2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00238

The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads

Jaewoo Shin
Department of Computer Science
Purdue University
shin152@purdue.edu

Jianguo Wang
Department of Computer Science
Purdue University
csjgwang@purdue.edu

Walid G. Aref
Alexandria University, Egypt, and
Purdue University
aref@purdue.edu

***Abstract*—Many applications require update-intensive workloads on spatial objects, e.g., social-network services and shared-riding services that track moving objects (devices). By buffering insert and delete operations in memory, the Log Structured Merge Tree (LSM) has been used widely in various systems because of its ability to handle insert-intensive workloads. While the focus on LSM has been on key-value stores and their optimizations, there is a need to study how to efficiently support LSM-based *secondary* indexes. We investigate the augmentation of a main-memory-based memo structure into an LSM secondary index structure to handle update-intensive workloads efficiently. We conduct this study in the context of an R-tree-based secondary index. In particular, we introduce the LSM RUM-tree that demonstrates the use of an Update Memo in an LSM-based R-tree to enhance the performance of the R-tree's insert, delete, update, and search operations. The LSM RUM-tree introduces novel strategies to reduce the size of the Update Memo to be a light-weight in-memory structure that is suitable for handling update-intensive workloads without introducing significant overhead. Experimental results using real spatial data demonstrate that the LSM RUM-tree achieves up to 9.6x speedup on update operations and up to 2400x speedup on query processing over the existing LSM R-tree implementations.**

I. Introduction

In recent years, massive amounts of location data have been generated continuously from mobile devices, social media, and shared-riding services. As devices or objects move in space, they update their locations and expect to have responsive services (e.g., getting weather emergencies and location-targeted advertisements). From a system's perspective, it is challenging to efficiently handle update-intensive location workloads, and answer queries with low latency.

A widely-used approach for write-intensive workloads is the Log-Structured Merge tree (or LSM, for short) [1]. The main idea of LSM is to buffer data ingestion in memory, and then periodically flush the buffers into disk. By the generic framework for LSM *secondary*-key indexes [2], the LSM R-tree has been proposed to handle write-intensive spatial workloads. In an LSM secondary index (e.g., LSM R-tree), determining the most recent state (e.g., the current location) of an object is challenging because a secondary key (e.g., location) is not able to uniquely identify the object and shared by multiple objects at different times. For example, the LSM R-tree can be used in a social media application that tracks the current location of users (e.g., smartphones) and wants to send targeted advertisements to users. In this application, the users continuously *update* their locations into the memory layer of the LSM R-tree and the advertisers *query*, e.g., a range search, the tree. Because an LSM index is an out-of-place update structure, it produces a new object for the update. Note that there is only one *current* location for users, and all their previous locations are outdated. Thus, querying the LSM R-tree may contain outdated locations of the users (i.e., false-positive) and an additional processing is needed to validate each of the results.

Alsubaiee et al. [2] address this issue by using an *eager* strategy, where an additional data structure, namely, a deleted-key B^+-tree, is associated with an in-memory R-tree to store the deleted objects' keys. This indicates that an old version of the object is deleted, and a new one is inserted. The deleted-key B^+-tree is also flushed to disk along with the corresponding R-tree. This scheme induces extra overhead due to the need to maintain and access the deleted-key B^+-tree, and thus affects negatively both the update and query performance.

Another solution to address the issue is to use a *validation* strategy [3], where the deleted-key B^+-tree that is coupled with the R-tree is removed. Instead, a primary key index (i.e., a B^+-tree holding the primary keys) with timestamps is used to validate search results. The decoupled primary key index with timestamps helps avoid the extra maintenance cost, and shows improvement in update performance. However, this approach induces extra overhead in the search operation to validate the most recent state of an object. While enhancing the update performance, this approach worsens the search in contrast to the Eager strategy.

In this paper, we introduce the LSM RUM-Tree, an LSM-based R-tree that utilizes an in-memory Update Memo (UM, for short) to support update-intensive spatial workloads while having excellent search performance. The LSM RUM-Tree maintains multiple R-trees as previous works and a UM component that resides only in memory. To control the size of UM, we introduce four UM cleaning strategies that not only reduce the consumed memory space of UM, but also improve the overall performance. The experimental results demonstrate that the LSM RUM-tree achieves up to 9.6x speedup on update operations and up to 2400x speedup on search performance over state-of-the-art LSM R-tree implementations.

