

Towards Optimal Utilization of Main Memory for Moving Object Indexing

Bin Cui, Dan Lin, and Kian-Lee Tan

School of Computing & Singapore-MIT Alliance,
National University of Singapore
{cuibin, lindan, tankl}@comp.nus.edu.sg

Abstract. In moving object databases, existing disk-based indexes are unable to keep up with the high update rate while providing speedy retrieval at the same time. However, efficient management of moving-object database can be achieved through aggressive use of main memory. In this paper, we propose an *Integrated Memory Partitioning and Activity Conscious Twin-index* (IMPACT) framework where the moving object database is indexed by a pair of indexes based on the properties of the objects' movement - a main-memory structure manages *active* objects while a disk-based index handles *inactive* objects. As objects become active (or inactive), they dynamically migrate from one structure to the other. Moreover, the main memory is also organized into two partitions - one for the main memory index, and the other as buffers for the frequently accessed nodes of the disk-based index. Our experimental study shows that the IMPACT framework provides superior performance.

1 Introduction

Over the years, many data structures have been proposed for moving object indexing [4, 6, 8, 10, 12, 13]. Most of these indexing techniques are disk-based and tend to leave memory exploitation to independent buffering strategies. Main memory is much faster than disk and becomes increasingly voluminous and inexpensive. However, it remains a scarce resource with increasingly sophisticated software and the rapid buildup of huge datasets. Buffer management is effective in gaining faster access to the main memory, and tends to result in the better memory utilization by loading popular paths of the index into memory. Nevertheless, most buffering strategies do not guarantee that the main memory is being utilized in an optimal fashion. Capturing continuous locations of moving objects would entail either performing very frequent updates or recording outdated, inaccurate data. Therefore, the traditional buffering scheme may not work efficiently for the moving object index structures.

To better manage moving object databases, we need to improve the utilization of the main memory. Our solution is based on three observations. First, the moving objects can be classified into two groups, *active* objects and *inactive* objects. Typically, active objects tend to have relatively higher speed and change

velocity frequently, and trigger relatively more updates. Second, we observe that most of the moving objects are inactive. Our analysis of the dataset generated by the City Simulator [5] reveals that only a small portion of users are active at a time, while the majority of the users are inactive. Third, although the active objects constitute a small fraction of the dataset, they incur the fast enlargements of MBRs (minimum bounding rectangle) in the TPR-tree like indexes [10, 12], which results in severe overlaps and degenerates the index performance.

Now, the first and third observations suggest that we should consider active and inactive objects separately. The second observation suggests that active objects can potentially be kept in the main memory. Thus, in this paper, we propose a novel framework, called *Integrated Memory Partitioning and Activity Conscious Twin-indexing* (IMPACT), that combines two mechanisms to make aggressive use of the main memory. First, a pair of indexes manages the moving object database based on the objects' activities - *active* objects are managed by a main memory index, while *inactive* objects are stored in a disk-based structure. As objects become active or inactive, they *dynamically* migrate from one structure to another. Second, the main memory is split into two partitions - one for the main memory index, and the other as buffers for the frequently accessed nodes of the disk-based index. In this way, active objects can be processed efficiently in memory, while there will be less activities occurring on the disk.

To realize the framework, we employ a grid structure for the main memory index and the TPR*-tree [12] as the disk-based structure. We also use an OLRU buffering strategy [9] to cache frequently accessed nodes of the TPR*-tree. For memory partitioning, we devise a scheme to optimally allocate space for the main memory index and buffers. We implemented the proposed framework, and evaluated its performance. Our experimental study shows that the proposed IMPACT framework significantly outperforms the TPR*-tree.

The rest of the paper is organized as follows: in Section 2, we review some related work, including buffering algorithms and existing moving object index structures. In Section 3, we present our proposed IMPACT framework and its realization. Section 4 reports the experimental results. Finally, we conclude our work in Section 5 .

2 Related Work

In this section, we shall first review buffering algorithms, and then look at some index structures for moving objects.

