Managing Frequent Updates in R-trees by Semi-Bulkloading*

MoonBae Song and Hiroyuki Kitagawa
University of Tsukuba

Abstract

Managing frequent updates is one of most important issues in many update-intensive applications, e.g., location-aware services, and stream databases. In this paper, we present an R-tree-based index structure which employs semi-bulkloading (SBL) technique for efficiently managing frequent updates from massive moving objects. The basic idea of SBL is to buffer the incoming updates in main-memory buffer, choose a proper subset from the buffer, and then bulk-insert them at once. For this purpose, we devise an efficient update buffer management scheme which provides an effective way to manage the incoming updates in memory-efficient manner. Our experimental results reveal that the proposed approach is far more efficient than previous approaches for managing frequent updates under various settings.

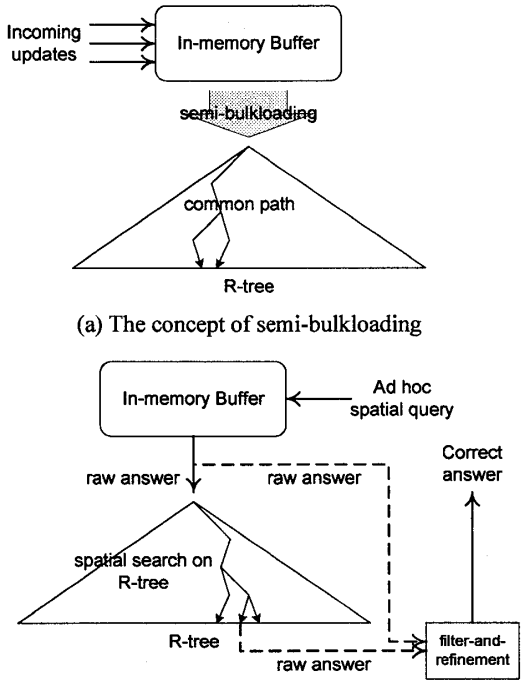
1. Introduction

The growing popularity of update-intensive applications such as location-aware services, monitoring applications, and stream databases has led to a flurry of recent researches on high-performance spatial index that supports high update rates for massive moving objects. One typical example is tracking the location of GPS-enabled mobile device that moves continuously.

Recently, several index structures based on the well-known R-tree [1],[2] have been proposed [3]–[5]. These techniques are mainly motivated by the fact that R-tree suffers from poor performance in update-intensive environments because it handles an update as a delete-insert pair separately. Kwon et al. [3] developed the Lazy Update R-tree to reduce the update cost. By adopting a secondary index on the R-tree, it can perform the update in a bottom-up manner and reduce its cost. Lee et al. [4] improved this idea by adopting an in-memory summary structure to help both updates and queries. More recently, Xiong and Aref proposed the R-tree with update memo (RUM-tree) to reduce the update cost [5]. In this technique, deletions are deferred and performed in a batch manner. For this purpose, the authors proposed a memory data structure called Update Memo that stores recent updates in R-tree.

In this paper, we present an R-tree-based index structure which employs semi-bulkloading (SBL) technique for efficiently managing frequent updates from massive moving objects. The basic idea of SBL is to buffer the incoming updates in main-memory buffer, choose a proper subset from the buffer, and then bulk-insert them at once.

* This research has been supported in part by the Grant-in-Aid for Scientific Research from JSPS.



(a) The concept of semi-bulkloading

(b) Query processing

Figure 1. The proposed index structure

2. Proposed Index Structure

2.1 The basic concept and structures

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

Figure 1 shows the basic concept of the proposed structure. When the buffer is full, the buffered updates are inserted into the disk-resident R-tree simultaneously by exploiting the common path (see Figure 1(a)). For query processing, combining two raw answers from the in-memory buffer and R-tree, and refining the result are needed (see Figure 1(b)).

2.2 Insert, Delete, and Update

The insert/update/delete operations are operated on the in-memory buffer completely. In the propose approach, update is identical to an insert operation. For each incoming update <*oid, p*>, the corresponding buffered entry is updated within the buffer, or a new buffer entry

is inserted into the memory buffer. When there is no enough space in buffer, we perform what we call *Flush* in order to make room for the newly incoming update.