The rest of this paper proceeds as follows. Section II presents background material and the related works. Section III introduces the LSM RUM-tree, Section IV presents cleaning strategies for the LSM RUM-tree, Section V presents extensive experiments of the LSM RUM-tree in comparison to previous approaches, and Section VI contains concluding remarks.

II. Background and Related Work

In this section, we summarize two prior approaches to handle spatial workloads in the LSM R-tree. The goal of this paper is to improve the performance of the LSM R-tree with the use of the Update Memo (UM, for short). Thus, it is important to understand how the LSM R-tree works for each operation and how UM can accelerate the update procedure for the R-tree.

A. The LSM R-tree

The LSM R-tree [2] is a spatial index to handle location data efficiently and benefits from the LSM mechanism. By applying a generic framework for secondary index LSM-ification [2], the LSM R-tree is an optimized secondary index for write-intensive spatial data workloads. To handle frequent updates in the LSM R-tree index, two strategies, Eager [2] and Validation [3], have been proposed. Traditionally, updating a secondary key value, e.g., updating the location of an object, requires maintaining both an LSM *primary* and *secondary* indexes to ensure consistency among both indexes. However, to simplify the presentation of the secondary index, we only highlight the maintenance on the LSM R-tree.

1) Eager Strategy for LSM Secondary Indexes: Alsubaiee et al. [2] introduce to LSM an additional deleted-key B$^+$-tree for each R-tree to make the LSM R-tree consistent. The deleted-key B$^+$ tree stores the primary keys (the object identifiers) of the deleted/updated objects to validate the state of an object given a query. The in-memory R-tree and its corresponding deleted-key B$^+$-tree are tightly coupled, and they are flushed to disk together as a component. Although the deleted-key B$^+$-tree buffers the delete operations in the memory layer, the extra maintenance cost and disk I/Os during a search operation degrade the overall search performance.

Assume that we have an object, say $o = \langle Loc, O_{id}, ...\rangle$, where Loc indicates the location of o, and O_{id} is o's object identifier. Because the R-tree indexes locations, Loc is the secondary key for the R-tree index while O_{id} is a foreign key that points to the primary key of the object in the primary index, where the latter may contain other attributes that describe o. To insert o into the LSM R-tree, we add o into the in-memory R-tree, and do not need to access the deleted-key B$^+$-tree. To delete an entry $o = \langle Loc, O_{id}\rangle$ from the LSM R-tree, we perform the following steps: (1) Remove $o = \langle Loc, O_{id}\rangle$ from the in-memory R-tree index, if it exists, and (2) Invalidate the outdated os in the disk components by adding O_{id} into the deleted-key B$^+$-tree in memory. To update a spatial object o inside the LSM R-tree, we perform the following: Delete the old object $o = \langle Loc_{old}, O_{id}\rangle$ (by following the delete procedure above), and then insert the new object $o = \langle Loc_{new}, O_{id}\rangle$ into the LSM R-tree (by following the insert procedure above). To search the LSM R-tree (e.g., find all objects around $Loc = (2, 2)$), all the R-trees, whether in memory or in disk, are searched to find candidate query results because all the R-trees could have qualifying objects. Then, the O_{id} of each candidate will need to be searched against the existing deleted-key B$^+$-trees, and will be reported as output only if O_{id} does not exist in any of the deleted-key B$^+$-trees of a newer component than the candidate's component.

2) Validation Strategy for LSM Secondary Index: The Validation strategy [3] addresses the update overhead of the Eager strategy. It avoids using the deleted-key B$^+$-tree and uses the *primary key index* to validate the query results. The Validation strategy adds a timestamp in each object in both the *primary key* and *secondary* indexes. For inserts, it inserts an object as $\langle Loc, O_{id}, ts\rangle$ into the R-tree where ts is a timestamp for the insert using local wall-clock time. For deletes, the Validation strategy only inserts a control entry into the primary key index to indicate that the object is deleted. Thus, it simplifies the delete or update procedures over the *Eager* strategy. The update is the delete of an old object followed by the insert of a new object. The query performance of the *Validation* strategy still suffers due to the needed validation steps using the primary index for direct validation or primary key index for timestamp validation.