Buffer management for indexes has been reported in the literature [3, 9]. ILRU (Inverse LRU) and OLRU (Optimal LRU) are studied in [9], and are shown to be better than the classic LRU. We shall briefly describe the OLRU strategy which we adapted in our work. In the OLRU strategy, the index pages are logically partitioned into L independent regions. Whenever an index page at level i is accessed, it is kept in region i of the buffer, or in the single free buffer in the case of *coalesced* regions. In general, the OLRU scheme allocates the available buffer according to reference frequency of nodes. Therefore this strategy can guarantee

that top-level pages of a tree have higher priority compared to those further from the root. As reported in [9], OLRU shows near optimal performance for both uniform and skew data.

According to the type of data being stored, the indexes [1, 6, 7, 10, 11, 12, 13] for moving objects can be classified into two categories: indexing the past positions of objects (i.e. trajectories) and indexing the current and anticipated future positions of objects. We focus on related works in the latter category because our methods belong to it. To index the current and near-future positions of moving objects, most existing approaches describe each object's location as a simple linear function, and update the database only when the predicted position deviates from the actual position larger than a threshold. One representative indexing is the time-parameterized R-tree (TPR-tree) [10]. The bounding rectangles in the TPR-tree are functions of time, as are the moving objects being indexed. Intuitively, the bounding rectangles are capable of continuously following the enclosed data points or other rectangles as these move. Most recently, the TPR*-tree [12] is proposed in order to optimize the TPR-tree. Next, Patel et al. [8] propose an indexing method, called STRIPES, which indexes predicted trajectories in a dual transformed space. The Q+R-tree makes use of the topography and the patterns of object movement and handles different types of moving objects separately [13]. In the Q+R tree, quasi-static objects are stored in an R*-tree and fast-moving objects are stored in a Quad-tree. Another recent index is the B^x-tree [4] which enables B⁺-tree to manage moving objects efficiently. The performance of the above index structures are all largely influenced by active objects which dominate the enlargements of MBRs or query windows.

3 The IMPACT Framework

This section presents the IMPACT (Integrated Memory Partitioning and Activity Conscious Twin-index) framework and the specific data structures and algorithms that we have employed to realize its implementation.

3.1 The Basic Framework

As observed in the introduction, a small portion of active moving objects have high speed and incur frequent updates. This prompted us to design the proposed IMPACT framework which comprises the following components:

1. A twin-index structure to dynamically manage the moving objects. One of the indexes is a main memory structure used to index active objects. Another index structure is a disk-based structure to index the inactive objects. Separating the active and inactive objects is expected to improve the overall system performance: (a) Active objects contribute to both high update cost and degrade query performance, and so keeping them away from the main bulk of the database can reduce the load on the database. (b) Managing active objects in main memory can further facilitate their processing, while

managing the inactive objects on disk also leads to less activities on the disk portion.

2. A buffering scheme for the disk-based structure to cache frequently and recently accessed paths or nodes. This is necessary as accesses to the disk-based structure would be costly if frequently accessed paths or nodes are not buffered.
3. A memory partitioning mechanism to optimally allocate the main memory to the memory-resident index and the buffer. Clearly, from the above point, allocating all the space to the main memory index is not likely to lead to good performance. Similarly, ignoring the main memory index reduces the problem to traditional buffering of a single disk-based index which, as we have argued, is not optimal either.

The IMPACT framework is generic. Any index structure can be used as the main memory index and/or disk-based index. In fact, the two structures can be different. Similarly, any existing index buffering strategies can also be employed. Moreover, we note that the memory partitioning mechanism is dependent on the structures used.

In this work, we employ a grid structure as the main memory index. Grid structure is preferred to be stored in the memory since (a) grid structure works well with hashing techniques to provide random and direct access; (b) each grid in memory only needs to store pointers to objects instead of storing full information of objects when residing in disk; (c) grid structure requires duplicating an object across all grids that it intersects, which speeds up query processing while results in poor update performance if it is disk-based.