Unlike the traditional R-tree, only an object-id *oid* is needed for Delete() procedure. The reason behind this is that we can find the obsolete entries not by their spatial locations, but instead by their *oid*. This characteristic will make an application simpler by removing maintenance cost for the location information required to delete the obsolete entries.

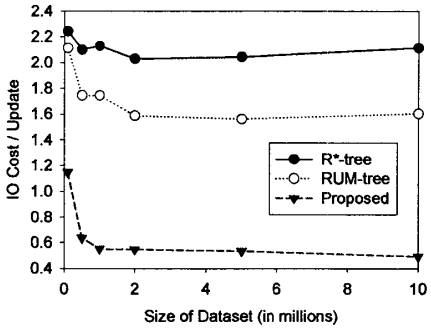
2.3 Semi-bulkloading

Semi-bulkloading is the process of storing the location information on the in-memory buffer to the disk-resident R-tree. It is far differ from the conventional bulk-loading which perform in an off-line manner, because the ultimate goal is not to maximize the quality of the R-tree, but to optimize the update and search costs together. In order to improve the I/O efficiency of semi-bulkloading, we choose a proper subset of OR, which have a strong probability to be inserted to the same leaf node. For brevity, we only consider the simplest policy; that is, first choose a cell that has the maximum cell in $g*g$ grid space, and then group-insert its buffered entries along their shared path.

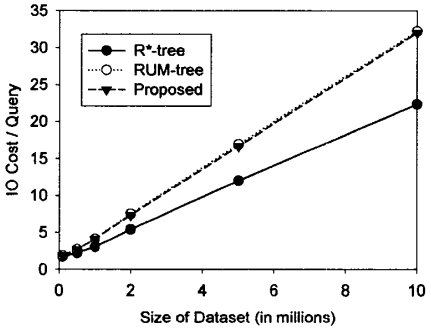
3. Performance Evaluation

In this experiment, we compare the update and query performance of R*-tree (the most powerful variant of R-tree), RUM-tree, and the proposed approach. We use a trace dataset which contains the trajectories of 100k ~ 10,000k moving objects which move in the road network of Oldenburge city. For fair comparison, the number of I/Os for updates and queries – as the primary performance metric – is carefully monitored in every experiment. Our experiments were conducted on Intel Pentium 4, 2GHz and 2GB RAM running on Linux system. In order to verify the query processing power, we run 100k range queries whose side lengths are randomly chosen from [0,0.03]. In all experiments, we set the page size as 4kB, and a page buffer that is 1% of the dataset size is utilized.

Figure 2 shows the experimental result in terms of update and query cost (the number of R-tree node accesses). In general, the proposed approach outperforms the existing techniques in terms of update and query performances under all conditions. In general, R*-tree yields the worst update performance since it is designed for query-intensive setting. In compared to RUM-tree, the overall update performance gain of the proposed approach ranges from 84.29% to 227.14%, which is significant. As the size of dataset increases, the average query costs of all techniques are clearly increased. The average query cost of the proposed approach is slightly better than that of RUM-tree.



(a) Average update cost



(b) Average query cost

Figure 2. Experimental results

4. Concluding Remarks

In this paper, we have investigated the problem of indexing moving objects and managing their frequent updates in update-intensive environments. We proposed an R-tree-based index structure which exploits a small in-memory buffer to buffer, defer, and group incoming updates. With reasonable memory overhead of 1% of database size (or even with the same memory requirement), our approach incurs I/O cost much lower by a factor of 2–3 than the existing techniques. We believe that the proposed approach is orthogonal and it can be applied to other index structures, e.g., B-trees, Quadtrees, and k-d-B-trees.

References

[1] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. of SIGMOD*, 1984, pp. 47–57.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, 1990.

[3] D. Kwon, S. Lee, and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," *Proc. Int'l Conf. Mobile Data Management* (MDM), 2002, pp. 113–120.

[4] Mong-Li Lee, Wynne Hsu, Christian S. Jensen, and Keng Lik Teo. "Supporting Frequent Updates in R-Trees: A Bottom-Up Approach", *Proc. of VLDB*, 2003.

[5] X. Xiong and W. G. Aref, "R-trees with Update Memos," in *Proceedings of International Conference on Data Engineering* (ICDE), 2006.