B. The RUM-Tree: The R-tree with Update Memo

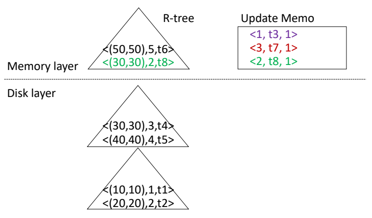
The RUM-tree [4], [5] is another approach to handle update-intensive spatial workloads. It augments an R-tree with an Update Memo (UM) structure. The RUM-tree is a disk-based R-tree index that has a UM in memory. It maintains a global timestamp counter, and marks each object to show a temporal relationship among the objects. Each UM entry is of the form: $\langle O_{id}, ts, cnt\rangle$, where ts is the timestamp of the most-recent update to O_{id} in the RUM-tree, and cnt is the number of obsolete versions of Object O_{id} in the index. By handling the insert and update operations in UM, the RUM-tree achieves significantly lower update cost without having big penalty during search. The UM has several cleaning strategies for removing the obsolete entries from both the R-Tree and the UM to restrict the latter's size. As a result, the RUM-tree shows improved performance on updates over the traditional R-tree [4]–[6].

III. The LSM RUM-tree

We introduce the new LSM RUM-tree; an LSM R-tree augmented with an Update Memo. The goal of the LSM RUM-tree is to efficiently handle update-intensive spatial workloads, and improve search performance. From Section II-A, existing strategies for LSM secondary indexes have degraded update and search performances. To address this issue, in the LSM R-tree we introduce an Update Memo within the LSM R-tree to simplify the processing of deletes and updates in the memory layer, and make disk-based search cost-efficient.

A. The Update Memo Structure (UM)

Refer to Figure 1a for illustration. In the LSM RUM-tree, UM is based on a hash map that resides in memory. The key



(a) Structure Overview

ts	oper.	object
t1	INSERT	$\langle(10,10),1\rangle$
t2	INSERT	$\langle(20,20),2\rangle$
t3	DELETE	$\langle 1\rangle$
t4	INSERT	$\langle(30,30),3\rangle$
t5	INSERT	$\langle(40,40),4\rangle$
t6	INSERT	$\langle(50,50),5\rangle$
t7	DELETE	$\langle 3\rangle$
t8	UPDATE	$\langle(30,30),2\rangle$

(b) Example operations

Fig. 1: The LSM RUM-tree

to UM is an object identifier, and the value is a pair of recent timestamp and obsolete objects counter. UM has several roles. It keeps track of the deletes and updates ingested into the LSM RUM-tree. Because UM resides only in memory, the update to an UM entity takes constant time. Also, UM validates candidate objects resulting from a search operation. As in Section II-A, previous strategies need to perform extra work to answer a query correctly. The *Validation* strategy needs to validate output candidates of a search by comparing the output candidates against *the primary key index* in a way similar to that of the *Eager* strategy. Because UM is light-weight and resides entirely in memory, we expect enhancement in search performance over the previous approaches.

From Figure 1a, each object in the LSM RUM-tree is represented as a triplet $\langle Loc, O_{id}, ts\rangle$, where Loc is the object's location, O_{id} is the object identifier, and ts is a global timestamp counter – an integer value incremented by 1 for each insert, delete, or update operation. The higher the value of ts, the fresher the object in the index. An entry e in UM is represented by a triplet $\langle O_{id}, ts, cnt\rangle$, where cnt is a counter of the number of outdated objects with the same O_{id} in the LSM RUM-tree. By default, the entry $e = \langle O_{id}, ts, \mathbf{1}\rangle$ is first inserted into UM when deleting or updating an object with no existing entry in UM. Otherwise, if O_{id} exists in UM in an entry, say e, from a previous delete or update, we increment $e.cnt$ by 1. When an obsolete object is found and is removed from the LSM RUM-tree, we decrement the object's $e.cnt$ by 1. Entry e is removed from UM when $e.cnt = 0$, i.e., there are no obsolete entries for O_{id} currently in the LSM RUM-tree.