For the disk-resident index, we employ the TPR*-tree. The TPR*-tree is built on the ideas of the TPR-tree, and has been shown to be a near-optimal structure for large moving object databases. To keep the commonly traversed index nodes in the cache, we adapted the OLRU scheme for our purpose. OLRU is chosen because it gives greater priority to the nodes of the tree nearer to the root. Furthermore, it is simpler and is reported to yield good performance.

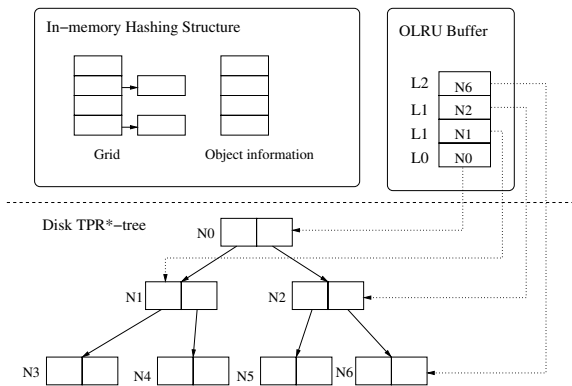


Fig. 1. The IMPACT structure

Figure 1 provides a pictorial representation of how the different data structures are related to one another. The figure contains two parts: the top portion shows that the available main memory space is split into two regions, one for the grid structure and the other for the OLRU; the bottom portion shows the disk-based TPR*-tree. Both the TPR*-tree and grid structure store the moving objects with different velocities, while the OLRU buffer manages frequent accessed nodes of the disk-based TPR*-tree. We note that as objects' activities change, they may be migrated from one structure to the next.

For the rest of this section, we shall first present the grid structure for active objects. Then we present the query and update algorithms on the twin-index structure. Finally, we will describe how we determine the memory allocation between the grid structure and the buffer for the TPR*-tree.

3.2 In-Memory Grid Structure

To manage the active moving objects in main memory, we employ a grid structure to capture these objects' current and future positions. Basically, we partition the two-dimensional domain space into a regular grid structure where each cell is a bucket. To support queries on future positions, we calculate the future trajectories of objects in a given time length and store the objects in the buckets their trajectories intersect. To define how far the trajectories need to be indexed, we use three time parameters as in [10], i.e. *query interval* (I), *index usage time* (U) and *time horizon* (H). Thus we can see that H represents how far into the future the index may reach and it is the upper limit of the index valid period. The grid structure must support queries that reach up to H time units into the future.

Each object may be hashed into several cells in the grid because of the existence of the trajectory. The entire grid is implemented as an array of cells, which efficiently supports random accesses to individual cells. To efficiently utilize the memory space, we do not store the detail information of moving objects in each corresponding bucket, but only keep the object identifier ID in the grid which can reduce memory consumption. A separate hash table with the key ID is used for the storage of objects, which supports fast random access to the detail information of moving objects. In case of a system crash, we backup these objects information to the disk periodically. The position of a moving object is represented by a reference position (x, y) , a corresponding velocity vector v , and reference time t . The overall grid structure is shown in Figure 2.

We present two algorithms to deal with the insert and query in the grid structure, i.e. *Insert_hash()* and *RangeQuery_hash()*. To insert a new object O , we first store the details of object into the hash table. Then we map the object into the cell and save its ID in the mapped cell. After that, we compute the trajectory of the object movement within time horizon H , and insert its ID into each intersected cells. When a range query is issued at time t , we hash the lower bound and upper bound points of the query rectangle to the cells in the grid, and retrieve the objects in the buckets overlapping with the query rectangle.

