 2003. 26(2). 2003.
- [I I] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable [II] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Incremental Processing of Continuous Queries in Spatiotemporal Databases. In *SIGMOD,* 2004. temporal Databases. In *SIGMOD. 2004.*
- [I21 J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid [12] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. File: An Adaptable, Symmetric Multikey File Structure. *TODS,* 9(1), 1984. *TODS,* 9(1),1984.
- (131 J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Ef-[13] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient lndex for Predicted Trajectories. In *SICMOD.* pages ficient Index for Predicted Trajectories. In *SIGMOD.* pages 637-646,2004. 637-646,2004.
- [I41 K. Porkaew, 1. Lazaridis, and S. Mehrotra. Querying Mobile [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. 78. Redondo Beach, CA. July 2001.
- [15] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and [15] S. Prabhakar. Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *lEEE Trnrisnc~ions on Cori7puters:* on Moving Objects. *IEEE Transactions on Computers.* 51(10):1 124-1 140, 2002. 51(10):1124-1140.2002.
- [161 S. Saltenis and C. S. Jensen. Indexing of Moving Objects for [16] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE,* 2002. Location-Based Services. In *ICDE, 2002.*
- [I71 S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. [17] S. Saltenis, C. S. Jensen, S. T Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In Indexing the Positions of Continuously Moving Objects. In *SIGMOD,* 2000. *SIGMOD,2000.*
- [I81 Z. Song and N. RoussopouIos. Hashing Moving Objects. In [18] Z. Song and N. RoussopouIos. Hashing Moving Objects. In *Mobile Data Manngeiiiei~t,* 2001. *Mobile Data Manage111elll, 2001.*
- [I91 Z. Song and N. Roussopoulos. SEB-tree: An Approach to [19] Z. Song and N. Roussopoulos. SEB-tree: An Approach to lndex Continuously Moving Objects. In *Mobile Dntn Man-*Index Continuously Moving Objects. In *Mobile Data Man*agement, MDM, pages 340-344, Jan. 2003.
- [20] Y. Tao, D. Papadias, and J. Sun. The TPR"-Tree: An [20] Y. Tao, D. Papadias, and J. Sun. The TPR"-Tree: An Optimized Spatio-temporal Access Method for Predictive Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB,* 2003. 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. 41(3),1998.
- [4] B. Gedik and L. Liu. MobiEyes: Distributed Processing of [22] **Y.** Theodoridis, J. R. Silva, and M. A. Nascimento. On the [22] Y. Theodoridis. J. R. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *SSD, 1999.*
	- bile System. In *EDBT*, 2004. **In** *EDBT*, 2004. **In** *EDBT*, 2004. **In** *EDBT*, 2004. **In EDBT**, 2004. **I** *ICDE,2006.*
- Searching. In *SIGMOD*, 1984. **[24] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scal-** [24] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scal-[6] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Updale Effi- able Processing of Continuous K-Nearest Neighbor Queries [6] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Effiable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE, 2005.*