B. Lazy Maintenance in UM

UM buffers deletes and updates in memory. Below, we discuss how the delete, update, and search operations in the LSM RUM-tree utilize UM.

1) Insert: During an insert, first we increment the global timestamp counter by 1Note that the timestamp counter is an integer that gets incremented for each insert, delete, or update operation. Once we read the global timestamp counter, say ts, a new object $o_{new} = \langle Loc, O_{id}, ts\rangle$ is added to the in-memory R-tree UM tracks only object deletes and updates and performs no special actions for inserts. The goal of the LSM RUM-tree is to support both update- and insert-intensive spatial workloads. If UM treats inserts as in the case of the original RUM-tree [4], [5], the UM size grows linearly with the number of objects for insert-intensive workloads. This would result in significant memory-space overhead. In the LSM RUM-tree, avoiding to maintain UM upon object inserts prevents the unnecessary growth in UM size. In Section III-C, we discuss in detail how we validate an object that has no entry in UM.

Algorithm 1: Delete operation

```
input: $O_{id}$: Object id (primary key)
1 $ts \leftarrow$ timestamp counter++;
2 if entry $e$ for $O_{id}$ exist in $UM$ then
3     $e.ts \leftarrow ts$;
4     $e.cnt$++;
5 else
6     put $e_{new} = \langle O_{id}, ts, 1\rangle$ into $UM$;
```

2) Delete: To delete an object, we only add or modify the object's corresponding UM entry. As in Algorithm 1, if there is an UM entry, say e, for a given O_{id}, we set $e.ts$ field to the global timestamp (Line 3) and increment $e.cnt$ by 1 (Line 4). If there is no such e in UM, we insert $\langle O_{id}, ts, 1\rangle$ into UM. Note that the value of $e.cnt$ corresponds to the number of obsolete copies of a given O_{id} in the R-trees. Naturally, each delete operation makes one additional obsolete object in the R-tree. By tracking ts for freshness and cnt for the number of obsolete objects of O_{id}, we not only have a clear sense of a given object copy in the index whether it is fresh or not, but also have a good grasp of the number of obsolete copies of the object in the LSM RUM-tree. In the example in Figure 1a, "t3 : DELETE $\langle 1\rangle$" and "t7 : DELETE $\langle 3\rangle$" add $\langle 1, t3, 1\rangle$ and $\langle 3, t7, 1\rangle$ into the UM, respectively.

3) Update: To process an update, we check whether or not UM contains an entry, say e, with the same O_{id}. If e exists in UM, we update $e.ts$ to the current timestamp ts and increment $e.cnt$ by 1. If e does not exist, we add a new entry $e_{new} = \langle O_{id}, ts, 1\rangle$ into UM. Then, we add the new object entry $\langle Loc, O_{id}, ts\rangle$ into the in-memory R-tree. Note that update consists of both delete and insert. In the example in Figure 1a, "t8 : UPDATE $\langle(30,30),2\rangle$" adds the entry $\langle 2, t8, 1\rangle$ into the UM and inserts the new object $\langle(30,30), 2, t8\rangle$ into the in-memory R-tree. Notice that if there is another UPDATE for the same object at t9, we update the UM entry from $\langle 2, t8, 1\rangle$ to $\langle 2, t9, 2\rangle$ and insert the new object with the timestamp $t9$ into the R-tree.

C. Search

The LSM RUM-tree does not require a disk I/O except for accessing LSM R-trees on disk. Algorithm 2 illustrates how to utilize the UM to validate search results that are returned from the LSM R-tree. First, we check whether a candidate O_{cand} from the R-tree search is fresh or not. If UM does not contain an entry with the same O_{id}, then O_{cand} is fresh, and is part of the search results. If there is an entry, say e, with the same O_{id}, then the ts field of the candidate is compared with $e.ts$ from UM. If $O_{cand}.ts < e.ts$, then the candidate is