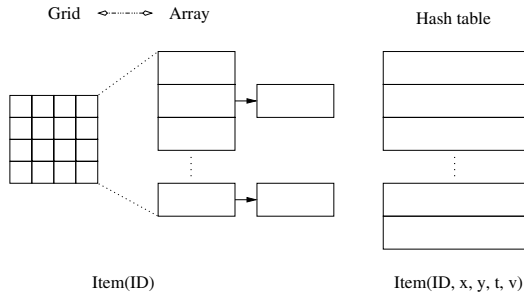


Fig. 2. The in-memory grid structure

3.3 Managing Moving Objects with the Twin-Index

We are now ready to look at how the twin-index is used to support moving objects. We shall address the primitive operations for construction, update and search. Delete operation is very simple: locate the object (either in memory or on disk), then delete the object accordingly, and so we omit the details here. In this discussion, we shall assume that certain amount of memory has been allocated to the grid structure. We defer how this amount is determined to the next subsection.

Building the Twin-Index. To build the twin-index, we first need to determine what constitutes an active object. We introduce a self-adaptive *velocity threshold* (V) for this purpose: those objects moving with velocities larger than the threshold will be stored in main memory, and those moving with lower speed are indexed on disk. Using speed as a splitting metric is effective because (a) no matter how frequently they change directions, fast objects may lead to huge expansion of MBRs; (b) slow moving objects which change directions frequently do not affect the TPR*-tree's performance but only introduce a few update numbers.

If we underestimate V , then the memory can be filled up. In this case, we will raise the threshold V . If we overestimate V , the memory will be under-utilized, and hence we have to decrease the value of V . Initially, we can estimate V as follows. We first get a sample from the dataset and store the statistic information of the speed with a histogram. The histogram partitions the speed domain into several sub-domains (called buckets) and counts the number of objects that fall into each bucket. According to the velocity distribution and available memory space, we get the initial V . We update the histogram for each object insertion, and V can be adjusted during the construction.

As shown in Figure 3, to construct the twin-index, the first step is to initiate the threshold V . As the V is self-adaptive, we can randomly select an initial value for it. However, to predict the V more precisely, we apply a sampling mechanism and calculate the initial value via the velocity histogram. For each object, we check whether the moving object is active. If the object is active and there is free space in memory, we insert the object into the in-memory

grid structure; otherwise the new object has to be inserted into the TPR*-tree on disk. During this process, we maintain a speed histogram in memory, and the velocity threshold can be adjusted periodically. Therefore, we can index the majority of active objects in memory.

Algorithm Construction()

Input: the moving object dataset

1. get a sample dataset;
2. initialize velocity histogram and V ;
3. for each object O in dataset
4. if ($O_v > V$)
5. if (Memory has free space)
6. invoke `Insert_hash(O)`;
7. else
8. insert O to the TPR*-tree;
9. update the velocity histogram;
10. adjust V if necessary;

Fig. 3. Construction algorithm

Range Query. The search operation consists of three steps: search the grid structure (which we have already described in the previous subsection), search the TPR*-tree (see [12]), and finally combine the answers. We expect the algorithm to be efficient because (a) the search in memory is much faster, (b) the search on the TPR*-tree is also faster (compared to a pure TPR*-tree that indexes all moving objects) since our structure only indexes inactive objects.

Moving Object Update. Whenever an update occurs, we need to determine whether to migrate the object that changes its activity status. Because the velocities of the moving objects may change from time to time, sometimes we need to switch the objects between the grid structure and the TPR*-tree to improve the efficiency. We deal with the objects in the main memory and disk using different policy since moving objects with high velocities tend to change the speed more frequently. To avoid oscillating between memory and disk too frequently, a moving object is allowed to reside in the memory even if its speed is less than the threshold. However, we tag these objects as inactive objects and store their IDs in an inactive object queue. Since we only store IDs in the queue, the space overhead is low. Whenever the speed of an object on disk exceeds the threshold and the main memory has no free space, these tagged objects will be replaced. In this way, we can fully utilize the memory space, and reduce the frequency of switch between memory and disk. The update algorithm is shown in Figure 4.

To update an object, we first locate the object to be updated. If the object is in memory, we update the object, and tag it as inactive (put its ID in the inactive object queue) if the new speed is less than the velocity threshold. If the object is on disk and still inactive, we just update the object in the TPR*-tree.