Algorithm 2: Validation with Update Memo

```
input : $candidates$: The list of candidates for a search
output: $results$: The list of results of a search
1 for $O_{cand} \leftarrow candidates$ do
2     $O_{id} \leftarrow O_{cand}.O_{id}$;
3     if entry $e$ for $O_{id}$ exists in $UM$ then
4         if $O_{cand}.ts == e.ts$ then
5             $results$.insert($O_{cand}$);
6     else
7         $results$.insert($O_{cand}$);
8 return $results$
```

obsolete, and is discarded. Observe that there is no case where $O_{cand}.ts > e.ts$ because we always maintain a UM entry to reflect the most recent timestamp of an object. Also, if there is a fresh object (e.g., "t5 : INSERT $\langle(40, 40), 4\rangle$" in Table 1b) and there is no other updates on the same object, there is no UM entry with the same $O_{id} = 4$.

Because UM is based on a hash map, massive amounts of delete/update operations will increase the size of UM. Also, as the size of the hash map increases, its lookup performance deteriorates due to the large number of entries in each hash map bucket and these result in degrading the search performance. For these reasons, in the next section, we introduce LSM-aware UM cleaning strategies to bound the size of UM and improve search performance.

IV. LSM-Aware UM Cleaning Strategies

As in Section III-A, an entry e in UM will be removed when the field *cnt* hits 0. Thus, our focus is on how to decrease $e.cnt$ for each operation running on the LSM RUM-tree. We present 4 cleaning strategies for the LSM RUM-tree: (1) Buffered Cleaning, (2) Vacuum Cleaning, (3) Clean Upon Flush, and (4) Clean Upon Merge. The first two are for UM cleaning through the in-memory R-tree. In contrast, the remaining two are for UM cleaning through the disk-side R-trees.

A. Buffered Cleaning

When an application uses the LSM RUM-tree as a secondary index to handle continuous update-intensive workloads (e.g., tracking moving objects continuously), there is a high chance that a node of the in-memory R-tree has multiple obsolete objects. To clean UM and those in-memory R-tree nodes, we introduce the Buffered Cleaning strategy that cleans an in-memory R-tree node based on the accumulated updates inside this node. This is particularly applicable for hot-spot cleaning. We maintain an *update counter* for each node of the in-memory R-tree. When the LSM RUM-tree buffers an update operation, the update counter on the node is incremented by 1. Once the counter hits some threshold, we remove the obsolete objects from the node and clean the UM entry by decrementing its *cnt* value by 1 for each of the removed obsolete objects. When a new R-tree node is created (e.g., due to a node split) or when Buffered Cleaning cleans a specific node, we set the node's update counter to 0. Notice that the threshold for the update counter is a variable that decides the frequency of invoking the Buffered Cleaning strategy. The lower the threshold is set, the more frequent the Buffered Cleaning strategy is invoked.

Algorithm 3: Node and Update Memo Cleaning

```
input : $objects$: The list of objects in a node
1 for $O \leftarrow objects$ do
2     if entry $e$ for $O_{id}$ exists in $UM$ then
3         if $O.ts < e.ts$ then
4             remove $O$ from $objects$;
5             $e.cnt--$;
6             if $e.cnt == 0$ then
7                 remove $e$ from the Update Memo;
```

Algorithm 3 illustrates how to clean an R-tree node and its corresponding entries in UM. While iterating over objects in a node, we discard the obsolete objects from the node by comparing the object's timestamp with the timestamp in the object's entry in UM (Lines 2-4). If an obsolete object is removed, we decrement by 1 the object's corresponding $e.cnt$ entry in UM to track the number of obsolete objects with the same O_{id} (Line 5). If $e.cnt = 0$, there are no obsolete objects having the same O_{id} in the LSM RUM-tree. Thus, we remove e from UM.

B. Vacuum Cleaning

Vacuum Cleaning complements Buffered Cleaning because it targets mostly the hot-spot nodes in the LSM RUM-tree. There are still some cases that Buffered Cleaning cannot handle very well: (1) A node is on a cold-spot so the counter for the Buffered Cleaning does not hit the threshold or (2) Objects in a node have been obsoleted because of updates in other nodes. Not cleaning these cold-spot nodes can result in growing the size of UM and being not able to control it. To make up for the cold-spot nodes not handled by Buffered Cleaning, we introduce Vacuum Cleaning for fair cleaning of in-memory R-tree nodes to bound UM's size.

In Vacuum Cleaning, we maintain a global counter and a vacuum cleaner. The global counter stores the number of updates in the entire LSM RUM-tree, and the vacuum cleaner holds the next leaf node to be cleaned in the in-memory R-tree. Once the global counter hits some threshold by update operations, the vacuum cleaner cleans the next node. The node and UM cleaning are the same as the ones in Algorithm 3. After this node's cleaning is finished, we reset the global counter to 0 and set the vacuum cleaner to the next leaf node.

Buffered and Vacuum Cleaning have several advantages. They help reduce the UM size. Having obsolete objects in the in-memory R-tree leads to unnecessary UM entries. By cleaning the UM, we bound its size. Also, both cleaning

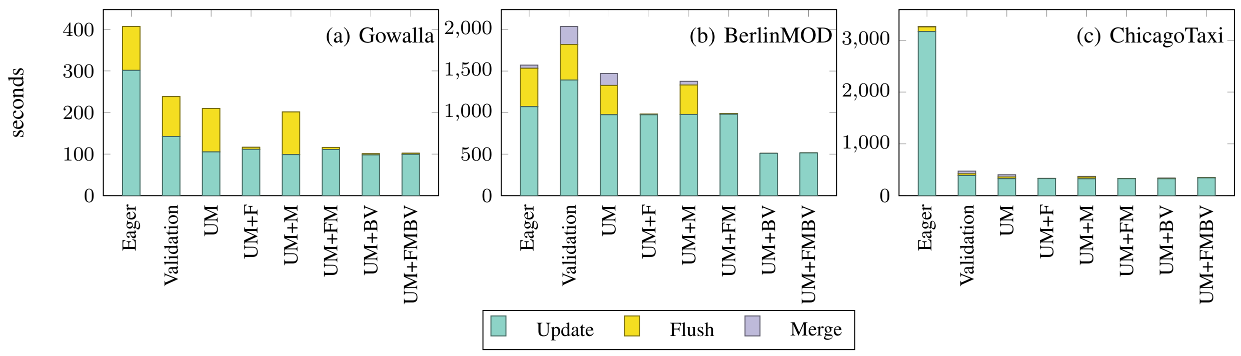


Fig. 2: Comparisons of Update Performance on LSM R-trees

strategies clean the nodes of the in-memory R-tree. Update-intensive workloads cause many obsolete objects in an R-tree node, and this leads to unnecessary R-tree node splits. By cleaning the nodes in the in-memory R-tree, the node remains fresh and avoids unnecessary splits due to being filled with obsolete objects. Also, by cleaning R-tree nodes, we expect to have improved search performance because there should be less false-positive candidates while searching the R-tree.

C. Clean Upon Flush

In Clean Upon Flush, we bring the flushed component up-to-date at the time of flushing without having too much computational overhead. Mainly, we add the R-tree and UM cleaning step just before flushing. When the LSM RUM-tree is flushed into disk, it orders the objects by the Z-order or Hilbert curves [7], [8] to be stored in disk efficiently. For Clean Upon Flush, when the flush operation orders the objects, we check whether the object is obsolete or not by comparing it with the entry in the UM as in Algorithm 3. If the object is obsolete, we discard the object so it does not get flushed to disk and clean the corresponding UM entry. If the *cnt* of the UM entry is 0, we remove the entry from UM.

D. Clean Upon Merge

Because the objects in disk components become obsolete as new updates accumulate into the in-memory component, we introduce Clean Upon Merge to clean the obsolete objects on disk as well as their corresponding UM entries. During LSM merge, multiple disk components are bulk-loaded and are merged into a single disk component. For Clean Upon Merge, we add a validation step as the one in Algorithm 3 on the bulk-loading procedures of the merge operation. When bulk-loading the existing components, we check each object on-the-fly against UM. If an object is obsolete, we discard it. Obviously, we update the UM entries accordingly in a way similar to Clean Upon Flush and the other cleaning strategies.