Algorithm Update(O)Input: O is the object to be updated

1. locate the object O ;
2. if (O is in memory)
3. update O in hash structure;
4. if ($O_v < \text{velocity threshold } V$)
5. tag O as inactive object;
6. insert O to the inactive object queue;
7. else
8. if ($O_v < V$)
9. update O in the TPR*-tree;
10. else
11. if (memory has free space)
12. delete O from the TPR*-tree;
13. invoke Insert_hash(O);
14. else
15. if ($\exists O'$ tagged as inactive)
16. switch(O, O');
17. else
18. update O in the TPR*-tree;
19. update the velocity histogram;
20. adjust V if necessary;

Fig. 4. Update algorithm

Otherwise, if there is free memory space, we move the object into memory. If there is no more memory space but there are inactive objects in memory, we replace the inactive objects with the new ones. In the last case, we have to update the object in the TPR*-tree even if the object becomes active. For each update, we adjust the counter of corresponding bucket in the velocity histogram and update the velocity threshold V if necessary.

Our algorithm is flexible even if the speed distribution is not constant. For example, the average speed of vehicles during the rush hour may be lower than that during the off-peak period. To reflect the variation of the speed distribution, the twin-index can automatically adjust the velocity threshold with respect to the objects' movements. This is done during the update operations with the aid of the speed histogram. According to the counter of V , when there are too many fast objects, we increase the value of V ; otherwise, we decrease it.

3.4 The Memory Partitioning Strategy

In this subsection, we present how the main memory can be allocated optimally to the main memory grid structure and the buffer for the TPR*-tree. We note that there is a relationship between the space allocated for buffering the TPR*-tree and that allocated for the main memory grid structure. Clearly, more space allocated to the buffer implies less space is left for the grid structure (i.e. fewer fast objects can be stored in memory). While this contributes to reduced I/O cost

to retrieve data from the TPR*-tree, it also means that the TPR*-tree is larger (since fewer objects can be retained in the main memory structure). Therefore, our goal is to solve this dilemma and achieve optimal memory utilization.

We analyze the performance of the twin-index and reveals the factors that determine the query cost, and then give the optimal memory allocation ratio between the grid structure and the OLRU buffer¹. Based on the cost analysis, we found that it is more beneficial to buffer the top two levels of the TPR*-tree but use the remainder memory space to index active objects for uniformly distributed data. Note that, the optimal memory allocation, i.e. buffering top two levels, may not be optimal for data with different distribution. However, this result suggests that a buffering strategy that gives priority to higher levels of index structure is preferred.

4 An Experimental Study

In this section, we report results of an experimental study to evaluate the IMPACT framework. As reference, we compare the IMPACT scheme against the TPR*-tree). For the TPR*-tree, all the available memory is used to buffer the index nodes/paths according to OLRU buffering scheme. We run 200 Range queries (4% of the space), and use the average I/O cost as the performance metrics. All the experiments are conducted on a 1.6G Hz P4 machine with 256M of main memory. Page size is set to 4KB and the default memory size is 8M bytes. Our evaluation comprises both uniform and skew datasets with 1M points. The default values of H (time horizon) and I (query interval) are 120 and 60 respectively.

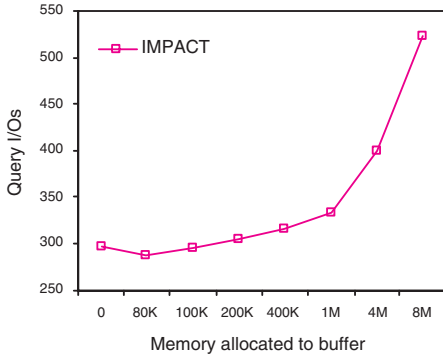
4.1 Performance on Uniform Dataset

In this set of experiments, we use randomly distributed uniform dataset as in [10], i.e., the objects are uniformly distributed in the space 1000×1000 , and also the velocities of moving objects are uniformly distributed between 0 and 3.