Clean Upon Flush and Clean Upon Merge are expected to improve the performance in various ways. First, UM size is reduced due to cleaning. Shrinking UM size is important in all cleaning strategies. Moreover, all the objects in a new disk component are fresh right after the time of the flushing or merging. Thus, both can enhance search performance because the size of the new component could be smaller than that without cleaning, specifically in update-intensive workloads.

V. Performance Study

We evaluate the LSM RUM-tree along with its insert, delete, update, and search performance. All the experiments are conducted on a machine running Mac OS 10.15.5 on Intel Core i7 with 2.3 GHz, 16 GB memory, and 512 GB SSD. We use three real datasets, Gowalla [9], BerlinMOD [10], and ChicagoTaxi [11] with millions of points: Gowalla (6.4m), BerlinMOD (56m) and ChicagoTaxi (15m). For the datasets, there are 107k, 2k, and 5.2k unique keys (Object IDs), respectively, along with their locations over time.

We implement the LSM RUM-tree inside AsterixDB [12], and compare the LSM RUM-tree with the existing Eager and Validation strategies already implemented in AsterixDB. The LSM RUM-tree is open-sourced at http://bit.ly/lsmrum. For the LSM RUM-tree, we set the budget of the in-memory R-tree to 256 MB and the page size to 2 KB following [2]. The merge policy is set to the *prefix* policy with Threshold=5. We augment the UM implementation into the already existing LSM R-tree. To study the effect of only UM, we do not use any optimization, such as a bloom filter [13] or range filter [14] in the LSM R-tree since they are orthogonal to the focus of this paper. Experiments run in a single-thread environment.

We use the following notation to refer to the various cleaning strategies. *UM* denotes LSM RUM-tree without any cleaning strategy, *UM+(cleaning strategies)* denotes LSM RUM-tree with combinations of cleaning strategies, where F, M, B, and V refer to Clean Upon **F**lush, Clean Upon **M**erge, **B**uffered Cleaning, and **V**acuum Cleaning, respectively. Due to space limitation, we only present comparisons of the update and search performance.

A. Update Performance

We measure the total execution time to complete the data ingestion of all inserts and updates, excluding setup and data feeding times. Figure 2 gives the update performance on three datasets with respect to update procedures, the time to process a flush operation, and the time to process a merge operation by comparing the LSM RUM-tree mechanisms with the *Eager* and the *Validation* strategies of the LSM R-tree.

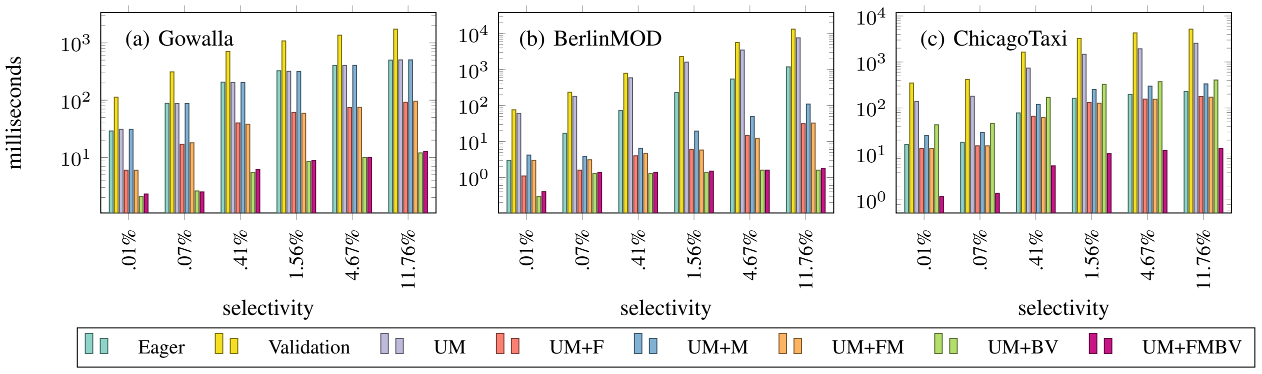


Fig. 3: Comparisons of query performance on LSM R-trees

As in Section IV, the LSM RUM-tree cleaning strategies clean obsolete objects in both the memory and disk layers as well as in the UM entries. In update-intensive workloads, cleaning obsolete objects improves the update performance because it avoids unnecessary node splits in the R-tree and also minimizes the flush/merge operations. From the experiments, observe that the LSM RUM-tree with the various cleaning strategies achieves 3x to 9.6x speedups over the *Eager* strategy and 1.4x to 4x over the *Validation* strategy depending on the different datasets.