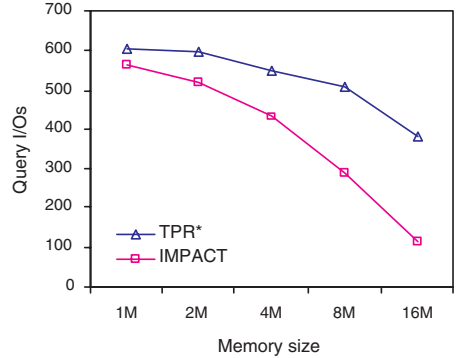
Effect of Memory Allocation. Given a fixed amount of memory space available for the processing of moving objects, we would like to study how the twin-index's performance is affected by the allocation of memory space between the grid structure and the OLRU buffer under varying amount of main memory. We fix the total memory to 8M. The results are shown in Figure 5 (a).

As expected, the results show that the twin-index yields better performance as more memory is allocated to the grid structure. When all the memory is used for the OLRU buffer, the twin-index behaves like the TPR*-tree. The active objects introduce heavy overlap in the disk index structure which degrades the query performance. As more memory is allocated to the grid structure, more active objects are kept in memory. This in turn significantly reduces the activities

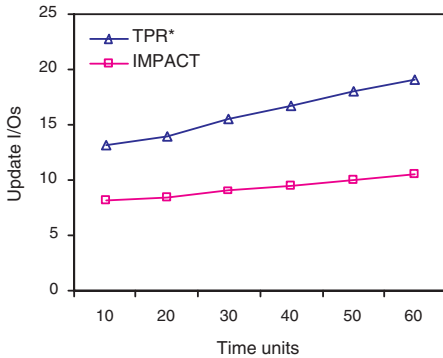
¹ For the details of cost analysis, interested readers are referred to [2].



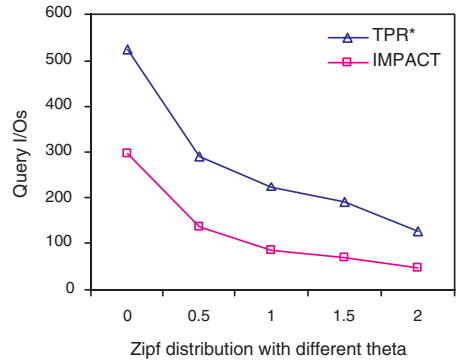
(a) Varying memory allocation



(b) Varying memory size



(c) Update



(d) Varying velocity distribution

Fig. 5. The experimental results

on the disk structure. Moreover, query processing on the grid structure is fast. We also observe that the difference in performance can be more than 40% between the two extreme memory allocations (either all for the OLRU buffer or all for the grid structure). The results suggest that even traditional buffering schemes that perform well (such as OLRU) may not cope well in moving object databases.

We also note that the best performance appears when we use around 80K memory space for OLRU buffer which is sufficient to keep the top two level nodes of the disk-based TPR*-tree in the buffer. From the view of buffer, buffering the top two levels is very effective. It does not cost much memory space, but is very helpful to improve the disk-based index performance, because most of these nodes have to be accessed by queries. Additionally, this OLRU buffer size is small compared to the overall memory available. The gain on the buffer surpasses the

loss on the disk-based TPR*-tree, because the TPR*-tree has to index a bit more objects whose space is occupied by the buffer. Therefore the optimal memory allocation is a compromise of these factors.

Effect of Memory Size. In this experiment, we compare the IMPACT against the TPR*-tree as we vary the total available memory size from 1M to 16M. We only show the performance of IMPACT with optimal memory allocation.

As shown in Figure 5 (b), both the schemes' performances improve with the increasingly larger memory size. When the available memory size is small ($< 1M$), the IMPACT is only marginally better than the TPR*-tree. This is because the small main memory allocated to the grid structure in the IMPACT does not hold enough active objects to result in significant workload reduction on the disk-based structure (as a large portion of the objects are still disk-based). However, as the memory size reaches beyond a certain point ($> 2M$), the IMPACT performs better than the TPR*-tree. The performance difference widens as the memory space increases. Indexing the active objects in main memory can significantly improve the efficiency of the disk portion (i.e. disk-based TPR*-tree) of the IMPACT. Additionally, the OLRU portion of IMPACT can still benefit from the buffering despite its small buffer size. The IMPACT can be 100% better than the TPR*-tree when the available memory space is larger than 8M. This study again confirms that traditional buffering techniques does not effectively utilize the main memory.