B. Search Performance

After data ingestion is completed, we select 100 random query points from each dataset and measure the average time to get the query results. As in [3], the *Validation* strategy is worse than the *Eager* strategy because of the extra validation steps. Figure 3 gives the search (query) performance on each dataset for various query selectivities. Overall, the UM strategy without any cleaning (i.e., *UM*) shows up to 3x better performance than the *Validation* strategy because its in-memory structure does not require I/O for validation. While *UM* is comparable to the *Eager* strategy on the Gowalla dataset (Figure 3a), it gets worse on the other datasets by one order of magnitude as in Figures 3b and 3c. Although *UM* resides in memory, obsolete objects from a query require massive amount of time to validate their states. Therefore, it is essential to have appropriate cleaning strategies as discussed in Section IV. Overall, *UM+FMBV* shows best performance in all datasets. The big improvements are due to the cleaning strategies. The cleaning strategies help reduce the size of UM, clean obsolete objects, and shrink the size of the disk component (i.e., the disk-based R-trees).

VI. Conclusions

In this paper, we introduce the LSM RUM-tree for update-intensive spatial data workloads. We illustrate how to utilize UM in the LSM RUM-tree for delete, update, and search operations. The in-memory UM structure provides efficient validation on query processing as well as simplified update operations. Making UM light-weight is important to be held in memory. To achieve this, we provide four UM cleaning strategies. These strategies not only clean UM entries to shrink its size, but also improve search performance as they also help shrink the size of R-trees, and hence reduce the I/O overheads. The performance study demonstrates that the LSM RUM-tree handles update-intensive workloads efficiently and outperforms the state-of-the-art LSM R-tree implementations.

Acknowledgment

Walid G. Aref acknowledges the support of the National Science Foundation under Grant Numbers III-1815796 and IIS-1910216.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[2] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage management in asterixdb," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 10, pp. 841–852, 2014.
[3] C. Luo and M. J. Carey, "Efficient data ingestion and query processing for lsm-based storage systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 5, pp. 531–543, 2019.
[4] X. Xiong and W. G. Aref, "R-trees with update memos," in *22nd International Conference on Data Engineering (ICDE)*, 2006, pp. 22–22.
[5] Y. N. Silva, X. Xiong, and W. G. Aref, "The rum-tree: supporting frequent updates in r-trees using memos," *The VLDB Journal*, vol. 18, no. 3, pp. 719–738, 2009.
[6] S. Chen, C. S. Jensen, and D. Lin, "A benchmark for evaluating moving object indexes," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 2, pp. 1574–1585, 2008.
[7] D. Hilbert, "Über die stetige abbildung einer linie aufein flächenstück," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.
[8] G. Peano, "Sur une courbe, qui remplit toute une aire plane," *Mathematische Annalen*, vol. 36, no. 1, pp. 157–160, 1890.
[9] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: user movement in location-based social networks," in *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2011, pp. 1082–1090.
[10] C. Düntgen, T. Behr, and R. H. Güting, "Berlinmod: a benchmark for moving object databases," *The VLDB Journal*, vol. 18, no. 6, p. 1335, 2009.
[11] C. D. Portal. (2019) Taxi Trips - 2019. [Online]. Available: https://data.cityofchicago.org/Transportation/Taxi-Trips-2019/h4cq-z3dy
[12] "AsterixDB." [Online]. Available: http://asterix.ics.uci.edu/
[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[14] S. Alsubaiee, M. J. Carey, and C. Li, "Lsm-based storage and indexing: An old idea with timely benefits," in *Second International ACM Workshop on Managing and Mining Enriched Geo-spatial Data (GeoRich)*, 2015, pp. 1–6.