Effect of Update. In this experiment, we investigate the update cost of the two schemes. Figure 5 (c) compares the average update cost as a function of the number of updates. The performance of these methods decreases with increasing number of updates. Each update needs to conduct a query to retrieve the object to be updated, and this query cost increases with the number of updates because the updates will degenerate the efficiency of the index. However, the IMPACT outperforms the TPR*-tree and yields better scalability. It has two advantages over the TPR*-tree. First, since the IMPACT indexes the active objects in memory, most of the updates are restricted in memory and can be performed quickly. Second, the IMPACT can locate the inactive objects faster on disk, because the inactive objects introduce less overlap and slower MBR enlargement.

4.2 Performance on Skew Dataset

In the previous experiments, the objects and their velocities are uniformly distributed. However, most of the objects do not move at high speeds most of the time, i.e., a large portion of objects are in inactive state most of the time. We first test the effect of speed distribution on the query performance, shown in Figure 5 (d). The skew data is generated by varying the θ of the Zipf distribution, e.g. $\theta = 2$ means 80% of the objects fall in the 20% low velocity of the domain. The active objects are the main culprits for the heavy overlap in the index structure, and hence the range query has to search more nodes. When the data is skewed, all the indexes yield better performance, however the gain (in terms of percentage) between the IMPACT and the TPR*-tree is widened. As

the skew dataset has fewer active objects, it is possible to index a larger proportion of the active objects in the main memory. Thus, the IMPACT can benefit more from the velocity skewness.

5 Conclusion

In this paper, we have revisited the problem of moving object indexing. We propose a framework, called IMPACT, that integrates efficient memory allocation and a twin-index structure to manage moving objects. The IMPACT partitions the dataset into active objects and inactive objects and then processes the moving objects respectively. To efficiently support the query in memory, we apply a grid structure to index the active objects. To reduce the disk I/O, the remainder memory space is used as an OLRU buffer that allows frequently accessed nodes of the disk index to remain in main memory. We also conducted a series of experiments which shows convincingly that the proposed framework can lead to better performance.

References

1. H. D. Chon, D. Agrawal, and A. El Abbadi. Query processing for moving objects with space-time grid storage model. In *Proc. MDM*, pages 121–128, 2002.
2. B. Cui, D. Lin, and K.L. Tan. Towards optimal utilization of memory for moving object indexing. In *Technical Report, National University of Singapore*, 2004.
3. C. H. Goh, B. C. Ooi, D. Sim, and K. L. Tan. GHOST: Fine granularity buffering of index. In *Proc. VLDB*, pages 339–350, 1999.
4. C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-Tree based indexing of moving objects. In *Proc. VLDB*, 2004.
5. J. Kaufman, J. Myllymaki, and J. Jackson. *City Simulator spatial data generator*. <http://alphaworks.ibm.com/tech/citysimulator>, 2001.
6. D. Kwon, S. J. Lee, and S. H. Lee. Index the current positions of moving objects using the lazy update R-tree. In *MDM*, pages 113–120, 2002.
7. M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *Proc. VLDB*, pages 608–619, 2003.
8. J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *Proc. ACM SIGMOD*, pages 637–646, 2004.
9. G. M. Sacco. Index access with a finite buffer. In *Proc. VLDB*, pages 301–309, 1987.
10. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD*, pages 331–342, 2000.
11. Z. Song and N. Roussopoulos. Hashing moving objects. In *Proc. MDM*, pages 161–172, 2001.
12. Y. Tao, D. Papadias, and Jimeng Sun. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pages 790–801, 2003.
13. Y. Xia and S. Prabhakar. Q+Rtree: Efficient indexing for moving object databases. In *Proc. DASFAA*, pages 175–182, 2003.